flos-documentation

Release 1.0

Jun 28, 2021

Contents:

SIESTA (Spanish Initiative for Electronic Simulations with Thousands of Atoms) is a DFT simulation software. This package allows to run Densisty Functional Theory (DFT) calculations to simulate atomic-scale structures, molecules, materials, and nanodevices. This page is a technical guide on how to set up and run SIESTA-LUA along with flos library. It is not meant to describe the underlying theory: information about the mathematical and physical foundations of SIESTA can be found on the official documentation. By embedding Lua in existing fortran codes, one can exchange information from powerful DFT software "SIESTA" with scripting languages such as Lua. By abstracting the interface in fortran one can easily generalize a communication layer to facilitate on-the-fly interaction with the program. To do so we can compile the siesta with the fortran-Lua-hook library "flook". Its main usage is the ability to change run-time variables at run-time in order to optimize, or even change, the execution path of the parent program. One of the library for which developed for siesta is the "flos" (flook+siesta). This library enables optimization schemes created in Lua to be used together with SIESTA via the flook library, hence the same flo+SIESTA=FLOS. This enables scripting level languages to inter-act and develop new MD schemes, such as new geometry constraints, geometry relaxations, Nudged Elastic Band (NEB),etc.

CHAPTER 1

Setting Up FLOS for SIESTA

1.1 Installation of FLOS

./images/1.png

Fig. 1: Flook Architecture

1.1.1 Requirements

The only requirement is the Lua language. The require Lua version is 5.3. However, if you are stuck with Lua 5.2 you can apply this patch

```
patch -p1 < lua_52.patch
```

Note: For sure running siesta with lua needs the compilation of siesta with flook library, which enabling the fortran lua hook (flook) that we will discusse () section.

1.1.2 Downloading and Installation of FLOS

This Lua library may be used out of the box. To enable the use of this library you only require the LUA_PATH to contain the path to the library. Importantly this library requires an explicit <path>/?/init.lua definition. As an example the following bash commands enables the library

```
cd $HOME
git clone https://github.com/siesta-project/flos.git
cd flos
git submodule init
git submodule update
export LUA_PATH="$HOME/flos/?.lua;$HOME/flos/?/init.lua;$LUA_PATH;;"
```

and that is it. Now you can use the flos library.

1.2 Enabling SIESTA LUA interface (FLOOK)

As we mentioned we have to compile siesta with flook library.

1.2.1 Downloading and installation FLOOK

Installing flook requires you to first fetch the library which is currently hosted at github at flook. To fetch all required files do this:

```
git clone https://github.com/ElectronicStructureLibrary/flook.git
cd flook
git submodule init
git submodule update
```

Now depending of compiler Vendor you have two options:

```
* gfortran
* ifort
```

To compile with gfortran do this:

```
make VENDOR=gfortran
make liball VENDOR=gfortran
```

To compile with ifort do this:

```
make VENDOR=intel
make liball VENDOR=intel
```

After compiling you we have above libs which needed for compiling siesta:

```
flook.mod
libflook.a
libflookall.a
```

1.2.2 Downloading and Compiling SIESTA with FLOOK

To Get the latest version of SIESTA from gitlab:

```
git clone git@gitlab.com:siesta-project/siesta.git
```

Go to siesta folder:

```
cd siesta
```

Now make folder Obj-* and go to the folder:

```
~/siesta$ mkdir Obj-lua
~/siesta$ cd Obj-lua
```

Note: Here Obj-* = Obj-lua

Now setup up your Obj-lua folder like this:

```
~/siesta/Obj-lua$ sh ../Src/obj_setup.sh
```

In this step we have to make our arch.make file, here we use the (`gfortran.make`) file in (`Obj`) Folder. We need to append the following lines to our arch.make:

```
INCFLAGS += -I/{PUT YOUR FLOOK ROOT PATH}
LDFLAGS +=-L/{PUT YOUR FLOOK ROOT PATH} -Wl,-rpath={PUT YOUR FLOOK ROOT PATH}
LIBS+= -lflookall -ldl
COMP_LIBS += libfdict.a
FPPFLAGS += -DSIESTA__FLOOK
```

Now everthing ready to compile siesta :

```
~/siesta/Obj-lua$ make
```

After Compilation you should have siesta binary.

CHAPTER 2

Basic of FLOS

./images/2.png

Fig. 1: Flos Architecture

2.1 Basics

2.1.1 How it Works?!

Imagine you have a program which has 3 distinct places where interaction might occur:

```
program main
call initialize()
call calculate()
call finalize()
end program
```

At each **intermediate point** one wishes to communicate with a scripting language. flook lets you communicate fortran and Lua hence it called Flook=fortran+Lua+hook.

2.1.2 SIESTA Intermediate Points

When you run SIESTA with FLOOK enabled you have 7 intermediate point to communicate:

(1) Right after reading initial options

(2) Right before SCF step starts, but at each MD step

(3) At the start of each SCF step

(4) After each SCF has finished

(5) When moving the atoms, right after the FORCES step

(6) When SIESTA start Analyse (Post-Processing)

(7) When SIESTA is complete, just before it exists

We call above **intermediate points** state in lua script you could communicate with SIESTA viastate defination like this:

```
if siesta.state == siesta.INITIALIZE
if siesta.state == siesta.INIT_MD
if siesta.state == siesta.SCF_LOOP
if siesta.state == siesta.FORCES
if siesta.state == siesta.MOVE
if siesta.state == siesta.ANALYSIS
if siesta.state == siesta.FINALIZE
```

2.1.3 How to Communicate with SIESTA

For Communicate with siesta with it consist of two step :

(1) **set these input SIESTA flags in fdf file:**

- set `MD.TypeOfRun LUA`
- set `LUA.Script {NAME OF YOUR SCRIPT}.lua`

(2) Provide the script `{NAME OF YOUR SCRIPT}.lua`

Note: The `{NAME OF YOUR SCRIPT}.lua` should be in same folder of `psf` & `fdf` files.

2.1.4 How to prepare the LUA script for SIESTA

The SIESTA LUA scripts contains two Parts:

(1) The Main Siesta Communicator function.

(2) The user defined specific function.

The Main function contains the **intermediate points** states :

```
function siesta_comm()
  -- Do the actual communication with SIESTA

     if siesta.state == siesta.INITIALIZE then
     .
     .
     .
     end

     if siesta.state == siesta.INIT_MD then
     .
     .
     .
     end

     if siesta.state == siesta.SCF_LOOP then
     .
     .
     .
     end

     if siesta.state == siesta.FORCES then
     .
     .
     .
     end

     if siesta.state == siesta.MOVE then
     .
     .
     .
     end

     if siesta.state == siesta.ANALYSIS then
     .
     .
     .
     end

     if siesta.state == siesta.FINALIZE then
     .
     .
     .
     end
end
```

in each part of `siesta.state` we could either send or receive data with siesta dictionary. we will discuss that in () section.

The user defined function which is a normal function defined by user for specific task. For instance the above function

is counter with a return :

```
-- Step the cutoff counter and return
-- true if successfull (i.e. if there are
-- any more to check left).
-- This function will also step past values
function step_cutoff(cur_cutoff)

    if icutoff < #cutoff then
       icutoff = icutoff + 1
    else
       return false
    end

    if cutoff[icutoff] <= cur_cutoff then
       cutoff[icutoff] = cutoff[icutoff-1]
       Etot[icutoff] = Etot[icutoff-1]
       return step_cutoff(cur_cutoff)
    end

    return true
end
```

2.1.5 SIESTA LUA Dictionary

In each **intermediate points** states we could send or receive data via special name we call them SIESTA LUA dictionary. Here we categorized them:

slabel SystemLabel

DM_history_depth DM.HistoryDepth

Output Options:

dumpcharge Write.DenChar

mullipop Write.MullikenPop

hirshpop Write.HirshfeldPop

voropop Write.VoronoiPop

SCF Options:

min_nscf SCF.MinIterations

nscf SCF.MaxIterations

mixH SCF.MixHamiltonian

mix_charge SCF.MixCharge

maxsav SCF.NumberPulay

broyden_maxit SCF.NumberBroyden

wmix SCF.MixingWeight

nkick SCF.NumberKick

wmixkick SCF.KickMixingWeight

SCF Mixing Options (NEW):

scf_mixs(1)%w SCF.Mixer.Weight

scf_mixs(1)%restart SCF.Mixer.Restart

scf_mixs(1)%n_itt SCF.Mixer.Iterations

monitor_forces_in_scf SCF.MonitorForces

temp electronicTemperature

SCF Convergence Criteria:

converge_Eharr SCF.Harris.Converge

tolerance_Eharr SCF.Harris.Tolerance

converge_DM SCF.DM.Converge

dDtol SCF.DM.Tolerance

converge_EDM SCF.EDM.Converge

tolerance_EDM SCF.EDM.Tolerance

converge_H SCF.H.Converge

dHtol SCF.H.Tolerance

converge_FreeE SCF.FreeE.Converge

tolerance_FreeE SCF.FreeE.Tolerance

dxmax MD.MaxDispl

ftol MD.MaxForceTol

strtol MD.MaxStressTol

ifinal MD.FinalTimeStep

dx MD.FC.Displ

ia1 MD.FC.First

ia2 MD.FC.Last

tt MD.Temperature.Target

RelaxCellOnly MD.Relax.CellOnly

varcel MD.Relax.Cell

inicoor MD.Steps.First

fincoor MD.Steps.Last

DM_history_depth MD.DM.History.Depth

Write Options:

saveHS Write.HS

writeDM Write.DM

write_DM_at_end_of_cycle Write.EndOfCycle.DM

writeH Write.H

write_H_at_end_of_cycle Write.EndOfCycle.H

writeF Write.Forces

UseSaveDM Use.DM

hirshpop Write.Hirshfeld

voropop Write.Voronoi

Mesh Options:

g2cut Mesh.Cutoff.Minimum

saverho Mesh.Write.Rho

savedrho Mesh.Write.DeltaRho

saverhoxc Mesh.Write.RhoXC

savevh Mesh.Write.HartreePotential

savevna Mesh.Write.NeutralAtomPotential

savevt Mesh.Write.TotalPotential

savepsch Mesh.Write.IonicRho

savebader Mesh.Write.BaderRho

savetoch Mesh.Write.TotalRho

Geometry Options:

na_u geom.na_u

ucell geom.cell

ucell_last geom.cell_last

vcell geom.vcell

nsc geom.nsc

r2 geom.xa

r2 geom.xa_last

va geom.va

Species Options:

isa(1:na_u) geom.species

iza(1:na_u) geom.z

lasto(1:na_u) geom.last_orbital

amass geom.mass

qa(1:na_u) geom.neutral_charge

Datm(1:no_u) geom.orbital_charge

Force & Stress Options

cfa geom.fa

fa geom.fa_pristine

cfa geom.fa_constrained

cstress geom.stress

stress geom.stress_pristine

cstress geom.stress_constrained

Energies

DEna E.neutral_atom

DUscf E.electrostatic

Ef E.fermi

Eharrs E.harris

Ekin E.kinetic

Etot E.total

Exc E.exchange_correlation

FreeE E.free

Ekinion E.ions_kinetic

Eions E.ions

Ebs E.band_structure

Eso E.spin_orbit

Eldau E.ldau

NEGF_DE E.negf.dN

NEGF_Eharrs E.negf.harris

NEGF_Etot E.negf.total

NEGF_Ekin E.negf.kinetic

NEGF_Ebs E.negf.band_structure

Charges Options:

qtot charge.electrons

zvaltot charge.protons

k-point Options

kpoint_scf%k_cell BZ.k.Matrix

kpoint_scf%k_displ BZ.k.Displacement

Now for example if we want to recieve the information of Total Energy we could communicate like this:

```
siesta.receive({"E.total"})
```

If we want to send some information to siesta we could communicate like this:

```
siesta.send({"MD.MaxDispl"})
```

2.2 Classes

2.2.1 MDStep

The MDStep class retains information on a single MD step. Such a step may be represented by numerous quantities. One may a

(1) **R** , the atomic coordinates

(2) **V** , the velocities

(3) **F** , the forces

(4) **E** , an energy associated with the current step.

2.2.2 Array

Array Class is a generic implementation of ND arrays in pure Lua. This module tries to be as similar to the Python numpy package as possible. Due to everything being in Lua there are not *views* of arrays which means that many functions creates unnecessary data-duplications. This may be leveraged in later code implementat ons. The underlying Array class is implemented as follows:

(1) Every Array gets associated a *Shape* which determines the size of the current Array.

(2) If the Array is > 1D all elements *Array[i]* is an array with sub-Arrays of one less dimension.

(3) This enables one to call any Array function on sub-partitions of the Array without having to think about the details.

(4) The special case is the last dimension which contains the actual data. The *Array* class is using the same names as the Python numerical library *numpy* for clarity.

2.2.3 Shape

Implementation of Shape to control the size of arrays (@see Array) @classmod Shape A helper class for managing the size of *Array's*.

Having the Shape of an array in a separate class makes it much easier to implement a flexible interface for interacting with Arrays. A Shape is basically a table which defines the size of the Array the dimensions of the Array is *#Shape* while each axis size may be queried by *Shape[axis]*. Additionally a Shape may have a single dimension with size *0* which may only be used to align two shapes, i.e. the *0* axis is inferred from the total size of the aligning Shape.

2.2.4 Optimizer

Basic optimization class that is to be inherited by all the optimization classes.

2.2.5 CG

An implementation of the conjugate gradient optimization algorithm. This class implements 4 different variations of CG defined by the so-called beta-parameter:

(1) Polak-Ribiere

(2) Fletcher-Reeves

(3) Hestenes-Stiefel

(4) Dai-Yuan

Additionally this CG implementation defaults to a beta-damping factor to achieve a smooth restart method, instead of abrupt CG restarts when *beta < 0*, for instance.

2.2.6 FIRE

The implementation has several options related to the original method.

The *FIRE* optimizer implements several variations of the original FIRE algorithm.

Here we allow to differentiate on how to normalize the displacements:

(1) *correct* (argument for *FIRE:new*)

(2) "global" perform a global normalization of the coordinates (maintain displacement direction)

(3) "local" perform a local normalization (for each direction of each atom) (displacement direction is not maintained)

(4) *direction* (argument for *FIRE:new*)

(5) "global" perform a global normalization of the velocities (maintain gradient direction)

(6) "local" perform a local normalization of the velocity (for each atom) (gradient direction is not maintained) This *FIRE* optimizer allows two variations of the scaling of the velocities and the resulting displacement.

2.2.7 LBFGS

This class contains implementation of the limited memory BFGS algorithm. The LBFGS algorithm is a straightforward optimization algorithm which requires very few arguments for a succesful optimization. The most important parameter is the initial Hessian value, which for large values (close to 1) may have difficulties in converging because it is more aggressive (keeps more of the initial gradient). The default value is rather safe and should enable optimization on most systems. This optimization method also implements a history-discard strategy, if needed, for possible speeding up the convergence. A field in the argument table, *discard*, may be passed which takes one of:

(1) "none", no discard strategy

(2) "max-dF", if a displacement is being made beyond the max-displacement we do not store the step in the history

This optimization method also implements a scaling strategy, if needed, for possible speeding up the convergence. A field in the argument table, *scaling*, may be passed which takes one of:

(1) "none", no scaling strategy used

(2) "initial", only the initial inverse Hessian and use that in all subsequent iterations

(3) "every", scale for every step

2.2.8 LINE

This class conatins implementation of a line minimizer algorithm. The *Line* class optimizes a set of parameters for a function such that the gradient projected onto a gradient-direction will be minimized. I.e. it finds the minimum of a function on a gradient line such that the true gradient is orthogonal to the direction. A simple implementation of a line minimizer. This line-minimization algorithm may use any (default to *LBFGS*) optimizer and will optimize a given direction by projecting the gradient onto an initial gradient direction.

2.2.9 NEB

NEB class Instantiating a new *NEB* object. For the *NEB* object it is important to pass the images, and _then_ all the NEB settings as named arguments in a table.

– The *NEB* object implements a generic NEB algorithm as detailed in:

(1) "Improved tangent estimate in the nudged elastic band method for finding minimum energy paths and saddle points", Henkelman & Jonsson, JCP (113), 2000

(2) "A climbing image nudged elastic band method for finding saddle points and minimum energy paths", Henkelman, Uberuaga, & Jonsson, JCP (113), 2000

Note: This particular implementation has been tested and initially developed by Jesper T. Rasmussen, DTU Nanotech, 2016.

When instantiating a new *NEB* calculator one _must_ populate the initial, all intermediate images and a final image in a a table. The easiest way to do this can be seen in the below usage field. To perform the NEB calculation all images (besides the initial and final) are relaxed by an external relaxation method (see *Optimizer* and its child classes). Due to the forces being highly non-linear as the NEB algorithm updates the forces depending on the nearest images, it is adviced to use an MD-like relaxation method such as *FIRE*. If one uses history based relaxation methods (*LBFGS*, *CG*, etc.) one should limit the number of history steps used. Running the NEB class will create a huge list of files with corresponding output. Check the *NEB:save* function for details.

2.2.10 DNEB

A modification of the nudged elastic band NEB method is implementation enables stable optimizations to be run using both the limited-memory Broyden–Fletcher–Goldfarb–Shanno~L-BFGS. quasi-Newton and slow-response quenched velocity Verlet minimizers. The DNEB bject implements a generic DNEB algorithm as detailed in:

- "A doubly nudged elastic band method for finding transition states", Semen A. Trygubenkoa and David J. Wales , J. Chem. Phys., Vol. 120, No. 5, 1 February 2004.

2.2.11 VCNEB

The VC-NEB method is a more general tool for exploring the activation paths between the two end points of a phase transition process within a larger configuration space. The VC-NEB object implements a generic VC-NEB algorithm as detailed in:

- "Variable cell nudged elastic band method for studying solid–solid structural phase transitions" ,G.-R.Qianetal, Computer Physics Communications 184 (2013) 2111–2118.

2.2.12 TNEB

TNEB is a method to introduce temperature corrections to a minimum-energyreaction path. The method is based on the maximization of the flux for the Smoluchowski equationand it is implemented using a nudged-elastic-band algorithm.

The TNEB object implements a generic TNEB algorithm as detailed in:

- "A temperature-dependent nudged-elastic-band algorithm", Ramon Crehuet and Martin J. Field,The Journal of Chemical Physics 118, 9563 (2003); doi: 10.1063/1.1571817

CHAPTER 3

Tutorials of FLOS

./images/3.png

Fig. 1: What Flos Could Do. . . !

3.1 Basics

To Run Siesta with LUA functionality, has threefold steps:

(1) providing lua script in the same run folder (alongside fdf and pseudos) for a particular task.

(2) Set the MD.TypeOfRun to LUA

(3) add the lua script in fdf file using this flag : Lua.Script <lua_file_name.lua>

Note: Following Examples are only the content and structure of lua script files. All the examples could be found in the tutorial folder.

3.2 Optimizations

3.2.1 Mech Cutoff

This tutorial can take any system and will perform a series of calculations with increasing Mesh.Cutoff and it will write-out a table file to be plotted which contains the Mesh.Cutoff vs Energy.

The task steps is the following:

(1) Read Starting Mesh cutoff (`siesta.state= siesta.INITIALIZE`)

(2) Run siesta with the Starting Mesh (`siesta.state= siesta.INI_MD`)

(3) Save the Mesh and Enegy then increase the Mesh cutoff and Run siesta with new mesh (`siesta.state == siesta.MOVE`) and (user defined function)

(4) If reach to last mesh write Mesh cutoff vs Energy in file (`siesta.state == siesta.ANALYSIS`)

For optimizing our mesh we may need 3 values:

(1) cutoff_start

(2) cutoff_end

(3) cutoff_step

To do so we have these lines in our script:

```
-- Load the FLOS module
local flos = require "flos"

local cutoff_start = 150.
local cutoff_end = 650.
local cutoff_step = 50.

-- Create array of cut-offs
local cutoff = flos.Array.range(cutoff_start, cutoff_end, cutoff_step)
local Etot = flos.Array.zeros(#cutoff)
-- Initial cut-off element
local icutoff = 1
```

Note: In above lines we use flos array class to generate our array cutoffs

For user defined function we have:

```
function step_cutoff(cur_cutoff)
 if icutoff < cutoff then
    icutoff = icutoff + 1
 else
    return false
 end
 if cutoff[icutoff] <= cur_cutoff then
    cutoff[icutoff] = cutoff[icutoff-1]
    Etot[icutoff] = Etot[icutoff-1]
    return step_cutoff(cur_cutoff)
 end
 return true
 end
```

Which only increase the value of mesh with step value cur_cutoff.

Now we are ready to write our main siesta communicator function:

```
function siesta_comm()
 -- Do the actual communication with SIESTA
 if siesta.state == siesta.INITIALIZE then
    -- In the initialization step we request the
    -- Mesh cutoff (merely to be able to set it
    siesta.receive({"Mesh.Cutoff.Minimum"})
    -- Overwrite to ensure we start from the beginning
    siesta.Mesh.Cutoff.Minimum = cutoff[icutoff]
    IOprint( ("\nLUA: starting mesh-cutoff: %8.3f Ry\n"):format(cutoff[icutoff]) )
    siesta.send({"Mesh.Cutoff.Minimum"})
 end
 if siesta.state == siesta.INIT_MD then
    siesta.receive({"Mesh.Cutoff.Used"})
    -- Store the used meshcutoff for this iteration
    cutoff[icutoff] = siesta.Mesh.Cutoff.Used
 end
 if siesta.state == siesta.MOVE then
    -- Retrieve the total energy and update the
    -- meshcutoff for the next cycle
    -- Notice, we do not move, or change the geometry
    -- or cell-vectors.
    siesta.receive({"E.total","MD.Relaxed"})
    Etot[icutoff] = siesta.E.total
    -- Step the meshcutoff for the next iteration
    if step_cutoff(cutoff[icutoff]) then
        siesta.Mesh.Cutoff.Minimum = cutoff[icutoff]
    else
        siesta.MD.Relaxed = true
    end
    siesta.send({"Mesh.Cutoff.Minimum","MD.Relaxed"})
 end
 if siesta.state == siesta.ANALYSIS then
    local file = io.open("meshcutoff_E.dat", "w")
    file:write("# Mesh-cutoff vs. energy\n")
    -- We write out a table with mesh-cutoff, the difference between
    -- the last iteration, and the actual value
    file:write( ("%8.3e  %17.10e  %17.10e\n"):format(cutoff[1], 0., Etot[1]) )
    for i = 2, #cutoff do
```

(continues on next page)

(continued from previous page)

```
    file:write( ("%8.3e  %17.10e  %17.10e\n"):format(cutoff[i], Etot[i]-Etot[i-1],␣
↪Etot[i]) )
 end
    file:close()
 end
```

Note: The important thing to take away is that, siesta in `siesta.MOVE` remains to that state unless we `siesta.MD.Relaxed = true`.

3.2.2 k points

This example will perform a series of calculations with increasing k-Mesh and it will write-out a table file to be plotted which contains the k-Mesh vs Energy.

The Initialization is :

```
local kpoint_start_x = 1.
local kpoint_end_x = 10.
local kpoint_step_x = 3.
local kpoint_start_y = 1.
local kpoint_end_y = 10
local kpoint_step_y = 3.
local kpoint_start_z = 1.
local kpoint_end_z = 1.
local kpoint_step_z = 1.
local flos = require "flos"
local kpoint_cutoff_x = flos.Array.range(kpoint_start_x, kpoint_end_x, kpoint_step_x)
local kpoint_cutoff_y = flos.Array.range(kpoint_start_y, kpoint_end_y, kpoint_step_y)
local kpoint_cutoff_z = flos.Array.range(kpoint_start_z, kpoint_end_z, kpoint_step_z)
local Total_kpoints = flos.Array.zeros(3)
Total_kpoints[1] = math.max(#kpoint_cutoff_x)
Total_kpoints[2] = math.max(#kpoint_cutoff_y)
Total_kpoints[3] = math.max(#kpoint_cutoff_z)
local kpoints_num =  Total_kpoints:max()
local kpoint_mesh = flos.Array.zeros(9)
kpoint_mesh = kpoint_mesh:reshape(3,3)
local Etot = flos.Array.zeros(kpoints_num)
local ikpoint_x = 1
local ikpoint_y = 1
local ikpoint_z = 1
local kpoints_num_temp = 0
```

For user defined function we have:

```
function step_kpointf_x(cur_kpoint_x)
  if ikpoint_x < #kpoint_cutoff_x then
     ikpoint_x = ikpoint_x + 1
  else
     return false
  end
  if kpoint_cutoff_x[ikpoint_x] <= cur_kpoint_x then
     kpoint_cutoff_x[ikpoint_x] = kpoint_cutoff_x[ikpoint_x-1]
     Etot[ikpoint_x] = Etot[ikpoint_x-1]
```

(continues on next page)

(continued from previous page)

```
      return step_kpointf_x(cur_kpoint_x)
   end

   return true
end

function step_kpointf_y(cur_kpoint_y)
   if ikpoint_y < #kpoint_cutoff_y then
      ikpoint_y = ikpoint_y + 1
   else
      return false
   end
   if kpoint_cutoff_y[ikpoint_y] <= cur_kpoint_y then
      kpoint_cutoff_y[ikpoint_y] = kpoint_cutoff_y[ikpoint_y-1]
      Etot[ikpoint_y] = Etot[ikpoint_y-1]
      return step_kpointf_y(cur_kpoint_y)
   end
   return true
end

function step_kpointf_z(cur_kpoint_z)
   if ikpoint_z < #kpoint_cutoff_z then
      ikpoint_z = ikpoint_z + 1
   else
      return false
   end
   if kpoint_cutoff_z[ikpoint_z] <= cur_kpoint_z then
      kpoint_cutoff_z[ikpoint_z] = kpoint_cutoff_x[ikpoint_z-1]
      Etot[ikpoint_z] = Etot[ikpoint_z-1]
      return step_kpointf_z(cur_kpoint_z)
   end
   return true
end
```

For our main siesta communicator function we have:

```
function siesta_comm()

   if siesta.state == siesta.INITIALIZE then
      siesta.receive({"BZ.k.Matrix"})
      kpoints = flos.Array.from(siesta.BZ.k.Matrix)
      IOprint ("LUA: Provided k-point :" )--.. tostring( kpoints_num))
      kpoint_mesh = kpoints
      IOprint("LUA: k_x :\n" .. tostring(kpoint_cutoff_x))
      IOprint("LUA: k_y :\n" .. tostring(kpoint_cutoff_y))
      IOprint("LUA: k_z :\n" .. tostring(kpoint_cutoff_z))
      IOprint("LUA: Total Number of k-points :" .. tostring(Total_kpoints:max() ))
      kpoint_mesh[1][1] = kpoint_start_x
      kpoint_mesh[2][2] = kpoint_start_y
      kpoint_mesh[3][3] = kpoint_start_y
      IOprint ("LUA: Number of k-points (".. tostring(kpoints_num_temp+1) .. "/" ..␣
↪tostring(Total_kpoints:max()).. ")" )
      IOprint("LUA: Starting Kpoint :\n" .. tostring(kpoint_mesh))
      siesta.BZ.k.Matrix = kpoint_mesh
      siesta.send({"BZ.k.Matrix"})
   end
```

(continues on next page)

(continued from previous page)

```
 if siesta.state == siesta.INIT_MD then

    siesta.receive({"BZ.k.Matrix"})
 end

 if siesta.state == siesta.MOVE then

    siesta.receive({"E.total",
                    "MD.Relaxed"})

    Etot[ikpoint_x ] = siesta.E.total

    if step_kpointf_x(kpoint_cutoff_x[ikpoint_x]) then
       kpoint_mesh[1][1] = kpoint_cutoff_x[ikpoint_x]
       if step_kpointf_y(kpoint_cutoff_y[ikpoint_y]) then
          kpoint_mesh[2][2] = kpoint_cutoff_y[ikpoint_y]
          if step_kpointf_z(kpoint_cutoff_z[ikpoint_z]) then
             kpoint_mesh[3][3] = kpoint_cutoff_z[ikpoint_z]
          end
        end
    end

    siesta.BZ.k.Matrix = kpoint_mesh

    kpoints_num_temp = kpoints_num_temp + 1
    if kpoints_num == kpoints_num_temp then
       siesta.MD.Relaxed = true
    else

    IOprint ("LUA: Number of k-points (".. tostring(kpoints_num_temp+1) .. "/" ..␣
↪tostring(Total_kpoints:max()).. ")" )
    IOprint("LUA: Next Kpoint to Be Used :\n" .. tostring(siesta.BZ.k.Matrix))
    end

    siesta.send({"BZ.k.Matrix", "MD.Relaxed"})

 end

 if siesta.state == siesta.ANALYSIS then
    local file = io.open("k_meshcutoff_E.dat", "w")
    file:write("# kpoint-Mesh-cutoff vs. energy\n")
    file:write( ("%8.3e %17.10e  %17.10e\n"):format(1, Etot[1], 0.) )
    for i = 2, Total_kpoints:max()  do
       file:write( ("%8.3e %17.10e  %17.10e\n"):format(i,Etot[i], Etot[i]-Etot[i-1])␣
↪)
    end
    file:close()
 end
```

3.3 Relaxations

Whithin Lua we could have plenty of options for Relaxations. Below there are couple of those methods to apply.

3.3.1 Cell Relaxation

This example can take any geometry and will relax the cell vectors according to the siesta input options:

- MD.MaxStressTol
- MD.MaxDispl

This example defaults to two simultaneous LBFGS algorithms which seems adequate in most situations.

For user defined function we have move function which take care of relaxations part:

```
function siesta_move(siesta)

  local cell = flos.Array.from(siesta.geom.cell) / Unit.Ang
  local xa = flos.Array.from(siesta.geom.xa) / Unit.Ang
  local tmp = -flos.Array.from(siesta.geom.stress) * Unit.Ang ^ 3 / Unit.eV
  local stress = flos.Array.empty(6)
  stress[1] = tmp[1][1]
  stress[2] = tmp[2][2]
  stress[3] = tmp[3][3]
  stress[4] = (tmp[2][3] + tmp[3][2]) * 0.5
  stress[5] = (tmp[1][3] + tmp[3][1]) * 0.5
  stress[6] = (tmp[1][2] + tmp[2][1]) * 0.5
  tmp = nil
  stress = stress * stress_mask
  local vol = cell[1]:cross(cell[2]):dot(cell[3])
  local all_strain = {}
  local weight = flos.Array.empty(#LBFGS)
  for i = 1, #LBFGS do
     all_strain[i] = LBFGS[i]:optimize(strain, stress * vol)
     LBFGS[i]:optimized(stress)
     weight[i] = LBFGS[i].weight
  end

  weight = weight / weight:sum()
  if #LBFGS > 1 then
     IOprint("\nLBFGS weighted average: ", weight)
  end

  local out_strain = all_strain[1] * weight[1]
  local relaxed = LBFGS[1]:optimized()
  for i = 2, #LBFGS do
     out_strain = out_strain + all_strain[i] * weight[i]
     relaxed = relaxed and LBFGS[i]:optimized()
  end
  all_strain = nil

  strain = out_strain * stress_mask
  out_strain = nil

  local dcell = flos.Array( cell.shape )
  dcell[1][1] = 1.0 + strain[1]
  dcell[1][2] = 0.5 * strain[6]
  dcell[1][3] = 0.5 * strain[5]
  dcell[2][1] = 0.5 * strain[6]
  dcell[2][2] = 1.0 + strain[2]
  dcell[2][3] = 0.5 * strain[4]
  dcell[3][1] = 0.5 * strain[5]
```

(continues on next page)

(continued from previous page)

```
  dcell[3][2] = 0.5 * strain[4]
  dcell[3][3] = 1.0 + strain[3]

  local out_cell = cell_first:dot(dcell)
  dcell = nil

  weight = weight / weight:sum()
  if #LBFGS > 1 then
     IOprint("\nLBFGS weighted average: ", weight)
  end

  local out_strain = all_strain[1] * weight[1]
  local relaxed = LBFGS[1]:optimized()
  for i = 2, #LBFGS do
     out_strain = out_strain + all_strain[i] * weight[i]
     relaxed = relaxed and LBFGS[i]:optimized()
  end
  all_strain = nil

  strain = out_strain * stress_mask
  out_strain = nil

  local dcell = flos.Array( cell.shape )
  dcell[1][1] = 1.0 + strain[1]
  dcell[1][2] = 0.5 * strain[6]
  dcell[1][3] = 0.5 * strain[5]
  dcell[2][1] = 0.5 * strain[6]
  dcell[2][2] = 1.0 + strain[2]
  dcell[2][3] = 0.5 * strain[4]
  dcell[3][1] = 0.5 * strain[5]
  dcell[3][2] = 0.5 * strain[4]
  dcell[3][3] = 1.0 + strain[3]

  local out_cell = cell_first:dot(dcell)
  dcell = nil

  local lat = flos.Lattice:new(cell)
  local fxa = lat:fractional(xa)
  xa = fxa:dot(out_cell)
  lat = nil
  fxa = nil

  siesta.geom.cell = out_cell * Unit.Ang
  siesta.geom.xa = xa * Unit.Ang
  siesta.MD.Relaxed = relaxed

  return {"geom.cell",
          "geom.xa",
          "MD.Relaxed"}
end
```

For our main siesta communicator function we have:

```
function siesta_comm()

  local ret_tbl = {}
```

(continues on next page)

(continued from previous page)

```
  if siesta.state == siesta.INITIALIZE then

     siesta.receive({"geom.cell",
                     "MD.Relax.Cell",
                     "MD.MaxDispl",
                     "MD.MaxStressTol"})

     if not siesta.MD.Relax.Cell then

        siesta.MD.Relax.Cell = true
        ret_tbl = {"MD.Relax.Cell"}

     end

     IOprint("\nLUA convergence information for the LBFGS algorithms:")

     cell_first = flos.Array.from(siesta.geom.cell) / Unit.Ang

     for i = 1, #LBFGS do
        LBFGS[i].tolerance = siesta.MD.MaxStressTol * Unit.Ang ^ 3 / Unit.eV
        LBFGS[i].max_dF = siesta.MD.MaxDispl / Unit.Ang

        if siesta.IONode then
           LBFGS[i]:info()
        end
     end

  end

  if siesta.state == siesta.MOVE then
     siesta.receive({"geom.cell",
                     "geom.xa",
                     "geom.stress",
                     "MD.Relaxed"})
     ret_tbl = siesta_move(siesta)
  end

  siesta.send(ret_tbl)
end
```

3.3.2 Cell and Geometry Relaxation

This example can take any geometry and will relax the cell vectors according to the siesta input options:

- MD.MaxForceTol
- MD.MaxStressTol
- MD.MaxCGDispl

To initiate we have :

```
local flos = require "flos"

-- Create the two LBFGS algorithms with
-- initial Hessians 1/75 and 1/50
```

(continues on next page)

(continued from previous page)

```
local geom = {}
geom[1] = flos.LBFGS{H0 = 1. / 75.}
geom[2] = flos.LBFGS{H0 = 1. / 50.}

local lattice = {}
lattice[1] = flos.LBFGS{H0 = 1. / 75.}
lattice[2] = flos.LBFGS{H0 = 1. / 50.}

-- Grab the unit table of siesta (it is already created
-- by SIESTA)
local Unit = siesta.Units

-- Initial strain that we want to optimize to minimize
-- the stress.
local strain = flos.Array.zeros(6)
-- Mask which directions we should relax
--   [xx, yy, zz, yz, xz, xy]
-- Default to all.
local stress_mask = flos.Array.ones(6)

-- To only relax the diagonal elements you may do this:
stress_mask[4] = 0.
stress_mask[5] = 0.
stress_mask[6] = 0.

-- The initial cell
local cell_first

-- This variable controls which relaxation is performed
-- first.
-- If true, it starts by relaxing the geometry (coordinates)
--    (recommended)
-- If false, it starts by relaxing the cell vectors.
local relax_geom = true
```

For user defined function we have move couple of functions. The Fucntion which take care of Stress part is :

```
function stress_from_voigt(voigt)

  local stress = flos.Array.empty(3, 3)
  stress[1][1] = voigt[1]
  stress[1][2] = voigt[6]
  stress[1][3] = voigt[5]
  stress[2][1] = voigt[6]
  stress[2][2] = voigt[2]
  stress[2][3] = voigt[4]
  stress[3][1] = voigt[5]
  stress[3][2] = voigt[4]
  stress[3][3] = voigt[3]

  return stress
end
```

The Function which take care of geometry relaxations part:

```
function siesta_geometry(siesta)
```

(continues on next page)

(continued from previous page)

```
  local xa = siesta.geom.xa
  local fa = siesta.geom.fa

  local all_xa = {}
  local weight = flos.Array.empty(#geom)
  for i = 1, #geom do
     all_xa[i] = geom[i]:optimize(xa, fa)
     weight[i] = geom[i].weight
  end

  weight = weight / weight:sum()
  if #geom > 1 then
     IOprint("\nGeometry weighted average: ", weight)
  end

  local out_xa = all_xa[1] * weight[1]
  for i = 2, #geom do
     out_xa = out_xa + all_xa[i] * weight[i]
  end
  all_xa = nil

  siesta.geom.xa = out_xa * Unit.Ang

  return {"geom.xa"}
end
```

The Function which take care of cell relaxations part:

```
function siesta_cell(siesta)

  local cell = siesta.geom.cell
  local xa = siesta.geom.xa
  local stress = stress_to_voigt(siesta.geom.stress)
  stress = stress * stress_mask

  local vol = cell[1]:cross(cell[2]):dot(cell[3])

  local all_strain = {}
  local weight = flos.Array.empty(#lattice)
  for i = 1, #lattice do
     all_strain[i] = lattice[i]:optimize(strain, stress * vol)
     lattice[i]:optimized(stress)
     weight[i] = lattice[i].weight
  end

  weight = weight / weight:sum()
  if #lattice > 1 then
     IOprint("\nLattice weighted average: ", weight)
  end

  local out_strain = all_strain[1] * weight[1]
  for i = 2, #lattice do
     out_strain = out_strain + all_strain[i] * weight[i]
  end
  all_strain = nil

  strain = out_strain * stress_mask
```

(continues on next page)

(continued from previous page)

```
  out_strain = nil

  local dcell = flos.Array( cell.shape )
  dcell[1][1] = 1.0 + strain[1]
  dcell[1][2] = 0.5 * strain[6]
  dcell[1][3] = 0.5 * strain[5]
  dcell[2][1] = 0.5 * strain[6]
  dcell[2][2] = 1.0 + strain[2]
  dcell[2][3] = 0.5 * strain[4]
  dcell[3][1] = 0.5 * strain[5]
  dcell[3][2] = 0.5 * strain[4]
  dcell[3][3] = 1.0 + strain[3]

  local out_cell = cell_first:dot(dcell)
  dcell = nil

  local lat = flos.Lattice:new(cell)
  local fxa = lat:fractional(xa)
  xa = fxa:dot(out_cell)
  lat = nil
  fxa = nil

  siesta.geom.cell = out_cell * Unit.Ang
  siesta.geom.xa = xa * Unit.Ang

  return {"geom.cell",
          "geom.xa"}
end
```

For our main siesta communicator function we have:

```
function siesta_comm()

  local ret_tbl = {}

  if siesta.state == siesta.INITIALIZE then
     siesta.receive({"geom.cell",
                     "MD.Relax.Cell",
                     "MD.MaxDispl",
                     "MD.MaxForceTol",
                     "MD.MaxStressTol"})

     if not siesta.MD.Relax.Cell then

        siesta.MD.Relax.Cell = true
        ret_tbl = {"MD.Relax.Cell"}

     end

     IOprint("\nLUA convergence information for the LBFGS algorithms:")

     cell_first = flos.Array.from(siesta.geom.cell) / Unit.Ang

     IOprint("Lattice optimization:")
     for i = 1, #lattice do
        lattice[i].tolerance = siesta.MD.MaxStressTol * Unit.Ang ^ 3 / Unit.eV
        lattice[i].max_dF = siesta.MD.MaxDispl / Unit.Ang
```

(continues on next page)

(continued from previous page)

```
      if siesta.IONode then
         lattice[i]:info()
      end
   end

   IOprint("\nGeometry optimization:")
   for i = 1, #geom do
      geom[i].tolerance = siesta.MD.MaxForceTol * Unit.Ang / Unit.eV
      geom[i].max_dF = siesta.MD.MaxDispl / Unit.Ang

      if siesta.IONode then
         geom[i]:info()
      end
   end

   if relax_geom then
      IOprint("\nLUA: Starting with geometry relaxation!\n")
   else
      IOprint("\nLUA: Starting with cell relaxation!\n")
   end

 end

 if siesta.state == siesta.MOVE then

   siesta.receive({"geom.cell",
                   "geom.xa",
                   "geom.fa",
                   "geom.stress",
                   "MD.Relaxed"})
   ret_tbl = siesta_move(siesta)
 end

 siesta.send(ret_tbl)
end
```

For the Move Part we have :

```
function siesta_move(siesta)
  siesta.geom.cell = flos.Array.from(siesta.geom.cell) / Unit.Ang
  siesta.geom.xa = flos.Array.from(siesta.geom.xa) / Unit.Ang
  siesta.geom.fa = flos.Array.from(siesta.geom.fa) * Unit.Ang / Unit.eV
  siesta.geom.stress = -flos.Array.from(siesta.geom.stress) * Unit.Ang ^ 3 / Unit.eV

  local voigt = stress_to_voigt(siesta.geom.stress)
  voigt = voigt * stress_mask
  local conv_lattice = lattice[1]:optimized(voigt)
  voigt = nil

  local conv_geom = geom[1]:optimized(siesta.geom.fa)

  if conv_lattice and conv_geom then

     siesta.MD.Relaxed = true
     return {'MD.Relaxed'}
```

(continues on next page)

(continued from previous page)

```
  end

  if relax_geom and conv_geom then

     relax_geom = false
     for i = 1, #geom do
        geom[i]:reset()
     end

     cell_first = siesta.geom.cell:copy()

     IOprint("\nLUA: switching to cell relaxation!\n")

  elseif (not relax_geom) and conv_lattice then

     relax_geom = true
     for i = 1, #lattice do
        lattice[i]:reset()
     end

     IOprint("\nLUA: switching to geometry relaxation!\n")

  end

  if relax_geom then
     return siesta_geometry(siesta)
  else
     return siesta_cell(siesta)
  end

end
```

3.3.3 Geometry Relaxation with CG

This example can take any geometry and will relax it according to the siesta input options:

- MD.MaxForceTol
- MD.MaxCGDispl

One should note that the CG algorithm first converges when the total force (norm) on the atoms are below the tolerance. This is contrary to the SIESTA default which is a force tolerance for the individual directions, i.e. max-direction force.

This example is prepared to easily create a combined relaxation of several CG algorithms simultaneously. In some cases this is shown to speed up the convergence because an average is taken over several optimizations.

The Initialization is :

```
local flos = require "flos"

local CG = {}
CG[1] = flos.CG{beta='PR', line=flos.Line{optimizer = flos.LBFGS{H0 = 1. / 75.} } }
CG[2] = flos.CG{beta='PR', line=flos.Line{optimizer = flos.LBFGS{H0 = 1. / 50.} } }
local Unit = siesta.Units
```

For the Move Part we have :

```
function siesta_move(siesta)

  local xa = flos.Array.from(siesta.geom.xa) / Unit.Ang
  local fa = flos.Array.from(siesta.geom.fa) * Unit.Ang / Unit.eV

  local all_xa = {}
  local weight = flos.Array.empty(#CG)
  for i = 1, #CG do
     all_xa[i] = CG[i]:optimize(xa, fa)
     weight[i] = CG[i].weight

  end

  weight = weight / weight:sum()
  if #CG > 1 then
     IOprint("\nCG weighted average: ", weight)
  end

  local out_xa = all_xa[1] * weight[1]
  local relaxed = CG[1]:optimized()
  for i = 2, #CG do

     out_xa = out_xa + all_xa[i] * weight[i]
     relaxed = relaxed and CG[i]:optimized()

  end
  all_xa = nil

  siesta.geom.xa = out_xa * Unit.Ang
  siesta.MD.Relaxed = relaxed

  return {"geom.xa",
          "MD.Relaxed"}
end
```

For our main siesta communicator function we have:

```
function siesta_comm()

  local ret_tbl = {}

  if siesta.state == siesta.INITIALIZE then
     siesta.receive({"MD.MaxDispl",
                     "MD.MaxForceTol"})

     IOprint("\nLUA convergence information for the LBFGS algorithms:")
     for i = 1, #CG do
        CG[i].tolerance = siesta.MD.MaxForceTol * Unit.Ang / Unit.eV
        CG[i].max_dF = siesta.MD.MaxDispl / Unit.Ang
        CG[i].line.tolerance = CG[i].tolerance
        CG[i].line.max_dF = CG[i].max_dF -- this is not used
        CG[i].line.optimizer.tolerance = CG[i].tolerance -- this is not used
        CG[i].line.optimizer.max_dF = CG[i].max_dF -- this is used
        if siesta.IONode then
           CG[i]:info()
        end
     end
```

(continues on next page)

(continued from previous page)

```
  end

  if siesta.state == siesta.MOVE then
     siesta.receive({"geom.xa",
                     "geom.fa",
                     "MD.Relaxed"})
     ret_tbl = siesta_move(siesta)
  end

  siesta.send(ret_tbl)
end
```

3.3.4 Geometry Relaxation with Fire

This example can take any geometry and will relax it according to the siesta input options:

- MD.MaxForceTol
- MD.MaxCGDispl

One should note that the FIRE algorithm first converges when the total force (norm) on the atoms are below the tolerance. This is contrary to the SIESTA default which is a force tolerance for the individual directions, i.e. max-direction force.

The Initialization is :

```
local flos = require "flos"
local FIRE = {}
local dt_init = 0.5
FIRE[1] = flos.FIRE{dt_init = dt_init, direction="global", correct="local"}
FIRE[2] = flos.FIRE{dt_init = dt_init, direction="global", correct="global"}
FIRE[3] = flos.FIRE{dt_init = dt_init, direction="local", correct="local"}
FIRE[4] = flos.FIRE{dt_init = dt_init, direction="local", correct="global"}
local Unit = siesta.Units
```

For the Move Part we have :

```
function siesta_move(siesta)

  local xa = flos.Array.from(siesta.geom.xa) / Unit.Ang
  local fa = flos.Array.from(siesta.geom.fa) * Unit.Ang / Unit.eV

  local all_xa = {}
  local weight = flos.Array.empty(#FIRE)
  for i = 1, #FIRE do
     all_xa[i] = FIRE[i]:optimize(xa, fa)
     weight[i] = FIRE[i].weight

  end

  weight = weight / weight:sum()
  if #FIRE > 1 then
     IOprint("\nFIRE weighted average: ", weight)
  end
```

(continues on next page)

(continued from previous page)

```
  local out_xa = all_xa[1] * weight[1]
  local relaxed = FIRE[1]:optimized()
  for i = 2, #FIRE do
     out_xa = out_xa + all_xa[i] * weight[i]
     relaxed = relaxed and FIRE[i]:optimized()
  end
  all_xa = nil

  siesta.geom.xa = out_xa * Unit.Ang
  siesta.MD.Relaxed = relaxed

  return {"geom.xa",
          "MD.Relaxed"}
end
```

For our main siesta communicator function we have:

```
function siesta_comm()

  local ret_tbl = {}

  if siesta.state == siesta.INITIALIZE then

     siesta.receive({"MD.MaxDispl",
                     "MD.MaxForceTol",
                     "geom.mass"})

     IOprint("\nLUA convergence information for the FIRE algorithms:")
     for i = 1, #FIRE do

        FIRE[i].tolerance = siesta.MD.MaxForceTol * Unit.Ang / Unit.eV
        FIRE[i].max_dF = siesta.MD.MaxDispl / Unit.Ang
        FIRE[i].set_mass(siesta.geom.mass)

        if siesta.IONode then
           FIRE[i]:info()
        end
     end
  end

  if siesta.state == siesta.MOVE then

     siesta.receive({"geom.xa",
                     "geom.fa",
                     "MD.Relaxed"})

     ret_tbl = siesta_move(siesta)

  end

  siesta.send(ret_tbl)

end
```

3.3.5 Geometry Relaxation with LBFGS

This example can take any geometry and will relax it according to the siesta input options:

- MD.MaxForceTol
- MD.MaxCGDispl

One should note that the LBFGS algorithm first converges when the total force (norm) on the atoms are below the tolerance. This is contrary to the SIESTA default which is a force tolerance for the individual directions, i.e. max-direction force.

The Initialization is :

```
local flos = require "flos"

local LBFGS = {}
LBFGS[1] = flos.LBFGS{H0 = 1. / 75.}
LBFGS[2] = flos.LBFGS{H0 = 1. / 50.}
local Unit = siesta.Units
```

For the Move Part we have :

```
function siesta_move(siesta)

  local xa = flos.Array.from(siesta.geom.xa) / Unit.Ang
  local fa = flos.Array.from(siesta.geom.fa) * Unit.Ang / Unit.eV

  local all_xa = {}
  local weight = flos.Array.empty(#LBFGS)
  for i = 1, #LBFGS do
     all_xa[i] = LBFGS[i]:optimize(xa, fa)
     weight[i] = LBFGS[i].weight

  end

  weight = weight / weight:sum()
  if #LBFGS > 1 then
     IOprint("\nLBFGS weighted average: ", weight)
  end

  local out_xa = all_xa[1] * weight[1]
  local relaxed = LBFGS[1]:optimized()
  for i = 2, #LBFGS do
     out_xa = out_xa + all_xa[i] * weight[i]
     relaxed = relaxed and LBFGS[i]:optimized()

  end
  all_xa = nil

  siesta.geom.xa = out_xa * Unit.Ang
  siesta.MD.Relaxed = relaxed

  return {"geom.xa",
          "MD.Relaxed"}
end
```

For our main siesta communicator function we have:

```
function siesta_comm()

  local ret_tbl = {}
  if siesta.state == siesta.INITIALIZE then
     siesta.receive({"MD.MaxDispl",
                     "MD.MaxForceTol"})

     IOprint("\nLUA convergence information for the LBFGS algorithms:")
     for i = 1, #LBFGS do
        LBFGS[i].tolerance = siesta.MD.MaxForceTol * Unit.Ang / Unit.eV
        LBFGS[i].max_dF = siesta.MD.MaxDispl / Unit.Ang
        if siesta.IONode then
           LBFGS[i]:info()
        end
     end

  end

  if siesta.state == siesta.MOVE then
     siesta.receive({"geom.xa",
                     "geom.fa",
                     "MD.Relaxed"})
     ret_tbl = siesta_move(siesta)

  end

  siesta.send(ret_tbl)
end
```

3.3.6 Constrained Cell Relaxation

3.4 Finding Transition States Minimum Energy Path (MEP)

3.4.1 Nudged Elastic Band

Example on how to use an NEB method.

The Initialization is :

```
local image_label = "image_"
local n_images = 5
local k_spring = 1
local flos = require "flos"
local images = {}

local read_geom = function(filename)
   local file = io.open(filename, "r")
   local na = tonumber(file:read())
   local R = flos.Array.zeros(na, 3)
   file:read()
   local i = 0
   local function tovector(s)
   local t = {}
   s:gsub('%S+', function(n) t[#t+1] = tonumber(n) end)
```

(continues on next page)

(continued from previous page)

```
   return t
end
  for i = 1, na do
     local line = file:read()
     if line == nil then break end
     -- Get stuff into the R
     local v = tovector(line)
     R[i][1] = v[1]
     R[i][2] = v[2]
     R[i][3] = v[3]
  end
  file:close()
  return R
end

for i = 0, n_images + 1 do
   images[#images+1] = flos.MDStep{R=read_geom(image_label .. i .. ".xyz")}
end

local NEB = flos.NEB(images,{k=k_spring})
if siesta.IONode then
   NEB:info()
end
n_images = nil

local relax = {}
for i = 1, NEB.n_images do
   relax[i] = {}
   relax[i][1] = flos.CG{beta='PR',restart='Powell', line=flos.Line{optimizer = flos.
↪LBFGS{H0 = 1. / 25.} } }
   if siesta.IONode then
      NEB:info()
    end

end

local current_image = 1

local Unit = siesta.Units
```

some user define functions:

```
function siesta_update_DM(old, current)

  if not siesta.IONode then
     return
  end
  local DM = label .. ".DM"
  local old_DM = DM .. "." .. tostring(old)
  local current_DM = DM .. "." .. tostring(current)
  local initial_DM = DM .. ".0"
  local final_DM = DM .. "."..tostring(NEB.n_images+1)
  print ("The Label of Old DM is : " .. old_DM)
  print ("The Label of Current DM is : " .. current_DM)
  if old==0 and current==0 then
    print("Removing DM for Resuming")
    IOprint("Deleting " .. DM .. " for a clean restart...")
```

(continues on next page)

(continued from previous page)

```
    os.execute("rm " .. DM)
  end

  if 0 <= old and old <= NEB.n_images+1 and NEB:file_exists(DM) then
     IOprint("Saving " .. DM .. " to " .. old_DM)
     os.execute("mv " .. DM .. " " .. old_DM)
  elseif NEB:file_exists(DM) then
     IOprint("Deleting " .. DM .. " for a clean restart...")
     os.execute("rm " .. DM)
  end

  if NEB:file_exists(current_DM) then
     IOprint("Deleting " .. DM .. " for a clean restart...")
     os.execute("rm " .. DM)
     IOprint("Restoring " .. current_DM .. " to " .. DM)
     os.execute("cp " .. current_DM .. " " .. DM)
  end

end

function siesta_update_xyz(current)
 if not siesta.IONode then
     return
  end
 local xyz_label = image_label ..tostring(current)..".xyz"

 local f=io.open(xyz_label,"w")
 f:write(tostring(#NEB[current].R)..."\n \n")
 for i=1,#NEB[current].R do
   f:write(string.format(" %19.17f",tostring(NEB[current].R[i][1])).. "    "..string.
↪format("%19.17f",tostring(NEB[current].R[i][2]))..string.format("   %19.17f",
↪tostring(NEB[current].R[i][3]))..."\n")
end
f:close()
 --
end
```

for the Move Part we have :

```
function siesta_move(siesta)

  local fa = flos.Array.from(siesta.geom.fa) * Unit.Ang / Unit.eV
  local E = siesta.E.total / Unit.eV

  NEB[current_image]:set{F=fa, E=E}

  if current_image == 0 then
     current_image = NEB.n_images + 1
     siesta.geom.xa = NEB[current_image].R * Unit.Ang

     IOprint("\nLUA/NEB final state\n")
     return {'geom.xa'}

  elseif current_image == NEB.n_images + 1 then

     current_image = 1
```

(continues on next page)

(continued from previous page)

```
    siesta.geom.xa = NEB[current_image].R * Unit.Ang
    IOprint(("\nLUA/NEB running NEB image %d / %d\n"):format(current_image, NEB.n_
↪images))
    return {'geom.xa'}

 elseif current_image < NEB.n_images then
    current_image = current_image + 1
    siesta.geom.xa = NEB[current_image].R * Unit.Ang
    IOprint(("\nLUA/NEB running NEB image %d / %d\n"):format(current_image, NEB.n_
↪images))
    return {'geom.xa'}
 end

 local relaxed = true
 IOprint("\nNEB step")
 local out_R = {}
 for img = 1, NEB.n_images do

    local F = NEB:force(img, siesta.IONode)
    IOprint("NEB: max F on image ".. img ..
              (" = %10.5f, climbing = %s"):format(F:norm():max(),
                                                  tostring(NEB:climbing(img))) )
    local all_xa, weight = {}, flos.Array( #relax[img] )
    for i = 1, #relax[img] do
       all_xa[i] = relax[img][i]:optimize(NEB[img].R, F)
       weight[i] = relax[img][i].weight
    end
    weight = weight / weight:sum()

    if #relax[img] > 1 then
       IOprint("\n weighted average for relaxation: ", tostring(weight))
    end

    local out_xa = all_xa[1] * weight[1]
    relaxed = relaxed and relax[img][1]:optimized()
    for i = 2, #relax[img] do
       out_xa = out_xa + all_xa[i] * weight[i]
       relaxed = relaxed and relax[img][i]:optimized()
    end

    out_R[img] = out_xa

 end

 NEB:save( siesta.IONode )

 for img = 1, NEB.n_images do
    NEB[img]:set{R=out_R[img]}
 end
 current_image = 1
 if relaxed then
    siesta.geom.xa = NEB.final.R * Unit.Ang
    IOprint("\nLUA/NEB complete\n")
 else
    siesta.geom.xa = NEB[1].R * Unit.Ang
    IOprint(("\nLUA/NEB running NEB image %d / %d\n"):format(current_image, NEB.n_
↪images))
```

(continues on next page)

(continued from previous page)

```
  end

  siesta.MD.Relaxed = relaxed

  return {"geom.xa",
          "MD.Relaxed"}
end
```

For our main siesta communicator function we have:

```
function siesta_comm()

  local ret_tbl = {}

  if siesta.state == siesta.INITIALIZE then
     siesta.receive({"Label",
                     "geom.xa",
                     "MD.MaxDispl",
                     "MD.MaxForceTol"})

     label = tostring(siesta.Label)
     IOprint("\nLUA NEB calculator")

     for img = 1, NEB.n_images do
        IOprint(("\nLUA NEB relaxation method for image %d:"):format(img))
        for i = 1, #relax[img] do
           relax[img][i].tolerance = siesta.MD.MaxForceTol * Unit.Ang / Unit.eV
           relax[img][i].max_dF = siesta.MD.MaxDispl / Unit.Ang
           if siesta.IONode then
              relax[img][i]:info()
           end
        end
     end

     siesta.geom.xa = NEB.initial.R * Unit.Ang
     IOprint("\nLUA/NEB initial state\n")
     current_image = 0
     siesta_update_DM(0, current_image)
     siesta_update_xyz(current_image)
     IOprint(NEB[current_image].R)
     ret_tbl = {'geom.xa'}
  end

  if siesta.state == siesta.MOVE then

     siesta.receive({"geom.fa",
                     "E.total",
                     "MD.Relaxed"})

     local old_image = current_image

     ret_tbl = siesta_move(siesta)

     siesta_update_DM(old_image, current_image)
     siesta_update_xyz(current_image)
     IOprint(NEB[current_image].R)
```

(continues on next page)

(continued from previous page)

```
  end

  siesta.send(ret_tbl)
end
```

3.4.2 Double Nudged Elastic Band

For Using Double Nudged Elastic Band Only difference in Scripts is the initialization of DNEB object, The DNEB initialization is :

```
local NEB = flos.DNEB(images,{k=k_spring})
```

3.4.3 Variable Cell Nudged Elastic Band

Example on how to use an NEB method.

The Initialization is :

```
local flos = require "flos"
local image_label = "image_coordinates_"
local image_vector_label= "image_vectors_"
local n_images = 5
local images = {}
local images_vectors={}
--local label = "MgO-3x3x1-2V"
local f_label_xyz = "image_coordinates_"
local f_label_xyz_vec = "image_vectors_"
local read_geom = function(filename)
   local file = io.open(filename, "r")
   local na = tonumber(file:read())
   local R = flos.Array.zeros(na, 3)
   file:read()
   local i = 0
   local function tovector(s)
      local t = {}
      s:gsub('%S+', function(n) t[#t+1] = tonumber(n) end)
      return t
   end
   for i = 1, na do
```

Some user define functions:

```
function stress_to_voigt(stress)
  local voigt = flos.Array.empty(6)
  voigt[1]=stress[1][1]
  voigt[2]=stress[2][2]
  voigt[3]=stress[3][3]
  voigt[4]=(stress[2][3]+stress[3][2])*0.5
  voigt[5]=(stress[1][3]+stress[3][1])*0.5
  voigt[6]=(stress[1][2]+stress[2][1])*0.5
  return voigt
end
```

(continues on next page)

(continued from previous page)

```
function siesta_update_xyz(current)
  if not siesta.IONode then
      return
   end
  local xyz_label = f_label_xyz ..tostring(current)..".xyz"

  local f=io.open(xyz_label,"w")
  f:write(tostring(#NEB[current].R)..."\n \n")
  for i=1,#NEB[current].R do
    f:write(string.format(" %19.17f",tostring(NEB[current].R[i][1])).. "    "..string.
↪format("%19.17f",tostring(NEB[current].R[i][2]))..string.format("    %19.17f",
↪tostring(NEB[current].R[i][3]))..."\n")
end
 f:close()

end

function siesta_update_xyz_vec(current)
  if not siesta.IONode then
      return
   end
  local xyz_vec_label = f_label_xyz_vec ..tostring(current)..".xyz"
  local f=io.open(xyz_vec_label,"w")
  f:write(tostring(#VCNEB[current].R)..."\n \n")
  for i=1,#VCNEB[current].R do
    f:write(string.format(" %19.17f",tostring(VCNEB[current].R[i][1])).. "    "..
↪string.format("%19.17f",tostring(VCNEB[current].R[i][2]))..string.format("    %19.17f
↪",tostring(VCNEB[current].R[i][3]))..."\n")
 end
f:close()
 --
end

   if not siesta.IONode then
      return
   end
   local DM = label .. ".DM"
   local old_DM = DM .. "." .. tostring(old)
   local current_DM = DM .. "." .. tostring(current)
   local initial_DM = DM .. ".0"
   local final_DM = DM .. "."... tostring(NEB.n_images+1)
   print ("The Label of Old DM is : " .. old_DM)
   print ("The Label of Current DM is : " .. current_DM)
   if old==0 and current==0 then
     print("Removing DM for Resuming")
     IOprint("Deleting " .. DM .. " for a clean restart...")
     os.execute("rm " .. DM)
   end

   if 0 <= old and old <= NEB.n_images+1 and NEB:file_exists(DM) then
      IOprint("Saving " .. DM .. " to " .. old_DM)
      os.execute("mv " .. DM .. " " .. old_DM)
   elseif NEB:file_exists(DM) then
      IOprint("Deleting " .. DM .. " for a clean restart...")
      os.execute("rm " .. DM)
   end
```

(continues on next page)

(continued from previous page)

```
   if NEB:file_exists(current_DM) then
      IOprint("Deleting " .. DM .. " for a clean restart...")
      os.execute("rm " .. DM)
      IOprint("Restoring " .. current_DM .. " to " .. DM)
      os.execute("cp " .. current_DM .. " " .. DM)
   end

end
```

For the Move Part we have :

```
function siesta_move(siesta)
  local fa = flos.Array.from(siesta.geom.fa) * Unit.Ang / Unit.eV
  local E = siesta.E.total / Unit.eV
  NEB[current_image]:set{F=fa, E=E}
  local Vfa = (-flos.Array.from(siesta.geom.stress) * Unit.Ang ^ 3 / Unit.eV)--* vol
  local VE = siesta.E.total / Unit.eV
  VCNEB[current_image]:set{F=Vfa,E=VE}
  if current_image == 0 then
     current_image = NEB.n_images + 1
     siesta.geom.xa = NEB[current_image].R * Unit.Ang
     siesta.geom.cell = VCNEB[current_image].R * Unit.Ang
     IOprint("\nLUA/NEB final state\n")
     IOprint("Lattice Vectors")
     IOprint(VCNEB[current_image].R)
     IOprint("Stresss")
     IOprint(VCNEB[current_image].F)
     return {'geom.xa',"geom.stress","geom.cell"}
  elseif current_image == NEB.n_images + 1 then
     current_image = 1
     siesta.geom.xa = NEB[current_image].R * Unit.Ang
     siesta.geom.cell = VCNEB[current_image].R * Unit.Ang
     IOprint(("\nLUA/NEB running NEB image %d / %d\n"):format(current_image, NEB.n_
↪images))
     IOprint("Lattice Vectors")
     IOprint(VCNEB[current_image].R)
     IOprint("Stresss")
     IOprint(VCNEB[current_image].F)
     return {'geom.xa',"geom.stress","geom.cell"}
  elseif current_image < NEB.n_images then
    current_image = current_image + 1
     siesta.geom.xa = NEB[current_image].R * Unit.Ang
     siesta.geom.cell = VCNEB[current_image].R * Unit.Ang
     IOprint(("\nLUA/NEB running NEB image %d / %d\n"):format(current_image, NEB.n_
↪images))
     IOprint("Lattice Vectors")
     IOprint(VCNEB[current_image].R)
     IOprint("Stresss")
     IOprint(VCNEB[current_image].F)
     return {'geom.xa',"geom.stress","geom.cell"}
  end
  local relaxed = true
  local vcrelaxed = true
  local tot_relax= false
  IOprint("\nNEB step")
  local out_R = {}
  local out_VR = {}
```

(continues on next page)

(continued from previous page)

```
for img = 1, NEB.n_images do
   local F = NEB:force(img, siesta.IONode)
   IOprint("NEB: max F on image ".. img ..
              (" = %10.5f, climbing = %s"):format(F:norm():max(),
                                                  tostring(NEB:climbing(img))) )
   local all_xa, weight = {}, flos.Array( #relax[img] )
   for i = 1, #relax[img] do
      all_xa[i] = relax[img][i]:optimize(NEB[img].R, F)
      weight[i] = relax[img][i].weight
   end
   weight = weight / weight:sum()
   if #relax[img] > 1 then
      IOprint("\n weighted average for relaxation: ", tostring(weight))
   end
   local out_xa = all_xa[1] * weight[1]
   relaxed = relaxed and relax[img][1]:optimized()
   for i = 2, #relax[img] do
      out_xa = out_xa + all_xa[i] * weight[i]
      relaxed = relaxed and relax[img][i]:optimized()
   end
   local icell = VCNEB[img].R --/ Unit.Ang
   local ivol=icell[1]:cross(icell[2]):dot(icell[3])
   local strain=flos.Array.zeros(6)
   local stress_mask=flos.Array.ones(6)
   stress_mask[3]=0.0
   stress_mask[4]=0.0
   stress_mask[5]=0.0
   stress_mask[6]=0.0
   local stress=-stress_to_voigt(siesta.geom.stress)--* Unit.Ang ^ 3 / Unit.eV
   stress = stress * stress_mask
   local VF = VCNEB:force(img, siesta.IONode)
   IOprint("VCNEB: max Strain F on image ".. img ..
              (" = %10.5f, climbing = %s"):format(VF:norm():max(),
                          tostring(VCNEB:climbing(img))) )
   IOprint(VCNEB[img].F)
   local all_vcxa, vcweight = {}, flos.Array( #vcrelax[img] )
   for i = 1, #vcrelax[img] do
      all_vcxa[i] = vcrelax[img][i]:optimize(strain, stress )--* ivol
      vcweight[i] = vcrelax[img][i].weight
   end
   vcweight = vcweight / vcweight:sum()
         if #vcrelax[img] > 1 then
      IOprint("\n weighted average for cell relaxation: ", tostring(vcweight))
   end
   local out_vcxa = all_vcxa[1] * vcweight[1]
   vcrelaxed = vcrelaxed and vcrelax[img][1]:optimized()
   for i = 2, #relax[img] do
      out_vcxa = out_vcxa + all_vcxa[i] * vcweight[i]
      vcrelaxed = vcrelaxed and vcrelax[img][i]:optimized()
   end

 all_vcxa = nil   --all_strain = nil
 strain = out_vcxa * stress_mask  --strain = out_strain * stress_mask
 out_vcxa = nil --strain = out_strain * stress_mask --out_strain = nil
 local dcell = flos.Array(icell.shape)
 dcell[1][1]=1.0 + strain[1]
 dcell[1][2]=0.5 * strain[6]
```

(continues on next page)

(continued from previous page)

```
   dcell[1][3]=0.5 * strain[5]
   dcell[2][1]=0.5 * strain[6]
   dcell[2][2]=1.0 + strain[2]
   dcell[2][3]=0.5 * strain[4]
   dcell[3][1]=0.5 * strain[5]
   dcell[3][2]=0.5 * strain[4]
   dcell[3][3]=1.0 + strain[3]
   local out_cell=icell:dot(dcell)
   dcell = nil
   local lat = flos.Lattice:new(icell)
   local fxa = lat:fractional(out_xa)
   xa =fxa:dot(out_cell)
   lat = nil
   fxa = nil
   out_VR[img] = out_cell
   out_R[img] = xa
  end

  NEB:save( siesta.IONode )

  for img = 1, NEB.n_images do
     NEB[img]:set{R=out_R[img]}
     VCNEB[img]:set{R=out_VR[img]}
  end
  current_image = 1
  if relaxed and vcrelaxed then
    tot_relax= true
     siesta.geom.xa = NEB.final.R * Unit.Ang
     siesta.geom.cell = VCNEB.final.R * Unit.Ang
     IOprint("\nLUA/NEB complete\n")
  else
     siesta.geom.xa = NEB[1].R * Unit.Ang
     siesta.geom.cell = VCNEB[1].R * Unit.Ang
     IOprint(("\nLUA/NEB running NEB image %d / %d\n"):format(current_image, NEB.n_
↪images))
     IOprint("Lattice Vectors")
     IOprint(VCNEB[1].R)
     IOprint("Stresss")
     IOprint(VCNEB[1].F)
  end
  siesta.MD.Relaxed = tot_relax
  return {"geom.xa","geom.stress","geom.cell",
          "MD.Relaxed"}
end
```

For our main siesta communicator function we have:

```
function siesta_comm()
  local ret_tbl = {}
  if siesta.state == siesta.INITIALIZE then
     siesta.receive({"Label",
                     "geom.xa",
                     "MD.MaxDispl",
                     "MD.MaxForceTol",
         "MD.MaxStressTol",
         "geom.cell",
         "geom.stress"})
```

(continues on next page)

(continued from previous page)

```
    label = tostring(siesta.Label)
    IOprint("\nLUA NEB calculator")
    for img = 1, NEB.n_images do
       IOprint(("\nLUA NEB relaxation method for image %d:"):format(img))
       for i = 1, #relax[img] do
          relax[img][i].tolerance = siesta.MD.MaxForceTol * Unit.Ang / Unit.eV
          relax[img][i].max_dF = siesta.MD.MaxDispl / Unit.Ang
          vcrelax[img][i].tolerance = siesta.MD.MaxStressTol * Unit.Ang ^ 3 / Unit.eV
          vcrelax[img][i].max_dF = siesta.MD.MaxDispl / Unit.Ang
          if siesta.IONode then
             relax[img][i]:info()
       vcrelax[img][i]:info()
          end
       end
    end
    siesta.geom.xa = NEB.initial.R * Unit.Ang
    siesta.geom.cell = VCNEB.initial.R * Unit.Ang
    IOprint("\nLUA/NEB initial state\n")
    current_image = 0
    siesta_update_DM(0, current_image)
    siesta_update_xyz(current_image)
    siesta_update_xyz_vec(current_image)
    IOprint("============================================")
    IOprint("Lattice Vector")
    IOprint(VCNEB[current_image].R)
    IOprint("============================================")
    IOprint("Atomic Coordinates")
    IOprint(NEB[current_image].R)
    IOprint("============================================")
    ret_tbl = {'geom.xa',"geom.stress","geom.cell"}
  end
  if siesta.state == siesta.MOVE then
    siesta.receive({"geom.fa",
                    "E.total",
                    "MD.Relaxed",
        "geom.cell",
        "geom.stress"})
    local old_image = current_image
    ret_tbl = siesta_move(siesta)
    siesta_update_DM(old_image, current_image)
    siesta_update_xyz(current_image)
    siesta_update_xyz_vec(current_image)
  end
  siesta.send(ret_tbl)
end
```

3.4.4 Temperature Dependent Nudged Elastic Band

For Using Temperature Nudged Elastic Band Only difference in Scripts is the initialization of TNEB object with Temperature, The TNEB initialization is :

```
local NEB = flos.TNEB(images,{k=k_spring},neb_temp=300)
```

where the `neb_temp` is in `K` .

3.5 Force Constants

This example reads the input options as read by SIESTA and defines the FC type of run:

- MD.FCFirst
- MD.FCLast
- MD.FCDispl (max-displacement, i.e. for the heaviest atom)

This script will emulate the FC run built-in SIESTA and will only create the DM file for the first (x0) coordinate.

There are a couple of parameters:

> (1) same_displ = truelfalse if true all displacements will be true, and the algorithm is equivalent to the SIESTA FC run. If false, the displacements are dependent on the relative masses of the atomic species. The given displacement is then the maximum displacement, i.e. the displacement on the heaviest atom.
>
> (2) displ = {} a list of different displacements. If one is interested in several different force constant runs with different displacements, this is a simple way to do it all at once.

The Initialization is :

```
local same_displ = true

local displ = {0.005, 0.01, 0.02, 0.03, 0.04}
local flos = require "flos"
local idispl = 1
local FC = nil
local Unit = siesta.Units
```

For the Move Part we have :

```
function siesta_move(siesta)

  local fa = flos.Array.from(siesta.geom.fa) * Unit.Ang / Unit.eV

  siesta.geom.xa = FC:next(fa) * Unit.Ang
  siesta.MD.Relaxed = FC:done()

  return {"geom.xa",
          "MD.Relaxed"}
end
```

For our main siesta communicator function we have:

```
function siesta_comm()

  local ret_tbl = {}
  if siesta.state == siesta.INITIALIZE then
     siesta.receive({"geom.xa",
                     "geom.mass",
                     "MD.FC.Displ",
                     "MD.FC.First",
                     "MD.FC.Last"})

     IOprint("\nLUA Using the FC run")
     if displ == nil then
        displ = { siesta.MD.FC.Displ / Unit.Ang }
     end
```

(continues on next page)

(continued from previous page)

```
      local xa = flos.Array.from(siesta.geom.xa) / Unit.Ang
      indices = flos.Array.range(siesta.MD.FC.First, siesta.MD.FC.Last)
      if same_displ then
         FC = flos.ForceHessian(xa, indices, displ[idispl])
      else
         FC = flos.ForceHessian(xa, indices, displ[idispl],
                                siesta.geom.mass)
      end

   end

   if siesta.state == siesta.MOVE then

      siesta.receive({"geom.xa",
                      "geom.fa",
                      "Write.DM",
                      "Write.EndOfCycle.DM",
                      "MD.Relaxed"})

        ret_tbl = siesta_move(siesta)

      siesta.Write.DM = false
      ret_tbl[#ret_tbl+1] = "Write.DM"
      siesta.Write.EndOfCycle.DM = false
      ret_tbl[#ret_tbl+1] = "Write.EndOfCycle.DM"

      FC:save( ("FLOS.FC.%d"):format(idispl) )
      FC:save( ("FLOS.FCSYM.%d"):format(idispl), true )

      if siesta.MD.Relaxed then
         idispl = idispl + 1

         if idispl <= #displ then
            FC:reset()
            FC:set_displacement(displ[idispl])
            siesta.geom.xa = FC:next() * Unit.Ang
            siesta.MD.Relaxed = false

         end

      end

   end

   siesta.send(ret_tbl)
end
```

CHAPTER 4

Indices and tables