Ncorr

C++ Port Instruction Manual

Version 1.0.0
6/20/2015

Justin Blaber (jblaber3@gatech.edu)

Table of Contents

1.1 - Installation Requirements

Compiler Requirements:

- Compiler must have C++11 support
- *Recommended compiler*: g++ 4.8+

NOTE: Ncorr has only been tested on g++ 4.8.

Build System:

- *Recommended*: CMake

NOTE: Ncorr includes CMakeLists.txt files for compilation using cmake (www.cmake.org/download).

Library Requirements:

- *Required:* OpenCV
- *Required:* FFTW
- *Required:* SuiteSparse
- *Required:* LAPACK
- *Required:* BLAS
- *Required:* gfortran (note this is included with g++)

NOTE: More information on these libraries is provided in section 1.2.

Operating System Requirements:

- *Recommended*: Linux

NOTE: Ncorr was developed on Ubuntu 12.04LTS. It currently will not work on the free version of Visual Studio on Windows because Visual Studio 2013 does not support some features of C++11.

1.2 – Dependent Libraries

Library: OpenCV

Version: 3.0.0

Link: http://opencv.org/downloads.html

Specific Libraries: libopencv_core, libopencv_highgui, libopencv_imgcodecs, libopencv_imgproc, and libopencv_videoio

Notes: Primarily used for `imread()`, `imshow()`, and the `VideoWriter` class

Library: FFTW

Version: 3.3.4

Link: http://www.fftw.org/download.html

Specific Libraries: fftw3

Notes: Used for convolution, deconvolution, and cross correlation

Library: SuiteSparse

Version: 4.4.4

Link: http://faculty.cse.tamu.edu/davis/suitesparse.html

Specific Libraries: libspqr, libcholmod, libsuitesparseconfig, libamd, libcolamd

Notes: Uses sparse QR solver for the digital inpainting of 2D data; uses the same algorithm as `inpaint_nans()` from Matlab's file exchange

Library: LAPACK, BLAS, and libgfortran

Link: http://www.netlib.org/lapack/ and http://www.netlib.org/blas/

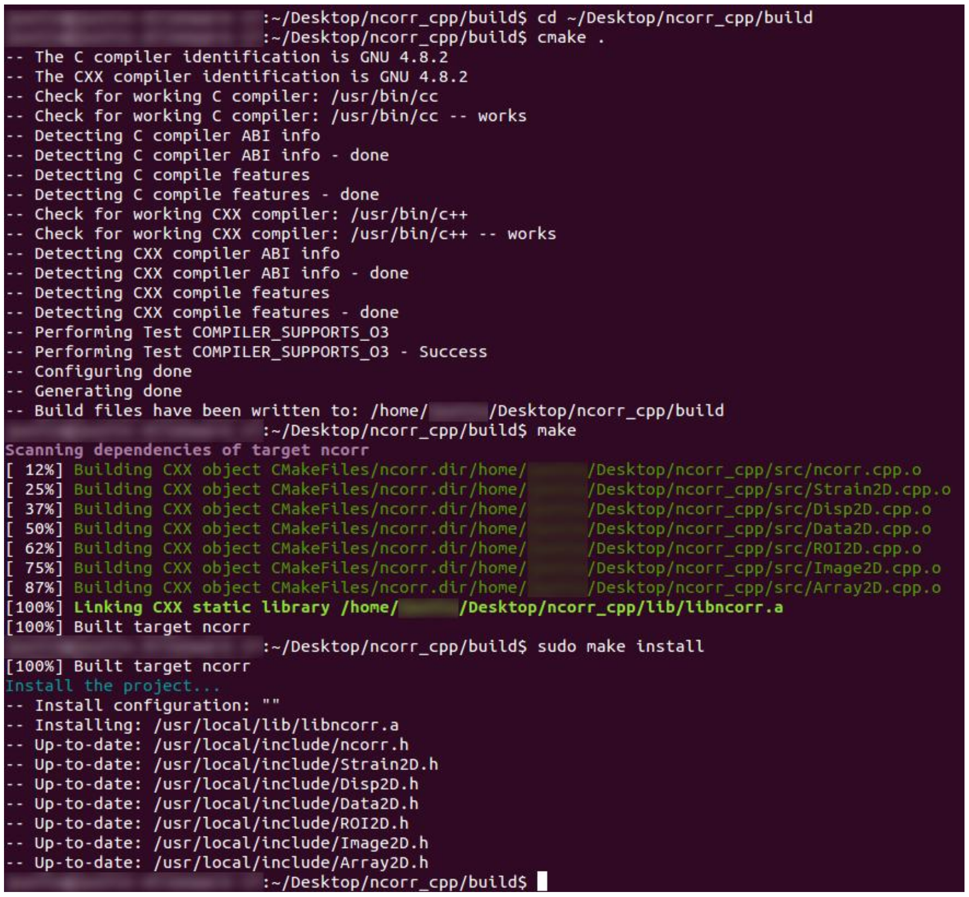
Notes: For g++ libgfortran must be linked when using BLAS.

1.3 – Ncorr Compilation

The compilation/installation of the Ncorr library is pretty easy using cmake. Start by navigating to the build directory. This guide assumes the files are downloaded and extracted to the desktop. Then call:

```
$ cmake .
$ make
$ sudo make install
```

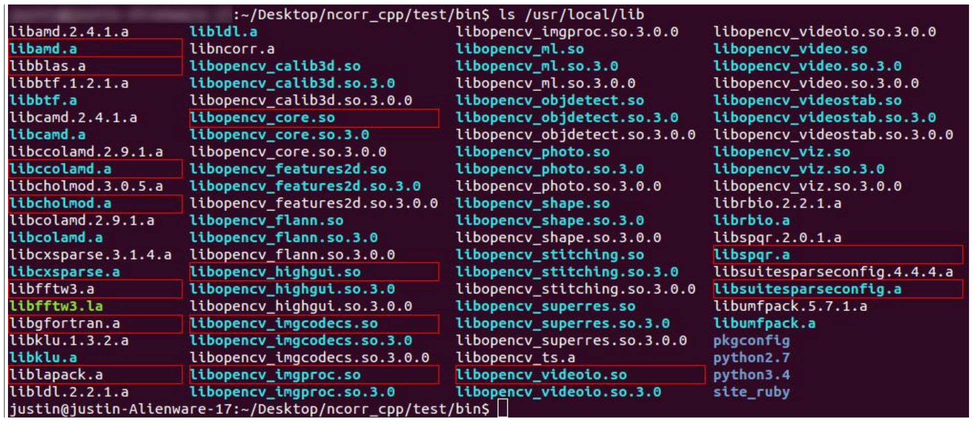
Make sure not to forget the "." after cmake. These steps are shown in greater detail below:



```
[redacted]:~/Desktop/ncorr_cpp/build$ cd ~/Desktop/ncorr_cpp/build
[redacted]:~/Desktop/ncorr_cpp/build$ cmake .
-- The C compiler identification is GNU 4.8.2
-- The CXX compiler identification is GNU 4.8.2
-- Check for working C compiler: /usr/bin/cc
-- Check for working C compiler: /usr/bin/cc -- works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Detecting C compile features
-- Detecting C compile features - done
-- Check for working CXX compiler: /usr/bin/c++
-- Check for working CXX compiler: /usr/bin/c++ -- works
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Detecting CXX compile features
-- Detecting CXX compile features - done
-- Performing Test COMPILER_SUPPORTS_O3
-- Performing Test COMPILER_SUPPORTS_O3 - Success
-- Configuring done
-- Generating done
-- Build files have been written to: /home/[redacted]/Desktop/ncorr_cpp/build
[redacted]:~/Desktop/ncorr_cpp/build$ make
Scanning dependencies of target ncorr
[ 12%] Building CXX object CMakeFiles/ncorr.dir/home/[redacted]/Desktop/ncorr_cpp/src/ncorr.cpp.o
[ 25%] Building CXX object CMakeFiles/ncorr.dir/home/[redacted]/Desktop/ncorr_cpp/src/Strain2D.cpp.o
[ 37%] Building CXX object CMakeFiles/ncorr.dir/home/[redacted]/Desktop/ncorr_cpp/src/Disp2D.cpp.o
[ 50%] Building CXX object CMakeFiles/ncorr.dir/home/[redacted]/Desktop/ncorr_cpp/src/Data2D.cpp.o
[ 62%] Building CXX object CMakeFiles/ncorr.dir/home/[redacted]/Desktop/ncorr_cpp/src/ROI2D.cpp.o
[ 75%] Building CXX object CMakeFiles/ncorr.dir/home/[redacted]/Desktop/ncorr_cpp/src/Image2D.cpp.o
[ 87%] Building CXX object CMakeFiles/ncorr.dir/home/[redacted]/Desktop/ncorr_cpp/src/Array2D.cpp.o
[100%] Linking CXX static library /home/[redacted]/Desktop/ncorr_cpp/lib/libncorr.a
[100%] Built target ncorr
[redacted]:~/Desktop/ncorr_cpp/build$ sudo make install
[100%] Built target ncorr
Install the project...
-- Install configuration: ""
-- Installing: /usr/local/lib/libncorr.a
-- Up-to-date: /usr/local/include/ncorr.h
-- Up-to-date: /usr/local/include/Strain2D.h
-- Up-to-date: /usr/local/include/Disp2D.h
-- Up-to-date: /usr/local/include/Data2D.h
-- Up-to-date: /usr/local/include/ROI2D.h
-- Up-to-date: /usr/local/include/Image2D.h
-- Up-to-date: /usr/local/include/Array2D.h
[redacted]:~/Desktop/ncorr_cpp/build$
```

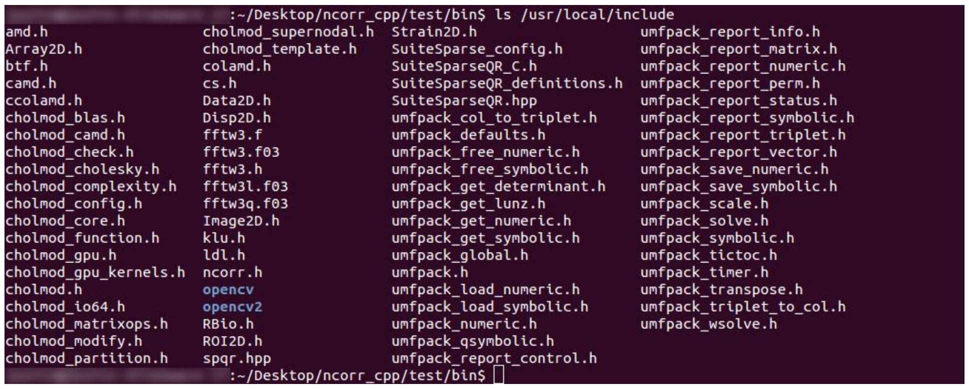
1.4 - Executable Compilation

Compilation of an executable using the Ncorr library is a little more complicated. The library dependencies (section 1.2) need to be included when the executable is compiled. Before compiling an executable, make sure that the dependent libraries are not only compiled but *installed*. This will allow g++ and cmake to find the required headers and libraries automatically. For Ubuntu, these libraries should be in the /usr/local/lib and /usr/local/include directories as shown below:



```
[redacted]:~/Desktop/ncorr_cpp/test/bin$ ls /usr/local/lib
libamd.2.4.1.a      libldl.a                       libopencv_imgproc.so.3.0.0    libopencv_videoio.so.3.0.0
libamd.a            libncorr.a                     libopencv_ml.so               libopencv_video.so
libblas.a           libopencv_calib3d.so           libopencv_ml.so.3.0           libopencv_video.so.3.0
libbtf.1.2.1.a      libopencv_calib3d.so.3.0       libopencv_ml.so.3.0.0         libopencv_video.so.3.0.0
libbtf.a            libopencv_calib3d.so.3.0.0     libopencv_objdetect.so        libopencv_videostab.so
libcamd.2.4.1.a     libopencv_core.so              libopencv_objdetect.so.3.0    libopencv_videostab.so.3.0
libcamd.a           libopencv_core.so.3.0          libopencv_objdetect.so.3.0.0  libopencv_videostab.so.3.0.0
libccolamd.2.9.1.a  libopencv_core.so.3.0.0        libopencv_photo.so            libopencv_viz.so
libccolamd.a        libopencv_features2d.so        libopencv_photo.so.3.0        libopencv_viz.so.3.0
libcholmod.3.0.5.a  libopencv_features2d.so.3.0    libopencv_photo.so.3.0.0      libopencv_viz.so.3.0.0
libcholmod.a        libopencv_features2d.so.3.0.0  libopencv_shape.so            librbio.2.2.1.a
libcolamd.2.9.1.a   libopencv_flann.so             libopencv_shape.so.3.0        librbio.a
libcolamd.a         libopencv_flann.so.3.0         libopencv_shape.so.3.0.0      libspqr.2.0.1.a
libcxsparse.3.1.4.a libopencv_flann.so.3.0.0       libopencv_stitching.so        libspqr.a
libcxsparse.a       libopencv_highgui.so           libopencv_stitching.so.3.0    libsuitesparseconfig.4.4.4.a
libfftw3.a          libopencv_highgui.so.3.0       libopencv_stitching.so.3.0.0  libsuitesparseconfig.a
libfftw3.la         libopencv_highgui.so.3.0.0     libopencv_superres.so         libumfpack.5.7.1.a
libgfortran.a       libopencv_imgcodecs.so         libopencv_superres.so.3.0     libumfpack.a
libklu.1.3.2.a      libopencv_imgcodecs.so.3.0     libopencv_superres.so.3.0.0   pkgconfig
libklu.a            libopencv_imgcodecs.so.3.0.0   libopencv_ts.a                python2.7
liblapack.a         libopencv_imgproc.so           libopencv_videoio.so          python3.4
libldl.2.2.1.a      libopencv_imgproc.so.3.0       libopencv_videoio.so.3.0      site_ruby
justin@justin-Alienware-17:~/Desktop/ncorr_cpp/test/bin$ 
```

Note that libgfortran.a is typically in gcc's installation directory. Locate the file on your system, and then copy it to the /usr/local/lib directory so that cmake can find this library automatically. You can also write your own Makefile and just link the libraries manually if you want as well.

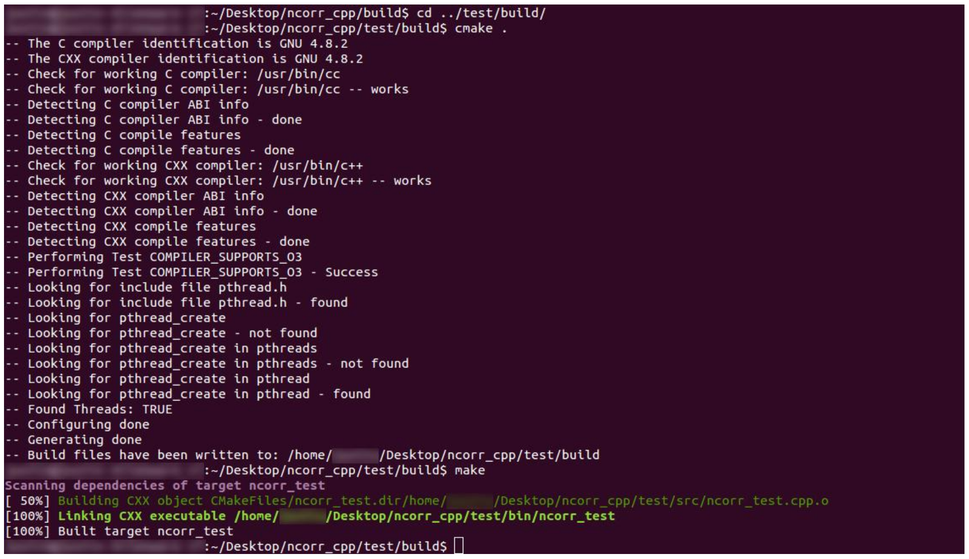


```
[redacted]:~/Desktop/ncorr_cpp/test/bin$ ls /usr/local/include
amd.h                   cholmod_supernodal.h  Strain2D.h                     umfpack_report_info.h
Array2D.h               cholmod_template.h    SuiteSparse_config.h           umfpack_report_matrix.h
btf.h                   colamd.h              SuiteSparseQR_C.h              umfpack_report_numeric.h
camd.h                  cs.h                  SuiteSparseQR_definitions.h    umfpack_report_perm.h
ccolamd.h               Data2D.h              SuiteSparseQR.hpp              umfpack_report_status.h
cholmod_blas.h          Disp2D.h              umfpack_col_to_triplet.h       umfpack_report_symbolic.h
cholmod_camd.h          fftw3.f               umfpack_defaults.h             umfpack_report_triplet.h
cholmod_check.h         fftw3.f03             umfpack_free_numeric.h         umfpack_report_vector.h
cholmod_cholesky.h      fftw3.h               umfpack_free_symbolic.h        umfpack_save_numeric.h
cholmod_complexity.h    fftw3l.f03            umfpack_get_determinant.h      umfpack_save_symbolic.h
cholmod_config.h        fftw3q.f03            umfpack_get_lunz.h             umfpack_scale.h
cholmod_core.h          Image2D.h             umfpack_get_numeric.h          umfpack_solve.h
cholmod_function.h      klu.h                 umfpack_get_symbolic.h         umfpack_symbolic.h
cholmod_gpu.h           ldl.h                 umfpack_global.h               umfpack_tictoc.h
cholmod_gpu_kernels.h   ncorr.h               umfpack.h                      umfpack_timer.h
cholmod.h               opencv                umfpack_load_numeric.h         umfpack_transpose.h
cholmod_io64.h          opencv2               umfpack_load_symbolic.h        umfpack_triplet_to_col.h
cholmod_matrixops.h     RBio.h                umfpack_numeric.h              umfpack_wsolve.h
cholmod_modify.h        ROI2D.h               umfpack_qsymbolic.h
cholmod_partition.h     spqr.hpp              umfpack_report_control.h
[redacted]:~/Desktop/ncorr_cpp/test/bin$ 
```

After you make sure all the proper libraries are installed and all the necessary header files are in /usr/local/include, you can compile the example in the /test directory by calling:

```
$ cmake .
$ make
```

The output in the terminal is shown below:

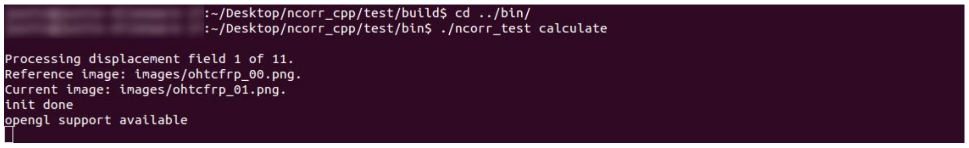


```
[redacted]:~/Desktop/ncorr_cpp/build$ cd ../test/build/
[redacted]:~/Desktop/ncorr_cpp/test/build$ cmake .
-- The C compiler identification is GNU 4.8.2
-- The CXX compiler identification is GNU 4.8.2
-- Check for working C compiler: /usr/bin/cc
-- Check for working C compiler: /usr/bin/cc -- works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Detecting C compile features
-- Detecting C compile features - done
-- Check for working CXX compiler: /usr/bin/c++
-- Check for working CXX compiler: /usr/bin/c++ -- works
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Detecting CXX compile features
-- Detecting CXX compile features - done
-- Performing Test COMPILER_SUPPORTS_O3
-- Performing Test COMPILER_SUPPORTS_O3 - Success
-- Looking for include file pthread.h
-- Looking for include file pthread.h - found
-- Looking for pthread_create
-- Looking for pthread_create - not found
-- Looking for pthread_create in pthreads
-- Looking for pthread_create in pthreads - not found
-- Looking for pthread_create in pthread
-- Looking for pthread_create in pthread - found
-- Found Threads: TRUE
-- Configuring done
-- Generating done
-- Build files have been written to: /home/[redacted]/Desktop/ncorr_cpp/test/build
[redacted]:~/Desktop/ncorr_cpp/test/build$ make
Scanning dependencies of target ncorr_test
[ 50%] Building CXX object CMakeFiles/ncorr_test.dir/home/[redacted]/Desktop/ncorr_cpp/test/src/ncorr_test.cpp.o
[100%] Linking CXX executable /home/[redacted]/Desktop/ncorr_cpp/test/bin/ncorr_test
[100%] Built target ncorr_test
[redacted]:~/Desktop/ncorr_cpp/test/build$
```

If all goes well, then the executable should be compiled and located in the bin folder. Navigate to the bin directory and run it by calling:

```
$ ./ncorr_test calculate
```

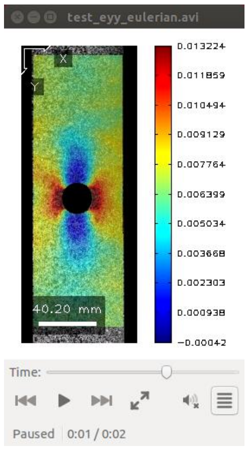
The output in the terminal is shown below:



```
[redacted]:~/Desktop/ncorr_cpp/test/build$ cd ../bin/
[redacted]:~/Desktop/ncorr_cpp/test/bin$ ./ncorr_test calculate

Processing displacement field 1 of 11.
Reference image: images/ohtcfrp_00.png.
Current image: images/ohtcfrp_01.png.
init done
opengl support available
```

This test executable requires a command line input of either “calculate” or “load”. “calculate” will calculate the displacement and strain fields, save the DIC/strain information as binary, and then save videos of the displacement and strains. After this information is calculated and saved, `ncorr_test` can be called again with “load” to save the videos using the saved data directly, instead of recalculating the plots. If the executable runs correctly, the E_{yy} Eulerian-Almansi strain video should look like:

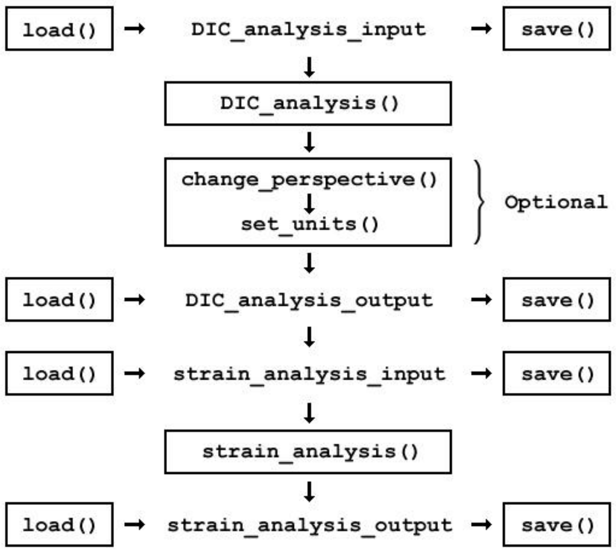


2.1 – Program Flow

There are four main data structures used in Ncorr. They are:

```
1. DIC_analysis_input
2. DIC_analysis_output
3. strain_analysis_input
4. strain_analysis_output
```

These data structures should not be altered directly. The `*_input`s are either formed through their constructors or loaded from saved data. The `*_output`s are either formed through interface functions or loaded from saved data. The overall program flow is shown in the figure below:



The first step for performing DIC is to form a `DIC_analysis_input`:

```
DIC_analysis_input(const std::vector<Image2D> &imgs,
                   const ROI2D &roi,
                   difference_type scalefactor,
                   INTERP interp_type,
                   SUBREGION subregion_type,
                   difference_type r,
                   difference_type num_threads,
                   DIC_analysis_config DIC_config,
                   bool debug);
```

The inputs to this constructor are all the basic things needed to perform digital image correlation. The `imgs` input is a vector of `Image2D`s that essentially hold the paths to image files so they can be loaded on demand. The first image is considered the reference image, and displacements are calculated with respect to this reference image. The `roi` input is a region of interest that's formed with respect to the reference image. `scalefactor` determines how much the output `Disp2D`s are scaled with respect to the reference image and is used to reduce computation time. A `scalefactor` of 1 means the output `Disp2D`s are the same size as the reference image, a `scalefactor` of 2 means the output `Disp2D`s are reduced by a factor of 2 in size, etc. The `interp_type` input specifies the type of interpolation used in the analysis.

Currently cubic spline and biquintic b-spline interpolation are available for use. The `subregion_type` specifies the shape of the subset that is used in the analysis. Currently, a circle or square `subregion_type` can be used. `r` represents the radius of the subset. `num_threads` specifies the number of threads used in the analysis. `DIC_config` specifies three preset parameters (based on the correlation coefficient) which are used as criterion for filtering out data and for how the reference image update scheme works in the case that large deformation is anticipated. These parameters can also be entered manually using a different `DIC_analysis` constructor; this is discussed in more detail in the next paragraph. Lastly, `debug` specifies whether debugging tools are enabled (debugging will show images of the processing updating, like a waitbar).

Specifying `DIC_analysis_config` set three parameters: `cutoff_corrcoef`, `update_corrcoef`, and `prctile_corrcoef`. I've tried to encapsulate all the correlation coefficient based heuristic parameters used in `DIC_analysis` by specifying this input, and provided some preset values for these parameters to help perform analyses for typical data sets. Any data points with a correlation coefficient over `cutoff_corrcoef` will be removed from the data plot. `update_corrcoef` and `prctile_corrcoef` specify when the reference image is updated. After the data plot is analyzed, `prctile_corrcoef` is used as in input to the `prctile` function on the corresponding correlation coefficient plot to select a specified value (i.e if `prctile_corrcoef` is 1.0, it will select the max value; If `prctile_corrcoef` is 0.5, it will select the median value, etc). If this specified value is over `cutoff_corrcoef`, it will trigger an update. The preset `DIC_analysis_config`s are displayed in the table below:

`DIC_analysis_config::NO_UPDATE`	`cutoff_corrcoef = 2.0;` `update_corrcoef = 4.0;` `prctile_corrcoef = 1.0;`	Used when you specifically do not want the reference image to update. This is used for datasets where the strain is known to be low, and is the preferred mode for the "plate hole" sample from SEM's DIC challenge
`DIC_analysis_config::KEEP_MOST_POINTS`	`cutoff_corrcoef = 2.0;` `update_corrcoef = 0.5;` `prctile_corrcoef = 1.0;`	Used for data where high deformation is expected, but the deformation is *not* discontinuous (i.e. no cracks form), and thus this mode attempts to keep all the data points and update as frequently as needed to prevent badly analyzed points
`DIC_analysis_config::REMOVE_BAD_POINTS`	`cutoff_corrcoef = 0.7;` `update_corrcoef = 0.35;` `prctile_corrcoef = 0.9;`	Used for data where some cracks form. This mode will attempt to remove the badly analyzed points from the data plots. Since points near discontinuous displacements have very high correlation coefficients, this mode prevents updating too frequently when only a small number of points have a very high correlation coefficient. This is the preferred mode for "weld" sample for SEM's DIC challenge.

If the user wants to specify these parameters manually, then they can also call the following overload for the `DIC_analysis_input`:

```
DIC_analysis_input(const std::vector<Image2D> &imgs,
                   const ROI2D &roi,
                   difference_type scalefactor,
                   INTERP interp_type,
                   SUBREGION subregion_type,
                   difference_type r,
                   difference_type num_threads,
                   double cutoff_corrcoef,
                   double update_corrcoef,
                   double prctile_corrcoef,
                   bool debug);
```

Once the `DIC_analysis_input` formed, it is used as an input to `DIC_analysis`:

```
DIC_analysis_output DIC_analysis(const DIC_analysis_input &DIC_input);
```

`DIC_analysis_output` contains information for the displacement fields as well as their perspective (Lagrangian by default, but can be converted to Eulerian) and units (pixels by default). After the analysis is complete, the `DIC_analysis_output` can be modified to change its perspective and/or set units for the displacement fields. Changing perspective can be done by using:

```
DIC_analysis_output change_perspective(const DIC_analysis_output &DIC_output,
                                       INTERP interp_type);
```

Note that changing perspective can only be done from the Langrangian to Eulerian perspective, and must be done before setting the units. Units can be set by using:

```
DIC_analysis_output set_units(const DIC_analysis_output &DIC_output,
                              const std::string &units,
                              double units_per_pixel);
```

The `units_per_pixel` parameter must be measured and is dependent on the user's experimental set up. Lastly, note that setting the units will not change the strain values (as expected), so if only strains are desired then setting the units for displacements can be skipped. After the `DIC_analysis_output` has been formatted, strains can be calculated by first forming a `strain_analysis_input`:

```
strain_analysis_input(const DIC_analysis_input &DIC_input,
                      const DIC_analysis_output &DIC_output,
                      SUBREGION subregion_type,
                      difference_type r);
```

`subregion_type` specifies the shape of the subset that is used for the least squares plane fit that will calculate displacement gradients and `r` specifies the radius used. Once formed, `strain_analysis` is called in order to calculate the strain fields:

```
strain_analysis_output strain_analysis(const strain_analysis_input &strain_input);
```

The `strain_analysis_output` will contain Green-Lagrangian strains if the `strain_analysis_input` contains a Lagrangian `DIC_analysis_output`; it will contain Eulerian-Almansi strains if the `strain_analysis_input` contains an Eulerian `DIC_analysis_output`.

After the analyses are completed, the `*_input`s and `*_output`s can be saved as binary by calling the `save` interface functions:

```
        friend void save(const DIC_analysis_input&, const std::string&);
        friend void save(const DIC_analysis_output&, const std::string&);
        friend void save(const strain_analysis_input&, const std::string&);
        friend void save(const strain_analysis_output&, const std::string&);
```

Once saved, the `*_input`s and `*_output`s can be loaded by calling the static factory `load` method:

```
        static DIC_analysis_input load(const std::string&);
        static DIC_analysis_output load(const std::string&);
        static strain_analysis_input load(const std::string&);
        static strain_analysis_output load(const std::string&);
```

An example of the overall program flow (copied and modified slightly from the ncorr_test.cpp file in the test directory) is given below:

```
// Initialize DIC and strain information ---------------//
DIC_analysis_input DIC_input;
DIC_analysis_output DIC_output;
strain_analysis_input strain_input;
strain_analysis_output strain_output;

// Determine whether to calculate or load data
bool calculate = true;
if (!calculate) {
    // Load inputs
    DIC_input = DIC_analysis_input::load("save/DIC_input.bin");
    DIC_output = DIC_analysis_output::load("save/DIC_output.bin");
    strain_input = strain_analysis_input::load("save/strain_input.bin");
    strain_output = strain_analysis_output::load("save/strain_output.bin");
} else {
    // Set images
    std::vector<Image2D> imgs;
    for (int i = 0; i <= 11; ++i) {
        std::ostringstream ostr;
        ostr << "images/ohtcfrp_" << std::setfill('0') << std::setw(2) << i << ".png";
        imgs.push_back(ostr.str());
    }
    Image2D roi_img("images/roi.png");

    // Set DIC_input
    DIC_input = DIC_analysis_input(imgs,                                    // Images
                                   ROI2D(roi_img.get_gs() > 0.5),           // ROI
                                   3,                                       // scalefactor
                                   INTERP::QUINTIC_BSPLINE_PRECOMPUTE,      // interpolation
                                   SUBREGION::CIRCLE,                       // Subregion shape
                                   20,                                      // Subregion radius
                                   4,                                       // # of threads
                                   DIC_analysis_config::NO_UPDATE,          // DIC configuration
                                   true);                                   // Debugging

    // Perform DIC_analysis
    DIC_output = DIC_analysis(DIC_input);

    // Convert DIC_output to Eulerian perspective
    DIC_output = change_perspective(DIC_output, INTERP::QUINTIC_BSPLINE_PRECOMPUTE);

    // Set units of DIC_output (assume 0.2 mm per pixel)
    DIC_output = set_units(DIC_output, "mm", 0.2);

    // Set strain input
    strain_input = strain_analysis_input(DIC_input,
                                         DIC_output,
                                         SUBREGION::CIRCLE,                 // Subregion shape
                                         5);                                // Subregion radius
    // Perform strain_analysis
    strain_output = strain_analysis(strain_input);

    // Save outputs as binary
    save(DIC_input, "save/DIC_input.bin");
    save(DIC_output, "save/DIC_output.bin");
    save(strain_input, "save/strain_input.bin");
    save(strain_output, "save/strain_output.bin");
}
```

2.2 – Saving Videos and Images

Once the `*_input`s and `*_output`s have been formed, the user can also save video and images using some built in interface functions.

To save a displacement video, you call `save_DIC_video`:

```
void save_DIC_video(const std::string &filename,
                    const DIC_analysis_input &DIC_input,
                    const DIC_analysis_output &DIC_output,
                    DISP disp_type,
                    double alpha,
                    double fps,
                    double min_disp = std::numeric_limits<double>::quiet_NaN(),
                    double max_disp = std::numeric_limits<double>::quiet_NaN(),
                    bool enable_colorbar = true,
                    bool enable_axes = true,
                    bool enable_scalebar = true,
                    double num_units = -1.0,
                    double font_size = 1.0,
                    ROI2D::difference_type num_tick_marks = 11,
                    int colormap = cv::COLORMAP_JET,
                    double end_delay = 2.0,
                    int fourcc = cv::VideoWriter::fourcc('M','J','P','G'));
```

For strain, you call `save_strain_video`:

```
void save_strain_video(const std::string &filename,
                       const strain_analysis_input &strain_input,
                       const strain_analysis_output &strain_output,
                       STRAIN strain_type,
                       double alpha,
                       double fps,
                       double min_strain = std::numeric_limits<double>::quiet_NaN(),
                       double max_strain = std::numeric_limits<double>::quiet_NaN(),
                       bool enable_colorbar = true,
                       bool enable_axes = true,
                       bool enable_scalebar = true,
                       double num_units = -1.0,
                       double font_size = 1.0,
                       ROI2D::difference_type num_tick_marks = 11,
                       int colormap = cv::COLORMAP_JET,
                       double end_delay = 2.0,
                       int fourcc = cv::VideoWriter::fourcc('M','J','P','G'));
```

Saving an image requires directly obtaining the `Data2D` of interest and calling `save_ncorr_data_over_img`:

```
void save_ncorr_data_over_img(const std::string &filename,
                              const Image2D &img,
                              const Data2D &data,
                              double alpha,
                              double min_data,
                              double max_data,
                              bool enable_colorbar,
                              bool enable_axes,
                              bool enable_scalebar,
                              const std::string &units,
                              double units_per_pixel,
                              double num_units,
                              double font_size,
                              ROI2D::difference_type num_tick_marks,
                              int colormap);
```

An example of how to save videos and images (copied and modified slightly from the ncorr_test.cpp file in the test directory) is given below:

```
// Save videos - use default values
save_DIC_video("video/test_v_eulerian.avi",
               DIC_input,
               DIC_output,
               DISP::V,
               0.5,           // Alpha
               15);           // FPS

save_DIC_video("video/test_u_eulerian.avi",
               DIC_input,
               DIC_output,
               DISP::U,
               0.5,           // Alpha
               15);           // FPS

save_strain_video("video/test_eyy_eulerian.avi",
                  strain_input,
                  strain_output,
                  STRAIN::EYY,
                  0.5,        // Alpha
                  15);        // FPS

save_strain_video("video/test_exy_eulerian.avi",
                  strain_input,
                  strain_output,
                  STRAIN::EXY,
                  0.5,        // Alpha
                  15);        // FPS

save_strain_video("video/test_exx_eulerian.avi",
                  strain_input,
                  strain_output,
                  STRAIN::EXX,
                  0.5,        // Alpha
                  15);        // FPS

// Save images - must specify all parameters
save_ncorr_data_over_img("images/test_eyy_last_eulerian.jpg",
                         strain_input.DIC_input.imgs.back(),
                         strain_output.strains.back().get_eyy(),
                         0.5,
                         0.0,
                         0.015,
                         true,
                         true,
                         true,
                         strain_input.DIC_output.units,
                         strain_input.DIC_output.units_per_pixel,
                         50,
                         1.0,
                         11,
                         cv::COLORMAP_JET);

save_ncorr_data_over_img("images/test_exy_last_eulerian.jpg",
                         strain_input.DIC_input.imgs.back(),
                         strain_output.strains.back().get_exy(),
                         0.5,
                         -0.0075,
                         0.0075,
                         true,
                         true,
                         true,
                         strain_input.DIC_output.units,
                         strain_input.DIC_output.units_per_pixel,
                         50,
                         1.0,
                         11,
                         cv::COLORMAP_JET);

save_ncorr_data_over_img("images/test_exx_last_eulerian.jpg",
                         strain_input.DIC_input.imgs.back(),
                         strain_output.strains.back().get_exx(),
                         0.5,
                         -0.015,
                         0.0,
                         true,
                         true,
                         true,
                         strain_input.DIC_output.units,
                         strain_input.DIC_output.units_per_pixel,
                         50,
                         1.0,
                         11,
                         cv::COLORMAP_JET);
```

2.3 – Accessing Data Directly

In order to access the data directly, you have to get the `Data2D` you'd like to access (using either the `.disps` field of the `DIC_analysis_output` structure, or the `.strains` field of the `strain_analysis_output` structure). These contain the `Disp2D` and `Strain2D` classes, respectively. `Disp2D` contains two `Data2D`'s paired together which can be retrieved by calling the `get_v()` and `get_u()` methods. `Strain2D` contains three `Data2D`'s paired together which can be retrieved by calling the `get_eyy()`, `get_exy()`, and `get_exx()` methods.

2D data in Ncorr is stored as a 2D array (in a class called `Array2D<double>`) with a corresponding 2D region of interest (in a class called `ROI2D`). These two classes are paired together in a class called `Data2D`. In order to retrieve the `Array2D<double>`, you call the `get_array()` method, and in order to retrieve the `ROI2D`, you call the `get_roi()` method. So if you want to cycle over the `Data2D` to access data values directly and perform additional analyses, the easiest way is to loop over the entire 2D array and only access data if it is in the region of interest.

The above description can be a little convoluted at first, so the best way to describe how to access the data directory is to just provide an example which is given below:

```
// Get the displacement field you want to access
Disp2D disp = DIC_output.disps.back();

// Get the corresponding v and u Array2Ds
const Array2D<double> &v_array = disp.get_v().get_array();
const Array2D<double> &u_array = disp.get_u().get_array();

// Get the corresponding ROI2D
ROI2D disp_roi = disp.get_roi();

// Cycle over Disp2D and print out values
for (int p2 = 0; p2 < disp.data_width(); ++p2) {
    for (int p1 = 0; p1 < disp.data_height(); ++p1) {
        if (disp_roi(p1,p2)) {
            std::cout << "v(" << p1 << "," << p2 << ") = " << v_array(p1,p2) << std::endl;
            std::cout << "u(" << p1 << "," << p2 << ") = " << u_array(p1,p2) << std::endl;
        }
    }
}

// Get the strain field you want to access
Strain2D strain = strain_output.strains.back();

// Get the corresponding eyy, exy, and exx Array2Ds
const Array2D<double> &eyy_array = strain.get_eyy().get_array();
const Array2D<double> &exy_array = strain.get_exy().get_array();
const Array2D<double> &exx_array = strain.get_exx().get_array();

// Get the corresponding ROI2D
ROI2D strain_roi = strain.get_roi();

// Cycle over Strain2D and print out values
for (int p2 = 0; p2 < strain.data_width(); ++p2) {
    for (int p1 = 0; p1 < strain.data_height(); ++p1) {
        if (strain_roi(p1,p2)) {
            std::cout << "eyy(" << p1 << "," << p2 << ") = " << eyy_array(p1,p2) << std::endl;
            std::cout << "exy(" << p1 << "," << p2 << ") = " << exy_array(p1,p2) << std::endl;
            std::cout << "exx(" << p1 << "," << p2 << ") = " << exx_array(p1,p2) << std::endl;
        }
        }
}
```