libFireDeamon

1.1

Generated by Doxygen 1.6.1

Thu Mar 2 15:21:16 2017

Contents

Chapter 1

Overview over libFireDeamon

1.1 Introduction

What you are currently viewing contains the documentation for *libFireDeamon*, a C++-library written to perform some tasks related to what I did during my time as a PhD student that will be detailed in this documentation. For any license-related information, please see the file called *COPYING* and the header of each individual C++ source file.

The library *libFireDeamon* contains functionality that I think is useful for people working in physical chemistry or quantum chemistry/physics, who perform quantum chemical calculations and evaluate them afterwards (or use them in any other way). It consists of functionality that I could not find anywhere at all or not anywhere I could just use it (like when it's in proprietary software). The functionality includes:

- a generic way to compute values defined on an arbitrary grid from values defined on an (not necessarily identical) arbitrary grid
  - realized via variadic templates
  - supports progress reports during the computation
- finding local minima in volumetric data on arbitrary grids
- compute the following chemical/physical quantities:
  - electron densities (from atomic basis sets)
  - electrostatic potentials from:
    - clouds of point charges
    - atomic basis sets
- compute isosurfaces through volumetric data sets
  - arbitrarily well discretized
  - only regular grids supported
- compute skin-surfaces around a set of spheres
  - arbitrarily well discretized
  - arbitrary radii supported
- interpolate quantities on arbitrary grids using:

  - nearest-neighbour interpolation
  - interpolation using inverse-distance weighting
- compute overlaps of atomic orbitals

The library *libFireDeamon* has been designed to be mainly used from Python via the provided language bindings. Many of the C++ functions are not that easy to use (i.e., their input is not that easily prepared in the correct format) and some sanity checks are missing. In contrast to that, the high-level Python wrapper functions perform many sanity checks and the input is more easily prepared properly. I highly recommend installing the language bindings as well.

1.2 Prerequisites

You need to have at least the following programmes/libraries installed to use *libFireDeamon:*

- a C++ compiler that supports the C++11 standard (tested with g++4.8 and icpc16.0)
- GNU make
- git (not needed if downloaded separately, e.g., as a zip-archive)

If you want to compute the electrostatic potential or density or if you want to perform any computation involving surfaces, you also need:

- CGAL (Computational Geometry Algorithms Library)
- the Boost C++ libraries

If you want to use the Python bindings (strongly recommended) you also need:

- a Python interpreter (version>=2.7.6)
- the NumPy Python module

If you change anything, you will have to regenerate the Python bindings, which you will have to do using:

- SWIG (Simplified Wrapper Interface Generator)

1.3 Installation

If you have everything installed and are running Ubuntu and are using the GNU C++ compiler, it should be sufficient to do:

```
git clone git://github.com/razziel89/libfiredeamon.git libFireDeamon
cd libFireDeamon
./configure --prefix PREFIX
make
make install
```

Please replace PREFIX with the location where you want to install *libFireDeamon*. You might have to adjust the include and library paths and the compiler type. Please run

```
./configure --help
```

for more information. The include and library paths for all necessary programmes can be adjusted separately.

Author:

Torsten Sachse

Date:

2015-2016

Version:

1.0 GNU General Public License

Chapter 2

Bug List

File isosurface.h The algorithm does not yield the correct iso surface if the points declared in *points_inside* are not actually located near the isosurface (they don't have to be inside, but they need to be close). This bug is no problem for molecules since its atoms should lie inside the isosurface.

The algorithm does not finish if the angular bound mesh criterion (first entry in *mesh_criteria*) smaller than 30.0 degrees.

The algorithm does not finish if the radii given in *radii* do not define spheres that completely enclose the to-be-generated isosurfaces.

Member make_neighbour_list_irregular segfault (at least undefined behaviour) if *max_nr_neighbours* is smaller than the number of possible neighbours a point might have

Member make_skin_surface crashes if *shrink_factor* is ≤ 0 or ≥ 1

if *nr_refinements* is large (≥ 4 for a system with 8GB RAM), the isosurface cannot be kept in memory but no error is thrown.

Chapter 3

Namespace Index

3.1 Namespace List

Here is a list of all documented namespaces with brief descriptions:

Chapter 4

Class Index

4.1 Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

Chapter 5

Class Index

5.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

Chapter 6

File Index

6.1 File List

Here is a list of all documented files with brief descriptions:

Chapter 7

Namespace Documentation

7.1 FireDeamon Namespace Reference

Python module for libFireDeamon.

Functions

- def InterpolationPy

  High level function that wraps the interpolation of arbitrary data on an irregular grid via multithreaded C++ code.

- def NeighbourListPy

  deprecated

- def IrregularNeighbourListPy

  Generate a list of neighbours of each point on an arbitrary grid.

- def RegularNeighbourListPy

  Generate a list of neighbours of each point on a regular grid.

- def LocalMinimaPy

  Given a neighbour list (as created by NeighbourListPy), find local minima.

- def SkinSurfacePy

  High level function that wraps the generation of a skin surface.

- def ElectrostaticPotentialPy

  High level function that wraps the computation of the electrostatic potential via multithreaded C++ code.

- def InitializeGridCalculationOrbitalsPy

  Initialize data required to perform some computations on a grid.

- def ElectronDensityPy

  Compute the electron density due to molecular orbitals.

- def IsosurfacePy

  High level wrapper to create an isosurface of arbitrary high discretization through volumetric data.

- def ElectrostaticPotentialOrbitalsPy

  Calculate the electron density due to some molecular orbitals on a grid.

7.1.1 Detailed Description

Python module for libFireDeamon.

7.1.2 Function Documentation

7.1.2.1 def FireDeamon::ElectronDensityPy (*coefficients_list*, *data*, *occupations* = `None`, *volume* = `1.0`, *prog_report* = `True`, *detailed_prog* = `False`, *cutoff* = `-1.0`, *correction* = `None`)

Compute the electron density due to molecular orbitals.

```
Calculate the electron density due to some molecular orbitals on a grid.

coefficients_list: list of lists of floats
    The coefficients of the molecular orbitals.
data: what InitializeGridCalculationOrbitalsPy returned

volume: float
    Scale the whole density by the inverse of this value.
prog_report: bool
    Whether or not to give progress reports over MOs.
detailed_prog:
    Whether or not to give progress reports while a MO
    is being treated.
cutoff: float in units of the grid
    No density will be computed if the difference between the
    gridpoint and the center of the basis function is larger
    than this value.
```

7.1.2.2 def FireDeamon::ElectrostaticPotentialOrbitalsPy (*coefficients_list*, *Smat*, *occupations*, *data*, *prog_report* = `True`)

Calculate the electron density due to some molecular orbitals on a grid.

```
Calculate the electron density due to some molecular orbitals on a grid.

coefficients_list: list of lists of floats
    The coefficients of the molecular orbitals.
Smat: list of lists of floats
    The overlap matrix between the primitive Gaussian functions
occupations: a list of floats
    The occupation number of the corresponding molecular orbital
data: what InitializeGridCalculationOrbitalsPy returned

prog_report: bool
    Whether or not to give progress reports over the grid.
```

7.1.2.3 def FireDeamon::ElectrostaticPotentialPy (*points*, *charges*, *coordinates*, *prog_report* = `True`, *cutoff* = `10000000.0`)

High level function that wraps the computation of the electrostatic potential via multithreaded C++ code.

```
High level function that wraps the computation of the electrostatic potential via
multithreaded C++ code.

points: a list of 3-element elements containing the Cartesian coordinates
        at which to compute the potential
charges: a list of charges at the coordinates
coordinates: a list of 3-element elements containing the Cartesian coordinates
             at which the previously given charges are localized
prog_report: whether or not to get progress reports during the computation
             (since it can take long)
```

7.1.2.4 def FireDeamon::InitializeGridCalculationOrbitalsPy (*grid*, *basis*, *scale* = `1.0`, *normalize* = `True`)

Initialize data required to perform some computations on a grid.

```
Create data structures suitable for efficiently computing
the elctron density on an arbitrary grid. Call this first
and then ElectronDensityPy(coefficients_list,data) where data
is what this function returns.

grid: list of [float,float,float]
    The Cartesian coordinates of the grid
basis: a list of [A,L,Prim]
       with
       A: a list of 3 floats
            The center of the contracted Cartesian Gaussian function
       L: a list of 3 ints
            The polynomial exponents of the contracted Cartesian Gaussian
       Prim: a list of [alpha,pre]
            with
            alpha: float
                The exponential factor of the primitive Gaussian function
            pre: float
                The contraction coefficient of the primitive Gaussian function
scale: float, optional (default: 1.0)
    Divide each coordinate by this value (coordinate transformation).
normalize: bool, optional (default: True)
    Whether or not to assume that the Gaussian functions that make up the
    primnitives are normalized or not.
```

7.1.2.5 def FireDeamon::InterpolationPy (*coordinates*, *vals*, *points*, *config* = `None`, *prog_report* = `True`)

High level function that wraps the interpolation of arbitrary data on an irregular grid via multithreaded C++ code.

```
High level function that wraps the interpolation of arbitrary data
on an irregular grid via multithreaded C++ code.

coordinates: a list of 3-element elements containing the Cartesian coordinates
             at which the given values are localized
vals: a list of values
points: a list of 3-element elements containing the Cartesian coordinates
        at which to interpolate
```

```
config: a dictionary of configuration values. Keys are:
            method: the interpolation method ('nearest', 'distance' (for weighted inverse distance))
            for distance, also are needed:
                exponent: exponent for norm
                function: 2 equals Eukledian norm, 3 equals the three norm
prog_report: whether or not to get progress reports during the computation
             (since it can take long)
```

7.1.2.6 def FireDeamon::IrregularNeighbourListPy (*grid*, *nr_neighbours*, *cutoff*, *max_nr_neighbours* = `None`, *prog_report* = `True`, *cutoff_type* = `'eukledian'`, *sort_it* = `False`)

Generate a list of neighbours of each point on an arbitrary grid.

```
Generate a list of neighbours of each point on an arbitrary grid.

grid: list of [float,float,float]
    The coordinates of each point in the grid.
nr_neighbours: int
    How many neighbours shall be seeked per gridpoint.
cutoff: float or [float,float,float] (depending on cutoff_type)
    Declare the cutoff distance for the given cutoff_type in units
    of the grid.
max_nr_neighbours: int, optional, default: nr_neighbours
    The maximum number of neighbours to be searched per gridpoint.
    This cannot be smaller than nr_neighbours. If the given number
    of neighbours has been found within the given cutoff, no further
    neighbours are being searched. So you might not get the nearest
    ones if this value is too small. Greatly impacts performance.
prog_report: boolean, optional, default: True
    Whether or not information about the progress of the calculation
    should be printed to stdout.
cutoff_type: string, optional, default: eukledian
    define how to determine whether a gridpoint is to far away from
    another to be considered its neighbour. Possible values:
    eukledian:
        The distance is the absolute value of the difference vector.
        Requires cutoff to be one float.
    manhattan_single:
        The sum of the distances in x,y and z directions is the
        distance. Requires cutoff to be one float.
    manhattan_multiple:
        Treat each Cartesian direction independently. Requires
        cutoff to be [float,float,float].
sort_it: boolean, optional, default: False
    Whether or not the neighbours found should be sorted with
    respect to the distance to the given point in increasing order.
```

7.1.2.7 def FireDeamon::IsosurfacePy (*data*, *origin*, *counts*, *delta*, *isovalue*, *points_inside*, *relative_precision* = `1.0e-05`, *mesh_criteria* = `[30]`)

High level wrapper to create an isosurface of arbitrary high discretization through volumetric data.

```
High level wrapper to create an isosurface of arbitrary high discretization
through volumetric data. The data is given on an implicit regular grid in 3
dimensions. One isosurface per element of points_inside is computed and
overlaps are discarded. Using few points for points_inside greatly speeds
up the computation.

WARNING: if points_inside does not fit the data, the algorithm might not
         yield the actual iso surface.
```

```
WARNING: the first mesh criterion (angular bound) is <30.0, the algorithm
         is not guaranteed to finish.
HINT: if creating an iso-density-surface around a molecule, it is usually
      sufficient to pass the poisition of only one atom via points_inside.

data: list of N floats
    A flat list of the volumetric data. The order of indices is that of
    dx-files, which is as follows:
        z - fast
        y - middle
        x - slow
origin: list of 3 floats
    The origin of the 3 dimensional regular grid.
counts: list of 3 int
    The number of points in each of the three directions of the grid.
    The product of these three values is the length of 'data'.
delta: a 3x3 matrix (list of 3 lists with 3 elements each)
    The three vectors stored in this parameter form the vertex of the
    regular grid on which the data is defined. The matrix than can be
    built from these vectors must have any values unequal 0.0 solely
    on its main diagonal. This means that the three axes of the grid
    have to be aligned parallel to the three Cartesian axes.
isovalue: float
    The isovalue at which to compute the isosurface.
points_inside: an iterable of [float,float,float]
    Points that are expected to lie inside of (or at least very close to)
    the resulting isosurface. In the case of molecules, this can be the
    atoms' coordinates.
relative_precision: float, optional (default: 1.0e-05)
    Precision value used to compute the isosurface. A lower value results
    in more highly discretized surfaces.
mesh_criteria: a list of A,R,D, all floats. optional (default: [30.0,5.0,5.0]
    Explanations from: http://doc.cgal.org/latest/Surface_mesher/index.html
    A: float
        Angular bound for surface mesh generation. If <30, the algorithm is
        not guaranteed to finish. This is the lower bound in degrees for
        the angles during mesh generation.
    R: float
        Radius bound used during mesh generation. It is an upper bound on
        the radii of surface Delaunay balls. A surface Delaunay ball is a
        ball circumscribing a mesh facet and centered on the surface.
    D: float
        Distance bound used during surface mesh generation. It is an upper
        bound for the distance between the circumcenter of a mesh facet and
        the center of a surface Delaunay ball of this facet.
```

7.1.2.8 def FireDeamon::LocalMinimaPy (*neighbour_list*, *values*, *degeneration*, *nr_neighbours*, *prog_report* = `False`, *upper_cutoff* = `None`, *lower_cutoff* = `None`, *sort_it* = `1`, *depths* = `None`)

Given a neighbour list (as created by NeighbourListPy), find local minima.

```
Given a neighbour list (as created by NeighbourListPy), find local minima. This is
done by comparing the data at each point to that of its neighbours. The point is
a local minimum if its associated value is at least 'degeneration' lower than that
of all its neighbours.

neighbour_list: std::vector<int> (or SWIG proxy)
    A list of neighbours. The format is: N, N1, N2, N3, ... NM
    and this repeats for every point. M is equal to nr_neighbours
    and N is the number of actual neighbours that have been found
    for the respective point.
values: list of floats
    The volumetric data in which the local minima shall be found.
```

```
degeneration: float
    As mentioned in the above description. Can be positive or negative.
prog_report: boolean, optional, default: False
    Whether or not information about the progress of the calculation
    should be printed to stdout.
upper_cutoff: float, optional, default: do not use
    Do not consider points as possible minima whose associated values
    are at least this large.
lower_cutoff: float, optional, default: do not use
    Do not consider points as possible minima whose associated values
    are at most this large.
sort_it: int, optional, default: 1
    If 0, do not sort the minima by depths and do not return the depths.
    If 1, sort the minima with respect to the difference between the value
    at the minimum and the average of all surrounding points. If depths
    is not None, also append the estimated depths.
    If 2, sort the minima with respect to the difference between the value
    at the minimum and the minimum value of all surrounding points. If depths
    is not None, also append the estimated depths.
depths: object that has an 'append' method
    if not None, append to the list (or other object) the estimated
    depths of the minima according to the value of sort_it. If it does
    not have this method, do not append the depths.
```

7.1.2.9 def FireDeamon::NeighbourListPy (*grid*, *nr_neighbours*, *cutoff*, *max_nr_neighbours* = `None`, *prog_report* = `True`, *cutoff_type* = `'eukledian'`, *sort_it* = `False`)

deprecated

```
Deprecated version of IrregularNeighbourListPy. Will be removed soon.
```

7.1.2.10 def FireDeamon::RegularNeighbourListPy (*nr_gridpoints_xyz*, *nr_neighbour_shells*, *prog_report* = `True`, *exclude_border* = `False`)

Generate a list of neighbours of each point on a regular grid.

```
Generate a list of neighbours of each point on a regular grid.
Returns a std::vector<int> with (2*nr_neighbour_shells+1)**3-1 elements per gridpoint indicating
how many neighbours there are and which ones are the neighbours. If fewer than the maximum
number of gridpoints was found (e.g. because the point is at a corner), -1's will be added.

nr_gridpoints_xyz: [int, int, int]
    The number of points in each of the three directions of the regular 3D-grid.
nr_neighbour_shells: int
    How many neighbour shells shall be treated (i.e., consider all those points to be
    neighbours who lie inside a cuboid that is spanned by 2*nr_neighbour_shells times the
    vectors that make up the grid. That cuboid is centered around each point.)
prog_report: boolean, optional, default: True
    Whether or not information about the progress of the calculation
    should be printed to stdout.
exclude_border: boolean, optional, default: False
    Whether or not points that do not have the maximum number of neighbours (i.e., those
    that lie close to or at the border) should be possible candidates for minima.
```

7.1.2.11 def FireDeamon::SkinSurfacePy (*shrink_factor*, *coordinates*, *radii*, *refinesteps* = `1`)

High level function that wraps the generation of a skin surface.

```
High level function that wraps the generation of a skin surface.

shrink_factor: shrink factor for the skin surface generation
coordinates: a list of cartesian coordinates declaring the centers of
             the spheres
radii: a list containing all the radii
refinesteps: refinement steps to perform. 0 will turn it off.
```

7.2 tuple_it Namespace Reference

namespace containing templates that can be used to perform actions for every entry in a tuple.

Classes

- struct seq

  generate a sequence of numbers

- struct gen_seq

  recursively generate a sequence of numbers and keep them in the template information

- struct gen_seq< 0, Is...>

  the struct that is the end of the recursion

Functions

- template<typename T , typename F , int... Is>
  void for_each (T *t, F f, seq< Is...>)

  Evaluate the functor for each element of the tuple. Not to be called directly.

- template<typename T , typename R , typename F , int... Is>
  void for_each_vector (T *t, R *r, F f, seq< Is...>)

  Evaluate the functor for each element of the tuple. Not to be called directly.

- template<typename... Ts, typename R , typename F >
  void for_each_in_tuple_vector (std::tuple< Ts...> *t, R *r, F f)

  Evaluate the functor for each element of the tuple. Can be called directly.

- template<typename... Ts, typename F >
  void for_each_in_tuple (std::tuple< Ts...> *t, F f)

  Evaluate the functor for each element of the tuple. Can be called directly.

7.2.1 Detailed Description

namespace containing templates that can be used to perform actions for every entry in a tuple. The "iteration" is no actual iteration as tuples are objects whose lengths have to be fully known at compile time. Access functions also have ot be known at compile time. Hence, templating is used to create a sequence 1..N where N is the length of the tuple over which to "iterate".

7.2.2 Function Documentation

7.2.2.1 template<typename T , typename F , int... Is> void tuple_it::for_each (T * *t*, F *f*, seq< Is...>) [inline]

Evaluate the functor for each element of the tuple. Not to be called directly.

Parameters:

t pointer to T - the tuple over which to "iterate" (elements will be passed to the functor)

f F - the functor who shall be called with *t* and *r* as arguments

seq<Is...> - a struct that contains the sequence of numbers in its template information

7.2.2.2 template<typename... Ts, typename F > void tuple_it::for_each_in_tuple (std::tuple< Ts...> * t, F f) [inline]

Evaluate the functor for each element of the tuple. Can be called directly.

Parameters:

t pointer to tuple - the tuple over which to "iterate" (elements will be passed to the functor)

f F - the functor who shall be called with *t* and *r* as arguments

7.2.2.3 template<typename... Ts, typename R , typename F > void tuple_it::for_each_in_tuple_vector (std::tuple< Ts...> * t, R * r, F f) [inline]

Evaluate the functor for each element of the tuple. Can be called directly. This template also allows passing an additional argument to the functor.

Parameters:

t pointer to tuple - the tuple over which to "iterate" (elements will be passed to the functor)

r pointer to R - an argument that will be passed to the functor

f F - the functor who shall be called with *t* and *r* as arguments

7.2.2.4 template<typename T , typename R , typename F , int... Is> void tuple_it::for_each_vector (T * t, R * r, F f, seq< Is...>) [inline]

Evaluate the functor for each element of the tuple. Not to be called directly. This template also allows passing an additional argument to the functor.

Parameters:

t pointer to T - the tuple over which to "iterate" (elements will be passed to the functor)

r pointer to R - an argument that will be passed to the functor

f F - the functor who shall be called with *t* and *r* as arguments

seq<Is...> - a struct that contains the sequence of numbers in its template information

Chapter 8

Class Documentation

8.1 AngInt Class Reference

Class that helps computing angular integrals that appear in pseudopotential integrals.

`#include <angular_integral.h>`

Public Member Functions

- AngInt ()

  Constructor (angular integrals are computed here).

- double GetInt (unsigned int lambda, int mu, unsigned int i, unsigned int j, unsigned int k) const

  Access the pretabulated integral values.

- ~AngInt ()

  Destructor (free all memory).

8.1.1 Detailed Description

Class that helps computing angular integrals that appear in pseudopotential integrals. The efficiency from this algorithm stems from the fact that all the angular integrals can be precomputed and then only have to be taken from the appropriate place. This class computes the integrals upon creation and provides a function to then access the data.

8.1.2 Member Function Documentation

8.1.2.1 double AngInt::GetInt (unsigned int *lambda*, int *mu*, unsigned int *i*, unsigned int *j*, unsigned int *k*) const

Access the pretabulated integral values. The integrals can be written as $\Omega_{00,\lambda\mu}^{ijk}$ when using the notation of the provided paper (DOI: 10.1002/jcc.20410). They are identical to the angular integrals that appear when computing the non-local part.

Parameters:

lambda unsigned int - the λ index (sum of angular momenta of the involved basis functions $+1$)

mu int - the μ index (magnetic quantum number of the combined basis function, satisfies $-\lambda \leq \mu \leq \lambda$

i unsigned int - first index stemming from the expansion in unitary sphere polynomials

j unsigned int - second index stemming from the expansion in unitary sphere polynomials

k unsigned int - third index stemming from the expansion in unitary sphere polynomials

Returns:

the value of the pretabulated angular integral $\Omega^{ijk}_{00,\lambda\mu}$

The documentation for this class was generated from the following files:

- include/FireDeamon/halfnum/angular_integral.h
- src/halfnum/angular_integral.cpp

8.2 copy_functor_interlace Struct Reference

Copy the data in a vector over to a number of C-type arrays each (supports interlacing).

`#include <deamon_functors.h>`

Public Member Functions

- copy_functor_interlace (int b, int s, int ni, bool i)
  Constructor for the functor.
- template<typename T >
  void operator() (T **t, std::vector< std::tuple< unsigned int, size_t, void * >> *r, int i)
  Operator that performs the operation.

Public Attributes

- unsigned int m_increment
  Size of the group that belongs together (important when interlacing).
- unsigned int m_nr_parts
  In how many parts the data shall be split, i.e., how many threads will be used for parallel computations.
- int m_nr_interlace
  Index of data stream to interlace (if at all).
- int m_interlace
  Whether or not to interlace the data stream defined by m_nr_interlace.

8.2.1 Detailed Description

Copy the data in a vector over to a number of C-type arrays each (supports interlacing). Data can be grouped together, meaing that it is possible to keep a set of data together even when interlacing the data during the copy process. Interlacing the data can help to balance the load when performing computations.

8.2.2 Constructor & Destructor Documentation

8.2.2.1 copy_functor_interlace::copy_functor_interlace (int *b*, int *s*, int *ni*, bool *i*) [inline]

Constructor for the functor.

Parameters:

b int - Size of the group that belongs together (important when interlacing)

s int - In how many parts the data shall be split, i.e., how many threads will perform a computation simultaneously

ni int - Index of data stream to interlace (if at all)

i bool - Whether or not to interlace the data stream defined by m_nr_interlace

8.2.3 Member Function Documentation

8.2.3.1 template<typename T > void copy_functor_interlace::operator() (T ** t, std::vector< std::tuple< unsigned int, size_t, void * >> * r, int i) [inline]

Operator that performs the operation.

Parameters:

- ***t*** T** - pointer to C-type array to which the data shall be copied
- ***r*** std::vector<std::tuple<unsigned int,size_t,void*>>* - a vector containing the information about the data that is to be copied over (generated by *get_size_in_bytes_and_pointer_functor*)
- ***i*** int - helper parameter that allows for looping over each element in a tuple (this is also the index for the data taken from *r*)

The documentation for this struct was generated from the following file:

- include/FireDeamon/deamon_functors.h

8.3 Copy_polyhedron_to< Polyhedron_input, Polyhedron_output > Struct Template Reference

A class that allows to copy a polyhedron declared on one kernel to a polyhedron declared on another kernel.

Public Member Functions

- **Copy_polyhedron_to** (const Polyhedron_input &in_poly)
- void **operator()** (typename Polyhedron_output::HalfedgeDS &out_hds)

8.3.1 Detailed Description

template<class Polyhedron_input, class Polyhedron_output> struct Copy_polyhedron_to< Polyhedron_input, Polyhedron_output >

A class that allows to copy a polyhedron declared on one kernel to a polyhedron declared on another kernel.

The documentation for this struct was generated from the following file:

- src/isosurface.cpp

8.4 deallocate_functor Struct Reference

Free each element in the tuple.

`#include <deamon_functors.h>`

Public Member Functions

- template<typename T >
  void operator() (T **t, int i)

  Operator that performs the operation.

8.4.1 Detailed Description

Free each element in the tuple.

8.4.2 Member Function Documentation

8.4.2.1 template<typename T > void deallocate_functor::operator() (T ** *t*, int *i*) [inline]

Operator that performs the operation.

Parameters:

- ***t*** T** - pointer to a pointer that shall be freed
- ***i*** int - helper parameter that allows for looping over each element in a tuple

The documentation for this struct was generated from the following file:

- include/FireDeamon/deamon_functors.h

8.5 tuple_it::gen_seq< N, Is > Struct Template Reference

recursively generate a sequence of numbers and keep them in the template information

`#include <iterate_over_tuple.h>`

8.5.1 Detailed Description

template<int N, int... Is> struct tuple_it::gen_seq< N, Is >

recursively generate a sequence of numbers and keep them in the template information

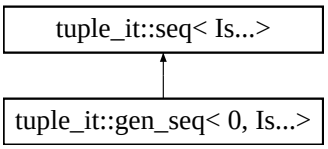
The documentation for this struct was generated from the following file:

- include/FireDeamon/iterate_over_tuple.h

8.6 tuple_it::gen_seq< 0, Is...> Struct Template Reference

the struct that is the end of the recursion

`#include <iterate_over_tuple.h>`Inheritance diagram for tuple_it::gen_seq< 0, Is...>::



8.6.1 Detailed Description

template<int... Is> struct tuple_it::gen_seq< 0, Is...>

the struct that is the end of the recursion

The documentation for this struct was generated from the following file:

- include/FireDeamon/iterate_over_tuple.h

8.7 get_size_functor Struct Reference

Add the sizes of a vector to a vector.

`#include <deamon_functors.h>`

Public Member Functions

- template<typename T >
  void operator() (std::vector< T > *t, std::vector< int > *r, int i)

  Operator that performs the operation.

8.7.1 Detailed Description

Add the sizes of a vector to a vector.

8.7.2 Member Function Documentation

8.7.2.1 template<typename T > void get_size_functor::operator() (std::vector< T > * *t*, std::vector< int > * *r*, int *i*) [inline]

Operator that performs the operation.

Parameters:

- ***t*** std::vector<T>* - pointer to the vector whose length shall be added to a vector
- ***i*** int - helper parameter that allows for looping over each element in a tuple

The documentation for this struct was generated from the following file:

- include/FireDeamon/deamon_functors.h

8.8 get_size_in_bytes_and_pointer_functor Struct Reference

Create a tuple that contains information about a vector and append that tuple to a vector.

`#include <deamon_functors.h>`

Public Member Functions

- template<typename T >
  void operator() (std::vector< T > *t, std::vector< std::tuple< unsigned int, size_t, void * >> *r, int i)

  Operator that performs the operation.

8.8.1 Detailed Description

Create a tuple that contains information about a vector and append that tuple to a vector. The information contained in the tuple that is creates is as follows:

1. number of elements in the vector
2. size in bytes of data type
3. pointers to the vector's data

8.8.2 Member Function Documentation

8.8.2.1 template<typename T > void get_size_in_bytes_and_pointer_functor::operator() (std::vector< T > * *t*, std::vector< std::tuple< unsigned int, size_t, void * >> * *r*, int *i*) [inline]

Operator that performs the operation.

Parameters:

- ***t*** std::vector<T>* - pointer to a vecotor whose information shall be extracted
- ***r*** std::vector<std::tuple<unsigned int,size_t,void*>>* - pointer to the vector to which to append the tuple
- ***i*** int - helper parameter that allows for looping over each element in a tuple

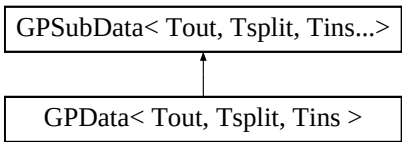
The documentation for this struct was generated from the following file:

- include/FireDeamon/deamon_functors.h

8.9 GPData< Tout, Tsplit, Tins > Class Template Reference

A templated class that contains all the data to be passed to all threads.

`#include <parallel_generic.h>`Inheritance diagram for GPData< Tout, Tsplit, Tins >::



Public Member Functions

- GPData ()

  Default constructor.

- GPData (bool progress_reports, int nr_subs, std::tuple< std::vector< Tsplit >, std::vector< Tins >...> &input, std::vector< Tout > *output, pthread_mutex_t *mutex, int *progress_bar, int split_factor_in, int split_factor_out, bool interlace)

  Alternate constructor.

- ~GPData ()

  Default destructor.

- GPSubData< Tout, Tsplit, Tins...> * GetSubData (int i)

  Get the i-th sub data (all data required for thread i in the form of an onject of type GPSubData).

- void TransferOutput (bool empty_check=true)

  Transfer the output values from the C-type array to the std::vector<Tout> used for output.

8.9.1 Detailed Description

template<typename Tout, typename Tsplit, typename... Tins> class GPData< Tout, Tsplit, Tins >

A templated class that contains all the data to be passed to all threads. This aggregates multiple instances of GPData and also spreads the data over all threads. GPData stands for "GenericParallelData". An arbitrary number of arguments can be passed to the single threads. Some bits for this class are taken from http://stackoverflow.com/questions/27941661/generating-one-class-member-per-variadic-

8.9.2 Constructor & Destructor Documentation

8.9.2.1 template<typename Tout , typename Tsplit , typename... Tins> GPData< Tout, Tsplit, Tins >::GPData (bool *progress_reports*, int *nr_subs*, std::tuple< std::vector< Tsplit >, std::vector< Tins >...> & *input*, std::vector< Tout > * *output*, pthread_mutex_t * *mutex*, int * *progress_bar*, int *split_factor_in*, int *split_factor_out*, bool *interlace*) [inline]

Alternate constructor. Allows to set most members directly.

Parameters:

progress_reports bool - whether or not progress reports are desired

nr_subs int - the number of threads to use in parallel

input

input std::tuple<std::vector<Tsplit>,std::vector<Tins>...> - the input data. The input data is given in the form of multiple objects of types std::vector<Tins> and the vector whose content shall be spread over the threads.

output pointer to std::vector<Tout> - the output data

mutex pointer to pthread_mutex_t - the mutex used to acces data thread-safely

progress_bar pointer to int - integer to be used to report progress

split_factor_in int - the number of consecutive values in the vector (whose content is to be spread over all threads) that shall remain together (e.g., would be 3 in the case of Cartesian coordinates). Only used when interlacing.

split_factor_out int - same as *split_factor_in* but for the output data

interlace bool - whether or not the input data shall be interlaced before being spread over all threads. This might help to equalize loads.

The documentation for this class was generated from the following file:

- include/FireDeamon/parallel_generic.h

8.10 GPSubData< Tout, Tins > Class Template Reference

A templated class that contains all the data to be passed to single threads.

`#include <parallel_generic.h>`

Public Member Functions

- GPSubData ()
  Default constructor.
- GPSubData (bool progress_reports, int sub_nr, std::vector< int > &len_data, std::tuple< Tins *...> &data, int len_output, Tout *output, pthread_mutex_t *mutex, int *progress_bar)
  Alternate constructor that allows to directly set most members.
- ~GPSubData ()
  Default destructor.
- template<unsigned int N>
  std::tuple_element< N, std::tuple< Tins *...> >::type GetData ()
  A method to get the n-th set of input data.
- template<unsigned int N>
  int GetNr ()
  A method to get the number of entries in the n-th set of input data.
- Tout * GetDataOutput ()
  A method to get the C-type array for the output data.
- int GetNrOutput ()
  A method to get the length of the C-type array for the output data.
- int GetSubNr ()
  Get the thread index.
- int * GetProgressBar ()
  Get the progress bar.
- bool GetProgressReports ()
  Get progress_reports.
- pthread_mutex_t * GetMutex ()
  Get mutex.

Protected Attributes

- std::tuple< Tins *...> m_data
  std::tuple<Tins...> - the input data sets*

- std::vector< int > m_lengths

  std::tuple<int> - the lenghts of the input data sets

- Tout * m_output

  pointer to Tout - C-type array for the output data

- int m_len_output

  int - length of the C-type array for the output dat

- int m_sub_nr

  int - thread index

- int m_nr_types

  int - number of template arguments

- int * m_progress_bar

  pointer to int - integer used to report progress

- pthread_mutex_t * m_mut

  pointer to pthread_mutex_t - mutex used for thread-safe access

- bool m_progress_reports

  bool - whether or not progress reports are desired

8.10.1 Detailed Description

template<typename Tout, typename... Tins> class GPSubData< Tout, Tins >

A templated class that contains all the data to be passed to single threads. Multiple instances of this class are aggregated in GPData. GPSubData stands for "GenericParallelSubData". An arbitrary number of arguments can be passed to the single threads. Some bits for this class are taken from http://stackoverflow.com/questions/27941661/generating-one-class-member-per-variadic-

8.10.2 Constructor & Destructor Documentation

8.10.2.1 template<typename Tout, typename... Tins> GPSubData< Tout, Tins >::GPSubData (bool *progress_reports*, int *sub_nr*, std::vector< int > & *len_data*, std::tuple< Tins *...> & *data*, int *len_output*, Tout * *output*, pthread_mutex_t * *mutex*, int * *progress_bar*) [inline]

Alternate constructor that allows to directly set most members.

Parameters:

progress_reports bool - whether or not a report on progress is desired

sub_nr int - a thread index (so that each threads knows its number in line)

len_data std::vector<int> - a vector that contains the lengths of all elements in *data*

data std::tuple<Tins*...> - a tuple aggregating all the data to be passed to the threads. The data has to be in C-type array format.

len_output int - the lengths of the output C-type array

output pointer to Tout - this array will be filled with the output data

mutex pointer to pthread_mutex_t - the mutex to be used

progress_bar pointer to int - the counter used for reporting progress

8.10.3 Member Function Documentation

8.10.3.1 template<typename Tout , typename... Tins> template<unsigned int N> std::tuple_element< N, std::tuple< Tins *...> >::type GPSubData< Tout, Tins >::GetData () [inline]

A method to get the n-th set of input data. The number n is passed as a template argument.

Returns:

the n-th input C-type array

8.10.3.2 template<typename Tout , typename... Tins> Tout * GPSubData< Tout, Tins >::GetDataOutput () [inline]

A method to get the C-type array for the output data.

Returns:

C-type array for the output data

8.10.3.3 template<typename Tout , typename... Tins> pthread_mutex_t * GPSubData< Tout, Tins >::GetMutex () [inline]

Get *mutex*.

Returns:

a pointer to the mutex to be used

8.10.3.4 template<typename Tout , typename... Tins> template<unsigned int N> int GPSubData< Tout, Tins >::GetNr () [inline]

A method to get the number of entries in the n-th set of input data. The number n is passed as a template argument.

Returns:

the length of the n-th input C-type array

8.10.3.5 template<typename Tout , typename... Tins> int GPSubData< Tout, Tins >::GetNrOutput () [inline]

A method to get the length of the C-type array for the output data.

Returns:

length of the C-type array for the output data

8.10.3.6 template<typename Tout , typename... Tins> int * GPSubData< Tout, Tins >::GetProgressBar () [inline]

Get the progress bar.

Returns:

a pointer to the int used to measure progress

8.10.3.7 template<typename Tout , typename... Tins> bool GPSubData< Tout, Tins >::GetProgressReports () [inline]

Get *progress_reports*.

Returns:

whether or not progress reports are desired

8.10.3.8 template<typename Tout , typename... Tins> int GPSubData< Tout, Tins >::GetSubNr () [inline]

Get the thread index.

Returns:

the thread index

The documentation for this class was generated from the following file:

- include/FireDeamon/parallel_generic.h

8.11 PG Class Reference

The class *PG* contains global information required for the parallelized computation.

`#include <parallel_generic.h>`

Public Member Functions

- PG ()

  constructor

- ~PG ()

  destructor

Public Attributes

- pthread_t * threads

  pointer to pthread_t - C-type array that allows for managing the threads (contains thread handles)

- pthread_mutex_t mutex

  pthread_mutex_t - a mutex that can be used to access data in a thread-safe way

- int nr_threads

  int - the number of threads used for the parallel computation

- int progress_bar

  int - a simple counter to estimate the progress of the compuation

8.11.1 Detailed Description

The class *PG* contains global information required for the parallelized computation. The name stands for "ParallelGlobals". Parallelization is realized using multiple threads via pthreads. A mutex (stands for "mutually exclusive") for manipulating values in onjects of the class by all threads. A progress bar is also provided to allow printing progress reports.

The documentation for this class was generated from the following files:

- include/FireDeamon/parallel_generic.h
- src/parallel_generic.cpp

8.12 Point3d Struct Reference

Public Member Functions

- Point3d (double *p)

  Alternate constructor.

- Point3d ()

  Default constructor.

- Point3d (double _x, double _y, double _z)

  Alternate constructor.

- Point3d (const Point3d &p)

  Copy constructor.

- struct Point3d operator- (struct Point3d p)

  subtract a vector

- struct Point3d operator+ (struct Point3d p)

  add a vector

- struct Point3d & operator+= (const struct Point3d p)

  add a vector directly

- struct Point3d & operator-= (const struct Point3d p)

  subtract a vector directly

- double operator[ ] (int i)

  access the vector's 3 elements

- struct Point3d operator* (struct Point3d p)

  Compute the cross product of 2 vectors.

- struct Point3d operator* (double d)

  Scale a vector by a factor.

- struct Point3d & operator/= (double d)

  Scale a vector by the inverse of a factor.

- struct Point3d & operator/= (unsigned int i)

  Scale a vector by the inverse of a factor.

- void normalize ()

  Normalize this vector.

Public Attributes

- double x

  double - the point's x-coordinate

- double y

  double - the point's y-coordinate

- double z

  double - the point's z-coordinate

8.12.1 Constructor & Destructor Documentation

8.12.1.1 Point3d::Point3d (double * p) [inline]

Alternate constructor. When given a pointer to a double, take what thsi pointer points to as the x-coordinate and the 2 values after that in memory as y- and z-coordinates.

Parameters:

p pointer to double - pointer to x-coordinate

8.12.1.2 Point3d::Point3d () [inline]

Default constructor. The point is initialized to the origin.

8.12.1.3 Point3d::Point3d (double _x, double _y, double _z) [inline]

Alternate constructor.

Parameters:

_x double - x-coordinate

_y double - y-coordinate

_z double - z-coordinate

8.12.1.4 Point3d::Point3d (const Point3d & p) [inline]

Copy constructor.

Parameters:

p Point3d - point to copy

8.12.2 Member Function Documentation

8.12.2.1 struct Point3d Point3d::operator∗ (double *d*) [inline, read]

Scale a vector by a factor.

Parameters:

d double - the scaling factor

Returns:

the scaled vector

8.12.2.2 struct Point3d Point3d::operator∗ (struct Point3d *p*) [inline, read]

Compute the cross product of 2 vectors.

Parameters:

p Point3d - vector with whom the cross product shall be computed

8.12.2.3 struct Point3d& Point3d::operator/= (unsigned int *i*) [inline, read]

Scale a vector by the inverse of a factor.

Parameters:

i int - the inverse of the scaling factor

Returns:

the scaled vector

8.12.2.4 struct Point3d& Point3d::operator/= (double *d*) [inline, read]

Scale a vector by the inverse of a factor.

Parameters:

d double - the inverse of the scaling factor

Returns:

the scaled vector

8.12.2.5 double Point3d::operator[ ] (int *i*) [inline]

access the vector's 3 elements

Returns:

an element of the vector

The documentation for this struct was generated from the following file:

- src/isosurface.cpp

8.13 RadInt Class Reference

A class that allows for computing radial integrals that appear in pseudopotential integrals.

`#include <radial_integral.h>`

Public Member Functions

- void Init (double eta, double P)

  Initialization function for the radial integration.

- double GetRadInt (int N, int lambda)

  Compute the radial integral.

8.13.1 Detailed Description

A class that allows for computing radial integrals that appear in pseudopotential integrals. Please see the documentation for angular_integral.h and the class AngInt for further details about the maths involved. These integrals are used to compute the electrostatic potential at arbitrary points in space due to molecular orbitals. The integrals are computed for the products of two primitive Cartesian Gaussian functions. The integrals can be written as T_N^λ.

The integral is computed in a coordinate system that is centered at the position at which the potential shall be computed. First, the integration is initialized using the exponential factor *eta* and the center of the combined Gaussian *P* and a lot of helper variables are initialized that allow for fast and numerically stable computation of the radial integral.

8.13.2 Member Function Documentation

8.13.2.1 double RadInt::GetRadInt (int *N*, int *lambda*)

Compute the radial integral.

Parameters:

N int - parameter N of the radial integral

lambda int - parameter λ of the radial integral

Returns:

the integral value

8.13.2.2 void RadInt::Init (double *eta*, double *P*)

Initialization function for the radial integration.

Parameters:

eta double - the exponential factor of the combined Gaussian (i.e., sum of the original ones)

P double - norm of the vector of the center of the combined Gaussian function

The documentation for this class was generated from the following files:

- include/FireDeamon/halfnum/radial_integral.h
- src/halfnum/radial_integral.cpp

8.14 tuple_it::seq< Is > Struct Template Reference

generate a sequence of numbers

`#include <iterate_over_tuple.h>`

8.14.1 Detailed Description

template<int... Is> struct tuple_it::seq< Is >

generate a sequence of numbers

The documentation for this struct was generated from the following file:

- include/FireDeamon/iterate_over_tuple.h

8.15 set_to_NULL_functor Struct Reference

Set a pointer to NULL.

`#include <deamon_functors.h>`

Public Member Functions

- template<typename T >
  void operator() (T **t, int i)

  Operator that performs the operation.

8.15.1 Detailed Description

Set a pointer to NULL.

8.15.2 Member Function Documentation

8.15.2.1 template<typename T > void set_to_NULL_functor::operator() (T ** *t*, int *i*) [inline]

Operator that performs the operation.

Parameters:

- ***t*** T** - pointer to a pointer that shall be set to NULL
- ***i*** int - helper parameter that allows for looping over each element in a tuple

The documentation for this struct was generated from the following file:

- include/FireDeamon/deamon_functors.h

8.16 Slices Class Reference

A class that aides in finding indices of neighbours to a point on a regular grid.

Public Member Functions

- Slices (int nx, int ny, int nz)

  Constructor.

- bool SetPoint (int index)

  Declare a reference point.

- int GetNeighbourIndex (int dx, int dy, int dz)

  Get the one dimensional index of a point relative to a central point.

8.16.1 Detailed Description

A class that aides in finding indices of neighbours to a point on a regular grid. The class is initialized using the grids dimensions (nx, ny, nz: number of points in each direction). Then, it is passed the one-dimensional index of a point (starting at 0 and ending at nx∗ny∗nz-1). Then, when given a displacement (in the form of index offsets in the 3 Cartesian directions) it returns the one-dimensional index of that point (if it exists in the grid). I implemented it this way because one-dimensional indices have to be used with flat data structures (which are easier to handle, IMHO) but it is easier to think in terms of three-dimensional indices when it comes to regular grids.

8.16.2 Constructor & Destructor Documentation

8.16.2.1 Slices::Slices (int *nx*, int *ny*, int *nz*) `[inline]`

Constructor.

Parameters:

nx int - number of points in x-direction

ny int - number of points in y-direction

nz int - number of points in z-direction

8.16.3 Member Function Documentation

8.16.3.1 int Slices::GetNeighbourIndex (int *dx*, int *dy*, int *dz*) `[inline]`

Get the one dimensional index of a point relative to a central point. The central point is declared using *SetPoint*.

Parameters:

dx int - index displacement in x-direction

dy int - index displacement in y-direction

dz int - index displacement in z-direction

Returns:

the one dimensional index of the point

8.16.3.2 bool Slices::SetPoint (int *index*) `[inline]`

Declare a reference point. When passing a 3d displacement to *GetNeighbourIndex*, the displacements are taken relative to the point declared in this function.

Parameters:

index int - one dimensional index of the point

Returns:

whether or not the poin is on the grid

The documentation for this class was generated from the following file:

- src/arbitrary_grid_local_minima.cpp

Chapter 9

File Documentation

9.1 include/FireDeamon/arbitrary_grid_local_minima.h File Reference

Header defining functions for searching volumetric data for local minima. `#include <vector>`

Functions

- void make_neighbour_list_irregular (bool progress_reports, int nr_gridpoints, int max_nr_neighbours, int nr_neighbours, int cutoff_type, std::vector< double > points, std::vector< double > distance_cutoff, std::vector< int > *neighbour_list, bool sort_it=true)

  Generate a list of all neighbours of an irregular grid within the given cutoff.

- void make_neighbour_list_regular (bool progress_reports, bool exclude_border, int nr_gridpoints_x, int nr_gridpoints_y, int nr_gridpoints_z, int nr_neighbour_shells, std::vector< int > *neighbour_list)

  Generate a list of all neighbours of a regular grid within the given cutoff.

- void local_minima_from_neighbour_list (bool progress_reports, int nr_neighbours, int nr_values, std::vector< int > neighbour_list, std::vector< double > values, std::vector< int > *minima, std::vector< double > degeneration_cutoffs, bool use_upper_cutoff=false, bool use_lower_cutoff=false, double upper_cutoff=0.0, double lower_cutoff=0.0, int sort_it=0, std::vector< double > *depths=NULL)

  Extract the indices of local minimum points using a pre-computed neighbour list.

9.1.1 Detailed Description

Header defining functions for searching volumetric data for local minima. The search for local minima is a two-step procedure:

1. creation of a neighbour list
2. comparison of each value with those of it's associated neighbours

This means that a point is considered to be a local minimum if and only if its associated value is smaller (you can define by how much) than those of its neighbours. First, you should call one of the two functions

- make_neighbour_list_irregular and
- make_neighbour_list_regular

depending on what type of grid your data are defined on. Then, pass the vector containing the neighbour list to local_minima_from_neighbour_list.

9.1.2 Function Documentation

9.1.2.1 void local_minima_from_neighbour_list (bool *progress_reports*, int *nr_neighbours*, int *nr_values*, std::vector< int > *neighbour_list*, std::vector< double > *values*, std::vector< int > * *minima*, std::vector< double > *degeneration_cutoffs*, bool *use_upper_cutoff* = `false`, bool *use_lower_cutoff* = `false`, double *upper_cutoff* = `0.0`, double *lower_cutoff* = `0.0`, int *sort_it* = `0`, std::vector< double > * *depths* = `NULL`)

Extract the indices of local minimum points using a pre-computed neighbour list. A local minimum is defined as a point whose associated value is smaller than that of all surrounding points (given the degeneration cutoff). Setting a negative degeneration cutoff means that a point has to have an associated value at least the absolute value of the given degeneration cutoff smaller than any sourrounding point to be considered a minimum.

Parameters:

progress_reports bool - whether or not to give progress reports

nr_neighbours int - the number of neighbours each point has (used to separate entries in *neighbour_list*

nr_values int - *nr_values* times *nr_neighbours* must be the length of *neighbour_list*

neighbour_list std::vector<int> - what make_neighbour_list_irregular or make_neighbour_list_regular fill

values std::vector<double> - the values associated with each point on the grid. If an irregular grid was used, they have to be in the same order as the points that were given to make_neighbour_list_irregular.

minima pointer to std::vector<int> - this will be filled with the indices of those points that are local minima

degeneration_cutoffs std::vector<double> - the first value will be used a s a degeneration cutoff, i.e., a point's associated value has to be this much larger than that of its neighbours to be considered a local minimum (can be negative)

use_upper_cutoff bool - whether or not to use the value in *upper_cutoff*

use_lower_cutoff bool - whether or not to use the value in *lower_cutoff*

upper_cutoff double - a point whose associated value is above this number can never be a minimum

lower_cutoff double - a point whose associated value is below this number can never be a minimum

sort_it bool - whether or not to sort the resulting minima by their depth

depths pointer to std::vector<double> - if not NULL, fill this vector with the depth of the minima (how much "lower" their values are than that of their neighbours)

9.1.2.2 void make_neighbour_list_irregular (bool *progress_reports*, int *nr_gridpoints*, int *max_nr_neighbours*, int *nr_neighbours*, int *cutoff_type*, std::vector< double > *points*, std::vector< double > *distance_cutoff*, std::vector< int > * *neighbour_list*, bool *sort_it* = `true`)

Generate a list of all neighbours of an irregular grid within the given cutoff.

Bug

segfault (at least undefined behaviour) if *max_nr_neighbours* is smaller than the number of possible neighbours a point might have

Parameters:

progress_reports bool - whether or not to give progress reports

nr_gridpoints int - the total number of points in the grid

max_nr_neighbours int - a number larger than the maximum number of points within the cutoff any single point might have

nr_neighbours int - the desired number of neighbours per point

cutoff_type int - the desired type of metric to compute whether or not points are neighbours - possible values are:

- 1: nearest neighbours
- 2: Manhattan metric independent for all 3 Cartesian directions
- 3: Manhattan metric

points std::vector<double> - a flat list of all the point Coordinates of the grid (i.e.: [x1, y1, z1, x2, y2, z2, ..., xN, yN, zN] if N == *nr_gridpoints*

distance_cutoff std::vector<double> - cutoff above which points are no longer considered

neighbour_list to be neighbours. If *cutoff_type* == 1 or 3, only the first entry in distance_cutoff is used. Otherwise, the first three elements are used (cutoff for x, y and z direction, respectively) pointer to std::vector<int> - this vector will be filled with the neighbour list, which is a flat list containing several entries. Each entry consists of the index of a point followed by the indices of its neighbours. If an entry is -1, it is to be ignored.

sort_it bool - whether or not to sort each point's neighbours by their distance from it. BEWARE: when set to *false*, you might not get the nearest neighbours if *max_nr_neighbours* > *nr_neighbours*

9.1.2.3 void make_neighbour_list_regular (bool *progress_reports*, bool *exclude_border*, int *nr_gridpoints_x*, int *nr_gridpoints_y*, int *nr_gridpoints_z*, int *nr_neighbour_shells*, std::vector< int > * *neighbour_list*)

Generate a list of all neighbours of a regular grid within the given cutoff. Although the parameters are called *nr_gridpoints_x*, *nr_gridpoints_y* and *nr_gridpoints_z*, grids whose axes are not perpendicular to each other can also be treated (by just calling the actual axes x, y and z). All explanations here, however, for the sake of simplicity, assume a cubic grid

Parameters:

progress_reports bool - whether or not to give progress reports

exclude_border bool - whether or not to allow points close to the border to be possible candidates for minima

nr_gridpoints_x int - how many points in the first direction the regular grid has

nr_gridpoints_y int - how many points in the second direction the regular grid has

nr_gridpoints_z int - how many points in the third direction the regular grid has

nr_neighbour_shells int - let *p* be the point we look at, then find all points that lie within a cube whose side length is two times *nr_neighbour_shells* the grid's lattice constant (e.g., 1 means all 26 points on the first enclosing cube)

neighbour_list pointer to std::vector<int> - this vector will be filled with the neighbour list, which is a flat list containing several entries. Each entry consists of the index of a point followed by the indices of its neighbours. If an entry is -1, it is to be ignored.

9.2 include/FireDeamon/constants.h File Reference

Definition of some constants needed for the treatment of atomic and molecular orbitals.

Variables

- const double Pi

  the number π (approx. 3.141592653589793)

- const double two_div_by_pi_to_three_fourth

  the number $(\frac{2}{\pi})^{\frac{3}{4}}$

- const double sqrt2

  the number $\sqrt{2}$

- const double sqrt_pihalf_to_3_4

  the number $\sqrt{(\frac{\pi}{2})^{\frac{3}{4}}}$

- const double odbsdfo2 [ ]
- const int factorial [ ]

  a C-type array containing the factorial of the first 11 integer numbers greater zero

- const double sqrt_two_lplus1_div4pi [ ]
- const double one_div_sqrt_factorial [ ]
- const double sqrt_factorial [ ]

  a C-type array containing the inverse values of one_div_sqrt_factorial

9.2.1 Detailed Description

Definition of some constants needed for the treatment of atomic and molecular orbitals.

9.2.2 Variable Documentation

9.2.2.1 const double odbsdfo2[ ]

a C-type array containing the integer numbers $\frac{1}{\sqrt{(2i-1)!!}} \forall i \wedge i > 0 \wedge i < 16$ and the array index is i

9.2.2.2 const double one_div_sqrt_factorial[ ]

a C-type array containing the integer numbers $\frac{1}{\sqrt{i!}} \forall i \wedge i > 0 \wedge i < 16$ and the array index is i

9.2.2.3 const double sqrt_two_lplus1_div4pi[ ]

a C-type array containing the integer numbers $\sqrt{\frac{2i+1}{4\pi}} \forall i \wedge i > 0 \wedge i < 16$ and the array index is i

9.3 include/FireDeamon/deamon_functors.h File Reference

A header that containes some functors that allow to do some things for each entry in a tuple. `#include <tuple>`

`#include <cstring>`

`#include <vector>`

Classes

- struct get_size_functor

  Add the sizes of a vector to a vector.

- struct set_to_NULL_functor

  Set a pointer to NULL.

- struct get_size_in_bytes_and_pointer_functor

  Create a tuple that contains information about a vector and append that tuple to a vector.

- struct copy_functor_interlace

  Copy the data in a vector over to a number of C-type arrays each (supports interlacing).

- struct deallocate_functor

  Free each element in the tuple.

9.3.1 Detailed Description

A header that containes some functors that allow to do some things for each entry in a tuple. They are used in conjunction with iterate_over_tuple.h to do that.

9.4 include/FireDeamon/electron_density.h File Reference

Routines to compute the electron density as well as the overlap between Gaussian-type atomic orbitals.
`#include <vector>`

Functions

- void electron_density (bool progress_reports, int num_gridpoints, std::vector< double > prim_centers, std::vector< double > prim_exponents, std::vector< double > prim_coefficients, std::vector< int > prim_angular, std::vector< double > density_grid, std::vector< double > mo_coefficients, std::vector< double > *density, double cutoff=-1.0)

  Compute the electron density on an arbitrary grid caused by molecular orbitals.

- void normalize_gaussians (std::vector< double > *prefactor, std::vector< double > exponent, std::vector< int > angular)

  Compute the normalization coefficients for a set of primitive Cartesian Gaussian functions.

9.4.1 Detailed Description

Routines to compute the electron density as well as the overlap between Gaussian-type atomic orbitals.

9.4.2 Function Documentation

9.4.2.1 void electron_density (bool *progress_reports*, int *num_gridpoints*, std::vector< double > *prim_centers*, std::vector< double > *prim_exponents*, std::vector< double > *prim_coefficients*, std::vector< int > *prim_angular*, std::vector< double > *density_grid*, std::vector< double > *mo_coefficients*, std::vector< double > * *density*, double *cutoff* = `-1.0`)

Compute the electron density on an arbitrary grid caused by molecular orbitals. Molecular orbitals are given as a linear combination of atomic orbitals and occupation numbers. The basis has to be specified in terms of normalized, primitive Cartesian Gaussian orbitals, which means that *prim_centers*, *prim_exponents*, *prim_coefficients* and *prim_angular* have to have the exact same length (considering that each primitive has one center, exponent and coefficient, but its angular momentum and center in space are each described by three values).

Parameters:

progress_reports bool - whether or not to output progress reports during the computation

num_gridpoints int - the number of points at which to compute the density

prim_centers std::vector<double> - a flat list of the Cartesian coordinates of the primitives' center (length==3N with N==no. of primitives)

prim_exponents std::vector<double> - a flat list of the exponential factors of the primitives

prim_coefficients std::vector<double> - a flat list of the preexponential factors of the primitives

prim_angular std::vector<int> - a flat list of the angular factors of the Cartesian primitives (length==3N with N==no. of primitives)

density_grid std::vector<double> - a flat list of the Cartesian coordinates at which to compute the density

mo_coefficients std::vector<double> - a flat list of coefficients specifying how te atomic basis described with the above parameters consitutes a molecular orbital

density pointer to std::vector<double> - this vector will hold the resulting density values

cutoff double - if the center of two primitives are farther away from each other than this value, do not compute the density due to the overlap of these orbitals

9.4.2.2 void normalize_gaussians (std::vector< double > * *prefactor*, std::vector< double > *exponent*, std::vector< int > *angular*)

Compute the normalization coefficients for a set of primitive Cartesian Gaussian functions.

Parameters:

prefactor pointer to std::vector<double> - this vector will hold the computed normalization coefficients in the same order used for *exponent* and *angular*

exponent std::vector<double> - a flat list of the exponential factors of the primitive Cartesian Gaussian functions

angular std::vector<int> - a flat list of the angular factors of the Cartesian primitives (length==3N with N==no. of Cartesian Gaussian functions)

9.5 include/FireDeamon/electrostatic_potential_charges.h File Reference

Compute the electrostatic potential due to a point cloud of charges. `#include <vector>`

Functions

- void electrostatic_potential (bool progress_reports, int num_points, std::vector< double > points, std::vector< double > charges_coordinates, std::vector< double > *potential, double cutoff)

  Compute the electrostatic potential due to a point cloud of charges.

9.5.1 Detailed Description

Compute the electrostatic potential due to a point cloud of charges.

9.5.2 Function Documentation

9.5.2.1 void electrostatic_potential (bool *progress_reports*, int *num_points*, std::vector< double > *points*, std::vector< double > *charges_coordinates*, std::vector< double > * *potential*, double *cutoff*)

Compute the electrostatic potential due to a point cloud of charges.

Parameters:

progress_reports bool - whether or not to print progress reports during the computation

num_points int - at how many points shall the potential be computed

points std::vector<double> - a flat list of the Cartesian coordinates of the points at which to compute the potential

charges_coordinates std::vector<double> - a flat list containing the information about the point cloud. Each charge in the cloud is described by four values:

1. its charge
2. its x-coordinate
3. its y-coordinate
4. its z-coordinate

potential pointer to std::vector<double> - this vector will hold the computed potential in the same order as the points were specified in *points*

cutoff double - if a charge is farther away than this from a point at which the the potential is to be computed, do not consider this charge. A negative value switches off this behaviour.

9.6 include/FireDeamon/electrostatic_potential_orbitals.h File Reference

Compute the electrostatic potential due to molecular orbitals. `#include <vector>`

Functions

- void electrostatic_potential_orbitals (bool progress_reports, int num_primitives, std::vector< double > prim_centers, std::vector< double > prim_exponents, std::vector< double > prim_coefficients, std::vector< int > prim_angular, std::vector< double > potential_grid, std::vector< double > P_matrix, std::vector< double > S_matrix, std::vector< double > *potential)

  Compute the electrostatic potential due to molecular orbitals.

9.6.1 Detailed Description

Compute the electrostatic potential due to molecular orbitals. These are defined as a linear combination of primitive Cartesian Gaussian functions.

9.6.2 Function Documentation

9.6.2.1 void electrostatic_potential_orbitals (bool *progress_reports*, int *num_primitives*, std::vector< double > *prim_centers*, std::vector< double > *prim_exponents*, std::vector< double > *prim_coefficients*, std::vector< int > *prim_angular*, std::vector< double > *potential_grid*, std::vector< double > *P_matrix*, std::vector< double > *S_matrix*, std::vector< double > * *potential*)

Compute the electrostatic potential due to molecular orbitals. Some matrices (P and S matrices) are usually computed on the level of contracted Cartesian Gaussian functions. However, this functions needs them *spread onto the primitives*, which means nothing more that, if a contracted function has j primitives, the value has to be duplicated j times in direct succession.

Parameters:

progress_reports whether or not to print progress reports during the computation

num_primitives int - the number of primitive functions making up the basis

prim_centers std::vector<double> - a flat list of the Cartesian coordinates of the primitives' center (length==3N with N==no. of primitives)

prim_exponents std::vector<double> - a flat list of the exponential factors of the primitives

prim_coefficients std::vector<double> - a flat list of the preexponential factors of the primitives

prim_angular std::vector<int> - a flat list of the angular factors of the Cartesian primitives (length==3N with N==no. of primitives)

potential_grid std::vector<double> - a flat vector containing the Cartesian coordinates of the points at which to compute the potential

P_matrix std::vector<double> - a flat vector containing the first order density matrix. This matrix has to be *spread onto the primitives*

S_matrix std::vector<double> - a flat vector containing the overlap matrix of the contracted Cartesian Gaussian functions. This matrix has to be *spread onto the primitives*

potential pointer to std::vector<double> - this vector will hold the computed potential in the same order as the coordinates were defined in *potential_grid*

9.7 include/FireDeamon/halfnum/angular_integral.h File Reference

Contains classes that help in computing angular integrals that appear in pseudopotential integrals.

Classes

- class AngInt

  Class that helps computing angular integrals that appear in pseudopotential integrals.

Defines

- #define **LMAXP1** 6

9.7.1 Detailed Description

Contains classes that help in computing angular integrals that appear in pseudopotential integrals. The algorithm that performs these computations is based on the following paper: Flores-Moreno, R., Alvarez-Mendez, R. J., Vela, A. and Köster, A. M. (2006), Half-numerical evaluation of pseudopotential integrals. J. Comput. Chem., 27: 1009–1019. doi:10.1002/jcc.20410

9.8 include/FireDeamon/halfnum/radial_integral.h File Reference

Contains a class that allows for computing radial integrals that appear in pseudopotential integrals.

Classes

- class RadInt

  A class that allows for computing radial integrals that appear in pseudopotential integrals.

9.8.1 Detailed Description

Contains a class that allows for computing radial integrals that appear in pseudopotential integrals. Please see the documentation for angular_integral.h for further details about the maths involved. These integrals can be written as T_N^λ.

9.9 include/FireDeamon/irregular_grid_interpolation.h File Reference

Interpolate data defined on an arbitrary grid onto another arbitrary grid. `#include <vector>`

Functions

- void generic_interpolation (bool progress_reports, int num_interpolation_points, std::vector< double > points, std::vector< double > values, std::vector< double > interpolation_points, std::vector< double > *interpolation, int interpolation_type, int distance_exponent, int distance_function, double cutoff=-1.0)

  Interpolate data defined on an arbitrary grid A onto another arbitrary grid B.

9.9.1 Detailed Description

Interpolate data defined on an arbitrary grid onto another arbitrary grid.

9.9.2 Function Documentation

9.9.2.1 void generic_interpolation (bool *progress_reports*, int *num_interpolation_points*, std::vector< double > *points*, std::vector< double > *values*, std::vector< double > *interpolation_points*, std::vector< double > * *interpolation*, int *interpolation_type*, int *distance_exponent*, int *distance_function*, double *cutoff* = `-1.0`)

Interpolate data defined on an arbitrary grid A onto another arbitrary grid B.

Parameters:

progress_reports bool - whether or not to print progress reports during the computation

num_interpolation_points int - the number of points of grid B

points std::vector<double> - a flat list containing the Cartesian coordinats of the points on grid A

values std::vector<double> - a list containung the values associated with the points whose coordinats are in *points* (i.e., those of grid A)

interpolation_points std::vector<double> - a flat list containing the Cartesian coordinats of the points on grid B

interpolation pointer to std::vector<double> - a list that will contain the values associated with the points on grid B (i.e., the interpolation result)

interpolation_type int - specify the type of interpolation to use. 1: nearest neighbour, 2: inverse distance

distance_exponent int - if using inverse-distance scaling, this is the exponent of the norm

distance_function int - if using inverse-distance scaling, declare the norm to use. The number 2 means the Eukledian norm, 3 the 3-norm, etc.

cutoff double - if a point in grid A is farther away from a point in grid B than this value, do not consider the value at that A-point to get the value at the B-point

9.10 include/FireDeamon/isosurface.h File Reference

Function to create an isosurface of arbitrary high quality through volumetric data. `#include <vector>`

Functions

- void make_isosurface (std::vector< double > data, std::vector< double > origin, std::vector< double > voxel, std::vector< int > extent, std::vector< double > points_inside, std::vector< double > mesh_criteria, std::vector< double > radii, double relative_precision, double isovalue, std::vector< int > *ivec, std::vector< double > *dvec, std::vector< double > *nvec, std::vector< int > *length)

9.10.1 Detailed Description

Function to create an isosurface of arbitrary high quality through volumetric data. The function *make_isosurface* has mainly been designed to create isosurfaces arount molecules. It is fast for single molecules but might take longer for multiple molecules (i.e., in the case of non-overlapping isosurfaces) and might not finish if certain conditions are not met. See bugs.

HINT: one isosurface computation is performed for each point specified in *points_inside*. So declaring only the required minimum (1 in the case of a single molecule) greatly speeds up the computation.

Bug

The algorithm does not yield the correct iso surface if the points declared in *points_inside* are not actually located near the isosurface (they don't have to be inside, but they need to be close). This bug is no problem for molecules since its atoms should lie inside the isosurface.

Bug

The algorithm does not finish if the angular bound mesh criterion (first entry in *mesh_criteria*) smaller than 30.0 degrees.

Bug

The algorithm does not finish if the radii given in *radii* do not define spheres that completely enclose the to-be-generated isosurfaces.

9.10.2 Function Documentation

9.10.2.1 void make_isosurface (std::vector< double > *data*, std::vector< double > *origin*, std::vector< double > *voxel*, std::vector< int > *extent*, std::vector< double > *points_inside*, std::vector< double > *mesh_criteria*, std::vector< double > *radii*, double *relative_precision*, double *isovalue*, std::vector< int > * *ivec*, std::vector< double > * *dvec*, std::vector< double > * *nvec*, std::vector< int > * *length*)

Parameters:

data std::vector<double> - a flat list containing the volumetric data. The order for the indices of the data is: z - fast, y - medium, x - slow

origin std::vector<double> - a flat list containing the origin point of the data (3 values)

voxel std::vector<double> - a flat list containing the lengths of the voxel sides. This must contain 3 values for x, y and z directions. This means that the voxel vectors need to be parallel to the 3 Cartesian axes. Of course, also non-cuboid voxels can be treated after mapping them to rectangular voxels.

extent std::vector<int> - a flat list containing the number of points in x, y and z directions.

points_inside std::vector<double> - a flat list containing the Cartesian coordinates for the points that lie within the isosurfaces. The length has to be divisible by 3.

mesh_criteria std::vector<double> - a flat list containing the three meshing criteria:

1. Angular bound for surface mesh generation. If <30, the algorithm is not guaranteed to finish. This is the lower bound in degrees for the angles during mesh generation.
2. Radius bound used during mesh generation. It is an upper bound on the radii of surface Delaunay balls. A surface Delaunay ball is a ball circumscribing a mesh facet and centered on the surface.
3. Distance bound used during surface mesh generation. It is an upper bound for the distance between the circumcenter of a mesh facet and the center of a surface Delaunay ball of this facet.

radii std::vector<double> - a flat list containing radii that, together with the points given in *points_inside*, define spheres that MUST completely enclose the isosurface that will be generated. I recommend choosing values large enough so that the entire volumetric data set is enclosed.

relative_precision double - precision value used to compute the isosurface (given relative to the radii). A lower value results in more highly discretized isosurfaces.

isovalue double - the isovalue at which to compute the isosurface

ivec pointer to std::vector<int> - this flat vector will be filled with triples of indices that specify the facets of the isosurface

dvec pointer to std::vector<double> - this flat vector will be filled with triples of values specifying the Cartesian coordinates of the vertices of the isosurface

nvec pointer to std::vector<double> - this flat vector will be filled with triples of values that specify the normal vectors associated with each vertex

length pointer to std::vector<int> - this flat vector will contain the number of vertices and the number of facets, in that order

9.11 include/FireDeamon/iterate_over_tuple.h File Reference

Header file aiding in executing code for every entry in a tuple. `#include <tuple>`

Classes

- struct tuple_it::seq< Is >

  generate a sequence of numbers

- struct tuple_it::gen_seq< N, Is >

  recursively generate a sequence of numbers and keep them in the template information

- struct tuple_it::gen_seq< 0, Is...>

  the struct that is the end of the recursion

Namespaces

- namespace tuple_it

  namespace containing templates that can be used to perform actions for every entry in a tuple.

Functions

- template<typename T , typename F , int... Is>
  void tuple_it::for_each (T *t, F f, seq< Is...>)

  Evaluate the functor for each element of the tuple. Not to be called directly.

- template<typename T , typename R , typename F , int... Is>
  void tuple_it::for_each_vector (T *t, R *r, F f, seq< Is...>)

  Evaluate the functor for each element of the tuple. Not to be called directly.

- template<typename... Ts, typename R , typename F >
  void tuple_it::for_each_in_tuple_vector (std::tuple< Ts...> *t, R *r, F f)

  Evaluate the functor for each element of the tuple. Can be called directly.

- template<typename... Ts, typename F >
  void tuple_it::for_each_in_tuple (std::tuple< Ts...> *t, F f)

  Evaluate the functor for each element of the tuple. Can be called directly.

9.11.1 Detailed Description

Header file aiding in executing code for every entry in a tuple.

9.12 include/FireDeamon/orbital_overlap.h File Reference

Functions to quickly compute normalization coefficients and overlaps of Cartesian Gaussian orbitals.

Functions

- double normalization_coefficient (double alpha, int l, int m, int n)

  Compute the normalization factor for a primitive Cartesian Gaussian orbital.

- double Sxyz (int a, int b, double diffA, double diffB, double gamma)

  compute the overlap between two one-dimensional Cartesian Gaussian functions

9.12.1 Detailed Description

Functions to quickly compute normalization coefficients and overlaps of Cartesian Gaussian orbitals.

9.12.2 Function Documentation

9.12.2.1 double normalization_coefficient (double *alpha*, int *l*, int *m*, int *n*)

Compute the normalization factor for a primitive Cartesian Gaussian orbital. The orbital is of the form: $G(\vec{r}) = (x - X_0)^l (y - Y_0)^m (z - Z_0)^n \cdot \mathrm{e}^{-\alpha(\vec{r}-\vec{R}_0)^2}$ with $\vec{R}_0 = \begin{pmatrix} X_0 \\ Y_0 \\ Z_0 \end{pmatrix}$ and $\vec{r} = \begin{pmatrix} x \\ y \\ z \end{pmatrix}$

Parameters:

alpha double - the exponential factor α

l int - first angular momentum factor l

m int - second angular momentum factor m

n int - third angular momentum factor n

Returns:

normalization coefficient

9.12.2.2 double Sxyz (int *a*, int *b*, double *diffA*, double *diffB*, double *gamma*)

compute the overlap between two one-dimensional Cartesian Gaussian functions Such functions have the form $G(x) = (x - X_0)^{a/b} \cdot \mathrm{e}^{-\alpha/\beta(x-X_0)^2}$. Such a computation can be simplified if both Gaussians are combined to one Gaussian and are regarded in a coordinate system whose origin is at the center of the combined Gaussian (i.e., the product of the two original ones).

Parameters:

a int - the Cartesian factor in front of the first Cartesian Gaussian function

a int - the Cartesian factor in front of the second Cartesian Gaussian function

diffA double - the difference between the center of the combined Gaussian and the center of the first Gaussian

diffB double - the difference between the center of the combined Gaussian and the center of the second Gaussian

gamma double - the exponent of the combined Gaussian computes as $\alpha + \beta$

9.13 include/FireDeamon/parallel_generic.h File Reference

A header containing template classes and function definitions that allow to perform parallelized computations. `#include <cstdlib>`

```
#include <pthread.h>
#include <vector>
#include <tuple>
#include <stdexcept>
#include <assert.h>
#include <stdio.h>
#include <math.h>
#include <signal.h>
#include <time.h>
#include <unistd.h>
#include <FireDeamon/iterate_over_tuple.h>
#include <FireDeamon/deamon_functors.h>
#include <iostream>
```

Classes

- class PG

  The class PG contains global information required for the parallelized computation.

- class GPSubData< Tout, Tins >

  A templated class that contains all the data to be passed to single threads.

- class GPData< Tout, Tsplit, Tins >

  A templated class that contains all the data to be passed to all threads.

Functions

- void signal_callback_handler (int signum)

  A function that is called whenever a signal is received (e.g., a keyboard interrupt).

- void init_parallel_generic (bool *progress_reports, PG *globals)

  initialize the global data structure (that is used for signal handling and reporting progress)

- template<typename... Ts>
  void do_parallel_generic (void *(*thread_func)(void *), PG *globals, bool progress_reports, int nr_calcs, GPData< Ts...> *data)

  Perform a parallelized compuation.

- void finalize_parallel_generic (bool progress_reports, PG *globals)

finalize everything after the parallel computation. This also transfers output data properly.

Variables

- PG * **pg_global**

9.13.1 Detailed Description

A header containing template classes and function definitions that allow to perform parallelized computations. This is achieved by mapping a function to every data set in a std::vector (with nigh-arbitrary template argument). The type *bool* is not supported as either input nor output type since std::vector<bool> is implemented as a bitfield and not as simply avector of Boolean values.

9.13.2 Function Documentation

9.13.2.1 template<typename... Ts> void do_parallel_generic (void *(*)(void *) *thread_func*, PG * *globals*, bool *progress_reports*, int *nr_calcs*, GPData< Ts...> * *data*) [inline]

Perform a parallelized compuation.

Parameters:

thread_func void *(*thread_func)(void*) - function pointer. This function is mapped to the data.

globals pointer to PG - global data that is, e.g., used for treating keyboard interrupts

progress_reports bool - whether or not progress reports are desired

nr_calcs int - how many computations shall be performed ,i.e., maximum counter for progress reports

data pointer to GPData<Ts...> - the data structure that contains all the data

9.13.2.2 void signal_callback_handler (int *signum*)

A function that is called whenever a signal is received (e.g., a keyboard interrupt). Clean-up of data and thread-handles is also performed.

9.14 include/FireDeamon/set_procname.h File Reference

Function to set the name of the current process. `#include <string>`

Functions

- void set_procname (std::string newname)

  Set the process name.

9.14.1 Detailed Description

Function to set the name of the current process.

9.14.2 Function Documentation

9.14.2.1 void set_procname (std::string *newname*)

Set the process name.

Parameters:

newname std::string - the new name to be used

argv char** - the argunent vector array

Returns:

whether or not setting the name succeeded

9.15 include/FireDeamon/skin_surface_deamon.h File Reference

Create a skin surface around a set of spheres. `#include <vector>`

Functions

- void make_skin_surface (double shrink_factor, std::vector< double > coord_radii_vec, std::vector< int > *ivec, std::vector< double > *dvec, std::vector< double > *nvec, std::vector< int > *length, int nr_refinements)

  Create a skin surface of arbitrary high discretization around a set of spheres.

9.15.1 Detailed Description

Create a skin surface around a set of spheres.

9.15.2 Function Documentation

9.15.2.1 void make_skin_surface (double *shrink_factor*, std::vector< double > *coord_radii_vec*, std::vector< int > * *ivec*, std::vector< double > * *dvec*, std::vector< double > * *nvec*, std::vector< int > * *length*, int *nr_refinements*)

Create a skin surface of arbitrary high discretization around a set of spheres. A definition for skin surfaces can be found here: http://doc.cgal.org/latest/Skin_surface_3/index.html You can imagine a skin surface as a rubber skin contracting around a set of spheres. The degree of contraction can be specified to get a sharper or smoother approximation of the spheres. First, a very weakly discretized surface is generated (a sphere roughly approximated by an octaeder), which can then be further refined by adding a further point in the middle of every edge (for each refinement step). Increasing the number of refinement steps by one more than doubles the memory requirement.

Bug

crashes if *shrink_factor* is ≤ 0 or ≥ 1

Bug

if *nr_refinements* is large (≥ 4 for a system with 8GB RAM), the isosurface cannot be kept in memory but no error is thrown.

Parameters:

shrink_factor double - the shrink factor that defined how "tight" the skin surface shall be A value closer to 1 causes a more accurate reproduction of the union of the spheres.

coord_radii_vec std::vector<double> - a flat vector containing the coordinates and radii For each sphere in the set, this vector contains the three Cartesian coordinates of its center followed by the radius. That means this vector has a length of 4 times the number of spheres in the set.

ivec pointer to std::vector<int> - this flat vector will be filled with triples of indices that specify the facets of the skin surface

dvec pointer to std::vector<double> - this flat vector will be filled with triples of values specifying the Cartesian coordinates of the vertices of the skin surface

nvec pointer to std::vector<double> - this flat vector will be filled with triples of values that specify the normal vectors associated with each vertex

length pointer to std::vector<int> - this flat vector will contain the number of vertices and the number of facets, in that order

nr_refinements int - the number of refinement steps to perform

Index