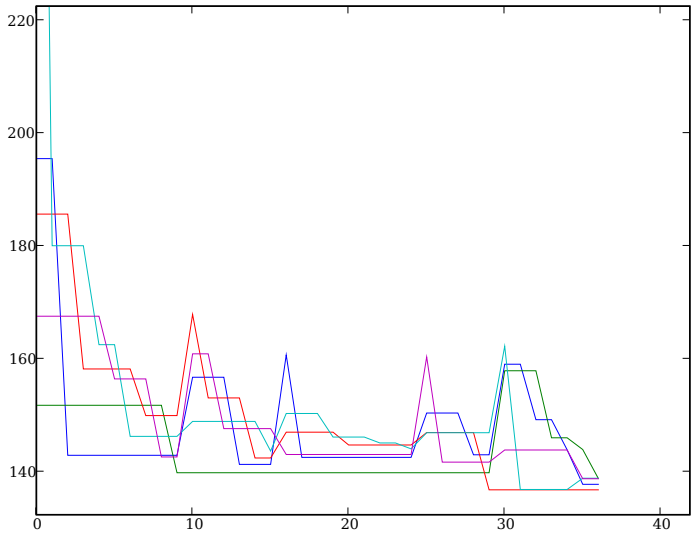
Parameter Optimisation for Grid-based Projector Augmented Wave Method using Downhill Simplex Algorithm

10302 — Electronic Structure Methods in Material Physics, Chemistry and Biology



Author:
Ask Hjorth Larsen, s021864

May 11, 2007

Technical University of Denmark
Lyngby

Abstract

The Grid-based Projector Augmented Wave method relies on a number of parameters specifying properties such as cutoff radii between inner and outer atomic regions. Exhaustive searching for optimal parameter choices are unfeasible due to calculation complexity. This text documents the implementation of the downhill simplex method, an optimisation algorithm which is capable of traversing a search space of such parameters and generating new combinations that conform better to a set of conditions such as adherence of calculated energies to reference values and similar properties. The algorithm is successfully tested with nitrogen setups, and generated setups produce results which seem to be better than default values. Further tweaking and testing can probably improve the algorithm considerably.

Contents

1 Introduction

GPAW[1], for Grid-based Projector-Augmented Wave method, is a Python software library implementing the projector-augmented wave method in a real-space grid, which is used for numerical calculations in density functional theory. GPAW calculations involving an element, say nitrogen, use an element-specific *setup* which defines suitable DFT parameters such as pseudo-wave function cutoff radius.

The purpose of this project is to find better GPAW setups using an optimisation procedure which can generate and evaluate setups. The input values are various setup generation parameters, and the output is a new set of such parameters generating a setup which (hopefully) yields better results than the original.

We are in this case concerned only with the case of nitrogen and shall use only the PBE approximations for exchange-correlation energies.

The following sections will provide a short summary of density functional theory and the projector augmented wave method. Further, section 2 provides a complete – and more technical – overview of the objectives and methods to be used. Section 3 documents efficiency and precision considerations plus the design of the functions used to evaluate setups. Finally Section 4 deals with the optimisation algoritm itself and the results.

It turns out that the algorithm successfully generates setups which do appear to yield better results. More testing and general assessment still needs to be done, perhaps mostly parameter tweaing. Longer optimisation runs might provide more information.

I am highly grateful for Jens Jørgen Mortensens much-needed help in using GPAW.

Thanks also to Jan Rossmeisl for general information about density functional theory.

1.1 Density functional theory

In 1964 it was shown by Hohenberg and Kohn[5] that the electron density $n(r)$, a quantity of much greater simplicity than the entire electronic wave function, is sufficient to derive all observable quantities with regards to the electronic behaviour of a quantum mechanical system.

Specifically, two theorems were proven for the ground states and energies of many-particle electronic systems:

- The energy of a system is a functional $E[n]$ of the electronic density $n(r)$. This implies e.g. that two wave functions corresponding to different energies cannot correspond to the same density. Furthermore the electron density *uniquely determines the wave function.*
- The energy functional has a minimum $E[n_0]$ at the electronic ground state n_0.

The electron-electron interactions turn out to be highly expensive to include in calculations. The *Kohn-Sham equations* provide one solution to this problem. Using these it is possible to convert the N-particle problem into N one-particle problems, wherein the effect of the multitude of electrons – barring the one under consideration – is replaced by an *effective potential* V_{eff}. There does exist a choice of potential which emulates the presence of other electrons *exactly*, but unfortunately the derivation of this potential is exceedingly difficult except for very simple systems such as the free electron gas. For more complex systems variational methods are usually applied to determine the effective potential.

A host of different methods have been used to implement density functional theory, some of which use projector-augmented waves.

1.2 Grid-based Projector Augmented Wave method

GPAW is an implementation of the projector augmented wave method[2] using a real-space grid rather than, say, its fourier transform, a k-space grid. The all-electron system is subjected to a transform which allows the inner regions to be treated with spherical symmetry, whereas the outer region, with the valence electrons, are treated differently. "Projector functions" are used to join (or augment) the two solutions at some radius from atom cores. In the outer region, the presence of the inner electrons is emulated by a smooth pseudo-wave function. Suitable variables that can be optimised, such as in this report, include the cutoff radii for the pseudo-wave functions.

2 The Grand Plan

The stated objective is to derive a method whereby it is possible to determine the in some sense "optimal" parameters used to generate *setups* for GPAW.

GPAW *setups* are sets of information pertaining to specific atoms. There is one setup for nitrogen using the PBE exchange-correlation functional, and this is the setup we are interested in optimising. One such parameter to be varied is the cutoff radius from atom cores within which pseudo-wave functions are used, but generally the setup generation function serves as a black box, the details of which are not critical.

2.1 Main properties of the algorithm

The algoritm to be used is the *downhill simplex method*, which will be explained in detail in Section 4.1. The method finds a (local) minimum of a function $f : \mathbb{R}^n \mapsto \mathbb{R}$ by evaluating function values only (not, say, derivatives).

The function to be minimized can therefore depend on any number of GPAW setup parameters, and it must return something indicative of the setup quality (henceforth referred to as the *badness* since it is to be minimized).

2.2 Test function principles

Evaluation of a setup involves five measures pertaining to precision, robustness and efficiency, each of which can be calculated independently:

- The deviation of atomisation energy from reference value
- The deviation of bond length from reference value
- The sensitivity to sub-resolution coordinate translations
- The rate of solution convergence with resolution
- Calculation time

The badness function is defined as a weighted square sum of these. A GPAW setup which scores well on all five tests is likely to be a good overall setup – if subsequent tests show the contrary, the tests can be adjusted or expanded, and the weights can be changed. The precise specifications for these test functions will be decided in Section 3.3

In conclusion: An optimisation process involves generating an initial setup with sensible parameters (Section 3.4), then using the simplex algorithm (Section 4.1) to adjust the parameters by means of repeated evaluations of the badness function (Section 3.3).

Atomic data for nitrogen	
E_a (PBE)	10.55 eV
E_a (GPAW)	10.62 eV
Bond length	1.103 Å
Magnetic moment (N)	3
Magnetic moment (N_2)	0
Band count (N)	4
Band count (N_2)	5

Table 1: *Atomic data for nitrogen. The PBE energy value is taken from [2], and the GPAW value from [3].*

In the next section GPAW calculations will be used to design these five test functions, taking into account the necessary resolutions, cell sizes and other factors.

3 Designing a setup evaluation function

The purpose of this section is to determine the cell sizes and resolutions which must be used in order to obtain sensible results. The relevant atomic data for nitrogen used in the calculations are shown in Table 1. Recall that we will use the PBE functional for exchange and correlation. For this reason the calculated PAW energies are compared to those derived by all-electron PBE methods, which means a reference atomisation energy[1] of 10.55 eV rather than the experimental value of 9.91 eV.

3.1 Cell configurations and energy calculations

Two different calculations are going to be relevant:

- The ground state energy $E[\mathrm{N}_2](d)$ of a system consisting of two nitrogen atoms at some separation d (not necessarily the bond length). The atoms are put on the x-axis with the desired separation, i.e. at $\boldsymbol{a}/2 \pm (d/2, 0, 0)$ for a cell of size $a \times a \times a$).
- The atomization energy $E_a = E[\mathrm{N}_2](d_0) - 2E[\mathrm{N}]$, i.e. the difference between the energy of a nitrogen molecule and twice that of an isolated nitrogen atom. The molecular energy is obtained by spacing the atoms like in the previous case but with fixed separation equal to the bond length d_0. The calculations on isolated atoms are performed with the atom located at the center of the otherwise identical unit cell.

These functions can be run with any cell size a and any grid spacing h. The duration of a calculation is highly dependent on these factors since they determine the total number of grid points. Other variables are generally left at the GPAW calculator defaults – exceptions to this will be specified when appropriate. The Python functions implementing these calculations are available in Appendix A.1, and the relevant functions are `energyAtDistance` and `calcEnergy`, respectively.

The setup in the case of two atoms can be seen on Figure 1. Next, a number of calculations will be made to test these functions.

[1] See [2], or [6]

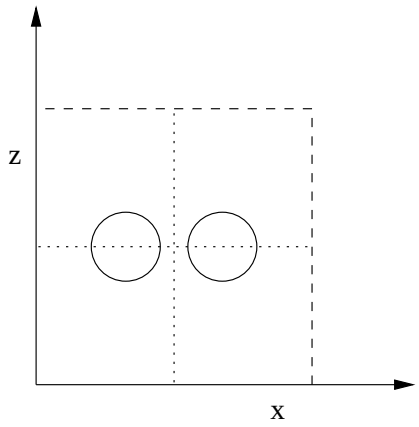


Figure 1: *Configuration with two nitrogen atoms. The y axis is collapsed. The dashed lines indicate the unit cell, and the dotted lines the centre, symmetrically about which the two atoms are arranged.*

3.2 Preliminary calculations

As an overall saneness-test, Figure 2a shows the ground-state energy of a two-molecule system as a function of the interatomic distance, i.e. an evaluation of $E[\mathrm{N}_2](d)$ for $d = 0.6 \ldots 2$ Å using 64 points with cell size $a = 6$ Å and resolution $h = 0.2$ Å (The GPAW calculator parameter `lmax`[2] is set to 0 instead of the default value 2 during this calculation because of errors occuring at smaller distances.).

Furthermore a very precise evaluation of E_a using a fine grid $h = 0.15$, a large cell $a = 10$ and `lmax=2` yields a result of 10.60 eV which is very close to the GPAW value from Table 1. Thus the written functions produce sensible results.

Next we shall test the impact of cell size on precision. Figure 2b shows a plot of the atomization energy as a function of cell sizes 0.4 to 9.5 Å for $h = 0.2$ Å (this plot uses non-default `lmax=0` since the calculations crash for small cell sizes otherwise). A cell size of 6Å should be reasonable for most calculations.

Figure 2c shows the ground state energy of two nitrogen atoms as a function of the grid resolution h. This plot also uses `lmax=0`. This calculation is done to investigate convergence rather than an actual value, so the cell size is set to 4 Å in order to speed up calculation. Most calculations will use $h = 0.2$ Å since smaller values take too much time, even though smaller values still increase precision.

If the system is translated a small distance less than the resolution h, numerical effects regarding the grid resolution will likely cause small undesirable deviations in the calculated energies. In order to examine this effect, consider Figure 2d. The figure shows the calculated energy of a nitrogen molecule aligned along the x-axis as a function of its dislocation along the z-axis from 0 to h from the center of the unit cell. The plot is made with the (small) cell size $a = 4.0$ Å (the cell size must be largely irrelevant for this effect) and resolution $h = 0.2$ Å. The system is periodic such that the part of the system which slides out one side of the unit cell due to the dislocation does not have impact on the result. This should mean that the deviation is periodic, which is indeed the case. Clearly

[2] `lmax` is the "maximum angular momentum for expansion of compensation charges"[4] and defaults to 2. For "difficult" geometries this choice sometimes crashes the calculations with an error about charge conservation violation.

the maximum energy fluctuation corresponds to dislocations of 0 and $h/2$, meaning that the magnitude of this effect can be determined simply by taking the difference between the energies at dislocations 0 and $h/2$:

$$\delta E = E_{z=0} - E_{z=h/2}. \tag{3.1}$$

3.3 Test functions

Five different tests are used to evaluate a GPAW setup, see Table 2.

1. The Energy test calculates E_a like above for $a = 6$ Å and compares it to the reference value.

2. The Distance test calculates the bond length deviation from the reference value. This is done by evaluating $E[\mathrm{N}_2]$ at three points close to the reference value, fitting with a parabola and finding the minimum. Presently a spatial deviation 0.038 Å is used, giving an energy deviation of around 0.1 eV. Thus the three points of evaluation are set to d_0 and $d_0 \pm 0.038$ Å. This test uses the parameter `lmax=2` since this improves precision.

3. The Fluctuation test finds the sub-grid-resolution energy fluctuation amplitude by evaluating $E[\mathrm{N}_2]$ at the center $\boldsymbol{c}$ of the unit cell, and then again at $\boldsymbol{c} + (0, 0, h/2)$ like in Equation (3.1). It might be better to do a test along the other axes as well, but this presently has not been implemented because it increases calculation time.

4. The Convergence test evaluates $E[\mathrm{N}_2]$ for the three different h values 0.2, 0.17 and 0.15 angstroms. If the energy difference is small, the energy estimate at $h = 0.2$ Å was good, meaning that the solution converges properly. The test returns the difference between the largest and the smallest energy value.

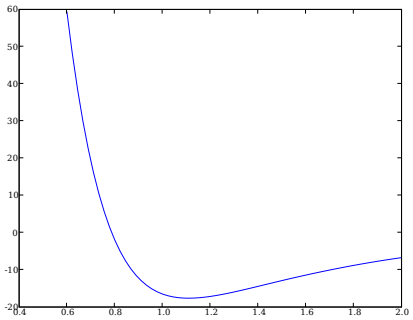
5. Finally the Time test is supposed to evaluate the CPU time necessary to solve a problem. This has not been implemented, so presently the wall-clock time T of the other tests is used instead, which is crude, but works. If this test is not performed then the algorithm might return a very precise but practically unusable setup.

The five tests are combined into a single measure of the *setup badness* by taking a weighted square sum of all the test results (subtracting reference values where applicable). The weights are selected such that a badness of 1 corresponds to a particular test result, and these definitions can be seen in Table 2. If a test fails, i.e. an exception is thrown during calculations, a badness of 10000 is returned as "penalty". The source code can be found in Appendix A.3, and the relevant function is called `badness`.

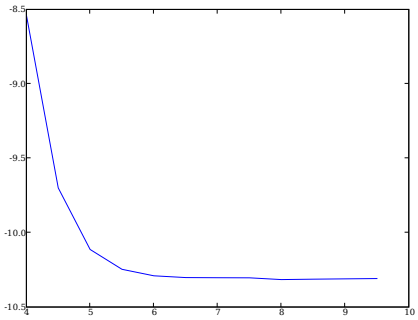
3.4 Setup generation

For reference, the parameters supplied to the setup generator along with default "sensible" values are listed in table 3. Optimisation run may optionally be run without all of these variables due to time constraints – the remaining parameters are made dependent on the previous ones (see Appendix A.2 for source) to reduce the dimension of the search space and thus calculation time.

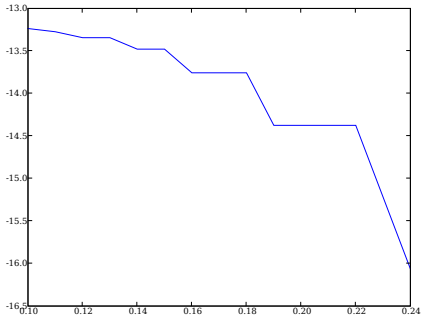
The next section concerns the implementation of the actual algorithm.



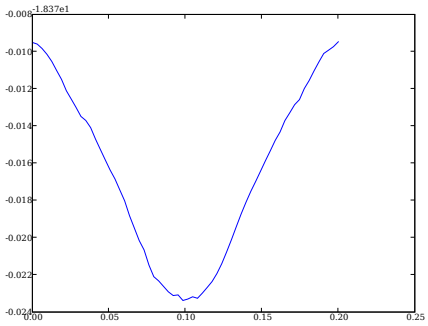
(a) *The energy of a system consisting of two nitrogen atoms as a function of separation. The known bond length 1.103 Å corresponds well to the location of the minimum.*



(b) *Energy as a function of cell size. 6 Å or higher is required for reasonable precision*



(c) *The ground state energy as a function of grid resolution h. The plateaus appear because some adjacent h-values correspond to the same number of actual grid points.*



(d) *The energy as a function of translation along the z-axis of a nitrogen molecule aligned along the x-axis. The amplitude is around 13.9 meV, and the deviation is clearly maximal between dislocations of 0 and h/2.*

Figure 2: *Various tests used to decide suitable cell and GPAW calculator properties.*

Test overview

Name	a	Unit badness
Energy test	6.0	0.05 eV
Distance test	5.5	0.005 Å
Fluctuation	4.0	5 meV
Convergence	4.0	0.2 eV
Time	-	20 s

Table 2: *The five tests, the used cell sizes and the test result wich is defined to have a badness value of 1. All tests use $h = 0.2$ where applicable, except the convergence test which varies h.*

Default setup parameters

Name	Default	Description
`r`	`1.1`	Cutoff radius for projector functions
`rvbar`	`r`	Cutoff radius zero potential
`rcomp`	`r`	Cutoff radius for compensation charges
`rfilter`	`2 r`	Cutoff for fourier-filtered projector function
`hfilter`	`0.4`	Target grid spacing

Table 3: *The parameter identifiers and default values supplied to the setup generator. These values are the ones adjusted when the algorithm runs. All values are in Bohr units.*

4 Optimisation procedure

By now we have defined a set of parameters to optimise, along with function to optimise. Only the algorithm remains.

4.1 The downhill simplex algorithm

An n-dimensional simplex is the convex hull bounded by $n+1$ (affinely independent) points plus their interconnecting lines and faces. For example a two-dimensional simplex is a triangle, and a three dimensional one is a tetrahedron.

The *downhill simplex method*[7, pp. 305–309] is an algorithm which can be used to minimise a function $f : \mathbb{R}^n \mapsto \mathbb{R}$ by evaluating the function on the $n+1$ vertices of a simplex in $\mathbb{R}^n$, then repeatedly moving the least-favourable points of the simplex (i.e. those corresponding to high function values) in the general direction of the more favourable points, possibly past them, and reevaluating f at the new location. The simplex will thus be made to move across the parameter space $\mathbb{R}^n$ until hopefully a minimum of f is obtained.

A detailed description of the algorithm follows.

Initialize a simplex with "reasonable" parameter values as the $n+1$ vertices $\boldsymbol{p}_1 \ldots \boldsymbol{p}_n$, and evaluate f there. Then repeat the following steps:

- Find the vertex indices $i_{\text{high}}, i_{\text{2nd high}}$ and i_{low} with highest, second-highest and lowest function values.
- Calculate the relative difference $|\frac{y_{\max}-y_{\min}}{y_{\max}+y_{\min}}|$ between maximum and minimum function values. If this number is smaller than some tolerance, there is no appreciable variation in the function here meaning that the vertices have converged on a minimum, and the algorithm terminates. Otherwise continue.
- Reflect the point $\boldsymbol{p}_{i_{\max}}$ with highest function value through the opposite simplex face and evaluate f here.
- If this yields a lower function value than the hitherto lowest, extrapolate some extra distance (say, twice the distance) in the same direction and evaluate f there.
- Else:

– If the reflected point is worse than the second-highest existing point, the minimum probably lies between the existing points; thus move the point back to a location halfway between the original position $\boldsymbol{p}_{i_{\max}}$ and the opposite face, and evaluate f.

– If this new point *still* has the highest function value aomg the vertices, the minimum must be near the currently best point. Contract *all* other points halfway towards the best point and evaluate f at these all locations.

The source code can be found in Appendix A.3. The algoritm is split into two functions, namely `amoeba` and the helper function `amotry`.

4.2 Running the algorithm

The algorithm can be run with any number of parameters. A start simplex of appropriate dimension is generated pseudorandomly. Recall that most of the default parameter values are around 1.1, so the initial points are distributed within 0.1 of the default values. The algorithm writes parameters, badness values and tolerance evaluations to a log file. It also saves the parameters and badness values of the last iteration in a dump file which can be used as initial conditions for another test run.

Test runs presently take very long time. Only one test run has been made where all five parameters are varied. For reasons of stability, all calculations use `lmax=0`

4.3 Results and analysis

The result of different test runs can be seen in Table 4. Consider the test run where all five parameters are varied. A subsequent calculation using $a = 7$ Å yields an atomisation energy of $E_a = 10.50$ eV, considerably closer to the reference value of 10.55 eV than the result 10.39 eV of the similar calculation using the default setup. However this is not exactly surprising since the optimisation algorithm is *designed* to optimise exactly this kind of problem. More general tests will have to be performed in order to better evaluate the quality of the optimised setup, but time constraints prevent large-scale testing.

Figure 3 shows during the 5-parameter run the evolution of the variable which is compared to tolerance during each iteration, i.e. the variable which ends the algorithm when it gets small enough. Large values of this variable tends to indicate that the vertices are moving large distances (or that the function to be optimised is oscillating weirdly). It can be seen that while the value falls off and rises again repeatedly. The explanation for this is most likely that the simplex contracts and expands "like an amoeba" when traversing through shallow paths in the parameter space. However it also means that it might take considerably more iterations than indicated, since the value might rise again had a lower termination value been used. Note also that while there are 37 values on the graph, the algorithm uses trial-and-error to decide whether to expand or contract – this means that frequently more than one function evaluation is done per step. In this case there were 79 function evaluations in total.

Figure 4 shows the badness progression for each vertex during the 5-variable run. The values peak sharply at several points. This can only happen when the simplex is contracted, which indicates that the algorithm thinks it has found a minimum. Evidently this is not quite the case. A likely explanation is that the badness function is not very smooth (which would make the optimisation proceed similarly smoothly), but full of 5-dimensional saddle points which slow the algorithm. It is unclear how long the algorithm

Initial and optimised setup parameters

Name	Default	Optimised
`r`	1.1	1.021
`rvbar`	1.1	1.081
`rcomp`	1.1	1.148
`rfilter`	2.2	2.288
`hfilter`	0.4	0.4525

Table 4: *The initial parameters and those obtained by running the optimisation algorithm.*

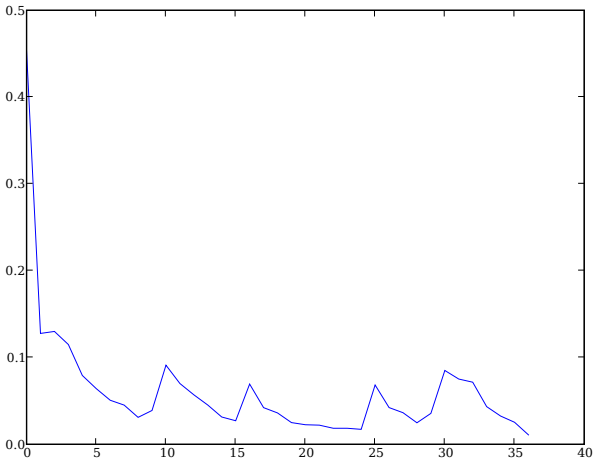


Figure 3: *Progression of the value of the termination parameter during an optimisation run.*

can run before finding an *actual* minimum. Nonetheless the derived parameters do yield considerably different results as shown above.

Figure 5 shows the convergence of the parameter `r` as the algorithm progresses. This plot is made during a two-parameter optimisation, so there are three simplex vertices corresponding to three curves. In this case the values seem to converge quite well in a limited amount of evaluations. The same will probably happen in more dimensions although, as we have seen, the procedure takes more iterations to settle when more points have to be moved.

5 Conclusion

The proposed algorithm has been written and is capable of generating GPAW nitrogen setups which seem to yield values better than those of the default setup. The algorithm has successfully been tested in a 5-dimensional parameter space. The algorithm relies on five different tests to assess the quality of a given setup.

The parameter space is somewhat difficult and time consuming to traverse, and the

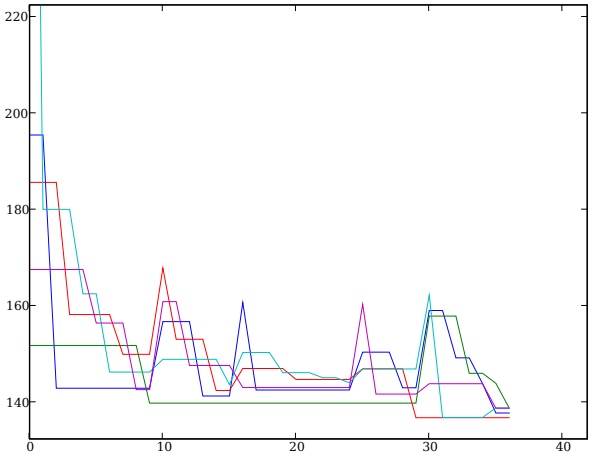


Figure 4: *The badness values at each vertex during the 5-dimensional optimisation run. The values peak sharply at several points. This can only happen when the simplex is contracted, meaning that significant deviations are observed even with small parameter changes.*

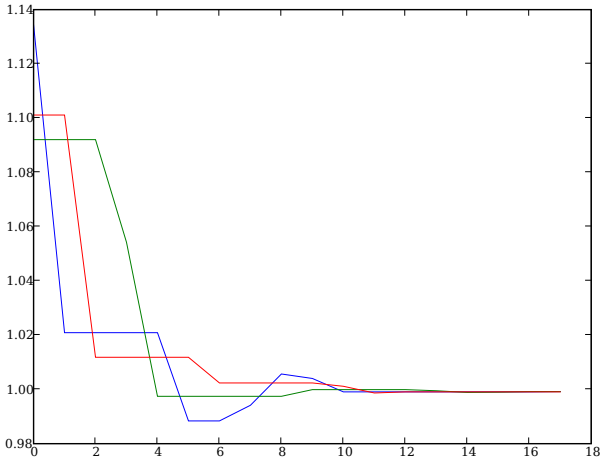


Figure 5: *The convergence of the first coordinate of each of the three vertices during a lengthy two-variable optimisation run.*

algorithm frequently speeds up after looking as if it were about to converge. For this reason it is difficult to tell actual minima from what turns out to be saddle points. This is in part a feature of the algorithm, and may not be a large problem. A very long test run could be made to check the algorithm behaviour more properly.

It is possible that the present form of the badness function, i.e. as a sum of squares, is not optimal. If one parameter outweighs other parameters considerably, some tests will have little impact on the overall badness. This is partly remedied by selecting proper weights, but there is no particular reason why a parabolic expressions should be inherently better than, say, fourth order ones. More theoretical consideration might be given to the badness function.

References

[1] GPAW home page: `https://wiki.fysik.dtu.dk/gpaw`

[2] J. J. Mortensen, L.B. Hansen and K. W. Jacobsen: *Real-space grid implementation of the projector augmented wave method*, Phys. Rev. B **71**, 035109. 2005.

[3] GPAW molecule tests at experimental geometries, as of May 11, 2007.
`https://wiki.fysik.dtu.dk/gpaw/Molecule_Tests`

[4] The GPAW manual as of May 11, 2007.
`https://wiki.fysik.dtu.dk/gpaw/Manual`

[5] P. Hohenberg and W. Kohn, 1964, Phys. Rev. **136**, B864.

[6] S. Kurth, J. P. Perdew and P. Blaha: *Molecular and Solid State Tests of Density Functional Approximations: LSD, GGAs, and Meta-GGAs.* Int. J. Quant. Chem. 75, 889-898. 1999.

[7] W.H. Press, B.P. Flannery, S.A. Teukolsky, W.T.Vetterling: *Numerical Recipes in C: The Art of Scientific Computing*, Cambridge University Press. 1988.

A Appendix

A.1 atomization.py

```
#!/usr/bin/python

import pylab

from ASE import Atom, ListOfAtoms
from gpaw import Calculator

"""
Utility class wrapping molecule informations
nbands1 and 2 are the numbers of bands to be used with single-atom and
molecular calculations, respectively
"""
class MoleculeInfo:
    def __init__(self, letter, d, magmom, nbands1, nbands2):
        self.d = d
        self.letter = letter
        self.magmom = magmom
        self.nbands1 = nbands1
        self.nbands2 = nbands2

#Also test H2 (4.5 eV). magmom=1, nbands=1 in Calculator for H as well as H2

molN = MoleculeInfo('N', 1.103, 3, 4, 5)
molH = MoleculeInfo('H', 0.740, 1, 1, 1)

"""
Creates an atom. If a separation greater than 0 is specified, creates
two atoms correspondingly spaced along the x axis.

Returns a ListOfAtoms containing whatever was created in this way
"""
def getListOfAtoms(molecule=molN, separation=0, a=5., dislocation=(0.,0.,0.), periodic
    =False):
    atoms = None
    (dx, dy, dz) = dislocation
    (cx, cy, cz) = (a/2. + dx, a/2. + dy, a/2. + dz)
    d = separation/2.
    if separation==0:
        #One atom only
        atoms = ListOfAtoms([Atom(molecule.letter, (cx, cy, cz),
                    magmom=molecule.magmom)],
                    periodic=periodic,
                    cell=(a,a,a))
    else:
        #Create two atoms separated along x axis
        #No magnetic moment then!
        atoms = ListOfAtoms([Atom(molecule.letter, (cx+d,cy,cz)),
                     Atom(molecule.letter, (cx-d,cy,cz))],
                    periodic=periodic,
                    cell=(a,a,a))
    return atoms

"""
Calculates the atomization energy, i.e. E[N2] - 2*E[N] where N2 and N denote
nitrogen molecule and atoms, respectively
"""
def calcEnergy(calc1=None, calc2=None, a=4., molecule=molN, dislocation=(0,0,0),
    periodic=False, setup='paw'):

    oneAtom = getListOfAtoms(molecule, a=a, dislocation=dislocation,
                             periodic=periodic)

    if calc1 == None:
        calc1 = MakeCalculator(nbands=molecule.nbands1, setup=setup)
    if calc2 == None:
        calc2 = MakeCalculator(nbands=molecule.nbands2, setup=setup)

    #bands: 2s and 2p yield a total of 4 bands; 1s is ignored
```

```
67      #setups='A1' => will search for /home/ask/progs/gpaw/setups/N.A1.PBE.gz
68      oneAtom.SetCalculator(calc1)
69      e1 = oneAtom.GetPotentialEnergy()
70
71
72      #gpts=(n,n,n) - to be varied in multiples of 4
73      d = molecule.d
74
75      twoAtoms = getListOfAtoms(molecule, a=a, dislocation=dislocation,
76                                periodic=periodic, separation=molecule.d)
77
78      #10 electrons in total from 2s and 2p.
79      #Thus it is necessary only to include 5 bands
80      twoAtoms.SetCalculator(calc2)
81      e2 = twoAtoms.GetPotentialEnergy()
82
83      return e2-2*e1
84
85  """
86  Using a particular resolution h, test whether energies deviate considerably
87  if the system is translated in intervals smaller than h.
88  """
89  def displacementTest(a=5., molecule=molN, h=.2):
90      print 'Displacement test:', molecule
91
92      h += 0. #floating point
93
94      testcount = 3
95      dislocations = []
96
97      #Initialise test coordinates
98      for value in range(testcount):
99          coordinate = h * value/testcount #linear distribution
100         dislocations.append((coordinate, 0., 0.))
101
102     print dislocations
103
104     energies = []
105     for dislocation in dislocations:
106         #e = calcEnergy(a, molecule, dislocation, h)
107         e = energyAtDistance(molecule.d, dislocation, h)
108         energies.append(e)
109
110     print 'Energies:'
111     print energies
112
113     print 'Max',max(energies)
114     print 'Min',min(energies)
115     print 'Diff',max(energies) - min(energies)
116
117 """
118 Creates two calculators for the given molecule with appropriate band counts
119 """
120 def atomizationCalculators(molecule=molN, out='-', h=.2, lmax=0, setup='paw'):
121     calc1 = MakeCalculator(molecule.nbands1, out, h, lmax, setup=setup)
122     calc2 = MakeCalculator(molecule.nbands2, out, h, lmax, setup=setup)
123     return (calc1, calc2)
124
125
126 """
127 Default calculator setup, however complicated it might become someday
128 This method allows you to forget about lmax and PBE and such
129 """
130 def MakeCalculator(nbands, out='-', h=.2, lmax=0, setup='paw'):
131     return Calculator(nbands=nbands, out=out, h=h, lmax=lmax, xc='PBE',setups=setup)
132
133 """
134 Calculates the ground-state energy of the given molecule when the atoms
135 are spaced by the given distance
136 """
137 def energyAtDistance(distance, calc=None, dislocation=(0,0,0),
138                      molecule=molN, a=5., periodic=False):
139     c = a/2.
140     (dx, dy, dz) = dislocation
```

```
141
142     coord1 = (c-distance/2. + dx, c + dy, c + dz)
143     coord2 = (c+distance/2. + dx, c + dy, c + dz)
144
145     twoAtoms = getListOfAtoms(molecule, distance, a, dislocation,
146                               periodic)
147
148     if calc == None:
149         calc = MakeCalculator(nbands=molecule.nbands2)
150
151     twoAtoms.SetCalculator(calc)
152
153     energy = twoAtoms.GetPotentialEnergy()
154     return energy
155
156 """
157 Write␣lists␣x␣and␣y␣to␣specified␣file
158 """
159 def writeResults(x, y, fileName, header=[]):
160     if len(x) != len(y):
161             raise Exception('Result␣list␣length␣mismatch')
162     length = len(x)
163     f = open(fileName, 'w')
164     lines = [''.join([str(x[i]),'\t',str(y[i]),'\n']) for i in range(length)]
165
166     for line in header:
167         line = '#␣'+line
168
169     f.writelines(header)
170
171     f.writelines(lines)
172     f.close()
173
174 """
175 Read␣list␣of␣(x,y)␣entries␣from␣datafiles,␣return␣as␣two␣lists
176 """
177 def readResults(fileName):
178     f = open(fileName, 'r')
179     lines = filter(stringFilter, f.readlines())
180     length = len(lines)
181     pairs = [s.split() for s in lines]
182     x = [float(pair[0]) for pair in pairs]
183     y = [float(pair[1]) for pair in pairs]
184     return (x,y)
185
186 """
187 Allow␣comments␣and␣empty␣lines␣in␣data␣files
188 """
189 def stringFilter(s):
190     return not (s.startswith('#') or s.isspace())
191
192 """
193 The␣gbar␣doesn't␣have␣pylab␣so␣use␣this␣function
194 """
195 def linspace(start, end, count):
196     return [start + float(i)/(count-1)*(end-start) for i in range(count)]
```

A.2 setupgen.py

```
1  import os
2  from gpaw.atom.generator import Generator
3
4  class SetupGenerator:
5
6      def __init__(self, name):
7          #We don't want anything to mess up with existing files
8          #so make sure a proper name is entered with a couple of chars
9          #(it should be enough to test for len==0, but what the heck)
10         if len(name) < 3:
11             raise Exception
12         self.name = name
13
14
```

```
15      def new_nitrogen_setup(self, r=1.1, rvbar=None, rcomp=None,
16                             rfilter=None, hfilter=0.4):
17          """Generate new nitrogen setup.
18
19      The new setup depends on five parameters (Bohr units):
20
21      * 0.6 < r < 1.9: cutoff radius for projector functions
22      * 0.6 < rvbar < 1.9: cutoff radius zero potential (vbar)
23      * 0.6 < rcomp < 1.9: cutoff radius for compensation charges
24      * 0.6 < rfilter < 1.9: cutoff radius for Fourier-filtered
25        projector functions
26      * 0.2 < hfilter < 0.6: target grid spacing
27
28      Use the setup like this::
29
30          calc = Calculator(setups={'N': 'opt'}, ...)
31
32      """
33
34          if rvbar is None:
35              rvbar = r
36          if rcomp is None:
37              rcomp = r
38          if rfilter is None:
39              rfilter = 2 * r
40
41          g = Generator('N', 'PBE', scalarrel=True, nofiles=True)
42          g.run(core='[He]',
43                rcut=r,
44                vbar=('poly', rvbar),
45                filter=(hfilter, rfilter / r),
46                rcutcomp=rcomp,
47                logderiv=False)
48          path = os.environ['GPAW_SETUP_PATH'].split(':')[0]
49          os.rename('N.PBE', path + '/N.'+self.name+'.PBE')
50
51      def f(self, par):
52          self.new_nitrogen_setup(*par)
53
54          #new_nitrogen_setup(1.1, 1.1, 1.1, 1.9, 0.4)
55          #f([1.2])
56          #f([1.2, 1.0, 1.0])
```

A.3 simplex.py

```
1  #!/usr/bin/python
2
3  import atomization, setupgen
4  import sys, pickle, random
5  from LinearAlgebra import inverse
6  import Numeric as N
7
8  from datetime import datetime, timedelta
9
10 N_MAX = 100
11 ALPHA = 1.
12 BETA = .5
13 GAMMA = 2.
14
15
16 """
17 Default test function with one minimum at (1,2,3, ...).
18 The minimum is exactly 42. Takes a list of coordinates as an argument
19 and returns a number.
20 """
21 def standardFunction(p):
22     y = 42
23     for i in range(len(p)):
24         y += (p[i]-(i+1))**2
25     return y
26
27 """
28 Performs the 'amoeba'-like downhill simplex method in ndim dimensions.
```

```
29
30 p: a list of (ndim+1) vectors each with ndim coordinates,
31 corresponding to the vertices of the simplex.
32
33 y: a list of function values evaluated at the vertices, ordered consistently with the
       vertices in p. y thus must have length (ndim+1) as well
34
35 ndim: the dimension count of the space in question. Of course this variable is mostly
       for show since it's not really necessary in python
36
37 fTolerance: fractional tolerance used to evaluate convergence criterion
38
39 function: the function to be minimized. The function must take exactly
40 ndim parameters, each parameter being one number
41
42 maxIterations: the maximal number of iterations to be performed before
43 returning, in case convergence is slow
44
45 Returns the number of times the function has been evaluated during the
46 procedure.
47
48 After invocation the argument lists p and y will have been modified to contain
49 the simplex vertices and associated function values at termination of the
50 procedure.
51
52 """
53 def amoeba(p, y, ndim, fTolerance, function=standardFunction, out=sys.stdout, dump='
       lastdump.dump.pckl'):
54
55     mpts = ndim + 1
56     evaluationCount = 0
57     #This is probably the coordinate sum, i.e.
58     #it probably has to do with the geometric center of the simplex
59     psum = getpsum(p)
60
61     while True:
62         print >> out, 'Points:', p
63         print >> out, 'yValues:', y
64         print >> out, 'EvalCount:',evaluationCount
65         print >> out
66         out.flush()
67
68         #Write current points to file for recovery if something goes wrong
69         pickleDump((p,y),dump)
70
71         iLow = 0 #index of lowest value
72         iHigh = None #index of highest value
73         i2ndHigh = None #index of second highest value
74         if y[0] > y[1]:
75             (iHigh, i2ndHigh) = (0, 1)
76         else:
77             (iHigh, i2ndHigh) = (1, 0)
78
79         #Loop through vertices to find index values for highest/lowest entries
80         for i in range(mpts):
81             if y[i] < y[iLow]:
82                 iLow = i
83             if y[i] > y[iHigh]:
84                 i2ndHigh = iHigh
85                 iHigh = i
86             elif y[i] > y[i2ndHigh]:
87                 if i != iHigh:
88                     i2ndHigh = i
89
90         #Things should be floats already, but it's good to be safe
91         relDeviation = float(abs(y[iHigh] - y[iLow]))/abs(y[iHigh]+y[iLow])
92
93         print >> out,'Rel. deviation', relDeviation
94         out.flush()
95
96         if relDeviation < fTolerance:
97             break
98
99         if evaluationCount >= N_MAX:
```

```
100             print '=== Max evaluation count',N_MAX,'exceeded, terminating! ==='
101             #Some would call this an error, but we'll just return
102             #as if nothing has happened
103             break
104 
105         yTry = amotry(p, y, psum, ndim, function, iHigh, -ALPHA)
106         evaluationCount += 1
107 
108         if yTry <= y[iLow]:
109             yTry = amotry(p, y, psum, ndim, function, iHigh, GAMMA)
110             evaluationCount += 1
111         elif yTry >= y[i2ndHigh]:
112             ySave = y[iHigh]
113             yTry = amotry(p, y, psum, ndim, function, iHigh, BETA)
114             evaluationCount += 1
115             if yTry >= ySave:
116                 for i in range(mpts):
117                     if i != iLow:
118                         for j in range(ndim):
119                             psum[j] = .5 * (p[i][j] + p[iLow][j])
120                             p[i][j] = psum[j]
121                         y[i] = function(psum)
122                 evaluationCount += ndim
123                 psum = getpsum(p)
124 
125     return evaluationCount
126 
127 """
128 Extrapolates through or partway to simplex face, possibly finding a better
129 vertex
130 """
131 def amotry(p, y, psum, ndim, function, iHigh, factor):
132     #Wonder what these 'factors' do exactly
133     factor1 = (1. - factor)/ndim
134     factor2 = factor1 - factor
135 
136     pTry = [psum[j]*factor1 - p[iHigh][j]*factor2 for j in range(ndim)]
137 
138     yTry = function(pTry)
139 
140     if yTry < y[iHigh]:
141         y[iHigh] = yTry
142         for j in range(ndim):
143             psum[j] += pTry[j] - p[iHigh][j]
144             p[iHigh][j] = pTry[j]
145 
146     return yTry
147 
148 
149 """
150 Given a list of (ndim+1) vectors each with ndim coordinates,
151 returns the list of coordinate sums across vectors,
152 i.e. the n'th element is the sum of the n'th coordinates of all vectors in p
153 """
154 def getpsum(p):
155     mpts = len(p)
156     ndim = mpts - 1
157 
158     psum = array(ndim)
159     for i in range(ndim):
160         psum[i] = sum([q[i] for q in p])
161 
162     return psum
163 
164 
165 """
166 Returns a list of vertex coordinates forming a regular simplex around the
167 designated center, where the size argument is the max vertex-center distance.
168 
169 This method simply generates a random simplex, and may fail to do so at a
170 very small probability (if randomly generated vectors are linearly dependent)
171 """
172 def getInitialPoints(center=[0,0], size=1, seed=0):
173     ndim = len(center)
```

```
174     mpts = ndim + 1
175     r = random.Random(seed)
176 
177     points = array(mpts)
178     for i in range(ndim+1):
179         points[i] = [(r.random()-.5)*size+center[j] for j in range(ndim)]
180 
181     return points
182 
183 """
184 Runs␣the␣amoeba␣optimization␣function␣with␣sensible␣values
185 """
186 def smalltest():
187     f = standardFunction
188     p = getInitialPoints([7,3,2,6,3])
189     print 'Initial␣points␣gotten'
190     print 'Mapping␣p␣through␣f'
191     y = map(f, p)
192     print 'Done␣mapping'
193     ndim = len(p)-1
194     fTolerance = .000001
195 
196     amoeba(p, y, ndim, fTolerance, f)
197 
198     print 'Done!'
199     #print 'p', p
200     #print 'y', y
201     print 'p[0]',p[0]
202 
203 class SetupEvaluator:
204 
205     def __init__(self, setup):
206         setup = 'opt.'+setup
207         self.setup = setup
208         self.generator = setupgen.SetupGenerator(setup)
209 
210     """
211 ␣␣␣␣Runs␣a␣full␣test␣of␣a␣given␣GPAW␣setup
212 ␣␣␣␣"""
213     def badness(self, args):
214         refEnergy = -10.55
215         refDist = 1.102
216 
217         try:
218             self.generator.f(args) #new setup
219             print 'New␣setup␣created'
220 
221             overallBadness = 0
222 
223             startTime = datetime.now()
224 
225             print 'Calculating␣atomization␣energy'
226             energyBadness = 1/.05**2 #badness == 1 for deviation == .05 eV
227             (c1,c2) = atomization.atomizationCalculators(out=None,
228                                                          setup=self.setup)
229             Ea = atomization.calcEnergy(c1,c2,a=6.0)
230             print 'Energy',Ea
231             db = energyBadness * (Ea - refEnergy)**2
232             print 'Energy␣badness',db
233             overallBadness += db
234 
235             print 'Calculating␣bond␣length'
236             d = bondLength(self.setup)
237             distanceBadness = 1./.005**2
238             db = distanceBadness * (d - refDist)**2
239             overallBadness += db
240             print 'Bond␣length',d
241             print 'Bond␣length␣badness',db
242 
243             print 'Calculating␣energy␣fluctuation␣amplitude'
244             DE = energyFluctuationTest(self.setup)
245             energyFluctuationBadness = 1./.005**2
246             db = energyFluctuationBadness * DE**2
247             overallBadness += db
```

```
            print 'Fluctuation magnitude',DE
            print 'Fluctuation badness',db

            print 'Calculating convergence rate'
            hVar = convergenceTest(self.setup)
            convergenceBadness = 1./.2**2
            db = convergenceBadness * hVar
            overallBadness += db
            print 'Energy difference',hVar
            print 'Energy difference badness',db

            print 'Calculating temporal badness'
            timeBadness = 1./20**2 #20 seconds --> badness == 1
            dt = (datetime.now() - startTime).seconds
            db =  timeBadness * dt**2
            overallBadness += db
            print 'Time',dt
            print 'Time badness',db

            print 'Overall badness',overallBadness

        except KeyboardInterrupt:
            raise KeyboardInterrupt #Don't ignore keyboard interrupts
        #except:
        #    return 10000.

        return overallBadness

"""
Returns the bond length. Calculates energy at three locations around the
reference bond length, interpolates with a 2nd degree polynomial and returns
the minimum of this polynomial which would be roughly equal to the bond length
without engaging in a large whole relaxation test
"""
def bondLength(setup):
    print 'Distance test'
    d0 = 1.102
    dd = ( .2 / 140. )**.5 #around .04 A. Bond properties correspond to
    #an energy of E = .5 k x**2 with k = 140 eV/A**2
    #If we want .1 eV deviation then the above dd should be used
    calc = atomization.MakeCalculator(atomization.molN.nbands2,
                                      out=None, setup=setup)
    D = [d0-dd, d0, d0+dd]
    #Calculate energies at the three points
    E = [atomization.energyAtDistance(d, calc=calc, a=5.5) for d in D]
    print 'Distances',D
    print 'Energies',E
    print
    #Now find parabola and determine minimum

    x = N.array(D)
    y = N.array(E)

    A = N.transpose(N.array([x**0, x**1, x**2]))
    c = N.dot(inverse(A), y)
    print 'Coordinates',c

    X = - c[1] / (2.*c[2]) # "-b/(2a)"
    print 'Bond length',X

    #print c
    #print N.dot(A, c) - y

    return X

"""
Returns whatever was written by pickledump
"""
def pickleLoad(fileName):
    content = pickle.load(open(fileName))
    return content

"""
Writes the data to the file, deleting any other content
```

```
"""
def pickleDump(data,fileName):
    pickle.dump(data, open(fileName, 'w'))

"""
Runs a complete optimization.

All files pertaining to the test will have file names containing
the testName parameter.

fTolerance is the termination tolerance of the
amoeba function.

If initData is specified it should be the file name of
a pickle file containing a set of points and values (p,y) that are
compatible with the amoeba function as specified in its documentation.
"""
def seriousTest(testName='test', fTolerance = .002, ndims=2, initData=None):

    fileName=testName+'.log'

    outFile = open(fileName, 'w')
    evaluator = SetupEvaluator(testName)

    f = evaluator.badness
    sensibledata = [1.1, 1.1, 1.1, 2.2, 0.4]
    #Defaults for new_nitrogen_setup. Warning: hardcoded here,
    #be sure to change this if new_nitrogen_setup is changed

    if initData is None:
        dumpFile = testName+'.dump.pckl'

        p = getInitialPoints(sensibledata[:ndims], size=.1)
        print >> outFile, 'Initial points gotten'
        print >> outFile, 'Points',p
        print >> outFile, 'Mapping p through f'
        outFile.flush()
        y = map(f, p)
        print >> outFile, 'Done mapping, dumping to file'
        print >> outFile, 'y values:',y
        pickleDump((p,y), dumpFile)
    else:
        (p,y) = pickleLoad(dumpFile)
        print >> outFile, 'Initial values loaded from',dumpFile
        print >> outFile, 'p: ',p
        print >> outFile, 'y: ',y

    outFile.flush()

    ndim = len(p)-1

    print >> outFile, 'Amoeba commencing'
    print >> outFile, '-----------------'
    outFile.flush()

    amoeba(p,y,ndim,fTolerance, f, out=outFile, dump=dumpFile)

"""
Plots the energy of a N2 moleculeas a function of different resolutions
(h-values) and returns the maximal difference
"""
def convergenceTest(setup):
    A = atomization
    h = [.15, .17, .20]
    calc = [A.MakeCalculator(A.molN.nbands2, out=None, h=h0,
                             setup=setup) for h0 in h]
    E = [A.energyAtDistance(A.molN.d, calc=c, a=4.) for c in calc]

    print 'h',h
    print 'E',E

    return max(E) - min(E)

"""
```

```
Returns the difference between the energy of a nitrogen molecule at the center
of the unit cell and the energy of one translated by h/2 along the z axis.
"""
def energyFluctuationTest(setup):
    A = atomization
    h = .2
    calc = A.MakeCalculator(A.molN.nbands2, out=None, setup=setup, h=h)
    d = A.molN.d
    E1 = A.energyAtDistance(d, calc=calc, a=4.)
    E2 = A.energyAtDistance(d, calc=calc, a=4., dislocation=(0.,0.,h/2.))

    return E2-E1

"""
'[None]*length' looks slightly silly so let's initialise our lists with a
nicely named function instead
"""
def array(length):
    return [None]*length
```