CPPTRAJ Development Notes

Daniel R. Roe (daniel.r.roe@gmail.com)
Jason M. Swails (Code Docs)

2010-07-21
Last Updated: 2015-08-14

Abstract

CPPTRAJ is code used for processing MD trajectory data as well as other types of data, derived from trajectories or otherwise. CPPTRAJ is a complete rewrite of the PTRAJ code in primarily C++, with the intent being to make the code more readable, leak-free, and thread-safe. The biggest functional change from PTRAJ is the ability to load and process trajectories with different topology files in the same run.

This guide assumes that the reader has at least a basic familiarity with C and C++ object-oriented programming. If you aren't sure what a constructor is or how pointers work you may have a difficult time coding in Cpptraj. There are several good introduction to C/C++ tutorials on the web that may be helpful.

Contents

Part I

Introduction

1 Coding Conventions

It is important to maintain a consistent coding style within cpptraj so that it remains easy to modify and understand. By following code conventions, it will be easier to read code written by anybody and determine what is happening.

- Code blocks are indented using 2 spaces. **DO NOT USE TABS** since these are in general not portable between different editors.
- Try to keep lines between 80 and 100 characters long.
- C++ files have '.cpp' suffix, C files have '.c' suffix, header files have '.h' suffix.
- All header files should have a '#define' guard to prevent multiple inclusion. The define guard has format:

  ```
  #ifndef INC_<basefilename>_H
  #define INC_<basefilename>_H
  ...
  #endif
  ```

- 'using namespace' should be used sparingly and NEVER in a header file.
- The order of #include directives should be (in general): C includes, C++ includes, class definition, any other Cpptraj includes.
- Use of STL classes/methods is acceptable; use C99 conventions to maximize portability. The only external libraries that should be used are NetCDF and ARPACK/LAPACK/BLAS (both included with AmberTools), i.e. no Boost etc.
- Do not use iostream for basic IO. All console output should be performed with the functions in CpptrajStdio.h (chiefly mprintf() and mprinterr() for STDOUT and STDERR respectively). All file IO should be performed with CpptrajFile or the derived classes BufferedLine and BufferedFrame. This choice has been made mainly for performance reasons (C file routines are in general much faster than iostream), but also so that all IO is centralized (e.g. CpptrajFile will automatically detect if an input file is compressed). This is also so output can be easily controlled; for example, using mprintf will make sure that during MPI only the master writes.
- Warnings should be written to STDOUT with mprintf with prefix 'Warning:'; errors should be written to STDERR with mprinterr with prefix 'Error:'.

- Classes:
  - Class types are named using *CapWords* (no spaces or underscores, start of each word is a capital letter).
  - Files containing a class should be named after the class (e.g. 'class TrajectoryFile {};' in TrajectoryFile.cpp).
  - Classes which inherit should be named after their base class (e.g. 'class Action_Distance : public Action { };').
  - Public class methods should be listed first; protected methods/variables second; private methods/variables last. All class member variables should be private if possible.
  - Public class methods are named using *CapWords*.
  - Private class methods are named using *mixedCase.*
  - Class variables that are `private` or `protected` are named using *mixedCase_* (with a trailing underscore).
- Abbreviations: 1st letter in each word is capitalized. For instance, `Data File List` may be abbreviated `DatFilList` or `DFL.`
- Variables that have function scope (or lower) and all public variables for classes are named using *mixedCase* (same as *CapWords* except the first letter is lower-case).
- No one-letter variable names except in loop scopes (e.g. for (int i = 0; i < N; ++i) { }), and even then they should be short loops (no more than 10 lines or so).
- All identifiers in an enumerated type are named using all *CAPS*, and the first identifier should be explicitly initialized (e.g. enum DirectionType { DX = 0, DY, DZ };).
- There is a *doxygen* rule file to automatically generate code documentation using *doxygen*, so please construct comments in such a doxygen-compatible manner (e.g. JavaDoc etc). See http://www.stack.nl/~dimitri/doxygen/manual.html for instructions.

2 Building Cpptraj and Documentation

Cpptraj is automatically built as part of AmberTools, or it can be built standalone using the configure script in the `$AMBERHOME/AmberTools/src/cpptraj` or `$CPPTRAJHOME` directory. The standalone build is particularly useful for development and testing. Type './configure –help' for a list of configure options. In order to build Cpptraj standalone one needs to specify the location of the NetCDF, zlib, bzlib2, and BLAS/LAPACK/ARPACK libraries if they aren't in your system path; configure will use the ones in $AMBERHOME if

'-amberlib' is specified. The -noX options can be used to disable use of certain libraries.

For example, to build cpptraj standalone:

```
./configure -amberlib gnu
make install OR cd src && make install
```

To build the documentation using *doxygen*, you must have *doxygen* installed, and you must have configured AmberTools. Run the command:

```
make docs
```

to build the documentation. PDF files and HTML files are generated during this process, showing class inheritance and descriptions from comments written in doxy-format. Open the file `$AMBERHOME/AmberTools/src/cpptraj/doc/html/index.html` to see the class heirarchy and descriptions.

3 General Layout and Concepts

Cpptraj currently lives in 2 key classes: **Cpptraj**, which is defined in main.cpp and controls overall flow (e.g. it is responsible for deciding whether to execute in batch mode or interactive moce), and **CpptrajState**, which is defined in **Cpptraj** and holds all of the data, Actions, Analyses, etc. The main components of **CpptrajState** are:

TopologyList parmFileList_; Hold all Topology classes. These are designed to be shared between trajectories to save memory.

DataSetList DSL_; Hold all DataSets. This is essentially how different components can talk to each other, e.g. an Action creates a DataSet in the DataSetList, which can then be used by a subsequent Analysis.

DataFileList DFL_; Hold all DataFiles. These are either for writing out DataSets or general text output, primarily from Actions/Analyses.

TrajinList trajinList_; Hold all input trajectories to be processed during a run. Whenever a user inputs a 'trajin' or 'ensemble' command, the trajectory/ensemble in question is added to this list. When a 'run' command is executed, these are the trajectories that are read in a frame at a time so that Actions in the ActionList can process them.

ActionList actionList_; Hold all Actions to be executed during a run. By default, whenver an Action command is issued the Action in question is initialized and queued up in the ActionList, to be processed during the next run.

TrajoutList trajoutList_; Hold output trajectories to be written during a run. This output occurs after all Actions have been processed.

AnalysisList analysisList_; Hold all Analyses to be executed after a run or when a 'runanalysis' command is given. Similar to ActionList, whenever an Analysis command is issued the Analysis in question is initialized and queued up in the AnalysisList.

A third "static" class, **Command**, controls how commands are processed. **Command** is the class responsible for translating user input. The file Command.cpp also contains all of the logic for executing commands.

4 Math-related Classes

4.1 Vec3

An array of 3 doubles, used to hold XYZ coords. Used for vector math.

4.2 Matrix_3x3

A 3x3 array of doubles, useful for performing rotations etc. Used for basic matrix math. Can be diagonalized via an internal routine (no need for external math library).

4.3 ComplexArray

Used to hold an array of complex numbers. Implemented as a double array instead of using the STL Complex class so that it easily interface with external routines.

4.4 PubFFT

Interface to FFT routines (either pubfft, which are the FFT routines used by Amber, or FFTW depending on how CPPTRAJ is configured). Currently only 1D forward and backwards FFTs are supported. Makes use of ComplexArray.

4.5 Corr.h: CorrF_Direct, CorrF_FFT

Classes used to calculate auto/cross correlation functions from arrays of complex numbers (ComplexArray).

5 Some Key Classes and Functions

The following is a brief list of some of the more commonly-used classes and functions in Cpptraj. Classes are more or less self-documented to a certain extent; this section will be focused on how these classes are/should be used.

5.1 Output to STDOUT/STDERR: CpptrajStdio.h

The file CpptrajStdio.cpp contains functions used to write output to standard output (STDOUT) and standard error (STDERR). This is accomplished with the C printf-like functions mprintf() and mprinterr() respectively:

```
mprintf(const char* format, ...); // STDOUT
mprinterr(const char* format, ...); // STDERR
```

The 'm' prefix stands for "master", and ensures that when CPPTRAJ is running via MPI that only the master thread is able to write with these functions (note the same does NOT apply for OpenMP). The syntax is the same as basic printf - a format string followed by any variables. There are numerous resources that describe printf syntax in detail. Some useful syntax is listed here:

%i Integer

%f Floating point number

%g Use scientific or floating point representation, whichever is shorter.

%s Character string

%c Single character

n Newline

t Tab

For example:

```
#include "CpptrajStdio.h"
int myInteger = 3;
double myDouble = 5.43;
string myString = "easy";
mprintf("Using printf is %s; %i is an integer and %f is a double.\n",
        myString.c_str(), myInteger, myDouble);
```

Note that the string function c_str() must be used to print C++ strings. Output:

```
Using printf is easy; 3 is an integer and 5.430000 is a double.
```

5.2 ArgList

The ArgList class is used throughout Cpptraj. It is the main way that user input is translated to actions, analyses, trajectory IO, etc. Basically, the ArgList class takes a string and separates it into tokens based on a given delimiter or delimiters. For example, the string:

```
myString = “trajin mytraj.nc 1 100 10”;
```

can be separated via a space (’ ’) delimeter (the default) into 5 tokens like so:

```
ArgList myArgs(myString);
```

The resulting ArgList internally looks something like:

```
0: trajin
1: mytraj.nc
2: 1
3: 100
4: 10
```

A custom delimeter string containing 1 or more characters can also be used. For example, the following string:

```
myString = “d01,d02,d03,d04”;
```

can be separated via a comma (’,’) delimiter into 4 tokens like so:

```
ArgList myArgs(myString, “,”);
```

The resulting ArgList internall lookws something like:

```
0: d01
1: d02
2: d03
3: d04
```

These tokens (or arguments) are stored internally as an STL vector of strings. ArgList provides many functions to access user arguments. A second array of boolean values records whether an argument has been accessed. This concept is functionally similar to the argumentStack in Ptraj; however, it avoids the constant memory allocation/deallocation when arguments are added/accessed, and allows an argument list to be re-used if desired. The two main ways arguments are usually accessed are through “GetNextX” and “GetKeyX” functions.

“GetNext” functions return the next argument of the desired type. For example, using the ArgList created in the first example from “trajin mytraj.nc 1 100 10” and assuming all arguments are unmarked, GetStringNext() would return “trajin”, while getNextInteger() would return “1”; in both cases the argument returned would be marked, so that a subsequent call to GetStringNext() would return “mytraj.nc” and so on. Another very commonly used “GetNext” function is the “GetMaskNext()” function, which returns the next atom mask expression (so noted because it will begin with ’:’, ’@’, ’*’); an example of this will be shown below.

“GetKey” functions return an argument next to a specified “key” string. Take for example the argument list created from “rmsd R1 @CA ref [myref] out rmsd.dat”:

```
0: rmsd
1: R1
2: @CA
2: ref
3: [myref]
4: out
5: rmsd.dat
```

If we want to access a specific argument, we use a “GetKey” function. For example, if we want to know the filename specified by ’out’, we would use GetStringKey(“out”); this would return “rmsd.dat”, and mark both “out” and “rmsd.dat”. Similarly, GetStringKey(“ref”) would return “[myref]”. At this point we could also use the GetMaskNext() function to get the atom mask expression “@CA”.

5.2.1 ArgList Example

```
ArgList myArgs(“test_command cutoff 2.0 nval 3 name MyTest :2-30@CA extra”);
double Cut = myArgs.getKeyDouble(“cutoff”, 0.0); // Value 2.0
int Nval = myArgs.getKeyInteger(“nval”, 0);      // Value 3
int Ntypes = myArgs.getKeyInteger(“ntypes”, 0);  // Value 0
std::string Name = myArgs.GetStringKey(“name”);  // Value “MyTest”
std::string Out = myArgs.GetStringKey(“out”);    // Empty
std::string maskExp = myArgs.GetMaskNext();      // Value “:2-30@CA”
std::string mask2Exp = myArgs.GetMaskNext();     // Empty
// At this point only “extra” will be unmarked.
```

5.3 Topology

The Topology class describes how a system is laid out in terms of Atoms, Residues, and Molecules (all of which are classes themselves). It may also hold parameters which describe interactions between Atoms (e.g. bonds, angles, dihedrals, etc). The Topology class is chiefly used in Trajectory input/output and setting up atom masks (see below).

The Topology class has several routines that return strings of atom and residue names:

std::string TruncResAtomName(int atom) Format: “<res name><res num>@<atom name>”

std::string AtomMaskName(int atom) Format: “:<res num>@<atom name>”

std::string TruncAtomNameNum(int atom) Format: “<atom name>_<atom num>”

std::string TruncResNameNum(int residue) Format: “<res name>:<res num>”

5.3.1 Examples

Iterate over all atoms in a certain residue. This can be accomplished like so:

```
// Iterate over all atoms in residue 4, print charge.
for (int atom_index = Top.Res(4).FirstAtom;
         atom_index != Top.Res(4).LastAtom;
       ++atom_index)
  mprintf(“Atom %i charge= %g\n”, atom_index+1, Top[atom_index].Charge());
```

Iterate over all atoms bonded to a certain atom and pick out the hydrogens:

```
// Iterate over all atoms bonded to atom 66
for (Atom::bond_iterator bond_atom = Top[66].bondbegin();
                         bond_atom != Top[66].bondend();
                       ++bond_atom)
  if (Top[*bond_atom].Element() == Atom::HYDROGEN)
    mprintf(“Hydrogen %s bonded to atom %s\n”,
            Top.AtomMaskName(*bond_atom).c_str(),
            Top.AtomMaskName(66).c_str());
```

5.4 AtomMask/CharMask

The AtomMask and CharMask classes keep track of what atoms for a given Topology are selected based on a given mask expression. The AtomMask class holds information on selected atoms only, while CharMask has the state of all atoms (selected/not selected). AtomMask is as an integer mask, where the atom numbers currently selected are stored as an array of integers. Since one is usually only interested in selected atoms, most times AtomMask is all that is needed and so is the most used mask class in Cpptraj; for example, all routines that take masks in the Frame class use the AtomMask class.

The CharMask class has an internal character array where the state of each atom is stored. It has functions that can then be used to interrogate if a certain atom or atoms are selected or not.

In typical use, there are 3 phases to using AtomMask or CharMask: 1) initialization with a mask expression, 2) setup via a Topology class, and 3) iteration over the mask/interrogation of the mask. Initialization with a mask expression performs all necessary tokenization of the mask expression string and prepares the mask to be set up, but does not actually select atoms. The mask expression can be used during AtomMask construction or passed in via SetMaskString():

```
AtomMask* Mask = new AtomMask(“@CA”);
AtomMask Mask(“@CA”);
AtomMask Mask; Mask.SetMaskString(“@CA”);
```

Setup occurs via a Topology class (since in order to set up a mask you need to know atom names/numbers, residue name/number/types etc). This can be done using SetupIntegerMask() or SetupCharMask() to set up an integer mask (more common) or a char mask:

```
AtomMask iMask(“@CA”);
Top.SetupIntegerMask( iMask );
CharMask cMask(“:1-20”);
Top.SetupCharMask( Mask );
```

Once a mask has been setup the Nselected() function returns the number of selected atoms, while the None() function returns true if no atoms were selected. If necessary, one can convert between the mask types post-setup by using AtomMask::ConvertToCharMask() / CharMask::ConvertToIntMask() routines:

```
AtomMask mask( CharMask.ConvertToIntMask(), CharMask.Natom() )
CharMask mask( AtomMask.ConvertToCharMask(), AtomMask.Nselected() )
```

The final stage is to make use of the atom mask. One can iterate over selected atoms in an integer atom mask using the STL-like const_iterator variable and begin() and end() functions - this is the recommended way to use atom masks:

```
for (AtomMask::const_iterator atomnum = Mask.begin();
                              atomnum != Mask.end();
                            ++atomnum)
  mprintf(“Selected atom %i\n”, *atomnum);
```

One can also access members of the integer array directly via the bracket (’[]’) operator:

```
for (int maskidx = 0; maskidx < Mask.Nselected(); ++maskidx)
  mprintf(“Selected atom %i\n”, Mask[atomidx]);
```

To see if atom(s) are selected in a CharMask, use the AtomInCharMask() and AtomsInCharMask() functions. The former returns true if a specified atom is selected, the latter returns true if any atoms within a given range are selected. For example:

```
for (int atom = 0; atom < Top.Natom(); ++atom)
  if (Mask.AtomInCharMask(atom))
    mprintf(“Selected Atom %i\n”, atom);
for (int rnum = 0; rnum < Top.Nres(); ++res)
  if (Mask.AtomsInCharMask( Top.Res(rnum).FirstAtom(),
                            Top.Res(rnum).LastAtom() ))
    mprintf(“Selected Residue %i\n”, rnum);
```

5.5 Frame

The Frame class is in many ways the workhorse of Cpptraj, as it holds all XYZ coordinates for a given input frame, and optionally box coordinates, masses, replica indices, temperature, time, and/or velocities. Note that although mass is stored in Topology, it is also stored in Frame since many calculations require it (center of mass, mass-weighted RMSD, etc). Coordinates and velocities are stored with double precision. Many routines are available to do things like calculate the center of mass of atoms, rotate, translate, scale, and so on. A major use of the Frame class is to perform RMSD calculations.

Frames are typically set up in two phases. The first phase is memory allocation, which occurs via constructors or the SetupFrameX routines. This should be done as little as possible since memory allocation is relatively expensive. The second phase is actually setting the coordinates, which occurs via the SetX routines. For example, the following code will set up a Frame (newFrame) with the coordinates from another Frame (oldFrame) based on a previously set up AtomMask (mask) and Topology (top) corresponding to oldFrame:

```
Frame newFrame;
// Allocate memory, copy in masses based on mask.
newFrame.SetupFrameFromMask( mask, top.Atoms() );
// Set only coordinates from oldFrame based on mask.
newFrame.SetCoordinates( oldFrame, mask );
```

Alternatively, this can be done in one step with a constructor:

```
Frame newFrame( oldFrame, mask );
```

The advantage of separating out Setup and Set is that memory reallocation is kept to a minimum. For example, if we wanted to use newFrame to hold a different set of coordinates (of the same size as newFrame or smaller) we might do something like:

```
newFrame.SetCoordinates( differentFrame );
```

If the new coordinates might be bigger than the current size of newFrame, we could explicitly call a SetupFrameX routine; this will only reallocate if the new size is greater than the current maximum size. A Frame remembers the largest size it was ever allocated for, so reallocation is kept to a minimum.

There are two basic ways to access coordinates within Frame:

const double* XYZ(atom) return pointer to beginning of XYZ coordinates for given atom (max Natom()).

const double* CRD(coord) return pointer to given coordinate (max size()).

Note that you can get pointers to the raw coordinates, but it is not recommened to use these in general.

5.5.1 Using Frame for RMSD calculations

Unlike some of the other functions of Frame, the RMSD functions do not take a mask - it is assumed all atoms in the Frame are involved in the RMSD calculation. This is done for performance reasons. If a subset of atoms is desired for an RMSD calculation the reference and target Frames should be modified beforehand. Since the reference structure usually does not change it is often beneficial to pre-center the reference at the origin. For example, given a reference Frame (Ref), a Farget frame (Tgt), and an AtomMask (mask):

```
bool useMass = false;
top.SetupIntegerMask( mask );
Frame selectedRef, selectedTgt;
// Set up and pre-center reference.
selectedRef.SetupFrameFromMask( mask, top.Atoms() );
selectedRef.SetCoordinates( Ref, mask );
// refTrans will contain translation from origin to reference.
Vec3 refTrans = selectedRef.CenterOnOrigin( useMass );
// Set up target.
selectedTgt.SetupFrameFromMask( mask, top.Atoms() );
selectedTgt.SetCoordinates( Tgt, mask );
// Calculate RMSD. tgtTrans is translation from target to origin.
Vec3 tgtTrans;
Matrix_3x3 rot_matrix;
double rmsd = selectedTgt.RMSD_CenteredRef( selectedRef,
                                            rot_matrix,
                                            tgtTrans,
                                            useMass );
// Best-fit rotate/translate current Target to Reference.
Tgt.Trans_Rot_Trans( tgtTrans, rot_matrix, refTrans );
```

Now for subsequent RMS calculations to the same Reference, only the selected Target frame needs to have its coordinates set:

```
selectedTgt.SetCoordinates( Tgt2, mask );
rmsd = selectedTgt.RMSD_CenteredRef( selectedRef, ...
```

6 Basic File Input/Output

6.1 CpptrajFile

The CpptrajFile class provides basic file input and output operations. It can handle reading and writing both Gzip and Bzip2 compressed files, and through the FileName class performs tilde-expansion on file names (via globbing) as well as separates the file name into its base name, extension, and compressed extension. The file can be opened immediately, or set up first and then opened later. Once it has been set up it can be opened or closed multiple times. The

CpptrajFile class destructor will automatically close the file if it is open at time of destruction.

6.2 BufferedLine

The BufferedLine class is a child of CpptrajFile used for text files that will be read in line by line (note that writing is not possible with this class). The class has an internal buffer, which chunks of the input file are read into. The Line() routine can be used to read that chunk line by line; this avoids potentially expensive file IO. When the chunk is empty a new chunk is read in. The line can be further split into Tokens (similar to ArgList) and read one token at a time; this can be useful for e.g. determining the number of columns in a file.

6.3 BufferedFrame

The BufferedFrame class is a child of CpptrajFile used for highly-formatted text files that will be read/written multiple lines at a time (such as the Amber ASCII trajectory format). This class is set up for a certain total number of elements of a certain character width with a certain number of elements per line, which can then be read to or written from a character buffer in one entire chunk.

7 Trajectory Input/Output

Trajectory input and output (IO) is handled via high-level and low-level classes. At the highest level trajectory input is provided by the Trajin_Single (for single input trajectories) and Trajin_Multi (for multiple input trajectories) classes, and Trajout for single output trajectories. At the lower level IO is handled by format-specific classes which inherit from TrajectoryIO (TrajectoryIO.h), which are called Traj_X by convention (e.g. Traj_AmberNetcdf for Amber NetCDF trajectories). TrajectoryIO classes are contained and set up within the higher level classes, with information specific to each TrajectoryIO class being stored in the TrajectoryFile base class.

7.1 Trajin_Single

7.2 Trajin_Multi

7.3 Trajout

8 Topology Input/Output

The *ParmIO* class is a base class for all topology file formats. This provides an easy mechanism for extracting the system topology from any number of file formats. The *ParmFile* class is a wrapper around the *ParmIO* classes that hides the implementation details for each data file type from you. You should

Table 1: Topology file formats currently implemented in Cpptraj. The first column has the *ParmIO* class name as well as the *ParmFormatType* enumeration type that corresponds to that class inside *ParmFile* in parentheses.

ParmIO Subclass (*ParmFormatType*)	Description
Parm_Amber (AMBERPARM)	Amber style topology file (OLD and NEW styles)
Parm_CharmmPsf (CHARMMPSF)	CHARMM PSF topology file format (used by NAMD, too)
Parm_Mol2 (MOL2FILE)	TRIPOS Mol2 file
Parm_PDB	PDB File

interact with *ParmIO* objects through *ParmFile* handlers. *ParmFile* provides the ability to both read and write topology file objects of any class.

One thing that sets *ParmIO* and *ParmFile* apart from *TrajectoryIO/TrajectoryFile* and *DataIO/DataFile* is its connection with the *Topology* class. *Topology* objects contain as much of the information in the Amber topology file as can be parsed from the information present in the ParmIO object (and figured out based on atomic arrangements). A *Topology* instance is the first argument passed to the *ParmFile::Read* function, followed by the name of the topology file. Unlike the *DataFile* and *TrajectoryFile* classes, *ParmFile* does not have a reference to the *ParmIO* object to forward read/write information to. It exists simply to fill the *Topology* class with the relevant data structures and inform it how to do the rest. The *Topology* class is format-independent, providing a layer of abstraction to make other parts of the code that require topology information less error-prone while coding.

Every *ParmIO* subclass implements a *ReadParm* method that takes a *Topology* instance as the first argument and fills as much of the information there as possible. Afterwards, the CommonSetup method of the Topology class is called to finish setup and determine bond information (from atom distances if not present directly in the file format) and molecule information (based on the bonded structure). The currently available types of topologies are summarized in 1.

9 The DataSet and DataFile Framework

One of the goals in writing cpptraj was to try and generalize data collection and output, so that any action could output any generated data in any format known to cpptraj without having to write any extra code. For example, data generated by a distance calculation can be output in columns, as a Grace file, as a Gnuplot file, or as all three. To that end Actions and Analyses have access to the main DataSetList in CpptrajState (usually named DSL inside actions) and the main DataFileList (usually named DFL inside actions). A DataSet can be generated by an Action, but because it is held outside the Action in the master DataSetList it can persist after the Action that generated it has been destroyed.

A simple usage example based on the 'distance' action (Action_Distance) is given here. The first step is to create the DataSet in the Init() routine.

```
dist_ = DSL->AddSet(DataSet::DOUBLE, actionArgs.GetStringNext(),"Dis");
if (dist_==0) return 1;
```

In the first line a DataSet class of type DOUBLE is added to the master DataSetList. The various types are enumerated in DataSet::DataType (DataSet.h). The DataSet will be named whatever the next string is in the actionArgs ArgList. If there is no name, a default one will be created based on the given default "Dis" and the DataSet's overall position in the DataSetList (so in this case the default could be something like Dis_00000). What is returned is a pointer to the DataSet; DataSet is actually a base class that specific DataSet types inherit (in this case DataSet_double). In this way the interface is generalized.

The next step is to add the DataSet to the DataFileList.

```
DFL->AddSetToFile(distanceFile, dist_);
```

In this line the DataSet dist_ is added to a file named distanceFile (string obtained previously with something like distanceFile = actionArgs.GetStringKey("out");). If distanceFile is empty the statement has no effect. If distanceFile is already in DFL the DataSet will be added to that file (in this way output from multiple actions can be combined rather than overwritten). The machinery of the DataFileList takes care of output from there.

The final phase is actually adding data to the DataSet. So for example in the action() routine you could have:

```
double distance = sqrt( DIST2_NoImage( V1, V2 ) );
dist_->Add(frameNum, &distance);
```

In the first line the value 'distance' is being calculated. In the next line the value from 'distance' is being added to DataSet dist_ with frame number 'frameNum' (automatically set within Action). Notice that the address of 'distance' is passed rather than the value; this is a necessity from the generalization of the DataSet interface. DataSet has no idea a prior what the data type might be, so in the Add routine the value is cast to what the underlying DataSet implementation expects. This allows the Add routine to be used for double, float, int, string, etc. This DataSet could also be cast back to it's actual type to access other routines, e.g.:

```
DataSet_double& ds_dist = static_cast<DataSet_double&>( *dist_ );
```

9.1 DataSet_1D

9.2 DataSet_2D

9.3 DataSet_3D

Part II
Adding New Functionality

Most development for Cpptraj will likely be in adding new functionality; actions, analyses, and trajectory/topology/data file formats. This part of the manual will provide guidance and some helpful hints to this end. In general, adding new functionality is done by writing an implementation of the desired class type (e.g. for actions, inherit from the Action class) and then adding that class to the container for that specific functionality (e,g, in the case of actions, ActionList).

10 Adding Actions - Example

All actions inherit from the Action abstract base class. The Action class itself inherits from the DispatchObject class so that it can be associated with an allocator (to create the action) and a help function. There are four functions that every action must implement: Init(), Setup(), DoAction(), and Print(). Init() is called when the action is first created, and processes input arguments, sets up DataSets/DataFiles, deals with reference frames, and sets the debug level. Setup() is called to set the action up for a specific topology, and so handles anything Topology-related (such as parsing atom masks). The DoAction() function is called to actually perform the action on input coordinate frames. The Init(), Setup(), and DoAction() functions return a special type of integer, Action::RetType, which described the result of the action:

Action::OK Action is successful.

Action::ERR Action is not successful.

Action::USEORIGINALFRAME Action requests that the original unmodified topology/frame be used (see e.g. Action_Unstrip in Action_Strip.h).

Action::SUPPRESSCOORDOUTPUT Action requests that further processing of the current coordinate frame be skipped (see e.g. Action_RunningAvg).

The final function is Print(), which is called after all trajectory processing is complete and performs any additional calculation or output necessary. This function can be blank if such functionality is not needed, but it still must be implemented.

In addition to Action, there are currently two additional action-related classes that actions may want to inherit from. The ImagedAction class is for classes that may need to calculate imaged distances, and the ActionFrameCounter class is for actions that may want to process subsets of input frames (see e.g. the Action_Matrix action).

As an example, we will go through the creation of the Action_Distance class for calculating distances; this will cover using the DataSet, DataFile, AtomMask, and ImagedAction classes as well.

10.1 Create the Class Header

As mentioned in the style guide, header files should be named after the class, so the Action_Distance class will go in a file named "Action_Distance.h". The first thing to do is create a "header guard" - this will prevent issues with multiple inclusion. The header guard should be named after the class and header file, so for Action_Distance.h:

```
#ifndef INC_ACTION_DISTANCE_H
#define INC_ACTION_DISTANCE_H
```

Next comes the class description. Since distance calculations may involve imaging we also include the ImagedAction class to simplify image handling:

```
class Action_Distance: public Action, ImagedAction {
```

Following the style guide, we first implement any public methods. For actions this is at least the constructor, the allocator (named Alloc() by convention), and the Help() function. The allocator and help functions need to be static so that they can be called without instantiating the class.

```
public:
  Action_Distance();  ///< Constructor
  /// Allocator
  static DispatchObject* Alloc() { return (DispatchObject*)new Action_Distance(); }
  static void Help(); ///< Help function
```

The implemented functions Init(), Setup(), DoAction(), and Print() can be either public or private, although the preference is private.

The private section is where all functions and variables specific to the class will go. First, we add entries for the functions inherited from the Action base class which must be implemented. Since we will not need to do any postprocessing for this action, the Print() function is empty:

```
Action::RetType Init(ArgList&, TopologyList*, DataSetList*,
                       DataFileList*, int);
  Action::RetType Setup(Topology*, Topology**);
  Action::RetType DoAction(int, Frame*, Frame**);
  void Print() {}
```

Next we define the class variables. For Action_Distance we will want a DataSet to hold the calculated distances, two AtomMasks to describe the points between which the distance should be calculated, and a variable to indicate whether the distance should be mass-weighted.

```
private:
  DataSet* dist_;  ///< Will hold DataSet of calculated distances.
  bool useMass_;   ///< If true, mass-weight distances.
  AtomMask Mask1_; ///< First atom selection.
  AtomMask Mask2_; ///< Second atom selection
```

All variables related to imaging are already include via the ImagedAction class.

Last, end the class definition and finish the header guard:

```
};
#endif
```

10.2 Create the Class Implementation

Following the naming scheme, the class implementation will go into Action_Distance.cpp. The first part of this file will have the necessary #include directives. We need <cmath> for the square root function, Action_Distance.h for the class definition, and CpptrajStdio.h for wiriting to the console.

```
#include <cmath>
#include "Action_Distance.h"
#include "CpptrajStdio.h"
```

First we will need to create the class constructor. It is encouraged that users make use of initalizer lists (which tend to be more efficient) for this purpose. In this case we have two non-class variables: dist_, which is a pointer to a DataSet, and useMass_, which is boolean:

```
Action_Distance::Action_Distance() : dist_(0), useMass_(true) {}
```

Next, ensure that the Help() function has an implementation. Note that in cpptraj "mprintf" is used over "printf" for making any future IO modifications easier:

```
void Action_Distance::Help() {
  mprintf("distance [<name>] <mask1> <mask2> [out <filename>] [geom] [noimage]\n");
}
```

10.2.1 Init() - Parse user arguments, set up DataSets/DataFiles etc

Init() is called when the action is created and is responsible for parsing the Argument list (ArgList) and inital setup. Init() has the same input arguments for every action:

```
Action::RetType
  Action_Distance::Init(ArgList& actionArgs, TopologyList* PFL,
                        DataSetList* DSL, DataFileList* DFL,
                        int debugIn)
{
```

The input arguments are as follows: actionArgs contains arguments from user input, PFL is the current list of Topology classes, DSL is the master DataSetList, DFL is the master DataFileList, and debugIn is the current debug level for actions. It is up to the action implementation whether it wants to record the debug level or not.

Typical order of argument processing is keywords, masks, DataSet name. First we will process the keywords 'noimage', 'geom', and 'out <filename>'.

In order to determine whether the action will try to use imaging we call the InitImaging() function (inherited from the ImagedAction class). If the ArgList actionArgs contains the string "noimage", false will be sent to InitImaging to disable imaging:

```
  InitImaging( !(actionArgs.hasKey("noimage")) );
```

Next we set useMass_. If actionArgs contains the string "geom", useMass_ will be set to false:

```
  useMass_ = !(actionArgs.hasKey("geom"));
```

Next, we will try to create an output DataFile:

```
DataFile* outfile = DFL->AddDataFile( actionArgs.GetStringKey("out"), actionArgs );
```

The behavior of AddDataFile() depends on the result from actionArgs.GetStringKey(); if "out" is present in actionArgs, the next string (presumably <filename>) is returned and passed to AddDataFile(), and a DataFile will be returned corresponding to <filename>. If "out" is not present nothing will be returned, no file will be set up, and outfile will be null (0).

Next, we will get two atom mask expressions. We will require that the user must specify two masks, so if either of the strings is empty return an error:

```
  std::string mask1 = actionArgs.GetMaskNext();
  std::string mask2 = actionArgs.GetMaskNext();
  if (mask1.empty() || mask2.empty()) {
    mprinterr("Error: distance: Requires 2 masks\n");
    return Action::ERR;
  }
```

Now we can use the mask expression strings to initialize the two AtomMask classes (note that this tokenizes the mask expressions but does not yet set them up since we need topology information to do that):

```
  Mask1_.SetMaskString(mask1);
  Mask2_.SetMaskString(mask2);
```

Next we will use the master DataSetList (DSL) to create a DataSet to store the calculated distances. We will set up a DataSet of type DOUBLE (for increased precision) using any string left in actionArgs, or if no strings are left a default name of "Dis". If any errors occur, NULL (0) will be returned:

```
dist_ = DSL->AddSet(DataSet::DOUBLE, actionArgs.GetStringNext(), "Dis");
  if (dist_==0) return Action::ERR;
```

If a DataFile was previously set up, we now add the DataSet to this DataFile:

```
  if (outfile != 0) outfile->AddSet( dist_ );
```

Last, we print out some information regarding how the Action has been initialized and return Action::OK to indicate successful intialization:

```
mprintf("    DISTANCE: %s to %s",Mask1_.MaskString(), Mask2_.MaskString());
  if (!UseImage())
    mprintf(", non-imaged");
  if (useMass_)
    mprintf(", center of mass");
  else
   mprintf(", geometric center");   mprintf(".\n");
  return Action::OK;
}
```

10.2.2 Setup() - Set up Topology-related parts of the Action

Setup() is called whenever the action needs to be set up for a given Topology file. Any component of the action that depends on Topology (in this case the AtomMasks and the Imaging) is handled here. The arguments to Setup() are:

```
Action::RetType
  Action_Distance::Setup(Topology* currentParm, Topology** parmAddress)
{
```

Note that both a pointer to the current Topology as well as the address of that pointer (Topology**) are given. This is for actions that may want to modify the current Topology (see e.g. Action_Strip).

IMPORTANT: Note that this is the only time in which a Topology is passed to the Action. If the Action requires Topology information later (such as in DoAction() or Print()) it must be saved in the Action itself (usually through a pointer).

First, we setup the AtomMasks. Each AtomMask is passed to the current topology using the SetupIntegerMask() function, which will create an integer array containing only the selected atoms based on the mask expression. If we needed to know both selected and unselected atoms we could use the SetupCharMask() function instead.

```
if (currentParm->SetupIntegerMask( Mask1_ )) return Action::ERR;
  if (currentParm->SetupIntegerMask( Mask2_ )) return Action::ERR;
```

After this, we print some information about what atoms are selected (note we could also use the MaskInfo() function of AtomMask for this). For calculating distance, we need to make sure atoms were actually selected (using the None() function of AtomMask):

```
mprintf("\t%s (%i atoms) to %s (%i atoms)",Mask1_.MaskString(), Mask1_.Nselected(),
          Mask2_.MaskString(),Mask2_.Nselected());
  if (Mask1_.None() || Mask2_.None()) {
    mprintf("\nWarning: distance: One or both masks have no atoms.\n");
    return Action::ERR;
  }
```

Next we determine if imaging can actually be performed based on the box information present in the current Topology; if there is no box information imaging cannot be performed. We do this with the SetupImaging() function, inherited from ImagedAction. The ImagingEnabled() function (also inherited from ImagedAction) will let us know if imaging for this Topology is possible or not:

```
SetupImaging( currentParm->BoxType() );
  if (ImagingEnabled())
    mprintf(", imaged");
  else
    mprintf(", imaging off");
  mprintf(".\n");
```

Now all Topology-dependent aspects of the action are set up. Return Action::OK.

```
  return Action::OK;
}
```

10.2.3 DoAction() - Process input Frame

Coordinates are read in a frame at a time and stored in a Frame class, which is then passed to each action in the ActionList. The DoAction() function is called to process a coordinate Frame. The arguments are:

```
Action::RetType
  Action_Distance::DoAction(int frameNum, Frame* currentFrame,
                            Frame** frameAddress)
{
```

The first argument is the current frame number (starting at 0). Note that like Setup(), both a pointer to the current Frame as well as the address of that pointer (Frame**) are given, again so that actions may modify the current Frame (see e.g. Action_Strip).

There are several variables needed for calculating the distance. First, we have two Vec3 classes (Vec3.h, which is already included from other headers) to store the XYZ coordinates of the points:

```
Vec3 a1, a2;
```

If we are performing non-orthorhombic imaging we need to store the matrices which perform conversion from Cartesian to fractional coordinates and vice versa (using Matrix_3x3 classes, Matrix_3x3.h). Note that these are called 'ucell' and 'recip' respectively throughout CPPTRAJ, as these were the names used for the analogous structures in PTRAJ.

```
Matrix_3x3 ucell, recip;
```

Finally, we need a double to store the actual result of the distance calculation:

```
double Dist;
```

In the first part of the actual calculation, we calculate the centers of the coordinates in Mask1_ and Mask2_, either mass-weighted or not depending on useMass_, using the appropriate functions from the Frame class (Frame.h):

```
if (useMass_) {
  a1 = currentFrame->VCenterOfMass( Mask1_ );
  a2 = currentFrame->VCenterOfMass( Mask2_ );
} else {
  a1 = currentFrame->VGeometricCenter( Mask1_ );
  a2 = currentFrame->VGeometricCenter( Mask2_ );
}
```

Next, we get the distance between the coordinates stored in a1 and a2. For non-orthorhombic imaging we first need to convert the current box coordinates (stored in the Frame class in double precision as 3 lengths and 3 angles) into the coordinate conversion matrices using the ToRecip() function of the Box class (Box.h). Then, depending on the type of imaging that needs to be performed we call the appropriate distance calculation routine (DIST2_XXX, found in DistRoutines.h):

```
    switch ( ImageType() ) {
      case NONORTHO:
        currentFrame->BoxCrd().ToRecip(ucell, recip);
        Dist = DIST2_ImageNonOrtho(a1, a2, ucell, recip);
        break;
      case ORTHO:
        Dist = DIST2_ImageOrtho(a1, a2, currentFrame->BoxCrd());
        break;
      case NOIMAGE:
        Dist = DIST2_NoImage(a1, a2);       break;
    }
    Dist = sqrt(Dist);
```

Last, we add the result to the DataSet and return Action::OK. Since DataSet is just an interface we pass in the address of Dist (&Dist) to let the underlying DataSet framework take care of the fact that it is a double.

```
    dist_->Add(frameNum, &Dist);
    return Action::OK;
  }
```

10.2.4 Print() - Any post-processing

The Print() function is called once all input frames have been read in, and is used if there is anything that should be printed outside the normal DataFile/DataSet framework (e.g. hydrogen bond averages in the hbond action) or if there are any additional calculations that need to be performed (e.g. finishing up matrix calculations in the matrix action). In this example we're only calculating a simple distance; the output is handled by the DataFile/DataSet framework, so we implement a blank Print() function in the header:

```
  void Print() {}
```

10.3 Add the Action to the Command class

Now that the class implementation is complete, we need to let cpptraj know how to call it. This is currently done using a static array in Command.cpp called Commands. An entry in Commands is called a Command::Token and consists of 5 fields:

1. Type: The CommandType (enumerated type), in this case ACTION.
2. Cmd: The keyword (const character string) that will call this action.

3. Alloc: The Allocator, which we previously defined in the header as Alloc().

4. Help: The help function, which we defined in the header and implemented as a static void function.

5. Fxn: A function pointer to the static function in Command.cpp which will handle this command; all Actions are handled by AddAction.

To make navigation of Commands.cpp easier, you can search for INC_ACTION (or INC_ANALYSIS if adding an Analysis) to go where things need to be added. First add the class to Commands.cpp with the appropriate '#include'.

```
#include "Action_Dihedral.h"
#include "Action_Distance.h"
#include "Action_Hbond.h"
```

Then will add the command entry to commands in the appropriate section (in this case DispatchObject::ACTION) alphabetically by keyword. The final array can look like e.g.:

```
const DispatchObject::Token Command::Commands[] = {
  { ACTION, "dipole", Action_Dipole::Alloc, Action_Dipole::Help, AddAction },
  { ACTION, "distance", Action_Distance::Alloc, Action_Distance::Help, AddAction },
  { ACTION, "drms", Action_DistRmsd::Alloc, Action_DistRmsd::Help, AddAction },
};
```