

A Brief Intro To

The *Ezprot Modules* &
The *Ezprot Class Library*
For Protein Structure Analysis

Frank K. Pettit

University of California at Los Angeles

Laboratory of Structural Biology & Molecular Medicine

The *Ezprot Modules*

- A collection of programs that do analyses of protein structures by reading (& sometimes writing) PDB files.

The *Ezprot Class Library*

- A library of C++ source code for functions and data type definitions shared by all the modules, allowing new modules to be written very quickly

Why are The Programs Called “Modules”?

- The Ezprot programs are highly standardized, and take very similar command-line arguments.
- They’re designed to *interconnect*: the output of one program can be piped directly into the input of another, even when hundreds of structures are being analyzed in one run.

A Simple Example of A Module

Module “**Hydrofob**” reads a PDB file and replaces the B-factor for each atom with the hydrophobicity (free energy of solvation) for its amino acid type.

Input File

Output File

hydrofob **-inpdb=** 1abc.pdb **-byresidue** **-outpdb=** 1abc.hyfo.pdb

All modules that read PDB files take argument “**-inpdb=**” for input file.

All modules that write PDB files take argument “**-outpdb=**” for output file.

Argument “**-byresidue**” is specific to this program only.

Now analyze the hydrophobicity of *hundreds* of structures just as easily!

Create a file listing the names of all PDB files to analyze:

File “proteases.lis”:

```
1wht.pdb
1svp.pdb
2kai.pdb
....and so on...
```

hydrofob **-pdblast=**proteases.lis **-byresidue** **-outpdb=**hyfo.pstrm

Output When Analyzing Many Structures at Once

File “proteases.lis”:

```
1wht.pdb
1svp.pdb
2kai.pdb
...and so on...
```

With an input list of filenames, Ezprot Modules will *LOOP* over many structures, analyzing them sequentially, one by one

```
hydrofob -pdblist=proteases.lis -outpdb= hyfo.pstrm
```

OR (easier) use Unix *output to file* operator “>”:

```
hydrofob -pdblist=proteases.lis > hyfo.pstrm
```

“PDB STREAM”

With either of the two commands above, *ALL OUTPUT STRUCTURES* (with hydrophobicities calculated) are *CONCATENATED* into a *SINGLE FILE* “hyfo.pstrm”

What if we want the output to be *in many files*, not just all crammed into one?

```
hydrofob -pdblist=proteases.lis -outpdb= %pdb%.hyfo.pdb
```

This writes many output PDB files, each with a name in which “%pdb%” is replaced by *input PDB code*

```
1wht.hyfo.pdb 1svp.hyfo.pdb 2kai.hyfo.pdb
```

“NAME SCHEME”

Modules Easily Connect Together

Example: Module “*Smoothout*” smoothes out the value for each atom by averaging values over its neighboring atoms within a fixed radius.

Let’s use *two steps* to smooth out hydrophobicities over a neighborhood.

```
hydrofob -pdblist=proteases.lis > hyfo.pstrm
smoothout -inpstrm= hyfo.pstrm -outpdb= %pdb%.smhyfo.pstrm
```

Command-line argument “-inpstrm=” tells the module that the input file is a **STRUCTURE STREAM**, the output of the previous command.

Let’s run both programs in *one step* -- on one command line!

```
hydrofob -pdblist=proteases.lis
| smoothout -inpstrm=- -outpdb= %pdb%.smhyfo.pstrm
```

Unix Operator “|” pipes
*output of 1st program into
input of 2nd program!!*

The “-” after “=” here means *standard input*
“-inpstrm=-” means “*read stream from std. input*”

Hundreds Of PDB Files Read, Analyzed & Written on ONE Command Line! Bitchin’!!

Summary of Module I/O Arguments

Input PDB

- inpdb= 1abc.pdb One PDB file on disk
- inpdb= - One PDB file from standard input
- pdblast= myprots.lis List of many PDB filenames
- inpstrm= myprots.pstrm Many structures concatenated in one file
- inpstrm= - Many structures concatenated on std. input

Output PDB

- outpdb= 1abc.pdb One PDB file on disk
- outpdb= myprots.pstrm Many structures concatenated in one file
- outpdb= - One or many structures to standard output
- outpdb= %pdb%.pdb Many PDB files, each with *different* names

More Cool Things with I/O Arguments: Reading *Part* of a PDB List using “:”, “%”, “^”

File “proteases.lis”:

```
1wht.pdb  
1svp.pdb  
2kai.pdb  
1qa7.pdb
```

```
hydrofob -pdblast=proteases.lis -outpdb= hyfo.pstrm
```

Reads **all** files: 1wht, 1svp, 2kai, 1qa7

```
hydrofob -pdblast=proteases.lis:3..4 -outpdb= hyfo.pstrm
```

Reads files #3 to 4: 2kai & 1qa7 **ONLY**

```
hydrofob -pdblast=proteases.lis:50..100% -outpdb= hyfo.pstrm
```

Reads second half (50-100%) of list: 2kai & 1qa7

```
hydrofob -pdblast=proteases.lis:^0..25% -outpdb= hyfo.pstrm
```

Reads **all EXCEPT** (“^”) first quarter (25%) of list: 1svp, 2kai & 1qa7

Jackknifing a Prediction Algorithm Made Easy By Using “:., %,^”

Suppose you have a *prediction algorithm* which is trained on a set of structures and tested on the same set. You want to avoid *training set bias* by making sure the algorithm is never tested on any structures it was trained on. The simplest way to guarantee this is called *jackknifing*.

Jackknifing is mind-bogglingly easy with Ezprot modules.

Usually there are two steps:

1. training with one program (e.g. “**trainer**”) to get parameters
2. predicting with another program (e.g. “**predictor**”)

```
trainer -pdblast=proteases.lis:^0..25% > training_params
```

Train on all structures *EXCEPT* (“^”) first quarter (25%)

```
predictor -pdblast=proteases.lis:0..25% < training_params
```

Test prediction algorithm on first quarter of structures *ONLY*

Repeat these steps for second quarter (25..50%), third quarter, fourth quarter, etc.

More Standard Arguments for Ezprot Modules

- Command-line argument “*-help*” will display full documentation file for that program (if it exists).
- Program name with *no* command-line arguments: displays list of possible arguments for that program (and their default values).

Some Ezprot Modules

Property Modules

- Generate biologically relevant oligomer from PDB file (Xtal AU)
- Build in hydrogens
- Atomic Charges
- Electrostatics w/Interface to Pymol Graphics
- Hydrophobicity
- Concavity
- Diffusion Accessibility (T. Yeates)
- Planarity
- Surface Roughness
- Find protein-protein interface
- Compute overall surface area
- Interface buried by substrate
- Compute oligomeric interface area

General Analysis Modules

- Smoothout Local Properties
- Find patches of a property
- Average Property over a Site
- Rotate Protein
- Strip Off Substrate/Heteroatoms
- Count Atoms

You Can Write Your Own Ezprot Programs!


(With the help of the [Ezprot Library](#),
if you know C and a *liittle* C++)

What You Need to Know About C Programming

Example: Compute Dot Product of Two Vectors

```
float x1, y1, z1;  
float x2, y2, z2;  
float dp;  
float dotproduct(float x1, float y1, float z1,  
                 float x2, float y2, float z2);
```

Call Function 6 Arguments



```
dp = dotproduct( x1, y1, z1, x2, y2, z2 );
```

Simplify this mess
by putting multiple
data into a “struct”:

```
struct vector3d { float x, y, z; };  
vector3d r1, r2;  
float dotproduct( vector3d r1, vector3d r2 );
```

Function 2 Arguments



```
dp = dotproduct( r1, r2 );
```

What You Need to Know About C++ Programming

How it works in C:

```
struct vector3d { float x, y, z; };  
vector3d r1, r2;  
float dotproduct(vector3d r1, vector3d r2);
```

Function

2 Arguments

dp = dotproduct(r1, r2);

How it works in C++:

```
class vector3d {  
    float x, y, z;  
    float dotproduct( vector3d r2);  
};  
vector3d r1, r2;
```

Multiple data is
stuffed in a “class”
not a “struct”

Called on:

Function

Argument

dp = **r1**.dotproduct(**r2**);

First argument “r1”
is *IMPLICIT* in
function call within
class declaration

A Class is a User-Defined Type of Data Object

Class Declaration:
Stored in header file "vector3d.h":

Class' Data Members



Function Members

(have *implicit* first argument)



```
class vector3d {  
    float x, y, z;  
    float dotproduct( vector3d r2);  
};
```

To Use the Class, you:

1. Include the pre-created *header file*
2. Declare *objects* of that class
3. Call member functions on the objects

```
#include "vector3d.h"  
  
vector3d r1, r2;  
  
dp = r1.dotproduct( r2);
```

What are “Objects” in Object-Oriented Programming (OOP)?

An “object” is a data type, like a struct, except you can’t access the data in it directly. *You can only access its data indirectly by calling its member functions.*

Member Function of object “r1”, of class “vector3d”:



```
dp = r1.dotproduct( r2 );
```

So a class often has *many* member functions for every conceivable purpose!!
How do you know what member functions a class has?

1. Read the documentation, or
2. Read the header file for that class (e.g. “vector3d.h”)

So What is the *Ezprot Library*?

The Ezprot Library is a large group of class definitions for objects relevant to protein structure analysis:

- `protein` class
- `aa_chain` class (amino acid chains)
- `amino_acid` class
- `atom` class

etc., etc... many others.

For example, the Ezprot header file “`protein.h`” declares the `protein` class:

Header file “`protein.h`”:

```
class protein
{
    /* data contained in proteins */
    /* member functions to call on proteins */
};
```

How Could You Use An Ezprot Class?

Simple Example: Read One PDB file, Count Atoms in Structure.

Your source file "myprog.cpp":

1. Include the pre-created *header file*
2. Declare *objects* of that class
3. Call member functions on the objects

```
#include "protein.h"
protein prot;
prot.read_pdb_file("1abc.pdb");
int n = prot.num_atoms();
```

Here, `read_pdb_file()` and `num_atoms()` are member functions of the `protein` class, declared in file `protein.h`.

Compiling an Ezprot Program

Compile your program by linking to the precompiled Ezprot library archive (called “libezp.a”):

Your source code Your output executable Standard math archive “libm.a”

```
gcc -I../ezprot myprog.cpp -o myprog.exe -L../ezprot -lezp -lm
```

Directory with all Ezprot header files Directory with Ezprot object archive “libezp.a” Names archive “libezp.a”

This assumes directory “../ezprot” has header files and archive libezp.a in it.

Stuff in **green** are standard C/C++ compiler arguments:
-I “include”, **-o** “output”, **-L** and **-l** “library directory and file”

Writing An Ezprot Module That Handles All Those Command-Line Arguments

Class `Ezp_instream` processes & stores all the *command-line arguments* relating to inputting one or more PDB files, like

`“-inpdb=”,`
`“-pdblast=”,`
`“-inpstrm=”`

`Ezp_instream` stores filenames only! (one or many)

The contents of PDB files are stored in `protein` objects.

```
#include "ezp_instream.h"
#include "protein.h"
int main( int argc, char *argv[])
{
    Count of command-line arguments      Array of command-line
    protein prot;                    argument strings
    instrm.get_options(&argc, argv);    Put filenames from
    instrm.open();                        command line into instrm
    while (prot.fscan_pdb(&instrm) == 1)
    {
        Each time we loop, read one PDB
        structure from stream
        /* Analyze Structure Here! */
    } /* End Loop Over Structures */
} /* Done Program */
```

Logical Structure of Ezprot Classes

- A `protein` contains multiple `aa_chains`.
- An `aa_chain` contains multiple `amino_acids`.
- An `amino_acid` contains multiple `atoms`.

All have many member functions to get at their data.

For example, there are `find()` member functions that let you find a particular `amino_acid` by name or number within a chain, find a particular `atom` by name within an `amino_acid`, etc.

Looping Through Collections

What if you want to *LOOP* through:

- All chains in a protein,
- Then all amino acids in each chain,
- Then all atoms in each amino acid,
and do something to *all* of them?

A Subtle Point About C++: Objects vs. Pointers to Objects

```
vector3d r1, r2;  
dp = r1.dotproduct( r2);
```

r1 is an object

```
vector3d *rp;  
rp = &r1; /* Get address */  
dp = (*rp).dotproduct( r2);
```

rp is a *pointer* to an object

**rp* is the *object pointed to* (i.e. r1)

Same thing, written differently:

```
dp = rp->dotproduct( r2);
```

Note “->” not period “.” for pointers

Pointers are often used to *iterate* through a collection of objects.
e.g., loop through all amino acids using a *pointer to amino acids*.

Ezprot Member Functions to Iterate Through Collections of Things

- **first()** Gives pointer to **first object** in collection
- **next()** Increments pointer to **next object** in collection
- **not_at_end()** Test if pointer has **gone beyond end**: returns 1 (keep going) or 0 (we're done)

If you call **first()** on **protein** class, it returns **pointer to an aa_chain**.

If you call **first()** on **aa_chain** class, it returns **pointer to an amino_acid**, etc.

These are member functions of all Ezprot *collections*, but the *type of pointer returned* depends on the kind of collection the function is called on.

Simple Example: Write All Atom Coordinates

```
#include "protein.h"
protein prot;
/* We use POINTERS as iterators: */
aa_chain    *aac;
amino_acid *aa;
atom        *at;
while (prot.fscan_pdb(&instrm) == 1)
{
    for (aac = prot.first(); prot.not_at_end(aac);
         aac = prot.next(aac))
        for (aa = aac->first(); aac->not_at_end(aa);
             aa = aac->next(aa))
            for (at = aa->first(); aa->not_at_end(at); ++at)
                printf("Atom Coords: %f,%f,%f\n", /* Writing Coords*/
                       at->x(), at->y(), at->z());
}
```

Three iterators for aa_chain, amino_acid, atom:
All are pointers to objects, not objects.

Loop over all chains in protein

Loop over amino acids in chain

Loop over all atoms in amino_acid

Biggest Improvements to Ezprot 2.0 vs. Ezprot 1.0

- In v. 2.0, we added the classes `Ezp_instream` and `Ezp_outstream`, which give all modules the ability to handle command-line I/O arguments “-inpdb=”, “-pdblist=”, “-inpstrm=”, “-outpdb=” in a standardized way. These were not present or not standardized in v. 1.0.
- In v. 2.0, we added **new iterator types** that let you loop through chains, amino_acids, atoms, etc. **without pointers**, in a fashion analogous to the **C++ Standard Template Library (STL)**. The original pointer-based iterating functions `first()`, `next()`, `not_at_end()` (from Ezprot v.1.0) still work.

Ezprot 2.0 Permits STL-Style Iteration Through Collections

In addition to the v.1.0 iterator functions `first()`, `next()`, `not_at_end()`, Ezprot 2.0 also allows iteration using Standard Template Library (STL)-style functions. STL is a collection of commonly-used classes for basic data types in C++.

Ezprot v.2.0 does not employ STL but allows STL-style iteration for those who are comfortable with STL. Ezprot2.0 collection classes (like `protein`, `aa_chain`, `amino_acid`) also include most (not all) functions required by STL standard for collection classes (e.g. functions `size()`, `clear()`)

Ezprot v. 1.0

STL-style

Iterate with: pointer to object

iterator object (called `iter` in Ezprot)

Declare iterators: `aa_chain *aac;`
`amino_acid *aa;`

`protein:iter aacit;`
`aa_chain::iter aait;`

Initialize loop: `aa = aac->first();`

`aait = aacit->begin();`

Test for end of loop: `aac->not_at_end(aa)`

`aait != aacit->end()`

Increment to next: `aa = aac->next(aa);`

`++aait;`

STL-Style Example: Write All Atom Coordinates

```
#include "protein.h"
protein prot;
/* We use ITERATOR OBJECTS in STL-style: */
protein::iter    aacit;
aa_chain::iter  aait;
amino_acid::iter atit;
while (prot.fscan_pdb(&instrm) == 1)
{
    for (aacit = prot.begin(); aacit != prot.end();
        ++aacit)
        for (aait = aacit->begin(); aait != aacit->end();
            ++aait)
            for (atit = aait->begin(); atit != aait->end(); ++atit)
                printf("Atom Coords: %f,%f,%f\n", /* Writing Coords*/
                    atit->x(), atit->y(), atit->z());
}
```

Three iterators for aa_chain, amino_acid, atom:
All are iterators for the class they're part of.

← Loop over all chains in protein

← Loop over amino acids in chain

← Loop over all atoms in amino_acid

Where to Start Coding?

Example Source Code Templates Are Useful for Learning to Write Modules

Some Ezprot modules are very simple, “bare bones” code and can be used as **templates** that you start with, then build on to write your own modules.

- [ezpmodule.cpp](#): no function, just PDB input/output
- [gen_olig.cpp](#): reads PDB file, transforms PDB crystal asymmetric unit (AU) to biological oligomer (by reading a spatial transformation (matrix)), then writes transformed PDB file

These modules & others are included with Ezprot source code library.

Acknowledgements

- David Grosfeld
- Jennifer Padilla
- James U. Bowie
- Duillio Cascio