Xplor-NIH: An Introduction

Charles Schwieters

Center for Information Technology
National Institutes of Health

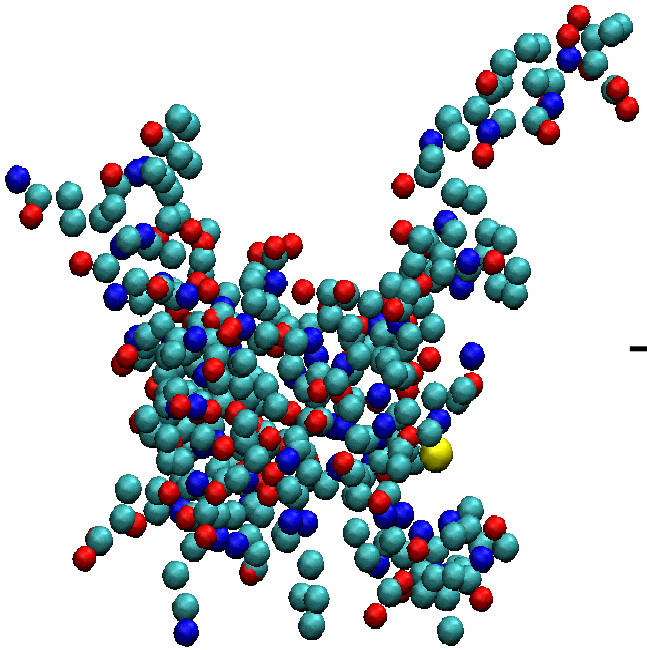
Bethesda, MD USA

December 8, 2006

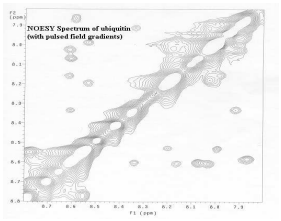
outline

1. description, history, installation
2. Scripting Languages: XPLOR, Python, TCL
   - Introduction to Python
3. Potential terms available from Python
4. IVM: dynamics and minimization in internal coordinates
5. Parallel determination of multiple structures
   - Using Beowulf clusters
6. VMD molecular graphics interface
7. line-by-line analysis of an Xplor-NIH script.
8. refinement against solution scattering data.
9. ensemble refinement.

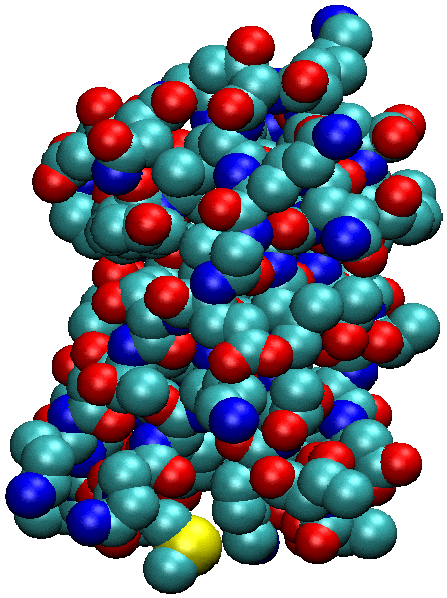
goal of this session:
Xplor-NIH's Python interface will be introduced, described in enough detail such that scripts can be understood, and modified.



unknown atom positions



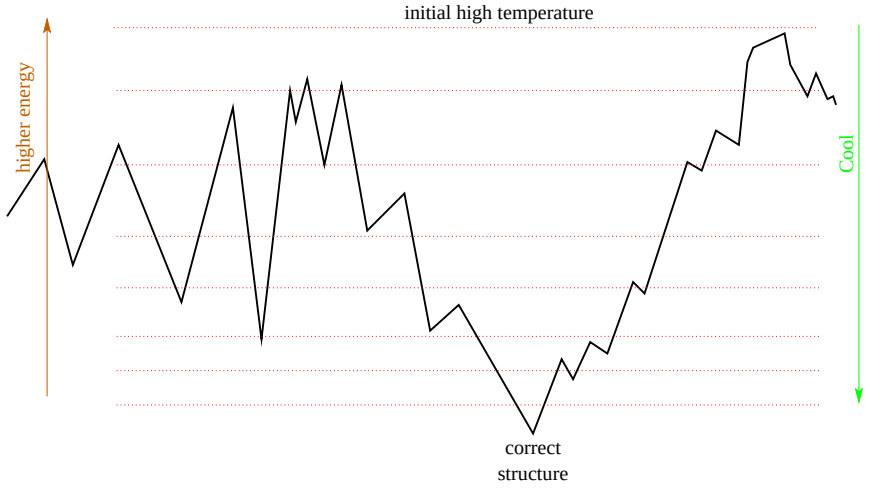
?



correct structure

Minimize energy: $V_{\text{tot}} = V_{\text{noe}} + V_{\text{bond}} + V_{\text{repel}} + \ldots$

- molecular dynamics to explore the energy surface.
- slowly decrease the temperature to find the global minimum.



What is Xplor-NIH?

Biomolecular structure determination/manipulation

- Determine structure using minimization protocols based on molecular dynamics/ simulated annealing.
- Potential energy terms:
  - terms based on input from NMR (and X-ray crystal) experiments: NOE, dipolar coupling, chemical shift data, SAXS, SANS, etc,
  - other potential terms enforce reasonable covalent geometry (bonds and angles).
  - knowledge-based potential terms incorporate info from structure database.
- includes: program, topology, covalent parameters , potential energy parameters, databases for knowledge-based potentials, helper programs, example scripts, and high level protocols for structure determination and analysis.
- freely available for non-industrial work. Source code is available.

Xplor-NIH Description

New contributions, additions are encouraged.

Source code of Xplor-NIH:

- original XPLOR Fortran source, with contributions from many groups.
- current work uses C++ for compute-intensive work.
- scripts and much code are written in Python, TCL scripting languanges.
- SWIG used to "glue" scripting languanges to C++.

XPLOR History

1. approx. 1984. Initially a fork of CHARMM by Axel Brünger – for NMR and X-ray structure determination.

2. rights sold to what is now Accelrys corp.

3. 1998. A. Brünger and coworkers develop replacement program CNS.

4. approx 2000. Development of CNS is stopped.

5. 2002. Agreement between Accelrys and NIH allowing distribution of legacy XPLOR code with Xplor-NIH. This allows complete backward compatibility with legacy XPLOR scripts. C++, Python and TCL code has no restrictions.

Installation

1. download two files from `http://nmr.cit.nih.gov/xplor-nih/`

   (a) a -db file: *e.g.* `xplor-nih-2.16-db.tar.gz`

   (b) a platform-specific file: *e.g.* `xplor-nih-2.16-Linux_2.4_i686.tar.gz`

2. unpack these files where you wish them to live:

```
zcat xplor-nih-2.16-db.tar.gz | (cd /opt ; tar xf -)
zcat xplor-nih-2.16-Linux_2.4_i686.tar.gz | (cd /opt ; tar xf -)
```

3. perform initial configuration:

```
cd /opt/xplor-nih-2.16
./configure -symlinks /usr/local/bin
```

   (the `-symlinks` option creates symbolic links in the specified directory for xplor and other commands - it is not necessary.)

4. test the new installation:

```
bin/testDist
```

Scripting Languages- three choices

scripting language:

- flexible interpreted language
- used to input filenames, parameters, protocols
- flexible enough to program non compute-intensive logic
- relatively user-friendly

XPLOR language:

strong point:

atom selection language quite powerful.

weaknesses:

String, Math support problematic.

no support for subroutines: difficult to encapsulate functionality.

Parser is hand-coded in Fortran: difficult to update.

XPLOR reference manual:

`http://nmr.cit.nih.gov/xplor-nih/doc/current/xplor/`

NOTE: all old XPLOR 3.851 scripts should run unmodified with Xplor-NIH

Language Examples - printing 1..10

XPLOR

```
eval ($i=1)
while ($i le 10) loop printloop
  display $i
  eval ($i=$i+1)
end loop printloop
```

```
1
2
3
4
5
6
7
8
9
10
```

Python

```
for i in range(1,11):
  print i
```

```
1
2
3
4
5
6
7
8
9
10
```

TCL

```
for {set i 1}
    {$i<11}
    {incr i} {
  puts $i
}
```

```
1
2
3
4
5
6
7
8
9
10
```

general purpose scripting languages: Python and TCL

- excellent string support.
- languages have functions and modules: can be used to better encapsulate protocols (e.g. call a function to perform simulated annealing.)
- well known: these languages are useful for other computing needs: replacements for AWK, shell scripting, etc.

- Facilitate interaction, tighter coupling with other tools.
  - NMRWish has a TCL interface.
  - pyMol has a Python interface.
  - VMD has TCL and Python interfaces.

separate processing of input files (assignment tables) is unnecessary: can all be done using Xplor-NIH.

Introduction to Python

assignment and strings

```
a = 'a string'  # <- pound char introduces a comment
a = "a string"  # ' and " chars have same functionality
```

multiline strings - use three ' or " characters

```
a = '''a multiline
string'''
```

C-style string formatting - uses the % operator

```
s = "a float: %5.2f   an integer: %d" % (3.14159, 42)
print s
a float:  3.14   an integer: 42
```

raw strings - special characters are not translated

```
a = r'strange characters: \%~!'  # introduced by an r
```

lists and tuples

```
l = [1,2,3]           #create a list
a = l[1]              #indexed from 0 (a = 2)
l[2] = 42             # l is now [1,2,42]
t = (1,2,3)           #create a tuple (read-only list)
a = t[1]              # a = 2
t[2] = 42             # ERROR!
```

Introduction to Python

calling functions

```
bigger = max(4,5)  # max is a built-in function
```

defining functions - whitespace scoping

```
def sum(item1,item2,item3=0):
    "return the sum of the arguments"  # comment string
    retVal = item1+item2+item3         # note indentation
    return retVal
print sum(42,1)                    #un-indented line: not in function
43
```

using keyword arguments - specify arguments using the argument name

```
print sum(item3=2,item1=37,item2=3)    # argument order is not important
42
```

loops - the for statement

```
for cnt in range(0,3):   # loop over the list [0,1,2]
    cnt += 10
    print cnt


10
11
```

Introduction to Python

Python is modular

most functions live in separate namespaces called modules

Loading modules - the import statement

```
import sys          #import module sys
sys.exit(0)         #call the function exit in module sys
```

or:

```
from sys import exit  #import exit function from sys into current scope
exit(0)               #don't need to prepend sys.
```

Introduction to Python

In Python objects are everywhere.

Objects: associated functions called *methods*

```
file = open("filename")   #open is built-in function returning an object
contents = file.read()    #read is a method of this object
                          # returns a string containing file contents

dir(file)                 # list all methods of object file
```

```
['__class__', '__delattr__', '__doc__', '__getattribute__',
  '__hash__', '__init__', '__iter__', '__new__', '__reduce__',
  '__repr__', '__setattr__', '__str__', 'close', 'closed', 'fileno',
  'flush', 'isatty', 'mode', 'name', 'read', 'readinto', 'readline',
  'readlines', 'seek', 'softspace', 'tell', 'truncate', 'write',
  'writelines', 'xreadlines']
```

Introduction to Python

A mapping type: Dictionaries

```
d={}
d['any'] = 4            #elements indexed like arrays
d['string'] = 5         # but the index can be (almost) any type
print d['string']
5
d.keys()                #return list of all index keys
d.values()              #return list of all indexed values
```

Tools for List Processing:
map - convert one list to another list:

```
map(int, ['1','2','3'])   # apply int() function to list of strings
[1, 2, 3]
```

lambda - a simple function with no name

```
twoTimes=lambda x: 2*x  # define twoTimes to be a lambda function
twoTimes(3)
6
```

lambdas are useful when used with map:

```
map(lambda c: 2*int(c), ['1','2','3'])  # convert string list to ints
                                        # with multiplication
[2, 4, 6]
```

Introduction to Python

interactive help functionality: `dir()` is your friend!

```
import sys
dir(sys)    #lists names in module sys
dir()       # list names in current (global) namespace
dir(1)      # list of methods of an integer object
```

the help function

```
import ivm
help( ivm )          #help on the ivm module
help(open)           # help on the built-in function open
```

browse the Xplor-NIH python library using your web-browser on your local workstation:

```
% xplor -py -pydoc -g
```

Xplor-NIH Python module reference:
`http://nmr.cit.nih.gov/xplor-nih/doc/current/python/ref/index.html`

Python in Xplor-NIH

current status: low-level functionality (similar to that of XPLOR script) implemented.

mostly implemented: high-level wrapper functions which will encode default values, and hide complexity.

future: develop repository of still-higher level protocols to further simplify structure determination.

stability: Python interface fairly stable. Small changes possible.

Accessing Xplor-NIH's Python interpreter

from the command-line: use the -py flag:

```
% xplor -py
                        XPLOR-NIH version 2.16

 C.D. Schwieters, J.J.  Kuszewski,       based on X-PLOR 3.851 by A.T. Brunger
 N. Tjandra, and G.M. Clore
 J. Magn. Res., 160, 66-74 (2003).       http://nmr.cit.nih.gov/xplor-nih

python>
```

or the pyXplor executable - a bit quieter- and can be used as a complete replacement for the python command:

```
% pyXplor
python>
```

or as an extension to an external Python interpreter:

```
% ( eval `xplor -csh-env` ; python)

Python 2.4.4 (#1, Oct 26 2006, 10:23:26)
[GCC 3.4.3] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import xplorNIH
>>> execfile('script.py')
```

[this requires that the version of Python be consistent between the external interpreter and Xplor-NIH.]

accessing Python from XPLOR: PYTHon command

```
% xplor
                        XPLOR-NIH version 2.16

 C.D. Schwieters, J.J.  Kuszewski,       based on X-PLOR 3.851 by A.T. Brunger
 N. Tjandra, and G.M. Clore
 J. Magn. Res., 160, 66-74 (2003).       http://nmr.cit.nih.gov/xplor-nih

 User: schwitrs     on: khaki     (x86/Linux     )  at:  7-Dec-06 12:37:40
X-PLOR>python            !NOTE: can't be used inside an XPLOR loop!
python> print 'hello world!'
hello world!
python> python_end()
X-PLOR>
```

for a single line: CPYThon command

```
X-PLOR>cpython "print 'hello world!'"   !can be used in a loop
hello world!
X-PLOR>
```

using XPLOR, TCL from Python

to call the XPLOR interpreter from Python

```
xplor.command('''struct @1gb1.psf end
                 coor @1gb1.pdb''')
```

`xplor` is a built-in module - no need to import it for simple scripts.

to call the TCL interpreter from Python

```
from tclInterp import TCLInterp              #import function
tcl = TCLInterp()                            #create TCLInterp object
tcl.command('xplorSim setRandomSeed 778') #initialize random seed
```

Atom Selections in Python

use the XPLOR atom selection language, described in the XPLOR manual.

```
from atomSel import AtomSel
sel = AtomSel('''resid 22:30 and
                 (name CA or name C or name N)''')
print sel.string()          #AtomSel objs remember their selection string


resid 22:30 and
                 (name CA or name C or name N)
```

AtomSel objects can be used as lists of Atom objects

```
print len(sel)                        # prints number of atoms in sel
for atom in sel:                      # iterate through atoms in sel
   print atom.string(), atom.pos()    # prints atom string, and its position.
```

AtomSel objects can be *reevaluated*:

```
xplor.command("delete sele=(resid 1:2) end")
sel.reevaluate()
print len(sel)              # prints the correct number
```

Using potential terms in Python

available potential terms in the following modules:

1. rdcPot - dipolar coupling
2. csaPot - Chemical Shift Anisotropy
3. noePot - NOE distance restraints
4. jCoupPot - ^{3}J-coupling
5. prePot - Paramagnetic relaxation enhancement
6. xplorPot - use XPLOR potential terms
7. solnScatPot - potential for solution X-ray and neutron scattering
8. posSymmPot - restrain atomic positions relative to those in a similar structure
9. potList - a collection of potential terms in a list-like object.

all potential objects have the following methods:

instanceName()	-	name given by user
potName()	-	name of potential term, e.g. "RDCPot"
scale()	-	scale factor or weight
setScale(val)	-	set this weight
calcEnergy()	-	calculate and return term's energy

residual dipolar coupling potential

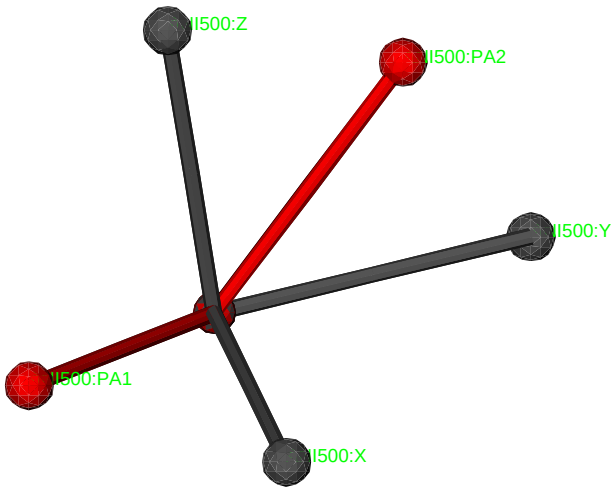
Provides orientational information relative to axis fixed in molecule frame.

$$D^{\mathsf{AB}} = D_a[(3u_z^2 - 1) + \frac{3}{2}R(u_x^2 - u_y^2)] \ ,$$

u_x, u_y, u_z- projection of bond vector onto axes of an alignment tensor.
D_a, R- measure of axial and rhombic tensor components.

rdcPot (in Python)

- tensor orientation encoded in four axis atoms
- allows Da, R to vary: values encoded using extra atoms.
- reads both SANI and DIPO XPLOR assignment tables.
- allows multiple assignments for bond-vector atoms - for averaging.
- allows ignoring sign of D_a (optional)
- can (optionally) include distance dependence: $D_a \propto 1/r^3$.
- tensor values can be computed using SVD.



How to use the rdcPot potential

```
from varTensorTools import create_VarTensor, calcTensor
ptensor = create_VarTensor('phage')   #create a tensor object

ptensor.setDa(7.8)          #set initial tensor Da, rhombicity
ptensor.setRh(0.3)
ptensor.setFreedom('varyDa, varyRh') #allow Da, Rh to vary

from rdcPotTools import create_RDCPot
rdcNH = create_RDCPot("NH",oTensor=ptensor,file='NH.tbl')

calcTensor(ptensor)   #calc tensor parameters from current structure using SVD
```

NOTE: no need to have psf files or coordinates for axis/parameter atoms- this is automatic.

analysis, accessing potential values:

```
print rdcNH.instanceName()              # prints 'NH'
print rdcNH.potName()                   # prints 'RDCPot'
print rdcNH.rms(), rdcNH.violations()   # calculates and prints rms, violations
print ptensor.Da(), ptensor.Rh()        # prints these tensor quantities
rdcNH.setThreshold(0)                   # violation threshold
print rdcNH.showViolations()            # print out list of violated terms
from rdcPotTools import Rfactor
print Rfactor(rdcNH)                    # calculate and print a quality factor
```

RDCPot: additional details

using multiple media:

```
btensor=create_VarTensor('bicelle')
rdcNH_2 = create_RDCPot("NH_2",tensor=btensor,file='NH_2.tbl')
#[ set initial tensor parameters ]
btensor.setFreedom('fixAxisTo phage') #orientation same as phage
                                      #Da, Rh vary
```

multiple expts. single medium:

```
rdcCAHA = create_RDCPot("CAHA",oTensor=ptensor,file='CAHA.tbl')
```

`rdcCAHA` is a new potential term using the same alignment tensor as `rdcNH`.

Scaling convention: scale factor of non-NH terms is determined using the experimental error relative to the NH term:

```
scale_toNH(rdcCAHA,'CAHA')   #rescales RDC prefactor relative to NH
```

Use harmonic potential with correct relative scaling

```
scale = (5/2)**2
      #  ^ inverse error in expt. measurement relative to that for NH
rdcCAHA.setScale( scale )
```

Chemical Shift Anisotropy potential

Provides additional orientational information from the full chemical shift tensor.

$$\Delta\delta = \sum_{i,j} A_i \sigma_j \cos^2(\theta_{i,j})$$

A_i - a principal moment of the alignment tensor
σ_j - a principal moment of the CSA tensor
$\theta_{i,j}$ - angle between the i^{th} orientation tensor principal axis and the j^{th} CSA tensor principal axis.

How to use the csaPot potential

```
from csaPotTools import create_CSAPot
csaP = create_CSAPot(name,oTensor=tensor,file='csaP.tbl')

csaP.setDaScale( val )    # s.t. can be used with RDC alignment tensor
csaP.setScale( forceConstant )
calcTensor(tensor)        #use if the structure is approximately correct
```

NOTE: `create_CSAPot` determines the atom type involved and uses built-in values for the chemical shift tensor. Alternate values can be specified by modifying csaPotTools.csaData.

NOE potential term

most commonly used effective NOE distance (sum averaging):

$$R = (\sum_{ij} |q_i - q_j|^{-6})^{-1/6}$$

Python potential in module noePot

- reads XPLOR-style NOE tables.
- potential object has methods to set averaging type, potential type, etc.
- most commonly use a soft piecewise quadratic potential which allows for errors.

creating an NOEPot object:

```
from noePotTools import create_NOEPot
noe = create_NOEPot("noe","noe_all.tbl")
```

analysis:

```
print noe.rms()
noe.setThreshold( 0.1 )          # violation threshold
print noe.violations()           # number of violations
print noe.showViolations()
```

J-coupling potential

Karplus relationship

$$^3J = A\cos^2(\theta + \theta^*) + B\cos(\theta + \theta^*) + C,$$

θ is a torsion angle, defined by four atoms.
A, B, C and θ^* are set using the COEF statement in the j-coupling assignment table (or using object methods).

Use in Python

```
from jCoupPotTools import create_JCoupPot
  # set Karplus parameters while creating the potential term.
jCoup = create_JCoupPot("hnha","jna_coup.tbl",
                        A=15.3,B=-6.1,C=1.6,phase=0)
```

analysis:

```
print Jhnha.rms()
print Jhnha.violations()
print Jhnha.showViolations()
```

using XPLOR potentials

Example using a Radius of Gyration (COLLapse) potential

```
import protocol
from xplorPot import XplorPot
                                        #helper setup function
protocol.initCollapse('resid 3:72')  #specify globular portion
rGyr = XplorPot('COLL')
xplor.command('collapse scale 0.1 end') #manipulate in XPLOR interface
```

default target=$(2.2 * \text{numResidues}^{0.38} - 1)$
(empirical relationship for compact globular proteins)

accessing associated values

```
print rGyr.calcEnergy().energy        #term's energy
print rGyr.potName()                  # 'XplorPot'
print rGyr.instanceName()             # 'COLL'
```

all other access/analysis done from XPLOR interface.

The XPLOR non-bonded potential

```
import protocol
from xplorPot import XplorPot
protocol.initNBond(repel=1.2)    #specify nonbonded paramters
vdw = XplorPot('VDW')
print vdw.violations()           #print number of overlapping atom pairs
```

All parameters for the nonbonded term are listed in the XPLOR manual

The XPLOR RAMA (torsion angle database) potential

```
import protocol
from xplorPot import XplorPot
protocol.initRamaDatabase()
potList.append( XplorPot('RAMA') )
```

Other commonly used XPLOR terms: BOND, ANGL, IMPR, HBDA, CDHI.

Enumeration of Xplor potential terms

- 3J couplings
- 1J couplings
- ^{13}C shifts
- ^{1}H shifts
- radius of gyration
- chemical shift anisotropy
- conformational database torsion angle potentials
- database residue-residue and base-base positioning potentials
- Generalized Born implicit solvent (Tom Simonson - Strasbourg) .
- PARArestraints module for including paramagnetism-based NMR restraints in refinement (Bertini's group - Italy)
- isac code for floating RDC alignment tensor (Grzesiek's group, Basel)
- vectang pot. for indirect RDC restraints (Michael Nilges' group)
- hbda pot.- empirical bb H-bond relationship
- hbdb pot.- precise database for for bb H-bonds (A. Grishaev. - NIH)

collections of potentials - PotList

potential term which is a collection of potentials:

```
from potList import PotList
pots = PotList()
pots.append(noe); pots.append(Jhnha); pots.append(rGyr)
pots.calcEnergy().energy                              # total energy
```

nested PotLists:

```
rdcs = PotList('rdcs')                    #convenient to collect like terms
rdcs.append( rdcNH ); rdcs.append( rdcNH_2 )
rdcs.setScale( 0.5)                       #set overall scale factor
pots.append( rdcs )
for pot in pots:                          #pots looks like a Python list
    print pot.instanceName()
```

```
noe
hnha
COLL
rdcs
```

Implementing a new potential term - in Python

```
from pyPot import PyPot ; from vec3 import norm
class BondPot(PyPot):
    ''' example class to evaluate energy, derivs of a single bond
    '''
    def __init__(self,name,atom1,atom2,length,forcec=1):
        ''' constructor - force constant is optional.'''
        PyPot.__init__(self,name) #first call base class constructor
        self.a1 = atom1     ; self.a2     = atom2
        self.length = length; self.forcec = forcec
        return
    def calcEnergy(self):
        self.q1 = self.a1.pos() ; self.q2 = self.a2.pos()
        self.dist = norm(q1-q2)
        return 0.5 * self.forcec * (dist-self.length)**2
    def calcEnergyAndDerivs(self,derivs):
        energy = self.calcEnergy()
        deriv1 = map(lambda x,y:x*y,
                     [self.forcec * (self.dist-self.length) / self.dist]*3 ,
                     ( self.q1[0]-self.q2[0],
                       self.q1[1]-self.q2[1],
                       self.q1[2]-self.q2[2] ))
        derivs[self.a1.index()] = deriv1
        derivs[self.a2.index()] = -deriv1
        return energy
    pass
```

to use:

```
p = BondPot('bond',AtomSel('resid 1 and name C')[0],
                   AtomSel('resid 1 and name O')[0], length=1.5)
```

The IVM (internal variable module)

Used for dynamics and minimization

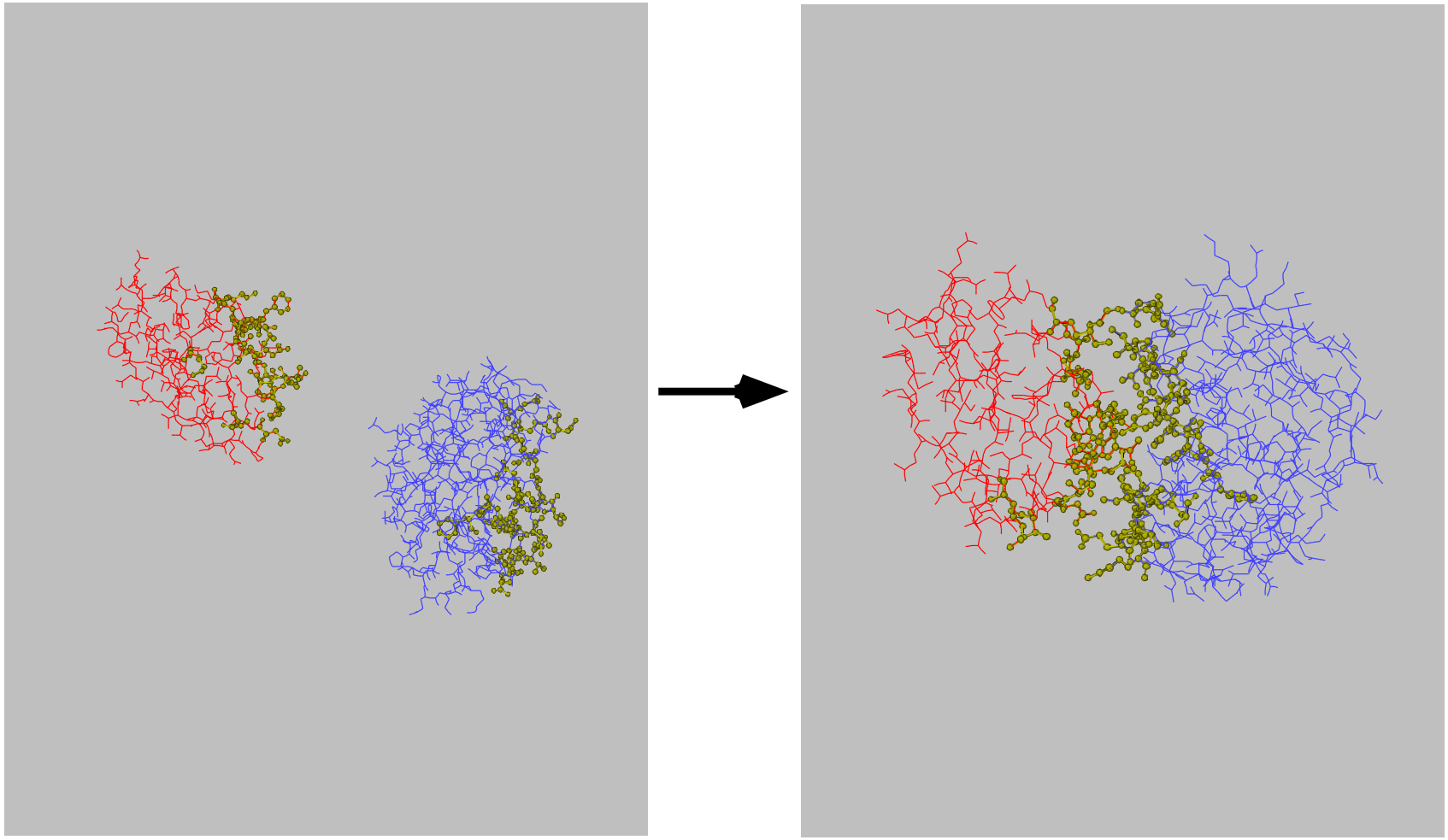
in biomolecular NMR structure determination, many internal coordinates are known or presumed to take usual values:

- bond lengths, angles- take values from high-resolution crystal structures.
- aromatic amino acid sidechain regions - assumed rigid.
- nucleic acid base regions - assumed rigid.
- non-interfacial regions of protein and nucleic acid complexes (component structures may be known- only interface needs to be determined)

Can we take advantage of this knowledge (find the minima more efficiently)?

- can take larger MD timesteps (without high freq bond stretching)
- configuration space to search is smaller:
  $N_{\text{torsion angles}} \sim 1/3N_{\text{Cartesian coordinates}}$
- don't have to worry about messing up known coordinates.

Hierarchical Refinement of the Enzyme II/ HPr complex



active degrees of freedom are displayed in yellow.

MD in internal coordinates is nontrivial

Consider Newton's equation:

$$F = Ma$$

for MD, we need a, the acceleration in internal coordinates, given forces F.
Problems:

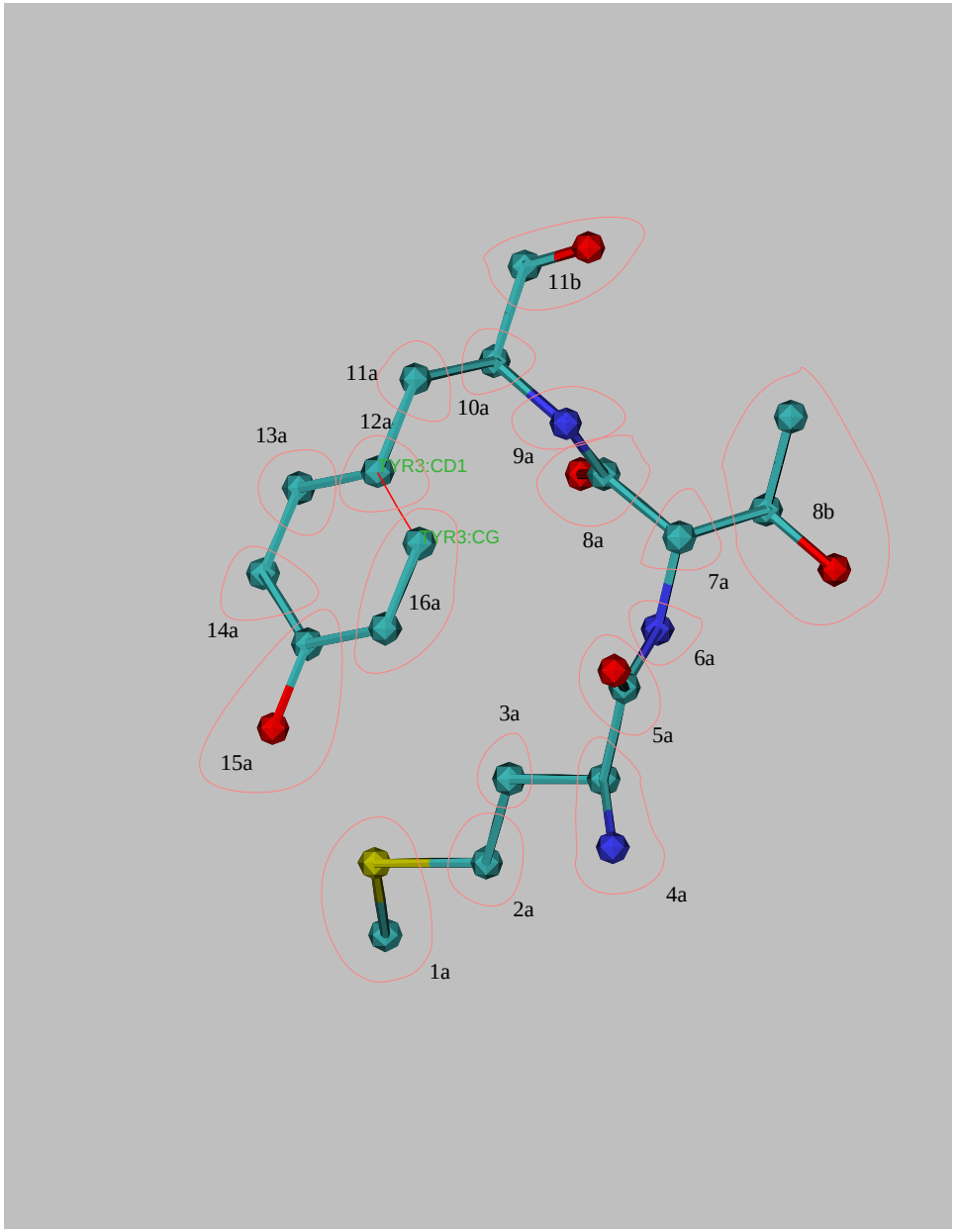
- express forces in internal coordinates
- solve the equation for a.

In Cartesian coordinates a is (vector of) atomic accelerations. M is diagonal.

In internal coordinates M is full and varies as a function of time: solving for a scales as $N^3_{\text{internal coordinates}}$.

Solution: comes to us from the robotics community. Involves clever solution of Newton's equation: The molecule is decomposed into a tree structure, a is solved for by iterating from trunk to branches, and backwards.

Tree Structure of a Molecule



atoms are placed in rigid bodies, fixed with respect to each other.

between the rigid bodies are "hinges" which allow appropriate motion

rings and other closed loops are broken- replaced with a bond.

Topology Setup

torsion angle dynamics with fixed region:

```
from ivm import IVM
integrator = IVM()                                  #create an IVM object
integrator.fix( AtomSel("resid 100:120") )          # these atoms are fixed in space
integrator.fix( AtomSel("resid 130:140") )          # fix relative to each other,
                                                    # but translate, rotate in space

import protocol
protocol.torsionTopology(integrator)                # group rigid sidechain regions
                                                    # break proline rings
                                                    # group and setup all remaining
                                                    #  degrees of freedom for
                                                    #  torsion angle dynamics
```

RDC topology setup - for tensor atoms

tensor axis should rotate only - not translate.

only single dof of D_a and Rh parameter atoms is significant.

```
from varTensorTools import topologySetup
topologySetup(integrator,listOfVarTensors)    #call before protocol.torsionTopology()
```

IVM Implementation details:

other coordinates also possible: e.g. mixing Cartesian, rigid body and torsion angle motions.

convenient features:

- variable-size timestep algorithm
- will also perform minimization
- facility to constrain bonds which cause loops in tree.

full example script in `eginput/gb1_rdc/refine.py` of the Xplor-NIH distribution.

dynamics with variable timestep

```
import protocol
bathTemp=2000
protocol.initDynamics(ivm=integrator,      #note: keyword arguments
                      bathTemp=bathTemp,
                      finalTime=1,         # use variable timestep
                      printInterval=10,    # print info every ten steps
                      potList=pots)

integrator.run()          #perform dynamics
```

script skeleton: high-level helper classes

```
simWorld.setRandomSeed( 785 )

coolParams=[]
# set up pot terms in potList
# initialize coolParams for annealing protocol

from ivm import IVM
dyn = IVM()

from simulationTools import AnnealIVM
coolLoop=AnnealIVM(dyn,...)      #create simulated annealing object

def calcOneStructure( structData ):
    # [ randomize velocities ]
    # [ perform high temp dynamics ]
    dyn.run()
    # [ cooling loop ]
    coolLoop.run()
    # [ final minimization ]
    dyn.run()
    structData.writeStructure(potList)  #write out pdb record to file
                                        # with energies, rmsd's in headers
                                        # a separate .viols file also written

from simulationTools import StructureLoop
StructureLoop(numStructures=100,
              pdbTemplate='SCRIPT_STRUCTURE.sa',
              structLoopAction=calcOneStructure).run()
```

High-Level Helper Classes

AnnealIVM: perform simulated annealing

```
from simulationTools import AnnealIVM
anneal= AnnealIVM(initTemp =3000,       #high initial temperature
                  finalTemp=25,         #final temperature
                  tempStep =25,         # temperature increment
                  ivm=dyn,              # ivm object used for molecular dynamics
                  rampedParams = coolParams) #list of energy parameters to scale

anneal.run()  # acutally perform simulated annealing
```

force constants of some terms are geometrically scaled during refinement:

$$k_{\mathrm{NOE}} = \gamma^n k^{(0)}_{\mathrm{NOE}}$$

$\gamma^N = k^{(f)}_{\mathrm{NOE}} / k^{(0)}_{\mathrm{NOE}}$

```
from simulationTools impoer MultRamp    #multiplicatively ramped parameter
coolParams=[]
coolParams.append( MultRamp(2,30,       #change NOE scale factor
                            "noe.setScale( VALUE )") )
```

StructureLoop: calculate multiple structures

```
from simulationTools import StructureLoop
StructureLoop(structureNums=range(10),        # calculate 10 structures
              structLoopAction=calcStructure, # calcStructure is function
              pdbTemplate=pdbTemplate)        # template for output structures

pdbTemplate = 'SCRIPT_STRUCTURE.sa'
#SCRIPT -> replaced with the name of the input script (e.g. 'anneal.py')
#STRUCTURE -> replaced with the number of the current structure
```

`StructureLoop` also helps wih analysis:

```
from simulationTools import StructureLoop, FinalParams
StructureLoop(structureNums=range(10),
              structLoopAction=calcStructure,
              pdbTemplate=outFilename,
              averageTopFraction=0.5,    # fraction of structures to use
              averageFitSel="not hydro ANI", #atoms used for fitting structures
              averagePotList=potList,    #terms to use in computation of ave. struct
              averageContext=FinalParams(rampedParams),  #force constants used
              averageFilename="ave.pdb", #output filename
              genViolationStats=True,    # generate a .stats file with
                                         #  energy/violation/structure stats
             ).run()
```

`StructureLoop` transparently takes care of parallel structure calculation.

parallel computation of multiple structures

computation of multiple structures with different initial velocities and/or coordinates: gives idea of structure precision, convergence of calculation.

```
xplor -parallel -machines <machine file>
```
or, on a Scyld cluster
```
xplor -scyld <number of CPUs>
```

convenient Xplor-NIH parallelization

- spawns multiple versions of xplor on multiple machines via ssh or rsh.
- structure and log files collected in the current local directory.
- robust to crashing compute nodes, crashing XPLOR runs, and the presence of dead nodes

requirements:

- ability to login to remote nodes via ssh or rsh, without password
- shared filesystem which looks the same to each node
- fully populated /bin and /usr/bin directories.

following environment variables set: `XPLOR_NUM_PROCESSES`, `XPLOR_PROCESS`

Using Biowulf

how to get a Biowulf account:
`http://biowulf.nih.gov/user_guide.html#account`

on Biowulf, compute jobs are managed using the PBS queuing system:
`http://biowulf.nih.gov/user_guide.html#q`

submit jobs using `qsub`:

```
qsub -l nodes=4 xplor.pbs
```

note that each node has two CPUs.

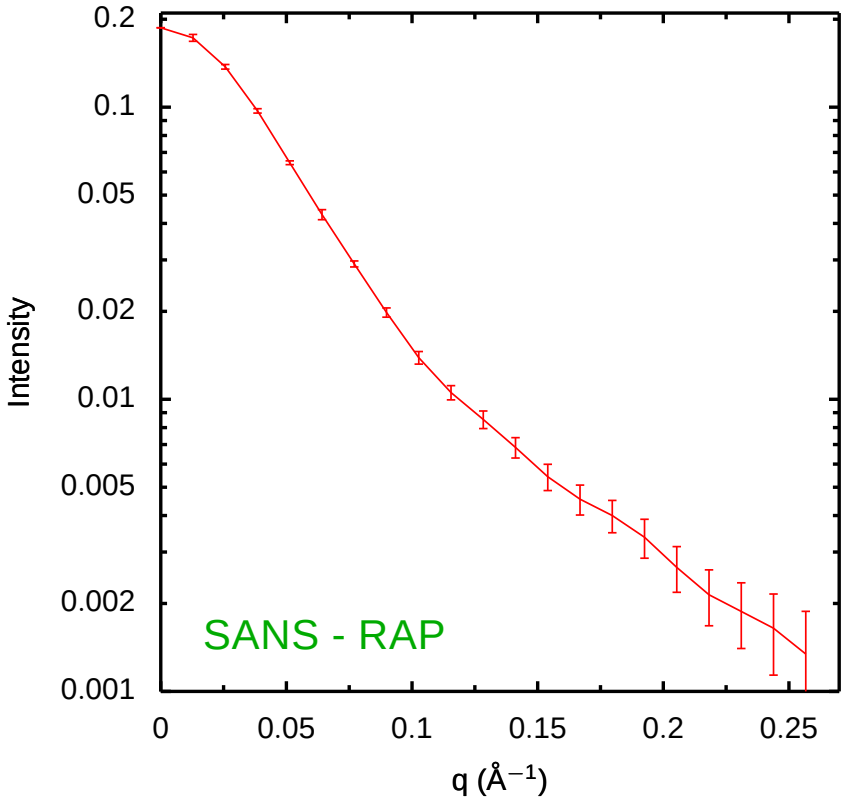
example Biowulf PBS script:
`http://nmr.cit.nih.gov/xplor-nih/nih/xplor.pbs`

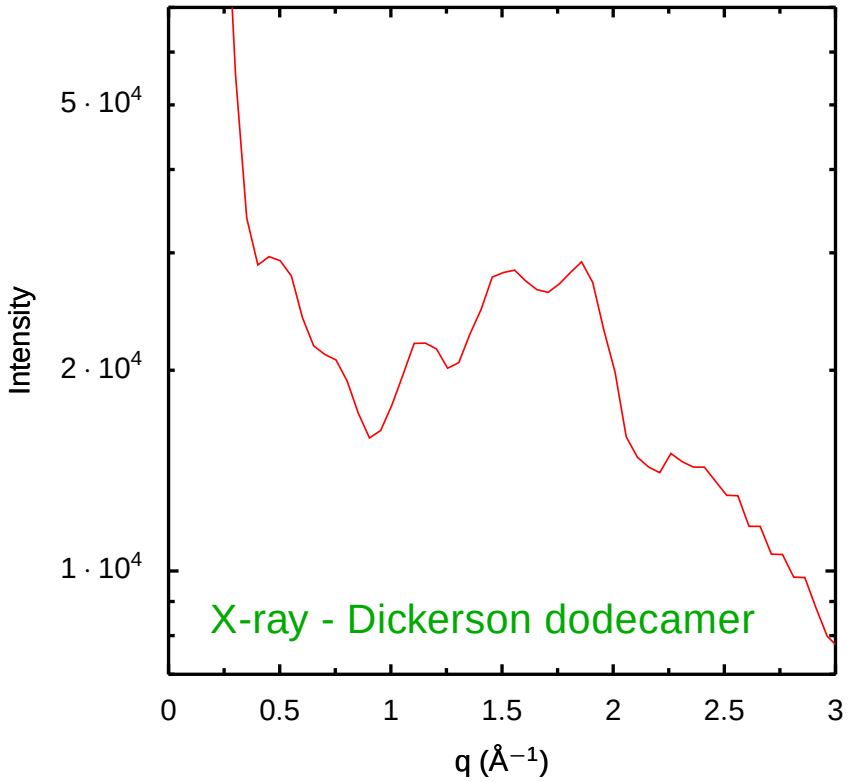
Solution Scattering Intensity

types of experiments:

- Neutron scattering (SANS)
- small-angle X-ray scattering (SAXS)
- wide (or large) angle X-ray scattering (WAXS or LAXS)

example spectra:





Calculating Scattering Intensity

Sum of point-source scatterers

$$A(\mathbf{q}) = \sum_j f_j^{\text{eff}}(q) e^{i\mathbf{q}\cdot\mathbf{r}_j},$$

sum over all atoms.

scattering vector amplitude: $q = 4\pi \sin(\theta)/\lambda$
$\theta = 0$ is the forward scattering direction

effective atomic scattering amplitude: $f_j^{\text{eff}}(q) = f_j(q) - \rho_s g_j(q)$
$f_j(q)$: vacuum atomic scattering amplitude
$\rho_s g_j(q)$: contribution from excluded solvent
->neglect boundary layer contribution<-

Difference between neutron and X-ray calculation: different $f_i^{\text{eff}}(q)$

Measured intensity

$$I(q) = \langle |A(\mathbf{q})|^2 \rangle_\Omega$$

$\langle \cdot \rangle_\Omega$: average over solid angle

Closed form solution: the Debye formula:

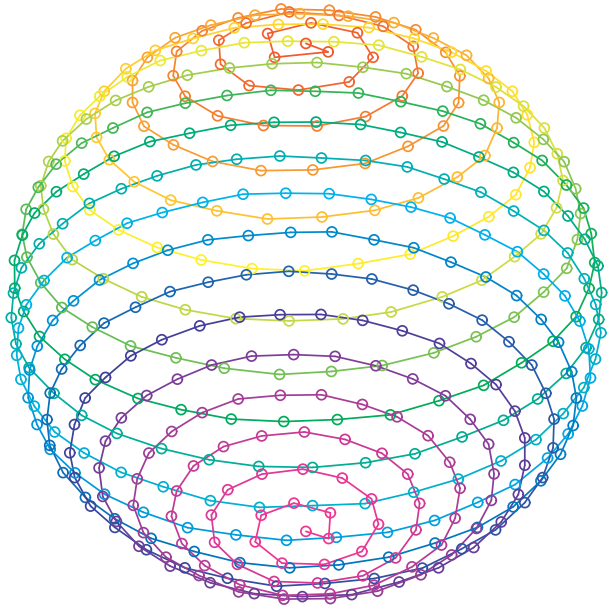
$$I(q) = \sum_{i,j} f_i^{\text{eff}}(q) f_j^{\text{eff}}(q) \text{sinc}(q r_{ij}),$$

sum is over all pairs of atoms. Expensive!

Scattering Intensity Approximations

Instead, compute $A(\mathbf{q})$ on a sphere and integrate over solid angle numerically.

Points are selected quasi-uniformly on the sphere using the Spiral algorithm:



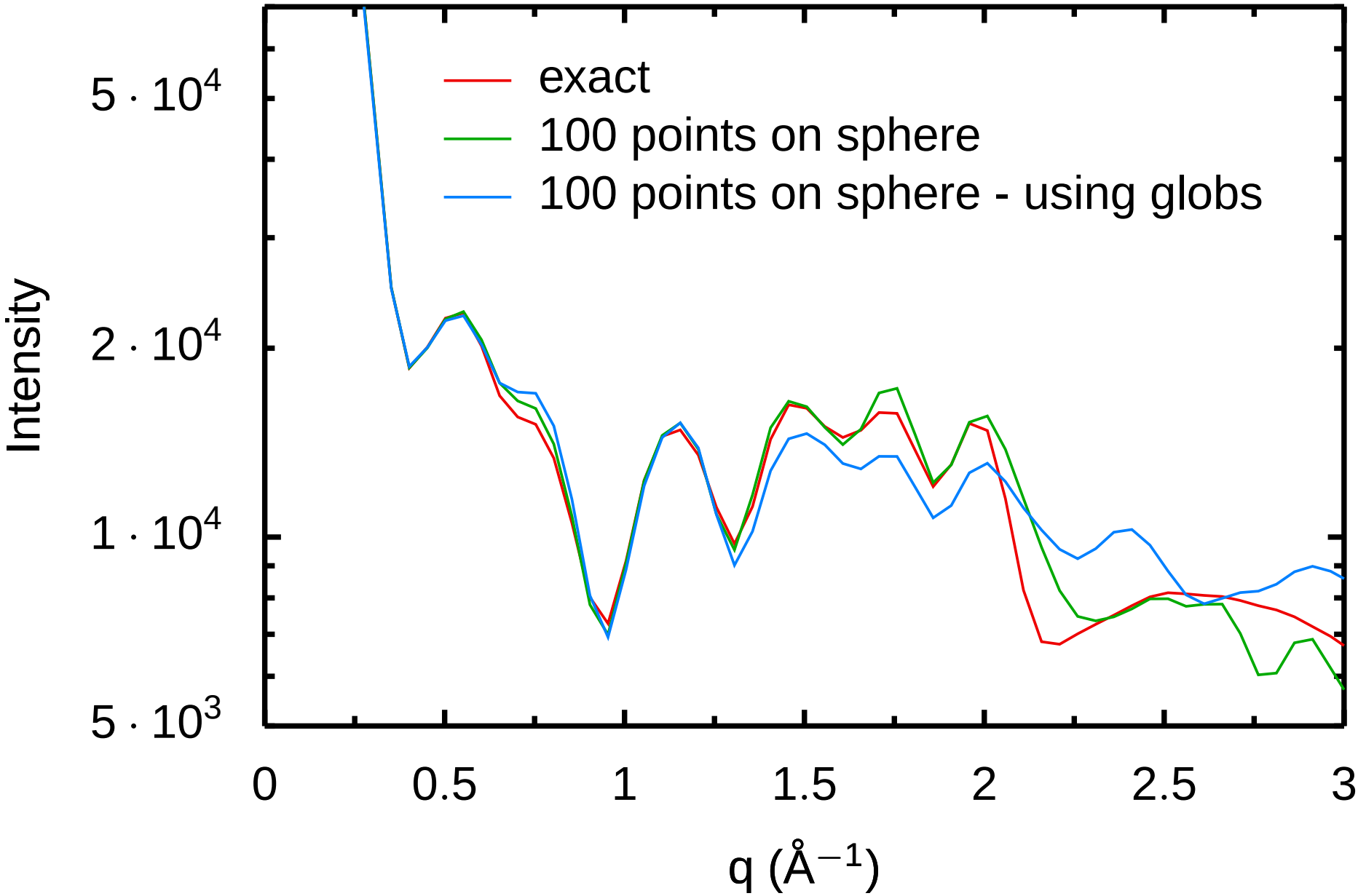
Additionally, combine atoms in "globs":

$$f_{\text{glob}}(q) = [\sum_{i,j} f_i^{\text{eff}}(q) f_j^{\text{eff}}(q) \text{sinc}(q r_{ij})]^{1/2},$$

Correct globbing, numerical integration errors with a multiplicative q-dependent correction factor c_{glob}:

$$I(q) = c_{\text{glob}}(q) I_{\text{glob}}(q),$$

Calculated intensity for DNA scattering: numerical and globbing approximations:



Refinement against solution scattering data

Refinement target function

$$E_{\text{scat}} = w_{\text{scat}} \sum_j \omega_j (I(q_j) - I^{\text{obs}}(q_j))^2,$$

w_{scat}, ω_j: weight factors
Typically set ω_j to inverse square of error in $I^{\text{obs}}(q_j)$

$I(q)$ typically normalized to $I(0)$.
Efficient computation of $I(q)$ requires uniform spacing in q.

SANS:

```
from sansPotTools import create_SANSPot, useGlobs
sans = create_SANSPot('sans',"resid 1:323","sans.data",
                      preweighted=False,fractionD2O=1)
sans.setNumAngles(240)
sansPotTools.useGlobs(sans)
rampedParams.append( StaticRamp("sans.calcGlobCorrect()") )
rampedParams.append( MultRamp(1.,100., "sans.setScale( VALUE )") )
```

X-Ray:

```
from solnXRayPotTools import create_solnXRayPot, useGlobs
xray = create_solnXRayPot('xray61',"not hydro","xray.dat")
xray.setNormalizeIndex(14)

useGlobs(xray)
xray.setNumAngles(100)

#corrects I(q) to the true Debye result
rampedParams.append( StaticRamp("xray.calcGlobCorrect('n2')") )
```

Refinement against an ensemble

```
esim = EnsembleSimulation('ensemble',3)  #creates a 3-membered ensemble
```

creates two extra copies of the current atom positions, velocities, *etc*.

Ensemble members don't interact, except with explicit potential terms.

Energy terms:
AvePot- average over the ensemble with no intra-ensemble interactions.

```
from avePot import AvePot
aveBond=AvePot(XplorPot,'bond') # ensemble averaged bond energy
```

aveBond's energy is $\langle E_{\mathrm{BOND}} \rangle_e$ averaged over the ensemble.

Refinement against an ensemble

most NMR observables must be averaged appropriately- AvePot is not appropriate- it only averages ensemble energies.

For example, the appropriate RDC value is $\langle D^{\mathsf{AB}} \rangle_e$ averaged over the ensemble. The resulting energy is then $E(\langle D^{\mathsf{AB}} \rangle_e)$.

Energy terms which are ensemble aware: rdcPot, csaPot, noePot, jCoupPot, solnScatPot, potList.

Additional potential terms: RAPPot, ShapePot - restrains atom positions within an ensemble - so members don't drift too far apart.

Example: restrain the positions of C_α atoms to be the same in all members of the ensemble.

```
from posRMSDPotTools import RAPPot
rap = RAPPot("ncs","name CA") # create term
rap.setScale( 100.0 )
rap.setPotType( "square" )    # harmonic potential has a flat region
rap.setTol( 0.3 )             # 1/2-width of flat region
```

Can also refine against bond-vector order parameter for ensemble of size N_e, with unit vector u_i along the appropriate bond vector in ensemble member i

$$S^2 = \frac{1}{2N_e^2} \sum_{ij} (3\cos(u_i \cdot u_j)^2 - 1)$$

[can use data from e.g. relaxation experiments.]

```
from orderPot import OrderPot
orderPot = OrderPot("s2_nh",open("nh_s2.tbl").read())
```

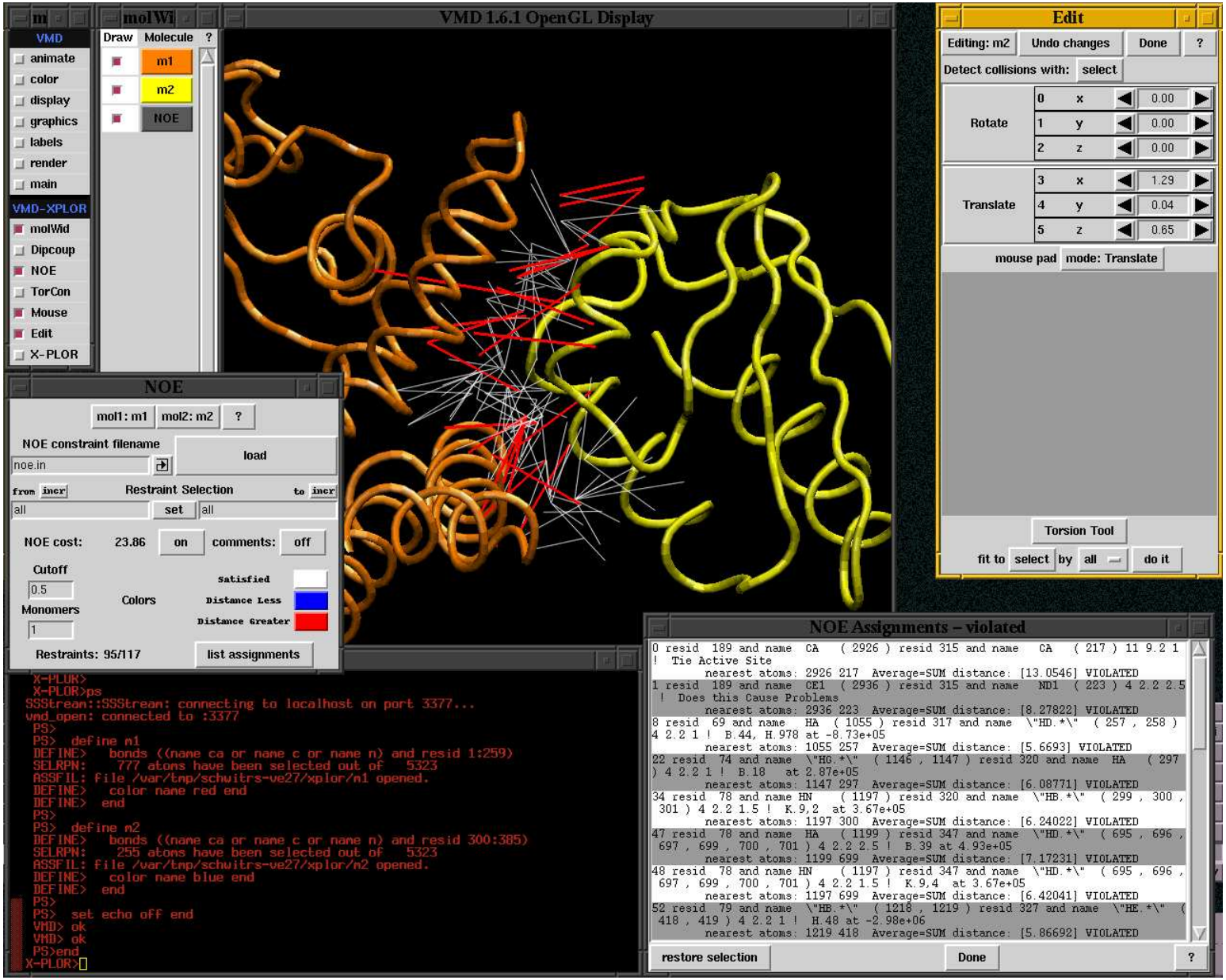
and crystallographic temperature factor for atom j in terms of q_{ij}, it's position in ensemble i, and it's esemble-averaged value q_j

$$B_j = 8\pi^2/N_e \sum_i |q_{ij} - q_j|^2$$

```
from posRMSDPotTools import create_BFactorPot
bFactor = PotList("bFactor")
```

Ensemble Feature: ensemble calculations can be parallelized by specifying the `-num_threads` option to the `xplor` command.

VMD interface



vmd-xplor screenshot

Use VMD-XPLOR to

- visualize molecular structures
- visualize restraint info
- manually edit structures

command-line invocation of separate Xplor-NIH and VMD-XPLOR jobs:

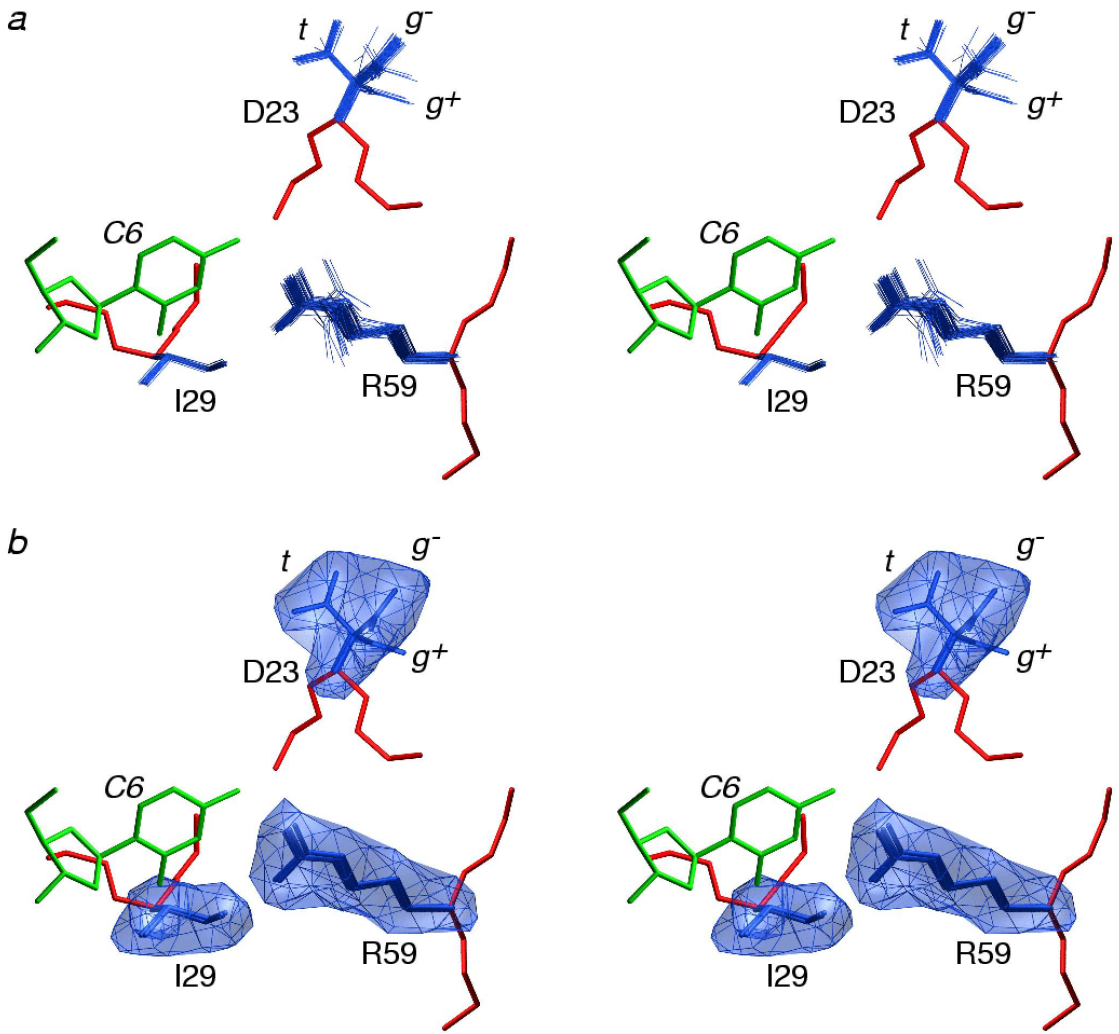
```
% vmd-xplor -port 3359 -noxplor
% xplor -port 3359 -py
```

Xplor-NIH snippet to draw bonds between backbone atoms, and labels:

```
import vmdInter

vmd = VMDInter()
x = vmd.makeObj("x")
x.bonds( AtomSel("name ca or name c or name n") )
label = vmd.makeObj("label")
label.labels( AtomSel("name ca") )
```

Graphical Representation of ensembles



intelligently convert ensemble of structures into a probability distribution.

Convenience Scripts

pdb2psf - generate a psf from a PDB file. Convenient when working from the PDB database.

```
% pdb2psf 1gb1.pdb
```
creates 1gb1.psf. Please send us pdb files which fail.

seq2psf - generate a psf file from primary sequence.

```
% seq2psf -segname PROT -startresid 300 -protein protG.seq
```
creates protG.psf with segid PROT starting with residue id 300.

tang.py - collect and average protein torsion angle values.

```
% eginput/protG/tang.py -psf=[psf file] [pdb files] >average.info
```

Putting it together: a full script

Full script for refining protein G from a random extended chain, using NOEs, RDCs, Jcoup data.

`http://nmr.cit.nih.gov/xplor-nih/doc/current/python/anneal.py.html`

Also available in the Xplor-NIH distribution in as eginput/protG/anneal.py

Where to go for help

online:

`http://nmr.cit.nih.gov/xplor-nih/`	-	home page
xplor-nih@nmr.cit.nih.gov	-	mailing list
`http://nmr.cit.nih.gov/xplor-nih/faq.html`	-	FAQ
`http://nmr.cit.nih.gov/xplor-nih/doc/current/`	-	current Documentation, including XPLOR manual

subdirectories within the xplor distribution:

eginputs	-	newer complete example scripts
tutorial	-	respository of older XPLOR scripts
helplib	-	help files
helplib/faq	-	frequently asked questions

Python:

M. Lutz and D. Ascher, "Learning Python, 2nd Edition" (O'Reilly, 2004); `http://python.org`

TCL:

J.K. Ousterhout "TCL and the TK Toolkit" (Addison Wesley, 1994);
`http://www.tcl.tk`

Please complain! and suggest!