*Ab Initio* Nanoreactor: User Guide

Table of Contents

I. Introduction 3

II. Getting Started 4

Installation 4

Installation Dependencies 4

III. Nanoreactor MD Simulations 6

IV. Reaction Extraction and Visualization 6

V. Reaction Refinement 7

VI. Work Queue 11

VII. Appendices 11

A. Known Bugs 11

B. References 11

# I. Introduction

The “*ab initio* nanoreactor” is a tool used to discover new molecules and mechanisms without preordained reaction coordinates or elementary steps. The nanoreactor method consists of three separate parts: first, a highly accelerated first-principles molecular dynamics (MD) simulation to generate chemical reaction events; second, an automated analysis of the MD trajectory to isolate reactions and their participants; third, refinement of each reaction to obtain a reaction pathway from stable reactants to products through the true transition state, complete with the accompanying energetic information.

This document first introduces the nanoreactor-type MD simulations, and briefly describes how these can be executed using TeraChem.1 It then explains the use of the scripts and libraries to analyze the reactive MD simulations.

Helpful references include:

1. Wang, L.-P. et al. Discovering chemistry with an ab initio nanoreactor. Nat Chem 6, 1044–1048 (2014). This article provides an overview of the method and two illustrative cases.
2. TeraChem’s UserGuideDev.doc, which includes a section describing the options for MD boundary conditions. This is helpful in designing MD nanoreactor simulations.

# II. Getting Started

## Installation

To install the package, run "*python setup.py install*". Make sure all dependencies are installed (below) - it's more challenging than installing the package itself.

## Installation Dependencies

1. Python 2.7, numpy, scipy, and networkx are required, while the sklearn package is recommended but not required. These dependencies are satisfied if the Python distribution is obtained from Enthought Canopy or Anaconda.
2. The Cooperative Computing Tools (http://ccl.cse.nd.edu/software/) is recommended for rolling out a highly parallel energy refinement calculation of many pathways. Using the Work Queue library, hundreds to thousands of Q-Chem calculations may be run in parallel across any combination of available computing resources. Ensure that the python bindings are built (the python libraries need to be identified during configuration for this to be successful) and the python work\_queue modules are subsequently properly installed.
3. Q-Chem 4.2 is required for running the energy refinement calculations. MPI and OpenMP parallelism are both required, because the stability analysis only works with MPI and the other components work with OpenMP. IMPORTANT NOTE: Old commercial versions of Q-Chem are somewhat problematic for the intrinsic reaction coordinate part of the calculation, because the user has no way to set the IRC initial direction. Be sure to use the latest version.

The following are required for drawing the summary PDF at the end of energy refinement:

1. Gnuplot for drawing the energy diagram. Make sure the SVG terminal is supported.
   1. Gnuplot depends on the wxWidget libraries, which are built on the fire head node, but *not* the compute nodes. These libraries should therefore be copied from /usr/lib64 on the head node to a directory within your home directory that will be available from a compute node.
2. lxml for parsing XML files. Comes with Anaconda and Entought Canopy. If installing from scratch, you need to install libxml2 and libxslt first.
3. Openbabel (version from GitHub newer than Sept. 26,2014). Used for generating SVG images of molecules. As of September 26, 2014, the latest release contained bugs, so please check out the code from GitHub (<https://github.com/openbabel/openbabel>) and build from source. Make sure to build with Python bindings enabled (cmake -DPYTHON\_BINDINGS=ON …).
   1. Building Openbabel requires Eigen, which is a pain, but it comes as an Ubuntu package. Alternatively, eigen is available from <http://eigen.tuxfamily.org>. It doesn’t need to be compiled or installed. Just unzip it and specify its location when configuring **cmake** for openbabel using -DEIGEN2\_INCLUDE\_DIR=*wherever*.
   2. To build the python bindings, you must first have installed swig. Swig is itself dependent on pcre- I have found that the best way to build pcre is to download the pcre tarball into the swig directory, and use the Tools/pcre-build.sh script to build it, so that pcre will automatically be detected by the swig configure script.
   3. I found it easiest to build openbabel without the python bindings, and then afterwards build the python bindings separately (can read the README.rst file in the openbabel/scripts/python directory for instructions).
4. rsvg-convert for converting SVG to PDF. SVG is a nice file format for editing vector graphics, but it may look different when viewed on different machines. This is part of the GNOME project and comes as a Ubuntu package. To build librsvg from source, you first need to build the following dependencies (in this listed order), if you don’t already have them: libffi, glib, libcroco, gobject-introspection, gdk-pixbuf, pixman, cairo, harfbuzz, freetype, fontconfig, pango). Helpful links and installation instructions can be found here: <http://wiki.dreamhost.com/Installing_librsvg>.
   1. glib had some problems compiling because of automake. After running *./configure*, the the version for automake is set to automake-1.13, which is not loaded on fire. Problem is fixed if first run *autoreconf* to update configure file before *./configure*.
   2. gdk-pixbuf had an additional dependency on libtiff, libjpeg, which both then had to be installed first. libjpeg had to be compiled with –fPIC included in $CFLAGS in order for gdk-pixbuf to compile.
   3. Configured fontconfig with --enable-libxml2

# III. Nanoreactor MD Simulations

Nanoreactor MD simulations are typically run using TeraChem. The TeraChem manual has a helpful section on MD with oscillating boundary conditions, describing the four different options for boundary shapes. Sample input parameters for the calculations can be found there. The key factors that characterize nanoreactor simulations are:

1. An oscillating boundary potential that acts as a piston to increase molecular collisions.
2. MD with a Langevin thermostat to regulate temperature at a relatively high temperature.
3. Low-quality basis set and level of theory (HF/DFT) to accelerate MD.
4. Large number of reactant molecules (~50-100).
5. Long trajectory run-times.

The nanoreactor package includes a helpful script (*qtcmd.py*) for submitting long continuous runs on runtime restricted SGE job queues (e.g., Stanford’s fire cluster), found in the *bin.fire* directory.

# IV. Reaction Extraction and Visualization

To analyze a simulation trajectory, run "*LearnReactions.py traj.xyz*". Use the "*-h*" argument to get help. It will generate *reaction\_123.xyz* files that contain your chemical reactions, as well as *bonds.dat, color.dat, charge.dat* and *spin.dat* used to highlight your reactive MD trajectory.

To view the highlighted trajectory, you need to install VMD and preferably be using a 3D-accelerated machine. Make sure *reactions.vmd* is in the same folder and run using: *'vmd -e reactions.vmd –args reaction\_123.xyz'*.

# V. Reaction Refinement

Once a reaction event is identified and isolated by *LearnReactions.py*, one can refine that reaction’s dynamics trajectory to produce the actual reaction pathway. To convert a nanoreactor "reaction event" into a minimum energy path, run "*Refine.py reaction.xyz*". *Refine.py* can take as input either *\*.xyz* file names, directories containing *\*.xyz* files (that start with ‘*reaction\_*’), or files containing lists of *.xyz* files, directories, or other lists. It will not go recursively into directories.

*Refine.py* will start a workflow which (1) optimizes the geometries of subsampled frames, (2) find frame pairs that contain chemically distinct species and construct a pathway connecting the energy basins, (3) smooth the pathway in internal coordinates, (4) perform string method + transition state + intrinsic reaction coordinate calculations to locate the minimum energy path, and (5) perform frequency calculations on the reactant, product, and transition state to obtain reaction free energies. Finally it will summarize the reaction as a PDF at the end, although note that the energetics reflected in the PDF include contributions from the *electronic energies only*. Because there are two different methods of reaction path determination (the freezing string and growing string methods), there will be two reaction pathways determined, each with its respective values for ΔG and ΔG‡. The default behavior of the *Refine.py* script is to use DFT with the B3LYP functional and the *6-31g(d)* and *6-31+g(d,p)* basis sets (there are two tiers of the transition state search, each of which use one of the basis sets).

Some calculations do not have well defined minima of their products or reactants (for example, in radical calculations), and consequently, the initial optimization of the frames of the trajectory fails to determine unique reactant and product states. For this reason, we have also implemented a calculation of the ΔG of the reaction associated with the sum of the isolated fragments, done prior to reaction pathway refinement. To do this, the calculation first identifies the individual molecules or fragments in each frame of the trajectory. It then optimizes the isolated fragments, summing their energies to obtain the total energy associated with that reactant or product state, and calculating a ΔG for each reaction found in the trajectory. (The sub-scripts related to this are *identify-fragments.py* and *optimize-fragments.py.*)

Below is a loose description of the flow of the refinement code, identifying the directory tree and file outputs (in *italics*) of each process:

* Refine.py takes in list of reaction file names and lauches a Trajectory calculation for each reaction. The Trajectory calculation is the parent for the Fragment, Optimization and Pathway parts of the calculation.
* Fragment (in *fragments* directory)
  + Fragment identification
    - Single point electronic structure calculation on a subsample of frames from the trajectory, whose interval can be defined by the -*–subsample* argument passed to Refine.py. Each calculation is done in subdirectories named by their frame number.
      * *initial.\*\*.sp, initial.\*\*.stb, initial.\*\*.sp*
    - Build topology of system based on bonding predicted by electronic structure to divide the calculation into fragments. Calculate “bondfactor”, a measure of how clearly bonds are defined.
    - Using Mulliken charges and spins, define each fragment’s charge and spin. Check for consistency and validity.
    - Print initial geometries of each fragment to *initial.sub\_\*.xyz*, comment line has formula, charge, and spin
    - Print fragment formulas, bondfactor, and validity to *fragmentid.txt*
  + Fragment Optimization
    - Identify frames with unique sets of fragments and the maximum bondfactors associated with those fragment sets.
    - For identified frames (in an */opt* subdirectory of the frame’s subdirectory), optimize each fragment individually, sum to get total energy for frame.
    - Print all optimized geometries to *fragmentopt.xyz*
    - Print total energy to *fragmentopt.nrg*. If calculation is invalid (molecule converts during optimization), append “invalid” to this file.
  + Calculate Δ*H* and Δ*G*
    - Read *fragmentopt.nrg* for all optimized frames, and calculate Δ*H* and Δ*G* associated with each reaction.
* Optimization (in *structures* directory)
  + Stability analysis and optimization of each frame to go to reactant/product minima
    - *optimize.\*\*.sp, optimize.\*\*.stb, optimize.\*\*.opt*
* Pathway Identification (in *pathways* directory)
  + Identify reaction pathways in trajectory with smallest interval between frames. Each pathway gets its own subdirectory in *pathways*, and is labeled by the frame numbers of the start and finish.
  + Optimize initial and final frames (in *pathways/\*\*/opt-init* and *pathways/\*\*/opt-final*)
  + Trajectory is constructed by connecting the dynamics trajectory between the two frames with the optimization trajectories on either end: *rejoined.xyz*
  + This trajectory is then evenly spaced/interpolated: *respaced.xyz*
* Freezing String (in *pathways/\*\*/FS* directory)
  + Optimizes endpoints
    - *qcopt\_prd* and *qcopt\_rct*
  + Freezing string calculation to build reaction path guess and guess of transition state
    - *qcfsm*
  + Transition state optimization with basis set 1
    - Initial HF/KS stability analysis
      * *qcts1.\*\*.sp, qcts1.\*\*.stb*
    - Frequency calculation
      * *qcts1.\*\*.freq*
    - Transition state optimization
      * *qcts1.\*\*.ts*
    - HF/KS stability analysis of optimized structure
      * *qcts1.\*\*.sp, qcts1.\*\*.stb*
  + Redo transition state optimization with a more accurate basis set
    - *qcts2.\**
  + Intrinsic reaction coordinate calculation
    - *qcirc.\*\*.rpath; qcirc.\*\*.opt* if the IRC doesn’t find endpoints right away
  + Final stability analysis, single point energy, and frequency calculations on IRC reactant, product, and transition states, to calculate zero point energies, entropies, and enthalpies for Δ*H*, Δ*H*‡, Δ*G*, and Δ*G*‡.
    - *irc\_reactant.\*, irc\_product.\*, irc\_transition.\*, deltaG.nrg*
  + Process reaction pathway information (arclength, electronic energies only) and print output in nice format
    - *irc\_spaced.xyz, irc.nrg, plot.nrg, reaction.svg, reaction.pdf*
    - *irc\_reactant.\*, irc\_product.\*, irc\_transition.\**
* Interpolation (in *pathways* directory)
  + Takes *spaced.xyz* and smoothes the trajectory by internal coordinate interpolation, to print *interpolated.xyz*
  + Equally spaces the interpolation to print *interspaced.xyz*, which will serve as the initial guess for the growing string calculation.
* Growing String (in *pathway/\*\*/GS* directory)
  + Alternates between two processes, growing string calculations and transition state calculations
  + Growing string
    - Calls *gstring.exe*
  + Transition state calculations and reaction path – in *TS* subfolder
    - Transition state optimization with basis set 1
      * Initial HF/KS stability analysis
        + *qcts1.\*\*.sp, qcts1.\*\*.stb*
      * Frequency calculation
        + *qcts1.\*\*.freq*
      * Transition state optimization
        + *qcts1.\*\*.ts*
      * HF/KS stability analysis of optimized structure
        + *qcts1.\*\*.sp, qcts1.\*\*.stb*
    - Redo transition state optimization with a more accurate basis set
      * *qcts2.\**
    - Intrinsic reaction coordinate calculation
      * *qcirc.00.rpath; qcirc.01.opt* if the IRC doesn’t find endpoints right away
    - Final stability analysis, single point energy, and frequency calculations on IRC reactant, product, and transition states, to calculate zero point energies, entropies, and enthalpies for Δ*H*, Δ*H*‡, Δ*G*, and Δ*G*‡.
      * *irc\_reactant.\*, irc\_product.\*, irc\_transition.\*, deltaG.nrg*
    - Process reaction pathway information (arclength, electronic energies only) and print output in nice format
      * *irc\_spaced.xyz, irc.nrg, plot.nrg, reaction.svg, reaction.pdf*

# VI. Work Queue

(Lee-Ping should probably write a section on running the refinement calculations using Work Queue. I haven’t been able to get it working for myself – LIB.)

# VII. Appendices

## A. Known Bugs

* The default Q-Chem parallel command (found in qchem.py) is to call a script entitled qchem42, which launches sets up and runs a Q-Chem job. This script must be located in the default path.
* The refinement code should be run with the environment variable PYTHONUNBUFFERED=True, otherwise the python stdout buffers will not flush to the logfiles, and the logfiles may remain empty (due to the files being tarred before the buffer is flushed).

## B. References

1 I.S. Ufimtsev and T.J. Martinez, J. Chem. Theory Comput. **5**, 2619 (2009).