Thu Feb 24 04:27:06 CET 2011

JuNoLo PROGRAM MANUAL - version 1.02

Written by:
Predrag Lazić
plazic@mit.edu

Contents

1

Bugs removed from the original version

Original version of this code contained a bug that was causing it not to run properly with the Intel compiler. Also it turned out that it did not run with SGI compiler. The bug was finally resolved by Levent (sly5) from forum at The Center for Simulation and Modeling at the university of Pittsburgh.
`http://core.sam.pitt.edu/node/251#comment-1069`
Basically the declaration of variable status in param.f90 was wrong. Instead of being INTEGER :: status it must be INTEGER :: status(MPI_STATUS_SIZE), which caused a bug.
Also in the functions that calculate LDA and GGA part of the energy it was checked whether the density value in each point was greater or equal to zero. Which is wrong but was not observed since all the plane wave codes that were used for testing are not very likely to have exactly zero value at any point. Now it is corrected in appropriate files to check whether the density at a point is strictly larger than zero. That are hopefully all the modifications needed.

2

Introduction

The purpose of this manual is to explain what and how the Juelich Non Local (JuNoLo or vdW) program calculates. The physical background of the calculation is given in Refs. [1] and [2]. This manual and the program source code you can find here `http://www.fz-juelich.de/iff/src/th1/JuNoLo/`. The purpose of the program is to calculate non-local contribution to total energy based on a real space charge density provided from some DFT code. In this sense program is applied as a post-processing tool. About justification of such procedure and about self-consistent implementation of the theory consult Ref. [3]. Program is intended for (massive)parallel execution and owing to that it can accompany the largest DFT calculations available today. The code is not so much commented so some explanations of its inner working are given also in this manual.

3

Compilation

The program is written in Fortran 90 and needs mpi libraries to compile. Besides that only other library needed is fftw which is freely available at *www.fftw.org* webpage. Once the program is compiled it can be executed as a single processor program or as a mpi multiprocessor program. Regarding MPI on Linux machines (cluster) you can use either MPICH(2) or LAM-MPI implementation. Both have been tested in combination with Portland Group F90 compiler (pgf90). Special solutions of MPI on IBM machines such as Blue-Gene/L, Blue-Gene/P in combination with IBM F90 compiler were thoroughly tested.
Compilation itself should go smoothly. Several Makefiles are provided and user should select the one that matches his architecture most closely and modify it as necessary. With the proper Makefile

```
> make -f Makefile
```

should create *vdw.exe* (or similar name) program executable.

3.1 Running the program on Linux machines

To run the program on Linux machines (cluster) you need an input file. It is a simple ASCII file explained in one of the later sections. To run a single processor job you should issue the command

```
> ./vdw.exe input_file
```

For multiprocessor job first you have to have mpd or lamd running on selected machines and than give a command

```
> mpirun -np 8 ./vdw.exe input_file
```

to run, for example, on 8 CPUs. For a typicall run you will need in the running directory: input file, kernel file (precalculated or input parameters only) and charge density file specified in an input file. For more details see the rest of the manual.

3.2 Running the program on other machines

On other type of machines such as Blue-Gene or similar cluster machines with scheduling policy you should consult documentation of the particular machine. A simple job script (run_vdw_jugene) for Blue-Gene computer is provided.

4

How to calculate

The purpose of this program is to calculate a new total energy for your, already done, DFT calculations. You will need electronic charge density from the DFT calculations given on a real space mesh (equidistant). Also, you will need the value of a total energy from a DFT code i.e. the energy that includes exchange-correlation contribution but also all the other contributions that one has in a DFT calculation (Hartree energy, Coulomb energy etc.). Let us suppose your DFT calculation is done using a PBE GGA exchange-correlation functional and that the obtained total energy value is $E_{\rm tot}^{\rm DFT}$. From the provided charge density file the *JuNoLo* program will recalculate: $E_{\rm x}^{\rm LDA}, E_{\rm c}^{\rm LDA}, E_{\rm x}^{\rm PBE}, E_{\rm c}^{\rm PBE}, E_{\rm x}^{\rm revPBE}$ and $E_{\rm c}^{\rm NL}$ which are respectively exchange contribution in LDA, correlation contribution in LDA, exchange contribution in PBE, correlation contribution in PBE, exchange contribution in revPBE and non-local correlation contribution to the total energy. The last, non-local correlation contribution is the most important and the most expensive to calculate.

To get a new value of the total energy you should use the following formula:

$$E_{\rm tot}^{\rm NL} = E_{\rm tot}^{\rm DFT} - E_{\rm x}^{\rm PBE} - E_{\rm c}^{\rm PBE} + (E_{\rm x}^{\rm PBE} + E_{\rm c}^{\rm LDA} + E_{\rm c}^{\rm NL}). \tag{4.1}$$

The point is to take out the old DFT correlation contribution and to put in a new non-local one according to Ref. [1]. Also we write explicitly the contribution $-E_{\rm x}^{\rm PBE} + E_{\rm x}^{\rm PBE}$ to emphasize that one should work with GGA flavor that was used in the DFT calculation, but if one wants to play with functionals one can take out $E_{\rm x}^{\rm PBE}$ (for which the DFT calculation was selfconsistently made) and put in for example $E_{\rm x}^{\rm revPBE}$, or vice-versa. The non-local correlation energy is given by formula (1) in Ref. [1] and we give it here:

$$E_{\rm c}^{\rm NL} = \frac{1}{2} \int d^3r d^3r' n(\boldsymbol{r}) \phi(\boldsymbol{r}, \boldsymbol{r'}) n(\boldsymbol{r'}), \tag{4.2}$$

which in the case of discrete density distribution reduces to:

$$E_{\rm c}^{\rm NL} = \frac{1}{2} \sum_i \sum_j n(r_i) \phi(r_i, r_j) n(r_j). \tag{4.3}$$

This is the main calculation that the program actually does. The program package DACAPO writes out all energy contributions separately so we have mostly used it to test our code regarding calculation of PBE and LDA energies.

4.1 Core charge

In many DFT codes one has the nonlinear core correction in addition to the total charge density (valence one). However it turns out that the inclusion of such contribution is not important for the *JuNoLo* program to yield a correct result. (In the light of the newest calculations some systems might be sensitive to this issue — for example the ones that include platinum atoms.) We will not discuss here the reasons in detail. With the Dacapo code we have tested explicitly calculation with and without the core charge and the results were the same. In VASP code we have encountered some problems in obtaining the core charge and we suggest not to use it. (CHG file is just enough). Using the charge density without the core charge from VASP gives perfectly the same results for energy differences as the results obtained for charge density with and without core charge obtained from DACAPO code. In figure 4.1 we show the energy curves for different calculations of Kr dimer.

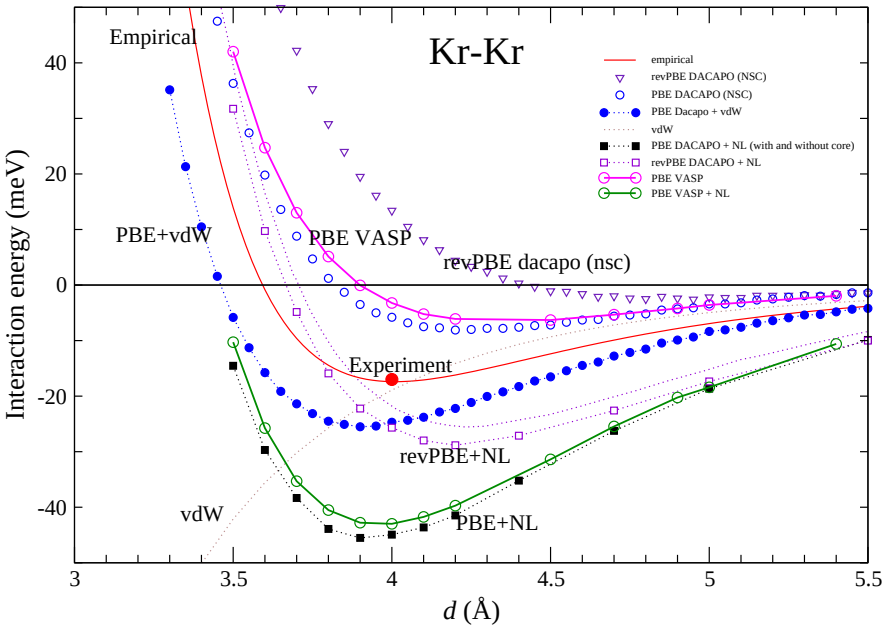

Figure 4.1: The binding energy for Kr dimer depending on the Kr atoms spacing. Pure DFT results are shown together with nonlocal energy results. The VASP curve shows result without the core charge while DACAPO curve shows both results - with and without the core charge. Small differences between dacapo PBE and VASP PBE DFT curve are due to a fact that VASP uses PBE functional self-consistently while dacapo uses PW91 functional selfconsistently and from charge density provided in this way we recalculate PBE energy. Also VASP is a PAW code while Dacapo is not, some difference could be due to that.

5

The input file

The listing of a typical input file looks like this (numeration of lines is added for clarity and is NOT A PART OF the input file):

```
1.  out_Xe_layer_3.5-dens3d              # name of the density file
2.  4                                        # periodicity
3.  64                                       # derivation order
4.  0                                        # treating derivations
5.  fft_derivs                             # derivations file
6.  1e-4                                     # trimming value
7.  0                                        # calculate kernel 1 or not 0
8.  kernel.txt                               # file with kernel parameters
9.  2                                        # per1
10.  2                                        # per2
11.  0                                        # per3
```

(The complete calculation examples with appropriate files are provided in a subdirectory EXAMPLES).

5.1 INPUT FILE: Line 1. - The charge density file

Line 1 contains the name of the charge density file. This file is the essence for your calculation and it should represent electronic charge density from your favorite DFT code. So far we have tested *vdW* program with charge densities obtained from codes such as VASP [5], Dacapo [6] and PWscf [7]. It is completely irrelevant what kind of code is used for DFT calculation (plane wave, some other basis set, real space, LCAO etc.) as long as it can provide you with a charge density on a

real space mesh (we assume equidistant mesh). The units used throughout the files are atomic units (a.u. for length, density etc.) except when the program writes out the energies in the log file - they are given in eV.
The beginning of a charge density file looks like this: (again numeration of lines is given for clarity only)

```
1.  48 48 240
2.  0.0 0.0 0.0
3.  7.01525312179 4.0502582783 0.0
4.  7.01525312179 -4.0502582783 0.0
5.  0.0 0.0 42.5503385415
6.  -1.40628535003e-07
7.  1.40099759641e-08
8.  -4.91261977308e-08
9.  4.81829504574e-08
```

•*Line 1* The first three integer numbers are the numbers of divisions in direction of each unit cell vector. Let us call them n_1, n_2 and n_3.
•*Line 2* The three float numbers represent the origin (so it should just be 0.0 0.0 0.0, we have no idea right now why did we introduce this in a first place.)
•*Line 3* Three float numbers are x,y and z components of the first unit cell vector $\mathbf{a}_1$ (and are of course given in a.u. of length).
•*Line 4* the same as line 3. but for $\mathbf{a}_2$.
•*Line 5* the same as line 3. but for $\mathbf{a}_3$.
What follows from line 6. to the end of this file are the values of the charge density written in order following from this construction of for loops:

```
for k=1,n3
for j=1,n2
for i=1,n1
density[i,j,k]
```

All together you should have $n_1 * n_2 * n_3$ charge density values.
Three python scripts are provided to obtain charge density in this format from the programs: Dacapo, VASP and PWscf. For VASP the script takes care if the calculation is spin polarized or not, for other two programs you will have to figure out how to proceed in a case of a spin polarized calculation. Moreover in the case of DACAPO the python script uses the old ASE modules so for the users of ASE2 modules it will be necessary to rewrite this script (and mail it to us if possible). Examples for all three program packages (Dacapo, VASP and PWscf) are included. Look into README files in appropriate subdirectory for

explanations. To check your density file visualization script is also included. To use it you will need a working installation of the VTK package with the python bindings. (www.vtk.org). Typical density visualization is given in figure 5.1.

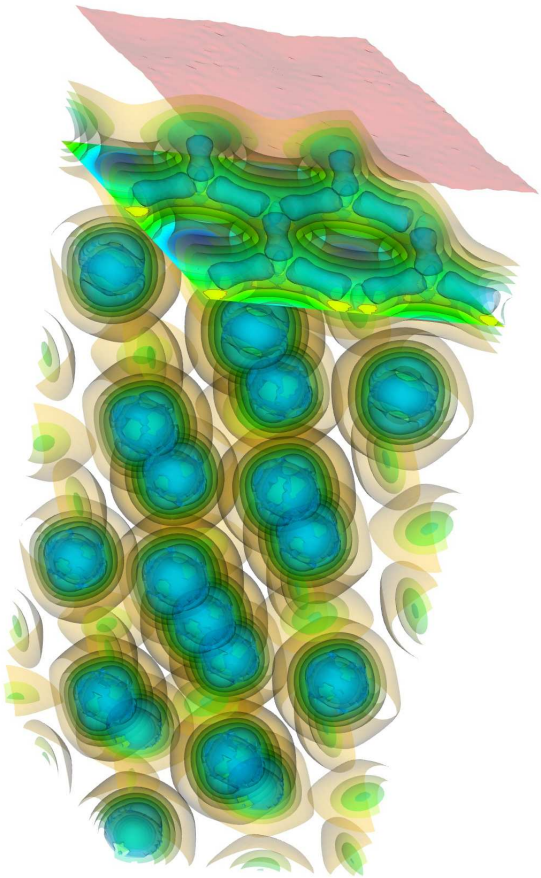

Figure 5.1: Example of density visualization with VTK python program. Graphene sheet on Ir(111) surface is shown.

5.2 INPUT FILE: Lines 2. and 3. - real space derivative

Pay attention that it is not necessary that if you have done your DFT calculation in plane waves approach that you should necessarily use only fft approach to calculate derivatives or vice versa. But to get perfect reproduction of the semi-local energy values it is suggested to use fft approach for charge densities obtained from the

plane wave code and real space approach if you have used real space code. If you have used some other implementation you should decide what suits you the best. So, if you use plane wave DFT code (probably 98% of you) just skip this subsection if you are not interested in testing real space derivatives. These two parameters have to do with the definition of a derivation (gradient) in a real space by means of finite differences. The code has the ability to calculate charge density gradient using the Fast Fourier Transform (FFT) or a finite differences formula (the later is only implemented for orthogonal unit cell vectors!!). More on this is explained along with the explanation of line 4. which contains the switch which tells how the gradient is calculated. To understand parameter *periodicity* (line 2.) first we have to explain parameter *derivation order* of line 3. In this case it is set up to a rather large number of 64. Implemented numbers that one can use here are 2, 4, 8, 16 and 64. This number says how many neighboring points are taken into account when the gradient in a certain point is calculated. For example if this number is set to 8 then derivation in direction of a unit vector $\mathbf{a}_1$ for example will take into account the points marked in figure 5.2.

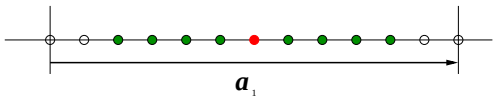

Figure 5.2: For calculation of the value of the function 1st derivative in red point by finite differences approach we need function values from the green points. This case corresponds to setting parameter *derivationorder* to 8. Notice that all the points are within the unit cell, in this case it is 1D but analogously one can extend this observation to 3 dimensions.

Analogously the neighboring points in directions of other unit cell vectors are taken into account. Once again this is implemented only for orthogonal unit cell vectors.
The parameter *periodicity* takes care about boundary conditions in the case that we are calculating the gradient by means of finite differences. Let us say that we want to calculate the gradient value in a red point marked in figure 5.3

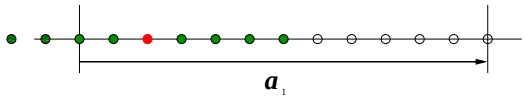

Figure 5.3: The same as figure 5.2 only that this time two of the needed points for derivation calculation lie outside the unit cell and a priori it is not known which values we should suppose in those points. That is determined by the parameter *periodicity*.

If the parameter *derivation order* is set to 8 we will need density values from the points marked in figure 5.3. However the two points furthest to the left are

outside the unit cell. If the *periodicity* parameter is set to 0 it means that all the points outside the unit cell are considered to have zero charge density. If we set parameter *periodicity*, for example, to 1 that means that in the first neighboring cells the density from the 0^{th} cell will be repeated. If large number of neighbors is defined for finite differences calculation it can happen that required neighboring points will lie in a second neighbor cell or even more distant ones (very improbable but in principle possible). In which cells the density from the 0^{th} unit cell is periodically repeated depending on the *periodicity* parameter is best explained in figure 5.4.

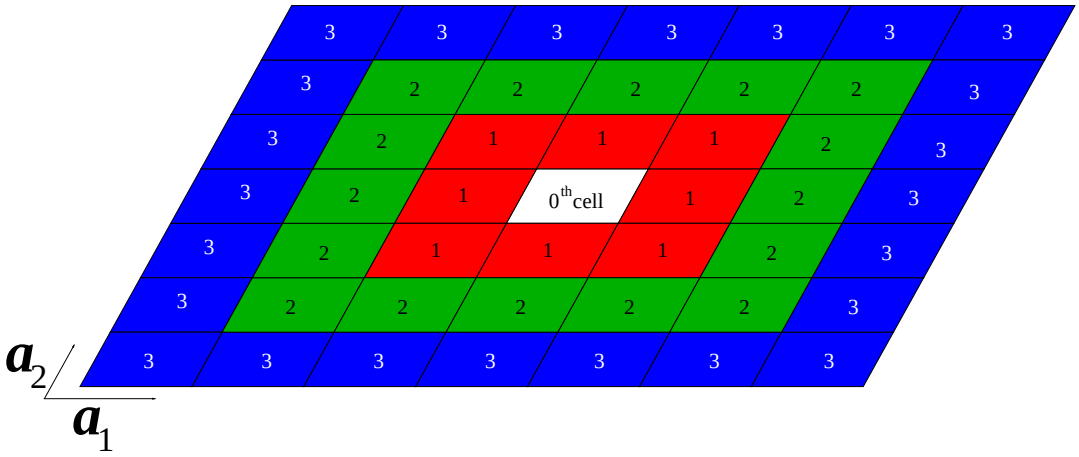

Figure 5.4: The figure shows how the 0th cell density is repeated in space according to the value of the *periodicity parameter*. It is shown only in 2D but extension to third dimension is straightforward. This image is a bit misleading because *periodicity parameter* and this repetition is only important if one calculates gradient by means of finite differences which are at present implemented only for orthogonal unit cells. Maybe in future it will be implemented for skew unit cell, than is this picture perfectly OK.

The possibility to change periodicity option allows the code to perfectly reproduce results from a real space DFT codes in which one can really have zero value boundary conditions outside the cell (which is not possible in the plane wave codes in which the periodic boundary conditions regarding the charge density are always periodic). Of course one can set periodic boundary conditions by means of the *periodicity* parameter for the finite differences calculation of a gradient and compare the result with the one obtained by FFT.

5.3 INPUT FILE: Lines 4. and 5. - choice of derivative calculation and fft output file

Parameter in line 4. says how the charge density gradient will be calculated, available values are:
0 - use the FFT
2 - use the finite differences real space approach for orthogonal unit cell only
3 - read gradient from file which name is given in the next line (line 5.)
First of all let us say (for the third time now) that the calculation of gradient by means of FFT works for all cases (orthogonal and non-orthogonal unit cell vectors) while finite differences approach works only for orthogonal unit cell. With FFT approach the density is intrinsically periodic while with finite differences one can play with the option *periodicity*.

0 - gradient by Fast Fourier Transform

This procedure takes into account all the density points and does a calculation of a gradient value by the means of FFT and in particular using the FFTW implementation for it . How it is done you can have a look into this code or in any plane wave DFT code of which you have the source. The small drawback of this choice for gradient calculation is that it is done exclusively on a master processor which is not a problem regarding an execution time (which is of order of seconds even for the largest calculations) but it can be a problem regarding memory. On a large supercomputer such as a Blue-Gene/L memory per core is limited to a small 256 Mb (in coprocessor mode to 512 Mb) which might not be enough in many cases. For this reason the code writes the calculated gradient values into a file whose name is given in line 5. The idea is to prepare this file on a local workstation with large amount of memory (2Gb or more) and to use it later on some other machine/cluster. Only the calculation of gradient by the FFT will result in writing into a file, using the finite differences approach does not write gradient data into a file.

3 - gradient - read from file

This option is self evident, once you have obtained gradient data by means of FFT you can restart your calculation with this option instead of doing a FFT again. The reason and purpose of this procedure is explained in the previous subsection on FFT usage.

2 - gradient - finite differences, beginning of parallelization

Again if you use plane wave DFT codes you are probably not interested in this subsection. If this option is chosen to calculate gradient then parallelization procedure begins already here. Master processor reads the density points and distributes them as evenly as possible to all CPUs including himself. Each CPU now has 2 sets of points one is called *my_points* and the other is called *received_points*. Set of *my_points* is considered as the set assigned to a CPU which has a task to calculate gradient values for this set of points. To calculate gradient value of the set *my_points* we proceed in the following manner: First step is to copy set *my_points*

to a set *received_points* and to start calculating the gradient for *my_points*. Each point has its indices i, j and k in a 3d grid. Once the *received_points* are loaded the new 3D array *indices_3d* (of the size n_1*n_2*n_3) is established in which is written -1 if point with given indices i,j and k is not contained in the set *received_points* or the index of this point in the set *received_points* otherwise. In this way it is possible to do a minimal check for required neighboring points (obeying also the *periodicity* parameter) while calculating gradient for the points in set *my_points*. After this initial step the CPUs start to send their points (*my_points*) to other CPUs which accept them in *received_points* array. How it works is shown in figure 5.5 for the example of 5 CPUs.

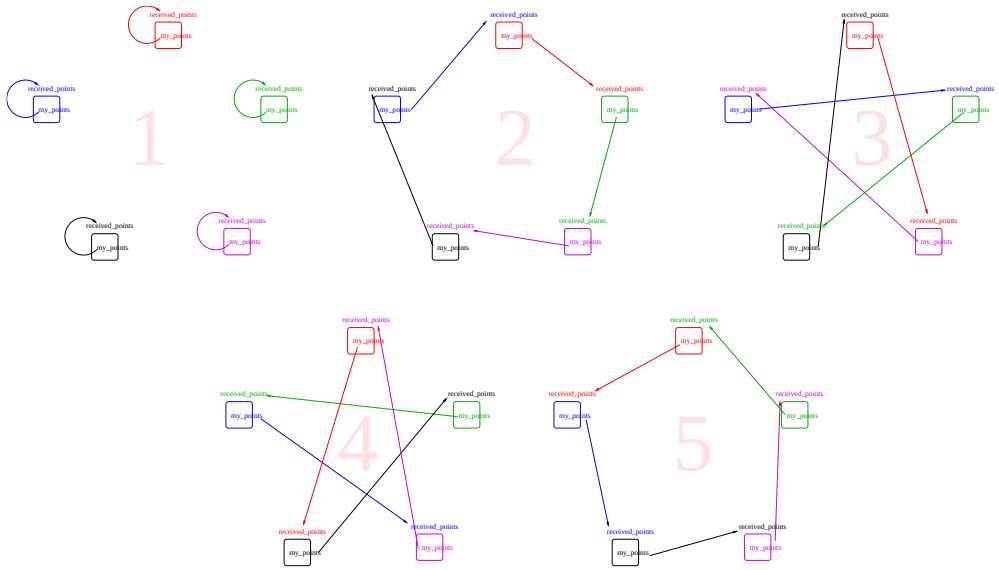

Figure 5.5: The ring. If calculation of the gradient by means of finite differences is selected this diagram shows how it is done. Each CPU has the task to determine gradient in set of points attributed to it. That set is called "my_points". In order to do that it needs to take into account values in all other points in the system (including its own of course - that is done in step 1) which are available only to other CPUs. By going through steps 2, 3, 4 and 5 each CPU receives points from other CPU and uses them do calculate gradient value contribution on its set of points. After step 5 all CPUs have calculated gradient in their respective set of points.

Notice that this procedure is not highly efficient because the spatial structure of the density is not mapped onto CPU distribution in any particular way. This makes the code more flexible but does not yield maximum efficiency for this type of calculation. (for more information see how this is done in efficient but inflexible way in Ref. [4]). After the value of the gradient is calculated in this way it is not written into a file which is not the case with FFT approach. This procedure takes

significantly more time than FFT approach but does not require much memory per CPU. The more CPUs you use in calculation linearly less memory per CPU will be required for this procedure.

5.4 INPUT FILE: Line 6. trimming value - beginning of a non-local calculation

Points of low density or even zero or negative density are insignificant for the calculation of the non-local energy contribution. Taking them into account would only increase computational time a lot without impact on the total energy value, especially in open systems containing surfaces or molecules. Therefore points at which the density is lower that a trimming value are not taken into calculation. The typical value here should be 1E-4 (see Ref. [2]).

5.5 INPUT FILE: Lines 9., 10. and 11. periodic repetition in space

We leave deliberately the lines 7. and 8. for the end of the chapter where we explain in detail what is kernel and how it is used.
The three parameters given in lines 9., 10. and 11. are called *per1*, *per2* and *per3* and they determine how many times the 0^{th} cell is being repeated in the directions of the unit cell vectors (in both directions + and -) for the needs of non-local energy calculation. If we have the calculation of intrinsically nonperiodic object such as atom in a box we do not want to repeat the density through the space. For such calculation one should use 0, 0, 0 values for those parameters. If we have, on the other hand, a unit cell that represents a bulk calculation one should use value larger than 0 for each of these parameters. The larger the value is the closer the result will resemble bulk calculation. For the graphical explanation of those parameters let us take an example of calculation of adsorption of a molecule on a surface laying in the plane spanned by the unit cell vectors a_1 and a_2. To obtain a good result for total energy differences one should repeat the unit cell in directions of a_1 and a_2 until the energy differences (for example between on-top and bridge site adsorption) have converged to desired accuracy (not the total energy of system! because it will keep changing even when the energy differences have converged). How does the repetition of the 0^{th} cell density look like depending on parameters *per1*, *per2* and *per3* is shown in figure 5.6. A word of caution is needed here. Be careful how you do your DFT calculations because for NLC calculation real space position of density matters. If you want, for example to compare CO adsorption energies for the on-top and FCC position on ithe (111) surface and you position CO molecule in a way that is shown in figure 5.7 you will obtain spurious energy differences if you do not use repetition of the 0th cell. Energy calculated for such

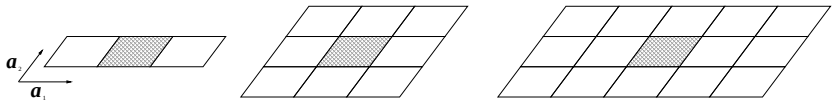

Figure 5.6: This image shows how the 0th unit cell (shaded) is repeated in space according to value of parameters $per1, per2, per3$. Number beneath each image shows concatenated string produced from mentioned parameters i.e. 210 means $per1 = 2, per2 = 1$ and $per3 = 0$.

density without repeating the 0^{th} cell will not be a good one because the points that are actually close to each other in space are considered, without a repetition, to be far away which gives a bad estimate for the total energy value. This is of course cured by repeating the cell but that is numerically rather expensive, it is much cheaper to always try to put the density of interest (molecule, atom etc.) in the middle of the unit cell. This problem typically occurs if you do an isolated atom calculation and you position it at a corner of a unit cell (which is perfectly OK in a plane wave DFT code) but here you have to think in terms of a real space density or otherwise to obtain a good result you will have to do the repetition of the 0^{th} cell (which is expensive) or density shifts in space which is sometimes cumbersome.

5.6 INPUT FILE: Line 7. and 8. - The kernel!

Line 7 - this is the switch that tells the code either to calculate the kernel `1` or to read it from the file `0`whose name is given in line 8. If this option is set to calculate the kernel the program will search for calculation parameters within the file whose name is given in line 8. This file looks like this:

```
1.  200.000000000000000                                          # amax
2.  2000                                                         # na
3.  200.000000000000000                                          # bmax
4.  2000                                                         # nb
5.  20.0000000000000000                                          # Dmax
6.  2000                                                         # nD
7.  2000                                                         # ndelta
8.  2696.60361799144721 0.00000000000000000E+00 0.00000000000000000E+00
9.  1.51980515498387669 0.10005002501250625E-01 0.00000000000000000E+00
10.  1.50409663617428646 0.20010005002501250E-01 0.00000000000000000E+00
11.  1.50236409308319518 0.30015007503751876E-01 0.00000000000000000E+00
```

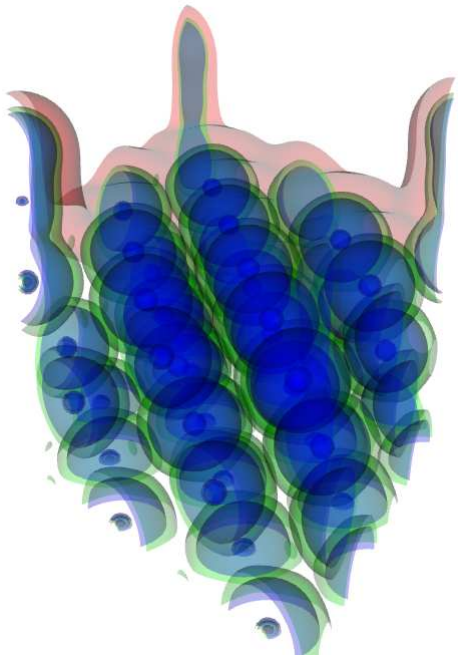

Figure 5.7: Image shows charge density for calculation of CO adsorption on Pt(111) surface. This is not the best choice for the position of CO molecule for such a large unit cell (low coverage). Much better is to try to put CO as much as possible in the middle of the cell to save some numerical work if possible, for details see text.

```
12.  1.50161575289707794 0.400200100050025001E-01 0.000000000000000000E+00
```

Once again the lines are numerated only for clarity. The lines from 8. till the end of the file are actually the values of the calculated kernel and will appear only after the program has calculated the kernel. So the lines considered as the parameters for the kernel calculation are the lines from 1. to 7.
What is the kernel $\phi(\delta, D)$ and what are the parameters $0 \leq D < \infty$ and $0 \leq |\delta| < 1$ you must search in Ref. [1]. From this point on in this subsection we will refer strongly to formulas in Ref. [1]. The parameters $amax$ and $bmax$ represent the upper integration limit that should correspond to infinity according to the formula (14) in Ref. [1]. na and nb represent the number of points for the nummerical integral calculation.
$Dmax$ is the value of D up to which the kernel is being calculated, above this value asymptotic expression (17) from Ref. [1] is being used. nD and $ndelta$ are the numbers of points in which the kernel values are calculated. The kernel is calculated in parallel and each processor gets certain amount of values of D for which it calculates kernel values for all the δ values. Due to this fact the number of D points i.e. nD must exceed (or match) the number of CPUs being used for

the calculation. If the kernel is already calculated and is being loaded from the file than the relation between the number of CPUs and nD is irrelevant. What one can notice is that the kernel changes more abruptly with the change of the value of D so one can play around with different kernel parameters. The kernel that is provided within the program package is a total overkill containing 2000 D and 2000 δ points (Large Kernel on the web site). Much smaller kernel can be used and we strongly encourage the user to find his own small size kernel because during the non-local calculation each CPU has to store kernel in the memory so it becomes a central memory consumption issue in the program if the kernel is large. When the kernel is calculated it is written out in the same file which defined parameters and the three values are written in each line staring from line 8. These values are respectively $\phi(D, \delta)$, D and δ. Notice the huge kernel value in the vicinity of $D = 0$, it is a nummerical issue and we call it $\phi(0)$ issue and it is discussed in detail in a separate section. Once you have a good kernel you can use it throughout all kind of calculations - it is not calculation sensitive (unlike $\phi(0)$).

Also it is worth mentioning here that kernel is loaded by the master and distributed to other CPU-s. Whenever such case occurs that master has data that has to be sent to all CPUs we employ self implemented binary tree communication which is schematically shown in figure 5.8.

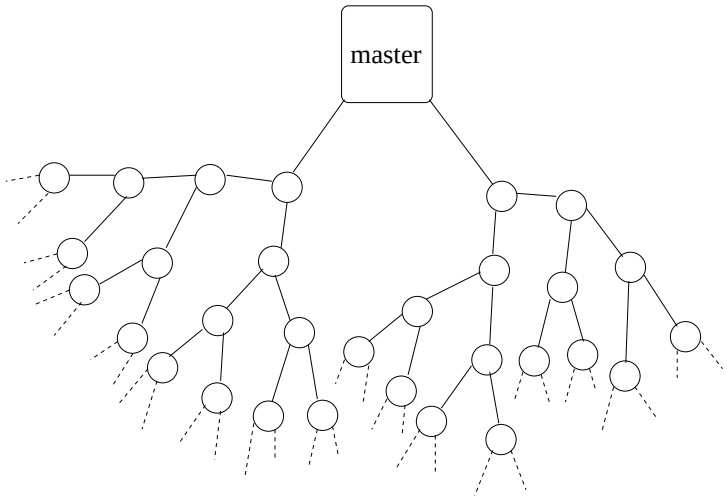

Figure 5.8: Schematical presentation of binary tree communication starting from the master. This type of communication enables transfer of large data quantity in a very short time.

6

The $\phi(0)$ issue

What we call the $\phi(0)$ issue can be seen in a non-local calculation when one tries to compare systems with different unit cells. This typically occurs when we want to calculate the optimal lattice constant for some system - let us say for a self standing Xe monolayer as a good example. As we change the lattice constant to obtain energy minimum by using the plane wave DFT code (VASP/Dacapo) we get a graph shown in figure 6.1.

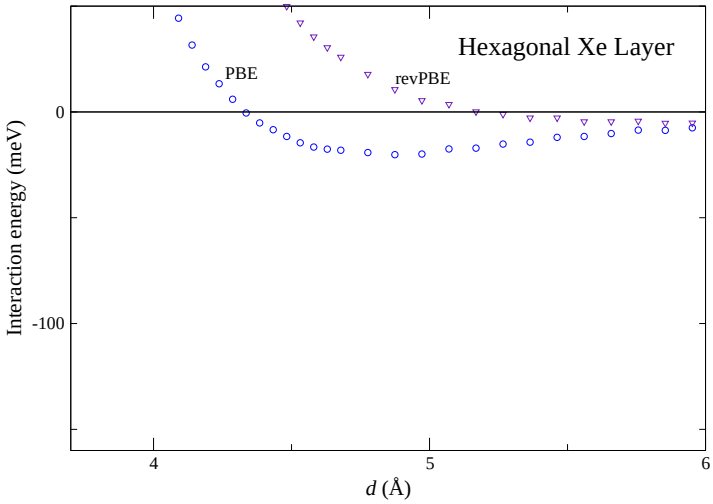

Figure 6.1: Binding energy for self standing Xenon monolayer in vacuum as a function of lattice constant - calculated by DFT. DFT was implemented in a plane wave code (Dacapo) and the energy cutoff used was 400 eV.

If we want to calculate now a new total energy with a non-local contribution and if we consider the value of $\phi(0)$ to be constant (see previous chapter on $\phi(0)$) we will obtain the graph shown in figure 6.2 which is obviously useless. The main reason for large jumps in energy is the so called $\phi(0)$ issue which is caused by a

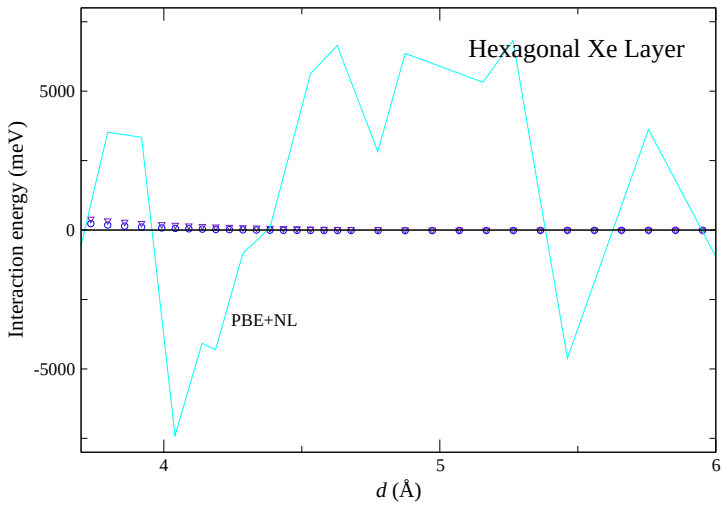

Figure 6.2: Xenon monolayer binding energy with NL theory implemented with constant $\phi(0)$ value i.e. with nonresolved $\phi(0)$ issue.

change of real space grid due to change in unit cell vectors. All DFT calculations were done with the same cutoff energy of 400 eV but the real space grid changes as we change the lattice constant, namely the numbers $n1$, $n2$ and $n3$ change with the lattice constant value. The reason for large jumps in energy values is that a charge density is differently "packed" in points as the grid parameters $n1$, $n2$ and $n3$ change. Large jumps in energy differences value are mostly caused by a diagonal non-local energy contribution i.e. if one writes the integral (4.2) as a discrete sum (4.3) the diagonal contributions are considered the parts of a sum for which $i = j$ and these contributions necessarily contain $\phi(0)$ value. This is obviously a numerical issue and up to now we did not find a proper solution for this problem but a rather good workaround. To improve on energy jumps we exploit the analytical property of the integral:

$$\int_{\delta=0} \phi(D, \delta) dD = 0. \tag{6.1}$$

This integral can be represented in a real space using the relation between D and $\boldsymbol{r}$ and it can be matched onto our real space grid obtained from the DFT calculation in which case this integral reduces to sum. We calculate this sum over large number of points (excluding the "zeroth point"!) in real space but the points are distributed on a grid that corresponds to selected DFT calculation. We take so many points in space that we can say that the value of the integral has converged. The value of the $\phi(0)$ is then considered to be negative of this integral value (so that the total sum - including now the zeroth point value, would be zero). In this way the particular geometry of the charge density grid is taken into account by using the analytical property of integral (6.1). How far in space this

summation over points goes is hardcoded in file *param.f90* by parameters *px*, *py* and *pz* which have the same meaning as parameters explained in figure 5.6 only here they are important for calculation of $\phi(0)$. Which values of *px*, *py* and *pz* are needed here can depend on the size of the unit cell but we do not explore this in detail here. The issue of $\phi(0)$ could occur also for a fixed unit cell size if one tries to calculate adsorption energies due to the fact that separated parts of the system and system itself yield different charge densities in space. In particular if one calculates adsorbate alone in the totally different position of the unit cell or in a different unit cell all together (this is sometimes necessary to take into account dipole effects for example in VASP - one has to use cubic cell). We have done such kind of calculations only after we have implemented $\phi(0)$ issue solution and we obtained very good results. $\phi(0)$ is calculated for values of q from *qmin* to *qmax* and in between it is being interpolated. (for meaning of the parameter q look into Ref. [1]). In how many points the $\phi(0)$ has been calculated is determined by parameter *nphi*0 which is also hardcoded in the file *param.f90*. It is calculated in parallel and again *nphi*0 has to be larger or equal than a number of CPUs used in calculation. This is again important only when $\phi(0)$ is actually being calculated, later it can be loaded from file (be careful to use appropriate $\phi(0)$ for the given unit cell) but enabling this loading from a file option and suppressing the $\phi(0)$ calculation requires a bit of hacking - namely you have to comment a line in file *calculate_phi*0.*f90*. If you comment line *CALL calculate_phi*0*_now* in this file (almost at the end of the file) the program will just load the values of $\phi(0)$ from the *phi*0.*dat* file which of course is calculated before. By employing this improvement for diagonal contribution we obtain a bit improved graph for Xe monolayer shown in figure 6.3.

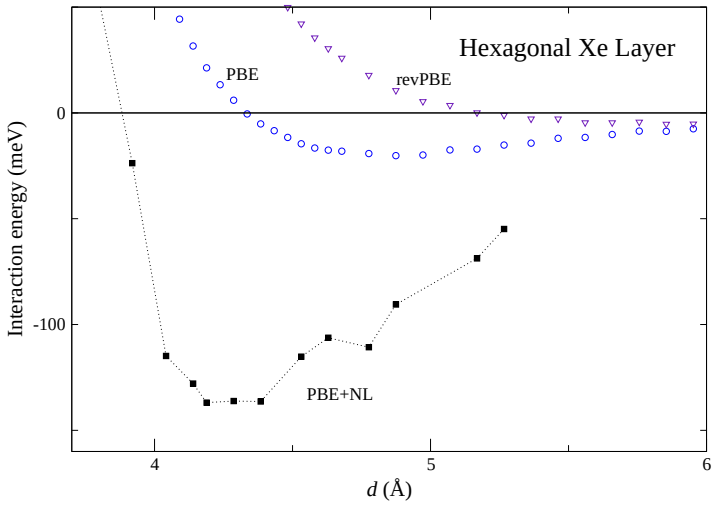

Figure 6.3: The same calculation as in figure 6.2 with implementation of $\phi(0)$ issue solution based on a sum rule given in the equation (6.1).

The improvement is achieved compared to figure 6.2 but still the non-local total energy curve is not smooth enough. We expect that it should be possible to obtain smooth curve here because the pure-DFT curves are smooth. However, the origin of this energy jumps is of the same kind as the $\phi(0)$ issue only now it comes from the first neighbors interaction which suffers from the same numerical deficiency as the diagonal contribution. For now we do not know the proper way to fix it. Nevertheless, as promised, there is a workaround and it consists in repeating DFT calculations with the higher cutoff energy (in plane wave codes) which results in denser real space grids for charge density outputs. Taking this charge densities and using our $\phi(0)$ procedure we obtain the final graph for Xe monolayer shown in figure 6.4 which is finally smooth.

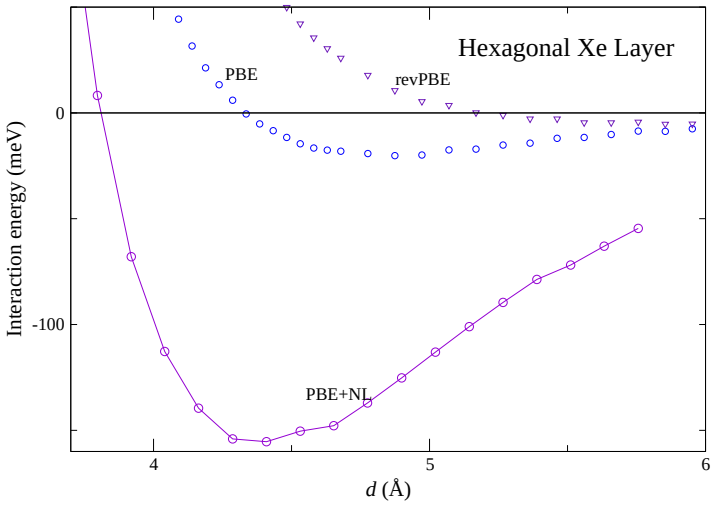

Figure 6.4: Finally the non-local calculation for Xe monolayer with $\phi(0)$ issue remedied by application of the exact sum rule and by increase of the grid density by increasing the plane wave cutoff in DFT program from 400 eV to 800 eV.

You can also notice that the binding energy is somewhat larger for the new calculations with denser grids.

7

The program Flow Chart and description

The flow chart of the JuNoLo code looks as follows:

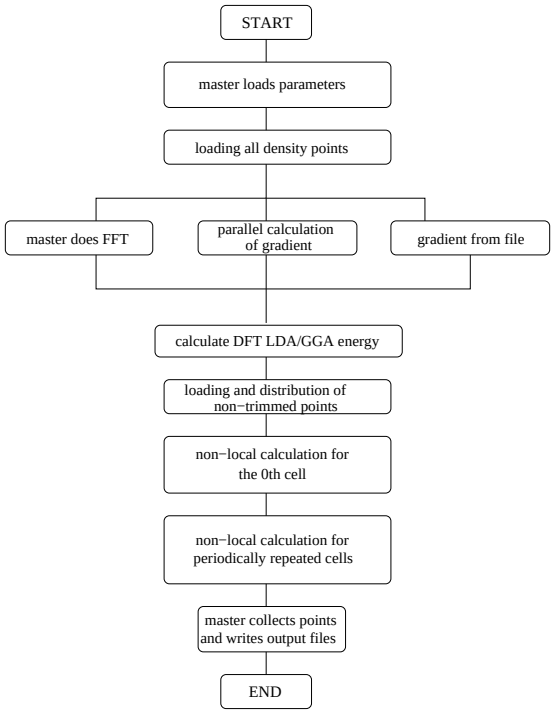

Figure 7.1: Flow chart of the JuNoLo program.

Of all the steps in the flowchart we describe only a non-local calculation for the 0^{th} cell and a non-local calculation for the periodically repeated cells.

7.1 Calculation of NLC contribution within the 0^{th} cell

Master loads non-trimmed points (which are written in file *calc_output_tmp.dat* with all the needed values such as gradient etc. that were calculated in a previous step) and distributes them evenly over all CPUs including himself. Master takes the points as they come in *calc_output_tmp.dat* file. The cost of calculation of nlc energy contribution between two points does not depend on a distance between them, therefore there is no need for shuffling the points in order to achieve good synchronization between CPUs (OK if this is not clear, never mind it is not important for understanding the code, it is actually an issue from some other physical problem and its parallel implementation but would also apply here if the cost of calculation would depend in any way on relative position of two points). Looking carefully at the integral (4.2) and its discretized version (4.3) and knowing that the interaction between two points is symmetrical one can rewrite the double sum immediately as:

$$E_{\mathrm{c}}^{\mathrm{NL}} = \frac{1}{2}\sum_i \sum_j n(r_i)\phi(r_i, r_j)n(r_j) = \sum_i \sum_{j\geq i} n(r_i)\phi(r_i, r_j)n(r_j). \qquad (7.1)$$

The transformed double sum is much cheaper (50%) to calculate than the original one. In order to perform such summation efficiently in parallel we proceed as follows: First, let us introduce graphical representation of that double sum as is shown in figure 7.2. On the horizontal and vertical line we mark the CPUs and

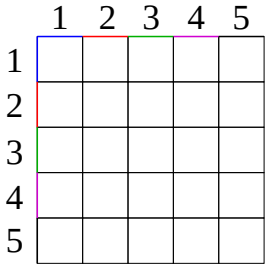

Figure 7.2: Graphical representation of the double sum (7.1). On each line (top and left) different line segment is colored in different color and represents set of points assigned to the corresponding CPU (1-5). For further explanation see figure 7.3.

those part of the line - for example 1 represents the points contained in CPU1, those points are considered his points (my_points) and this CPU is responsible for calculation of all the values for those points. If we want to show that we have calculated interaction between all the points of CPU1 and CPU2 we show it schematically by shading one of the two squares shown in figure 7.3. Using this

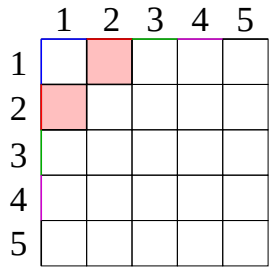

Figure 7.3: Image shows how do we schematically show that non-local sum was calculated between all the points from CPU 1 and CPU 2.

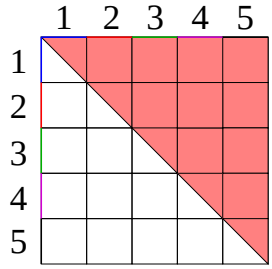

Figure 7.4: Following explanations from figure 7.3 this image shows how the schematical representation of the full sum (7.1) should look like.

scheme at the end of procedure we should have situation shown in figure 7.4. To make the optimal parallel algorithm for achieving figure 7.4 we have two different procedures depending on the parity of the number of CPUs used. In general for odd number of processors the number of steps required to calculate non-local energy for 0^{th} cell will be $(n_{CPUs} - 1)/2 + 1$ while in the case of even number of CPUs we will need $n_{CPUs}/2 + 1$ step.

7.2 Odd number of processors

By using the notation explained in the previous subsection we can easily explain the calculation procedure. Again each processor has two arrays of points, one being called *my_points* and the other *recived_points* as was already used at explaining gradient calculation in the real space. The first step is to calculate interaction among his own points which is really done by a double sum $\sum_i \sum_{j \geq i}$. How does it work is explained by figure 7.5 in which example calculation with 3 CPUs is shown. One can deduce how the procedure works in a case of a larger number of CPUs as long as their number is odd.

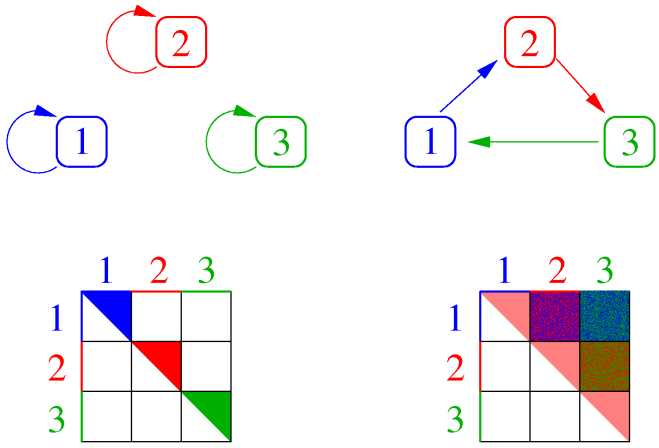

Figure 7.5: Image shows how the sum (4.3) is calculated in the case of 3 CPUs used in calculation. In the first step each CPU calculates the part of the sum using his points only while in the second step each CPU calculates interaction (sum parts) between his points and the points received from the other CPU. Already at the second step we see that the final state shown in figure 7.4 is achieved so the calculation of the sum (4.3) in the 0th cell is finished.

7.3 Even number of processors

Let us suppose that we have 4 CPUs. Following the procedure for the odd number of CPUs we end up in a situation described by the second image on figure 7.6. We are left with two squares empty. To keep the symmetry in communication once again processors send all of their points around but during the calculation processor 1 for example calculates only the interaction of the first half of his points with all the points he received from processor 3 while processor 3 calculates interaction of all of his points with the other half of points received from processor 1 which is schematically shown on the last image in figure 7.6. In this way each processor has the same amount of work to do so that synchronization is nice.

7.4 Important notice for the 0^{th} cell NLC calculation (regardless of parity of the number of CPUs)

One important thing has to be noticed here. Once the processor has received points from other processor (or from himself) and is going to calculate nonlocal energy contribution everything is pretty much clear but we want also to keep track of this energy contribution per point. In order to do this one should consider that interaction between two different points, a and b for example, is E_{ab}, but one half

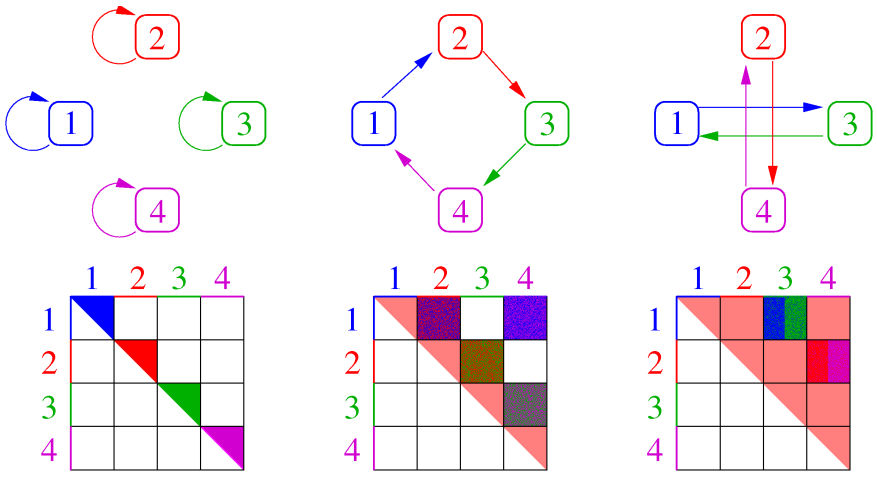

Figure 7.6: Image is similar to the figure 7.5 with the difference that we have even number of CPUs here. That causes difference in the last step which is thoroughly explained in the text.

of this contribution should be contributed to point a and the other half to point b. This is kept in mind so after the calculation processor that did the calculation returns this information to processor from which it had just received the points and did the calculation with. This is of utmost importance later if one wants to visualize non-local energy contribution in space.

8

The program output

Once you have successfully compiled and started the program it will start writing several files depending on the number of CPUs it's running on. Some of the output will be sent directly to a screen device. The most important file will be the log file for which the name will be generated in this manner:
$log\_file\_name = "Log\_" + name\_of\_the\_density\_file\_used + "\_n1n2n3.dat"$
where $n1, n2$ and $n3$ are the parameters for periodicity given in the lines 9., 10. and 11. of the input file. Listing of this Log file with explanations is given in Appendix A.
Each CPU will write its own file named $file\_x.txt$ where x is the rank of the CPU according to MPI. The master CPU will always write into $file\_0.txt$ file. Other files written are file fft_derivs (or with a different name that you have defined in the input file). This file will be written only if you use FFT to calculate the gradient. This file has a header line which explains what is written in it and it says: $x\ y\ z\ dx\ dy\ dz\ grad$.
x, y and z are the real space coordinates of a point while dx, dy and dz are gradient components. The *gradient* is the absolute value of the density gradient at this point.
The file *calc_output.txt* is written at the very end of a calculation and it also contains the header line as the first line. Before explaining the header line of *calc_output.txt* let us say that the file *calc_output_tmp.txt* also exist. It has the same meaning as *calc_output.txt* with the major difference that it is written before the nonlocal values of energy are calculated. Both of these files contain only the data for nontrimmed points. The header lines in file *calc_output.txt* are as follows:

```
nx,ny,nz        120        120        224
x0,y0,z0         0.00000000         0.00000000         0.00000000
ax,ay,az         16.80907372         0.00000000         0.00000000
bx,by,bz         -8.40453686         14.55707807         0.00000000
cx,cy,cz         0.00000000         0.00000000         32.49846143
name of the log file LogFile_CO_clean_revPBE.dat_000.dat
```

```
i j k index x y z n dnx dny dnz dn d2n ldac ldax kf q0 pbec nlc_diag pbex nlc revpbex
```

All the values given bellow the header line are the values at this particular point in space.
• - Indices i, j and k represent indices of the point in a 3d grid, by having these indices one can reconstruct the full 3d grid structure for visualization.
• *index* - total index of a point as it was used in calculation, counting also the nontrimmed points.
• x, y, z - real space coordinates of a point.
• n - charge density.
• dnx, dny, dnz - components of gradient (along unit cell directions?).
• dn - absolute value of density gradient.
• $d2n$ - Laplacian value.
• *ldac* - LDA correlation.
• *ldax* - LDA exchange energy.
• kf - Fermi wavevector value.
• $q0$ - parameter q from Ref. [1].
• *pbec* - PBE correlation energy.
• *nlc_diag* - diagonal contribution to non-local energy.
• *pbex* - PBE exchange energy.
• *nlc* - total nonlocal energy contribution (i.e. diagonal + offdiagonal part).
• *revpbex* - revPBE exchange energy.
All the values given above are in a.u. Most interesting for you is probably energy which is in Hartree (this is not energy density in a real sense of the word but rather energy contribution at a given point, to get energy density value you have to divide it by the volume of the volume element of the grid.) However - do not expect that summing for example LDA values from this file will yield exactly the same number that program writes in Log file or on screen. As mentioned above in this file we write only nontrimmed points while in fortran local calculations (i.e. calculation of LDA and PBE energies) we use all the points from the unit cell. Hence the difference. The number obtained if fortran local calculation (i.e. with all the points) should agree with the number from a DFT code from which you obtained the charge density.

9

Parallelization - Speedup

As it is visible from the description of the program it is almost completely parallelized. How does it behave with the number of CPUs we have tested on two examples. The first example is a Kr dimer problem and the speedup for it is shown for the number of CPUs going up to 256.

The curve starts do deviate from the ideal speedup curve as the number of CPUs get larger. This is due to a fact that this problem is rather small - after trimming it contains mere 69977 points so with the increase of the numbers of CPUs the number of points per CPU becomes smaller and the calculation time starts to be comparable to communication time. In this example with 256 CPUs in usage the total calculation takes 24 seconds of which almost 3 are spent in communication. We made a test with a larger problem and we obtained the result shown in figure 9.2.

The conclusion is that the program parallelizes nicely and makes calculation feasible for the largest systems available from the present DFT codes. It has been tested on 65536 CPUs of a Blue-Gene/P supercomputer and runs smoothly.

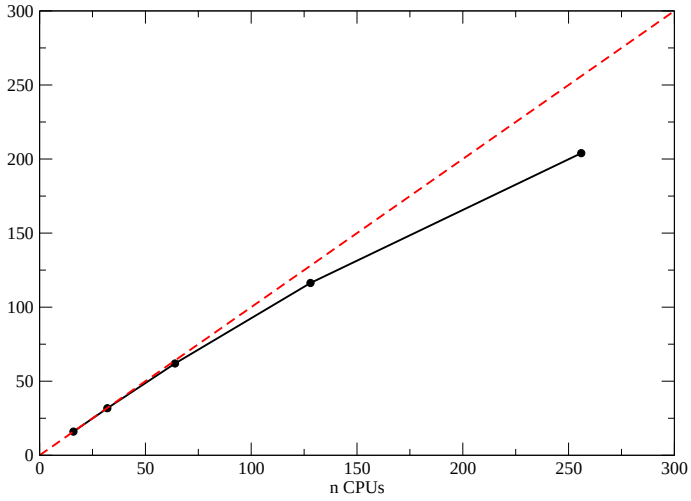

Figure 9.1: Speedup measurement done on a relatively small problem (69977 point after trimming). One can see that deviation from the ideal speedup curve appears as the number of CPUs gets larger. This happens due to a fact that number of points per CPU gets too low and too much communication sets in compared to calculation. On 256 CPUs the calculation is done in 24 seconds. However on larger problems scaling follows ideal curve to much larger number of CPUs - see figure 9.2.

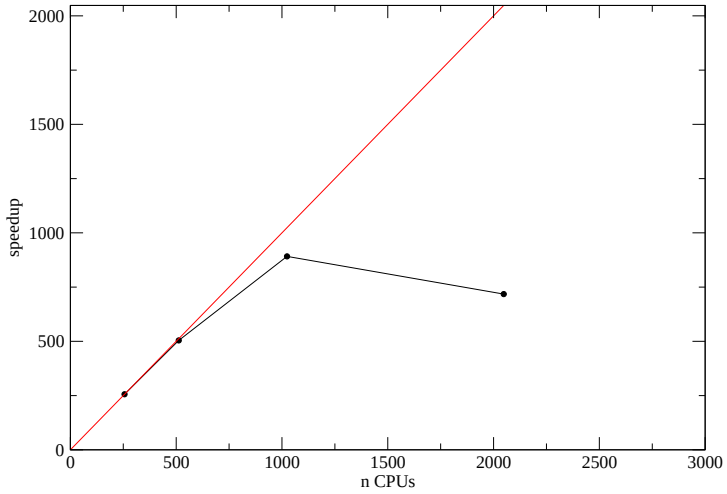

Figure 9.2: Similar effect can be seen as on image 9.1. Here the calculated problem was somewhat bigger around million points. One can see that when deviation from the ideal speedup curve occurs here it is much stronger due to a fact that we deal with much larger number of CPUs so that doubling of number of CPUs means introduction of much more communication while the number of points per CPU drops. The rule of thumb that we develop for our Blue-Gene/P machine is that optimal value of points per CPU is 1500. In both images, this one and figure 9.1 at the penultimate point of the measured speedup curve the number of points per CPU is around 1300. We conclude that when we achieve 1500 points per CPU it makes no sense to add more CPUs to calculation. Blue-Gene/P has relatively slow CPUs (850MHz) so at your local cluster this number could be much higher. For a really large problems with >10 Million points (after trimming) that we have also calculated we can use the whole Blue-Gene/P machine and all of its 65536 CPUs. Do not get a wrong impression, for a good usage of the code a modest cluster of 10 modern CPUs will make it possible to tackle very interesting and relatively big problems but maybe you will have to wait some 10 hours or so. For us the scaling was necessity because of the time limit of 6 hours on our Jugene supercomputer.

10

Conclusion - Summary

In conclusion we believe that we present a reliable and scalable code maybe the first one to promote further larger usage of vdW-DF functional [1]. Moreover our code is prepared to take densities from any DFT program and due to that we hope many people who do DFT calculations will use it because Dion's functional (vdW-DF) represents a major step forward in density functional theory calculations.

11

Further - Development

Further development of the code could go in a few directions.
• profiling of the code shows that it spends most of the time in kernel function, the function that returns kernel value. This is due to the fact that interpolation is done in that function and maybe not in the best possible way. This could probably be improved by doing some state of the art implementation for this problem (by memorizing also derivative and not only kernel values at predefined points). Code is written as modularly as possible so any kind of intervention on the kernel function should be easy to change/implement.
• maybe the most important trick to save time. We are mostly interested in adsorption calculations in which one necessarily has a substrate. That substrate has usually large number of significant (i.e. density<>0) points (much larger than adsorbate) and most of these points do not change their density or gradient values due to adsorption. The idea is to compare separated parts of the system and the system itself and to see at which points the gradient and density did not change much (or at all) - in any case one should set numerical limit to that. Such points in which there is no significant change are named *f*ixed points, all the other points are called *c*hanged points. For the good energy difference (adsorption energy) we are only interested in the interaction between the following pairs of points (i.e. parts of the sum (4.3)) $changed - changed$ and $fixed - changed$. Calculation of $fixed - fixed$ is just a loss of time because this part cancels out! How it might look see in figure 11.1. This will probably be introduced in the code in such a way that a script that prepares charge density files makes comparison of the system vs. system parts densities and when writing out density files for each point write out density value and an index 0 for changed point and 1 for fixed point. In the code points should be sorted according to this index and distributed to CPUs so that each CPU has equal number of fixed and changed points in order to achieve synchronization.
• Unfortunately at the moment there is no possibility to stop the program at a certain point and to continue calculation later. This could be important on computer systems with limited calculation running time.

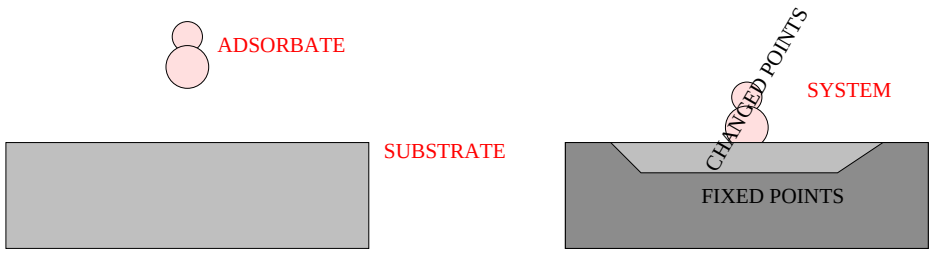

Figure 11.1: Simplified picture. Along with the explanation of the code improvement for the adsorption calculations. Comparing densities in equivalent points in the unit cell for system and for system parts (substrate and adsorbate - both relaxed) we can determine so called fixed points among which NL interaction is not important because it cancels out in calculation of adsorption energy (and that is what we are after).

12

Visualization

Having printed out all kind of physical values in the file *calc_output.txt* we can visualize all kind of things. As an example we show in figure 12.1 the overbinding PBE correlation energy in the CO molecule adsorption at TOP position on Cu(111) surface at 1/12 ML coverage.

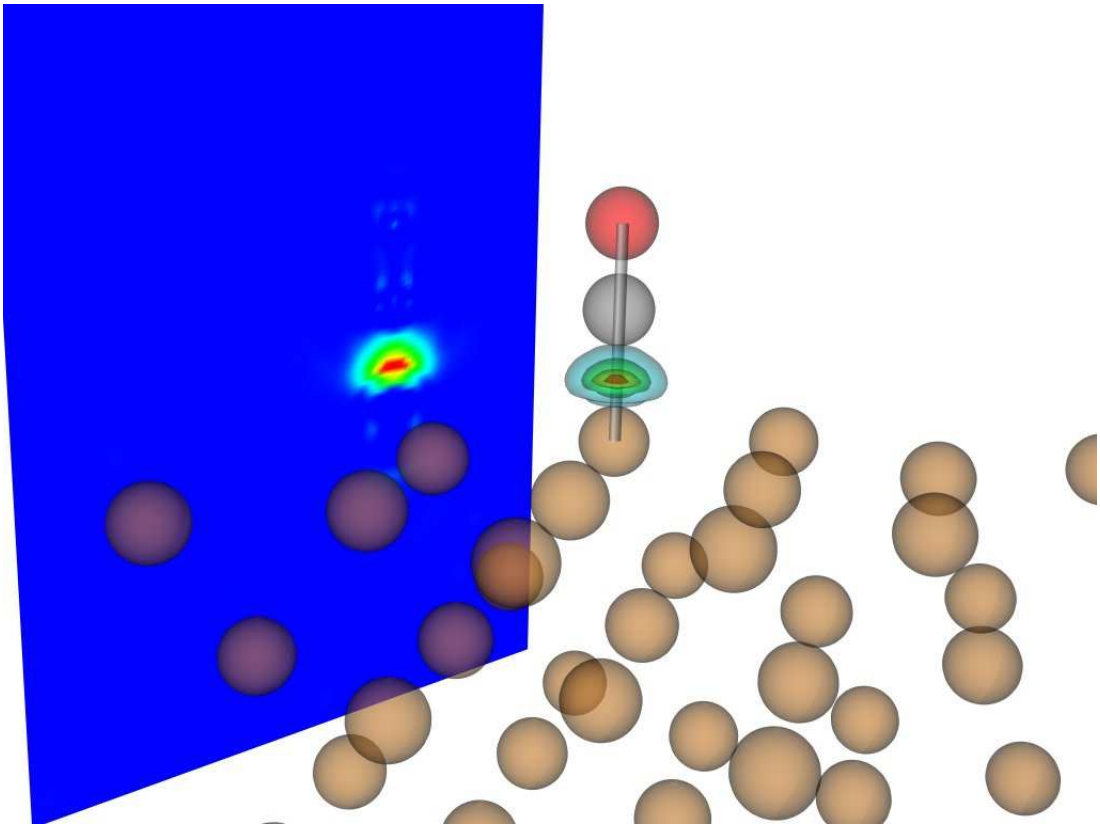

Figure 12.1: Visualization of the overbinding energy done by PBE correlation functional. We calculate binding energy due to PBE correlation in every point and then we do the same by using NL correlation energy. What we visualize is the difference of these two binding energy values - as PBEc binding energy is larger and NL is more correct we call visualized value PBEc overbinding energy.

13

Appendix A

The complete printout of a Log file looks like this:

```
Job started on 2048 processors.
using input file input_Thymine
input density file Thymine.dat
periodicity 4
derivation order 64
file used for gradientfft_thymine
trimming limit 0.10000000000000005E-03
perx,pery,perz 0 0 0
calculating gradient by means of reading from file
bufer_size_pp= 100
bufer_size= 204800
nphi0= 2048
r_cutoff= 2000.00000000000000 AA
nx,ny,nz 280 280 280
x0,y0,z0 0.00000000 0.00000000 0.00000000
ax,ay,az 37.80718336 0.00000000 0.00000000
bx,by,bz 0.00000000 37.80718336 0.00000000
cx,cy,cz 0.00000000 0.00000000 37.80718336
dx,dy,dz 0.13502565 0.13502565 0.13502565
b1x,b1y,b1z 0.16619025 0.00000000 0.00000000
b2x,b2y,b2z 0.00000000 0.16619025 0.00000000
b3x,b3y,b3z 0.00000000 0.00000000 0.16619025
dV 0.00246178
total number of points 21952000
number of points per processor 10718
starting=================
y 2008 d 30 m 5
20:  3:55:834
```

```
running on 2048 CPUs
TOTAL NUMBER OF ELECTRONS IS 47.9999977787699024
min and max density -0.263781449712000009E-08 0.961918702235999956
to distribute density******************
elapsed_time 0 D: 0 H: 14 M: 10 S: 802 MS
to calculate derivative******************
elapsed_time 0 D: 0 H: 0 M: 0 S: 0 MS
to calculate local energy******************
elapsed_time 0 D: 0 H: 0 M: 0 S: 148 MS
total number of points after trimming 719374
NUMBER OF ELECTRONS AFTER TRIMMING 47.9515497745944614
q0min,q0max 0.232349660000000013 3.34849590000000008
to load and distribute trimmed points******************
elapsed_time 0 D: 0 H: 3 M: 6 S: 618 MS
to load kernel******************
elapsed_time 0 D: 0 H: 1 M: 2 S: 417 MS
nphi,qmin_cut,phi0_analitic 2048 1.10000000000000009 2.50000000000000000
px,py,pz 3 3 3
to calculate phi0******************
elapsed_time 0 D: 2 H: 46 M: 56 S: 672 MS
to calculate energy******************
elapsed_time 0 D: 0 H: 19 M: 54 S: 561 MS
------ENERGIES (eV) ----------------
Ex_lda Ec_lda Exc_lda
TOTAL_LDA -537.49861191 -73.00488786 -610.50349978
Ex_pbe Ec_pbe Exc_pbe
TOTAL_PBE -570.80100425 -50.01874199 -620.81974624
X_rev_PBE -573.338158044232841
E_OFFDIAG_nlc E_DIAG_nlc E_TOTAL_xc_nlc
TOTAL_NL 9.78696797 0.45686059 10.24382856
needed number -Ec_pbe+Ec_lda+Enl= -12.742317309999994
------END----------------------------
to collect points******************
elapsed_time 0 D: 0 H: 0 M: 37 S: 418 MS
GAME OVER==================
y 2008 d 30 m 5
23:29:  7:866
```

Considering the above listed log file - besides techical data and time measuring values you are probably mostly interested in energy values given in eV. You should use them according to formula (4.1).

14

References

[1] M. Dion, H. Rydberg, E. Schröder, D. C. Langreth and B. I. Lundqvist *Van der Waals Denisty Functional for General Geometries* Phys. Rev. Lett. **92**, 246401, 2004.

[2] M. Dion *Van der Waals forces in density functional theory* PhD Thesis, http://www.physics.rutgers.edu/ dionmax/thesis.ps , 2004.

[3] T. Thonhauser, V. R. Cooper, S. Li, A. Puzder, P. Hyldgaard and D. C. Langreth *Van der Waals density functional: Self-consistent potential and the nature of the van der Waals bond*, Phys. Rev. B,**76**, 125112, 2007.

[4] K. Hirose, T. Ono, Y. Fujimoto, S. Tsukamoto, *First-Principles Calculations in Real-Space Foramlism*, Imperial College Press, 2005.

[5] http://cms.mpi.univie.ac.at/vasp/

[6] https://wiki.fysik.dtu.dk/dacapo

[7] http://www.pwscf.org/