The Thesis Committee for Jerry Chil Kim
Certifies that this is the approved version of the following thesis:

Computations of Electrostatic Potentials and Energies of Dynamic Biomolecules

APPROVED BY
SUPERVISING COMMITTEE:

Supervisor:

Pengyu Ren

Co-Supervisor:

Alexander A. Demkov

Computations of Electrostatic Potentials and Energies of Dynamic Biomolecules

by

Jerry Chil Kim, BA&S

Thesis

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Master of Arts

The University of Texas at Austin

May 2016

Acknowledgements

I would like to thank Dr. Pengyu Ren for his support and knowledge of biophysics. I would also like to thank Dr. Alex Demkov for offering additional critiques and feedback for this thesis

Abstract

Computations of Electrostatic Potentials and Energies of Dynamic Biomolecules

Jerry Chil Kim, MA

The University of Texas at Austin, 2016

Supervisor: Pengyu Ren

Co-Supervisor: Alexander A. Demkov

We present ongoing research into the simulation of biomolecules in a solvent. We first describe the various models used to model molecular electrostatics including the methods that attempt to modify the Poisson-Boltzmann equation due to its limitations. We then modify the boundary element solver *MolEnergy*, which solves the linearized Poisson-Boltzmann equation to determine the polarization energies of biomolecules. With our modifications it can now rely on an automated process, utilize a low-discrepancy pseudo-random number generator to yield different molecular configurations, and implement parallel computing to determine the potentials and energies of those configurations in parallel. Our modifications increase the accuracy and speed of *MolEnergy*.

Contents

List of Figures

1 Introduction

Biophysics is a rapidly growing field of scientific research. Proteins are the basic components that are necessary for life because they are the catalysts for all processes that allow beings to live. Proteins consist of amino acids. Amino acids are consisted of a carboxyl group, amino group, Hydrogen atom, and "side-chain" all bonded to a central alpha-Carbon atom. The side-chain varies depending on each amino acid. Side-chains of different amino acids can also bond with each other to alter the configuration of the entire protein and this leads to protein folding [3]. Numerous side-chain configurations can exist for a given protein and it has been shown that proteins are actually ensembles of the various conformations caused by side-chain rotations [72]

We can determine the equilibrium states of proteins in solutions by looking at the free energy function. However, this is not a simple task because there are numerous local minima and wells in the energy function. One of the components of the energy function is the configurational entropy [41, 32], which is composed of vibrational and conformational entropy [39]. As Goethe et al. [32] mentioned, the conformational entropy, $S_{Cfm} = -k_B \sum_i p_i log(p_i)$, is the sum over each energy minima i and p_i is the probability that the system is in the minima state i, and the vibrational entropy is the weighted average of the vibrational motions within each minima.

Calculating the polarization energy, which is a component of solvation energy and describes the interactions between the immersed molecule and solvent, is a significant problem because it is needed in numerous applications, such as determining the binding effect in a protein-protein interaction. Although proteins by themselves do not serve much purpose, when proteins bind with other proteins, they can trigger reactions that have biological effects. The rapidly emerging field of computer-aided drug design uses computers to predict which drugs, or target proteins, can bind with

which receptor proteins and also determine the resulting biological effects from these interactions which can then be used in medicine and pharmacy to treat human beings with certain health problems [45]. The binding affinity is an important variable because it can give us a measure of how proteins bind with other proteins. Current research is using protein docking, virtual screening and other computational methods to identify more potential targets for drugs [75]

Although Molecular Dynamics simulations (MD) are popular methods to study these interactions, they are also often computationally expensive. Therefore, we resort to using the Poisson-Boltzmann (PB) equation to model the interactions and solve the equation for the potentials and energies. The Poisson-Boltzmann (PB) equation is one popular method to model these interactions, and we will explain other models that attempt to model these interactions in Chapter 2.

In addition to the Poisson-Boltzmann equation, we can add a Stern layer or use the Debye-Huckel approximation so that the resulting Poisson-Boltzmann equation becomes the linearized Poisson-Boltzmann equation (LPBE).

Solving the LPBE analytically is difficult, so it is common to solve it using numerical methods and discretizing the region of space [46]. Although finite-element [12, 20, 25, 26, 11, 22, 36] and finite-difference [31, 48, 57, 35, 37, 56] solvers exist to solve the PB equation, including the well-known finite-difference solver $APBS$ [12], as Bajaj et al. [7] pointed out, they require $O(h^{-3})$ degrees of freedom whereas the boundary element method only requires $O(h^{-2})$ degrees of freedom, where h is the grid size.

Bajaj et al. [7] developed a boundary element solver called *MolEnergy* that can solve the linearized-PB equation for proteins in solutions. As explained in detail in their paper, it solves the dBIE (derivative boundary integral equations developed

by Juffer et al. [38], more details in Chapter 3) formulation of the PB equation, it represents the surface of the biomolecule with an algebraic spline molecular surface [74], and it utilizes the scientific computing package KIFMM3d (Kernel Independent Fast Multipole Method) to efficiently solve the dBIE equations. They also mentioned that it outperformed several alternative approaches such as the nBIE formulation (non-derivative boundary integral equations [69, 70, 71]) and gave more accurate results on a single ion example [7] and computed energy values more consistently on certain biomolecules when compared to *APBS*

The goal of this project is to modify *MolEnergy* so that it uses an automated process instead of reading in external files, it includes the *PDBUQ* library from the *MolSurf* [2] application to generate samples of Cartesian coordinates of atoms to generate dynamic configurations of atoms for each molecule, and so it can utilize parallel computing to compute the potentials and energies of the various molecular configurations in parallel. Previously, one had to generate external RAWN and QUAD files using the *MolSurf* application. Also, *MolEnergy* could previously only handle static configurations of molecules, but the point of our modifications is so it can handle dynamic molecular configurations as well.

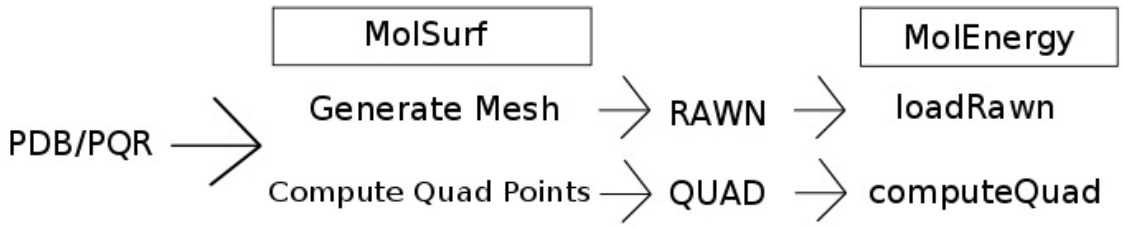

Figure 1: MolEnergy previously relied on obtaining the RAWN and QUAD files generated from MolSurf

Previously, to generate dynamic molecular configurations, one had to use *PDBUQ* library within *MolSurf* externally from *MolEnergy* to generate new molecular con-

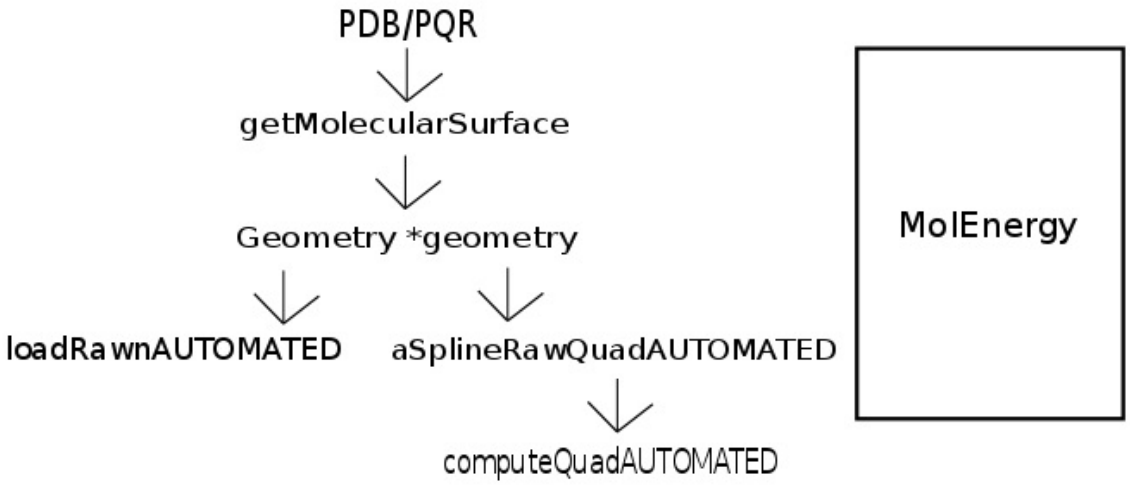

Figure 2: We modified MolEnergy so that it includes the getMolecularSurface and aSplineRawQuad functions from MolSurf to generate the mesh and quadrature points within MolEnergy. This reduces the need to obtain the external RAWN and QUAD files from MolSurf

figurations and then use *MolEnergy* on each new configuration. We now include the $PDBUQ$ library within *MolEnergy*, which generates samples of the xyz coordinates of atoms based on their $temperature factor$ values to simulate dynamic atoms with vibrational movements. Finally, we want to modify *MolEnergy* so that it can support parallel computing in order to compute energies and potentials of these dynamic configurations in parallel, whereas previously one would have to run *MolEnergy* independently on seperate cores using scripts. While our research does not really remove any shortcomings of previous research, modifying the *MolEnergy* so that it uses an automated process, generates dynamic molecular configurations, and computes the potentials and energies of those configurations in parallel would allow us to run many energy computations quickly and we only have to run *MolEnergy* once.

For future work with the *MolEnergy* application, we wish to include an algorithm that simulates molecular conformations by sampling the torsional angles of the molecules. In the following we will describe the techniques used in each of these areas

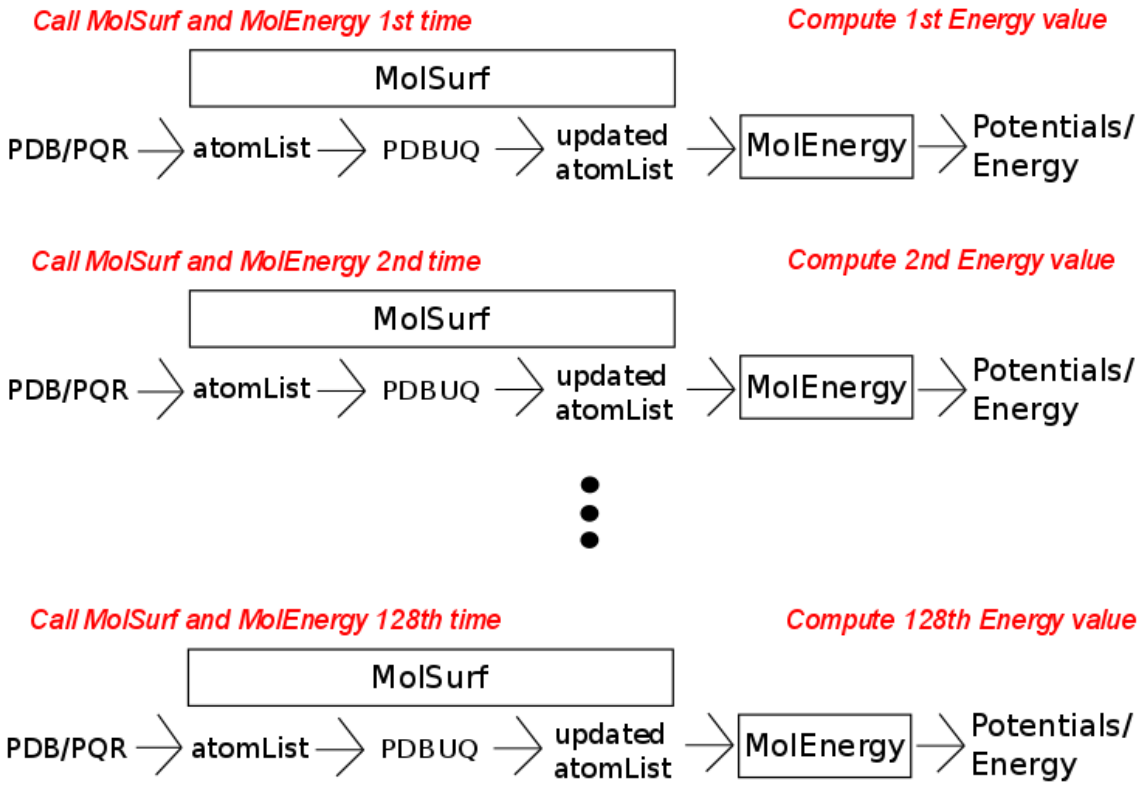

Figure 3: To compute the potentials and energies of a dynamic molecular configuration with atomic vibrations, one had to compute an ensemble average of the configurations by running MolEnergy to generate each new configuration and then run MolEnergy on each of those configurations to compute the potentials and energies

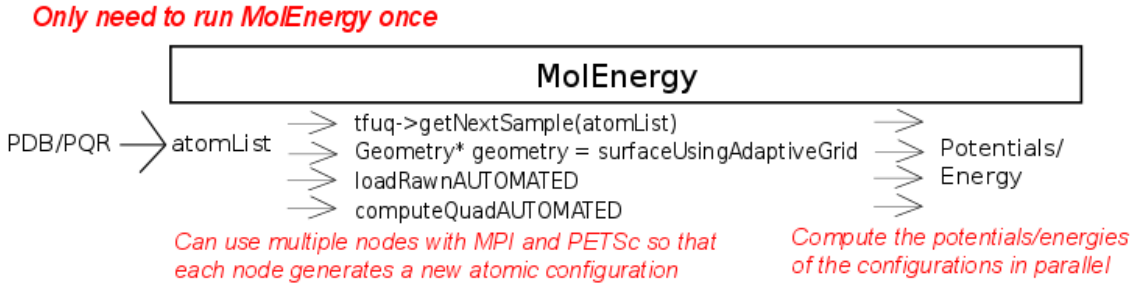

Figure 4: With our modifications to MolEnergy, we only need to run MolEnergy once, and it can generate the atomic configurations within itself and then compute the potentials and energies for each configuration. These modifications allow all of this to run in parallel

and present the current state of their implementations and integration. In chapter 2 compare the Poisson-Boltzmann equation with the Size-Modified Poisson-Boltzmann equation and other models. Chapter 3 describes the boundary integral formulation, the geometry construction, and the details about how *MolEnergy* previously worked. In chapter 4 we describe the derivation of molecular configurations based on sampling Cartesian coordinates of atoms and torsional angles of molecular bonds. Chapter 5 explains our modifications *MolEnergy* and the results we obtained.

2 Theoretical Background

To model electrostatic interactions of biomolecules in solvents, there exist implicit and explicit solvent models. As mentioned in [55], implicit solvent models treat the solvent as if it fills space uniformly and as an ensemble average of solvent configurations while explicit solvent models explicitly consider the effects of each solvent particle and configuration. The authors also mention that explicit solvent models require significantly more computational resources and are considerably more time-consuming than implicit solvent models. This is one of the reasons why MolEnergy uses the implicit solvent model PBE to model the interactions of biomolecules immersed in solvents. However, implicit solvent models are not without their weaknesses, which we will address later in this chapter

First, we examine the Poisson-Boltzmann equation. Then, we consider methods to combine continuum solvent models, in particular the Generalized Born (GB) model, with Molecular Dynamics simulations. We will also discuss the Stern Layer, Size-Modified Poisson-Boltzmann Equation, and other methods that attempt to resolve the problems with the Poisson-Boltzmann equation.

2.1 Poisson-Boltzmann Equation

The PB equation treats the solvent implicitly and computes the electrostatic potential in a system consisting of a biomolecule with a solvent. Figure 5 is a representation of this system. The PB equation is below (from [7]):

$$\nabla(\epsilon(\mathbf{z})\nabla\phi(\mathbf{z})) = \rho(\mathbf{z}) \tag{2.1}$$

where the ϵ is the dielectric constant, ϕ is the potential, and ρ is the charge density.

However, the charge density ρ consists of the atoms of the molecule $\rho_c(\mathbf{z})$ and the ions of the solvent $\rho_b(\mathbf{z})$, giving the equation below (also from Bajaj et al. [7])

$$\rho(\mathbf{z}) = \rho_c(\mathbf{z}) + \rho_b(\mathbf{z}) = -4\pi \sum_k^{n_c} \frac{q_k}{\epsilon_i} \delta(\mathbf{z} - \mathbf{z}_k) + \lambda(\mathbf{z}) \sum_i e_c z_i c_i e^{\frac{-e_c z_i \phi(\mathbf{z})}{k_B T}} \tag{2.2}$$

They defined the parameters in the equation above and additional parameters needed later as:

$\mathbf{z}_k$ location of atom k

$\mathbf{q}_k$ charge of atom k

n_c number of atoms

λ characteristic function of the set $\mathbb{R}^3$ e_c charge of an electron

k_B Boltzmann's constant

T absolute temperature

$\mathbb{I} = \frac{1}{2} \sum_i c_i z_i^2$

c_i concentration of i^{th} ionic species

z_i charge of i^{th} ionic species

They then showed that it is possible to apply the Debye-Huckel approximation to get a linear approximation to the $\rho_b(\mathbf{z})$ term to derive the linearized-PB equation (also from Bajaj et al. [7])

$$\nabla(\epsilon(\mathbf{z})\nabla\phi(\mathbf{z})) = \rho_c(\mathbf{z}) + \rho_b^L(\mathbf{z}) \tag{2.3}$$

where $\rho_b^L(\mathbf{z})$ is the resulting term from applying a linear approximation to $\rho_b(x)$

This approximation is applied for dilute solutions. Numerical techniques can then be used to solve this equation.

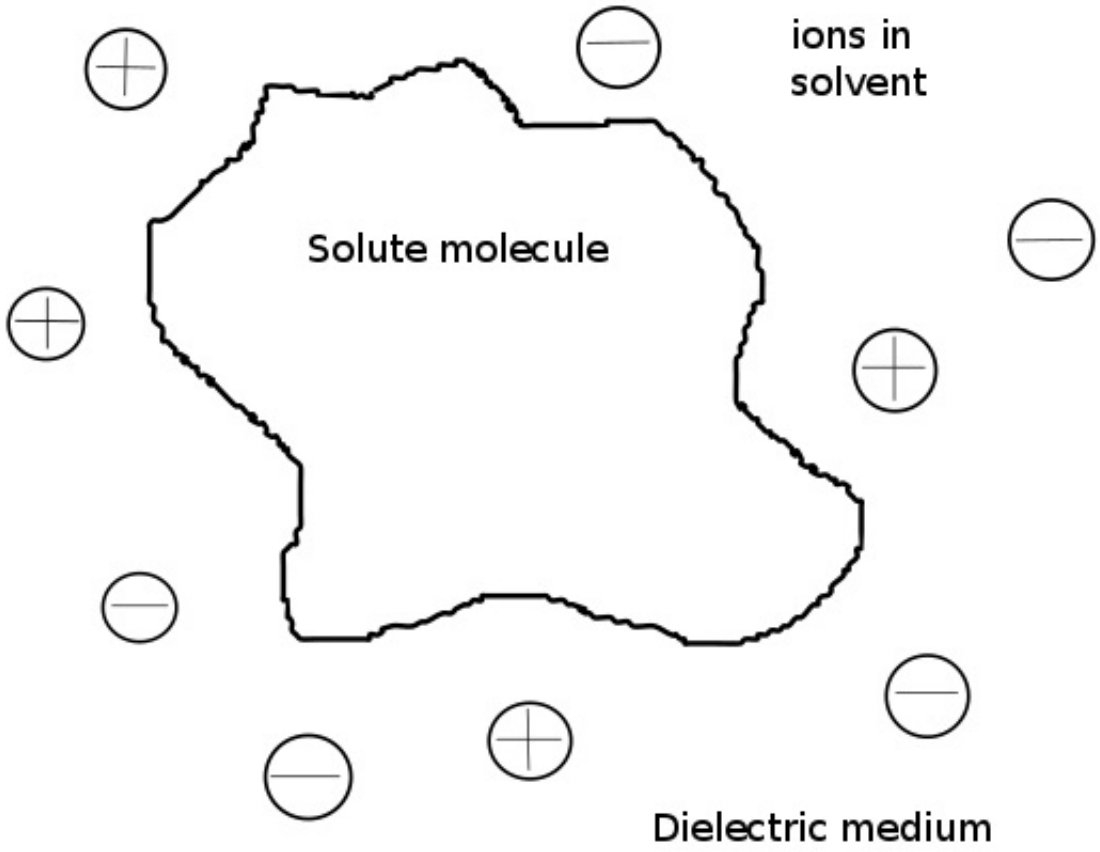

Figure 5: Representation of a biomolecule immersed in an ionic solution

One of the problems with PB include sometimes producing results that are not produced in nature, such as when Bai et al. [5] determined it insufficiently considered

the screening of magnesium ions amongst DNA duplexes. PB also treats ions as point-like, which the Stern layer and the Size-Modified Poisson-Boltzmann Equation (SMPBE) attempt to address. We discuss these issues further in sections 2.3 and 2.4.

2.2 Continuum models becoming microscopic

There are strengths and weaknesses to implicit solvent and explicit solvent models. As pointed out in [59], problems with implicit solvent models include having to explicitly specify the parameters and producing inaccurate results when considering short-range effects. Ritchie and Webb [54] also mentioned another problem of implicit solvents models is the difficulty in verifying its results with experimental results.They tried to address this by computing the electrostatic fields of nitrile bonds and comparing them with vibrational Stark effect spectroscopy (VSE). As a result, Simonson [59] mentioned that recent studies have attempted to address these issues by modifying implicit solvent models to incorporate elements of explicit solvent models such as Molecular Dynamics (MD) on the solute [55, 52, 4, 42], microscopic dieletric relaxation and solute conformations. Simonson also mentioned that recent studies of using MD with GB has produced conformational free energies that are improvements on previous models [64, 51, 28, 62] and they sampled conformations noticeably faster than explicit solvent models [59]. Moreover, Simonson noted that MD with implicit solvent models performed and generated conformational samples noticeably faster than explicit solvent models[27, 66, 62], and the probabilities they assign to molecular conformations match reality better as well [55, 63].

In summary, due to the problems with prior implicit solvent models, recent studies have tried to incorporate MD and have shown to run more efficiently than explicit solvent models

2.3 Stern Layer

One direct attempt to resolve some of the issues with the PB equation is including the Stern layer. Because PB treats ions as point-like, Harris [34] mentioned that this can produce unrealistically high ion density near the surface of a biomolecule in a solution. Adding the Stern layer, which is an ion-exclusion region near the molecular surface, and treating the ions as finite-size addresses these issues [18, 60].

2.4 Size-Modified Poisson-Boltzmann Equation

The Size-Modified Poisson-Boltzmann Equation (SMPBE) also attempts to address the issue of point-like ions and ion densities exceeding a realistic limit by treating ions as finite-like, including a packing energy so the ion densities cannot exceed a certain limit, and considering a saturation layer [17, 23, 34]. Silalahi et al. [58] obtained electrostatic free energies that were vastly different to those from PB for the system they examined. However, they could not compare their results to experimental results. Although SMPBE has produced results that match experiment better than PB [23], further studies are necessary to validate the comparisons. In addition, in general, it is much slower than PB [76, 43]. Moreover, Wang et al. [65] determined that it only offers marginal advantages to PBE for the systems they examined.

2.5 Other Methods

Other methods to address the flaws with PB include [58] using a cutoff on the maximum local salt concentration [33], Coulomb gas with finite size [24], modified Poisson-Boltzmann based on the generalized-Poisson Fermi formalism [61], lattice statistics models [17, 47], and more.

3 Boundary Integral Formulation and Geometry Construction and *MolEnergy* Implementation Details

In this chapter we cover the details of the components that $MolEnergy$ consists of in order to compute the polarization energies of biomolecules immersed in solutions. As mentioned by Bajaj et al. [7], the solver $MolEnergy$ consists of solving the dBIE (derivative boundary integral equations of PB), generating the algebraic spline molecular surface (ASMS), computing the integrals with various quadrature rules, and then utilizing scientific computing methods to solve the resulting equations for the polarization energies. We cover the various components of $MolEnergy$ in detail here.

3.1 Boundary Integral Formulation

Because $MolEnergy$ solves the PBE represented as a system of derivative boundary integral equations (dBIE) as developed by Juffer et al. [38], we discuss here how we about the dBIE from the LPBE. Starting with equations 2.2 and 2.3, Bajaj et al. [7] showed how we can obtain a set of boundary conditions for how the equations must be satisfied on the molecular boundary. Using Green's second identity and, as Bajaj et al. [7] pointed out, "taking linear combinations of the resulting equations and their derivatives", we can obtain the dBIE (below equations from Bajaj et al. [7]):

$$\frac{1}{2} + (1 + \frac{\epsilon_E}{\epsilon_I})\phi(\mathbf{x}) + \int [\frac{\partial G_0(\mathbf{x},\mathbf{y})}{\partial \vec{n}} - \frac{\epsilon_E}{\epsilon_I}\frac{\partial G_\kappa(\mathbf{x},\mathbf{y})}{\partial \vec{n}}(\mathbf{y})]\phi(\mathbf{y})d\mathbf{y} - \int [G_0(\mathbf{x},\mathbf{y}) - G_\kappa(\mathbf{x},\mathbf{y})]\frac{\partial \phi(\mathbf{y})}{\partial \vec{n}(\mathbf{y})}d\mathbf{y} = \sum_k \frac{q_k}{\epsilon_I} G_0(\mathbf{x},\mathbf{z}_k) \quad (3.1)$$

$$\frac{1}{2} + (1 + \frac{\epsilon_I}{\epsilon_E})\frac{\partial \phi(\mathbf{x})}{\partial \vec{n}(\mathbf{x})} + \int [\frac{\partial^2 G_0(\mathbf{x},\mathbf{y})}{\partial \vec{n}(x)\vec{n}(y)} - \frac{\partial^2 G_\kappa(\mathbf{x},\mathbf{y})}{\partial \vec{n}(x)\partial \vec{n}(y)}]\phi(\mathbf{y})d\mathbf{y} - \int [\frac{\partial G_0(\mathbf{x},\mathbf{y})}{\partial \vec{n}} - \frac{\epsilon_E}{\epsilon_I}\frac{\partial G_\kappa(\mathbf{x},\mathbf{y})}{\partial \vec{n}}(\mathbf{y})]\frac{\partial \phi(\mathbf{y})}{\partial \vec{n}(\mathbf{y})} = \sum_k \frac{q_k}{\epsilon_I} G_0(\mathbf{x},\mathbf{z}_k) \quad (3.2)$$

3.2 Collocation Method

Because it can be difficult to solve the boundary integral equations analytically, Bajaj et al. [7] showed that it is possible to discretize the equations by representing $\phi(\mathbf{x})$ as a linear combination of basis functions ψ (by letting $\phi(\mathbf{x}) = \sum_i \phi_i \psi_i(\mathbf{x})$) and selecting collocation points $\mathbf{x}_i$ such that the equations must be satisfied at those points. They showed that the nBIE (non-derivative boundary integral equations [69, 70, 71]) system of equations assumes the following form with the collocation method

$$\frac{1}{2}\sum_j \phi_j \psi_j(\mathbf{x}_i) + \int \frac{\partial G_0(\mathbf{x}_i,\mathbf{y})}{\partial \vec{n}} \sum_j \phi_j \psi_j(\mathbf{y})d\mathbf{y} - \int G_0(\mathbf{x}_i,\mathbf{y}) \sum_j \partial \phi_j \psi_j(\mathbf{y})d\mathbf{y} = \sum_k \frac{q_k}{\epsilon_I} G_0(\mathbf{x}_i,\mathbf{z}_k) \quad (3.3)$$

$$\frac{1}{2}\sum_j \phi_j\psi_j(\mathbf{x}_i) + \int \frac{\partial G_k(\mathbf{x},\mathbf{y})}{\partial \vec{n}} \sum_j \phi_j\psi_j(\mathbf{y})d\mathbf{y} - \int \frac{\epsilon_I}{\epsilon_E} G_\kappa(\mathbf{x},\mathbf{y}) \sum_j \partial\phi_j\psi_j(\mathbf{y})d\mathbf{y} = 0 \quad (3.4)$$

We want to solve these equations for the potentials ϕ, but as Bajaj et al. [7] pointed out, we must first parametrize the surface and use various quadrature rules to evaluate the surface integrals.

3.3 Construction of the Molecular Surface and Surface Parametrization

As Bajaj et al. [7] mentioned, *MolEnergy* constructs the surface of a protein by first obtaining the PDB file of the protein from the PDB database [16]. The PDB files contain the positions and radii of the atoms that compose the bio-molecules. Bajaj et al. [7] pointed out that for *MolEnergy*, "We utilize the molecular surface constructed in [9, 10]; the surface is generated by constructing a Gaussian density function for the atom based on atomic positions and radii, evolving this function according to a variational formulation and then considering a level-set of this function. For the resulting surface, a triangular mesh with surface normal vectors at the vertices is constructed using a dual contouring method [73]."

They also mention that *MolEnergy* then parametrizes the surface using the algebraic spline molecular surface (ASMS) [74] method, so that the surface is composed of algebraic patches (A-patches) [8].

They also note that the system of equations (equations (3.3) and (3.4)) is then solved using the GMRES method that is part of the scientific computing software package PETSc (Portable, Extensible Toolkit for Scientific Computation) [13] and the

algorithms provided in KIFMM3d (Kernel-Independent Fast Multipole 3d Method) [67]

3.4 Polarization Energy Computation

Once we compute ϕ, we can obtain the polarization energies. Once we know ϕ_{rxn}, we can obtain G_{pol} using the following equations as stated by Bajaj et al. [7]:

$$\phi_{rxn}(\mathbf{x}) = \phi(\mathbf{x}) - \phi_{gas}(\mathbf{x}) \tag{3.5}$$

$$\phi_{rxn}(\mathbf{z}) = \int (\frac{\epsilon_E}{\epsilon_I}\frac{\partial G_k(\mathbf{z},\mathbf{y})}{\partial \vec{n}(\mathbf{y})} - \frac{\partial G_0(\mathbf{z},\mathbf{y})}{\partial \vec{n}(\mathbf{y})})\phi(\mathbf{y}) + (G_0(\mathbf{z},\mathbf{y}) - G_k(\mathbf{z},\mathbf{y}))\frac{\partial \phi(\mathbf{y})}{\partial \vec{n}(\mathbf{y})}d\mathbf{y} \tag{3.6}$$

and

$$G_{pol} = \int \phi_{rxn}(\mathbf{z}) \sum_k q_k \delta(\mathbf{z} - \mathbf{z}_k) d\mathbf{z} = \frac{1}{2}\sum_k \phi_{rxn}(\mathbf{z}_k) q_k \tag{3.7}$$

where G_{pol} is the polarization energy, G_0 and G_k are Green's kernels, ϵ_E and ϵ_I are the dielectric coefficients in the exterior and interior of the biomolecule respectively, ϕ_{gas} is the potential of the biomolecule in a gas, and ϕrxn is the difference in the potential of the biomolecule between being in solution and gas

4 Molecular Configurations Using Low-Discrepancy Pseudo-random Number Generator

Another theoretical aspect of our project that is relevant for discussion is the topic of generating molecular configurations due to the vibrations of atoms in biomolecules and the rotations of side-chains. We discuss the theory behind this because one aspect of our project involved modifying *MolEnergy* so that it could incorporate molecular configurations due to atomic vibrations and compute the potentials and energies of dynamic configurations of atoms. In this chapter we first describe how molecular configurations occur due to vibrations and conformations due to rotations. We then discuss how to generate low-discrepancy samples from the product space $\mathbf{SO}(3)^n$.

4.1 Molecular Configurations Due to Vibrations and Rotations

Baldwin and Rose [14] determined that molecular conformations are due to vibrations of atoms and rotations of side-chains in biomolecules. To consider the vibrational movements of atoms, we must first look at their temperature factor values. The temperature factor is defined as $B = 8\pi < u^2 >$, where u is a measure of the uncertainty of the position of the atom which is derived experimentally from x-ray crystallography [53]. We can get the temperature factor values of all the atoms in the biomolecules by obtaining the PDB files of biomolecules, which also lists the positions and charges of every atom in the molecule. The temperature factor value tells us how likely certain atoms are to become dynamic and stable and be involved in vibrational motions. However, it has been determined that modeling molecular configurations due to vibrational motions does not match reality as well as modeling conformations

due to rotations of side-chains [I need to find the source]. Modeling vibrations of atoms is particularly inaccurate for larger vibrations because atoms could collide and bonds could break. As a result, most studies treat conformations solely due to side-chain rotations [21]. Biomolecules can taken on multiple conformational states due to rotations [68, 40, 15, 30].

To model molecular conformations due to bond rotations, most studies assume that the rotations are solely due to torsional angle rotations, and that bond lengths and rotations are irrelevant. The probability that a given biomolecule that has certain side-chains rotate producing a resulting molecular conformation is given by the Boltzmann factor, which is determined by the value in the change of energy of the resulting conformation. Studies have attempted to model conformations due to rotations with either Monte Carlo methods or MD simulations. Although MD can describe the rotational conformations for short timescales in the nanosecond to submicrosecond range [44], it has problems capturing the conformations for longer time-scales, such as for microsecond to millisecond time scales. As a result, many recent studies focus on using Monte Carlo techniques to simulate how the rotational conformations occur in biomolecules [72, 29, 30].

Although we would like *MolEnergy* to generate molecular conformations based on sampling the torsional angles, we first implement sampling based on Cartesian coordinates of atoms using code from *MolSurf* into *MolEnergy*. This is sufficient because we first just want*MolEnergy* to be able to handle any kind of molecule that consists of dynamic atoms. Even though the generation of conformations due to torsional angle sampling is more realistic and meaningful, we have to leave it to future research due to time constraints and the fact that we already have code from *MolSurf* that samples the Cartesian coordinates of atoms. We leave it to Chapter 5

to discuss the details about how the pseudo-random number generator from *MolSurf* generates samples of Cartesian coordinates to simulate dynamic atoms.

4.2 Low-Discrepancy Sampling of Cartesian Coordinates

Because we modified *MolEnergy* to utilize the $PDBUQ$ library from *MolSurf*, which uses a low-discrepancy pseudo-random number generator, to simulate the molecular configurations due to vibrational movements of atoms, we shall now discuss the theory behind low-discrepancy pseudo-random number generators. Discrepancy for a random number generator is a measure of how uniformly the generator samples the points in a given space [1, 19, 49, 50]. As mentioned in those papers, the lower the discrepancy, the greater the points sampled from the generator fill the given space uniformly. Bajaj et al. [6] determined that there is a way to efficiently sample the points from $\mathbf{SO}(3)^n$ for a biomolecule consisting of n residues.

5 Modifications to *MolEnergy*

5.1 Implementing an Automated Process

We modified the code in *MolEnergy* so that it implements functions from *MolSurf*. Originally, *MolEnergy* takes the PQR as input file, constructs a molecular surface through a level-set formulation which is then approximated with a triangular mesh and outputted as a RAWN file. Then, it uses that RAWN file to parametrize the surface with the algebraic spline construction with quadrature points to create a QUAD file. Finally, that QUAD file, which contains the quadrature points of the discretization of the molecular surface, is inserted into the part of the code that solves the linearized Poisson-Boltzmann equation. This is all file-based, but we wanted to reduce the need to write files. Although we still want *MolEnergy* to generate a molecular surface with quadrature points, our goal is to modify the *MolEnergy* application so that it does not have to generate these RAWN and QUAD files.

We wanted to modify it so that it takes the PQR input file, implements the *MolSurf* function $getMolecularSurface$ which converts the PQR file into a molecular surface and then implement the *MolSurf* function *aSplineQuad* to compute quadrature points using the algebraic spline construction.

A more detailed analysis of the changes to the *MolEnergy* is code is described as follows: We first examine the *computePotentialCommandLine* function in the PBLinear.cpp file in *MolEnergy*. The *computePotentialCommandLine* is for the potential computation in the interior of the molecule. First, we will add the following lines:

```
int size =128;
Geometry *geometry = MolecularSurface::getMolecularSurface(molecule,size
    ).
```

The variable *size* specifies the grid size. The $getMolecularSurface$ function was originally called in the $surfaceUsingAdaptiveGrid$ function in the *MolSurf* code in the $MolSurf.cpp$ file, while $getMolecularSurface$ was defined in the MolecularSurface.cpp file. The *molecule* data is first obtained by loading in the PDB/PQR input file, and then the $getMolecularSurface$ function uses the information from *molecule* to generate the solvent-excluded molecular surface using the adaptive grid algorithm.

This new surface is stored as a pointer to the *geometry* class. We then modify the *loadRawn* function so that it takes this *geometry* pointer as an input parameter instead of the RAWN file and change the name of this function to *loadRawnAutomated*. This *loadRawnAutomated* function first obtains the number of vertices and triangles from this *geometry* pointer and then the xyz coordinates, normals, and indices of the vertices from the *geometry* pointer. This is in contrast to how *loadRawn* determined these values by reading the input RAWN file line-by-line. The changes in the code are below:

```
PetscErrorCode PBSolver::loadRawnAutomated(Geometry *geometry,
    PBRawnAndQuadInfo * surf){

  surf->NumVertices = geometry->m_NumTriVerts;
  //geometry->m_NumTriVerts, is from Aspline.cpp
  surf->NumTriangles = geometry->m_NumTris;
  for (int i=0; i<surf->NumVertices; i++) {
    double xx,yy,zz;
    xx  = geometry->m_TriVerts[i*3+0];
    //From ASpline::ASpline(Geometry* geometry) from Aspline.cpp in
        MolSurf
    yy  = geometry->m_TriVerts[i*3+1];
    zz  = geometry->m_TriVerts[i*3+2];
    surf->rawn_x.push_back(xx);
```

```
    surf->rawn_y.push_back(yy);
    surf->rawn_z.push_back(zz);
    xx = geometry->m_TriVertNormals[i*3+0];
    yy = geometry->m_TriVertNormals[i*3+1];
    zz = geometry->m_TriVertNormals[i*3+2];
    surf->rawn_nx.push_back(xx);
    surf->rawn_ny.push_back(yy);
    surf->rawn_nz.push_back(zz);
  }

//  ...
  for (int i=0; i<surf->NumTriangles; i++) {
    int a,b,c;
    a = geometry->m_Tris[i*3+0];//these are indices of the vertices of
        the triangles.
    b = geometry->m_Tris[i*3+1];
    c = geometry->m_Tris[i*3+2];

    surf->rawn_t1.push_back(a);
    //rawn_t1, rawn_t2 and rawn_t2 store the indices of the vertices of
        the triangles.
    surf->rawn_t2.push_back(b);
    surf->rawn_t3.push_back(c);

    surf->vert2tris[a].push_back(i);
    surf->vert2tris[b].push_back(i);
    surf->vert2tris[c].push_back(i);
    /*vert2tris - given the index of a vertex, you get back a list of
    the triangles that the vertex is part of.*/
  }
```

whereas the original *loadRawn* was written as:

```
PetscErrorCode PBSolver::loadRawn(char * fname, PBRawnAndQuadInfo * surf
    ) {

  ifstream fin(fname);
  fin >> surf->NumVertices >> surf->NumTriangles;
  for (int i=0; i<surf->NumVertices; i++) {
    double xx,yy,zz;
    fin >> xx >> yy >> zz;
    surf->rawn_x.push_back(xx);
    surf->rawn_y.push_back(yy);
    surf->rawn_z.push_back(zz);
    fin >> xx >> yy >> zz;
    surf->rawn_nx.push_back(xx);
    surf->rawn_ny.push_back(yy);
    surf->rawn_nz.push_back(zz);
  }

  ...
  for (int i=0; i<surf->NumTriangles; i++) {
    int a,b,c;
    fin >> a >> b >> c;
    surf->rawn_t1.push_back(a);
    surf->rawn_t2.push_back(b);
    surf->rawn_t3.push_back(c);

    surf->vert2tris[a].push_back(i);
    surf->vert2tris[b].push_back(i);
    surf->vert2tris[c].push_back(i);
  }
```

```
//..
}
```

Similarly, we modify *computeQuad* so that it takes the *geometry* pointer as an input parameter and change the name of this function to *computeQuadAutomated*.

The first change in *computeQuadAutomated* compared to *computeQuad* occurs when *computeQuadAutomated* uses the *aSplineQuadAutomated* function to generate the Gaussian quadrature points for an ASMS, with 1 point per triangle, from a triangular mesh

```
aSplineQuadAutomated(geometry, "gaussian", 1);
```

This function was originally called in $MolSurf$ as follows:

```
aSplineQuad(options->mesh_fname, "gaussian", 1, options->quad_fname);
```

So *aSplineQuadAutomated* serves the same purpose as *aSplineQuad*, but we made some modifications. We modified its parameters so that it automatically converts a mesh to a surface containing quadrature points instead of relying on user input in the command prompt to specify the types of files it inputs and outputs. We changed the parameters of the function so that it takes $Geometry * input$, $string\ type$, and $int\ numOfPts$ as its parameters, whereas it had originally taken user input in the command prompt. By setting *type* to *gaussian* and $numOfPts$ to 1, we specify that this function generate Gaussian quadrature points of order 1 on every triangular patch of a given A-spline surface. In *MolSurf*, *aSplineRaw* is called if the user specified "raw" in the command prompt and *aSplineRawQuad* was called if the user specified "raw+quad" in the command prompt. These functions compute the A-spline surface with normal vectors at the vertices in the .RAWN format.

The implementation details for *aSplineQuadAutomated* are as follows:

```
{
        ASPLINE::ASpline* aspline = new ASPLINE::ASpline(input);
        aspline->Triangle_Bezier_Patch_Nodes(type.c_str(), numOfPts);
        return true;
}
```

Whereas the function details for *aSplineQuad* in $MolSurf$ is seen below:

```
{
        if(argc != 7)
        {
                usageASplineQuad();
                return false;
        }
        FILE* input     = fileRead(argv[3]);
        string type     = string(argv[4]);
        int numOfPts    = atoi(argv[5]);
        FILE* output    = fileWrite(argv[6]);
        ASpline* aspline = new ASpline(input);
        aspline->Triangle_Bezier_Patch_Nodes(type.c_str(), numOfPts);
        ..
}
```

After changing *aSplineQuad* to *aSplineQuadAutomated*, in order perform the surface triangulation and obtain the A-spline, the original PB program called the function *getAspline* as follows, where 'fname' refers to the file it reads.

```
surf->aspline = parser->getASpline(fname);
```

This *getASpline* function takes the *rawn* file as input and stores the quadrature points after it is implemented. The *getASpline* function calls the function $SurfaceTriangulation$ to determine quadrature point calculations.

```
TriElement* GeometryParser::getASpline(const char* fname) {
```

```
 return getASpline(fname, 0);
}

TriElement* GeometryParser::getASpline(const char* fname, int permu)
{
 FILE* fp = fopen(fname, "r");
 //..
 TriElement* surface = new ASpline();
 return SurfaceTriangulation(surface,fp,permu);
}
```

However, since the new *computeQuadAutomated* function reads the *geometry* pointer instead of an external file, we have to modify the call from *getASpline* to *getASplineAutomated*, as shown below:

```
surf->aspline = parser getASplineAutomated (geometry);
```

We also modify the *getASplineAutomated* function as follows:

```
TriElement* GeometryParser::getASplineAutomated(Geometry* geometry) {
 return getASplineAutomated(geometry, 0);
}

TriElement* GeometryParser::getASplineAutomated(Geometry* geometry, int
permu)
{
 TriElement* surface = new ASpline();
 return SurfaceTriangulationAutomated(surface,geometry,permu);
}
```

The new $SurfaceTriangulationAutomated$ function is nearly identical to the original $SurfaceTriangulation$ function, except for the changes noted below:

```
TriElement* GeometryParser::SurfaceTriangulationAutomated(TriElement*
```

```
    surface,
Geometry *geometry, int permu)
{
    int i,ii,jj,kk,i1,i2,i3,minindex,nscan;
    double x,y,z,f,nx,ny,nz,red,green,blue;
    int howmany = 6;
    //this value was 6 if the original SurfaceTriangulationAutomated
    // read a .rawn file as input
    normalFlag = 1;
    surface->normalFlag = true;

    surface->numbpts = geometry->m_NumTriVerts;
    //geometry->m_NumTriVerts, is from Aspline.cpp
    surface->numbtris = geometry->m_NumTris;
    //...

    for (i = 0 ; i < surface->numbpts; i++)
    {
        x = geometry->m_TriVerts[i*3+0];
        //From ASpline::ASpline(Geometry* geometry) from Aspline.cpp in
            MolSurf
        y = geometry->m_TriVerts[i*3+1];
        z = geometry->m_TriVerts[i*3+2];
        nx = geometry->m_TriVertNormals[i*3+0];
        ny = geometry->m_TriVertNormals[i*3+1];
        nz = geometry->m_TriVertNormals[i*3+2];
    //...
    }

    for (i = 0 ; i < surface->numbtris; i++)
    {
```

```
            ii = geometry->m_Tris[i*3+0];
            jj = geometry->m_Tris[i*3+1];
            kk = geometry->m_Tris[i*3+2];
            //..

            Face* facet = new Face();
            facet->Index[0] = i1;
            facet->Index[1] = i2;
            facet->Index[2] = i3;
            //..
            surface->addFacet(facet);
            //..
            surface->m_Faces[0]->Normal[0] = 0;
            surface->m_Faces[0]->Normal[1] = 0;
            surface->m_Faces[0]->Normal[2] = 0;
            //..
        }
}
```

There were also problems with this function in that the normals of the newly created faces were not initially set to 0, as in the original *SurfaceTriangulation* function. I added the necessary lines at the end of the code above.

The original *SurfaceTriangulation* function was written as:

```
TriElement* GeometryParser::SurfaceTriangulation(TriElement* surface,
FILE *fp, int permu)
{
    int i,ii,jj,kk,i1,i2,i3,minindex,nscan;
    double x,y,z,f,nx,ny,nz,red,green,blue;
    howmany = ReadHowmanyComponents(fp);
    //this value was 6 if the input file is a .rawn file
```

```
//..
if (howmany == 6) {normalFlag = 1; colorFlag = 0;}
//..
(normalFlag == 0) ? surface->normalFlag = false : surface->
    normalFlag = true;

rewind(fp);//Sets the position indicator associated with fp to the
    beginning of fp.
fscanf(fp,"%d %d\n",&surface->numbpts, &surface->numbtris);
//...

if (normalFlag == 1)
{
       if (howmany == 6 )
           nscan = fscanf(fp,"%lf %lf %lf %lf %lf %lf\n",
                          &x, &y, &z, &nx, &ny, &nz);
}

for (i = 0 ; i < surface->numbtris; i++)
{
    fscanf(fp,"%d %d %d\n", &ii, &jj, &kk);
    //..

    Face* facet = new Face();
    facet->Index[0] = i1;
    facet->Index[1] = i2;
    facet->Index[2] = i3;
    //..
    surface->addFacet(facet);
    //..
```

```
    }
}
```

The Duffy quadrature points are also obtained by using the *getAsplineAutomated* function to read in the *geometry* pointer as opposed to an external file

In addition to adding the relevant functions from *MolSurf* to *MolEnergy*, we also had to include the relevant header files to the PBLinear.cpp file in *MolEnergy*:

```
#include "Geometry/Geometry.h"
#include "Blurmaps/MolecularSurface.h"
#include "Utility/utility.h"
#include "LBIE_lib/LBIE_Mesher.h"
#include "LBIE_lib/Geoframe.h"
#include "TriElement/trielement.h"
#include "ASpline/ASpline.h"
```

5.2 Implementing a Pseudo-Random Number Generator with Parallel Computing

After completing the automation process, our next goal is to modify the *MolEnergy* application so that it can generate new molecules containing certain atoms that become dynamic with new updated positions and then calculate the potentials and energies of those configurations in parallel. We utilize the random sampler code from the $PDBUQ$ library of $MolSurf$, which uses a low-discrepancy pseudo-random Monte Carlo generator to generate new copies of the atoms and update their positions. The magnitude of the change in the positions of the atoms is determined by the temperature factor values of the atoms, which is contained in the PQR file for each molecule.

Using the newly created automated code in *PBLinear.cpp*, we implement the random sampling of the atomic positions in the *computePotentialCommandLine* function, as seen in the code below:

```
TempFactorUQ* tfuq = new TempFactorUQ(atomList);


tfuq->getNextSample(atomList);
```

We verified that this code is correctly producing a correct sampling by printing out the values of the positions of each atomic before and after using *getNextSample*. We confirmed that the values were different and the atoms with larger temperature factor values had greater changes than atoms with smaller temperature factor values. The PQR file is then converted into a *molecule* at

```
GOALoader* gLoader = new GOALoader();
PDBParser::GroupOfAtoms* molecule = gLoader->loadFile( options->
    pqr_fname ,
true);
```

Then, this *molecule* is converted into a vector of *Atom*s, *atomList*, from the *FlattenGOA* function. Then, we use *TempFactorUQ* and *getNextSample* to update *atomList* based on the temperature factor values for each atom. We then continue with the process described above to generate the surface mesh, quadrature points, potential values, and energy.

The next step in the code is to implement a loop such that it repeats the process of generating an updated *atomList*, running *loadRawnAutomated*, running *computeQuadAutomated*, computing the potential values, and computing the energy values multiple times. In order to improve the efficiency of the code, we will implement parallel computing using MPI and PETSc.

First, we have to test the code to make sure it can run the loop multiple times

with no problems before proceeding to implementing parallel computing. We first set the loop so that it runs about 16 times. We set a loop enclosing the *getNextSample* function and other functions as follows

```
for (int j=1; j<17; j++)
{
    tfuq->getNextSample(atomList);
    Geometry* geometry = surfaceUsingAdaptiveGrid(molecule, size,width);
    loadRawnAutomated(geometry,&molSurf);
    computeQuadAutomated(geometry,options,&molSurf);
    PetscErrorCode rval = computePotential(options);
    PrintBoundaryPotential(options, pot_fname.c_str(),
        mult_fmm2->getNumTarget(),phi);
    if (options->rawncOutput)
        PrintRawnc(NULL, options, mult_fmm2->getNumSource(),phi);
    delete mult_fmm2;
    computeEnergyUPDATED(options, atomList, j, molecule)
}
```

Now that we got this to work, we proceeded to implement parallel computing into the code. Because we wanted multiple nodes to run the *computePotentialCommandLine* line function, we had to set the MPI variables in the *runPB* function in *main.cpp* since that is where *computePotentialCommandLine* is called. We added the following lines of code to *runPB* function

```
MPI_Status status;
PetscMPIInt rank,size;
PetscInitialize(&argc2,&argv2,0,0);
MPI_Comm_size(PETSC_COMM_WORLD,&size);
MPI_Comm_rank(PETSC_COMM_WORLD,&rank);
PetscPrintf(PETSC_COMM_WORLD,"Number of processors = %d, rank = %d\n",
    size,rank);
```

```
PetscSynchronizedPrintf(PETSC_COMM_WORLD,"synchronized rank = %d\n",rank
    );
PetscSynchronizedFlush(PETSC_COMM_WORLD);
MPI_Barrier(PETSC_COMM_WORLD);
```

We also had to modify *computePotentialCommandLine* so that it take the different nodes as parameter values:

```
PBOptions options;
PBSolver* pb = new PBSolver();
pb->getOptionsFromFile(&options,argv[0]);
if (options.computePotential) {
    pb->computePotentialCommandLine(&options, rank+1);
}
```

To ensure that each node only performed one step in the loop in *computePotentialCommandLine* once, we had to make the following change to *computePotentialCommandLine*

```
for (int j=1; j<21; j++)
{
    tfuq->getNextSample(atomList);
    if (j%v==0)
    {
        Geometry* geometry = surfaceUsingAdaptiveGrid(molecule, size,
            width);
        //...
        computeEnergyUPDATED(options, atomList, j, molecule)
    }
}
```

After trying to run this code, we were getting new problems with the *setRHS* function called in *computePotential*. The problem was with the line

```
VecSetSizes(b, PETSC_DECIDE, NumTrg)
```

This was a problem because we wanted each node to use $NumTrg$ as the local size and let PETSc decide on the global size, since we could use a variable number of nodes and thus total number of target points. We changed this to

```
VecSetSizes(b, NumTrg, PETSC_DECIDE)
```

After the parallel version of the code worked, we wanted to run the loop in *computePotentialCommandLine* about 20 times. While this loop worked with no problems, there was a problem in that the program crashed after running about 16 times. Using valgrind, we determined that there were numerous memory leaks in the serial version of the program. We determined that the memory leaks occurred because there were pointers being created but not deleted. In the $surfaceTriangulationAutomated$ function in *geometryParser.cpp*, the memory leaks occurred at

```
Vertex* point = new Vertex();
Face* facet = new Face();
```

These pointers were used throughout the program so it would be difficult to track down when they are no longer used in the program and therefore when it is safe to delete them. We decided to fix this problem by using $std :: shared\_ptr$ in C++11, as it automatically deletes the pointer when it goes out of scope. Therefore, the pointers above were converted to

```
std::shared_ptr<Vertex> point(new Vertex() );
std::shared_ptr<Face> facet(new Face());
```

Below is the rest of the changes we made using $std :: shared\_ptr$. The changes we made to the functions in *geometryParser.cpp* are as follows

```
std::shared_ptr<TriElementNS::TriElement> GeometryParser::getASplineAUTO
    (Geometry* geometry)
{
    return getASplineAUTO(geometry, 0);
```

```
}

std::shared_ptr<TriElementNS::TriElement> GeometryParser::getASplineAUTO
    (Geometry* geometry, int permu)
{
    std::shared_ptr<TriElementNS::TriElement> surface(new ASpline() );
    SurfaceTriangulationAutomated(surface,geometry,permu);
    return surface;
}

void GeometryParser::SurfaceTriangulationAutomated(std::shared_ptr<
    TriElementNS::TriElement> &surface, Geometry* geometry, int permu)
```

We changed $SurfaceTriangulationAutomated$ so that it accepts shared pointer instead of a raw pointer and so it does not return anything, whereas it had originally returned $surface$, which was a pointer to $TriElement$.

In *trielement.h*, we modified the variables $m\_Vertices$ and m_Faces so that they are vectors of shared pointers instead of raw pointers. In addition, we had to modify $addPoint$, $addFacet$, and get_aspect_ratio functions so that they accepted shared pointers instead of raw pointers

In *trielement.cpp*, we modified the $NormalizeNormal$ and $calculateAspectRatio()$ function so that it used the shared pointers in $m_Vertices$ instead of raw pointers

We also modified the $Triangle\_Patch$ and $Triangle\_Patch\_Nodes$ functions in *bezier.cpp* in the $TriElement$ directory so that they used shared pointers to $Vertex$ instead of raw pointers. We made the same changes to the same functions in *bezier.cpp* in the *ASpline* directory, except it also contained a function $Triangle\_Patch\_with\_Quadrature$ that used raw pointers to $Vertex$, so we changed those raw pointers to shared pointers. In the *ASpline* directory, we also made changes in the *ASpline.h* and

ASplinePrimitives.h files so that they used shared pointers to $Vertex$ and $Face$ instead of raw pointers

Finally, in $PBLinear.h$ we changed *aspline* and $asplineDuffy$ so that they were shared pointers to $TriElement$ instead of raw pointers. In $PBLinear.cpp$ we also changed *aSplineQuadAutomated* so that *aspline*, which calls $Triangle\_Bezier\_Patch\_Nodes$ is a shared pointer to *ASpline* instead of a raw pointer

6 Results and Discussion

6.1 Implementing an Automated Process

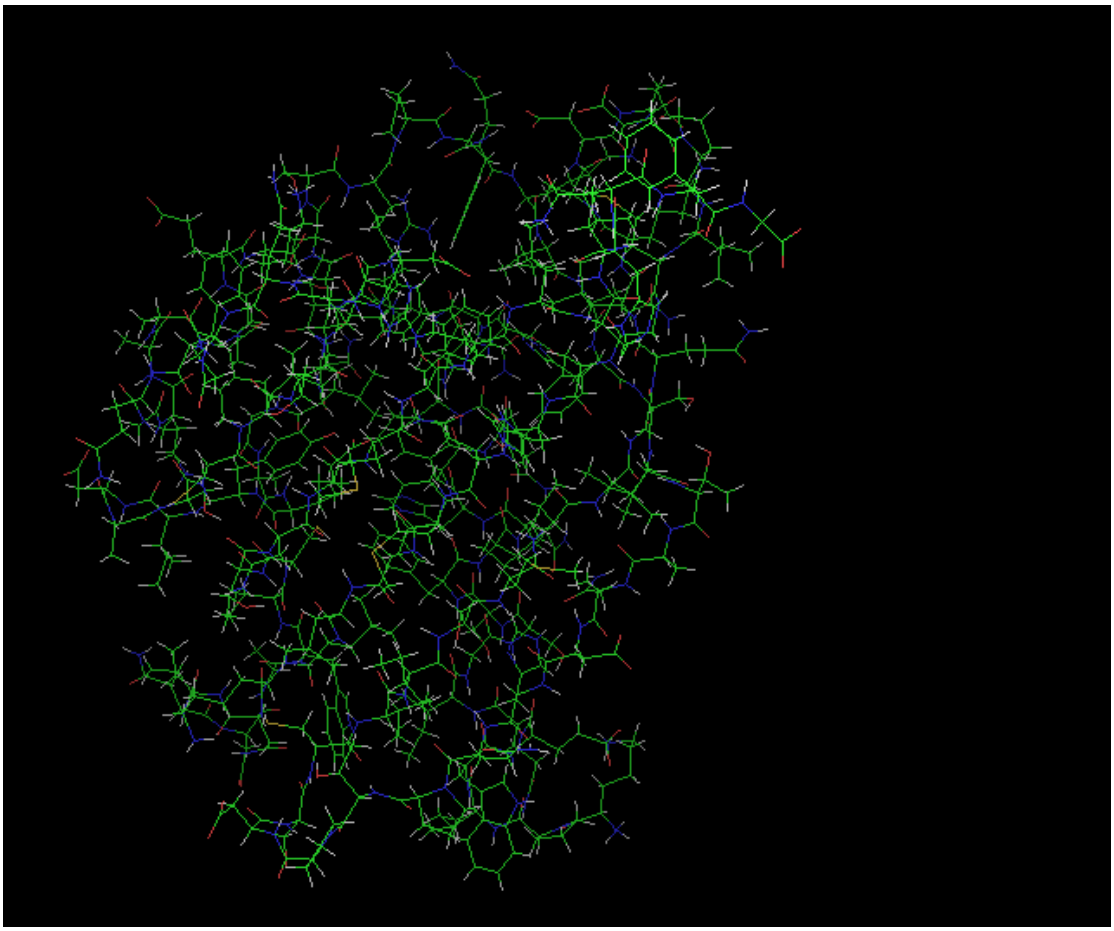

Figure 6: Diagram of 1A2K protein with just its chain A

We wanted to verify that changing the *computePotentialCommandLine* function so that it uses an automated process as opposed to reading in input files would not severely affect the results generated from the original code. We verified this by checking if our new code generated similar values for the number of triangles and vertices in the surface mesh and the final potential and energy values compared with the original code. We checked the new code on a protein-ligand binding example by

using the 1A2K.pdb as an input file and determined that the mesh generated by the updated PB code on chain A (nuclear transport factor 2) of 1A2K.pdb contained 21,680 vertices and 43,356 triangles with an energy of -765 kcal/mol, whereas the original $MolSurf$ also generated a mesh containing about 22,000 vertices and 43,300 triangles with an energy of -772 kcal/mol, when using only chain A of the 1A2K molecule with $size$=128.

6.2 Implementing a Pseudo-Random Number Generator with Parallel Computing

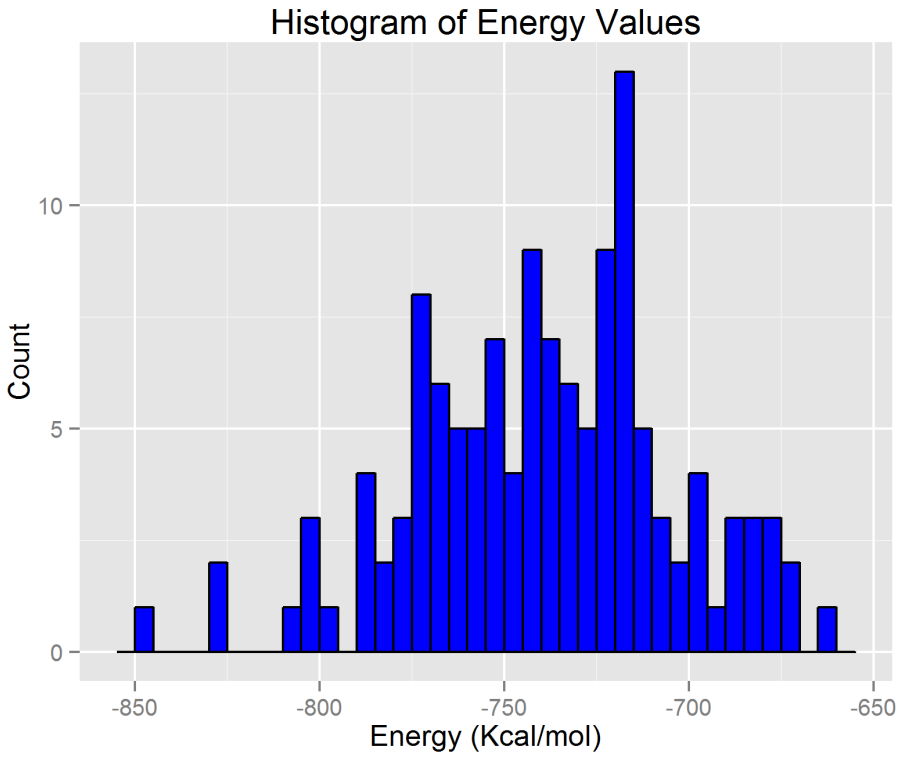

Figure 7: Histogram of the energy values of the ensemble of configurations generated from vibrations for 1A2K. The energy values ranged from -662.33 kcal/mol to -848.01 kcal/mol with a mean value of -739.18 kcal/mol and a standard deviation of 35.21 kcal/mol

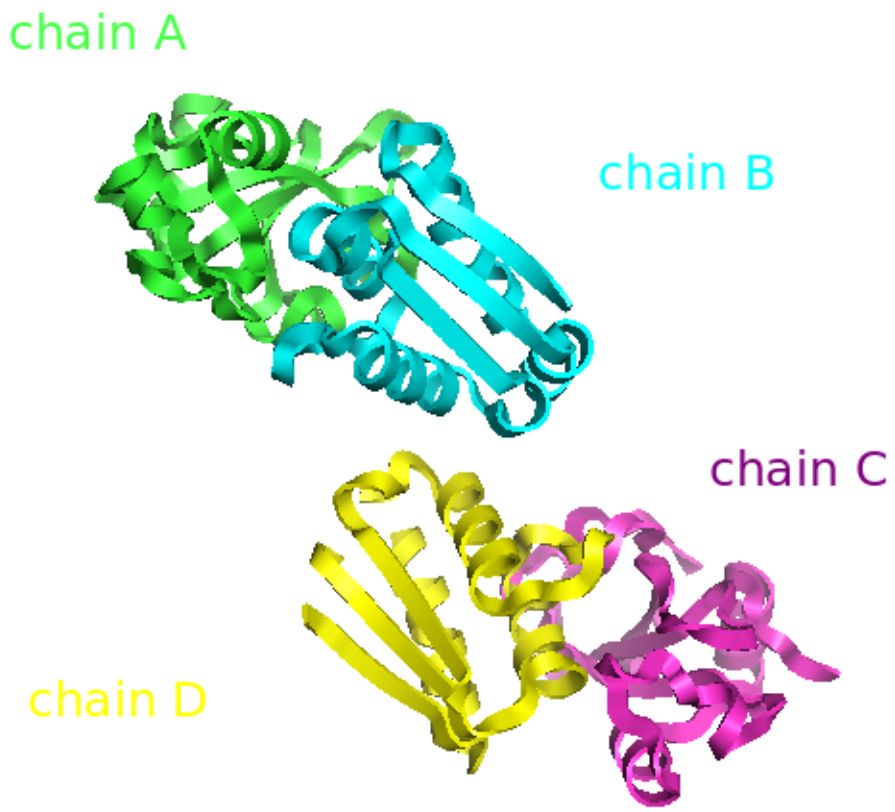

Figure 8: Cartoon representation of the 2ZA4 protein consisting of Barnase and Barstar bounded together

As a result of the changes we made to *MolEnergy* so that it could generate new configurations and utilize parallel computing, by again using the 1A2K.pdb as a test protein-ligand binding, we were able to generate 128 different atomic configurations and compute their energy values in parallel. These 128 different configurations is an ensemble average of the possible configurations that the 1A2K protein could undergo as a result of atomic vibrations

The energy values ranged from -662.33 kcal/mol to -848.01 kcal/mol with a mean value of -739.18 kcal/mol and a standard deviation of 35.21 kcal/mol for 1A2K. This seems reasonable because the energy value of 1A2K using the original *MolEnergy*

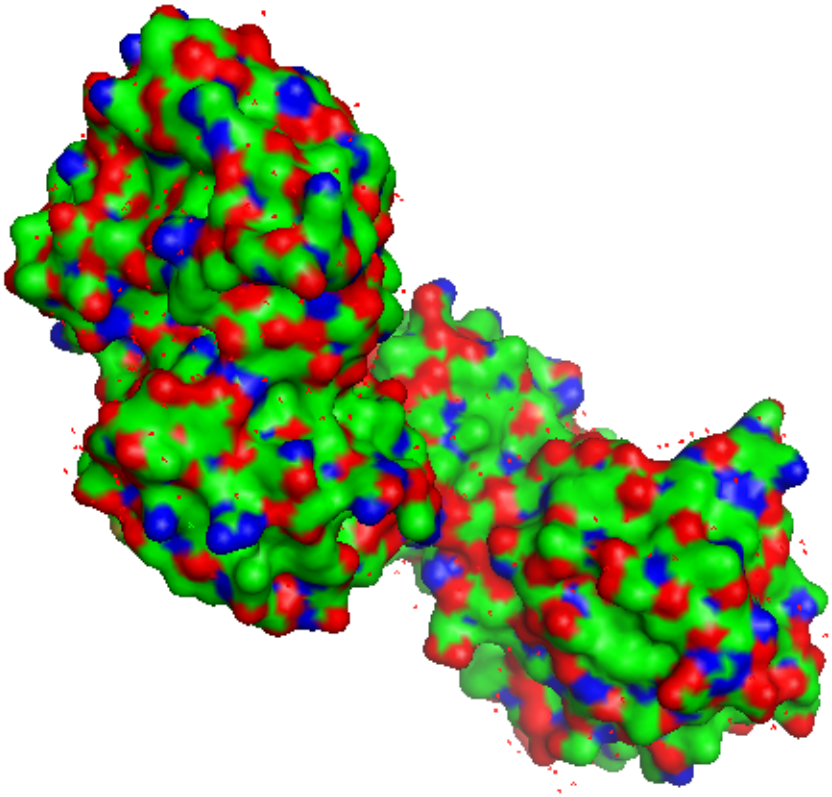

Figure 9: Surface view of the atomic structure of the 2ZA4 protein. C atoms are the green atoms, O atoms are in red, and N atoms are in blue

code was -772 kcal/mol, so the mean value of -739.18 kcal/mol is within 5% of it.

We also attempted to compute the binding energy of another protein. We used the 2ZA4 protein, which consists of chains A and C belonging to the Barnase protein binded to the Barstar protein consisting of chains B and D. We were able to generate 132 configurations of 2ZA4 with a mean energy value of -655.50 kcal/mol, 132 configurations of Barnase with a mean energy of -649.59 kcal/mol, and 156 configurations of Barstar with a mean energy of -722.19 kcal/mol. This results in a binding energy of 716.18 kcal/mol

Although generating samples of the Cartesian coordinates of atoms based on their

temperature factor values is good for modeling small vibrational motions and is sufficient for many cases, it does not preserve the molecular bond constraints if larger motions are sampled that way. Another problem with modeling vibrational motions is that atoms could collide. Therefore, we want to further modify *MolEnergy* so that it takes into account the uncertainty in the flexibility and rotations. That is, we wish to modify *MolEnergy* so it cagenerate molecular conformations by sampling internal torsion angles using the low-discrepancy pseudo-random number generator. We then want to calculate the potentials and energies of these conformations in parallel. We leave this to future work on *MolEnergy*

7 References

[1] *Discrepancy as a quality measure for sample distributions*. Elsevier Science, 1991.

[2] MolSurf, 2011. http://www.cs.utexas.edu/ bajaj/cvcwp/?page_id=281.

[3] Protein Structure, 2011. http://www.nature.com/scitable/topicpage/protein-structure-14122136.

[4] A. Papazyan A. Warshel. Electrostatic effects in macromolecules: fundamental concepts and practical modelling. *Curr Opin Struct Biol*, 8:211–217, 1998.

[5] Y. Bai, V. Chu, J. Lipfert, V. Pande, D. Herschlag, and S. Doniach. Critical assessment of nucleic acid electrostatics via experimental and computational investigation of an unfolded state ensemble. *J. Am. Chem. Soc.*, 130(37):12334–12341, 2008.

[6] C. Bajaj, A. Bhowmick, E. Chattopadhyay, and D. Zuckerman. On low discrepancy samplings in product spaces of motion groups. 2014. Manuscript.

[7] C. Bajaj, S. Chen, and A. Rand. An efficient higher-order fast multipole boundary element solution for poisson-boltzmann-based molecular electrostatics. *SIAM J. Sci. Comput.*, 33(2):826–848, 2011.

[8] C. Bajaj and G. Xu. A-Splines: Local interpolation and approximation using G_k- continuous piecewise real algebraic curves. *Comput. Aided Geom. Des.*, 16:557–578, 1999.

[9] C. L. Bajaj, G. Xu, and Q. Zhang. Higher-order level-set method and its application in biomolecule surfaces construction. *J. Comput. Sci. Technol.*, 23(6):1026–1036, 2008.

[10] C.L. Bajaj, G. Xu, and Q. Zhang. A fast variational method for the construction of resolution adaptive C^2-smooth molecular surfaces. *Computer methods in applied mechanics and engineering*, 198(21-26):1684–1690, 2009.

[11] N. Baker, M. Holst, and F. Wang. Adaptive multilevel finite element solution of the poisson-boltzmann equation ii. refinement at solvent-accessible surfaces in biomolecular systems. *J. Comput. Chem.*, 21:1343–1352, 2000.

[12] N. Baker, D. Sept, S. Joseph, M.J. Holst, and J.A. McCammon. Electrostatics of nanosystems: application to microtubules and the ribosome. *Proc. Natl. Acad. Sci*, pages 10037–10041, 1998.

[13] S. Balay, K. Buschelman, W. D. Gropp, D. Kaushik, M. G. Knepley, L. C. McInnes, B. F. Smith, and H. Zhang. PETSc Web page, 2001. http://www.mcs.anl.gov/petsc.

[14] R.L. Baldwin and G.D. Rose. Molten globules, entropy-driven conformational change and protein folding. *Curr. Opin. Struct. Biol.*, 23:4–10, 2013.

[15] I.N. Berezovsky, W.W. Chen, P.J. Choi, and E.I. Shakhnovich. Entropic stabilization of proteins and its proteomic consequences. *PLoS Comput Biol*, 1(4):e47, 2005.

[16] H. M. Berman, J. Westbrook, Z. Feng, G. Gilliland, TN Bhat, H. Weissig, I. N. Shindyalov, and P. E. Bourne. The Protein Data Bank. *Nucleic Acids Res.*, 28:235–242, 2000.

[17] I. Borukhov, D. Andelman, and H. Orland. Steric effects in electrolytes: A modified poisson-boltzmann equation. *Phys. Rev. Lett*, 79(435):435–438, 1997.

[18] I. Borukhov, D. Andelman, and H. Orland. Adsorption of large ions from an electrolyte solution: A modified poisson-boltzmann equation. *Electrochimi. Acta*, 46:221–238, 2000.

[19] N. Bouleau and D. Lépingle. *Numerical Methods for Stochastic Processes*. John Wiley and Sons, 1993.

[20] W. R. Bowen and A. O. Sharif. Adaptive finite element solution of the nonlinear Poisson-Boltzmann equation: A charged spherical particle at various distances from a charge cylindrical pore in a charged planar surface. *J. Colloid Interface Sci.*, 187:363–374, 1997.

[21] G.P. Brady and K.A. Sharp. Entropy in protein folding and in protein-protein interactions. *Curr. Opin. Struct Biol*, 7:215–221, 1997.

[22] L. Chen, M. Holst, and J. Xu. The finite element approximation of the nonlinear Poisson-Boltzmann equation. *SIAM J. Numer. Anal.*, 45(6):2298–2320, 2007.

[23] V. Chu, Y. Bai, J. Lipfert, D. Herschlag, and S. Doniach. Evaluation of ion binding to dna duplexes using a size-modified poisson-boltzmann theory. *Biophysical Journal*, 93(9):3202–3209, 2007.

[24] R.D. Coalson, A.M. Walsh, A. Duncan, and N. Ben-Tal. Statistical mechanics of a coulomb gas with finite size particles: A lattice field theory approach. *J. Chem. Phys*, 102:4584–4594, 1995.

[25] C. M. Cortis and R. A. Friesner. An automatic three-dimensional finite element mesh generation system for the poisson-boltzmann equation. *J. Comput. Chem.*, 18:1570–1590, 1997.

[26] C. M. Cortis and R. A. Friesner. Numerical solution of the poisson-boltzmann equation using tetrahedral finite-element meshes. *J. Comput. Chem.*, 18:1591–1608, 1997.

[27] L. David, R. Luo, and M. Gilson. Comparison of generalized born and poisson models: energetics and dynamics of hiv protease. *J Comp Chem*, 21:295–309, 2000.

[28] Brooks C III Dominy B. Development of a generalized born model parameterization for proteins and nucleic acids. *J Physical Chemistry B*, 103:3765–3773, 1999.

[29] K.H. Dubay and P.L. Geissler. Calculation of proteins' total side-chain torsional entropy and its influence on protein-ligand interactions. *J. Mol. Biol*, 391:484–497, 2009.

[30] G.D. Friedland, A.J. Linares, C.A. Smith, and T. Kortemme. A simple model of backbone flexibility improves modeling of side-chain conformational variability. *J. Mol. Biol*, 380:757–774, 2008.

[31] M. K. Gilson, K. A. Sharp, and B. H. Honig. Calculating the electrostatic potential of molecules in solution: Method and error assessment. *J. Comput. Chem.*, 9:327–335, 1987.

[32] M. Goethe, I. Fita, and J.M. Rubi. Vibrational entropy of a protein: Large differences between distinct conformations. *J. Chem. Theory Comput.*, 11(1):351–359, 2005.

[33] P. Gruchowski and J. Trylska. Continuum molecular electrostatics, salt effects, and counterion binding–a review of the poisson-boltzmann theory and its modifications. *Biopolymers*, 89(2):93–113, 2008.

[34] R. Harris. *Comparing The Poisson-Boltzmann Equation To Alternative Electrostatic Theories And Improving Stochastic Techniques For Implicit Solvent Models*. PhD thesis, 2012.

[35] M. Holst. *Multilevel Methods for the Poisson-Boltzmann Equation*. PhD thesis, University of Illinois at Urbana-Champaign, 1993.

[36] M. Holst, N. Baker, and F. Wang. Adaptive multilevel finite element solution of the poisson-boltzmann equation i. algorithms and examples. *J. Comput. Chem.*, 21:1319–1342, 2000.

[37] M. Holst and F. Saied. Multigrid solution of the Poisson-Boltzmann equation. *J. Comput. Chem.*, 14(1):105–113, 1993.

[38] A. H. Juffer, E. F. F. Botta, B. A. M. van Keulen, A. van der Ploeg, and H. J. C. Berensen. The electric potential of a macromolecule in a solvent: a fundamental approach. *J. Chem. Phys.*, 97:144–171, 1991.

[39] M. Karplus, T. Ichiye, and B.M. Pettitt. Configurational entropy of native proteins. *Biophys J*, 52:1083–1085, 1987.

[40] E. Kussell, J. Shimada, and E.I. Shakhnovich. Excluded volume in protein side-chain packing. *J Mol Biol*, 311:183–193, 2001.

[41] T. Lazaridis and M. Karplus. Effective energy functions for proteins in solution. *Curr Opin Struct Biol*, 1999.

[42] T. Lazaridis and M. Karplus. Effective energy functions for protein structure prediction. *Curr Opin Struct Biol*, 10:139–145, 2000.

[43] B. Li, P. Liu, Z. Xu, and S. Zhou. Ionic size effects: Generalized boltzmann distributions, counterion stratification, and modified debye length. *Nonlinearity*, 26(10):2899–2922, 2013.

[44] D.-W. Li and R. Bruschweiler. A dictionary for protein side-chain entropies from nmr order parameters. *J. Am. Chem. Soc.*, 131:7226–7227, 2009.

[45] C. Liao, M. Sitzmann, A. Pugliese, and M.C. Nicklaus. Software and resources for computational medicinal chemistry. *Future Medicinal Chemistry*, 3(8):1057–1085, 2011.

[46] B. Z. Lu, Y. C. Zhou, M. J. Holst, and J. A. McCammon. Recent progress in numerical methods for the Poisson-Boltzmann equation in biophysical applications. *Comm. Comput. Phys.*, 3(5):973–1009.

[47] M. Manciu and E. Ruckenstein. Lattice site exclusion effect on the double layer interaction. *Langmuir*, 18:5178–5185, 2002.

[48] A. Nicholls and B. Honig. A rapid finite difference algorithm, utilizing successive over-relaxation to solve the Poisson-Boltzmann equation. *J. Comput. Chem.*, 12:435–445, 1991.

[49] H. Niederreiter. Quasi-monte carlo methods and pseudo-random numbers. *Bulletin of the American Mathematical Society*, 84:957–1041, 1978.

[50] H. Niederreiter. Random number generation and quasi-monte carlo methods. *CBSM*, 63, 1978.

[51] Case D Onufriev A. Modification of the generalized born model suitable for macromolecules. *J Phys Chem B*, 104:3712–3720, 1999.

[52] M. Orozco and F. Luque. Theoretical methods for the description of the solvent effect in biomolecular systems. *Chem Rev*, 100:4187–4226, 2000.

[53] M. Rasheed, N. Clement, A. Bhowmick, and C. Bajaj. Quantification and visualization of uncertainties in molecular modeling. IEEE SciVis, 2015. Manuscript.

[54] A. Ritchie and L. Webb. Optimizing electrostatic field calculations with the adaptive poisson-boltzmann solver to predict electric fields at protein-protein interfaces i: Sampling and focusing. *J. Phys. Chem. B*, 117:11473–11489, 2013.

[55] B. Roux and T. Simonson. Implicit solvent models. *Biophys. Chem.*, 78(1-2):1–20, 1999.

[56] A. Sayyed-Ahmad, K. Tuncay, and P. J. Ortoleva. Efficient solution technique for solving the Poisson-Boltzmann equation. *J. Comput. Chem.*, 25:1068–1074, 2004.

[57] K. Sharp and B Honig. Applications of the finite defference Poisson-Boltzmann method to proteins and nucleic acids. *Struct. Methods: DNA Protein Complexes Proteins*, 2:211–214, 1990.

[58] A. R. J. Silalahi, A. H. Boschitsch, R. C. Harris, and M. O. Fenley. Comparing the predictions of the nonlinear poisson-boltzmann equation and the ion size-modified poisson-boltzmann equation for a low-dielectric charged spherical cavity in an aqueous salt solution. *Journal of Chemical Theory and Computation*, 6(12):3631–3639, 2010.

[59] T. Simonson. Macromolecular electrostatics: continuum models and their growing pains. *Current opinion in structural biology*, 11(2):243–252, 2001.

[60] O. Stern. *Z. Elektrochem*, 30:508–516, 1924.

[61] G. Tresset, W.C.D. Cheong, and Y.M. Lam. Role of multivalent cations in the self-assembly of phospholipiddna complexes. *J. Phys. Chem. B*, 111(51):14233–14238, 2007.

[62] V. Tsui and D. Case. Molecular dynamics simulations of nucleic acids with a generalized born model. *J Am Chem Soc*, 122:2489–2498, 2000.

[63] Y. Vorobjev and J. Hermans. Es/is: estimation of conformational free energy by combining dynamics simulations with explicit solvent with an implicit solvent continuum model. *Biophys Chem*, 78:195–205, 1999.

[64] F. Wagner and T. Simonson. Implicit solvent models: combining an analytical formulation of continuum electrostatics with simple models of the hydrophobic effect. *J Comp Chem*, 20:322–335, 1999.

[65] N. Wang, S. Zhou, P.M. Kekenes-Huskey, B. Li, and J.A. McCammon. Poisson-boltzmann versus size-modified poisson-boltzmann electrostatics applied to lipid bilayers. *The Journal of Physical Chemistry B*, 118(51):14827–14832, 2014.

[66] D. Williams and K. Hall. Unrestrained simulations of the uucg tetraloop using an implicit solvation model. *Biophys J*, 76:3192–3205, 1999.

[67] L. Ying, G. Biros, and D. Zorin. A kernel-independent adaptive fast multipole method in two and three dimensions. *J. Comput. Phys.*, 196(2):591–626, 2004.

[68] Y.B. Yu, P. Lavigne, P.L. Privalov, and R.S. Hodges. The measure of interior disorder in a folded protein and its contribution to stability. *J. Am Chem Soc*, 121:8443–8449, 1999.

[69] R. J. Zauhar and R. S. Morgan. A new method for computing the macromolecular electric potential. *J. Mol. Biol.*, 186:815–820, 1985.

[70] R. J. Zauhar and R. S. Morgan. The rigorous computation of the molecular electric potential. *J. Comput. Chem.*, 9(2):171–187, 1988.

[71] R. J. Zauhar and R. S. Morgan. Computing the electric potential of biomolecules: application of a new method of molecular surface triangulation. *J. Comput. Chem.*, 11:603–622, 1990.

[72] J. Zhang and J.S. Liu. On side-chain conformational entropy of proteins. *PLoS Comput Biol*, 2(12):e168, 2006.

[73] Y. Zhang, G. Xu, and C. Bajaj. Quality meshing of implicit solvation models of biomolecular structures. *Computer Aided Geometric Design*, page in press, 2006.

[74] W. Zhao, G. Xu, and C. Bajaj. An algebraic spline model of molecular surfaces. *Proc. ACM Symp. Solid Phys. Model.*, pages 297–302, 2007.

[75] M. Zheng, X. Liu, Y. Xu, H. Li, C. Luo, and H. Jiang. Computational methods for drug design and discovery: focus on china. *Trends in Pharmacological Sciences*, 34(10):549–559, 2013.

[76] S. Zhou, Z. Wang, and B. Li. Mean-field description of ionic size effects with nonuniform ionic sizes: A numerical approach. *Phys. Rev. E, Statistical, Nonlinear, and Soft Matter Physics*, 84, 2011.