卒業論文

第一原理原子構造緩和法の実装

関西学院大学理工学部
情報科学科 西谷研究室 8602 真嶋亨

2012 年 3 月

指導教員　西谷 滋人 教授

概 要

第一原理計算ソフトである VASP には標準で用意されている内部緩和のルーチンが搭載されている．しかし，その信頼性が問われる事例が先行研究によって報告された．そこで本研究では VASP の内部緩和ルーチンの代わりとなる VASP と連携したプログラムを作成した．エネルギーが最小となる最安定な原子位置を探索するために，多次元の中から探索方向を 1 次元に決定し最小化を行う CG 法を使用しており，その中で共役な方向を決定するために Polak-Ribiere 法を利用している．更に 1 次元の最小化には黄金分割法と放物線補間を併用する Brent 法を採用した．プログラムはスクリプト言語である Ruby を用いた．

作成したプログラムの検証方法として，VASP による結果との比較を行った．計算対象には，先行研究によってエネルギー曲面が確認されている SiO を用いた．ここで，比較には構造緩和前のモデルと，構造緩和後のモデルの両方を利用した．

構造緩和前のモデルを用いて O 原子の初期位置をボンドセンターにした場合，原子が移動することなく初期位置にとどまった．また，初期位置をボンドセンターから少し移動させたボンド上の位置にした場合，O 原子が Si 原子からの力を受け，O 原子が大きく移動した．これらの結果は最安定構造を再現しなかったものの，VASP の内部緩和ルーチンを利用した場合と一致した．そのことから，VASP の内部緩和ルーチンと本研究で作成した CG 法を用いたプログラムはアルゴリズムが同じであることがわかった．正確に最安定構造を再現出来なかった理由として大きく 2 つのことが挙げられる．1 つ目は O 原子の初期位置である．force が 0 の初期位置にした場合，O 原子は移動することはない．2 つ目は周囲の原子の緩和である．周囲の原子の緩和を行わない場合，原子同士の距離が近くなりすぎることにより O 原子にかかる force が強くなりすぎるため，原子の移動距離が大きくなりすぎるということが考えられる．

目 次

第1章　序論

1.1　SiO

1.1.1　背景

西谷研究室では，第一原理計算ソフト VASP を用いて金属や半導体材料および，格子欠陥を対象にエネルギーを計算している．それらのエネルギーは内部緩和，外部緩和の構造最適化を行い，最安定な構造を利用して求めている．ところが，先行研究によると Si 結晶中に酸素を挿入し構造最適化を行うと Si 原子同士の結合の中心であるボンドセンターが最安定となり，実験結果と矛盾した（図 1.1（a））．SiO_2 の構造は O 原子と最近接 Si 原子が約 160 度のボンド角を形成することがわかっている（図 1.1（b））ため，第一原理計算ソフト VASP に用意されている内部緩和のルーチンの信頼性が問われる結果となった．

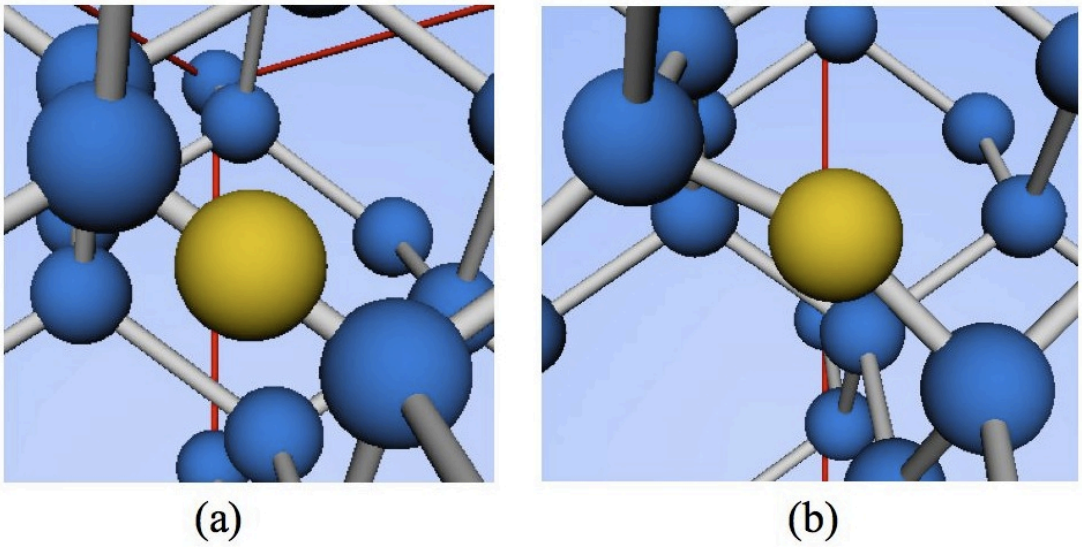

図 1.1:　（a）ボンドセンター，（b）実際の最安定位置 [3].

1.1.2 目的

1.1.1 節の VASP の内部緩和のルーチンの問題を解消するために，Ruby 言語を用いて独自の内部緩和のプログラムを作成する．本研究では対象となる原子のみに対して内部緩和をおこない，最安定位置を探索する．ここで，最安定位置とは O 原子を動かし，エネルギーが最も小さくなる位置のことである．エネルギーの最小値を求めるために，多次元の最適化に用いられる CG 法を用いる．ここで CG 法が実際に行うことは 1 次元の探索方向の決定のみで，1 次元の最適化には Brent 法を採用する．また，作成した CG 法を利用した構造緩和プログラムで正しく緩和が出来ているかの検証するために VASP の構造緩和機能による結果との比較を行う．

第2章　手法

本研究では，原子の種類だけから電子構造を求め，様々な物性の予測を可能とする第一原理計算（First principle calculations）[1] を用いた．それにより，構築した計算モデルに対して，エネルギー計算を行った．第一原理計算には VASP と呼ばれるプログラムを用いた．

2.1　VASP（Vienna Ab-initio Simulation Package）

第一原理計算ソフト VASP は平面波・擬ポテンシャル法を用いることで，高速かつ高精度に計算を進めるプログラムである．VASP の詳細は VASPManual[6] を参照．

2.2　VASP の入出力ファイル

本節では VASP の入出力ファイルについて簡単に説明する．

2.2.1　INCAR

INCAR file は計算を制御するための入力ファイルであり，計算条件を設定する．ここでは構造緩和に関するパラメータである，IBRION，ISIF を取り上げる．

IBRION

イオンのリラクゼーションの方法を決定する．パラメータと方法の組み合わせは表 2.1 に示した．

表 2.1: IBRION とリラクゼーション方法.

IBRION	方法
0	分子動力学（Molecular Dynamics）
1	準ニュートン法（quasi-Newton）
2	共役勾配法（Conjugate-Gradient）
3	最急降下法

ISIF

応力テンソルをどのように計算するかを決めるパラメータである．force や応力テンソル，セルの形や体積の変化などを考慮するかしないかを決定できる．ISIF に応じた計算手法の詳細は表 2.2 に示した.

表 2.2: ISIF による相違点.

ISIF	calculate force	calculate stress tensor	relax ions	change cell shape	change cell volume
0	yes	no	yes	no	no
1	yes	trace only	yes	no	no
2	yes	yes	yes	no	no
3	yes	yes	yes	no	no
4	yes	yes	yes	yes	yes
5	yes	yes	no	no	no
6	yes	yes	no	yes	yes
7	yes	yes	no	no	yes

表 2.3: 系の緩和に関する INCAR の記述.

緩和	IBRION	ISIF
内部緩和	2	2
外部緩和	0	6
内部緩和＋外部緩和	2	3

表 2.3 に，VASP での計算において，計算モデルの構造最適化のために行う緩和の手法に関するパラメータの数値を示した．内部緩和とは，構造の内部の原子位置を動かすことによる最適化であり，外部緩和とは，構造自体の体積やボンド角を変化させることでの最適化である．なお，内部緩和，外部緩和の両方を考慮した場合，計算量が膨大となり時間が多くかかる.

POSCAR

POSCAR file は計算モデルに関するファイルである．モデル構築において，格子ベクトルなどユニットセルの形状に関する情報や原子の位置を決定する．内部緩和を行う場合は POSCAR fail で原子ごとに移動させるかどうかの指定をすることも出来る．

OUTCAR

OUTCAR file は計算終了後に作成されるファイルであり，計算結果が出力される．このファイルには計算モデルの安定構造のエネルギーやその原子座標，体積，更に計算時間などが記載されている [3].

2.2.2 Maple

Maple は，1980 年にカナダのウォータールー大学で生まれた数式計算機能をコアテクノロジーとして持った統合技術計算，技術文書作成環境である．その手軽で直感的なインタフェースにおいて，電卓代わりの計算から連立方程式や微分方程式から微積分計算，フーリエ変換に至るまでの基本的な数式処理数値計算を，計算ミスを軽減した上で可能とする [3]．本研究ではエネルギー曲面やグラフの表示などの作業で利用した．

2.3 計算モデル

O 原子を配置する Si 完全結晶は，ダイアモンド構造を持つ Si 8 原子のユニットセルを 3 軸方向にそれぞれ 2 倍ずつ拡張したスーパーセルを用いた．このモデルの Si-Si 結合付近，または結合上に O 原子を一つ配置し，内部緩和を行った．その結果が最安定構造を再現するか検証した．原子数は Si 原子（青色）が 64 個，O 原子（黄色）が 1 個である．

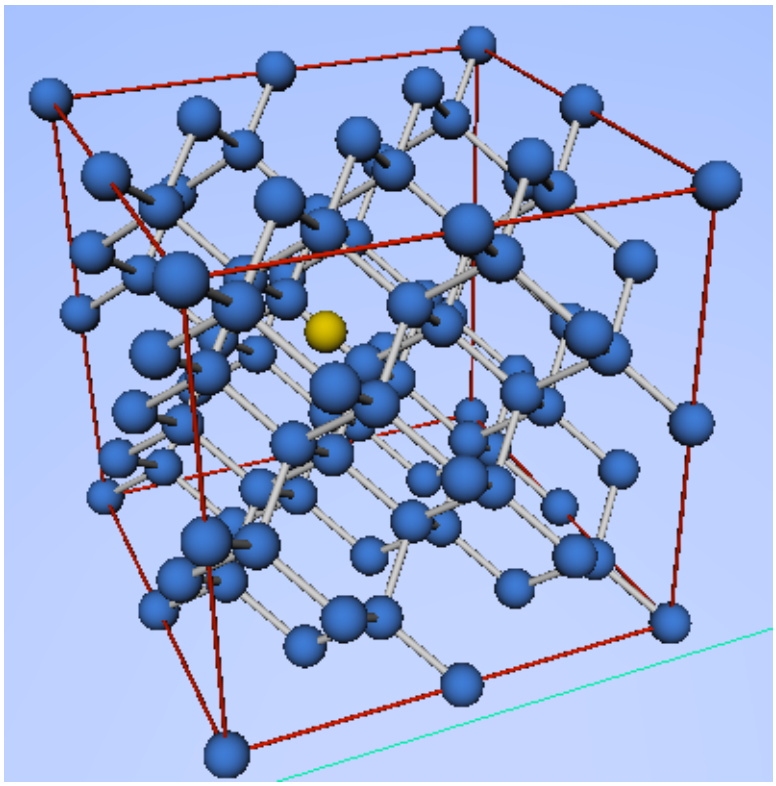

図 2.1: 2*2*2 に拡張したスーパーセル.

2.4 最小値の計算

多次元空間における最小値は以下の流れで計算する．

(i) OUTCAR から O 原子にかかる force を取り出す．

(ii) CG 法で共役な方向を求める．

(iii) mnbrak で極小を囲い込む．

(iv) Brent 法で 1 次元における極小を求める．

(v) 以上を最小にたどり着くまで繰り返す．

図 1(a) が上記の構造緩和の流れを示した．エネルギー計算を行う際，Ruby 言語によるコードで原子座標や force を取得し，新たな座標を作成する．この座標を VASP の入力ファイル POSCAR に書き込み，第一原理計算を行っている．図 1(b) に VASP との連携の流れを示した．本節では上記の各方法の詳細な説明を行う．

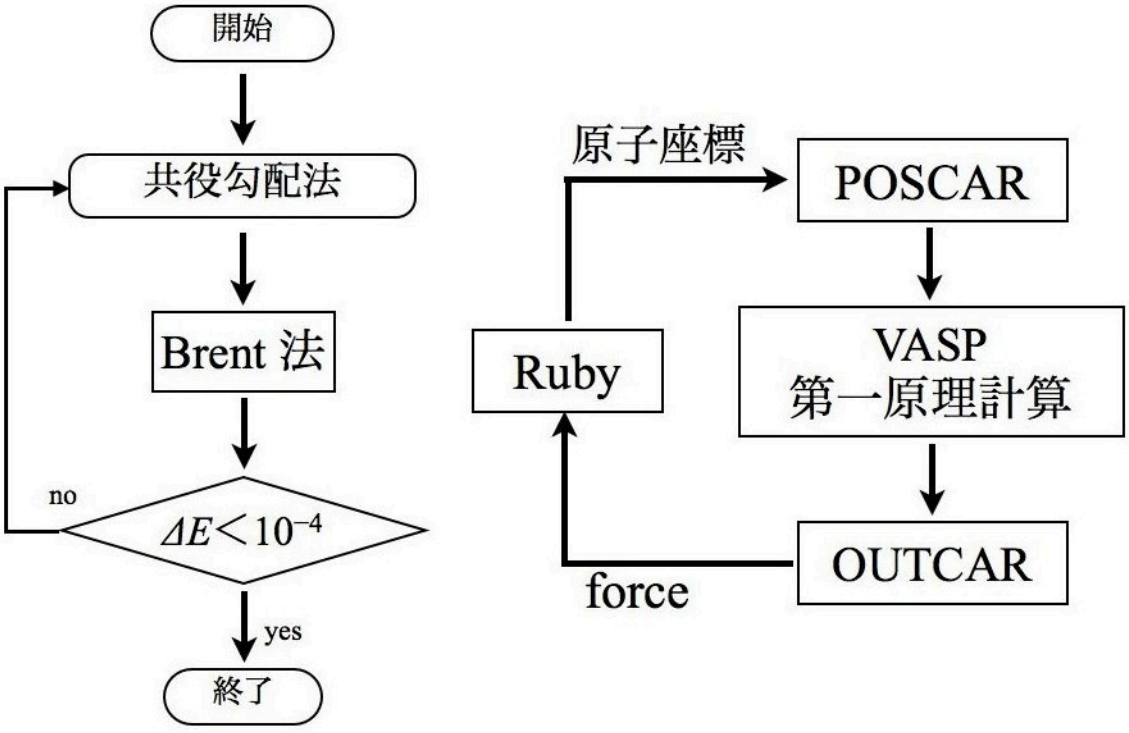

図 2.2: (a) 構造緩和の流れ，(b) VASP との連携の流れ．

2.4.1 mnbrak

1 次元での最小値を求める手法である Brent 法（詳細は 2.9 節）では極小値を挟むような 3 点を初期値として与えないといけないため，そのような点を選ぶ関数 mnbrak を用いる．ある極小の周りに点を選んだ状態のことを極小を「囲い込む」という．図（2.3）(a) は極小を囲い込んでいない 3 点であり，図（2.3）(b) は囲い込みが出来ている 3 点を示したものである．

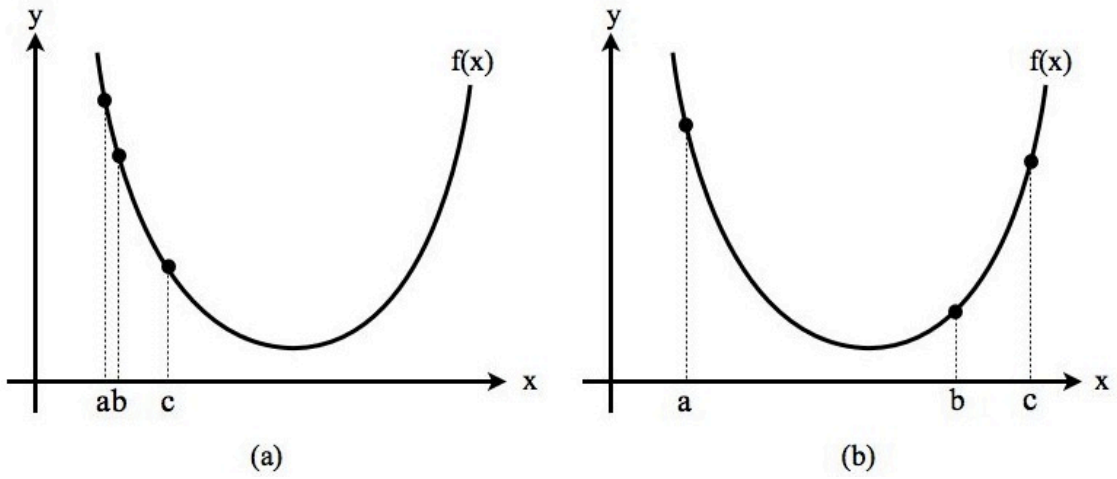

図 2.3: 囲い込みが (a) 出来ていない 3 点，(b) 出来ている 3 点

ある区間 (a,b) に根（関数値が 0 の点）があるかどうかを考える場合，a，b の関数値が異符号ならば確実に根があると言える．しかし，極小値があるかどうかは 2 点だけでは判定できない．そこで $a < b < c$ となる 3 点を与えたとする．もし図（2.3）(b) のように $f(b)$ が $f(a)$ と $f(c)$ の両方よりも小さければ，区間 (a,c) 内に必ず極小を持つということがわかる．[4] 関数 mnbrak ではこのような 3 点を求める．本研究 では緩和する原子の座標 (pos1) と，その原子の最近接原子間距離の 1/10 倍の点の座標 (pos2) の 2 点を与え，これらの 2 点の座標を入力として第一原理計算を行う． この計算結果からエネルギーを取り出し，これらの座標とエネルギーを初期値として囲い込む 3 点を求める．3 点目の座標は pos2+gold∗(pos2-pos1) として計算する．図（2.4） (a)，(b) が関数 mnbrak で囲い込むまでの 3 点を示したものである．図（2.4） (a) では，極小を囲い込めているように見えるが，真ん中の青の点が右側の緑の点よりも大きいため，極小を囲い込んでると断定できない．そのためこの段階では終了せず，図（2.4） (b) のような 3 点になるまで動く．ここで初期点 (pos1) を極小の右側にとった場合を図（2.5）に示した．ここで初期値として pos1 に $x = 6$，pos2 に $x = 7$ をとっている．この場合は pos1 よりも pos2 が下にあるため，pos1 と pos2 を入れ替え，左側に進んでいく．ただし gold=$(1 + \sqrt{5})/2$ とし，黄金比を表している．黄金比を用いるのは，最小値を求

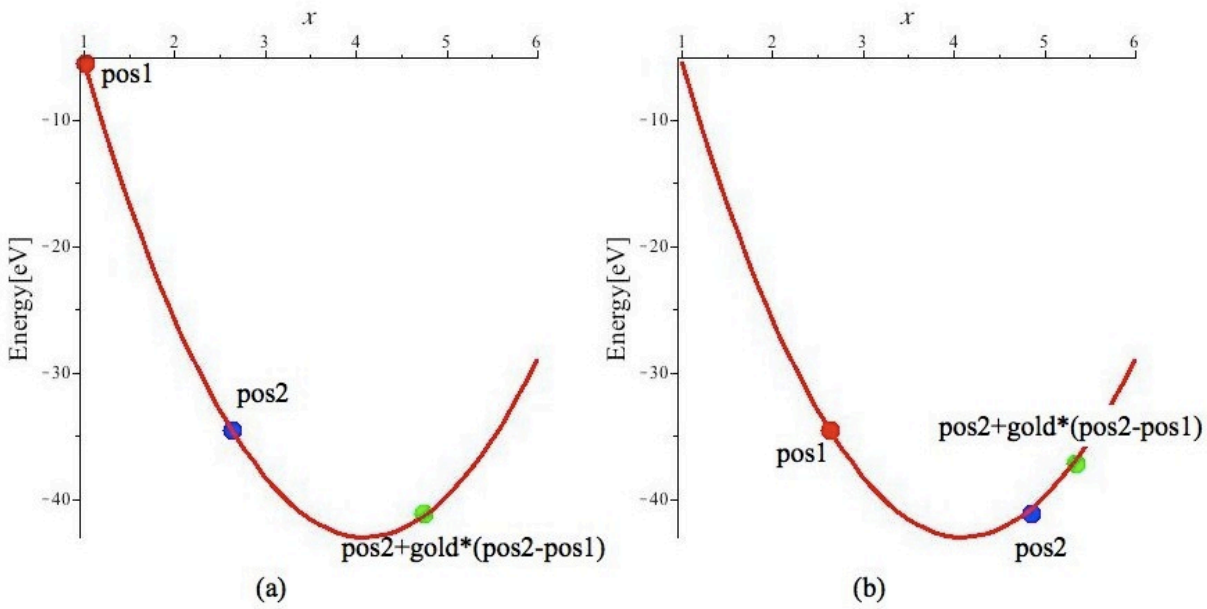

図 2.4: mnbrak を動かした時の (a) 囲い込み中の 3 点，(b) 終了時の 3 点.

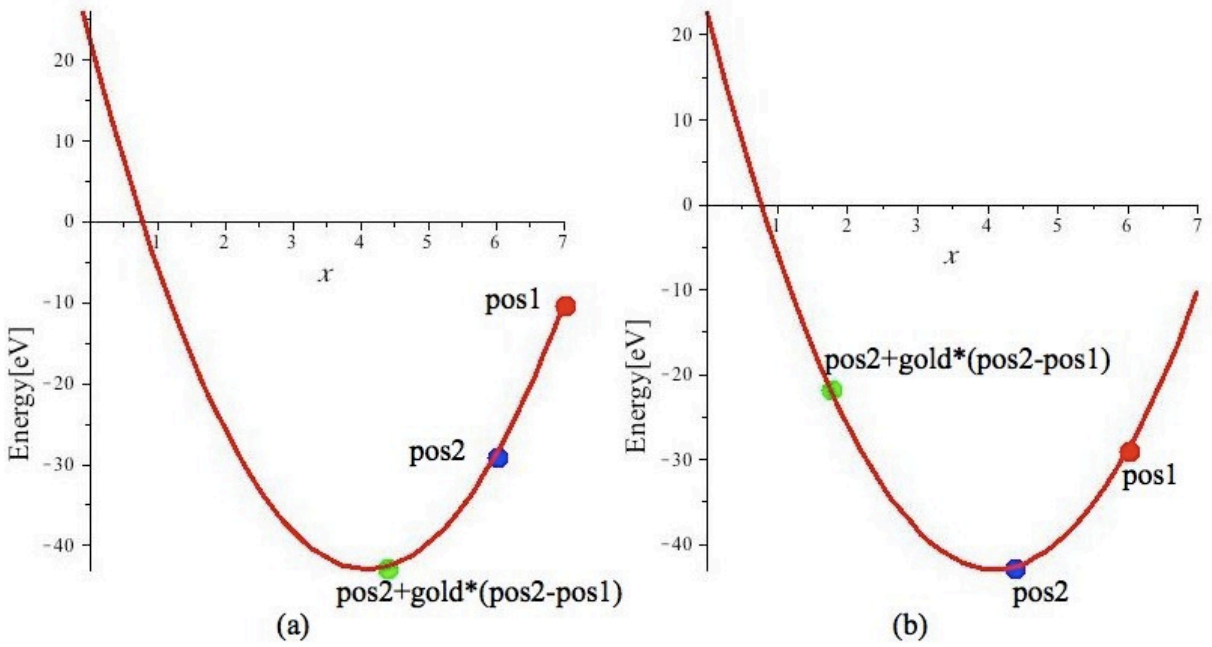

図 2.5: 初期値を極小の右側にとった時の (a) 囲い込み中の 3 点，(b) 終了時の 3 点.

める際に黄金分割法を用いるからである．黄金分割法は根を求める手法である二分法に相当する．

二分法　二分法は，根が区間 (a,b) にあるとき，中点 x で関数値を計算し，区間 (a,x) ，または区間 (x,b) のどちらに根があるのかを調べ，次第に根を狭い区間に囲い込んでいく．しかし極値が中点 x の右側にあるか，左側にあるかわからない場合は二分法よりも初期値として 3 点を与えている黄金分割法の方が適している．

黄金分割法　黄金分割は区間 (a,c) $(a < b < c)$ に極小値を囲い込み，新しい点 x を a と b の間または b と c の間にとる．ここで b は a から右に $c-a$ の W 倍 $(W < 1)$ だけ進んだ点とすると，

$$\frac{b-a}{c-a} = W \tag{2.1}$$

$$\frac{c-b}{c-a} = 1-W \tag{2.2}$$

となる．また，x は b から右に $c-a$ の Z 倍 $(Z < 1)$ だけ進んだ点（図 (2.6)）とする．

$$\frac{x-b}{x-a} = Z \tag{2.3}$$

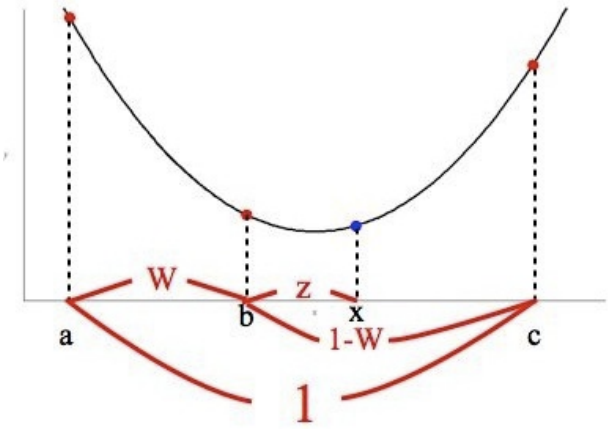

図 2.6: 新しい点の取り方．

次に，x と b の関数値を比較して，関数値が大きい方の端点である a または c を除去する．図 (2.6) の場合は b の関数値が x の関数値よりも大きいため，端点 a を除去する．すると，囲い込みの区間幅は，新しい 3 点が a，b，x の場合は現在の $W+Z$ 倍，b，x，c の場合は現在の $1-W$ 倍となる．この際，W を大きい

値と決めると，a を除去した場合は区間の狭まる幅は大きくなる．しかし，逆に c が除去された場合には狭まる幅は小さくなってしまう．そこで除去する点によって区間の狭まり方に偏りがなくなるようにするには区間（a,b）と区間（x,c）が等しくなるようにすればよい．つまり，$|c-x| = W$ になるようにする．よって

$$Z = 1 - 2W \tag{2.4}$$

となる．これは，$|b-a| = |x-c|$ となる位置に新しい点がくることになるため，x は区間（a,b）, (b,c）のうちの広い方に来ることになる．また区間（b,c）に対する x の位置関係は，(a,c）に対する b の位置関係と同じになる．つまり，

$$\frac{Z}{1-W} = W \tag{2.5}$$

となる．よって式（2.4）, (2.5）より

$$W = \frac{3-\sqrt{5}}{2} \fallingdotseq 0.38197 \tag{2.6}$$

よって最適な囲い込みの 3 点 $a < b < c$ は，中央の点 b を近い方の端点からの距離と遠い方の端点からの距離が 0.38197:0.61803 になるように選べばよい．この比は黄金比になっている．図（2.7）に黄金分割法が進んでいく様子を示した．図（2.7）(a) の青で示した点，および図（2.7）(b) の緑で示した点が新しい点 x，x' である．また，(a) の b，c は (b) の a'，b' と更新される．二分法では 1 回で区間が全体の 0.5 倍に減少するが，黄金分割では 1 回で全体の 0.61803 倍になり，区間の減少の仕方は二分法よりも遅い．しかし，黄金分割法では最初に極小値を囲い込んでいれば確実に極小値を求められるため，黄金分割法が用いられている．

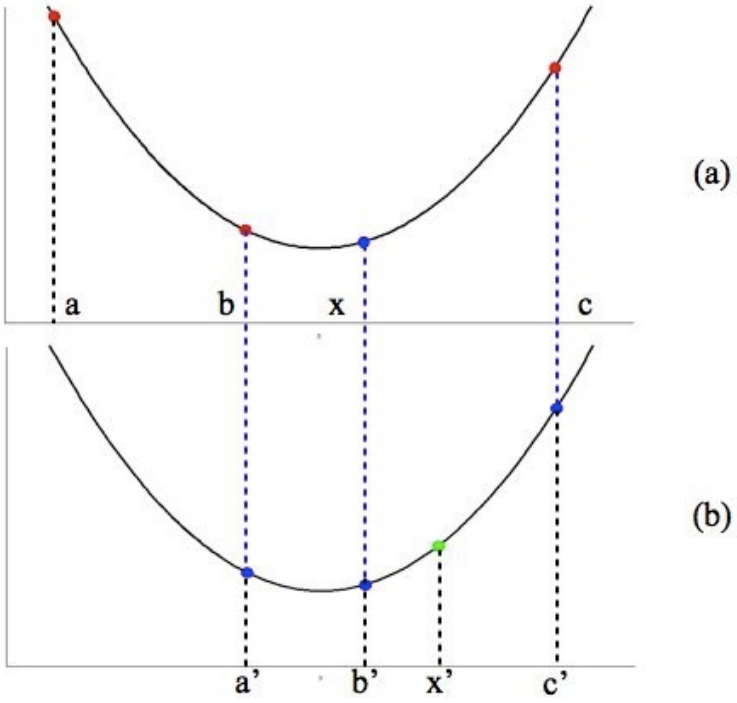

図 2.7: 黄金分割法.

2.4.2 Brent

Brent の方法とは 1 次元のにおける最適化手法の 1 つで，黄金分割法と放物線補間を組み合わせたものである．黄金分割法による関数の最少化は導関数が連続でないような場合でも最小化が出来る方法であるが，収束するまでに時間がかかる．そこで導関数が連続であるような関数の場合は計算時間の早い放物線補間を利用する．Brent の方法では関数 mnbrak で得られた 3 点を入力とし，最小値を求める．Brent の方法の典型的な最終配置は両端点 a，b が $2 * x * 10^{-5}$ だけ離れる（式 2.7）．さらに x は a と b の中点になる．(式 2.8) [4]

$$b - a = 2 * x * 10^{-5} \tag{2.7}$$

$$x = \frac{b-a}{2} \tag{2.8}$$

よって相対精度は $\pm 10^{-5}$ になる．

放物線補間　放物線補間は初期値として極小を囲い込む座標を 3 点を与え， 関数値を求める（図 2.8 の①，②，③）．次に選んだ 3 点を通るような放物線を作り，極小値の x 座標を

$$x = b - \frac{1}{2}\frac{(b-c)^2|f(b)-f(c)| - (b-c)^2|f(b)-f(a)|}{(b-a)|f(b)-f(c)| - (b-c)|f(b)-f(a)|} \tag{2.9}$$

から求める．この時の関数値（図 2.8 の④）と最初の 3 点を比較し，値の小さい 3 点を新たな 3 点として選び再度放物線を作り，式（2.9）を利用し極小値の x 座標を求める（図 2.8 の⑤）．この操作を繰り返すことでもとの関数の極小に近づいていく．[4] Brent の方法で求めたエネルギーの変化を図 2.9（a）に，最安定位置におけるエネルギーとの差の両対数グラフを図 2.9（b）に示した．これらの図から，計算回数に応じてエネルギー値が収束していることがわかる．

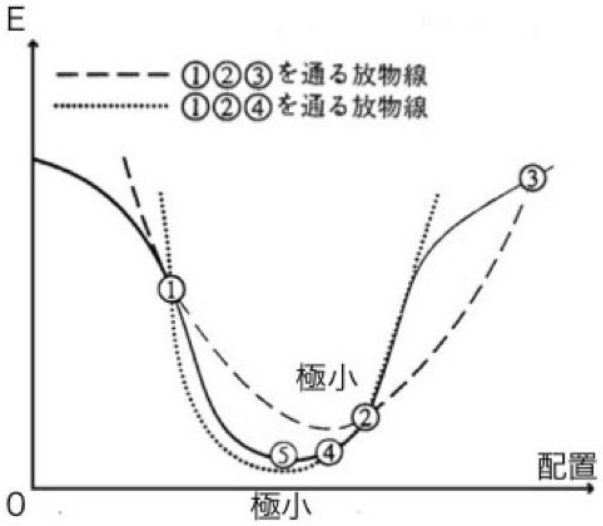

図 2.8: 放物線補間.

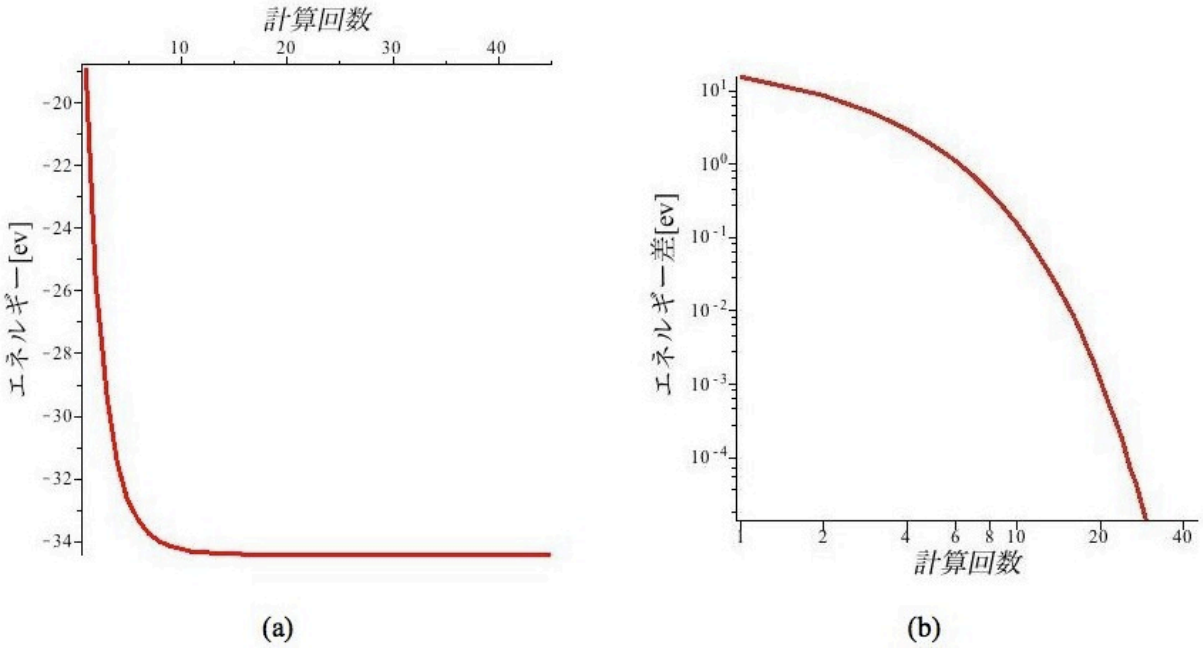

図 2.9: Brent 法の (a) 計算回数とエネルギーの変化　(b) 最小値とのエネルギー差.

2.4.3 共役勾配法

共役勾配（Conjugate Gradient：CG）法は多次元空間での極小を求める手法で，点 P での関数値 $f(P)$ だけでなく，勾配（1 階偏導関数）$\nabla f(P)$ も計算できる場合に使用するものである．実際の計算では，まず初期位置からある方向における 1 次元の極小値を Brent 法を利用して求める．次に得られた極小の位置から再度別の方向に向かって 1 次元の極小値を求める．この作業を繰り返し行い，最終的に多次元での極小値にたどり着くというものである．1 次元の方向を決定する際，前回までのある方向にそった最小化が無駄にならないような方向を採用する必要がある．このような方向を「共役な方向」と呼ぶ．ある点を P_i とし，そこからある方向 u にそって極小 P_{i+1} まで進み終え，次に新しい方向 v にそって進もうとしているとする．この時，P_i から方向 u での最小化が無駄にならないようにする為の条件は，P_{i+1} における勾配ベクトルが u に垂直であることである．勾配ベクトルとは点 P_k において関数値を最も大きく増やす方向を示すベクトルである．CG 法では減少する方向のベクトルとして利用するため，この勾配ベクトルを逆向きにしたベクトルを利用する．以下この減少方向のベクトルを勾配ベクトルと表記する．「共役」の数学的な定義は N 次対称行列 A に対して二つのベクトル u，v が

$$u^t A v = 0 \tag{2.10}$$

を満たすとき u と v は互いに共役であるという．さらに A が単位行列のとき上式はベクトルの直行条件となるため「共役」は「直行」の拡張概念である．ここで g_i を座標 P_i における勾配ベクトルとして，各ループでの 1 次元の探索を行う方向である共役な方向のベクトル h_i を以下のように定める．

$$g_i = -\nabla f(P_i) \tag{2.11}$$

$$h_{i+1} = g_{i+1} + \gamma_i h_i \tag{2.12}$$

式（2.12）で得られた共役な方向のベクトル h_{i+1} を利用すると，次の極小の位置 P_{i+2} は

$$P_{i+2} = P_{i+1} + \alpha h_{i+1} \tag{2.13}$$

と表すことが出来る．ここで α は定数とする．図 2.10 に点 P_i における探索の方向のベクトル h_i と点 P_{i+1} における勾配ベクトル g_{i+1}，次の探索の方向である h_i と共役な方向のベクトル h_{i+1} を示した．

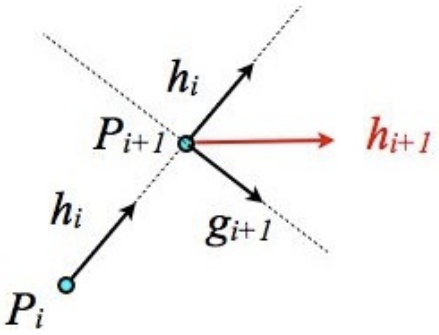

図 2.10: 共役な方向.

式（2.12）の γ の決め方には

(i) Fletcher-Reeves 法

$$\gamma_i = \frac{g_{i+1}g_{i+1}}{g_i g_i} \tag{2.14}$$

(ii) Polak-Ribiere 法

$$\gamma_i = \frac{(g_{i+1} - g_i)g_{i+1}}{g_i g_i} \tag{2.15}$$

が知られている．最初に Fletcher-Reeves が式（2.14）を用いていた．しかしその後，Polak-Ribiere が式（2.15）を用いることを提案した．これらの式は厳密な 2 次形式の時のみ同値となる．本研究では Polak-Ribiere 法を用いている．以下に 3 次元の関数を用いて実際に CG 法で極小値を求めるまでの過程を示す，図 2.11（a）にその関数のグラフと初期点 P_1 から極小値である P_{16} まで，図 2.11（b）にそのグラフを (x, y) 平面に等高線として示し，(a）と同様に初期点 P_1 から P_{16} まで移動する様子を示した．ここで用いたモデルの関数は $f(x, y) = x^2 + y^2 + xy$ で初期位置 P_1 の（x，y）座標は（1.0，9.0），極小値は，原点（0.0，0.0）である．

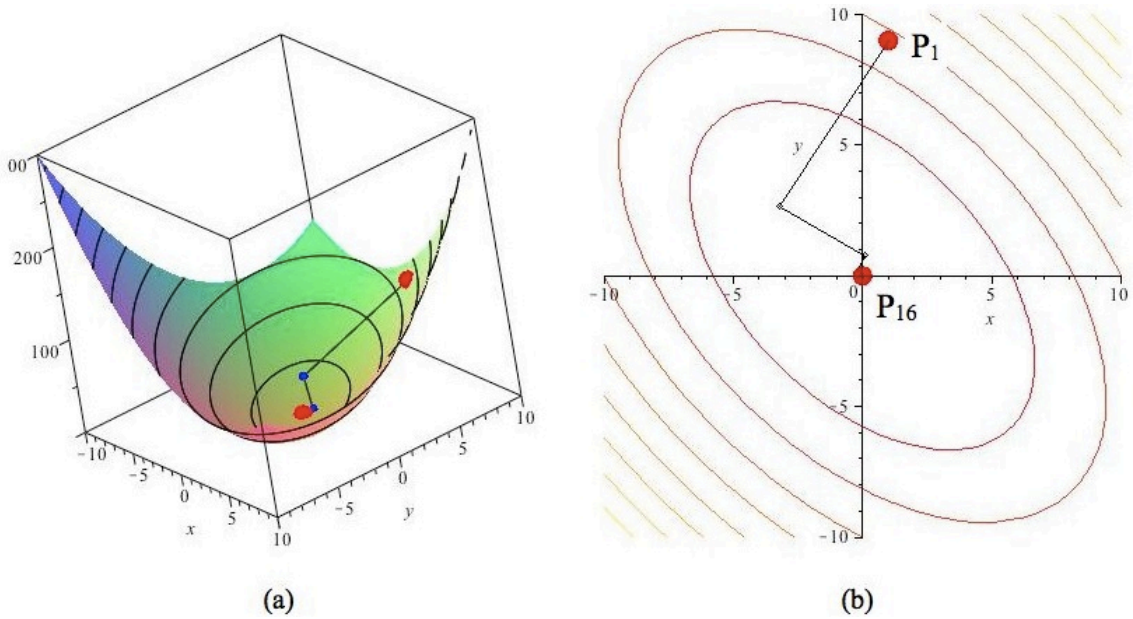

図 2.11: (a) 3 次元表示における CG 法, (b) 等高線表示における CG 法.

第3章　結果

本研究において作成した ruby 言語による CG 法を用いた構造緩和プログラムを利用した構造緩和と，VASP に標準で用意されている構造緩和機能を用いたものの結果をそれぞれ比較して CG 法による緩和プログラムの妥当性をしらべた．検証には 2.3 節に述べたスーパーセルを用いている．このスーパーセルの中に挿入した O 原子の初期位置を変えて内部緩和を行った．

3.1　Si 原子を固定

2.3 節のスーパーセルの Si 原子を初期状態のまま固定して検証を行った．

3.1.1　ボンドセンター

はじめに O 原子を Si-Si 結合のボンドセンターに配置した（図 3.1）(a)．図 3.1 (b) には（a）の O 原子周辺の拡大図を示した．この位置は O 原子にかかる force が 0 であったため CG 法で原子が移動しないという結果になった．各計算の詳細な座標とエネルギーを表 3.1 に示した．

表 3.1: 計算結果の比較．

	作成したコード	VASP
座標 [Å]((X : Y : Z))	6.15229:6.15229: 6.15229	6.15229:6.15229: 6.15229
Energy[eV]	-338.394	-338.394

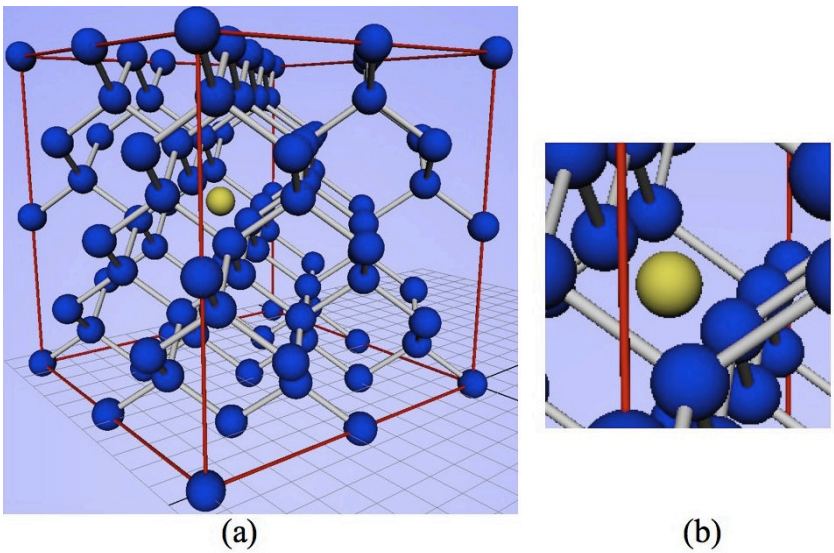

図 3.1: （a）ボンドセンターに配置した O 原子,（b）拡大図.

3.1.2 ボンドにそって移動

上記のボンドセンターに O 原子を配置した時の結果を受け，O 原子をボンドにそって移動させた（図 3.2）（a）．図 3.2（b）には図 3.1（b）と同様の拡大図を示した．この計算では O 原子のみを構造緩和させているため，O 原子が左側の Si 原子からの影響を強く受けることになり，図 3.3（a）の位置に O 原子が移動した．この結果は CG 法によるプログラムと同じ条件で VASP の構造緩和機能を用いて計算したものと同じ結果となった．図 3.3（b）に O 原子のボンドセンターの初期位置，ボンド上を移動させた後の位置，計算終了後の位置の 3 点を同時に表示した．計算終了後の原子は赤色の球で示している．本節でも同様に表 3.2 に 各計算の結果を示した.

表 3.2: 計算結果の比較.

	作成したコード	VASP
座標 [Å]((X : Y : Z))	3.41794:3.41794:3.41794	3.41794:3.41794:3.41794
Energy[eV]	-355.222	-355.222

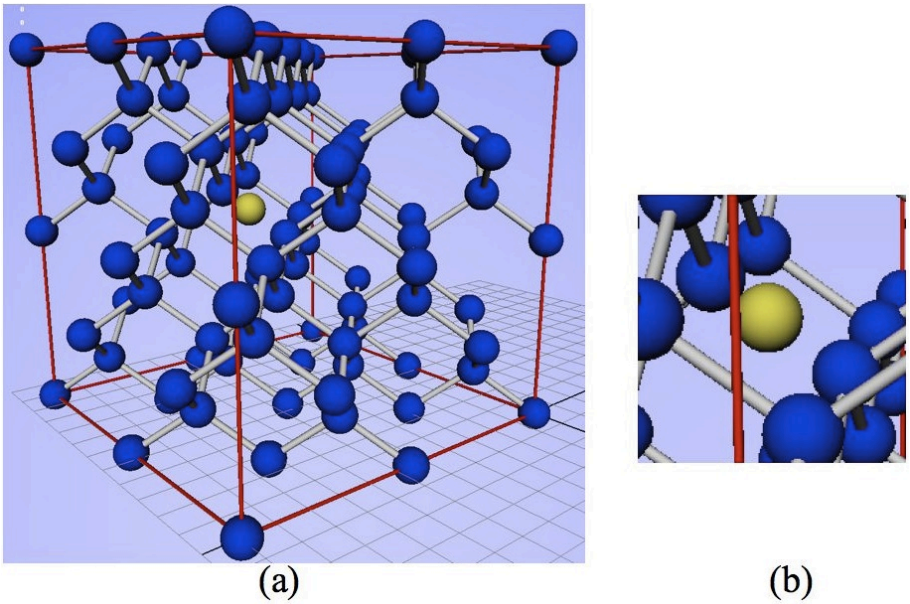

図 3.2: （a）O 原子をボンドにそって移動,（b）拡大図.

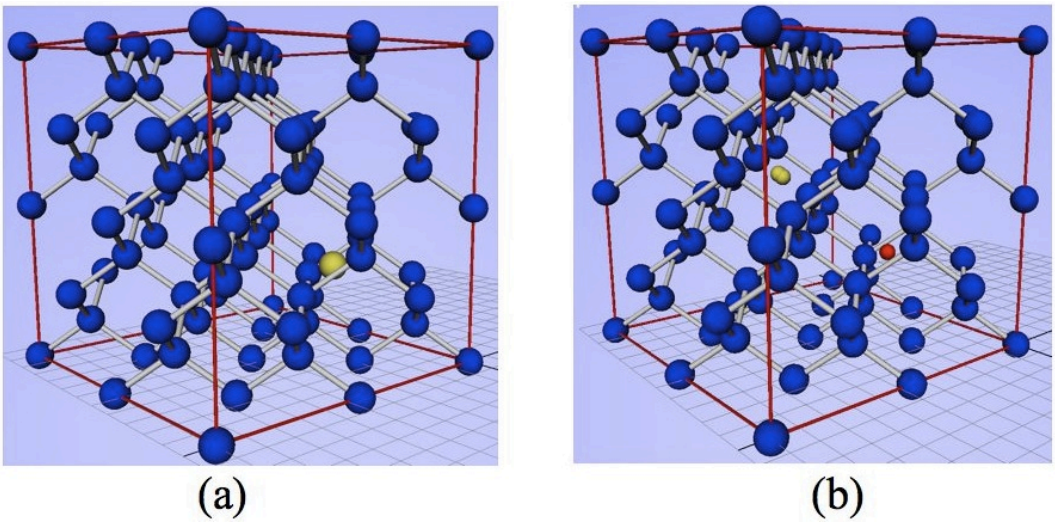

図 3.3: （a）計算結果,（b）初期位置，移動後，最終位置の 3 点

3.2　全原子緩和

続いて，Si 同士の結合のボンドセンターに O 原子を 1 つ挿入した状態をから VASP の緩和機能を用いて全原子に対して構造緩和を行った．緩和後の Si 原子を固定し，O 原子のみを CG 法，VASP それぞれで再度構造緩和を行った．以下の図は半透明な球が初期位置，濃い黄色が計算後の O 原子を示す．

3.2.1　ボンドにそって移動

全原子構造緩和後の Si 原子中に 3.1.2 節と同じ位置に O 原子を 1 つ挿入し，計算を行った．この位置を初期位置とすると，O 原子はボンドセンターの方向に力を受けることになる．その結果 O 原子はボンドセンターの force が 0 の位置で止まった．この結果も VASP と作成したコードの結果は同様のものとなった．

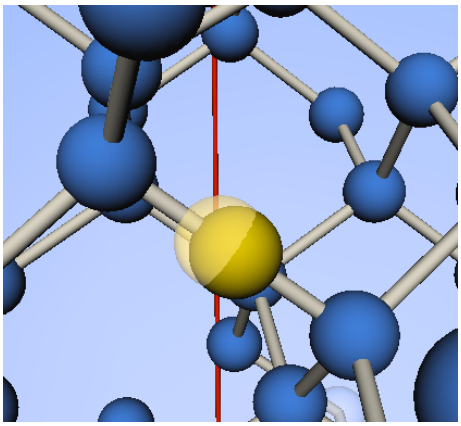

図 3.4: ボンド上に O 原子を配置.

3.2.2　オフセンターに配置

同様にボンドに対して垂直な方向に O 原子を移動させた場合にも計算を行った．この計算を VASP で行うと図 3.5（a）の位置が最安定となった．この，ボンドセンターが最安定になるという結果は，先行研究の計算結果と同一のものとなっている．この結果が VASP 緩和機能の信頼性が問われる要因となるものである．このモデルに対して本研究で作成したコードを用いて構造緩和を行うと図 3.5（b）が最安定となった．これらの座標とエネルギーを表 3.3 に示した．ここで，計算後の座標がボンドに対して逆側に移動しているが，結晶の対称性を考慮しているためボンドの周辺に円状の安定域が存在している．この事実は西谷研究室塚原の計算によって確認されている．このことからボンドの逆側に O 原子が移動しても問題

はない．逆側に移動した要因としては，O 原子のみを緩和しているため，ボンドよりも上にある Si 原子からの force を受け O 原子がボンドの下側に移動したと示唆される．

表 3.3: 計算結果の比較．

	作成したコード	VASP
座標 [Å]((X : Y : Z))	6.22234:6.22233:6.21484	6.30698:6.30695:6.05387
Energy[eV]	-354.616668	-354.601963

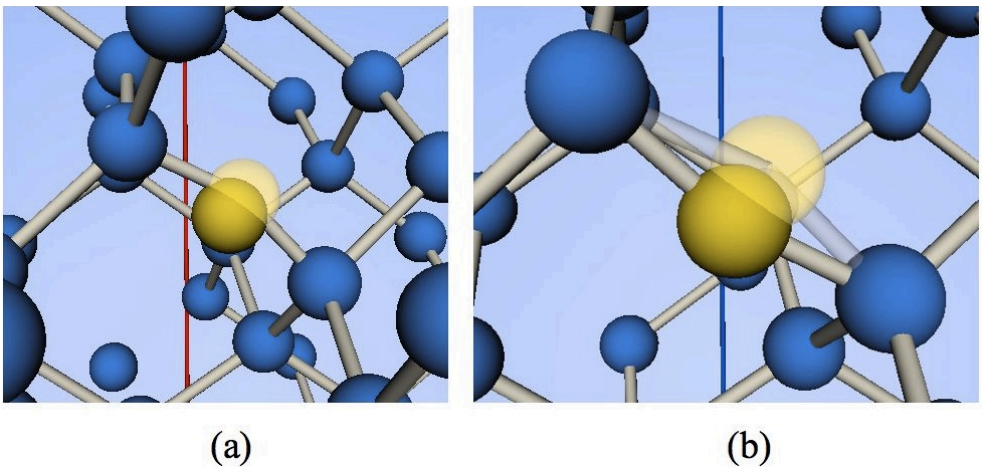

図 3.5: （a）VASP の結果,（b）作成したコードによる計算結果．

3.3 考察

以上の結果から VASP と同様に構造緩和を行うプログラムを作成することが出来た．通常構造緩和を行う際は，移動させる原子の第二近接の原子まで構造緩和を行うことが多い．しかし，今回は緩和を行わずに計算をした．そのため，O 原子が強い force を受けるということも再現され，3.1.2 節の様な結果になった．この計算結果が，VASP と同等の結果が得られたため，作成したプログラムが正しく構造緩和を行っていることが示された．また，3.2.2 節の結果のように VASP の構造緩和とは異なる結果も得られた．3.2.2 節の VASP における構造緩和の結果は 1.1 節でも述べたように誤っている．作成したプログラムではその結果を再現することなく，より安定な位置に O 原子が移動した．

第4章　総括

第一原理原子構造緩和法の実装に関する研究において，近年，VSAP に搭載されている構造緩和のルーチンの信頼性が疑われているという事実があった．そのため，Ruby 言語を利用して独自の構造緩和のプログラムを作成した．

計算モデルは Si 完全結晶を各軸方向にそれぞれ 2 倍に拡大したスーパーセルを用いた．さらにこのモデルを以下の 2 種類に分けた．

(I) Si 完全結晶を初期位置に固定した構造緩和前のモデル．

(II) Si 完全結晶中に O 原子を挿入し，全原子を構造緩和後のモデル．

これらのモデルの Si-Si 結合上，または結合付近に O 原子を配置し，構造緩和を行った．

構造緩和前

ボンドセンター　構造緩和前のモデルで O 原子をボンドセンターに配置した場合，原子が移動することがなかった．これは O 原子にかかる force が 0 であったため，原子が移動しなかったことが考えられる．

ボンド上　同様に構造緩和前のモデルでの O 原子の初期位置をボンドセンターから少し移動させたボンド上の位置にした場合，O 原子が大きく移動した．これは O 原子のみを緩和させたことで，Si 原子との距離が近くなりすぎたことから O 原子が受ける force が大きくなりすぎたと考えられる．

これらの結果は VASP の内部緩和ルーチンを利用した場合のものと一致した．よって，VASP の内部緩和ルーチンと本研究で作成したプログラムのアルゴリズムが同様のものであるという結論を得られた．そのため，構造緩和に関するパラメータを変更する際に，VASP の内部プログラムを操作する必要がなくなった．

構造緩和後

ボンド上　この構造緩和後のモデルでも O をボンド上の少し移動させた位置に配置し，構造緩和を行った．その結果，ボンドセンターで最安定となり，VASP の結果と同様になった．

オフセンター　O 原子をボンド上でない位置であるオフセンターに配置した．VASP の構造緩和の結果はこの場合でもボンドセンターが最安定となった．それに対して，作成したプログラムを利用した場合は Si 同士の結合に対して初期位置と逆の位置に移動し，オフセンターの位置が最安定となった．ここで対称性を考慮しているため，ボンドセンターを中心とした円状の安定域が存在することが西谷研究室塚原によって明らかになっている．このことから，より安定な構造を再現することに成功した．

以上の結果から最安定構造を求める計算プログラムが正しく動いていることが確認できた．1 原子のみを緩和させ，最安定構造を再現するためには，適切な初期位置と多原子における緩和の実装が必要であることがわかった．

参考文献

[1] 西谷滋人,「個体物理の基礎」,(森北出版　2006)

[2] 原田智史,「粒界エネルギーの転位模型」,(関西学院大学　理工学部　情報科学科　卒業論文　2011)

[3] 谷口僚,「Cz 法における SiO の核生成　ならびに　歪み Si のための SiGe の局所的な規則性」,(関西学院大学　理工学部　情報科学科　卒業論文　2010)

[4] William H. Press 他,「ニューメリカルレシピ・イン・シー」,(技術評論社 1993)

[5] 島田　赳仁,「無制約非線形最適化問題に対するアルゴリズムの比較」, http://www.is.titech.ac.jp/ kojima/lab/thesis/2007/0211889.pdf

[6] 西谷滋人,西谷研究室 2011 年度在籍生,「はじめての VASP 原理から使用法,適用例まで」,(2011)

謝辞

本研究を遂行する日々において，終始多大なる心温かい御指導，御鞭撻を頂いた西谷滋人教授に対し，深く感謝致します．また，本研究の成就に至るまで，同研究室に所属する同輩達，ならびに山本洋佑先輩，正木佳宏先輩，谷口僚先輩方からの多くのご協力があり，そして知識の共有を頂いたことをここに記します．最後ではありますが，私を支えてくれた全ての友人・仲間・家族に対し，心より深く御礼申し上げます．

付 録A　ファイルの配置とコード

A.1　ファイルの配置

VASP の入出力ファイルと作成した main.rb のみを配置する ”vasp” というディレクトリと，これら以外の Ruby コードを配置する ”ruby” というディレクトリを同じ階層に配置しておく．

ディレクトリ

```
/Users/akira/Desktop/relax/relax/refact/e-vec% ls
ruby vasp
```

A.2　コード

A.2.1　main.rb

作成したコードのメインとなるコード．ここではディレクトリの整理と，変数の初期化，初期値における第一原理計算のみを行い，原子座標と次元のみを次に渡す．最初に ”vasp” ディレクトリの中の ”result” に前回の結果が入っている場合はすべて消去されるので注意が必要である．

```
t0 = Time.now
require "../ruby/frprmn.rb"
require "../ruby/relax1.rb"
require "../ruby/find.rb"

require 'pp'

#残っている結果の削除
system("rm -rf  ./result")
system("mkdir ./result")
system("mkdir ./result/result")
system("rm -rf ./POSCARsub")
system("cp POSCAR POSCARsub ")
```

```
#残っている VASP の出力を削除
system("rm -rf vasp-job.*")
system("rm -rf majima.vasp.*")
system("rm -rf OUTCAR")
pp "delete OUTCAR"

sleep(3)
system("qsub run.sh")
sleep(5)

loop do
  if find == true then
    pp "check2"
    break
  end
  pp "wait 1min"
  sleep(60)
end

#次元:n
n=3
n=n-1

xfret=[0,0]
#原子座標の取得
p1=outfile

$ncom=n
$pcom=Array.new(n+1)
$xicom=Array.new(n+1)

pp frprmn(p1,n)

t1 = Time.now
system("rm -rf vasp-job.*")
system("rm -rf majima.vasp.*")
print t1-t0,"s","\n"
```

A.2.2 frprmn.rb

関数 frprmn は CG 法を実行する．ただし，実際にこの関数が行っているのは前回の探索方向を受け取り，Fletcher-Reeves 法または，Polak-Ribiere 法で新たな探索方向を作成している．また，1 次元の最小が得られたときに原子座標とエネルギーをテキストファイルに書き出す．

```
require 'pp'
include Math
 require "../ruby/linmin.rb"
 require "../ruby/relax1.rb"
 require "../ruby/f1dim.rb"

def frprmn(p1,n)
  ftol=1.0e-4
  eps=1.0e-4
  g=Array.new(n+1)
  h=Array.new(n+1)
  xi=Array.new(n+1)

  fp=outene
  xi=outforce

  for j in 0..n do
    g[j]=xi[j]
    h[j]=xi[j]
  end

  for its in 0..200 do
    xi,xfret,p1=linmin(p1,xi,n)
    File.open("result.txt","a+"){ |file|
      file.puts(p1,xfret[1])
    }
    system("mv result.txt ./result/result")
    system("mv ./result/result ./result/result#{its+1}")
    system("mkdir ./result/result")
    if (2.0*(xfret[1]-fp).abs) <= ftol*((xfret[1]).abs+(fp).abs+eps) then
      return "fin."
    end
```

```
    dgg=0.0
    gg=0.0

    for j in 0..n do
      gg+=g[j]*g[j]
      #dgg+=xi[j]*xi[j] # Fletcher-Reeves
      dgg+=(xi[j]+g[j])*xi[j] #Polak-Ribiere
    end

    if gg==0.0 then
      print "gg-frp ",xfret,"\n"
      return xfret
    end
    gam=dgg/gg
    for j in 0..n do
      g[j]=-xi[j]
      h[j]=g[j]+gam*h[j]
      xi[j]=g[j]+gam*h[j]
    end
  end
  return p1
end
```

A.2.3 linmin.rb

関数 linmin では main.rb で初期化した大域変数の定義と mnbrak , brent への数値の受け渡しを行う仲介のための関数である.

```
require 'pp'
include Math
require "../ruby/mnbrak.rb"
require "../ruby/brent.rb"

def linmin(p1,xi,n)
  for j in 0..n do
    $pcom[j]=p1[j]#座標
    $xicom[j]=xi[j]#探索方向
  end
  ax,xx,cx=mnbrak
```

```
  xfret=brent(ax,xx,cx)[0]
  p1=outfile
  xi=outforce
  return xi,xfret,p1
end
```

A.2.4 mnbrak.rb

関数 mnbrak は 2.4 で述べたように，極小を囲い込むためのものである．このファイルにある関数 mnsign と関数 fmax はそれぞれ，同じ符号の値を返すためのものと，大きい方の値を返すためのものである．ここで，p2=naighbor[2]/10 としてあるが，これは最近接原子間距離の 10 分の 1 である．この数字は経験的パラメータであるため，検証が必要である．

```
include Math
require 'pp'
require "../ruby/makepos.rb"
require "../ruby/naighbor.rb"
require "../ruby/atom.rb"
require "../ruby/f1dim.rb"

def mysign(a,b)
  if b<0 then
    return -a
  else return a
  end
end

def fmax(a,b)
  if a>b then
    return a
  else return b
  end
end

def mnbrak
  gold=(1+sqrt(5))/2
  p1=0.0
  p2=naighbor[2]/10
```

```
pp  fp1=outene
system("mkdir ./result/result/mnres1")
system("cp * ./result/result/mnres1")
pp  fp2=f1dim(p2)
system("mkdir ./result/result/mnres2")
system("cp * ./result/result/mnres2")

if fp2>fp1 then
  tmp=p1
  p1=p2
  p2=tmp
  tmp=fp2
  fp2=fp1
  fp1=tmp
end

  p3=p2+gold*(p2-p1)
pp  fp3=f1dim(p3)
system("mkdir ./result/result/mnres3")
system("cp * ./result/result/mnres3")

i=0

while fp2>fp3 do
  i=i+1
  r=p2-p1*fp2-fp3
  q=p2-p3*fp2-fp1
  u=p2-((p2-p3)*q-(p2-p1)*r)/(2.0*mysign(fmax((q-r).abs,1.0e-20),q-r))
  ulim=p2+100.0*(p3-p2)
  if (p2-u)*(u-p3)>0.0 then #a
    fu=f1dim(u)
    system("mkdir ./result/result/mnres#{i+3}")
    system("cp * ./result/result/mnres#{i+3}")

    if fu<fp3 then
      p1=p2
      p2=u
      fp1=fp2
      fp2=fu
```

```
    break
  elsif fu>fp2 then
    p3=u
    fp3=fu
    break
  end
  u=p3+gold*(p3-p2)
  fu=f1dim(u)
  system("mkdir ./result/result/mnres#{i+3}")
  system("cp * ./result/result/mnres#{i+3}")

elsif (p3-u)*(u-ulim) >0.0 then #b
  fu=f1dim(u)
  system("mkdir ./result/result/mnres#{i+3}")
  system("cp * ./result/result/mnres#{i+3}")
  if fu<fp3 then
    p2=p3
    p3=u
    u=p3+gold*(p3-p2)
    fp2=fp3
    fp3=fu
    fu=f1dim(u)
    system("mkdir ./result/result/mnres#{i+3}")
    system("cp * ./result/result/mnres#{i+3}")
  end

elsif (u-ulim)*(ulim-p3)>0.0 then #c
  u=ulim
  fu=f1dim(u)
  system("mkdir ./result/result/mnres#{i+3}")
  system("cp * ./result/result/mnres#{i+3}")
else #d
  u=p3+gold*(p3-p2)
  fu=f1dim(u)
  system("mkdir ./result/result/mnres#{i+3}")
  system("cp * ./result/result/mnres#{i+3}")


end
```

```
    p1=p2
    p2=p3
    p3=u
    fp1=fp2
    fp2=fp3
    fp3=fu
  end
  pp i
  return p1,p2,p3
end

#pp mnbrak
```

A.2.5 brent.rb

??にあるような手法で 1 次元の最小値を求めるための関数である．このファイルの関数 hatena(a,b,c,d) は a が b より大きければ c を，そうでなければ d を返す関数である．なお，関数 mnbrak と関数 brent における各回の計算後に毎回 ”vasp” ディレクトリ内の ”result” というディレクトリにバックアップとして全ファイルをコピーしている．

```
include Math
require 'pp'
require "../ruby/relax1.rb"
require "../ruby/mnbrak.rb"
require "../ruby/makepos.rb"
require "../ruby/f1dim.rb"

def hatena(a,b,c,d)
  if a>=b then return c
  else return d
  end
end

def brent(p1,p2,p3)
  fret=[0,0]
  tol=1.0e-5
  cgold=1.0-(sqrt(5.0)-1)/2
```

```
zeps=1.0e-5
e=0.0
d=0.0
ax=p1
bx=p2
cx=p3

if ax < cx then
  a=ax
else a=cx
end

if ax>cx then
  b=ax
else  b=cx
end

x=bx
w=bx
v=bx
fw=f1dim(bx)
fv=fw
fx=fw

count=0

for j in 1 .. 150 do
  count = count + 1
  pp count
  pp [x,fx][1]
  xm=0.5*(a+b)
  tol1=tol*x.abs+zeps
  tol2=2.0*tol1

  #収束判定
  if (x-xm).abs <= tol2-0.5*(b-a) then
    xmin=x
    fret[1]=xmin
    fret[2]=fx
```

```
  printf("fin\n")
  break
end

if e.abs>tol1 then
  r=(x-w)*(fx-fv)
  q=(x-v)*(fx-fw)
  p=(x-v)*q-(x-w)*r
  q=2.0*(q-r)

  if q>0.0 then
    p=-p
  end

  q=q.abs
  etemp=e
  e=d
  if p.abs >= (0.5*q*etemp).abs or p <= q*(a-x) or p >= q*(b-x) then
    e=hatena(x,xm,a-x,b-x)
    d=cgold*e
  else
    d=p/q
    u=x+d
    if (u-a) < tol2 or (b-u)<tol2 then
      d=mysign(tol1,xm-x)
    end
  end
else
  e=hatena(x,xm,a-x,b-x)
  d=cgold*e
end

u=hatena(d.abs,tol1,x+d,x+mysign(tol1,d));
fu=f1dim(u)
system("mkdir ./result/result/brentres#{j}")
system("cp * ./result/result/brentres#{j}")

if fu<=fx then
  if u>=x then
```

```
        a=x
      else
        b=x
      end
      v=w
      w=x
      x=u
      fv=fw
      fw=fx
      fx=fu
    else
      if u<x then
        a=u
      else
        b=u
      end
      if fu<=fw or w=x then
        v=w
        w=u
        fv=fw
        fw=fu
      elsif fu <= fv or v=x or v=w then
        v=u
        fv=fu
      end
    end
  end

  xmin=x
  fret[1]=xmin
  fret[2]=fx

  return fret[1,2],count#[[position,energy],count]
end

#pp brent(mnbrak[0],mnbrak[1],mnbrak[2])
```

A.2.6 f1dim.rb

関数 frprmn によって得られた 1 次元の方向に新たな探索点をとり，第一原理計算を行う関数である．

```
include Math
require 'pp'
require "../ruby/relax1.rb"
require "../ruby/find.rb"

def f1dim(x)
#関数 linmin で設定した大域変数の読み込み
  ncom=$ncom
  pcom=$pcom
  xicom=$xicom

  xt=Array.new(ncom+1)

#探索方向のベクトルを単位ベクトルに変換
  long=0
  for j in 0..ncom do
    long+=xicom[j]**2
  end
  long=sqrt(long)

#新しい探索点の作成
  for j in 0..ncom do
    xt[j]=pcom[j]+x*(xicom[j]/long)
  end

  mkposfile(xt)
  path=File.expand_path(".")
  system("rm -rf "+path+"/vasp-job.*")
  system("rm -rf "+path+"/majima.vasp.*")
  system("rm -rf "+path+"/OUTCAR")
  pp "delete OUTCAR"
  sleep(5)

  system("qsub "+path+"/run.sh")
```

```
  loop do
    if find == true then
      pp "check2"
      break
    end
    pp "wait 1min"
    sleep(10)
  end

  return outene
end
```

A.2.7 makepos.rb

このファイルには f1dim.rb で作成した探索点の絶対座標を相対座標に変換する関数がある．関数 mkposfile は POSCAR を書き換えるために，現在の POSCAR をコピーをとり，同時に書き込み用に開いている．関数 mkpos では POSCAR のコピーである POSCARsub を読み込み対象となる原子の座標を書き込む．そのために，関数 position で基本並進ベクトルを取り出し，相対座標を作成している．

```
require 'pp'
require "../ruby/relax1.rb"
require 'matrix'

def position(coord)
  poscar=direct
#基本並進ベクトルの作成　pos=poscar[2][0].to_f
  a=Matrix[[pos,0,0],[0,pos,0],[0,0,pos]]
  ai=a.inv
  move_atom=Vector[coord[0],coord[1],coord[2]]
#相対座標の計算
  m=ai*move_atom
  return m
end

def mkpos(coord)#変更する
  poscar=[]
  a=[]
```

```
  line=[]
  newpos=""
  n = 0
  num = 0
  File.open("POSCARsub").each do |l|
    n = n + 1
    poscar.push(l)
#POSCARsub から"DIrect"をキーワードに読み出す
    if l == "Direct\n"
      num = n
    end
  end

#小数点以下を d 桁で切り捨て
  d=10**5
  for i in 0 ..2 do
    line[i]=position(coord)[i]
    line[i]=((line[i].to_f*d).floor)/d.to_f
    a[i]=line[i].to_s
  end

#POSCAR に書き込み
  newpos =  ("        ")
  for i in 0..2 do
    b=""
    for j in 0..9-a[i].size do
      b = b + "0"
    end
    newpos += a[i] + b + (" ")
  end
  poscar[num + infile - 1] = newpos
  return poscar
end

def mkposfile(coord)#ファイル書き換え
  system("cp POSCAR POSCARsub")
  File.open("POSCAR","w") {|file|
      file.puts(mkpos(coord))}
end
```

A.2.8 naighbor.rb

関数 naighbor は OUTCAR から最近接原子位置と距離を計算するものである.

```
include Math
require 'pp'
require "../ruby/relax1.rb"

def naighbor
outcar=[]
mainline=[]
compline=[]
position=[]
#OUTCAR からPOSITION                                    TOTAL-FORCE (eV/Angst)
を含む行を取り出す.
  File.open("OUTCAR").each do |line1|
    outcar.push(line1)
  end
  for i in 0..outcar.size-1
    if outcar[i] ==  " POSITION
         TOTAL-FORCE (eV/Angst)\n"
      number = i
    end
  end

#取り出した行数までを削除
  outcar.slice!(0, number)
  mainline = outcar[infile + 1].split(" ")

  for i in 0..infile-1 do
    compline << outcar[infile + 1 - i].split(" ")
  end
  compline.reverse!

  for i in 0..compline.size-1 do
    for j in 0..compline[i].size-1
      compline[i][j] = compline[i][j].to_f
    end
  end
```

```
  for i in 0..2 do
    position << mainline[i].to_f
  end

#最近接原子間距離の計算
  distance = 100
  count = 1
  for i in 0..compline.size-2
    if distance > sqrt((position[0]-compline[i][0])**2+(position[1]-compline[i][1]
      distance = sqrt((position[0]-compline[i][0])**2+(position[1]-compline[i][1])
      count = count + 1
    end
  end

  return compline[count],position,distance#[最近接原子の座標とエネルギー][着
目原子の座標とエネルギー][最近接原子間距離]
end
```

A.2.9 relax1.rb

このファイルは POSCAR または，OUTCAR から情報を取得するための関数がある．関数 str は引数として与えたキーワードとなる文字列を正規表現に変換し，マッチする行数を取得する．更に，得られた行数までを OUTCAR から削除し，残りを返す関数である．また関数 direct は POSCAR から ”Direct” のみの行を取得するための関数である．これらの関数を利用しているのが関数 infile，outfile，outene，outforce である．これらはそれぞれ入力された原子の個数，酸素の座標，系のエネルギー，酸素にかかる force を返す関数である．

```
require 'pp'

#キーワードを引数にする
def str(x)
  outcar=[]
  iter=0
  File.open("OUTCAR").each do |line1|
    outcar.push(line1)
    iter+=1
#正規表現に変換しマッチするかどうか検証
```

```
    if Regexp.new(x)  =~ line1
      $number = iter
    end
  end
  outcar.slice!(0, $number)
  return outcar
end

def direct
  poscar=[]
  num=0
  File.open("POSCARsub").each do |line|
    poscar.push(line.chomp.split(" "))
    if line == "Direct\n"
      break
    end
  end
  return poscar
end

def infile#POSCAR number of atoms
  num=0
  poscar=direct
  for i in 0..poscar[-2].size
    num = num + poscar[-2][i].to_i
  end
  return num
end

def outfile#position from OUTCAR
  lookatom=[]
  position=[]
  outcar = str("TOTAL-FORCE")
  lookatom = outcar[infile].split(" ")
  for i in 0..2 do
    position[i] = lookatom[i].to_f
  end
  return position
end
```

```
def outene#energy from OUTCAR
  lookene=[]
  position=[]
  outcar=str("ION-ELECTRON")
  lookene = outcar[1].split(" ")
  energy = lookene[4].to_f
  return energy
end

def outforce#force from OUTCAR
  lookatom=[]
  force=[]
  outcar=str("TOTAL-FORCE")
  lookatom = outcar[infile].split(" ")

  for i in 0..2 do
    force[i] = lookatom[i+3].to_f
  end
  return force
end
```