FACULTY OF SCIENCE
UNIVERSITY OF COPENHAGEN

Bachelor Thesis in NanoScience

Kasper Primdal Lauritzen

Computational Methods for Automated Generation of Enzyme Mutants

Department of Chemistry, University of Copenhagen

Supervisors: Jan Jensen & Martin Hediger

June 1, 2012

Contents

Abstract

This report demonstrates the feasibility of easily and quickly generating mutants of a given protein, *in silico.*

Using primarily MOPAC and PyMOL it is possible to produce mutations on a specific site, generate relevant rotamers of that mutant, evaluate and discard suboptimal rotamers, and prepare the new mutation for a reactivity evaluation using an interpolation of the two end states of the enzymatic reaction.

This report also shows that when one generates these mutants from a heavily optimized enzyme structure, the CPU time needed for optimizing the mutant is drastically reduced compared to the initial optimization of the wild type.

Resume

Denne rapport demonstrerer muligheden for nemt og hurtigt at generere mutanter af et givent protein, *in silico.*

Ved primært at bruge MOPAC og PyMOL er det muligt at generere mutationer på et specifikt sted på enzymet, generere rotamere til mutanten, evaluere og afvise suboptimale rotamere, og klargøre den nye mutant til en evaluation af reaktiviteten ved bruge af en interpolation mellem start og slut tilstanden i den enzymatiske reaktion.

Det vises også at når man genererer mutanter udfra en stærkt optimeret enzym struktur, reducerer man den krævede CPU tid til mutant optimeringen kraftigt, sammenlignet med optimeringen af wild typen.

1 Introduction

The enzyme examined in this study, *Candida Antarctica* Lipase B (CalB) [1], is a esterase, but examined for its activity as an amidase. In wild type (WT), CalB has a low amidase activity and to increase this activity, the enzyme is mutated.

Predicting which mutations will produce the desired effect can be, at best, very hard if not impossible. In principle, it is possible do mutations at every position of the enzyme backbone, and mutate into all amino acids (for single mutations in CalB, that would mean a total of around $317 \cdot 19 \approx 6000$ possible mutations).

For this reason it is unfeasible to carefully study each mutation. Describing an enzyme with a high level of theory to find the transition state can take weeks [2]. Instead, it is possible to quickly screen mutants using lower levels of theory, and identify possibly promising candidates for further study.

This study presents an *in silico* method for an automated approach to this problem. In addition to the computer time required, the time needed for manual steps must be considered. When examining several hundreds of mutants, this manual work can severely bottleneck the process. Therefore the process of mutating must be as automated as possible.

After some initial manual work preparing the wild type enzyme for mutation, each mutant can be optimized and prepared for a barrier height estimation (to approximate the reaction activity), in roughly 24 hours. After this, the actual barrier estimate can be made by 10 cores, in another 24 hours [3].

All input files and scripts used in this report will be available at `www.github.com/KPLauritzen/auto-enzyme-mutants/`.

2 Computational Details

In this report all quantum mechanical calculation were carried out using MOPAC2009 [4]. The `MOZYME` keyword was used to do calculations due to the large system size (CalB contains more than 4000 atoms, including hydrogens). This allows MOPAC to use *localized molecular orbitals* (LMOs), and allows the calculation time to scale linearly with system size [5].

Mutations were done using PyMOL [6] and its library of amino acids. PyMOL also provides a library of rotamers for the proteinogenic amino acids (with one or more conformational degrees of freedom) and this was exploited as well. In addition, PyMOL can do non-QM optimizations, which we used to do quick-and-dirty local optimization of mutants.

An effort was made to use OpenBabel [7] as an alternative to both MOPAC's energy calculations and PyMOL's local optimizations, but this idea was discarded due to OpenBabel's inability to handle system-sizes as large as was needed in this case.

To compare the activity of a mutant with the WT and different mutants with each other, the respective barrier heights are compared. The barrier heights can be estimated, given the structure of the enzyme substrate complex (**ES**) and the transition state (**TS**). The reaction shown in figure 1 is studied.

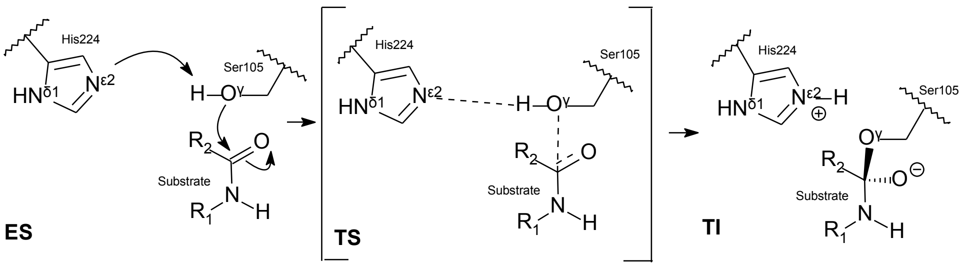

Figure 1 – The first step of the enzymatic reaction studied. Nucleophilic attack by O^γ of Ser105 and proton abstraction by $N^{\epsilon 2}$ of His224 [8]. R_1: $-CH_2C_6H_5$, R_2: $-CH_2CH_3$.

A structure of the **TS** is not available, so an approximation of the structure was made, using a linear interpolation method.

Given the initial and final state of reaction, and assuming that the atoms move linearly between these states, the reaction coordinate was divided into 10 evenly spaced steps, where an intermediate structure was generated by moving each atom 1/10th of the distance between the initial and final state. An approximation of the transition state can then be made as the structure along the reaction coordinate with the highest energy, and the barrier height is then approximated as the energy difference between the initial state and the transition state. The hypothesis is that a lower reaction barrier estimate translates into a higher activity for the reaction.

Calculations were run on two different high performance, distributed systems. One is “sunray” provided by the Department of Chemistry, University of Copenhagen, and the other is “steno”, provided by the Danish Center for Scientific Computing.

3 Results

The main results from this study are the methodology used for preparing structures for mutation, the constraints needed to do quantum mechanical calculations on a full enzyme and finally the time requirements for mutation optimizations and barrier height calculations.

3.1 Structure Preparation

3.1.1 Obtaining the initial structures

The enzyme structures were obtained from the Protein Data Bank (PDB) [9]. The crystal structure used, `1LBS` [10], was resolved at a resolution of 2.6 Å. For this study carthesian coordinates of the atoms in the enzyme are required, which are available from the PDB. There are some steps required before the PDB file can be used for further study. These are: removing non-crystal waters, protonation of the structure and placing the substrate. Figure 2 outlines the sequence of steps needed. They will be described below (and the implementation will be described in appendix A.2).

Due to the optimization scheme used, the **ES** structure was derived from the optimized **TI** structure.

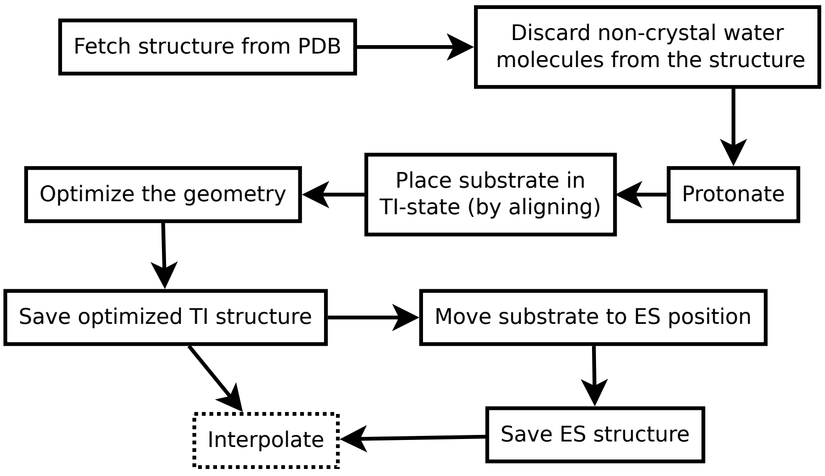
Fetch structure from PDB
Discard non-crystal water molecules from the structure
Protonate
Place substrate in TI-state (by aligning)
Optimize the geometry
Save optimized TI structure
Move substrate to ES position
Save ES structure
Interpolate

Figure 2 – Flowchart of the steps required to prepare an initial crystal structure for interpolation.

3.1.1.1 Generation of the Tetrahedral Intermediate The set of atomic coordinates obtained from PDB contains several coordinates for water molecules, placed both on the surface of the enzyme, and inside as crystal waters. However, a continuum solvent model (implemented as *conductor-like screening model* (COSMO) [11]), was used, therefore the non-crystal waters were discarded. Using COSMO, a solvent can be simulated, not by explicitly placing each solvent molecule, but by placing a surface around the enzyme, with the dielectric constant of the solvent (the dielectric constant for water is 78.4 at 25°C). This omission of surface waters reduces the total number of atoms in the model and therefore the total number of calculations needed for any given method.

When retrieving the enzyme structure from PDB, no hydrogens are present in the model. The protonation state of the enzyme was corrected using PyMOLs protonation function. As an example of the importance of accurate protonation, examine the histidine involved in the enzymatic reaction (see figure 1 for details on the reaction). The $N^{\epsilon 2}$ involved in the proton transfer has a pK_a value of 6.9 [12], and should be deprotonated in the **ES** state.

As an enzyme can not be crystallized with the substrate at the active site (the enzyme would catalyze the reaction) an inhibitor is typically placed in the substrates place. In this study N-phenylmethylacetamide was used as substrate, see figure 3 for the structure. To place the tetrahedral substrate (ie. the substrate in the **TI**-state) in a reasonable position in the enzyme, the central atoms of the substrate and the inhibitor are aligned with each other. The assumption is that if the central atoms are aligned, they will bond in a similar way. As seen in figure 4, the carbonyl carbon (C-1) of the substrate is aligned to the phosphor of the inhibitor, the nitrogen of the substrate aligns to carbon next to phosphor on the inhibitor, and the substrate carbonyl oxygen aligns to the lone-pair oxygen of the inhibitor.

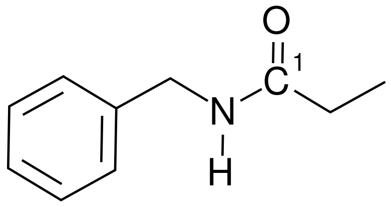

Figure 3 – N-phenylmethylacetamide. The substrate used in this study.

The rest of the substrate was optimized locally. Alternatives to this approach would be, aligning more of the substrate with the inhibitor or an exhaustive search through many different conformations.

Finally, the geometry of the enzyme should be optimized. This should be done as long as the CPU budget can afford, as a good optimization saves time when doing mutations (see figure 5).

3.1.1.2 Generating the Enzyme Substrate Complex Using the previously obtained **TI** structure to generate the **ES**-state, the substrate should be moved into a reasonable position. This is a vague term, and many possibilities exists, with no clear way to discern which is the better one. For this

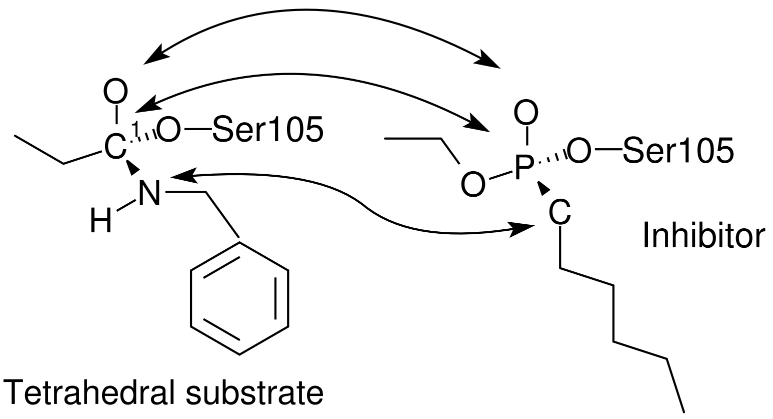

Figure 4 – Pairwise alignment of the tetrahedral substrate (left) and the inhibitor (right) present in the PDB structure.

study, the best approach is the one requiring the least amount of effort. In this approach the substrate was moved about one bond length (around 1.5 Å) along the axis of the bond between the carbonyl carbon (C-1) and the serine oxygen (O^{γ}).

Alternatively, one could use molecular dynamics (MD) to generate the **ES** and **TI** states, possibly increasing the accuracy of the results, but requiring optimizing two structures instead of one, doubling the CPU time for the initial optimizations.

In addition the proton at $N^{\epsilon 2}$ in His224 was moved towards O^{γ} in Ser105.

3.1.2 Mutating

A mutation is here defined as exchanging one amino acid in the enzyme for another amino acid. The mutant is then the new amino acid, and the mutant enzyme is the full new structure of the enzyme.

In this study, when given a set of possible rotamers of a mutant, each of them were first locally optimized and then the energy of the full mutant was calculated. The rotamer with the lowest energy was then picked as the best conformation for the mutant.

Another possible approach could be to discard any unphysical rotamers, ie. in PyMOL the mutant side-chains might be placed at the same position as neighboring side-chains. This could save calculation time, as the energy evaluation might discard these anyway.

When the configuration of the mutant has been found, it is inserted in both **ES** and **TI** states, and these structures can be optimized, and then the effectiveness of the mutation can be evaluated, based on the reaction barrier height.

Appendix A.4 details how mutations were implemented, and appendix A.5 shows details of how the mutations were executed automatically.

3.1.3 Checklist for structure preparation

Here is provided a short list of the mistakes the author frequently made when preparing structures and doing mutations. The hope is that the reader may avoid these in the future.

- Check that enzyme is properly protonated. In the **TI** state, this means hydrogens are present at $N^{\epsilon 2}$ and $N^{\delta 1}$ in His224 and no hydrogen at the O bonding Ser105 to the substrate (O^{γ}).
- Make sure PyMOL is set to `set retain_order, 0` when sequencing the enzyme, to configure saving of PDB file to save added hydrogens near the residue they belong to.
- Run a `charges` calculation in MOPAC before running the full optimization. In the bottom of the output file charge locations are printed. Looking through these can reveal mistakes. A residue with an unexpected charge might be the results of a missing hydrogen, or a side-chain in a bad configuration.
- PyMOL scripts can be run from the terminal with both `python <script>` and `pymol -qcr <script>`. They do *not* produce the same results. Use the latter.
- Make new directories when preparing a new enzyme. Discard files with mistakes in them, or tests, before running scripts. Some scripts require output files from previous scripts to be present in the directory, and will produce bad results if these files are not correct.

3.2 Full Enzyme Quantum Mechanical Calculations

To examine the feasibility of doing quantum mechanical (QM) calculations on a full enzyme, and to examine under which sets of optimization constraints reliable results can be produced.

As input, **ES** and **TI** structures, obtained by MD simulations[1], were used. Each structure was examined with a single layer of water molecules at the surface and a vacuum calculation model, with no waters and a vacuum calculation model and with no waters and a COSMO solvent model.

For each of these six possible inputs, a combination of NDDO cutoff distances (`cutoff`) and gradient convergence criteria for optimization termination (`gnorm`) were tried. The `cutoff` was selected between 3 and 15 Å, and `gnorm` between 5 and 20 kcal/mol/Å.

When not allowing use of LMOs, the geometry optimizations failed, but when allowing use of LMOs, the geometry optimizations converged with final formation energies shown in tables 1 and 2. The final energies were

[1]MD structures obtained by Allan Svendsen, Novozymes

found after doing a reorthonormalization on the optimizied geometry using a PM6 method. The full set of energies and calculation times are available in appendix B.

The formation energy reached by using the strictest set of optimization constraints (`cutoff=15 gnorm=5`) are taken as the canonical value. This value is compared to energies reached by using less strict constraints. If these values were similar within a small margin of error, one could argue for the use of less strict constraints to speed up calculations. But, the results gained from less strict constraints were too far from the canonical value, so in further calculations `cutoff=15` and `gnorm=5` were used.

`gnorm \ cutoff`	3	6	9	12	15
5	−88060.2	−101434	−101568	−101578	−101633
10	−88060.2	−101346	−101427	−101518	−101540
15	−88060.2	−101204	−101446	−101436	−101475
20	−88060.2	−101164	−101397	−101378	−101408

Table 1 – Energy of the wildtype in the **ES** state, in kJ/mol, as a function of MOPAC keywords `gnorm` and `cutoff`. Calculations are done using `COSMO`.

`gnorm \ cutoff`	3	6	9	12	15
5	−87217.1	−101101	−101189	−101226	−101245
10	−87217.1	−101011	−101176	−101146	−101205
15	−87217.1	−100886	−101138	−101139	−101106
20	−87217.1	−100749	−101062	−101049	−101064

Table 2 – Energy of the wildtype in the **TI** state, in kJ/mol, as a function of MOPAC keywords `gnorm` and `cutoff`. Calculations are done using `COSMO`.

3.3 Time Requirements

As seen in table 3 and 4, an initial structure optimization using a strict configuration (`cutoff=15 gnorm=5`) can require several days to complete.

When doing mutations, generated from an already optimized structure, the initial geometry optimization time reduces drastically compared to the WT. See figure 5. Optimizing the geometry of a full mutant takes roughly 24 hours.

Results are pending for time requirements for an interpolation of a full mutant with crystal waters included.

gnorm \ cutoff	3	6	9	12	15
5	3.11	2.83	6.49	4.73	5.98
10	2.84	1.22	2.26	3.76	4.39
15	3.12	1.05	1.80	1.96	2.71
20	3.54	0.963	1.46	1.71	2.11

Table 3 – Time, in days, for optimization of the wild type **ES** state, as a function of MOPAC keywords `gnorm` and `cutoff`. Calculations are done using `COSMO`.

gnorm \ cutoff	3	6	9	12	15
5	2.24	2.70	3.88	3.26	4.02
10	2.35	1.76	1.86	2.61	3.97
15	2.19	1.03	1.48	1.85	1.87
20	2.23	0.75	1.18	1.34	1.74

Table 4 – Time, in days, for the wild type **TI** state to finish optimizing, as a function of MOPAC keywords `gnorm` and `cutoff`. Calculations are done using `COSMO`

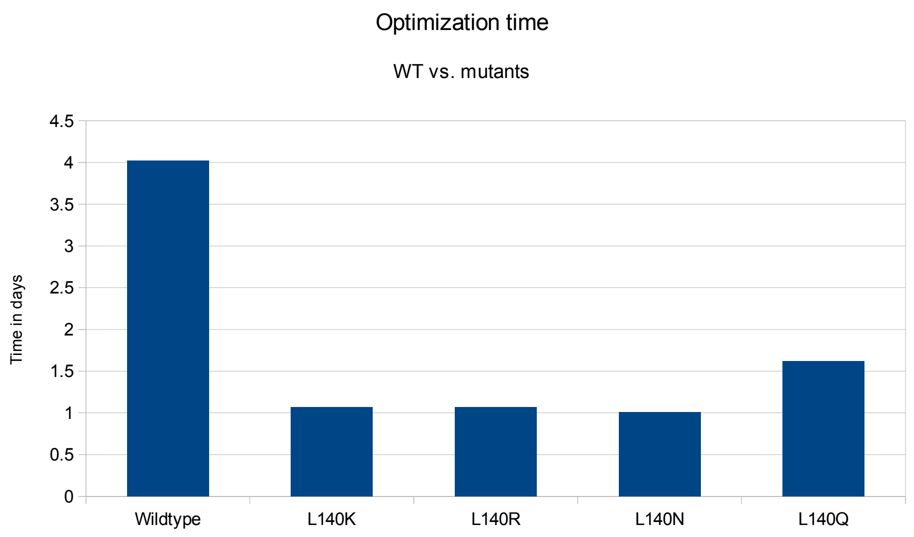

Figure 5 – The optimization time of mutants compared to the wild type they are based on. Settings: `gnorm`=5, `cutoff`=15 and `eps`=78.4

4 Conclusion

This study presents an automated computational approach to doing mutations in an enzyme, in preparation for screening the mutant reactivity.

An reproducible method for preparing the *Candida Antarctica* lipase B (CalB) structure for mutation and interpolation has been presented.

It was shown, that it is possible to do calculations on full enzymes using localized molecular orbitals (`MOZYME`), and a continuum solvent model (`COSMO`). Optimizing the crystal structure of CalB took several days (depending on optimization settings), and allowed for mutants based on that structure to be optimized in roughly 24 hours.

Automating the mutant creation process allows for creation of hundreds or thousands of mutants with minimal human interaction in the process.

5 Outlook

There are several choices made that could be explored further:

Substrate conformation The substrate is placed manually in place of the inhibitor. The placement necessarily done with certain choices regarding the conformation of the substrate, and a different orientation could potentially mean several hydrogen bonds made or broken.

Smarter choices for rotamers To decrease the number of calculations needed to select the best rotamer for a mutation, the root-mean-square distance (RMSD) between each rotamer could be checked after PyMOL has finished its initial, local optimization. If the RMSD is very short between two rotamers, it is probable that they are in the same local minimum, and effectively equal.

Another possibility is to check for impossible conformations of rotamers at the initial placement. If the side-chain of a rotamer overlaps with the side-chain of a neighboring amino acid, PyMOL will create artificial bonds between them, and be unable to change the conformation into a physically reasonable position.

Automate the interpolation As it stands, the interpolation of each full mutant still requires several manual steps to complete, each step consisting of running a script on the correct set of files. If it is possible to link the termination of a MOPAC calculation with the execution of the next script, it should be possible to fully automate the process from initial enzyme structures to barriers for mutants.

Introducing more than one mutation at a time In this study, only single mutations were considered.

A possible future step in the development could be to examine the possibilities of multi-fold mutations. When introducing two mutants at a large distance from each other, so they have no direct interactions, no additional steps are needed.

However, it is not trivial to do double mutations at neighboring sites. At the moment this requires manual work, when placing each mutant to make sure partial charges see each other, and other similar interaction behave as expected. An automated approach to this is needed.

6 Acknowledgements

The author acknowledges Martin Hediger for many hours of help in all areas of the project. Further acknowledgments to Jan Jensen for letting me do this project, Allan Svendsen (Novozymes) for MD structures, Luca De Vico for support and discussions and both the Department of Chemistry, University of Copenhagen and the Danish Center for Scientific Computing for use of their high-performance computers.

References

[1] Allan Svendsen. Lipase protein engineering. *Biochimica et Biophysica Acta (BBA) - Protein Structure and Molecular Enzymology*, 1543(2):223 – 238, 2000. Protein engineering of enzymes.

[2] Johannes C. Hermann, Juliette Pradon, Jeremy N. Harvey, and Adrian J. Mulholland. High level qm/mm modeling of the formation of the tetrahedral intermediate in the acylation of wild type and k73a mutant tem-1 class a β-lactamase. *The Journal of Physical Chemistry A*, 113(43):11984–11994, 2009. PMID: 19791786.

[3] M. R. Hediger, L. De Vico, A. Svendsen, W. Besenmatter, and J. H. Jensen. A Computational Methodology to Screen Activities of Enzyme Variants. *ArXiv e-prints*, 2012.

[4] James J. P. Stewart. Mopac: A semiempirical molecular orbital program. *Journal of Computer-Aided Molecular Design*, 4:1–103, 1990. 10.1007/BF00128336.

[5] James Stewart. Optimization of parameters for semiempirical methods v: Modification of nddo approximations and application to 70 elements. *Journal of Molecular Modeling*, 13:1173–1213, 2007. 10.1007/s00894-007-0233-4.

[6] LLC Schrödinger. The pymol molecular graphics system. Version 1.4.1.

[7] Noel O'Boyle, Michael Banck, Craig James, Chris Morley, Tim Vandermeersch, and Geoffrey Hutchison. Open babel: An open chemical toolbox. *Journal of Cheminformatics*, 3(1):33, 2011.

[8] Lizbeth Hedstrom. Serine protease mechanism and specificity. *Chemical Reviews*, 102(12):4501–4524, 2002.

[9] Helen M. Berman, John Westbrook, Zukang Feng, Gary Gilliland, T. N. Bhat, Helge Weissig, Ilya N. Shindyalov, and Philip E. Bourne. The protein data bank. *Nucleic Acids Research*, 28(1):235–242, 2000.

[10] Jonas Uppenberg, Mogens Trier Hansen, Shamkant Patkar, and T.Alwyn Jones. The sequence, crystal structure determination and refinement of two crystal forms of lipase b from candida antarctica. *Structure*, 2(4):293 – 308, 1994.

[11] A. Klamt and G. Schuurmann. Cosmo: a new approach to dielectric screening in solvents with explicit expressions for the screening energy and its gradient. *J. Chem. Soc., Perkin Trans. 2*, pages 799–805, 1993.

[12] Hui Li, Andrew D. Robertson, and Jan H. Jensen. Very fast empirical prediction and rationalization of protein pka values. *Proteins: Structure, Function, and Bioinformatics*, 61(4):704–721, 2005.

List of Abbreviations

`1LBS`	PDB ID for *Candida Antarctica* lipase B
CalB	*Candida Antarctica* lipase B
COSMO	Conductor-like Screening Model
ES	Enzyme Substrate Complex
LMO	Localized Molecular Orbital
MD	Molecular Dynamics
MOPAC	Molecular Orbital PACkage
NDDO	Neglect of diatomic differential overlap
PDB	Protein Data Bank
QM	Quantum Mechanics
RMSD	Root-mean-square distance

TI	Tetrahedral Intermediate
TS	Transition state

Appendices

A Implementation

A.1 Full Enzyme Quantum Mechanical Calculations

Here follows a more detailed description of the steps described in 3.2. See also figure 6 for a flowchart outlining the process.

As input, structures for the **ES** and **TI** states were used, obtained from MD simulations done by Novozymes. Included in these structures were a large amount of water molecules, to simulate the enzyme inside a solvent.

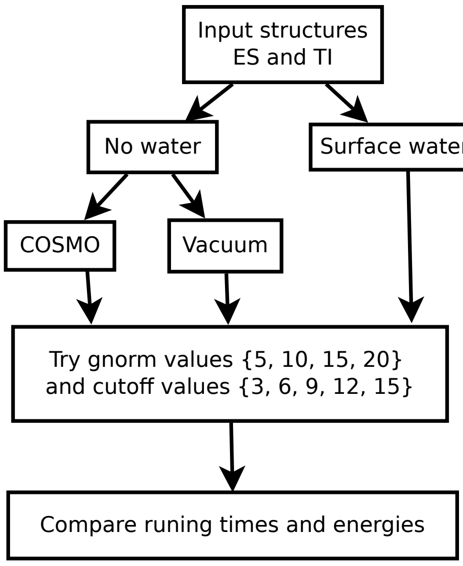
Input structures ES and TI
No water
Surface water
COSMO
Vacuum
Try gnorm values {5, 10, 15, 20} and cutoff values {3, 6, 9, 12, 15}
Compare runing times and energies

Figure 6 – Flowchart of the steps involved in doing full enzyme QM calculations

The molecular structures, as defined in the PDB files, were loaded into PyMOL. Then one of two operations was done. Either all water molecules were removed from the model (using PyMOLs functionality, `remove waters`) and saved as `noH2O`, or all water molecules except a single layer at the surface of the enzyme were removed and saved as `wH2O`. This was achieved in PyMOL by selecting the enzyme and using the function `modify -> expand 4A`. This selects everything within 4 Å of the enzyme. By inverting the selection and deleting it, what is left is roughly a single layer of water molecules surrounding the enzyme.

For the `noH2O` structures, another division was done. In one set of calculation, a continuous solvent model, COSMO, simulating the dielectric constant of water was used (MOPAC keyword `eps=78.4`) and another set where no solvent was used, thus simulating the enzyme in vacuum.

At this point 6 input files are available. Each of these were modified with the following keywords: `mozyme charge=$CHARGE gnorm=$GNORM cutoff=$CUTOFF`, where `$CHARGE` was calculated in advance, `$GNORM` was picked from the list {5, 10, 15, 20} kcal/mol/Å and `$CUTOFF` was picked from the list {3, 6, 9, 12, 15} Å .

After the calculations had finished running, the optimized structure was re-orthonormalized to get a more accurate final energy. This was done by copying the output coordinates from the `arc`-file to a `mop`-file and adding the keywords `charge=$CHARGE cutoff=$CUTOFF mozyme 1scf`.

A.2 Generation of Initial Structures

As described in section 3.1.1, a reliable method of generating input files containing the atomic coordinates of the enzyme is needed. See figure 2 for an overview of the steps required.

The first step is to acquire the crystal structure. The PDB ID for CalB is `1LBS`, so the PyMOL command to download the structure file is `PyMOL> fetch 1LBS`. Note that this fetches several chains of the enzyme. Only one is required, in this example all chains except for `A` were removed, using `PyMOL> select ///A`, inverting the selection and using `PyMOL> remove sele`.

Next step, removing non-crystal water. Listing 1 displays the PyMOL commands to remove any waters with few contacts to the protein and any waters further than 3.5 Å from the surface of the protein.

```
PyMOL> remove solvent beyond 3.5 of polymer
PyMOL> set dot_solvent
PyMOL> get_area solvent, load_b=1
PyMOL> remove solvent and b > 20
```

Listing 1 – PyMOL commands required to discard non-crystal water molecules from an enzyme.

To protonate the enzyme, PyMOL is first configured using `PyMOL> set retain_order, 0` to allow the coordinates of the added hydrogens to be saved near the residues they belong to. Next, protons are added using: `PyMOL> h_add`. OpenBabel could be used as well, but due to the coordinates of the added protons being saved at the bottom of the output file (which scripts needed for the interpolation can not parse), PyMOL is used. PyMOL does not protonate exactly as is needed. Depending on the surrounding structure the hydrogen on `HIS'224/NE2` or `HIS'224/ND1` might be missing. To fix this, another proton from His224 is selected in PyMOL, copied to a new object. This object is then moved to a position bonding to the vacant nitrogen, and saved as a separate file (as `new-proton.pdb`). The full structure is saved as well (as `3-wt.pdb`). This contents of `new-`

`proton.pdb` is copied and pasted into `3-wt.pdb`, next to the other lines containing coordinates for His224. The file is saved, and reloaded into PyMOL.

Next, placing the substrate. The atomic coordinates of the has to be loaded into PyMOL. For completeness the PDB-file containing these coordinates (for the **TI**-state) are included in appendix C.

As described in section 3.1.1.1 and illustrated in figure 4, central atoms of the inhibitor and the substrate should be aligned. Using PyMOLs wizard `Pair Fitting` the following atoms are aligned: `LIG'500/C` to `HEE'900/P`, `LIG'500/O` to `HEE'900/O1P` and `LIG'500/N` to `HEE'900/C1`.

After the alignment has finished, the inhibitor should be removed. Then the position of the tail end of the substrate can be optimized using commands shown in listing 2. The optimized structure of the substrate should be saved to a separate file. To ensure compatibility with future scripts, some conventions are followed and the substrate file should be edited to reflect this: The substrate residue name is `LIG` and is given the residue sequence number 500.

```
PyMOL> cmd.protect('(not mutant)')
PyMOL> cmd.sculpt_activate('enzyme')
PyMOL> cmd.sculpt_iterate('enzyme', cycles=5000)
PyMOL> cmd.sculpt_deactivate('enzyme')
PyMOL> cmd.deprotect()
```

Listing 2 – PyMOL commands used to optimize the geometry of a selection named `mutant`, in a PyMOL object named `enzyme`. The number of `cycles` can be raised to achieve better results. 5000 cycles completes in a few minutes on a average CPU.

The contents of the file containing the atomic coordinates of the substrate are pasted into the end of the file containing the enzyme coordinates. In this file every line specifying the coordinates of water molecules should be edited, so that the residue names are `HOH` and the residue sequence numbers are 550.

This file is now submitted to MOPAC for geometry optimization with the the keywords: `charge=$CHARGE cutoff=15 gnorm=5 eps=78.4 mozyme pdbout nores T=8D`.

The output of this optimization is the **TI**-state, ready for interpolation. The **ES**-state are still needed. To prepare this state, a number of steps is required. See figure 7 for an overview in flowchart form.

For the sake of clarity, the following shorthand notation will be used: The file containing the optimized coordinates of the **TI**-structure is called `3-wt.pdb`, the wanted output file, containing coordinates of the **ES**-state is called `1-wt.pdb`. The intermediate files with coordinates for the substrate and the transferred proton are called `subst.pdb` and `proton.pdb` respectively.

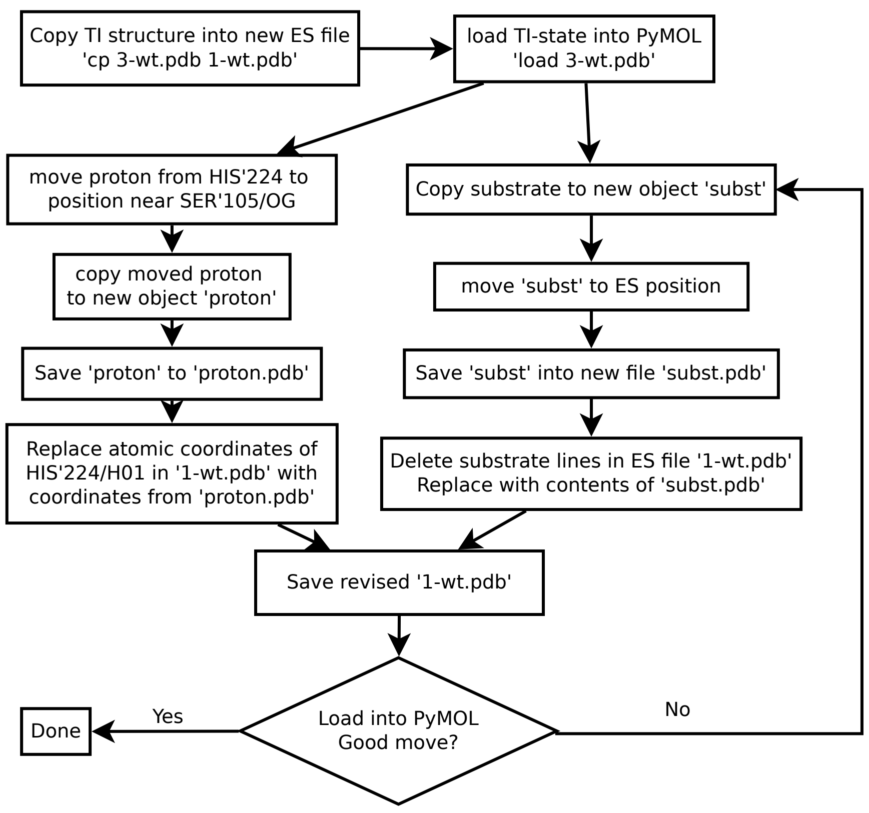
Copy TI structure into new ES file
'cp 3-wt.pdb 1-wt.pdb'
load TI-state into PyMOL
'load 3-wt.pdb'
move proton from HIS'224 to position near SER'105/OG
copy moved proton to new object 'proton'
Save 'proton' to 'proton.pdb'
Replace atomic coordinates of HIS'224/H01 in '1-wt.pdb' with coordinates from 'proton.pdb'
Copy substrate to new object 'subst'
move 'subst' to ES position
Save 'subst' into new file 'subst.pdb'
Delete substrate lines in ES file '1-wt.pdb'
Replace with contents of 'subst.pdb'
Save revised '1-wt.pdb'
Load into PyMOL
Good move?
Yes
Done
No

Figure 7 – Flowchart outlining the process of generating the **ES** structure from an optimized **TI** structure.

First, `3-wt.pdb` is copied and renamed to `1-wt.pdb`. Then, `3-wt.pdb` is loaded into PyMOL. Then the proton bound to `HIS'224/NE2` is copied to a new object. This object is moved towards `SER'105/OG`. In this new position, the object is saved as `proton.pdb`. Next, the substrate is selected and copied to a new object. This object is then moved into a reasonable position for the **ES**-state (see section 3.1.1.2). After the move the object is saved as `subst.pdb`.

Then, `1-wt.pdb` is opened in a text editor. The line containing the coordinates for the proton involved in the proton transfer is located. The coordinates are replaced with the coordinates from `proton.pdb`. Similarly, the lines containing the substrate coordinates in `1-wt.pdb` are located. These coordinates are replaced by the coordinates from `subst.pdb`.

The edited file are saved as `1-wt.pdb`, and loaded into PyMOL. Here the new positions of the substrate and the proton can be evaluated. If they are not good enough, the process can be repeated.

After this process, both the initial and final state of the enzymatic reaction are ready, and an interpolation can be done.

A.3 Interpolation

To do an interpolation between two enzyme structures, the files containing the coordinates of each "end point" (here, the **ES** and **TI**-states) need to be *in sync*. This means that the atom represented by the first line of the **ES** file is also represented by the first line of the **TI** file, same thing for the second line, and so on.

First point is to run `intcha.py <ES-state> <TI-state> wt` (appendix D.6) to create 10 interpolation frames and prepare each of them for submission to MOPAC for determination of closed-shell overall charge.

On the output of these files run `cha2scf.sh` (appendix D.7) which creates new input files for MOPAC. After MOPAC has finished running these files the output now has a format in which constraints can be specified on specific atoms. This is done using `scf2opt.sh` (appendix D.8). The atoms that have to be constrained are specified in the file.

Further constraints can optionally be added using `fix.py` (appendix D.9). A list of residues to constrain are kept in this script. By adding some residues to this list, large rearrangements can be avoided during the optimization.

Now use MOPAC optimize each interpolation frame. After it has finished running, run `opt2spe.sh` (appendix D.10) on each output .arc-file to prepare it for a final single point energy calculation.

When those calculations are done, the energy of each frame can be extracted and a barrier can be created using `profiles.py` (appendix D.11).

A.4 Mutation

The method for introducing mutations into an already optimized structure is outlined here.

Using the most strictly optimized structures available (the **ES** and **TI**-states produced in section 3.1.1 in this case), and assuming naming conventions are kept (substrate named `LIG` in the PDB file, with residue number 500, and waters named `HOH` with number 550), the first step is to load the **ES** structure into PyMOL.

Then the structure is separated by residue into separate files using the code shown in listing 3. The calling sequence from PyMOL is `run seq_files.py`, and then `seq 1`. The **TI** structure should be sequenced as well, by loading it into PyMOL and running `seq 3`.

When this is done, mutant generation can begin. PyMOLs *mutagenesis wizard* was used for most of the heavy lifting. PyMOL has a library of rotamers available for each mutant. When iterating through rotamers, the code shown in listing 2 was used to locally optimize each one. This is shown in listing 4 where the optimization function is called with `localSculpt`. Listing 4 shows the central part of the `vsc.py` script, with the calling se-

```python
def seq(state, selection="name ca or resn hoh or resn lig"):
    cmd.select("prot", selection)
    while cmd.pop("_tmp", "prot"):
        cmd.iterate("_tmp", "stored.x=(resn,resv)")

        # Special case 1: Waters.
        if stored.x[0] == 'HOH':
            filename = 'seq-x%s-%s.pdb' % (stored.x[1], state)
        # Special case 2: Substrate.
        elif stored.x[0] == 'LIG':
            filename = 'seq-x%s-%s.pdb' % (stored.x[1], state)
        # Other: protein back-bone.
        else:
            filename = 'seq-%s%d-%s.pdb' \
              %(one_letter[stored.x[0]].lower(), stored.x[1], state)
        cmd.save(filename, "byres _tmp")
    cmd.delete('_tmp prot')
cmd.extend('seq', seq)
```

Listing 3 – `seq_files.py`: Python code for use in PyMOL. Makes a selection from an enzyme, one residue at a time, and saves each residue into its own file.

quence: `PyMOL> run vsc.py`, and then `PyMOL> frag 3`. This saves each of the optimized rotamer fragments (derived from the **TI**-state) into its own file.

Another script, `assemble-rotamers.py` (shown in full in appendix D.3), is available to assemble each rotamer into a full enzyme, and submit each of these new files to MOPAC for a single point energy calculation.

The first part of `assemble-rotamers.py` is to define the backbone chain. If, as is the case in this study, the full enzyme is used, then this can be done automatically, otherwise it should be defined manually in the script. Listing 5 shows the commands required to define the backbone for the full enzyme.

Next, a file is created, `seq.sh`, wherein the commands for the actual assembly are defined. For each mutant, the script loops over every rotamer of that mutant, and copies the coordinates of each residue into a new file (see the central loop in listing 6).

A MOPAC energy calculation is run on each assembled file. The best rotamer is found using the script `evaluate-rotamers.sh`, shown in appendix D.2.

A.5 Fully Automated Mutation

The goal of this project is ultimately to be able to automate the process of generating and evaluating a large number of mutations in an enzyme. This is achieved in 2 scripts:

- One that generates the mutants and their rotamers, and sets them up

```python
for site in mutations.keys():
    variants = mutations[site]
    # Run over all variants.
    for variant in variants:
        cmd.load(obj)
        cmd.do('wizard mutagenesis')
        cmd.do('refresh_wizard')
        cmd.get_wizard().set_mode(variant)
        cmd.get_wizard().do_select(site + '/')
        # Get the number of available rotamers at that site
        nRots = getRots(site, variant)
        cmd.rewind()
            for i in range(1, nRots + 1):
                cmd.get_wizard().do_select("(" + site + "/)")
                cmd.frame(i)
                cmd.get_wizard().apply()
                localSculpt(obj, site)

                # Protonate the N and the C here
                # Save the mutant fragment here

        cmd.do('delete all')
        cmd.set_wizard('done)
```

Listing 4 – Central part of `vsc.py`. The `frag` method, iterating though previously defined mutations, and all available rotamer for each mutation. A local optimization is run, and the mutated fragment is saved to a separate file.

for an energy calculations, `create-mutant-fragments.py` (shown in appendix D.1).

- One that parse the formation energy of each rotamer, and picks the one with lowest energy for further calculations, `evaluate-rotamers.sh` (shown in appendix D.2).

The first script is a modified combination of `assemble-rotamers.py` and `vsc.py` used in the previous sections, but the full script are added in the appendix for completeness.

The **ES** and **TI** states are required as input files. They should be *synchronized*, that is, each line in one file (containing the coordinates of one atom) should correspond to the same atom on the same line in the other file. As before, the ligand should have the residue name `LIG` with residue number 500, and all waters should be named `HOH` with number 550. These input files should of course be as optimized as possible, to reduce subsequent optimization time for mutants (See figure 5).

Then the automutation script `create-mutant-fragments.py` should be run. But, because there are some bugs when running a PyMOL instance from python, it should be run directly with PyMOL. That can be accom-

```python
# Generate backbone list
chain = []
for i in range(1,551):
    for file in os.listdir("."):
        if fnmatch.fnmatch(file, 'seq-?%i-%s.pdb'%(i,state)):
            # Remove the 6 last chars (eg "-3.pdb")
            # And append to sequence of amino acids
            chain.append(file[0:-6])
```

Listing 5 – Part of `assemble-rotamers.py`. Defines the sequence of the backbone chain of a full enzyme. Requires the output files of `seq_files.py` to be present to function correctly.

plished from the shell like so: `pymol -qcr create-mutant-fragments.py`

The first thing the script does is to divide the input file into separate files for each residue.

Then the mutant fragments are created, using the `frag` method (see listing 4) This step can take a long time, depending on how many mutations are done how many rotamers each mutant have, and for how many cycles are allowed for each rotamer optimization.

PyMOL executes the commands given one at a time, but it does not wait for a return code before continuing to the next command. This is an issue if the output of a previous slow-running command is needed later in the script, as it is in this case. This can be addressed by inserting the command `py-mol.cmd.sync(10000.0 * len(mutations))` after a slow-running method. This tells PyMOL to wait for a return code (for a maximum of 10000 seconds per mutation site, roughly 3 hours) from the previous command before running the next command.

Then the enzyme is assembled again with one mutant at a time. This is done much like described previously in listing 6 . One change is that we use a for-loop over the number of rotamers instead of relying on a while-loop to exit when a given rotamer did not exist. Also, a command was added to submit the mutant to MOPAC.

After the rotamers have finished running, the second script (`evaluate-rotamers.sh`, see appendix D.3) is run, to extract energies for each rotamer from the MOPAC output, select the rotamer with lowest energy, and prepare that rotamer for a geometry optimization.

```
writeSeq =\
[...]
'ROT=1\n' +\
'while [ -e frag-' + variant.lower()+'-rot$ROT-?.pdb ] \n' +\
'do \n' +\
'cat /dev/null > %s-res-' % state + \
'-'.join(varlist) + '-rot$ROT.pdb\n' +\
'echo "1scf mozyme cutoff=15 \n\n" \
> %s-res-' % state + '-'.join(varlist) + '-rot$ROT.pdb \n' +\
'for i in\\\n' +\
'     {'

varTmp = variant.split('+')
varNum = [i[1:-1] for i in varTmp]
for s in chain:
   if s[5:] in varNum:
       ind=varNum.index(s[5:])
       writeSeq += 'frag-' + \
       variant.split('+')[ind].lower() + '-rot$ROT,\\\n'
   else:
        # Appending of the WT backbone.
        writeSeq += s + ',\\\n'

writeSeq += '}\n' +\
'do\n' +\
'     cat $i-%s.pdb >> %s-res-' % \
     (state, state) + '-'.join(varlist) + '-rot$ROT.pdb\n' +\
'done\n' +\
[...housekeeping...]
'let ROT=ROT+1 \n' +\
'done \n\n'
```

Listing 6 – Central part of `assemble-rotamers.py`: Python code that generates the bash-script `seq.sh`. The string `writeSeq` is eventually written to `seq.sh`, which then can be executed to assemble the full mutant.

B Time and Energy as a function of `gnorm` and `cutoff`

Tables of running times for optimizations and final energies of geometry optimizations for wild type CalB in both **ES** and **TI** states. The strictest condition placed on the optimization are at the upper right of the tables (`gnorm`=5, `cutoff`=15).

`gnorm \ cutoff`	3	6	9	12	15
5	−88060.2	−101434	−101568	−101578	−101633
10	−88060.2	−101346	−101427	−101518	−101540
15	−88060.2	−101204	−101446	−101436	−101475
20	−88060.2	−101164	−101397	−101378	−101408

Table 5 – Energy of the wild type in the **ES** state, in kJ/mol, as a function of MOPAC keywords `gnorm` and `cutoff`. Calculations are done using `COSMO`.

`gnorm \ cutoff`	3	6	9	12	15
5	3.11	2.83	6.49	4.73	5.98
10	2.84	1.22	2.26	3.76	4.39
15	3.12	1.05	1.80	1.96	2.71
20	3.54	0.96	1.46	1.71	2.11

Table 6 – Time, in days, for optimization of the wild type **ES** structure, as a function of MOPAC keywords `gnorm` and `cutoff`. Calculations are done using `COSMO`.

`gnorm \ cutoff`	3	6	9	12	15
5	−87217.1	−101101	−101189	−101226	−101245
10	−87217.1	−101011	−101176	−101146	−101205
15	−87217.1	−100886	−101138	−101139	−101106
20	−87217.1	−100749	−101062	−101049	−101064

Table 7 – Energy of the wild type in the **TI** state, in kJ/mol, as a function of MOPAC keywords `gnorm` and `cutoff`. Calculations are done using `COSMO`.

`gnorm \ cutoff`	3	6	9	12	15
5	2.24	2.70	3.88	3.26	4.02
10	2.35	1.76	1.86	2.61	3.97
15	2.19	1.03	1.48	1.85	1.87
20	2.23	0.75	1.18	1.34	1.74

Table 8 – Time, in days, for optimization of the wild type **TI** structure, as a function of MOPAC keywords `gnorm` and `cutoff`. Calculations are done using `COSMO`.

gnorm \ cutoff	3	6	9	12	15
5	−69462.1	−97556.2	−97727.8	−97782.8	−97730.6
10	−82041.9	−97278.5	−97690	−97632.7	−97693.3
15	−81274.2	−97118.4	−97390.8	−97592.1	−97573.8
20	N/A	−96874.8	−97216	−97332.9	−97201.5

Table 9 – Energy of the wildtype in the **ES** state, in kJ/mol, as a function of MOPAC keywords `gnorm` and `cutoff`. Calculations are done in vacuum. The N/A entry is caused by a corrupted file.

gnorm \ cutoff	3	6	9	12	15
5	1.84	3.00	3.75	3.94	5.16
10	1.83	1.12	2.35	3.69	5.93
15	1.82	0.76	1.29	2.25	3.27
20	3.52	0.78	1.08	2.17	2.28

Table 10 – Time for the wildtype **ES** state to finish optimizing, as a function of MOPAC keywords `gnorm` and `cutoff`. Calculations are done in vacuum.

gnorm \ cutoff	3	6	9	12	15
5	−81575.5	−97180.4	−97295.4	−97306.7	−97324.3
10	−81959.4	−96996.8	−97132.2	−97044.8	−97104.9
15	−82072.2	−96903.1	−96919.8	−96910.3	−96966.1
20	−82034	−96607.3	−96768.3	−96768.8	−96789.3

Table 11 – Energy of the wildtype in the **TI** state, in kJ/mol, as a function of MOPAC keywords `gnorm` and `cutoff`. Calculations are done in vacuum.

gnorm \ cutoff	3	6	9	12	15
5	0.82	2.43	4.39	5.69	6.87
10	0.71	1.32	2.17	2.92	3.41
15	0.77	1.04	1.42	2.21	2.55
20	0.71	0.74	1.26	1.50	1.90

Table 12 – Time, in days, for the wildtype **TI** state to finish optimizing, as a function of MOPAC keywords `gnorm` and `cutoff`. Calculations are done in vacuum.

C Tetrahedral Intermediate Substrate geometry in PDB format

```
ATOM      1  N   LIG A 500     55.926   1.966  34.888  1.00  0.00      PROT N
ATOM      2  C   LIG A 500     56.709   1.954  36.151  1.00  0.00      PROT C
ATOM      3  O   LIG A 500     56.023   1.999  37.272  1.00  0.00      PROT O
ATOM      4  C   LIG A 500     54.800   3.777  33.629  1.00  0.00      PROT C
ATOM      5  C   LIG A 500     57.876   2.939  36.070  1.00  0.00      PROT C
ATOM      6  C   LIG A 500     54.749   2.882  34.844  1.00  0.00      PROT C
ATOM      7  C   LIG A 500     54.175   5.034  33.691  1.00  0.00      PROT C
ATOM      8  C   LIG A 500     54.201   5.885  32.585  1.00  0.00      PROT C
ATOM      9  C   LIG A 500     54.855   5.503  31.409  1.00  0.00      PROT C
ATOM     10  C   LIG A 500     55.473   4.253  31.342  1.00  0.00      PROT C
ATOM     11  C   LIG A 500     55.435   3.389  32.442  1.00  0.00      PROT C
ATOM     12  H   LIG A 500     56.540   2.138  34.080  1.00  0.00      PROT H
ATOM     13  H   LIG A 500     58.697   2.601  36.716  1.00  0.00      PROT H
ATOM     14  H   LIG A 500     57.564   3.927  36.438  1.00  0.00      PROT H
ATOM     15  H   LIG A 500     58.280   3.039  35.059  1.00  0.00      PROT H
ATOM     16  H   LIG A 500     53.823   2.257  34.810  1.00  0.00      PROT H
ATOM     17  H   LIG A 500     54.665   3.498  35.770  1.00  0.00      PROT H
ATOM     18  H   LIG A 500     53.691   5.354  34.617  1.00  0.00      PROT H
ATOM     19  H   LIG A 500     53.703   6.856  32.626  1.00  0.00      PROT H
ATOM     20  H   LIG A 500     54.858   6.178  30.553  1.00  0.00      PROT H
ATOM     21  H   LIG A 500     55.980   3.942  30.429  1.00  0.00      PROT H
ATOM     22  H   LIG A 500     55.919   2.405  32.375  1.00  0.00      PROT H
```

D Scripts

D.1 create-mutant-fragments.py

```python
import __main__
__main__.pymol_argv = ['pymol','-qc'] # Pymol: quiet and no GUI
from time import sleep
import pymol
pymol.finish_launching()
from pymol import cmd
from pymol import stored
from pymol.exporting import _resn_to_aa as one_letter
import os
from os.path import splitext
import sys
import fnmatch

# Calling Sequence (from terminal)
# £ pymol -qcr create-mutant-fragments.py
# NB. There is no need to have an open PyMOL session.
# This is all run from the terminal.


###########
#PARAMETERS
###########
# The state the mutation will be done on.
state = "3"

# Filenames for initial and end-state of the reaction
obj3 = '3-wt-opt.pdb'
obj1= '1-wt-opt.pdb'

# For convience. Allows for defining mutations like: '140' : allAminoAcids
allAminoAcids = [ 'ALA', 'ARG', 'ASN', 'ASP', 'CYS', 'GLU', 'GLN',
                  'GLY', 'HIS', 'ILE', 'LEU', 'LYS', 'MET', 'PHE',
                  'PRO', 'SER', 'THR', 'TRP', 'TYR', 'VAL' ]
# Define which mutation should be done. e.g. '140': ['ARG', 'LYS']
mutations = {
'140': ['ARG', 'LYS']
}

# commandname for MOPAC submit-script
# To stop MOPAC from autostarting, set: mopacCommand = "echo"
mopacCommand = "moppdb"

# Settings mopac should be run with for rotamer evaluation
mopacSettings = "mozyme 1scf cutoff=15"
# ******************************************************************
def seq(state, selection="name ca or resn hoh or resn lig"):
    print "Generating seqs."
    cmd.select("prot", selection)
    while cmd.pop("_tmp", "prot"):
        cmd.iterate("_tmp", "stored.x=(resn,resv)")
        #print stored.x[0], stored.x[1]
        # Special case 1: Waters.
        if stored.x[0] == 'HOH':
            filename = 'seq-x%s-%s.pdb' % (stored.x[1], state)
        # Special case 2: Substrate.
        elif stored.x[0] == 'LIG':
            filename = 'seq-x%s-%s.pdb' % (stored.x[1], state)
        # Other: protein back-bone.
        else:
            filename = 'seq-%s%d-%s.pdb' \
            % (one_letter[stored.x[0]].lower(), stored.x[1], state)
        cmd.save(filename, "byres _tmp")
    cmd.delete('_tmp prot')



# **************************************************************
def setup(obj):

    # Various set up
    pwd = os.getcwd()
    #print "os.getcwd()", os.getcwd()
    cmd.do('wizard mutagenesis')
    cmd.do('refresh_wizard')

    # Save residue names and numbers.
    orig_sequence = setNames(obj)
    return pwd, orig_sequence
# ------------------------------------------------------------


# ************************************************************
# 'state=state': The first variable is the variable used within
# the scope of this function. The second variable is the one
# in the global scoped and defined at the top of the module.
def frag(state=state, obj=obj3):
    pwd, orig_sequence = setup(obj)
    stored.rotamerDict = {}
    # Add and retain hydrogens
    cmd.get_wizard().set_hyd("keep")

    # Run over all sites where to mutate
    for site in mutations.keys():

        variants = mutations[site]

        # Run over all variants.
        for variant in variants:
            cmd.load(obj)

            cmd.do('wizard mutagenesis')
            cmd.do('refresh_wizard')
            cmd.get_wizard().do_select("(%s/)" % site)
            cmd.do("cmd.get_wizard().set_mode('%s')"%variant)

            # Get the number of available rotamers at that site
            # Introduce a condition here to check if
            # rotamers are requested.
            # <<OPTION>>
            print variant, "variant"
            nRots = getRots(site, variant)
            nRots = 2
            stored.rotamerDict[str(site)+getOne(variant)] = nRots

            cmd.rewind()
            for i in range(1, nRots + 1):

                cmd.get_wizard().do_select("(" + site + "/)")
                cmd.frame(i)
                cmd.get_wizard().apply()

                # Optimize the mutated sidechain
                #<<OPTION>>
                print "Sculpting."
                localSculpt(obj, site)

                # Protonation of the N.
                cmd.do("select n%d, name n and %d/" % (int(site), int(site)))
                cmd.edit("n%d" % int(site), None, None, None, pkresi=0, pkbond=0)
                cmd.do("h_fill")

                # Protonation of the C.
                cmd.do("select c%d, name c and %d/" % (int(site), int(site)))
                cmd.edit("c%d" % int(site), None, None, None, pkresi=0, pkbond=0)
                cmd.do("h_fill")

                # Definition of saveString
                saveString  = '%s/' % pwd
                saveString += 'frag-' + getOne(orig_sequence[site]).lower() +\
                              site + getOne(variant).lower() + '-rot%i-%s.pdb, ' \
                              % (i,state) +'((%s/))' % site
                #print saveString
                cmd.do('save %s' % saveString)
            cmd.do('delete all')
            cmd.set_wizard('done')
    print "Frag is all done"

# ------------------------------------------------------------


# ************************************************************
# Convenience Functions
def getRots(site, variant):
    cmd.get_wizard().set_mode("\""+variant+"\"")

    # Key lines
    # I dont know how they work, but they make it possible.
    # Jason wrote this: If you just write "site" instead of
    #                   "(site)", PyMOL will delete your
    #                   residue. "(site)" makes it an
    #                   anonymous selection.
    #print 'getRots'
    cmd.get_wizard().do_select("(" + str(site) + "/)")
    nRot = cmd.count_states("mutation")
    return nRot

def setNames(obj):
    orig_sequence = {}
    cmd.load(obj)
    cmd.select("prot", "name ca")
    cmd.do("stored.names = []")
    cmd.do("iterate (prot), stored.names.append((resi, resn))")
    for i in stored.names:
        orig_sequence[i[0]] = i[1]
    cmd.do('delete all')
    stored.orig_sequence = orig_sequence
    return orig_sequence

```

```python


# Credit: Thomas Holder, MPI
# CONSTRUCT: - 'res'
#            - 'cpy'
#            -
def localSculpt(obj, site):
    res = str(site)
    cmd.protect('(not %s/) or name CA+C+N+O+OXT' % (res))
    print "Activating Sculpting."
    cmd.sculpt_activate(obj[:-4])
    cmd.sculpt_iterate(obj[:-4], cycles=5000)
    cmd.sculpt_deactivate(obj[:-4])
    cmd.deprotect()


def getOne(three):
    trans = {
        'ALA':'A',
        'ARG':'R',
        'ASN':'N',
        'ASP':'D',
        'CYS':'C',
        'GLU':'E',
        'GLN':'Q',
        'GLY':'G',
        'HIS':'H',
        'ILE':'I',
        'LEU':'L',
        'LYS':'K',
        'MET':'M',
        'PHE':'F',
        'PRO':'P',
        'SER':'S',
        'THR':'T',
        'TRP':'W',
        'TYR':'Y',
        'VAL':'V'
        }
    return trans[three]
# -----------------------------------------------------------


# ***********************************************************
# Expose to the PyMOL shell
cmd.extend('setup', setup)
cmd.extend('frag', frag)
cmd.extend('getRots', getRots)
cmd.extend('localSculpt', localSculpt)
cmd.extend('seq', seq)
# -----------------------------------------------------------

######################
# Modified avf.py
```

```
####################

def writeCatSeq(variant):
    # Convenience list of variants, replacing the '+'.
    varlist = variant.split('+')

    # Initialization of the bash script
    # Escaping BASH syntax.
    # Resetting content of mutant structure file.
    numOfMutations = len(variant.split('+'))

    # Defining a list which contains only the numbers of the
    # residues that will be mutated.
    varTmp = variant.split('+')
    varNum = [i[1:-1] for i in varTmp]
    site = varNum[0]
    nRots = stored.rotamerDict[variant[1:]]

    writeSeq  =\
            '# ' + '*'*50 + '\n' +\
            '# ' + '-'.join(variant.split('+')) + '\n' +\
            '# ' + str(numOfMutations) + '-fold mutant.\n' +\
            '# Resetting mutant file content.\n' +\
            'echo ' + '%'*50 + '\n'\
            'echo "Generating variant structure file of mutant:"\n' +\
            'echo ' + variant + '\n' +\
            'echo ' + '-'*50 + '\n'\
            'echo ' + '\n' +\
            'echo ' + '\n' +\
            'for ROT in $(seq %s)'%nRots +'\n' +\
            'do \n' +\
            'cat /dev/null > %s-' % state + '-'.join(varlist) + '-rot$ROT.pdb\n' +\
            'echo "%s \n\n" >' %mopacSettings +\
            '%s-' % state + '-'.join(varlist) + '-rot$ROT.pdb \n' +\
            'for i in\\\n' +\
            '    {'


    for s in chain:
        # Checking if the number of the residue is in the ones
        # to be mutated, if so, then the write sequence is
        # adjusted.
        if s[5:] in varNum:
            # Locate the index of the side chain which needs
            # to be mutated, then choose the corresponding index
            # in the varTmp list.
            ind=varNum.index(s[5:])
            writeSeq += 'frag-' + variant.split('+')[ind].lower() +"-rot$ROT"+ ',\\\n'
        else:
            # Appending of the WT backbone.
            writeSeq += s + ',\\\n'

    # <<PARAM>>:
    # - Reactant or product state file:        '?-res'
    # - Initial file or optimization job file: '-opt'.
    writeSeq += '}\n' +\
            'do\n' +\
            '    cat $i-%s.pdb >> %s-' % (state, state) + '-'.join(varlist) +\
                '-rot$ROT.pdb\n' +\
            'done\n' +\
            'grep -v \'END\' ' + '%s-' % state + '-'.join(varlist) +\
                '-rot$ROT.pdb > tmp.pdb\n' +\
            'mv tmp.pdb ' + '%s-' % state + '-'.join(varlist) + '-rot$ROT.pdb\n' +\
            '%s %s-' %(mopacCommand, state) + '-'.join(varlist) + '-rot$ROT.pdb\n' +\
            'done\n' +\
            'echo ' + '-'.join(varlist) + ' >> mutantList.dat\n'
    return writeSeq

if __name__ == '__main__':
    pymol.cmd.reinitialize()
    pymol.cmd.load(obj1)
    pymol.cmd.do("run create-mutant-fragments.py")
    # Sequence the 1-state
    pymol.cmd.do("seq 1")
    sleep(2)
    pymol.cmd.do("delete all")
    # Sequence the 3-state
    pymol.cmd.load(obj3)
    pymol.cmd.do("seq 3")
    sleep(2)
    pymol.cmd.sync()
    pymol.cmd.refresh()
    # Create the mutant fragments
    pymol.cmd.do("frag")
    pymol.cmd.refresh()
    pymol.cmd.sync(100000.0 * len(mutations))

    # This assumes that seq_files.py has already been run.
    # Define the backbone chain
    chain = []
    for i in range(1,551):
        for file in os.listdir("."):
            if fnmatch.fnmatch(file, 'seq-?%i-%s.pdb'%(i,state)):
                # Remove the 6 last chars (eg "-3.pdb")
                # And append to sequence of amino acids
                chain.append(file[0:-6])

    #************************
    # Reset the sequence script.
    seqFile = open('seq.sh', 'w')
    seqFile.close()
    seqFile = open('seq.sh', 'a')
    #------------------------

    # Get a list of the mutations in one-letter, site, one-letter format. E.g. L140K
    vars = []
    for site,mutationList in mutations.items():
        for mutant in mutationList:
                vars.append(getOne(stored.orig_sequence[site]) + str(site) + getOne(mutant))
    print vars
    #************************
    # Generating the 'seq.sh' script
    for variant in vars:
        seqFile.write(writeCatSeq(variant))
    seqFile.close()
    #-----------------------
    os.system("bash seq.sh")
    pymol.cmd.do("quit")
# EOF
#-------------------------------------------------------
```

D.2 evaluate-rotamers.sh

```
for mutant in $(cat mutantList.dat)
do
    ROT=1
    while [ -e 3-$mutant-rot$ROT.pdb.out ]
    do
        grep -i "final heat" 3-$mutant-rot$ROT.pdb.out |
          sed 's:.* = ::' | sed 's:KJ/MOL::' >> $mutant.dat
        let ROT=ROT+1
    done
    lowest=$(cat $mutant.dat | sort | tail -1)
    bestRot=`grep -n -e $lowest $mutant.dat | sed 's/\(.*\):.*/\1/'`
    mutLower=`echo $mutant | awk '{print tolower($0)}'`
    cp frag-$mutLower-rot$bestRot-3.pdb frag-$mutLower-3.pdb
    cp frag-$mutLower-rot$bestRot-3.pdb frag-$mutLower-1.pdb
    rm $mutant.dat
    python avf.py 1 $mutant
    bash seq.sh
    python avf.py 3 $mutant
    bash seq.sh
done
```

D.3 assemble-rotamers.py

```
#!/usr/bin/env python

# ***************************************************
# ..................................................
# Assemble rotamers
# ..................................................
# ***************************************************

# This script writes the BASH script which assembles
# the final mutant structure file(s).

# CALLING SEQUENCE EXAMPLE:
# £ python assemble-rotamers.py 3 'L140K'
```

```python

import sys
import fnmatch
import os

state=sys.argv[1]
vars=sys.argv[2:]
if type(vars) == type(""):
    vars = [vars]
print sys.argv
print vars

# Generate backbone list
chain = []
for i in range(1,551):
    for file in os.listdir("."):
        if fnmatch.fnmatch(file, 'seq-?%i-%s.pdb'%(i,state)):
            # Remove the 6 last chars (eg "-3.pdb")
            # And append to sequence of amino acids
            chain.append(file[0:-6])

def writeCatSeq(variant):
    # Convenience list of variants, replacing the '+'.
    varlist = variant.split('+')

    numOfMutations = len(variant.split('+'))

    # <<PARAM>>:
    # - Reactant or product state file:        '?-res'
    # - Initial file or optimization job file: '-opt'.
    writeSeq  =\
            '# ' + '*'*50 + '\n' +\
            '# ' + '-'.join(variant.split('+')) + '\n' +\
            '# ' + str(numOfMutations) + '-fold mutant.\n' +\
            '# Resetting mutant file content.\n' +\
            'echo ' + '%'*50 + '\n'\
            'echo "Generating variant structure file of mutant:"\n' +\
            'echo ' + variant + '\n' +\
            'echo ' + '-'*50 + '\n'\
            'echo ' + '\n' +\
            'echo ' + '\n' +\
            'ROT=1\n' +\
            'while [ -e frag-' + variant.lower()+'-rot$ROT-?.pdb ] \n' +\
            'do \n' +\
            'cat /dev/null > %s-res-' % state + '-'.join(varlist) + '-rot$ROT.pdb\n' +\
            'echo "1scf mozyme cutoff=15 \n\n" > %s-res-' % state +\
                '-'.join(varlist) + '-rot$ROT.pdb \n' +\
            'for i in\\\n' +\
            '    {'

    # Defining a list which contains only the numbers of the
    # residues that will be mutated.
    # First recast from 'G39A+L278A' form to ['G39A', 'L278A'],
    # then generate a list with only numbers ['39', '278'].
    varTmp = variant.split('+')
    varNum = [i[1:-1] for i in varTmp]

    for s in chain:
        # Checking if the number of the residue is in the ones
        # to be mutated, if so, then the write sequence is
        # adjusted.
        if s[5:] in varNum:
            # Locate the index of the side chain which needs
            # to be mutated, then choose the corresponding index
            # in the varTmp list.
            ind=varNum.index(s[5:])
            writeSeq += 'frag-' + variant.split('+')[ind].lower() + '-rot$ROT,\\\n'
        else:
            # Appending of the WT backbone.
            writeSeq += s + ',\\\n'

    # <<PARAM>>:
    # - Reactant or product state file:        '?-res'
    # - Initial file or optimization job file: '-opt'.
    writeSeq += '}\n' +\
            'do\n' +\
            '    cat $i-%s.pdb >> %s-res-' % (state, state) +\
                '-'.join(varlist) + '-rot$ROT.pdb\n' +\
            'done\n' +\
            'grep -v \'END\' ' + '%s-res-' % state + '-'.join(varlist) +\
                '-rot$ROT.pdb > tmp.pdb\n' +\
            'mv tmp.pdb ' + '%s-res-' % state + '-'.join(varlist) +\
                '-rot$ROT.pdb\n' +\
            'let ROT=ROT+1 \n' +\
            'done \n\n'
    return writeSeq

if __name__ == '__main__':
    #***********************
    # Reset the sequence script.
    seqFile = open('seq.sh', 'w')
    seqFile.close()
    seqFile = open('seq.sh', 'a')
    #-----------------------


    #***********************
    # Generating the 'seq.sh' script
    for variant in vars:
        seqFile.write(writeCatSeq(variant))
    seqFile.close()
    #-----------------------

# EOF
#-----------------------------------------------------
```

D.4 avf.py

```
#!/usr/bin/env python

# **************************************************
# ..................................................
# Assemble variant files
# ..................................................
# **************************************************

# DESCRIPTION:
# The variant structure files are being assembled
# from sequence files. Each sequence file contains the
# data of one side chain and  every mutant has a
# fragment file.
# This script writes the BASH script which assembles
# the final mutant structure file(s).
# The variants are entered at the end of the script.

# CALLING SEQUENCE:
# £ python asv.py {1,3} -> seq.sh
# £ bash seq.sh

# PARAMETERS (search for by <<PARAM>>):
# - Reactant [= '1'] or product [= '3'] state files:

import sys
import fnmatch
import os


state=sys.argv[1]
vars=sys.argv[2:]
if type(vars) == type(""):
    vars = [vars]
print sys.argv
print vars

# Available backbone sequence fragements.
chain = []
for i in range(1,501):
    for file in os.listdir("."):
        if fnmatch.fnmatch(file, 'seq-?%i-%s.pdb'%(i,state)):
            # Remove the 6 last chars (eg "-3.pdb")
            # And append to sequence of amino acids
            chain.append(file[0:-6])

def writeCatSeq(variant):
    # Convenience list of variants, replacing the '+'.
    varlist = variant.split('+')

    # Initialization of the bash script
    # Escaping BASH syntax.
    # Resetting content of mutant structure file.
    numOfMutations = len(variant.split('+'))

    # <<PARAM>>:
    # - Reactant or product state file:        '?-res'
    # - Initial file or optimization job file: '-opt'.
    writeSeq  =\
            '# ' + '*'*50 + '\n' +\
            '# ' + '-'.join(variant.split('+')) + '\n' +\
            '# ' + str(numOfMutations) + '-fold mutant.\n' +\
            '# Resetting mutant file content.\n' +\
            'echo ' + '%'*50 + '\n'\
            'echo "Generating variant structure file of mutant:"\n' +\
            'echo ' + variant + '\n' +\
            'echo ' + '-'*50 + '\n'\
            'echo ' + '\n' +\
            'echo ' + '\n' +\
            'cat /dev/null > %s-res-' % state + '-'.join(varlist) + '-ste-ini.pdb\n' +\
            'for i in\\\n' +\
            '    {'

    # Defining a list which contains only the numbers of the
    # residues that will be mutated.
    # First recast from 'G39A+L278A' form to ['G39A', 'L278A'],
    # then generate a list with only numbers ['39', '278'].
    varTmp = variant.split('+')
    varNum = [i[1:-1] for i in varTmp]

    for s in chain:
        # Checking if the number of the residue is in the ones
        # to be mutated, if so, then the write sequence is
        # adjusted.
        if s[5:] in varNum:
            # Locate the index of the side chain which needs
            # to be mutated, then choose the corresponding index
            # in the varTmp list.
            ind=varNum.index(s[5:])
            writeSeq += 'frag-' + variant.split('+')[ind].lower() + ',\\\n'
        else:
            # Appending of the WT backbone.
            writeSeq += s + ',\\\n'

    # <<PARAM>>:
    # - Reactant or product state file:        '?-res'
    # - Initial file or optimization job file: '-opt'.
    writeSeq += '}\n' +\
            'do\n' +\
            '    cat $i-%s.pdb >> %s-res-' % (state, state) +\
                '-'.join(varlist) + '-ste-ini.pdb\n' +\
            'done\n' +\
            'grep -v \'END\' ' + '%s-res-' % state + '-'.join(varlist) +\
                '-ste-ini.pdb > tmp.pdb\n' +\
            'mv tmp.pdb ' + '%s-res-' % state + '-'.join(varlist) +\
                '-ste-ini.pdb\n\n\n'
    return writeSeq

if __name__ == '__main__':
    #***********************
    # Reset the sequence script.
    seqFile = open('seq.sh', 'w')
    seqFile.close()
    seqFile = open('seq.sh', 'a')
    #-----------------------


    #***********************
    # Generating the 'seq.sh' script
    for variant in vars:
        seqFile.write(writeCatSeq(variant))
    seqFile.close()
    #-----------------------

# EOF
#------------------------------------------------------
```

D.5 vsc.py

```
from pymol import cmd
import os
import sys
from pymol import stored
from os.path import splitext

# ***********************************************************
# -----------------------------------------------------------
# VSC: Variant Side Chains
# -----------------------------------------------------------
# ***********************************************************

# DESCRIPTION:
# - Mutate a residue and save the fragment amino acid.

# REQUIRES:
# - PDB file to mutate

# CALLING orig_sequence:
# PyMOL> run vsc.py
# PyMOL> frag <state>
# where <state> is either 1 or 3

# PARAMETERS:
# - <mutations> dictionary below
obj = '3-wt-opt.pdb'

# OPTIONS:
# - The number of conformers can be adjusted.
# - The mutated side chain can be optimized locally
#   by vdW minimization.


# ************************************************************
def setup(obj):

    # Various set up
    pwd = os.getcwd()
    #print "os.getcwd()", os.getcwd()
    cmd.do('wizard mutagenesis')
    cmd.do('refresh_wizard')

    # Save residue names and numbers.
    orig_sequence = setNames(obj)
    #print orig_sequence

    # Keeping track of the mutations
    # Example: '42':  ['GLY', 'ASN', 'VAL', 'ALA']
    # Important: Trailing commata.
    mutations = {
                 '140': ['ARG', 'LYS', 'ASN', 'GLN'],
                 #'141': ['ASN', 'GLN'],
                 #'189': ['ALA', 'LYS', 'TYR', 'ASN', 'GLN'],
                 # '38': ['HIS', 'ASN'],
                 # '39': ['ALA'],
                 # '40': ['GLY'],
                 # '41': ['SER'],
                 # '42': ['ALA', 'GLY', 'ASN', 'VAL'],
                 # '46': ['LYS'],
                 # '49': ['LYS', 'ASN'],
                 #'103': ['GLY'],
                 #'104': ['PHE', 'GLN', 'TYR'],
                 #'132': ['SER', 'ASN'],
                 #'134': ['ALA', 'PHE', 'ILE', 'LEU', 'THR', 'VAL'],
                 #'157': ['VAL'],
                 #'188': ['GLN', 'ARG'],
                 #'191': ['ARG'],
                 #'221': ['ARG'],
                 #'223': ['GLY', 'ASN'],
                 #'225': ['ILE', 'LYS', 'MET'],
                 #'278': ['ALA', 'GLY'],
                 #'281': ['GLY'],
                 #'282': ['GLY'],
                 #'285': ['ALA'],
                 #'286': ['ALA']
    }
    return pwd, orig_sequence, mutations
# ------------------------------------------------------------


# ************************************************************
# 'state=state': The first variable is the variable used within
# the scope of this function. The second variable is the one
# in the global scoped and defined at the top of the module.
def frag(state, obj=obj):
    pwd, orig_sequence, mutations = setup(obj)

    # Add and retain hydrogens
    cmd.get_wizard().set_hyd("keep")

    # Run over all sites where to mutate
    for site in mutations.keys():

        variants = mutations[site]

        # Run over all variants.
        for variant in variants:
            print variant
            print site
            cmd.load(obj)

            cmd.do('wizard mutagenesis')
            cmd.do('refresh_wizard')
            cmd.get_wizard().set_mode(variant)
            cmd.get_wizard().do_select(site + '/')

            # Get the number of available rotamers at that site
            # Introduce a condition here to check if
            # rotamers are requested.
            # <<OPTION>>
            nRots = getRots(site, variant)
            #if nRots > 3:
            #    nRots = 3
            #nRots=1

            cmd.rewind()
            for i in range(1, nRots + 1):

                cmd.get_wizard().do_select("(" + site + "/)")
                cmd.frame(i)
                cmd.get_wizard().apply()

                # Optimize the mutated sidechain
                #<<OPTION>>
                print "Sculpting."
                localSculpt(obj, site)

                # Protonation of the N.
                cmd.do("select n%d, name n and %d/" % (int(site), int(site)))
                cmd.edit("n%d" % int(site), None, None, None, pkresi=0, pkbond=0)
                cmd.do("h_fill")

                # Protonation of the C.
                cmd.do("select c%d, name c and %d/" % (int(site), int(site)))
                cmd.edit("c%d" % int(site), None, None, None, pkresi=0, pkbond=0)
                cmd.do("h_fill")

                # Definition of saveString
                saveString  = '%s/' % pwd
                saveString += 'frag-' + getOne(orig_sequence[site]).lower() +\
                              site + getOne(variant).lower() + '-rot%s-%s.pdb, ' % (i,state) +\
                              '((%s/))' % site
                #print saveString
                cmd.do('save %s' % saveString)
                #cmd.do('save %s' % saveString.lower()) # lower() breaks paths with uppercase chars
            cmd.do('delete all')
            cmd.set_wizard('done')
# ------------------------------------------------------------


# ************************************************************
# Convenience Functions
def getRots(site, variant):
    cmd.get_wizard().set_mode(variant)

    # Key lines
    # I dont know how they work, but they make it possible.
    # Jason wrote this: If you just write "site" instead of
    #                   "(site)", PyMOL will delete your
    #                   residue. "(site)" makes it an
    #                   anonymous selection.
    #print 'getRots'
    cmd.get_wizard().do_select("(" + str(site) + "/)")
    nRot = cmd.count_states("mutation")
    return nRot

def setNames(obj):
    orig_sequence = {}
    cmd.load(obj)
    cmd.select("prot", "name ca")
    cmd.do("stored.names = []")
    cmd.do("iterate (prot), stored.names.append((resi, resn))")
    for i in stored.names:
        orig_sequence[i[0]] = i[1]
    cmd.do('delete all')
    #print stored.names
    return orig_sequence



# Credit: Thomas Holder, MPI
# CONSTRUCT: - 'res'
#            - 'cpy'
#            -
def localSculpt(obj, site):
    res = str(site)
    cmd.protect('(not %s/) or name CA+C+N+O+OXT' % (res))
    print "Activating Sculpting."
    cmd.sculpt_activate(obj[:-4])
    cmd.sculpt_iterate(obj[:-4], cycles=5000)
    cmd.sculpt_deactivate(obj[:-4])
    cmd.deprotect()


def getOne(three):
    trans = {
       'ALA':'A',
       'ARG':'R',
       'ASN':'N',
       'ASP':'D',
       'CYS':'C',
       'GLU':'E',
       'GLN':'Q',
       'GLY':'G',
       'HIS':'H',
       'ILE':'I',
       'LEU':'L',
       'LYS':'K',
       'MET':'M',
       'PHE':'F',
       'PRO':'P',
       'SER':'S',
       'THR':'T',
       'TRP':'W',
       'TYR':'Y',
       'VAL':'V'
       }
    return trans[three]
# -------------------------------------------------------------


# *************************************************************
# Expose to the PyMOL shell
cmd.extend('setup', setup)
cmd.extend('frag', frag)
cmd.extend('getRots', getRots)
cmd.extend('localSculpt', localSculpt)
# -------------------------------------------------------------
```

D.6 intcha.py

```
#!/usr/bin/env python

# ***************************************************
# ...................................................
# Separate Interpolation Files for overall charge
# ...................................................
# ***************************************************

# DESCRIPTION:
# - Write separate interpolation files to calculate overall charge.

# CALLING SEQUENCE:
```

```
# £ python intcha.py £1 £2 £name

import sys
reac, prod, name = sys.argv[1], sys.argv[2], sys.argv[3]

reactData, interData = open(reac, 'r'), open(prod, 'r')
reactValue, interValue = reactData.readlines(), interData.readlines()

n = 10
diff = []
diffFrac = []
interpolation = []

# Run over coordinates
for i in enumerate(reactValue):
    # Change in each coordinate for every atom.
    dx, dy, dz =\
       eval(interValue[i[0]][31:38]) - eval(reactValue[i[0]][31:38]),\
       eval(interValue[i[0]][38:46]) - eval(reactValue[i[0]][38:46]),\
       eval(interValue[i[0]][46:54]) - eval(reactValue[i[0]][46:54])
    #print dx, dy, dz
    diff.append([round(dx, 3), round(dy, 3), round(dz, 3)])

print 'Number of vectors in the diff-list'
print len(diff)

for atom in diff:
    diffFrac.append([c/float(n) for c in atom])

# Run over interpolation step
for i in range(n):
    # Run over atom
    set = []
    for atom in range(len(reactValue)):
        xi = eval(reactValue[atom][31:38]) + diffFrac[atom][0]*i
        yi = eval(reactValue[atom][38:46]) + diffFrac[atom][1]*i
        zi = eval(reactValue[atom][46:54]) + diffFrac[atom][2]*i
        set.append('%7.3f %7.3f %7.3f' % (xi, yi, zi))
    interpolation.append(set)
print 'len(interpolation)', len(interpolation)
print 'len(set)', len(set)

def makeSeparateFiles(name):
    for step in range(1, len(interpolation)+1):
        writeMopFile = open('1-3-cha-%s-%03i.mop' % (name,step), 'w')
        writeString = ''
        #writeString = 'mozyme charge=-2 cutoff=3 1scf \n\n\n'
        writeString = 'charges\n\n\n'
        for atom in range(len(reactValue)):
            writeString += reactValue[atom][:31]\
                          + interpolation[step-1][atom]\
                          + reactValue[atom][54:]
        writeMopFile.write(writeString)
        writeMopFile.close()
```

```

makeSeparateFiles(name)
```

D.7 cha2scf.sh

```
#!/bin/bash

# Generated: 16.08.2011

# ACTION:
# 1-3-cha-W104F-001.out -> 1-3-1scf-W104F-001.mop

# REQUIRES:
# - 'cha.mop' files (Templates for 1scf files)
# - 'cha.out' files (carry charge information)

# Calculation of MOPAC charges and auto-writing
# of correct input header for 1SCF calculation.
# The 1SCF output is used for the optimization
# jobs, since only the MOPAC files allow to set
# the optimization flags.

# CALLING SEQUENCE:
# £ auto_scf.sh 1-3-cha-W104F-001.out

# Second call below:
# £ vi -c "1,1s/charge= /charge=/g" -c "wq" 1-3-1scf-W104F-001.mop

# NOTE:
# Good practice to run the script only until the
# two 'echo' calls to check if the naming works.

# 'name' is, e.g., 'G39A-001'
# 'nameIni' is e.g.k, 'G39A'
name=${1/1-3-cha-/}
name=${name/.out/}
nameIni=${name/%-[0-9][0-9][0-9]/}

echo $name
echo $nameIni

ch=`grep "COMPUTED CHARGE ON SYSTEM" $1|cut -d ":" -f 2`
echo $ch

# Optimization keyword line
#echo charge=£ch mozyme cutoff=15 gnorm=0.5 pdbout >> tmp-£name.mop

# 1SCF keyword line and two emply lines.
echo charge=$ch mozyme cutoff=3 1scf eps=78.4 >> tmp-$name.mop
echo >> tmp-$name.mop
echo >> tmp-$name.mop

```

```
# Append coordinates back to new input file.
# The coordinates are the same as in the 'cha.mop' input file (which
# is in PDB format). Taking all but the first three lines.
lenTot=`cat ${1/out/mop}|wc -l`
mv tmp-$name.mop 1-3-scf-$name.mop
tail -n -$((lenTot-3)) ${1/out/mop} >> 1-3-scf-$name.mop

# Remove whitespace between 'charge=' and '-3'.
vi -c "1,1s/charge= /charge=/g" -c "wq" 1-3-scf-$name.mop
```

D.8 scf2opt.sh

```
#!/bin/bash

# ***************************************************
# ..................................................
# Transfer 1scf output to optimization input.
# ..................................................
# ***************************************************

# £HOME/scripts_model_generation/scf2opt.sh

# DOES:
# 1) '1-scf.arc' -> '1-3-opt.mop'
# 2) Sets optimization flags of reaction coordinate.

# REQUIRES:
# - '1scf.arc' files, which are the templates for 'opt.mop' files.
# - '1scf.mop' files, required to locate 'SER OG' line.

# PARAMETERS:
# - How many lines above the last line is the carbonyl
#   carbon of the substrate       -> <offset>
# - Serine identifier: identify the nucleophilic
#   side chain oxygen             -> <serIdentifier>
# - MOPAC optimization parameters -> Set at the end of the file.

# ----------------------------------
# Adjust these two parameters below:
# £(Line_number_of_last_atom_of_substrate)-£(Line_number_of_atom_above_atom_to_constrain)
offset=21
# Serine oxygen identifier.
serIdentifier="1543  C   SER A 105"
# ----------------------------------

# CALLING SEQUENCE:
# £ scf2opt 1-3-1scf-W104Q-001.arc

# Write 'opt' file.
scfarc=$1
optarc=${scfarc/scf/opt}
optmop=${optarc/arc/mop}
```

```
cp $scfarc $optmop

# User info.
echo Writing $optmop

# Removing '1scf.arc' header.
tot=`cat $optmop|wc -l`
lineFin=`grep -i -n "FINAL GEOMETRY" $optmop|cut -d ":" -f 1`
tail -n -$((tot-lineFin)) $optmop > tmp.mop; mv tmp.mop $optmop

# Setting SER OG '+0' flag.
serLine=`grep -n "$serIdentifier" ${optmop/opt/scf}|cut -d ":" -f 1`
vi -c "$serLine,${serLine}s/+1/+0/g" -c "wq" $optmop

# Setting substrate C '+0' flag.
tot=`cat $optmop|wc -l`
cLine=$((tot-offset))
vi -c "$cLine,${cLine}s/+1/+0/g" -c "wq" $optmop

# Replace '1scf' keywords. Set MOPAC optimization parameters.
vi -c "1,1s/cutoff=3 1scf/cutoff=15 gnorm=0.5 pdbout/" -c "wq" $optmop
```

D.9 fix.py

```
#!/usr/bin/python

# ***************************************************
# ...................................................
# Fix selected side chains.
# ...................................................
# ***************************************************


# DESCRIPTION:
# The script loads the 'opt.mop' file and sets the
# optimization flags to '+0', i.e. constrains the
# atom.

# PARAMETERS:
# Choose the side chains in the list below:
residues_to_fix = [
                   '50',
                  '133',
                  '156',
                  '277',
                  '280'
                  ]

# CALLING SEQUENCE:
# python 1-3-opt-*.mop

import sys
```

```python
from os.path import splitext

opt_dat = open(sys.argv[1], 'r')
opt_val = opt_dat.readlines()
opt_dat.close()

mop_string = ''

tmp_dat = open(sys.argv[1], 'w')
print "Fixing side chains in:", splitext(sys.argv[1])[0]

for line in opt_val:
    # Only consider lines with more than 4 elements,
    # ends up being lines with atom data.
    if len(line.split()) != 0:
        # Discard the last character of the '3' element,
        # its a closing brace ')'.
        if line.split()[3][:-1] in residues_to_fix:
            # Replace the optimization flags.
            mop_string += line[:34] + '0' + line[35:50] + '0' + line[51:66] + '0' + line[67:]
        else:
            mop_string += line
    else:
        mop_string += line

tmp_dat.write(mop_string)
tmp_dat.close()
```

D.10 opt2spe.sh

```bash
#!/bin/bash

# £HOME/scripts_model_generation/app-spe.sh

# - Copies optimization output to SPE input.
# - Replaces optimization keywords by
#   spe keywords using vim.

# REQUIRES:
# - 'opt.arc' files

# CALLING SEQUENCE:
# £ ./app-spe.sh <1-3-opt*arc>

# Convenience variable.
name=${1/1-3-opt-/}
name=${name/-???.arc/}

# Copy the arc files from the optimization
# to new files.
spearc=${1/opt/spe}
spemop=${spearc/arc/mop}
```

```
cp $1 $spemop

# Remove all ’WARNING’ lines
grep -i -v -w "warning" $spemop > tmp.dat
mv tmp.dat $spemop

echo Writing $spemop

# Remove the MOPAC header.
# 1) Determining header length: £lenHead.
# 2) Determining total length:  £lenTot.
#    <tr -s>: squeezing multiple white space.
lenHead=`grep -n "FINAL GEOMETRY" $spemop|cut -d ":" -f 1`
lenTot=`cat $spemop|wc -l`
tail -n -$((lenTot-lenHead)) $spemop > ./tmp.dat
mv ./tmp.dat $spemop

# Replace the ’+0’ flags from the optimization.
# Old version.
# vi "+:%s/+0/+1/g" "+wq" £spemop
vi -c "%s/+0/+1/g" -c "wq" $spemop
vi -c "1,1s/mozyme cutoff=9 gnorm=5.0 pdbout/1scf/" -c "wq" $spemop
```

D.11 profiles.py

```
#!/usr/bin/python

# -> £HOME/scripts_model_generation/analyse.py

# NOTE:
# - Extracts and presents the data from the ’spe.arc’ files
# - Generates a profile graph.

# REQUIRES:
# - grep -i "heat" ./*spe-*arc > data.txt

# CALLING SEQUENCE:
# £ python ./profiles data.txt

import sys
dat=open(sys.argv[1], ’r’)
val=dat.readlines()
dat.close()

# ***************************************************
# Register the variants.
def register():
    for line in val:
        name = get_name(line)
        if name not in names:
            names.append(name)
    # Highest order multiple variants first.
    names.sort(key=len, reverse=True)

def get_name(line):
    # Discarding '1-3-' part.
    name = line.split()[0]
    name = name.split('-')[3:-1]
    name = '-'.join(name)
    return name

def get_x_range(name, counter):
    data = open('%04i-' % counter + name + '.dat', 'r')
    valu = data.readlines()
    e_tmp = get_energy(name)
    return eval(min(e_tmp))-2, eval(max(e_tmp))+2

def get_energy(name):
    energy = []
    for frame in val:
        if get_name(frame) == name:
            energy.append(frame.split()[5])
    return energy
# --------------------------------------------------


# **************************************************
def write_dat(name, counter):
    gnu_dat = open('%04i-' % counter + name + '.dat' , 'w')
    energy = get_energy(name)
    e0 = energy[0]
    for e in energy:
        gnu_dat.write(str(eval(e)-eval(e0)) + '\n')
    gnu_dat.close()

def write_gnu(name, counter):
    gnus  = 'set terminal png\n'
    gnus += 'set output \'%04i-%s.png\'\n' % (counter, name)
    e0 = get_energy(name)[0]
    x_min, x_max = get_x_range(name, counter)
    #gnus += 'set xrange[%s:%s]\n' % (str(x_min),str(x_max))
    gnus += 'plot \'' + '%04i-' % counter + name + '.dat' + '\' with lines lw 2\n'
    gnus += 'set output\n'

    gnuf  = open('%04i-' % counter + name + '.gnu', 'w')
    gnuf.write(gnus)
    gnuf.close()
# --------------------------------------------------


# **************************************************
# LAUNCH PART
if __name__ == '__main__':

    # Get the available variants and store them in 'names'.
    names = []
    names.sort(key=len)
    register()
    stop = 3

    name_len  = 0.0
    counter = 0
    for n in names:
        print n
        write_dat(n, counter)
        write_gnu(n, counter)
        counter += 1
# --------------------------------------------------
```