O_2scl_part - Particles and Nuclei Sub-Library for O_2scl

Version 0.910

Copyright © 2006-2012, Andrew W. Steiner
Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled “License Information”.

Contents

1 O2scl Particles and Nuclei Sub-Library User's Guide

1.1 Feature Overview

O_2scl_part is sub-library for working with particles and nuclei designed to work with O_2scl . It includes

- Several classes for computing the thermodynamic properties of interacting and non-interacting fermions, bosons and nuclei
- Currently available experimental nuclear mass data
- Theoretical nuclear mass models and mass formulas
- Containers holding distributions of nuclei

1.2 Quick Reference to User's Guide

2 Particles

These classes in the library O_2scl_part calculate the thermodynamic properties of interacting and non-interacting quantum and classical particles.

The class part is the basic structure for a particle:

- part::m - mass
- part::g - degeneracy factor (e.g. $2j+1$)
- part::n - number density
- part::ed - energy density
- part::pr - pressure
- part::en - entropy density
- part::ms - effective mass, i.e. m^*
- part::nu - effective chemical potential, i.e. ν
- part::inc_rest_mass - True if the rest mass is included
- part::non_interacting - False if the particle includes interactions

The data members part::ms and part::nu allow one to specify modifications to the mass and the chemical potential due to interactions. This allows one to calculate the properties of particle due to interactions so long as the basic form of the free-particle dispersion relation is unchanged, i.e.

$$\sqrt{k^2+m^2}-\mu \to \sqrt{k^2+m^{*2}}-\nu$$

If the particle is non-interacting, then part::nu and part::ms are sometimes used by O$_2$scl_part functions for temporary storage.

If part::inc_rest_mass is `true` (as is the default in all of the classes except nucleus), then all functions include the rest mass energy density in the energy density, the "mu" functions expect that the rest mass is included in part::mu or part::nu as input and the "density" functions output part::mu or part::nu including the rest mass.

When part::inc_rest_mass is true, antiparticles are implemented by choosing the antiparticle chemical potential to be $-\mu$, and when inc_rest_mass is false, antiparticles are implemented by choosing the chemical potential of the antiparticles to be $-\mu-2m$.

The thermodynamic identity used to compute the pressure for interacting particles is

$$P=-\varepsilon+sT+\nu n$$

where part::nu is used. This way, the particle class doesn't need to know about the structure of the interactions to ensure that the thermodynamic identity is satisfied. Note that in the O$_2$scl_eos library, where in the equations of state the normal thermodynamic identity is used

$$P=-\varepsilon+sT+\mu n$$

Frequently, the interactions which create an effective chemical potential which is different than part::mu thus create extra terms in the pressure and the energy density for the given equation of state.

At zero temperature, fermions and bosons can be treated exactly in the classes fermion and boson. The quark class is a descendant of the fermion class which contains extra data members for the quark condensate and the contribution to the bag constant. The classical classical is a descendant of both fermion and boson and calculates everything in the classical limit.

At finite temperature, there are different classes corresponding to different approaches to computing the integrals over the distribution functions. The approximation scheme from Johns96 is used in eff_boson, eff_fermion, and eff_quark. An exact method employing direct integration of the distribution functions is used in rel_boson and rel_fermion, but these are necessarily quite a bit slower.

The class eff_fermion usually works to within about 1 part in 10^4, but can be as bad as 1 part in 10^2 in some more extreme cases. The default settings for rel_fermion give an accuracy of at least 1 part in 10^6 (and frequently better than this). For rel_fermion, some additional accuracy may be obtained by decreasing the integration tolerances.

The class nonrel_fermion assumes a non-relativistic dispersion relation for fermions. It includes zero-temperature methods and an exact method for finite temperatures. The non-relativistic integrands are much simpler and nonrel_fermion uses the appropriate GSL functions (which are nearly exact) to compute them.

Units:

Factors of $\hbar, c$ and k_B have been removed everywhere, so that mass, energy, and temperature all have the same units. Number and entropy densities have units of mass cubed (or energy cubed). This particle classes can be used with any system of units which is based on powers of one unit, i.e. $[n]=[T]^3=[m]^3=[P]^{3/4}=[\varepsilon]^{3/4}$, etc.

Derivative information:

Sometimes it is useful to know derivatives like ds/dT in addition to the energy and pressure. There are three classes which compute these derivatives for fermions and classical particles. The class sn_classical handles the nondegenerate limit, sn_fermion handles fermions and sn_nr_fermion handles nonrelativistic fermions. These classes compute the derivatives

$$\left(\frac{dn}{d\mu}\right)_T, \quad \left(\frac{dn}{dT}\right)_\mu, \quad \text{and} \quad \left(\frac{ds}{dT}\right)_\mu .$$

All other first derivatives of the thermodynamic functions can be written in terms of these three. To see how to compute the specific heat, for example, see the discussion in the documentation of part_deriv.

3 Atomic Nuclei

Nuclei

Atomic nuclei, class nucleus, are implemented as descendants of classical. This class sets the value of nucleus::inc_rest_mass to false by default.

Nuclear mass formulas are given as children of nuclear_mass. The class ame_mass provides the experimental data from Audi95 or Audi03, the class mnmsk_mass provides the mass formula from Moller95, and the class hfb_mass provides the mass formula from Goriely02, Samyn04, or Goriely07. A simple semi-empirical mass formula is given in semi_empirical_mass and this can be fit to experimentally measured masses using mass_fit.

The class nuclear_dist provides an generic base class for a collection of several nuclei with an STL-like iterator. There are two implementations of this base class, simple_dist which provides a simple distribution and full_dist which enumerates all the nuclei for a given mass formula.

4 Example Source Code

4.1 Example list

- Particle example
- Nuclear mass fit example

4.2 Particle example

```
/* Example: ex_part.cpp
   -------------------------------------------------------------------
*/

#include <cmath>
#include <o2scl/test_mgr.h>
#include <o2scl/constants.h>
#include <o2scl/eff_fermion.h>
#include <o2scl/rel_fermion.h>
#include <o2scl/classical.h>

using namespace std;
using namespace o2scl;
using namespace o2scl_const;

int main(void) {
  test_mgr t;
  t.set_output_level(1);

  // Compare the method from rel_fermion to the more approximate
  // scheme used in eff_fermion. We work in units of inverse Fermis,
  // so that energy density is fm^{-4}. We also use a classical
  // particle, to compare to the nondegenerate approximation.
  eff_fermion eff;
  rel_fermion relf;
  classical cla;

  fermion e(o2scl_fm::mass_electron,2.0);
  fermion e2(o2scl_fm::mass_electron,2.0);
  fermion e3(o2scl_fm::mass_electron,2.0);

  // Compute the pressure at a density of 0.0001 fm^{-3} and a
  // temperature of 10 MeV. At these temperatures, the electrons are
  // non-degenerate, and Boltzmann statistics nearly applies.
  e.n=0.0001;
  eff.calc_density(e,10.0/hc_mev_fm);
```

```
  e2.n=0.0001;
  relf.calc_density(e2,10.0/hc_mev_fm);
  e3.n=0.0001;
  cla.calc_density(e3,10.0/hc_mev_fm);

  cout << e.pr << " " << e2.pr << " " << e3.pr << " "
       << e.n*10.0/hc_mev_fm << endl;

  // Test
  t.test_rel(e.pr,e2.pr,1.0e-2,"EFF vs. exact");
  t.test_rel(e2.pr,e3.pr,4.0e-1,"classical vs. exact");
  t.test_rel(e.n*10.0/hc_mev_fm,e3.pr,1.0e-1,"classical vs. ideal gas law");

  // Compute the pressure at a density of 0.1 fm^{-3} and a
  // temperature of 1 MeV. At these temperatures, the electrons are
  // strongly degenerate
  e.n=0.0001;
  eff.calc_density(e,10.0/hc_mev_fm);
  e2.n=0.0001;
  relf.calc_density(e2,10.0/hc_mev_fm);
  cout << e.pr << " " << e2.pr << endl;

  // Test
  t.test_rel(e.pr,e2.pr,1.0e-2,"EFF vs. exact");

  // Now add the contribution to the pressure from positrons using the
  // implmentation of part::pair_density()
  e.n=0.0001;
  eff.pair_density(e,10.0/hc_mev_fm);
  e2.n=0.0001;
  relf.pair_density(e2,10.0/hc_mev_fm);
  cout << e.pr << " " << e2.pr << endl;

  // Test
  t.test_rel(e.pr,e2.pr,1.0e-2,"EFF vs. exact");

  t.report();
  return 0;
}
// End of example
```

4.3 Nuclear mass fit example

```
/* Example: ex_mass_fit.cpp
   -------------------------------------------------------------------
*/

#include <iostream>
#include <o2scl/test_mgr.h>
#include <o2scl/mass_fit.h>
#ifdef O2SCL_HDF_IN_EXAMPLES
#include <o2scl/hdf_file.h>
#include <o2scl/hdf_nucmass_io.h>
#endif

using namespace std;
using namespace o2scl;
using namespace o2scl_const;
using namespace o2scl_fm;


int main(void) {
  test_mgr t;
  t.set_output_level(1);

  cout.setf(ios::scientific);
```

```
#ifdef O2SCL_HDF_IN_EXAMPLES

  // The RMS deviation of the fit
  double res;
  // The mass formula to be fitted
  semi_empirical_mass sem;
  // The fitting class
  mass_fit mf;

  // Load the experimental data
  ame_mass ame;
  o2scl_hdf::ame_load(ame,"");
  mf.set_exp_mass(ame);

  // Perform the fit
  mf.fit(sem,res);

  // Output the results
  cout << sem.B << " " << sem.Sv << " " << sem.Ss << " "
       << sem.Ec << " " << sem.Epair << endl;
  cout << res << endl;
  t.test_gen(res<4.0,"Successful fit.");

#else
  cout << "No fitting was performed because O2scl appears not to have been "
       << "compiled with HDF support." << endl;
#endif

  t.report();
  return 0;
}
// End of example
```

5 Bibliography

Some of the references which contain links should direct you to the work referred to directly through dx.doi.org.

Audi95: G. Audi and A. H. Wapstra, Nucl. Phys. A **595** (1995) 409-480.

Audi03: G.Audi, A. H. Wapstra and C. Thibault , Nucl. Phys. A **729** (2003) 337.

Callen H. B. Callen, "Thermodynamics and an Introduction to Thermostatistics.", 2nd edition.

Eggleton73: P. P. Eggleton, J. Faulkner, and B. P. Flannery, Astron. and Astrophys. **23** (1973) 325.

Goriely02: S. Goriely, M. Samyn, P.-H. Heenen, J. M. Pearson, and F. Tondeur, Phys. Rev. C **66** (2002) 024326.

Goriely07: S. Goriely, M. Samyn, and J. M. Pearson, Phys. Rev. C **75** (2007) 064312.

Johns96: Johns, P.J. Ellis, and J.M. Lattimer, Astrophys. J. **473** (1996) 1020.

Landau: L. D. Landau, "Statistical Physics: Part 1", 3rd Edition.

Moller95: P. Moller, J.R. Nix, W.D. Myers, and W.J. Switecki, At. Data Nucl. Data Tables **59** (1995) 185.

Moller97: P. Moller, J.R. Nix, and K.-L. Kratz At. Data Nucl. Data Tables **66** (1997) 131.

Samyn04: M. Samyn, S. Goriely, M. Bender and J. M. Pearson, Phys. Rev. C **70** (2004) 044309.

6 Ideas for Future Development

Class ame_mass

There are definitions of the atomic mass unit and other constants that are defined by the 1995 and 2003 atomic mass evaluations which are not used at present. These could be implemented by making a separate copy of the neutron and proton masses inside

this class.

Create a caching and more intelligent search system for the table. The table is sorted by A and then N, so we could probably just copy the search routine from mnmsk_mass, which is sorted by Z and then N.

Allow the user to select only massses which are not estimated.

Class classical

Write a calc_density_zerot() function for completeness?

Class eff_fermion

Use bracketing to speed up one-dimensional root finding.

Class fermion_eval_thermo

Create a Chebyshev approximation for inverting the the Fermi functions for massless_calc_density() functions?

Global fermion_eval_thermo::calibrate (fermion &f, int verbose=0, std::string fname="")

Also calibrate massless fermions?

Convert into separate class?

Global fermion_eval_thermo::massless_pair_density (fermion &f, double temper)

This could be improved by including more terms in the expansions.

Class fermion_zerot

Use hypot() and other more accurate functions for the analytic expressions for the zero temperature integrals. [Progress has been made, but there are probably other functions which may break down for small but finite masses and temperatures]

Class frdm_mass

Add microscopic part.

Class mass_fit

Convert to a real fit with errors and covariance, etc.

Class nonrel_fermion

This could be improved by performing a Chebyshev approximation (for example) to invert the density integral so that we don't need to use a solver.

Class nuclear_mass

Find a way to include the electron binding energy contribution, possibly with the help of a theoretical model.

Class nuclear_mass_fit

This shouldn't be a child of nuclear_mass_cont?

Global nuclear_mass_info::parse_elstring (std::string ela, int &Z, int &N, int &A)

Warn about malformed combinations like Carbon-5

Right now, `n4` is interpreted incorrectly as Nitrogen-4, rather than the tetraneutron.

Class rel_fermion

The expressions which appear in in the integrand functions density_fun(), etc. could likely be improved, especially in the case where inc_rest_mass=false. There should not be a need to check if `ret` is finite.

It appears this doesn't compute the uncertainty in the chemical potential or density with calc_density(). This could be fixed.

I'd like to change the lower limit on the entropy integration, but the value in the code at the moment (stored in `ll`) makes bm_part2.cpp worse.

pair_mu() should set the antiparticle integrators as done in sn_fermion.

Class simple_dist

Make the vector constructor into a template so it accepts any type. Do the same for set_dist().

Class sn_fermion

It might be worth coding up direct differentiation, or differentiating the eff results, as these may succeed more generally.

This class will have difficulty with extremely degenerate or extremely non-degnerate systems. Fix this.

Create a more intelligent method for dealing with bad initial guesses for the chemical potential in calc_density().

7 Todo List

Class eff_boson

Better documentation (see eff_fermion)

Remove the 'meth2' stuff

Remove static variables fix_density and stat_temper

Remove exit() calls

Class eff_fermion

There's still def_err_hnd.set_mode(0) in the testing code, probably because the solver has a hard time for extreme values.

Global eff_fermion::calc_mu (fermion &f, double temper)

Should see if the function actually works if $(\mu - m)/T = -199$.

Class frdm_mass

Fix pairing energy and double vs. int

Document drip_binding_energy(), etc.

Decide on number of fit parameters (10 or 12?) or let the user decide

Document the protected variables

Set the neutron and proton masses and hbarc to Moller et al.'s values

Class nonrel_fermion

Check behaviour of calc_density() at zero density, and compare with that from eff_fermion, rel_fermion, and classical.

Implement pair_density() and pair_mu().

Make sure to test with non-interacting equal to true or false, and document whether or not it works with both inc_rest_mass equal to true or false

Class rel_boson

Testing not completely finished.

Class sn_classical

This does not work with inc_rest_mass=true

Class sn_fermion

This needs to be corrected to calculate $\sqrt{k^2+m^{*2}}-m$ gracefully when $m^* \approx m << k$.

Call error handler if inc_rest_mass is true or update to properly treat the case when inc_rest_mass is true.

8 Data Structure Documentation

8.1 ame_entry Class Reference

Atomic mass entry structure.

`#include <nuclear_mass.h>`

8.1.1 Detailed Description

Atomic mass entry data object for ame_mass.

In cases where the decimal point in the original table was replaced with a #, the associated accuracy field is set to estimated. In cases where the original table contained a asterisk to indicate a value was not calculable, the accuracy field is set to not_calculable and the value is zero. If O_2scl internally computed the value instead of reading it from the original table, the accuracy field is set to intl_computed. In cases where either orig or bdmode in the original table was blank, the string is set to `"blank"`.

Binding energies are defined with a positive sign, so that lead has a binding energy of +8 MeV.

Definition at line 457 of file nuclear_mass.h.

Data Fields

- int NMZ
  N-Z.
- int N
  Neutron number.
- int Z
  Proton number.
- int A
  Atomic number.
- char el [4]
  Element name.
- char orig [5]
  Data origin.
- double mass
  Mass excess (in keV)
- double dmass
  Mass excess uncertainty (in keV)
- int mass_acc
  Mass accuracy flag.
- double be
  Binding energy (in keV, given in the '95 data)
- double dbe
  Binding energy uncertainty (in keV, given in the '95 data)
- int be_acc
  Binding energy accuracy flag.
- double beoa
  Binding energy / A (in keV, given in the '03 data)
- double dbeoa
  Binding energy / A uncertainty (in keV, given in the '03 data)
- int beoa_acc
  Binding energy / A accuracy flag.
- char bdmode [3]
  Beta decay mode.
- double bde
  Beta-decay energy (in keV)
- double dbde
  Beta-decay energy uncertainty (in keV)
- int bde_acc
  Beta-decay energy accuracy flag.
- int A2
  ?
- double amass
  Atomic mass (in keV)
- double damass
  Atomic mass uncertainty (in keV)
- int amass_acc
  Atomic mass accuracy flag.

Static Public Attributes

Accuracy modes

- static const int measured = 0
  Measured value from source data.

- static const int estimated = 1
  Value estimated in source data.
- static const int not_calculable = 2
  Value listed in data as not calculable.
- static const int intl_computed = 3
  Value computed by O_2scl .

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

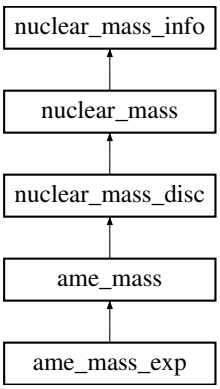
- nuclear_mass.h

8.2 ame_mass Class Reference

Mass formula from the Atomic Mass Evaluation (2005 and 1993)

`#include <nuclear_mass.h>`

Inheritance diagram for ame_mass:



8.2.1 Detailed Description

Note

This class requires data stored in an HDF file and thus requires HDF support for normal usage.

This class provides an interface to the atomic mass table using data from Audi95 and Audi03.

There are four data sets, selected by the specification of the `version` string in the constructor.

- "95rmd" - "Recommended" data from Audi95 (ame95rmd.o2)
- "95exp" - "Experimental" data from Audi95 (ame95exp.o2)
- "03round" - "Rounded" data from Audi03 (ame03round.o2)
- "03" - Data from Audi03 (default) (ame03.o2)

If any string other than these four is used, the default data is loaded. If the constructor cannot find the data file (e.g. because of a broken installation), then ame::is_loaded() returns false.

The 1995 data provided the binding energy stored in ame_entry::be and ame_entry::dbe, while the 2003 data provided the binding energy divided by the atomic number stored in ame_entry::beoa and ame_entry::dbeoa. When the 1995 data is used ame_entry::beoa and ame_entry::dbeoa are calculated automatically, and when the 2003 data is used ame_entry::be and ame_entry::dbe are

calculated automatically. To indicate that O_2scl has automatically calculated a value in this way, the associated accuracy field is set to ame_entry::intl_computed.

Note that all uncertainties are 1 sigma uncertainties.

The functions mass_excess() and nuclear_mass::mass_excess_d() directly return the value from the Audi et al. data. For consistency, the functions nuclear_mass::binding_energy(), nuclear_mass::binding_energy_d(), nuclear_mass::total_mass(), and nuclear_mass::total_mass_d() return values which are automatically computed with the O_2scl values for the neutron and proton mass in **o2scl_fm** (which are obtained from CODATA). In order to obtain the value of the binding energy as reported in the original Audi et al. data instead of the value computed from the mass excess, you can use the function get_ZN(), and access the corresponding entry from the Audi et al. data directly.

See also the documentation for the class structure for each table entry in ame_entry.

Idea for Future There are definitions of the atomic mass unit and other constants that are defined by the 1995 and 2003 atomic mass evaluations which are not used at present. These could be implemented by making a separate copy of the neutron and proton masses inside this class.

Idea for Future Create a caching and more intelligent search system for the table. The table is sorted by A and then N, so we could probably just copy the search routine from mnmsk_mass, which is sorted by Z and then N.

Idea for Future Allow the user to select only massses which are not estimated.

Definition at line 609 of file nuclear_mass.h.

Public Member Functions

- ame_mass ()
  Create a collection specified by `version`.
- virtual const char * type ()
  Return the type, `"ame_mass"`.
- virtual bool is_included (int Z, int N)
  Return false if the mass formula does not include specified nucleus.
- virtual double mass_excess (int Z, int N)
  Given `Z` *and* `N`*, return the mass excess in MeV.*
- ame_entry get_ZN (int l_Z, int l_N)
  Get element with Z=l_Z and N=l_N (e.g. 82,126).
- ame_entry get_ZA (int l_Z, int l_A)
  Get element with Z=l_Z and A=l_A (e.g. 82,208).
- ame_entry get_elA (std::string l_el, int l_A)
  Get element with name l_el and A=l_A (e.g. "Pb",208).
- ame_entry get (std::string nucleus)
  Get element with string (e.g. "Pb208")
- bool is_loaded ()
  Returns true if data has been loaded.
- int set_data (int n_mass, ame_entry *m, std::string ref)
  Set data.
- int get_nentries ()
  Return number of entries.

Protected Attributes

- int n
  The number of entries (about 3000).
- std::string reference
  The reference for the original data.
- ame_entry * mass
  The array containing the mass data of length ame::n.

8.2.2 Member Function Documentation

8.2.2.1 int ame_mass::set_data (int *n_mass*, ame_entry * *m*, std::string *ref*)

This function is used by the HDF I/O routines.

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

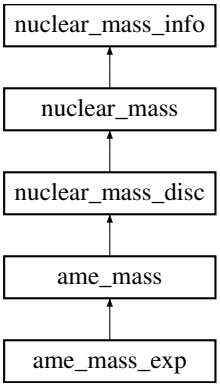
- nuclear_mass.h

8.3 ame_mass_exp Class Reference

The experimental values from Moller, Nix, Myers and Swiatecki.

`#include <nuclear_mass.h>`

Inheritance diagram for ame_mass_exp:



8.3.1 Detailed Description

Note

This class requires data stored in an HDF file and thus requires HDF support for normal usage.

Definition at line 674 of file nuclear_mass.h.

Public Member Functions

- virtual bool is_included (int Z, int N)
  Return false if the mass formula does not include specified nucleus.

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

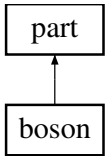
- nuclear_mass.h

8.4 boson Class Reference

Boson class.

`#include <boson.h>`

Inheritance diagram for boson:



8.4.1 Detailed Description

For bosons:

- if either nu or mu is greater than ms, then they are taken to be equal to ms
- All contributions from any type of condensate are ignored.

This Mathematica notebook contains the series expansions for the bosonic integrals. functions.

```
doc/o2scl/extras/boson.nb
doc/o2scl/extras/boson.pdf
```

Definition at line 66 of file boson.h.

Public Member Functions

- boson (double m=0.0, double g=0.0)
  Create a boson with mass m and degeneracy g.
- virtual void massless_calc_mu (double temper)
  Calculate properties of massless bosons.
- virtual const char * type ()
  Return string denoting type ("boson")

Data Fields

- double co
  The condensate.

8.4.2 Member Function Documentation

8.4.2.1 virtual void boson::massless_calc_mu (double *temper*) [virtual]

The expressions used are exact. The chemical potentials are ignored and the scalar density is set to zero

8.4.3 Field Documentation

8.4.3.1 double boson::co

The condensate variable is mostly ignored by class boson and its descendants, and is provided for user storage.

Definition at line 75 of file boson.h.

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

- boson.h

8.5 classical Class Reference

Classical particle class.

`#include <classical.h>`

8.5.1 Detailed Description

Idea for Future Write a calc_density_zerot() function for completeness?

Definition at line 44 of file classical.h.

Public Member Functions

- classical ()
  Create a classical particle with mass m and degeneracy g.
- virtual void calc_mu (part &p, double temper)
  Calculate properties as function of chemical potential.
- virtual void calc_density (part &p, double temper)
  Calculate properties as function of density.
- virtual const char * type ()
  Return string denoting type ("classical")

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

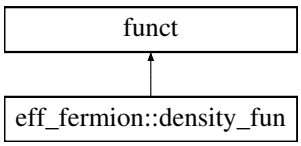
- classical.h

8.6 eff_fermion::density_fun Class Reference

Define the function which solves for the chemical potential given the density [protected subclass of eff_fermion].

`#include <eff_fermion.h>`

Inheritance diagram for eff_fermion::density_fun:



8.6.1 Detailed Description

Definition at line 227 of file eff_fermion.h.

Public Member Functions

- **density_fun** (eff_fermion &ef, fermion &f, double T)
- double operator() (double x)
  Fix density for eff_fermion::calc_density()

Protected Attributes

- eff_fermion & **ef_**
- fermion & **f_**
- double **T_**

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

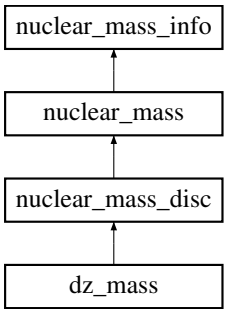
- eff_fermion.h

8.7 dz_mass Class Reference

Duflo-Zuker mass formula.

`#include <nuclear_mass.h>`

Inheritance diagram for dz_mass:



8.7.1 Detailed Description

Definition at line 1129 of file nuclear_mass.h.

Public Member Functions

- dz_mass (std::string model="96", bool external=false)
  Create a new mass formula object.
- virtual bool is_included (int Z, int N)
  Return false if the mass formula does not include specified nucleus.
- virtual double mass_excess (int Z, int N)
  Given `Z` and `N`, return the mass excess in MeV.
- bool is_loaded ()
  Verify that the constructor properly loaded the table.
- virtual const char * type ()
  Return the type, `"dz_mass"`.
- int get_nentries ()
  Return number of entries.

Protected Attributes

- std::string reference
  The reference for the original data.
- **table** data
  Table containing the data.

- int last
  The last table index for caching.
- int n
  The total number of entries.

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

- nuclear_mass.h

8.8 eff_boson Class Reference

Boson class from fitting method.

`#include <eff_boson.h>`

8.8.1 Detailed Description

Todo Better documentation (see eff_fermion)

Remove the 'meth2' stuff

Remove static variables fix_density and stat_temper

Remove exit() calls

Definition at line 52 of file eff_boson.h.

Public Member Functions

- eff_boson ()
  Create a boson with mass m and degeneracy g.
- int load_coefficients (int ctype)
  Load coefficients for finite-temperature approximation.
- virtual void calc_mu (boson &b, double temper)
  Calculate thermodynamic properties as function of chemical potential.
- virtual void calc_density (boson &b, double temper)
  Calculate thermodynamic properties as function of density.
- virtual void pair_mu (boson &b, double temper)
  Calculate thermodynamic properties with antiparticles as function of chemical potential.
- virtual void pair_density (boson &b, double temper)
  Calculate thermodynamic properties with antiparticles as function of density.
- void set_psi_root (**root**< **funct** > &rp)
  Set the solver for use in calculating ψ.
- void set_density_mroot (**mroot**< **mm_funct**<> > &rp)
  Set the solver for use in calculating the chemical potential from the density.
- void set_meth2_root (**root**< **funct** > &rp)
  Set the solver for use in calculating the chemical potential from the density (meth2=true)
- virtual const char * **type** ()

Data Fields

- **gsl_mroot_hybrids**< **mm_funct**<> > def_density_mroot
  The default solver for calc_density() and pair_density()
- **cern_mroot_root**< **funct** > def_psi_root
  The default solver for ψ.
- **cern_mroot_root**< **funct** > def_meth2_root
  The default solver for calc_density() and pair_density()

Static Public Attributes

- static const int cf_boselat3 = 1
  A set of coefficients from Jim Lattimer.
- static const int cf_bosejel21 = 2
  A set of coefficients from Johns96.
- static const int cf_bosejel22 = 3
  A set of coefficients from Johns96.
- static const int cf_bosejel34 = 4
  A set of coefficients from Johns96.
- static const int cf_bosejel34cons = 5
  The set of coefficients from Johns96 which retains better thermodynamic consistency.

Protected Member Functions

- double solve_fun (double x, double &psi)
  The function which solves for h from ψ.
- int density_fun (size_t nv, const **ovector_base** &x, **ovector_base** &y)
  Fix density for calc_density()
- int pair_density_fun (size_t nv, const **ovector_base** &x, **ovector_base** &y)
  Fix density for pair_density()

Protected Attributes

- **umatrix** Pmnb
  The coefficients.
- int sizem
  The number of coefficient rows.
- int sizen
  The number of coefficient columns.
- double parma
  The parameter, a.
- double fix_density
  Temporary storage.
- boson * bp
  Desc.
- double T
  Desc.
- **mroot**< **mm_funct**<> > * density_mroot
  The solver for calc_density()
- **root**< **funct** > * psi_root
  The solver to compute h from ψ.
- **root**< **funct** > * meth2_root
  The solver for calc_density()

8.8.2 Member Function Documentation

8.8.2.1 int eff_boson::load_coefficients (int *ctype*)

Presently acceptable values of `fn` are: `boselat3` from Lattimer's notes `bosejel21`, `bosejel22`, `bosejel34`, and `bosejel34cons` from Johns96.

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

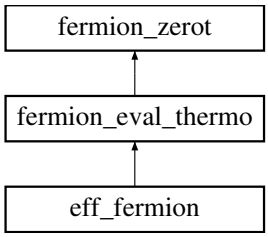
- eff_boson.h

8.9 eff_fermion Class Reference

Fermion class from fitting method.

`#include <eff_fermion.h>`

Inheritance diagram for eff_fermion:



8.9.1 Detailed Description

Based on the fitting method of Johns96 which is an update of the method from Eggleton73 . This method is approximate, but very fast. For a more accurate (but slower) method, use rel_fermion.

Given the chemical potential and the temperature the functions calc_mu() and pair_mu() work by solving the equation (c.f. Eq. 15 in Johns96)

$$\psi = 2\sqrt{1+f/a}+\log\left(\frac{\sqrt{1+f/a}-1}{\sqrt{1+f/a}+1}\right)$$

for f given $\psi=(\mu-m)/T$. If $f/a<10^{-10}$, then the alternative expression

$$\psi = 2\,(1+f/(2a))+\log\left[\frac{f/(2a)}{(1+f/(2a))}\right]$$

is used. The pressure, energy density, and entropy, are determined as polynomials in f with a set of precomputed coefficients as done in Johns96 .

If ψ is too small (less than about -200), the above procedure fails. To handle this, this class uses the classical result if $\psi <$ min_psi, where min_psi defaults to -200.

When the density and temperature is given instead (calc_density() and pair_density()), then there are two ways to proceed.

- Use the density to solve for f .
- Use the density to solve for the chemical potential.

Because the density is a complicated polynomial in f, the former procedure does not work very well even though it might be less time consuming. In this class, the density is solved for the effective chemical potential instead. The initial guess is just taken from the present value of part::nu .

Note

It is important to note that the coefficients are static and apply to all objects of type eff_fermion.
The coefficients are stored in static data, and there is a small possibility of a memory leak in a multithreaded program if more than one instance of eff_fermion tries to initialize these coefficients in the constructor. This can easily be avoided by ensuring that the static coefficients are initialized by a single thread beforehand.

Todo There's still def_err_hnd.set_mode(0) in the testing code, probably because the solver has a hard time for extreme values.

Idea for Future Use bracketing to speed up one-dimensional root finding.

Definition at line 102 of file eff_fermion.h.

Data Structures

- class density_fun
  Define the function which solves for the chemical potential given the density [protected subclass of eff_fermion].
- class pair_density_fun
  Define the function which solves for the chemical potential given the density of particles and antiparticles [protected subclass of eff_fermion].

Public Member Functions

- eff_fermion ()
  Create a fermion with mass `mass` and degeneracy `dof`.
- virtual void calc_mu (fermion &f, double temper)
  Calculate thermodynamic properties as function of chemical potential.
- virtual void calc_density (fermion &f, double temper)
  Calculate thermodynamic properties as function of density.
- virtual void pair_mu (fermion &f, double temper)
  Calculate thermodynamic properties with antiparticles as function of chemical potential.
- virtual void pair_density (fermion &f, double temper)
  Calculate thermodynamic properties with antiparticles as function of density.
- int set_psi_root (**root**< **funct** > &rp)
  Set the solver for use in calculating ψ.
- int set_density_root (**root**< **funct** > &rp)
  Set the solver for use in calculating the chemical potential from the density.
- virtual const char * type ()
  Return string denoting type ("eff_fermion")

Data Fields

- double tlimit
  If the temperature is less than `tlimit` then the zero-temperature functions are used (default 0).
- **cern_mroot_root**< **funct** > def_psi_root
  The default solver for ψ.
- **cern_mroot_root**< **funct** > def_density_root
  The default solver for calc_density() and pair_density()
- double min_psi
  The minimum value of ψ (default -200)

Protected Member Functions

- double solve_fun (double x, double &psi)
  The function which solves for f from ψ.

Protected Attributes

- **umatrix** Pmnf
  The matrix of coefficients.
- double parma
  The parameter a.
- int sizem
  The array row size.
- int sizen
  The array column size.
- **root**< **funct** > * psi_root
  The solver for ψ.
- **root**< **funct** > * density_root
  The other solver for calc_density()

Coefficients for finite-temperature approximation

- static const int cf_fermilat3 = 1
  A set of coefficients from Jim Lattimer.
- static const int cf_fermijel2 = 2
  The smaller set of coefficients from Johns96.
- static const int cf_fermijel3 = 3
  The larger set of coefficients from Johns96.
- static const int cf_fermijel3cons = 4
  The set of coefficients from Johns96 which retains better thermodynamic consistency.
- void load_coefficients (int ctype)
  Load coefficients.

8.9.2 Member Function Documentation

8.9.2.1 void eff_fermion::load_coefficients (int *ctype*)

The argument `ctype` Should be one of the constants below.

8.9.2.2 virtual void eff_fermion::calc_mu (fermion & *f*, double *temper*) [virtual]

If the quantity $(\mu - m)/T$ (or $(\nu - m^*)/T$ in the case of interacting particles) is less than -200, then this quietly sets the density, the scalar density, the energy density, the pressure and the entropy to zero and exits.

Todo Should see if the function actually works if $(\mu - m)/T = -199$.

Implements fermion_eval_thermo.

8.9.2.3 virtual void eff_fermion::calc_density (fermion & *f*, double *temper*) [virtual]

Warning

This function needs a guess for the chemical potential, and will fail if that guess is not sufficiently accurate.

Implements fermion_eval_thermo.

8.9.2.4 virtual void eff_fermion::pair_mu (fermion & *f*, double *temper*) [virtual]

Warning

This function needs a guess for the chemical potential, and will fail if that guess is not sufficiently accurate.

Implements fermion_eval_thermo.

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

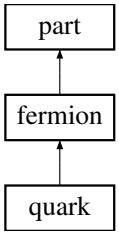
- eff_fermion.h

8.10 fermion Class Reference

Fermion class.

`#include <fermion.h>`

Inheritance diagram for fermion:



8.10.1 Detailed Description

This class adds two member data variables, kf and del, for the Fermi momentum and the gap, respectively.

Definition at line 49 of file fermion.h.

Public Member Functions

- fermion (double mass=0, double dof=0)
  Create a fermion with mass `mass` *and degeneracy* `dof`*.*
- virtual const char * type ()
  Return string denoting type ("fermion")

Data Fields

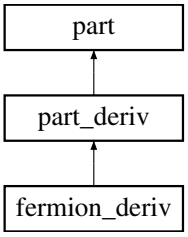
- double kf
  Fermi momentum.
- double del
  Gap.

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

- fermion.h

8.11 fermion_deriv Class Reference

Inheritance diagram for fermion_deriv:



8.11.1 Detailed Description

Definition at line 279 of file deriv_part.h.

Public Member Functions

- **fermion_deriv** (double mass=0.0, double dof=0.0)

Data Fields

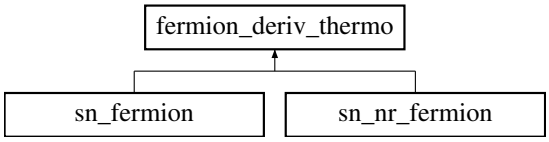
- double kf
  Fermi momentum.

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

- deriv_part.h

8.12 fermion_deriv_thermo Class Reference

Inheritance diagram for fermion_deriv_thermo:



8.12.1 Detailed Description

Definition at line 291 of file deriv_part.h.

Public Member Functions

- virtual void calc_mu (fermion_deriv &f, double temper)=0
  Calculate properties as function of chemical potential.
- virtual void calc_density (fermion_deriv &f, double temper)=0
  Calculate properties as function of density.
- virtual void pair_mu (fermion_deriv &f, double temper)=0
  Calculate properties with antiparticles as function of chemical potential.
- virtual void pair_density (fermion_deriv &f, double temper)=0
  Calculate properties with antiparticles as function of density.
- virtual void nu_from_n (fermion_deriv &f, double temper)=0
  Calculate effective chemical potential from density.

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

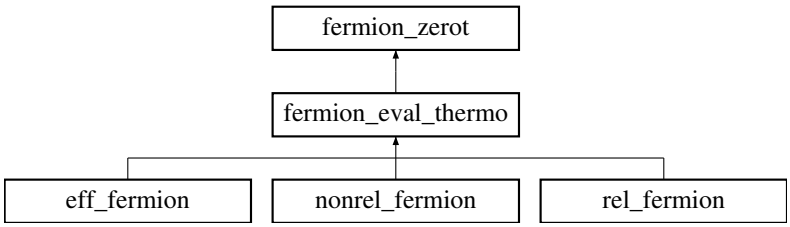
- deriv_part.h

8.13 fermion_eval_thermo Class Reference

Fermion with finite-temperature thermodynamics [abstract base].

`#include <fermion.h>`

Inheritance diagram for fermion_eval_thermo:



8.13.1 Detailed Description

This is an abstract base for the computation of finite-temperature fermionic statistics. Different children (e.g. eff_fermion and rel_fermion) use different techniques to computing the momentum integrations.

Because massless fermions at finite temperature are much simpler, there are separate member functions included in this class to handle them. The functions massless_calc_density() and massless_calc_mu() compute the thermodynamics of massless fermions at finite temperature given the density or the chemical potentials. The functions massless_pair_density() and massless_pair_mu() perform the same task, but automatically include antiparticles.

The function massless_calc_density() uses a **root** object to solve for the chemical potential as a function of the density. The default is an object of type **cern_mroot_root**. The function massless_pair_density() does not need to use the **root** object because of the simplification afforded by the inclusion of antiparticles.

Idea for Future Create a Chebyshev approximation for inverting the the Fermi functions for massless_calc_density() functions?

This Mathematica notebook contains the derivations of related series expansions and some algebra for the massless_pair() functions.

```
doc/o2scl/extras/fermion.nb
doc/o2scl/extras/fermion.pdf
```

Definition at line 181 of file fermion.h.

Public Member Functions

- virtual void calc_mu (fermion &f, double temper)=0
  Calculate properties as function of chemical potential.
- virtual void calc_density (fermion &f, double temper)=0
  Calculate properties as function of density.
- virtual void pair_mu (fermion &f, double temper)=0
  Calculate properties with antiparticles as function of chemical potential.
- virtual void pair_density (fermion &f, double temper)=0
  Calculate properties with antiparticles as function of density.
- void set_massless_root (**root**< **funct** > &rp)
  Set the solver for use in massless_calc_density()
- virtual const char * type ()
  Return string denoting type ("fermion_eval_thermo")
- virtual double calibrate (fermion &f, int verbose=0, std::string fname="")
  Test the thermodynamics of calc_density() and calc_mu()

Massless fermions

- virtual void massless_calc_mu (fermion &f, double temper)
  Finite temperature massless fermions.
- virtual void massless_calc_density (fermion &f, double temper)
  Finite temperature massless fermions.

- virtual void massless_pair_mu (fermion &f, double temper)
  Finite temperature massless fermions and antifermions.
- virtual void massless_pair_density (fermion &f, double temper)
  Finite temperature massless fermions and antifermions.

Data Fields

- **cern_mroot_root**< **funct** > def_massless_root
  The default solver for massless_calc_density()

8.13.2 Member Function Documentation

8.13.2.1 virtual void fermion_eval_thermo::massless_pair_density (fermion & *f*, double *temper*) [virtual]

In the cases $n^3 >> T$ and $T >> n^3$, expansions are used instead of the exact formulas to avoid loss of precision.

In particular, using the parameter

$$\alpha = \frac{g^2 \pi^2 T^6}{243 n^2}$$

and defining the expression

$$\mathrm{cbt} = \alpha^{-1/6} \left(-1 + \sqrt{1+\alpha} \right)^{1/3}$$

we can write the chemical potential as

$$\mu = \frac{\pi T}{\sqrt{3}} \left(\frac{1}{\mathrm{cbt}} - \mathrm{cbt} \right)$$

These expressions, however, do not work well when α is very large or very small, so series expansions are used whenever $\alpha > 10^4$ or $\alpha < 3 \times 10^{-4}$. For small α,

$$\left(\frac{1}{\mathrm{cbt}} - \mathrm{cbt} \right) \approx \frac{2^{1/3}}{\alpha^{1/6}} - \frac{\alpha^{1/6}}{2^{1/3}} + \frac{\alpha^{5/6}}{6\, 2^{2/3}} + \frac{\alpha^{7/6}}{12\, 2^{1/3}} - \frac{\alpha^{11/6}}{18\, 2^{2/3}} - \frac{5\alpha^{13/6}}{144\, 2^{1/3}} + \frac{77\alpha^{17/6}}{2592\, 2^{2/3}}$$

and for large α,

$$\left(\frac{1}{\mathrm{cbt}} - \mathrm{cbt} \right) \approx \frac{2}{3}\sqrt{\frac{1}{\alpha}} - \frac{8}{81}\left(\frac{1}{\alpha}\right)^{3/2} + \frac{32}{729}\left(\frac{1}{\alpha}\right)^{5/2}$$

This approach works to within about 1 part in 10^{12}, and is tested in `fermion_ts.cpp`.

Idea for Future This could be improved by including more terms in the expansions.

8.13.2.2 virtual double fermion_eval_thermo::calibrate (fermion & *f*, int *verbose* = `0`, std::string *fname* = `" "`) [virtual]

This compares the approximation to the exact results over a grid with $T = \{10^{-2}, 1, 10^2\}$, $\log_{10}(m/T) = \{-3,-2,-1,0,1,2,3\}$, and $\log_{10}\psi = \{-3,-2,-1,0,1\}$, where $\psi \equiv (\mu - m)/T$ using calc_density() and calc_mu(), with both inc_rest_mass taking both values `true` and `false`.

The `verbose` parameter controls the amount of output, and `fname` is the filename for the file `fermion_cal.dat`.

Idea for Future Also calibrate massless fermions?

Idea for Future Convert into separate class?

8.13.3 Field Documentation

8.13.3.1 cern_mroot_root<funct> fermion_eval_thermo::def_massless_root

We default to a solver of type **cern_mroot_root** here since we don't have a bracket or a derivative.

Definition at line 284 of file fermion.h.

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

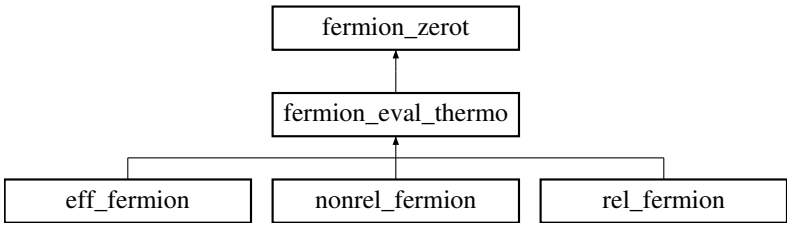
- fermion.h

8.14 fermion_zerot Class Reference

This is a base class for the computation of fermionic statistics at zero temperature. The more general case of finite temperature is taken care of by fermion_eval_thermo objects. The primary functions are calc_mu_zerot() and calc_density_zerot() which compute all the thermodynamic quantities as a function of the chemical potential, or the density, respectively.

`#include <fermion.h>`

Inheritance diagram for fermion_zerot:



8.14.1 Detailed Description

Idea for Future Use hypot() and other more accurate functions for the analytic expressions for the zero temperature integrals. [-Progress has been made, but there are probably other functions which may break down for small but finite masses and temperatures]

Definition at line 83 of file fermion.h.

Public Member Functions

Zero-temperature fermions

- void kf_from_density (fermion &f)
  Calculate the Fermi momentum from the density.
- void energy_density_zerot (fermion &f)
  Energy density at T=0 from fermion::kf and part::ms.
- void pressure_zerot (fermion &f)
  Pressure at T=0 from fermion::kf and part::ms.
- virtual void calc_mu_zerot (fermion &f)
  Zero temperature fermions from part::nu and part::ms.
- virtual void calc_density_zerot (fermion &f)
  Zero temperature fermions from part::n and part::ms.

8.14.2 Member Function Documentation

8.14.2.1 void fermion_zerot::kf_from_density (fermion & *f*)

Uses the relation $k_F = (6\pi^2 n/g)^{1/3}$

8.14.2.2 void fermion_zerot::energy_density_zerot (fermion & *f*)

Calculates the integral

$$\varepsilon = \frac{g}{2\pi^2}\int_0^{k_F} k^2\sqrt{k^2+m^{*2}}dk$$

8.14.2.3 void fermion_zerot::pressure_zerot (fermion & *f*)

Calculates the integral

$$P = \frac{g}{6\pi^2}\int_0^{k_F} \frac{k^4}{\sqrt{k^2+m^{*2}}}dk$$

8.14.2.4 virtual void fermion_zerot::calc_mu_zerot (fermion & *f*) [virtual]

This function always returns `gsl_success`.

Reimplemented in nonrel_fermion.

8.14.2.5 virtual void fermion_zerot::calc_density_zerot (fermion & *f*) [virtual]

This function always returns `gsl_success`.

Reimplemented in nonrel_fermion.

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

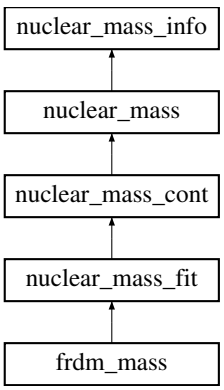
- fermion.h

8.15 frdm_mass Class Reference

FRDM semi-empirical mass formula (macroscopic part only with no deformation)

`#include <frdm_mass.h>`

Inheritance diagram for frdm_mass:



8.15.1 Detailed Description

The macroscopic part of the finite-range droplet model from Moller95 .

Using the relations

$$\bar{\delta} = (n_n - n_p)/n$$

and

$$\bar{\varepsilon} = -(n - n_0)/3/n_0$$

we get

$$n_n = \frac{1}{2}(1+\bar{\delta})(1-3\bar{\varepsilon})n_0$$

and

$$n_p = \frac{1}{2}(1-\bar{\delta})(1-3\bar{\varepsilon})n_0$$

Assuming that

$$\frac{4\pi}{3}R_n^3 n_n = N$$

and

$$\frac{4\pi}{3}R_p^3 n_p = Z$$

we get

$$R_n^3 = 3N/\alpha_n$$
$$R_p^3 = 3Z/\alpha_p$$

where α's are

$$\alpha_n = 2\pi(1+\bar{\delta})(1-3\bar{\varepsilon})n_0$$
$$\alpha_p = 2\pi(1-\bar{\delta})(1-3\bar{\varepsilon})n_0$$

Note that the above relations are somehow self-consistent because they imply

$$R^3 n = R_n^3 n_n + R_p^3 n_p$$

Since we're using (is there a better way?)

$$R = r_0 A^{1/3}$$

with $r_0 = 1.16$ fm, then $n_0 = 0.152946\text{fm}^{-3}$.

Todo Fix pairing energy and double vs. int
Document drip_binding_energy(), etc.
Decide on number of fit parameters (10 or 12?) or let the user decide
Document the protected variables
Set the neutron and proton masses and hbarc to Moller et al.'s values

Idea for Future Add microscopic part.

Definition at line 103 of file frdm_mass.h.

Public Member Functions

- virtual double mass_excess_d (double Z, double N)
  Given Z and N, return the mass excess in MeV.
- virtual int fit_fun (size_t nv, const **ovector_base** &x)
  Fix parameters from an array for fitting.
- virtual int guess_fun (size_t nv, **ovector_base** &x)
  Fill array with guess from present values for fitting.
- virtual double drip_binding_energy_d (double Z, double N, double npout, double nnout, double chi)
  Return the binding energy in MeV.
- virtual double drip_mass_excess_d (double Z, double N, double np_out, double nn_out, double chi)
  Given Z and N, return the mass excess in MeV.

Data Fields

- double a1
  Volume-energy constant in MeV (default 16.247)
- double J
  Symmetry-energy constant in MeV (default 32.73)
- double K
  Nuclear compressibility constant in MeV (default 240)
- double a2
  Surface-energy constant in MeV (default 22.92)
- double Q
  Effective surface-stiffness constant in MeV (default 29.21)
- double a3
  Curvature-energy constant in MeV (default 0)
- double ca
  Charge-asymmetry constant in MeV (default 0.436)
- double W
  Wigner constant in MeV (default 30)
- double ael
  electronic-binding constant in MeV (default 1.433×10^{-5} *).*
- double rp
  Proton root-mean-square radius in fm (default 0.80)
- double r0
  Nuclear-radius constant in fm (default 1.16)
- double MH
  Hydrogen atom mass excess, 7.289034 MeV.
- double Mn
  Neutron mass excess, 8.071431 MeV.
- double e2
  Electronic charge squared, 1.4399764 MeV fm.
- double a
  Range of Yukawa-plus-exponential potential, 0.68 fm.
- double aden
  Range of Yukawa function used to generate nuclear charge distribution, 0.70 fm.
- double rmac
  Average pairing-gap constant, 4.80 MeV.
- double h
  Neutron-proton interaction constant, 6.6 MeV.
- double L
  Density-symmetry constant, 0 MeV.
- double C
  Pre-exponential compressibility-term constant, 60 MeV.
- double gamma
  Exponential compressibility-term range constant, 0.831.
- double amu
  Atomic mass unit, 931.5014 MeV.
- double nn
  Internal average neutron density.
- double np
  Internal average proton density.
- double Rn
  Neutron radius.
- double Rp
  Proton radius.

Protected Attributes

- double Deltap
  Proton pairing coefficient.
- double Deltan

Neutron pairing coefficient.
- double deltanp
  Isovector pairing coefficient.
- double deltabar
  Average bulk nuclear asymmetry.
- double epsbar
  Average relative deviation of bulk density.
- double Bs
  Desc.
- double Bk
  Desc.
- double Br
  Desc.
- double Bw
  Desc.
- double Bv
  Desc.
- double c1
  Coulomb energy coefficient.
- double c2
  Volume redistribution energy coefficient.
- double c4
  Coulomb exchange correction coefficient.
- double c5
  Surface redistribution energy coefficient.
- double f0
  Coefficient for the proton form-factor correction to the Coulomb energy.
- double a0
  Desc.
- double B1
  Desc.
- double B2
  Desc.
- double B3
  Desc.
- double B4
  Desc.

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

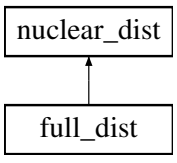
- frdm_mass.h

8.16 full_dist Class Reference

Full distribution including all nuclei from a discrete mass formula.

`#include <nuclear_dist.h>`

Inheritance diagram for full_dist:



8.16.1 Detailed Description

For example, to create a collection of all nuclei from the most recent (2003) Atomic Mass Evaluation, and then output all the nuclei in the collection

```
ame_mass ame;
full_dist fd(&ame);
for(nuclear_dist::iterator ndi=fd.begin();ndi!=fd.end();ndi++) {
  cout << ndi->Z << " " << ndi->A << " " << ndi->m << endl;
}
```

Definition at line 248 of file nuclear_dist.h.

Public Member Functions

- full_dist (nuclear_mass &nm, int maxA=400, bool include_neutron=false)
  Create a distribution including all nuclei with atomic numbers less than `maxA` from the mass formula `nm`.
- int set_dist (nuclear_mass &nm, int maxA=400, bool include_neutron=false)
  Set the distribution to all nuclei with atomic numbers less than `maxA` from the mass formula `nm`.
- virtual iterator begin ()
  The beginning of the distribution.
- virtual iterator end ()
  The end of the distribution.
- virtual size_t size ()
  The number of nuclei in the distribution.

8.16.2 Member Function Documentation

8.16.2.1 int full_dist::set_dist (nuclear_mass & *nm*, int *maxA* = `400`, bool *include_neutron* = `false`)

The information for the previous distribution is cleared before a new distribution is set.

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

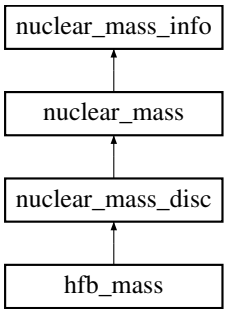
- nuclear_dist.h

8.17 hfb_mass Class Reference

HFB Mass formula.

`#include <nuclear_mass.h>`

Inheritance diagram for hfb_mass:



8.17.1 Detailed Description

Definition at line 1006 of file nuclear_mass.h.

Public Member Functions

- hfb_mass ()
  Create a new mass formula object.
- virtual bool is_included (int Z, int N)
  Return false if the mass formula does not include specified nucleus.
- virtual double mass_excess (int Z, int N)
  Given `Z` and `N`, return the mass excess in MeV.
- hfb_mass_entry get_ZN (int l_Z, int l_N)
  Get the entry for the specified proton and neutron number.
- bool is_loaded ()
  Verify that the constructor properly loaded the table.
- double blank ()
  The value which corresponds to a blank entry.
- virtual const char * type ()
  Return the type, `"hfb_mass"`.
- int set_data (int n_mass, hfb_mass_entry *m, std::string ref)
  Set data.
- int get_nentries ()
  Return number of entries.

Protected Attributes

- int n
  The number of entries (about 3000).
- std::string reference
  The reference for the original data.
- hfb_mass_entry * mass
  The array containing the mass data of length ame::n.
- int last
  The last table index for caching.

8.17.2 Member Function Documentation

8.17.2.1 hfb_mass_entry hfb_mass::get_ZN (int *l_Z*, int *l_N*)

This method searches the table using a cached binary search algorithm. It is assumed that the table is sorted first by proton number and then by neutron number.

8.17.2.2 int hfb_mass::set_data (int *n_mass*, hfb_mass_entry * *m*, std::string *ref*)

This function is used by the HDF I/O routines.

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

- nuclear_mass.h

8.18 hfb_mass_entry Struct Reference

Mass formula entry structure for HFB mass formula.

`#include <nuclear_mass.h>`

8.18.1 Detailed Description

Definition at line 907 of file nuclear_mass.h.

Data Fields

- int N
  Neutron number.
- int Z
  Proton number.
- int A
  Atomic number.
- double bet2
  Beta 2 deformation.
- double bet4
  Beta 4 deformation.
- double Rch
  RMS charge radius.
- double def_wig
  Deformation and Wigner energies.
- double Sn
  Neutron separation energy.
- double Sp
  Proton separation energy.
- double Qbet
  Beta-decay energy.
- double Mcal
  Calculated mass excess.
- double Err
  Error between experimental and calculated mass excess.

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

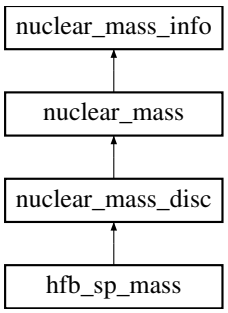
- nuclear_mass.h

8.19 hfb_sp_mass Class Reference

HFB Mass formula with spin and parity information.

`#include <nuclear_mass.h>`

Inheritance diagram for hfb_sp_mass:



8.19.1 Detailed Description

Definition at line 1072 of file nuclear_mass.h.

Public Member Functions

- hfb_sp_mass ()
  Create a new mass formula object.
- virtual bool is_included (int Z, int N)
  Return false if the mass formula does not include specified nucleus.
- virtual double mass_excess (int Z, int N)
  Given `Z` and `N`, return the mass excess in MeV.
- hfb_sp_mass_entry get_ZN (int l_Z, int l_N)
  Get the entry for the specified proton and neutron number.
- virtual const char * type ()
  Return the type, `"hfb_mass"`.
- int set_data (int n_mass, hfb_sp_mass_entry *m, std::string ref)
  Set data.

Protected Attributes

- hfb_sp_mass_entry * mass
  The array containing the mass data of length ame::n.
- int n
  The number of entries (about 3000).
- std::string reference
  The reference for the original data.
- int last
  The last table index for caching.

8.19.2 Member Function Documentation

8.19.2.1 hfb_sp_mass_entry hfb_sp_mass::get_ZN (int *l_Z*, int *l_N*)

This method searches the table using a cached binary search algorithm. It is assumed that the table is sorted first by proton number and then by neutron number.

8.19.2.2 int hfb_sp_mass::set_data (int *n_mass*, hfb_sp_mass_entry * *m*, std::string *ref*)

This function is used by the HDF I/O routines.

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

- nuclear_mass.h

8.20 hfb_sp_mass_entry Struct Reference

Version of hfb_mass_entry with spin and parity.

```
#include <nuclear_mass.h>
```

8.20.1 Detailed Description

Note

This cannot be a child of hfb_mass_entry in order for the HDF I/O preprocessor macros, like HOFFSET, to work

Definition at line 952 of file nuclear_mass.h.

Data Fields

- int N
  Neutron number.
- int Z
  Proton number.
- int A
  Atomic number.
- double bet2
  Beta 2 deformation.
- double bet4
  Beta 4 deformation.
- double Rch
  RMS charge radius.
- double def_wig
  Deformation and Wigner energies.
- double Sn
  Neutron separation energy.
- double Sp
  Proton separation energy.
- double Qbet
  Beta-decay energy.
- double Mcal
  Calculated mass excess.
- double Err
  Error between experimental and calculated mass excess.
- double Jexp
  Experimental spin.
- double Jth
  Theoretical spin.
- int Pexp
  Experimental parity.
- int Pth
  Theoretical parity.

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

- nuclear_mass.h

8.21 nuclear_dist::iterator Class Reference

An iterator for the nuclear distribution.

`#include <nuclear_dist.h>`

8.21.1 Detailed Description

The standard usage of this iterator is something of the form:

```
mnmsk_mass mth;
simple_dist sd(5,6,10,12,&mth);
for(nuclear_dist::iterator ndi=sd.begin();ndi!=sd.end();ndi++) {
// do something here for each nucleus
}
```

which would create a list consisting of three isotopes (A=10, 11, and 12) of boron and three isotopes carbon for a total of six nuclei.

Definition at line 69 of file nuclear_dist.h.

Public Member Functions

- iterator (nuclear_dist *ndpp, nucleus *npp)
  Create an iterator from the given distribution using the nucleus specified in `npp`.
- iterator operator++ ()
  Proceed to the next nucleus.
- iterator operator++ (int unused)
  Proceed to the next nucleus.
- nucleus * operator-> () const
  Pointing at operator.
- nucleus & operator* () const
  Dereference the iterator.

Protected Attributes

- nucleus * np
  A pointer to the current nucleus.
- nuclear_dist * ndp
  A pointer to the distribution.

Friends

- bool operator== (const nuclear_dist::iterator &i1, const nuclear_dist::iterator &i2)
  Give access to the == operator.
- bool operator!= (const nuclear_dist::iterator &i1, const nuclear_dist::iterator &i2)
  Give access to the != operator.

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

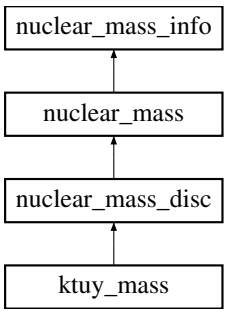
- nuclear_dist.h

8.22 ktuy_mass Class Reference

KTUY Mass formula.

`#include <nuclear_mass.h>`

Inheritance diagram for ktuy_mass:



8.22.1 Detailed Description

Definition at line 1208 of file nuclear_mass.h.

Public Member Functions

- ktuy_mass (std::string model="05", bool external=false)
  Create a new mass formula object using the specified model number.
- virtual bool is_included (int Z, int N)
  Return false if the mass formula does not include specified nucleus.
- virtual double mass_excess (int Z, int N)
  Given `Z` and `N`, return the mass excess in MeV.
- ktuy_mass_entry get_ZN (int l_Z, int l_N)
  Get the entry for the specified proton and neutron number.
- bool is_loaded ()
  Verify that the constructor properly loaded the table.
- virtual const char * type ()
  Return the type, `"ktuy_mass"`.
- int get_nentries ()
  Return number of entries.

Protected Attributes

- int n
  The number of entries (about 3000).
- std::string reference
  The reference for the original data.
- ktuy_mass_entry * mass
  The array containing the mass data of length ame::n.
- int last
  The last table index for caching.

8.22.2 Member Function Documentation

8.22.2.1 ktuy_mass_entry ktuy_mass::get_ZN (int *l_Z*, int *l_N*)

This method searches the table using a cached binary search algorithm. It is assumed that the table is sorted first by proton number and then by neutron number.

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

- nuclear_mass.h

8.23 ktuy_mass_entry Struct Reference

Mass formula entry structure for KTUY mass formula.

`#include <nuclear_mass.h>`

8.23.1 Detailed Description

Definition at line 1178 of file nuclear_mass.h.

Data Fields

- int N
  Neutron number.
- int Z
  Proton number.

- int A
  Atomic number.
- double Mcal
  Calculated mass excess.
- double Esh
  Shell energy.
- double alpha2
  Alpha 2 deformation.
- double alpha4
  Alpha 4 deformation.
- double alpha6
  Alpha 6 deformation.

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

- nuclear_mass.h

8.24 mass_fit Class Reference

Fit a nuclear mass formula.

`#include <mass_fit.h>`

8.24.1 Detailed Description

There is an example of the usage of this class given in Nuclear mass fit example.

Idea for Future Convert to a real fit with errors and covariance, etc.

Definition at line 44 of file mass_fit.h.

Public Member Functions

- virtual int fit (nuclear_mass_fit &n, double &res)
  Fit the nuclear mass formula.
- virtual int eval (nuclear_mass &n, double &res)
  Evaluate quality without fitting.
- int set_mmin (**multi_min**< **multi_funct**<> > &umm)
  Change the minimizer for use in the fit.
- int set_dist (nuclear_dist &uexp)
  Set the distribution of nuclei to fit.
- int set_exp_mass (nuclear_mass &nm)
  Set the experimental nuclear mass formula.

Data Fields

- bool even_even
  If true, then only fit doubly-even nuclei (default false)
- int minZ
  Minimum proton number to fit (default 8)
- int minN
  Minimum neutron number to fit (default 8)
- **gsl_mmin_simp2**< **multi_funct**<> > def_mmin
  The default minimizer.
- full_dist def_dist
  The default distribution of nuclei to fit (defaults to all nuclei in def_exp_mass)
- ame_mass def_exp_mass
  The default experimental nuclear mass object for def_dist.

8.24.2 Field Documentation

8.24.2.1 gsl_mmin_simp2<multi_funct<> > mass_fit::def_mmin

The value of def_mmin::ntrial is automatically multiplied by 10 in the constructor because the minimization frequently requires more trials than the default.

Definition at line 73 of file mass_fit.h.

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

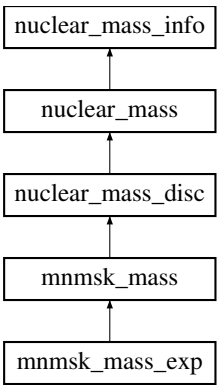
- mass_fit.h

8.25 mnmsk_mass Class Reference

Mass formula from Moller, Nix, Myers, Swiatecki and Kratz.

`#include <nuclear_mass.h>`

Inheritance diagram for mnmsk_mass:



8.25.1 Detailed Description

This is based on the tables given in Moller95 and Moller97.

Note

This class requires data stored in an HDF file and thus requires HDF support for normal usage.

The data containing an object of type moller_mass_entry for 8979 nuclei is automatically loaded by the constructor. If the file (nucmass/mnmsk.o2) is not found, then is_loaded() will return `false` and all calls to get_ZN() will return an object with `N=Z=0`.

There are several entries in the original table which are blank because they are in some way not known, measured, or computable. To distinguish these values from zero, blank entries have been replaced by the number `1.0e99`. For convenience, this value is returned by blank().

Definition at line 816 of file nuclear_mass.h.

Public Member Functions

- virtual bool is_included (int Z, int N)
  Return false if the mass formula does not include specified nucleus.
- virtual double mass_excess (int Z, int N)

Given Z and N, return the mass excess in MeV.

- mnmsk_mass_entry get_ZN (int l_Z, int l_N)
  Get the entry for the specified proton and neutron number.
- double blank ()
  The value which corresponds to a blank entry.
- double neither ()
  Neither beta+ or beta- is possible.
- double beta_stable ()
  The value which corresponds to a blank entry.
- double beta_plus_and_minus ()
  Both beta+ and beta- are possible.
- double greater_100 ()
  The value is greater than 100.
- double very_large ()
  The value is greater than 10^{20}.
- virtual const char * type ()
  Return the type, "mnmsk_mass".
- int set_data (int n_mass, mnmsk_mass_entry *m, std::string ref)
  Set data.

Protected Attributes

- int n
  The number of entries (about 3000).
- std::string reference
  The reference for the original data.
- mnmsk_mass_entry * mass
  The array containing the mass data of length ame::n.
- int last
  The last table index for caching.

8.25.2 Member Function Documentation

8.25.2.1 mnmsk_mass_entry mnmsk_mass::get_ZN (int *l_Z*, int *l_N*)

This method searches the table using a cached binary search algorithm. It is assumed that the table is sorted first by proton number and then by neutron number.

8.25.2.2 int mnmsk_mass::set_data (int *n_mass*, mnmsk_mass_entry * *m*, std::string *ref*)

This function is used by the HDF I/O routines.

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

- nuclear_mass.h

8.26 mnmsk_mass_entry Struct Reference

Mass formula entry structure for Moller, et al.

```
#include <nuclear_mass.h>
```

8.26.1 Detailed Description

Definition at line 687 of file nuclear_mass.h.

Data Fields

- int N
  Neutron number.
- int Z
  Proton number.
- int A
  Atomic number.
- double Emic
  The ground-state microscopic energy.
- double Mth
  The theoretical mass excess (in MeV)
- double Mexp
  The experimental mass excess (in MeV)
- double sigmaexp
  Experimental mass excess error.
- double EmicFL
  The ground-state microscopic energy in the FRLDM.
- double MthFL
  The theoretical mass excess in the FRLDM.
- char spinp [6]
  Spin and pairity of odd proton.
- char spinn [6]
  Spin and pairity of odd neutron.
- double gapp
  Lipkin-Nogami proton gap.
- double gapn
  Lipkin-Nogami neutron gap.
- double be
  Total binding energy.
- double S1n
  One neutron separation energy.
- double S2n
  Two neutron separation energy.
- double PA
  Percentage of daughters generated in beta decay after beta-delayed neutron emission.
- double PAm1
  Desc.
- double PAm2
  Desc.
- double Qbeta
  Energy released in beta-decay.
- double Tbeta
  Half-life w.r.t. GT beta-decay.
- double S1p
  One proton separation energy.
- double S2p
  Two proton separation energy.
- double Qalpha
  Energy released in alpha-decay.
- double Talpha
  Half-life w.r.t. alpha-decay.

Ground state deformations (perturbed-spheroid parameterization)

- double eps2
  Quadrupole.
- double eps3
  Octupole.
- double eps4
  Hexadecapole.

- double eps6
  Hexacontatetrapole.
- double eps6sym
  Hexacontatetrapole without mass asymmetry.

Ground state deformations in the spherical-harmonics expansion

- double beta2
  Quadrupole.
- double beta3
  Octupole.
- double beta4
  Hexadecapole.
- double beta6
  Hexacontatetrapole.

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

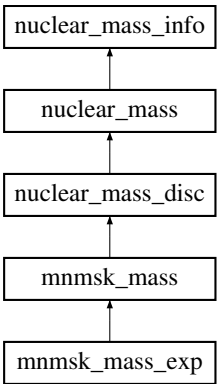
- nuclear_mass.h

8.27 mnmsk_mass_exp Class Reference

The experimental values from Moller, Nix, Myers and Swiatecki.

`#include <nuclear_mass.h>`

Inheritance diagram for mnmsk_mass_exp:



8.27.1 Detailed Description

Note

This class requires data stored in an HDF file and thus requires HDF support for normal usage.

Definition at line 891 of file nuclear_mass.h.

Public Member Functions

- virtual bool is_included (int Z, int N)
  Return false if the mass formula does not include specified nucleus.
- virtual double mass_excess (int Z, int N)
  Given `Z` and `N`, return the mass excess in MeV.

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

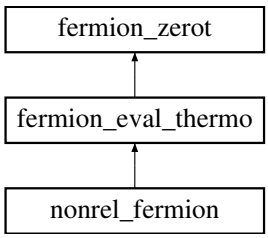
- nuclear_mass.h

8.28 nonrel_fermion Class Reference

Nonrelativistic fermion class.

`#include <nonrel_fermion.h>`

Inheritance diagram for nonrel_fermion:



8.28.1 Detailed Description

The rest mass energy density is given by n∗m not n∗ms. Note that the effective mass here is the Landau mass, not the Dirac mass.

Pressure is computed with

$$P = 2\varepsilon/3$$

and entropy density with

$$s = \frac{5\varepsilon}{3T} - \frac{n\mu}{T}$$

These relations can be verified with an integration by parts. See, e.g. Callen pg. 403 or Landau pg. 164.

The functions fermion::pair_density() and pair_mu() have not been implemented.

Todo Check behaviour of calc_density() at zero density, and compare with that from eff_fermion, rel_fermion, and classical.

Todo Implement pair_density() and pair_mu().

Todo Make sure to test with non-interacting equal to true or false, and document whether or not it works with both inc_rest_mass equal to true or false

Idea for Future This could be improved by performing a Chebyshev approximation (for example) to invert the density integral so that we don't need to use a solver.

Definition at line 75 of file nonrel_fermion.h.

Public Member Functions

- nonrel_fermion ()
  Create a nonrelativistic fermion with mass 'm' and degeneracy 'g'.
- virtual void calc_mu_zerot (fermion &f)
  Zero temperature fermions.
- virtual void calc_density_zerot (fermion &f)
  Zero temperature fermions.
- virtual void calc_mu (fermion &f, double temper)
  Calculate properties as function of chemical potential.
- virtual void calc_density (fermion &f, double temper)
  Calculate properties as function of density.
- virtual void pair_mu (fermion &f, double temper)

Calculate properties with antiparticles as function of chemical potential.

- virtual void pair_density (fermion &f, double temper)
  Calculate properties with antiparticles as function of density.
- virtual void nu_from_n (fermion &f, double temper)
  Calculate effective chemical potential from density.
- int set_density_root (**root**< **funct** > &rp)
  Set the solver for use in calculating the chemical potential from the density.
- virtual const char * type ()
  Return string denoting type ("nonrel_fermion")

Data Fields

- **cern_mroot_root**< **funct** > def_density_root
  The default solver for calc_density().

8.28.2 Member Function Documentation

8.28.2.1 virtual void nonrel_fermion::calc_density (fermion & *f*, double *temper*) [virtual]

If the density is zero, this function just sets part::mu, part::nu, part::ed, part::pr, and part::en to zero and returns without calling the error handler (even though at zero density and finite temperature, the chemical potentials formally are equal to $-\infty$).

Implements fermion_eval_thermo.

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

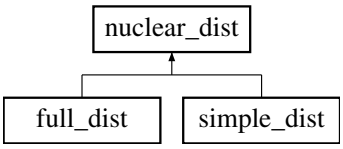
- nonrel_fermion.h

8.29 nuclear_dist Class Reference

A distribution of nuclei [abstract base].

`#include <nuclear_dist.h>`

Inheritance diagram for nuclear_dist:



8.29.1 Detailed Description

The virtual base class for a collection of objects of type nucleus . See simple_dist and full_dist for implmentations of this base class.

Definition at line 40 of file nuclear_dist.h.

Data Structures

- class iterator
  An iterator for the nuclear distribution.

Public Member Functions

- virtual iterator begin ()=0
  The beginning of the distribution.
- virtual iterator end ()=0
  The end of the distribution.
- virtual size_t size ()=0
  The number of nuclei in the distribution.

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

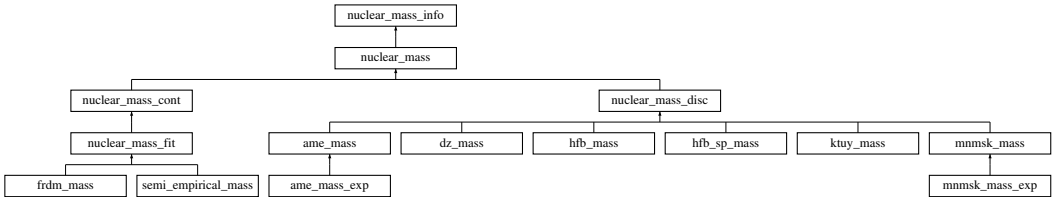
- nuclear_dist.h

8.30 nuclear_mass Class Reference

Nuclear mass formula base [abstract base].

`#include <nuclear_mass.h>`

Inheritance diagram for nuclear_mass:



8.30.1 Detailed Description

This base class provides some default functionality for the nuclear mass formulas. For typical usage, use ame_mass, mnmsk_mass, mnmsk_mass_exp, or semi_empirical_mass.

Elements 113, 115, 117 and 118 are named "Uut", "Uup", "Uus", and "Uuo", respectively.

Binding energies are determined from mass excesses by

$$\text{binding energy} = Au - Z(m_p + m_e) - Nm_n + \text{mass excess}$$

where u, m_n, m_p and m_e are all in **o2scl_fm** . "Total masses" are the mass of the nuclide (without the electron mass contribution)

$$\text{total mass} = \text{mass excess} + Au - Zm_e$$

Note that O_2scl generally ignores contributions from electron binding energies.

Some mass formulas are undefined for sufficiently exotic nuclei. You can use the function is_included() to find if a particular nucleus is included or not in a particular mass formula.

Generally, descendants of this class only need to provide an implementation of mass_excess() and mass_excess_d() and possibly a new version of is_included().

Some common reaction Q-values and separation energies:

$Q(\beta^-) = M(A,Z) - M(A,Z+1)$: Beta-decay energy

$Q(2\beta^-) = M(A,Z) - M(A,Z+2)$: Double beta-decay energy

$Q(4\beta^-) = M(A,Z) - M(A,Z+4)$: Four beta-decay energy

$Q(\alpha) = M(A,Z) - M(A-4,Z-2) - M(He^4)$: Alpha-decay energy

$Q(\beta - n) = M(A,Z) - M(A-1,Z+1) - M(n)$: Beta-delayed neutron emission decay energy

$Q(d,\alpha) = M(A,Z) - M(A-2,Z-1) - M(He^4) - M(H^2)$: (d,α) reaction energy

$Q(EC) = M(A,Z) - M(A,Z-1)$: Electron capture decay energy

$Q(ECp) = M(A,Z) - M(A-1,Z-2)$: Electron capture with delayed proton emission decay energy

$Q(n,\alpha) = M(A,Z) - M(A-3,Z-2) - M(He^4) + M(n)$: (n,α) reaction energy

$Q(p,\alpha) = M(A,Z) - M(A-3,Z-1) - M(He^4) + M(p)$: (p,α) reaction energy

$S(n) = -M(A,Z) + M(A-1,Z) + M(n)$: Neutron separation energy

$S(p) = -M(A,Z) + M(A-1,Z-1) + H^1$: Proton separation energy

$S(2n) = -M(A,Z) + M(A-2,Z) + 2M(n)$: Two neutron separation energy

$S(2p) = -M(A,Z) + M(A-2,Z-2) + 2M(H^{1)}$: Two proton separation energy

Idea for Future Find a way to include the electron binding energy contribution, possibly with the help of a theoretical model.

Definition at line 192 of file nuclear_mass.h.

Public Member Functions

- virtual const char * type ()
  Return the type, "nuclear_mass".
- virtual bool is_included (int Z, int N)
  Return false if the mass formula does not include specified nucleus.
- virtual int get_nucleus (int Z, int N, nucleus &n)
  Fill n with the information from nucleus with the given neutron and proton number.
- virtual double mass_excess (int Z, int N)=0
  Given Z and N, return the mass excess in MeV.
- virtual double mass_excess_d (double Z, double N)=0
  Given Z and N, return the mass excess in MeV.
- virtual double electron_binding (double Z)
  Return the approximate electron binding energy in MeV.
- virtual double binding_energy (int Z, int N)
  Return the binding energy in MeV.
- virtual double binding_energy_d (double Z, double N)
  Return the binding energy in MeV.
- virtual double total_mass (int Z, int N)
  Return the total mass of the nucleus (without the electrons) in MeV.
- virtual double total_mass_d (double Z, double N)
  Return the total mass of the nucleus (without the electrons) in MeV.
- virtual double atomic_mass (int Z, int N)
  Return the atomic mass of the nucleus in MeV (includes electrons and their binding energy)
- virtual double atomic_mass_d (double Z, double N)
  Return the atomic mass of the nucleus in MeV (includes electrons and their binding energy)

Data Fields

- double m_neut
  Neutron mass in MeV *(defaults to* **o2scl_fm::mass_neutron** *times* **o2scl_const::hc_mev_fm***)*
- double m_prot
  Proton mass in MeV *(defaults to* **o2scl_fm::mass_proton** *times* **o2scl_const::hc_mev_fm***)*
- double m_elec
  Electron mass in MeV *(defaults to o2scl_fm::mass_elec times* **o2scl_const::hc_mev_fm***)*
- double m_amu
  Atomic mass unit in MeV *(defaults to* **o2scl_fm::mass_amu** *times* **o2scl_const::hc_mev_fm***)*

8.30.2 Member Function Documentation

8.30.2.1 virtual int nuclear_mass::get_nucleus (int *Z*, int *N*, nucleus & *n*) [virtual]

All masses are given in fm^{-1}. The total mass (withouth the electrons) is put in part::m and part::ms, the binding energy is placed in nucleus::be, the mass excess in nucleus::mex and the degeneracy (part::g) is arbitrarily set to 1 for even A nuclei and 2 for odd A nuclei.

8.30.2.2 virtual double nuclear_mass::binding_energy (int *Z*, int *N*) [inline, virtual]

The binding energy is defined to be negative for bound nuclei, thus the binding energy per baryon of Pb-208 is about -8∗208 = -1664 MeV.

Definition at line 257 of file nuclear_mass.h.

8.30.2.3 virtual double nuclear_mass::binding_energy_d (double *Z*, double *N*) [inline, virtual]

The binding energy is defined to be negative for bound nuclei, thus the binding energy per baryon of Pb-208 is about -8∗208 = -1664 MeV.

Definition at line 267 of file nuclear_mass.h.

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

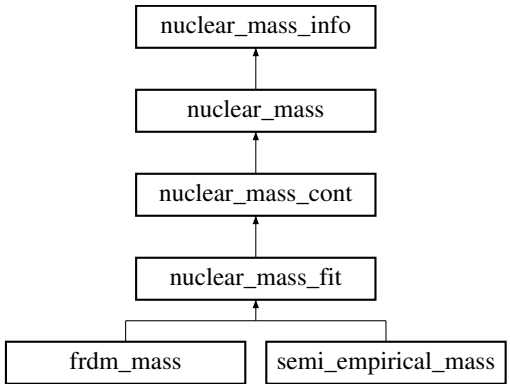
- nuclear_mass.h

8.31 nuclear_mass_cont Class Reference

Continuous nuclear mass formula [abstract base].

`#include <nuclear_mass.h>`

Inheritance diagram for nuclear_mass_cont:



8.31.1 Detailed Description

Generally, descendants of this class only need to provide an implementation of mass_excess_d() and possibly a version of nuclear_mass::is_included()

Definition at line 330 of file nuclear_mass.h.

Public Member Functions

- virtual double mass_excess (int Z, int N)

Given Z and N, return the mass excess in MeV.
- virtual double mass_excess_d (double Z, double N)=0
  Given Z and N, return the mass excess in MeV.

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

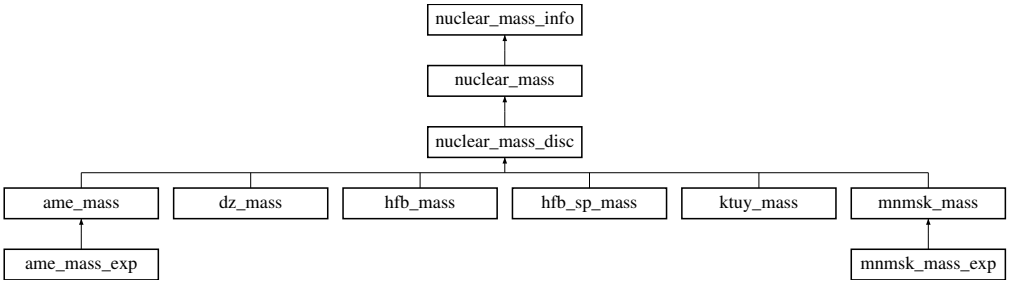
- nuclear_mass.h

8.32 nuclear_mass_disc Class Reference

Discrete nuclear mass formula [abstract base].

`#include <nuclear_mass.h>`

Inheritance diagram for nuclear_mass_disc:



8.32.1 Detailed Description

This uses simple linear interpolation to obtain masses of nuclei with non-integer Z and N which may be particularly sensitive to the form of the pairing.

Generally, descendants of this class only need to provide an implementation of mass_excess() and possibly a version of nuclear_mass::is_included()

Definition at line 312 of file nuclear_mass.h.

Public Member Functions

- virtual double mass_excess (int Z, int N)=0
  Given Z and N, return the mass excess in MeV.
- virtual double mass_excess_d (double Z, double N)
  Given Z and N, return the mass excess in MeV.

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

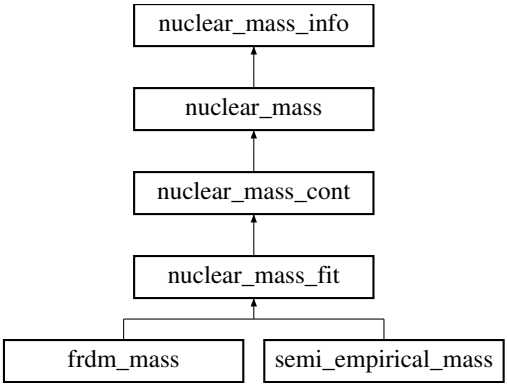
- nuclear_mass.h

8.33 nuclear_mass_fit Class Reference

Fittable mass formula.

`#include <nuclear_mass.h>`

Inheritance diagram for nuclear_mass_fit:



8.33.1 Detailed Description

Nuclear mass formulas which are descendants of this class can be fit to experiment using mass_fit.

Within O_2scl_part , this class has only two children, frdm_mass and semi_empirical_mass. There is also a child **ldrop_mass** in O_2scl_eos .

Idea for Future This shouldn't be a child of nuclear_mass_cont?

Definition at line 355 of file nuclear_mass.h.

Public Member Functions

- virtual const char * type ()
  Return the type, "nuclear_mass_fit".
- virtual int fit_fun (size_t nv, const **ovector_base** &x)=0
  Fix parameters from an array for fitting.
- virtual int guess_fun (size_t nv, **ovector_base** &x)=0
  Fill array with guess from present values for fitting.

Data Fields

- size_t nfit
  Number of fitting parameters.

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

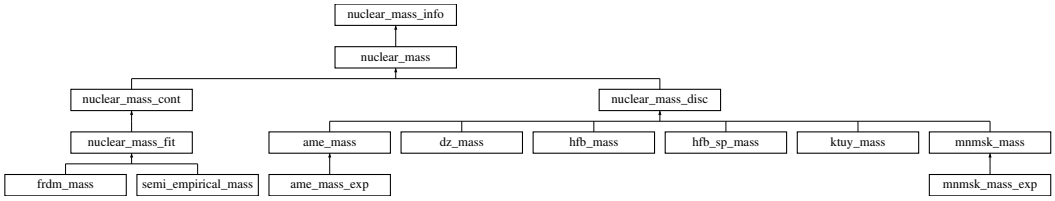
- nuclear_mass.h

8.34 nuclear_mass_info Class Reference

Nuclear mass info.

`#include <nuclear_mass.h>`

Inheritance diagram for nuclear_mass_info:



8.34.1 Detailed Description

Definition at line 38 of file nuclear_mass.h.

Public Member Functions

- int parse_elstring (std::string ela, int &Z, int &N, int &A)
  Parse a string representing an element.
- int eltoZ (std::string el)
  Return Z given the element name.
- std::string Ztoel (size_t Z)
  Return the element name given Z.
- std::string tostring (size_t Z, size_t N)
  Return a string of the form "Pb208" for a given Z and N.

Protected Types

- typedef std::map< std::string, int, **string_comp** >::iterator table_it
  A convenient typedef for an iterator for element_table.

Protected Attributes

- std::map< std::string, int, **string_comp** > element_table
  A map containing the proton numbers organized by element name.
- std::string element_list [nelements]
  The list of elements organized by proton number.

Static Protected Attributes

- static const int nelements = 119
  The number of elements (proton number)

8.34.2 Member Function Documentation

8.34.2.1 int nuclear_mass_info::parse_elstring (std::string *ela*, int & *Z*, int & *N*, int & *A*)

Accepts strings of one of the following forms:

- `Pb208`
- `pb208`
- `Pb 208`
- `Pb-208`

- `pb 208`
- `pb-208` or one of the special strings `n`, `p`, `d` or `t` for the neutron, proton, deuteron, and triton, respectively. This function also allows the value of A to precede the element symbol.

Note

At present, this allows nuclei which don't make sense because A<Z, such as Carbon-5.

Idea for Future Warn about malformed combinations like Carbon-5

Idea for Future Right now, `n4` is interpreted incorrectly as Nitrogen-4, rather than the tetraneutron.

8.34.2.2 int nuclear_mass_info::eltoZ (std::string *el*)

If the string parameter `el` is invalid, the error handler is called and the value -1 is returned.

8.34.2.3 std::string nuclear_mass_info::Ztoel (size_t *Z*)

Note

This function returns `"n"` indicating the neutron for Z=0, and if the argument `Z` is greater than 118, an empty string is returned after calling the error handler.

8.34.2.4 std::string nuclear_mass_info::tostring (size_t *Z*, size_t *N*)

Note that if `Z` is zero, then and `'n'` is used to indicate the a nucleus composed entirely of neutrons and if the argument `Z` is greater than 118, an empty string is returned (independ.

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

- nuclear_mass.h

8.35 nuclear_reaction Class Reference

A simple nuclear reaction specification.

`#include <reaction_lib.h>`

8.35.1 Detailed Description

This class is very experimental.

Definition at line 44 of file reaction_lib.h.

Public Member Functions

- std::string to_string ()
  Convert the reaction to a string for screen output.
- int clear ()
  Clear the rate.
- nuclear_reaction (const nuclear_reaction &nr)
  Copy constructor.
- nuclear_reaction & operator= (const nuclear_reaction &nr)
  Copy constructor.
- double rate (double T9)
  Compute the reaction rate from the temperature in units of $10^9 K$.

Data Fields

- size_t chap
  Chapter.
- std::string name [6]
  Names of the participating nuclei.
- std::string ref
  Reference.
- char type
  Type of rate (resonant/non-resonant/weak)
- char rev
  Forward or reverse.
- double Q
  Q value.
- double a [7]
  Coefficients.
- size_t Z [6]
  Proton number of participating nuclei.
- size_t A [6]
  Mass number of participating nuclei.
- size_t isomer [6]
  Isomer designation of participating nuclei.

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

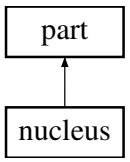
- reaction_lib.h

8.36 nucleus Class Reference

A simple nucleus class.

`#include <nucleus.h>`

Inheritance diagram for nucleus:



8.36.1 Detailed Description

The variable part::m is typically used for the mass of the nucleus with no electrons.

The binding energy of the nucleus (be) is typically defined as the mass of the nucleus (without the electrons) minus Z times the mass of the proton minus N times the mass of the neutron.

The mass excess (be) is defined as the mass of the nucleus including the electron contribution minus a times the mass of the atomic mass unit.

The variable part::inc_rest_mass is set to `false` by default, to insure that energies and chemical potentials do not include the rest mass. This is typically appropriate for nuclei.

Definition at line 50 of file nucleus.h.

Data Fields

- int Z
  Proton number.
- int N
  Neutron number.
- int A
  Atomic number.
- double mex
  Mass excess in fm^{-1}.
- double be
  Binding energy in fm^{-1} *(with a minus sign for bound nuclei)*

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

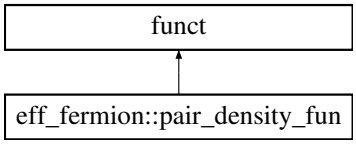
- nucleus.h

8.37 eff_fermion::pair_density_fun Class Reference

Define the function which solves for the chemical potential given the density of particles and antiparticles [protected subclass of eff_fermion].

`#include <eff_fermion.h>`

Inheritance diagram for eff_fermion::pair_density_fun:



8.37.1 Detailed Description

Definition at line 248 of file eff_fermion.h.

Public Member Functions

- **pair_density_fun** (eff_fermion &ef, fermion &f, double T)
- double operator() (double x)
  Fix density for eff_fermion::pair_density()

Protected Attributes

- eff_fermion & **ef_**
- fermion & **f_**
- double **T_**

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

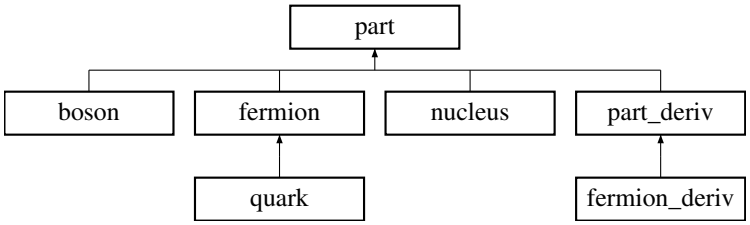
- eff_fermion.h

8.38 part Class Reference

Particle base class.

`#include <part.h>`

Inheritance diagram for part:



8.38.1 Detailed Description

Calculate the properties of particles from their chemical potential (calc_mu() and pair_mu()) or from the density (calc_density() and pair_density()).

When non-interacting is false, the thermodynamic integrals need both a value of "mu" and "nu". "nu" is an effective chemical potential which appears in the argument of the exponential of the Fermi-function.

Keep in mind, that the pair functions use anti(), which assumes that nu -> -nu and mu -> -mu for the anti-particles, which might not be true for interacting particles. When non-interacting is true, then "ms" is set equal to "m", and "nu" is set equal to "mu", everywhere.

The "density" functions use the value of nu (or mu when non_interacting is true) for an initial guess. Zero is very likely a bad guess, but these functions will not warn you about this.

Definition at line 92 of file part.h.

Public Member Functions

- part (double m=0.0, double g=0.0)
  make a particle of mass m and degeneracy g.
- virtual int init (double m, double g)
  Set the mass m and degeneracy g.
- virtual int anti (part &ax)
  Make an anti-particle.
- virtual const char * type ()
  Return string denoting type ("part")

Data Fields

- double g
  degeneracy
- double m
  mass
- double n
  density
- double ed
  energy density
- double pr
  pressure
- double mu

chemical potential
- double en
  entropy
- double ms
  effective mass (Dirac unless otherwise specified)
- double nu
  effective chemical potential
- bool inc_rest_mass
  derivative of energy with respect to effective mass
- bool non_interacting
  True if the particle is non-interacting (default true)

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

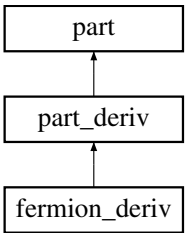
- part.h

8.39 part_deriv Class Reference

Storage for deriviatives wrt μ and T.

`#include <deriv_part.h>`

Inheritance diagram for part_deriv:



8.39.1 Detailed Description

The variables `dndmu`, `dndT`, and `dsdT` correspond to

$$\left(\frac{dn}{d\mu}\right)_{T,V}, \quad \left(\frac{dn}{dT}\right)_{\mu,V}, \quad \text{and} \quad \left(\frac{ds}{dT}\right)_{\mu,V}$$

respectively. All other derivatives can be expressed simply in terms of these three. Note that volume derivatives are trivial, since both the entropy and number scale linearly with the volume.

Derivatives wrt to chemical potential and temperature:

There is a Maxwell relation

$$\left(\frac{ds}{d\mu}\right)_{T,V} = \left(\frac{dn}{dT}\right)_{\mu,V}$$

The pressure derivatives are trivial

$$\left(\frac{dP}{d\mu}\right)_{T,V} = n, \quad \left(\frac{dP}{dT}\right)_{\mu,V} = s$$

The energy density derivatives are related through the thermodynamic identity:

$$\left(\frac{d\varepsilon}{d\mu}\right)_{T,V} = \mu\left(\frac{dn}{d\mu}\right)_{T,V} + T\left(\frac{ds}{d\mu}\right)_{T,V}$$

$$\left(\frac{d\varepsilon}{dT}\right)_{\mu,V} = \mu\left(\frac{dn}{dT}\right)_{\mu,V} + T\left(\frac{ds}{dT}\right)_{\mu,V}$$

Other derivatives:

Note that the derivative of the entropy with respect to the temperature above is not the specific heat per particle, c_V. The specific heat per particle is

$$c_V = \frac{T}{N}\left(\frac{\partial S}{\partial T}\right)_{V,N} = \frac{T}{n}\left(\frac{\partial s}{\partial T}\right)_{V,n}$$

As noted in Particles in the User's Guide for O$_2$scl_part , we work in units so that $\hbar = c = k_B = 1$. In this case, c_V is unitless as defined here. To compute c_V in terms of the derivatives above, note that the descendants of deriv_part provide all of the thermodynamic functions in terms of μ, V and T, so we have

$$s = s(\mu,T,V) \quad \text{and} \quad n = n(\mu,T,V)\,.$$

We can then construct a function

$$s = s[\mu(n,T,V),T,V]$$

and then write the required derivative directly

$$\left(\frac{\partial s}{\partial T}\right)_{n,V} = \left(\frac{\partial s}{\partial \mu}\right)_{T,V}\left(\frac{\partial \mu}{\partial T}\right)_{n,V} + \left(\frac{\partial s}{\partial T}\right)_{\mu,V}\,.$$

Now we use the identity

$$\left(\frac{\partial \mu}{\partial T}\right)_{n,V} = -\left(\frac{\partial n}{\partial T}\right)_{\mu,V}\left(\frac{\partial n}{\partial \mu}\right)^{-1}_{T,V}\,,$$

and the Maxwell relation above to give

$$C_V = \frac{T}{n}\left[\left(\frac{\partial s}{\partial T}\right)_{\mu,V} - \left(\frac{\partial n}{\partial T}\right)^2_{\mu,V}\left(\frac{\partial n}{\partial \mu}\right)^{-1}_{T,V}\right]$$

which expresses the specific heat in terms of the three derivatives which are given.

For, c_P, defined as

$$c_P = \frac{T}{N}\left(\frac{\partial S}{\partial T}\right)_{N,P}$$

(which is also unitless) we can write functions

$$S = S(N,T,V) \qquad \text{and} \qquad V = V(N,P,T)$$

which imply

$$\left(\frac{\partial S}{\partial T}\right)_{N,P} = \left(\frac{\partial S}{\partial T}\right)_{N,V} + \left(\frac{\partial S}{\partial V}\right)_{N,T}\left(\frac{\partial V}{\partial T}\right)_{N,P}\,.$$

Thus we require the derivatives

$$\left(\frac{\partial S}{\partial T}\right)_{N,V}, \left(\frac{\partial S}{\partial V}\right)_{N,T}, \qquad \text{and} \qquad \left(\frac{\partial V}{\partial T}\right)_{N,P}\,.$$

To compute the new entropy derivatives, we can write

$$S = S(\mu(N,T,V),T,V)$$

to get

$$\left(\frac{\partial S}{\partial T}\right)_{N,V} = \left(\frac{\partial S}{\partial \mu}\right)_{T,V}\left(\frac{\partial \mu}{\partial T}\right)_{N,V} + \left(\frac{\partial S}{\partial T}\right)_{\mu,V}\,,$$

---

and

$$\left(\frac{\partial S}{\partial V}\right)_{N,T} = \left(\frac{\partial S}{\partial \mu}\right)_{T,V}\left(\frac{\partial \mu}{\partial V}\right)_{N,T} + \left(\frac{\partial S}{\partial V}\right)_{\mu,T}.$$

These require the chemical potential derivatives which have associated Maxwell relations

$$\left(\frac{\partial \mu}{\partial T}\right)_{N,V} = -\left(\frac{\partial S}{\partial N}\right)_{T,V} \qquad \text{and} \qquad \left(\frac{\partial \mu}{\partial V}\right)_{N,T} = -\left(\frac{\partial P}{\partial N}\right)_{T,V}.$$

Finally, we can rewrite the derivatives on the right hand sides in terms of derivatives of functions of μ, V and T,

$$\left(\frac{\partial S}{\partial N}\right)_{T,V} = \left(\frac{\partial S}{\partial \mu}\right)_{T,V}\left(\frac{\partial N}{\partial \mu}\right)^{-1}_{T,V},$$

and

$$\left(\frac{\partial P}{\partial N}\right)_{T,V} = \left(\frac{\partial P}{\partial \mu}\right)_{T,V}\left(\frac{\partial N}{\partial \mu}\right)^{-1}_{T,V}.$$

The volume derivative,

$$\left(\frac{\partial V}{\partial T}\right)_{N,P},$$

is related to the coefficient of thermal expansion, sometimes called α,

$$\alpha \equiv \frac{1}{V}\left(\frac{\partial V}{\partial T}\right)_{N,P}.$$

We can rewrite the derivative

$$\left(\frac{\partial V}{\partial T}\right)_{N,P} = -\left(\frac{\partial P}{\partial T}\right)_{N,V}\left(\frac{\partial P}{\partial V}\right)^{-1}_{N,T}.$$

The first term can be computed from the Maxwell relation

$$\left(\frac{\partial P}{\partial T}\right)_{N,V} = \left(\frac{\partial S}{\partial V}\right)_{N,T},$$

where the entropy derivative was computed above. The second term (related to the inverse of the isothermal compressibility, $\kappa_T \equiv (-1/V)(\partial V/\partial P)_{T,N}$ can be computed from the function $P = P(\mu(N,V,T),V,T)$

$$\left(\frac{\partial P}{\partial V}\right)_{N,T} = \left(\frac{\partial P}{\partial \mu}\right)_{T,V}\left(\frac{\partial \mu}{\partial V}\right)_{N,T} + \left(\frac{\partial P}{\partial V}\right)_{\mu,T}$$

where the chemical potential derivative was computed above.

The results above can be collected to give

$$\left(\frac{\partial S}{\partial T}\right)_{N,P} = \left(\frac{\partial S}{\partial T}\right)_{\mu,V} + \frac{S^2}{N^2}\left(\frac{\partial N}{\partial \mu}\right)_{T,V} - \frac{2S}{N}\left(\frac{\partial N}{\partial T}\right)_{\mu,V},$$

which implies

$$c_P = \frac{T}{n}\left(\frac{\partial s}{\partial T}\right)_{\mu,V} + \frac{s^2 T}{n^3}\left(\frac{\partial n}{\partial \mu}\right)_{T,V} - \frac{2sT}{n^2}\left(\frac{\partial n}{\partial T}\right)_{\mu,V},$$

This derivation also gives the well-known relationship between the specific heats at constant volume and constant pressure,

$$c_P = c_V + \frac{T\alpha^2}{n\kappa_T}.$$

No derivative with respect to the bare mass is given, since classes cannot know how to relate the effective mass to the bare mass.

Definition at line 258 of file deriv_part.h.

Public Member Functions

- **part_deriv** (double mass=0.0, double dof=0.0)

Data Fields

- double dndmu
  Derivative of number density with respect to chemical potential.
- double dndT
  Derivative of number density with respect to temperature.
- double dsdT
  Derivative of entropy density with respect to temperature.
- double dndm
  Derivative of number density with respect to the effective mass.

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

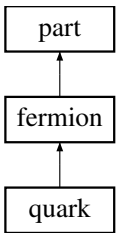
- deriv_part.h

8.40 quark Class Reference

Quark class.

`#include <quark.h>`

Inheritance diagram for quark:



8.40.1 Detailed Description

Definition at line 43 of file quark.h.

Public Member Functions

- quark (double m=0.0, double g=0.0)
  Create a boson with mass `m` and degeneracy `g`.
- virtual const char * type ()
  Return string denoting type ("quark")

Data Fields

- double B
  Contribution to the bag constant.
- double qq
  Quark condensate.

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

- quark.h

8.41 reaction_lib Class Reference

Simple reaction library.

`#include <reaction_lib.h>`

8.41.1 Detailed Description

This class is very experimental.

Units:

- Chapters 1,2,3, and 11: 1/s
- Chapters 4,5,6, and 7: cm^3/g/s
- Chapter 8 and 9: cm^6/g^2/s
- Chapter 10: cm^9/g^3/s

Chapters:

- 1: nuc1 -> nuc2
- 2: nuc1 -> nuc2 + nuc3
- 3: nuc1 -> nuc2 + nuc3 + nuc4
- 4: nuc1 + nuc2 -> nuc3
- 5: nuc1 + nuc2 -> nuc3 + nuc4
- 6: nuc1 + nuc2 -> nuc3 + nuc4 + nuc5
- 7: nuc1 + nuc2 -> nuc3 + nuc4 + nuc5 + nuc6
- 8: nuc1 + nuc2 + nuc3 -> nuc4
- 9: nuc1 + nuc2 + nuc3 -> nuc4 + nuc5
- 10: nuc1 + nuc2 + nuc3 + nuc4 -> nuc5 + num6
- 11: nuc1 -> nuc2 + nuc3 + nuc4 + nuc5

Original FORTRAN format:

```
FORMAT(i1,4x,6a5,8x,a4,a1,a1,3x,1pe12.5)
FORMAT(4e13.6)
FORMAT(3e13.6)
```

Definition at line 206 of file reaction_lib.h.

Public Member Functions

- int read_file_reaclib2 (std::string fname)
  Read from a file in the REACLIB2 format.
- int find_in_chap (std::vector< nuclear_reaction > &nrl, size_t chap, std::string nuc1, std::string nuc2="", std::string nuc3="", std::string nuc4="", std::string nuc5="", std::string nuc6="")
  Find a set of nuclear reactions in a specified chapter.

Data Fields

- std::vector< nuclear_reaction > lib
  The library.

Protected Member Functions

- bool matches (size_t ul, size_t ri)
  Test if entry `ul` in the arrays matches the library reaction.

Protected Attributes

Storage for the find function

- int **fN** [6]
- int **fZ** [6]
- int **fA** [6]
- size_t **fi**

8.41.2 Member Function Documentation

8.41.2.1 int reaction_lib::read_file_reaclib2 (std::string *fname*)

Note

This function does not check that the chapter numbers are correct for the subsequent reaction.

8.41.2.2 int reaction_lib::find_in_chap (std::vector< nuclear_reaction > & *nrl*, size_t *chap*, std::string *nuc1*, std::string *nuc2* = " ", std::string *nuc3* = " ", std::string *nuc4* = " ", std::string *nuc5* = " ", std::string *nuc6* = " ")

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

- reaction_lib.h

8.42 rel_boson Class Reference

Equation of state for a relativistic boson.

`#include <rel_boson.h>`

8.42.1 Detailed Description

Todo Testing not completely finished.

Definition at line 48 of file rel_boson.h.

Public Member Functions

- rel_boson ()
  Create a boson with mass `m` and degeneracy `g`.
- virtual void calc_mu (boson &b, double temper)
  Calculate properties as function of chemical potential.
- virtual void calc_density (boson &b, double temper)
  Calculate properties as function of density.
- virtual void pair_mu (boson &b, double temper)
  Calculate properties with antiparticles as function of chemical potential.
- virtual void pair_density (boson &b, double temper)
  Calculate properties with antiparticles as function of density.
- virtual void nu_from_n (boson &b, double temper)
  Calculate effective chemical potential from density.
- void set_inte (**inte**< **funct** > &l_nit, **inte**< **funct** > &l_dit)
  Set inte object.
- void set_density_root (**root**< **funct** > &rp)
  Set the solver for use in calculating the chemical potential from the density.
- virtual const char * type ()
  Return string denoting type ("rel_boson")

Data Fields

- int mroot_err
  The error value from mroot.
- int inte_err
  The error value from inte.
- **cern_mroot_root**< **funct** > def_density_root
  The default solver for calc_density().
- **gsl_inte_qagiu**< **funct** > def_nit
  Default nondegenerate integrator.
- **gsl_inte_qag**< **funct** > def_dit
  Default degenerate integrator.

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

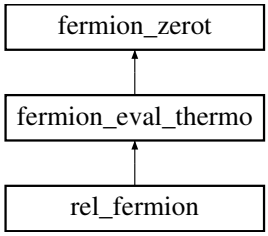
- rel_boson.h

8.43 rel_fermion Class Reference

Equation of state for a relativistic fermion.

`#include <rel_fermion.h>`

Inheritance diagram for rel_fermion:



8.43.1 Detailed Description

This class computes the thermodynamics of a relativistic fermion either as a function of the density or the chemical potential. It employs direct integration, using two different integrators for the degenerate and non-degenerate regimes. The default integrators are **gsl_inte_qag** (for degenerate fermions) and **gsl_inte_qagiu** (for non-degenerate fermions). For the functions calc_mu() and calc_density(), if the temperature argument is less than or equal to zero, the functions fermion_zerot::calc_mu_zerot() and fermion_zerot::calc_density_zerot() will be used to compute the result.

Degeneracy parameter:

Define the degeneracy parameter

$$\psi = (\nu - m^*)/T$$

where ν is the effective chemical potential and m^* is the effective mass. For ψ greater than deg_limit (degenerate regime), a finite interval integrator is used and for ψ less than deg_limit (non-degenerate regime), an integrator over the interval from $[0,\infty)$ is used. In the case where part::inc_rest_mass is false, the degeneracy parameter is

$$\psi = (\nu + m - m^*)/T$$

Integration limits:

The upper limit on the degenerate integration is given by

$$\text{upper limit} = \sqrt{\mathscr{L}^2 - m^{*,2}}$$

where $\mathscr{L} \equiv uT + \nu$ and u is rel_fermion::upper_limit_fac . In the case where part::inc_rest_mass is false, the result is

$$\text{upper limit} = \sqrt{(m + \mathscr{L})^2 - m^{*2}}$$

The entropy is only significant at the Fermi surface, thus in the degenerate case, the lower limit of the entropy integral can be given be determined by the value of k which solves

$$-u = \frac{\sqrt{k^2 + m^{*2}} - \nu}{T}$$

The solution is

$$\text{lower limit} = \sqrt{(-uT + \nu)^2 - m^{*,2}}$$

but this solution is only valid if $(m^* - \nu)/T < -u$. In the case where part::inc_rest_mass is false, the result is

$$\text{lower limit} = \sqrt{(-uT + m + \nu)^2 - m^{*,2}}$$

which is valid if $(m^* - \nu - m)/T < -u$.

Entropy integrand:

In the degenerate regime, the entropy integrand

$$-k^2 \left[f \log f + (1-f) \log (1-f) \right]$$

where f is the fermionic distribution function can lose precision when $(E^* - \nu)/T$ is negative and sufficiently large in absolute magnitude. Thus when $(E^* - \nu)/T < S$ where S is stored in deg_entropy_fac (default is -30), the integrand is written as

$$-k^2 \left(E/T - \nu/T \right) e^{E/T - \nu/T} .$$

If $(E - \nu)/T < S$ is less than -1 times exp_limit (e.g. less than -200), then the entropy integrand is assumed to be zero.

Non-degenerate integrands:

The integrands in the non-degenerate regime are written in a dimensionless form, by defining $p = \sqrt{(Tu + m^*)^2 - m^{*2}}$, $\nu \equiv yT$, and $m^* \equiv \text{mx}\, T$. The density integrand is

$$(\text{mx} + u) \sqrt{u^2 + 2(\text{mx})u} \left(\frac{e^y}{e^{\text{mx}+u} + e^y} \right) ,$$

---

the energy integrand is

$$(\mathrm{mx}+u)^2\sqrt{u^2+2(\mathrm{mx})u}\left(\frac{e^y}{e^{\mathrm{mx}+u}+e^y}\right),$$

and the entropy integrand is

$$(\mathrm{mx}+u)\sqrt{u^2+2(\mathrm{mx})u}\,(t_1+t_2),$$

where

$$\begin{aligned} t_1 &= \log\left(1+e^{y-\mathrm{mx}-u}\right)/\left(1+e^{y-\mathrm{mx}-u}\right) \\ t_2 &= \log\left(1+e^{\mathrm{mx}+u-y}\right)/\left(1+e^{\mathrm{mx}+u-y}\right). \end{aligned}$$

Accuracy:

The default settings for for this class give an accuracy of at least 1 part in 10^6 (and frequently better than this).

When the integrators provide numerical uncertainties, these uncertainties are stored in unc. In the case of calc_density() and pair_density(), the uncertainty from the numerical accuracy of the solver is not included. (There is also a relatively small inaccuracy due to the mathematical evaluation of the integrands which is not included in unc.)

One way to improve the accuracy of the computation is just to decrease the tolerances on the default integration objects. This can be done, using, for example

```
rel_fermion rf(1.0,2.0);
rf.def_dit.tolx/=1.0e2;
rf.def_dit.tolf/=1.0e2;
rf.def_nit.tolx/=1.0e2;
rf.def_nit.tolf/=1.0e2;
```

which decreases the both the relative and absolute tolerances for both the degenerate and non-degenerate integrators. If one is using either the calc_density() or pair_density() functions, one may also have to improve the accuracy of the solver which determines the chemical potential from the density. For the default solver, this could be done with

```
rf.def_density_root.tolx/=1.0e2;
rf.def_density_root.tolf/=1.0e2;
```

Of course if these tolerances are too small, the calculation may fail.

Todos:

Idea for Future The expressions which appear in in the integrand functions density_fun(), etc. could likely be improved, especially in the case where inc_rest_mass=false. There should not be a need to check if `ret` is finite.

Idea for Future It appears this doesn't compute the uncertainty in the chemical potential or density with calc_density(). This could be fixed.

Idea for Future I'd like to change the lower limit on the entropy integration, but the value in the code at the moment (stored in `ll`) makes bm_part2.cpp worse.

Idea for Future pair_mu() should set the antiparticle integrators as done in sn_fermion.

Definition at line 215 of file rel_fermion.h.

Public Member Functions

- rel_fermion ()

Create a fermion with mass m and degeneracy g.

- virtual void calc_mu (fermion &f, double temper)
  Calculate properties as function of chemical potential.
- virtual void calc_density (fermion &f, double temper)
  Calculate properties as function of density.
- virtual void pair_mu (fermion &f, double temper)
  Calculate properties with antiparticles as function of chemical potential.
- virtual void pair_density (fermion &f, double temper)
  Calculate properties with antiparticles as function of density.
- virtual void nu_from_n (fermion &f, double temper)
  Calculate effective chemical potential from density.
- int set_inte (**inte**< **funct** > &non_it, **inte**< **funct** > &deg_it)
  Set integrators.
- int set_density_root (**root**< **funct** > &rp)
  Set the solver for use in calculating the chemical potential from the density.
- virtual const char * type ()
  Return string denoting type ("rel_fermion")

Data Fields

- fermion unc
  Storage for the uncertainty.
- **cern_mroot_root**< **funct** > def_density_root
  The default solver for calc_density().
- **gsl_inte_qag**< **funct** > def_dit
  The default integrator for degenerate fermions.
- **gsl_inte_qagiu**< **funct** > def_nit
  The default integrator for non-degenerate fermions.

Numerical parameters

- double deg_limit
  The critical degeneracy at which to switch integration techniques (default 2)
- double exp_limit
  The limit for exponentials to ensure integrals are finite (default 200)
- double upper_limit_fac
  The factor for the degenerate upper limits (default 20)
- double deg_entropy_fac
  A factor for the degenerate entropy integration (default 30)

Protected Member Functions

- double density_fun (double u)
  The integrand for the density for non-degenerate fermions.
- double energy_fun (double u)
  The integrand for the energy density for non-degenerate fermions.
- double entropy_fun (double u)
  The integrand for the entropy density for non-degenerate fermions.
- double deg_density_fun (double u)
  The integrand for the density for degenerate fermions.
- double deg_energy_fun (double u)
  The integrand for the energy density for degenerate fermions.
- double deg_entropy_fun (double u)
  The integrand for the entropy density for degenerate fermions.
- double solve_fun (double x)
  Solve for the chemical potential given the density.
- double pair_fun (double x)
  Solve for the chemical potential given the density with antiparticles.

Protected Attributes

- **inte**< **funct** > ∗ nit
  The non-degenerate integrator.
- **inte**< **funct** > ∗ dit
  The degenerate integrator.
- **root**< **funct** > ∗ density_root
  The solver for calc_density()
- double T
  Temperature.
- fermion ∗ fp
  Current fermion pointer.

8.43.2 Member Function Documentation

8.43.2.1 virtual void rel_fermion::calc_density (**fermion** & *f*, double *temper*) [virtual]

This function uses the current value of `nu` (or `mu` if the particle is non interacting) for an initial guess to solve for the chemical potential. If this guess is too small, then this function may fail.

Implements fermion_eval_thermo.

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

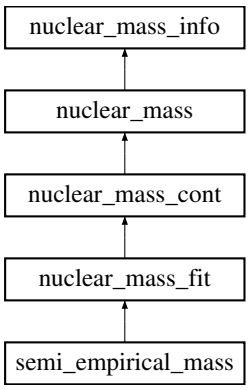
- rel_fermion.h

8.44 semi_empirical_mass Class Reference

Semi-empirical mass formula.

`#include <nuclear_mass.h>`

Inheritance diagram for semi_empirical_mass:



8.44.1 Detailed Description

A simple semi-empirical mass formula of the form

$$E/A = B + S_s \frac{1}{A^{1/3}} + E_c \frac{Z^2}{A^{4/3}} + S_v \left(1 - \frac{2Z}{A}\right)^2 + E_{\rm pair}(Z,N)$$

where the pairing energy is given by

$$E_{\rm pair}(Z,N) = -\frac{E_{\rm pair}}{2A^{3/2}} \left[\cos(\pi Z) + \cos(\pi N)\right]$$

which is equivalent to the traditional prescription

$$E_{\rm pair}(Z,N) = \frac{E_{\rm pair}}{A^{3/2}} \times \begin{cases} -1 & \text{N and Z even} \\ +1 & \text{N and Z odd} \\ 0 & \text{otherwise} \end{cases}$$

when Z and N and integers.

Note

The default parameters are arbitrary, and are not determined from a fit.

There is an example of the usage of this class given in Nuclear mass fit example.

Definition at line 405 of file nuclear_mass.h.

Public Member Functions

- virtual const char * type ()
  Return the type, "semi_empirical_mass".
- virtual double mass_excess_d (double Z, double N)
  Given Z and N, return the mass excess in MeV.
- virtual int fit_fun (size_t nv, const **ovector_base** &x)
  Fix parameters from an array for fitting.
- virtual int guess_fun (size_t nv, **ovector_base** &x)
  Fill array with guess from present values for fitting.

Data Fields

- double B
  Binding energy (negative and in MeV, default -16)
- double Sv
  Symmetry energy (in MeV, default 23.7)
- double Ss
  Surface energy (in MeV, default 18)
- double Ec
  Coulomb energy (in MeV, default 0.7)
- double Epair
  Pairing energy (MeV, default 13.0)

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

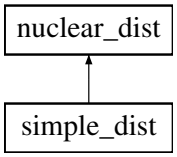
- nuclear_mass.h

8.45 simple_dist Class Reference

A simple nuclear distribution given a range in A and Z.

`#include <nuclear_dist.h>`

Inheritance diagram for simple_dist:



8.45.1 Detailed Description

The iterator for this distribution begins with the nucleus with the lowest Z and A, and increases A before incrementing Z and beginning again with the lowest A for that value of Z. In other words, it proceeds through all the isotopes of an element first, and then proceeds to the next element.

For example, to create a collection of isotopes of Carbon, Nitrogen and Oxygen using the most recent (2003) Atomic Mass - Evaluation, and then output the nuclei in the collection

```
ame_mass ame;
simple_dist fd(6,8,2,30,&ame);
for(nuclear_dist::iterator ndi=fd.begin();ndi!=fd.end();ndi++) {
  cout << ndi->Z << " " << ndi->A << " " << ndi->m << endl;
}
```

Idea for Future Make the vector constructor into a template so it accepts any type. Do the same for set_dist().

Definition at line 165 of file nuclear_dist.h.

Public Member Functions

- simple_dist ()
  Create an empty distribution.
- simple_dist (int minZ, int maxZ, int minA[ ], int maxA[ ], nuclear_mass &nm)
  Create a distribution from ranges in A specified for each Z.
- simple_dist (int minZ, int maxZ, int minA, int maxA, nuclear_mass &nm)
  Create a square distribution in A and Z.
- virtual iterator begin ()
  The beginning of the distribution.
- virtual iterator end ()
  The end of the distribution.
- virtual size_t size ()
  The number of nuclei in the distribution.
- int set_dist (int minZ, int maxZ, int minA[ ], int maxA[ ], nuclear_mass &nm)
  Set the distribution from ranges in A specified for each Z.
- int set_dist (int minZ, int maxZ, int minA, int maxA, nuclear_mass &nm)
  Set a square distribution in A and Z.

8.45.2 Constructor & Destructor Documentation

8.45.2.1 simple_dist::simple_dist (int *minZ*, int *maxZ*, int *minA[]*, int *maxA[]*, nuclear_mass & *nm*)

The length of the arrays minA and maxA should be exactly maxZ − minZ + 1.

8.45.3 Member Function Documentation

8.45.3.1 int simple_dist::set_dist (int *minZ*, int *maxZ*, int *minA[]*, int *maxA[]*, nuclear_mass & *nm*)

The length of the arrays minA and maxA should be exactly maxZ − minZ + 1.

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

- nuclear_dist.h

8.46 sn_classical Class Reference

Equation of state for a classical particle with derivatives.

`#include <sn_classical.h>`

8.46.1 Detailed Description

Todo This does not work with inc_rest_mass=true

Definition at line 42 of file sn_classical.h.

Public Member Functions

- virtual void calc_mu (part_deriv &p, double temper)
  Desc.
- virtual void calc_density (part_deriv &p, double temper)
  Desc.
- virtual const char * type ()
  Return string denoting type ("sn_classical")

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

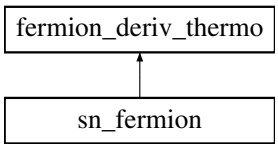
- sn_classical.h

8.47 sn_fermion Class Reference

Equation of state for a relativistic fermion.

`#include <sn_fermion.h>`

Inheritance diagram for sn_fermion:



8.47.1 Detailed Description

Note

This class does not work with inc_rest_mass=true. For example, the integration limits in calc_mu() need to be reworked for this case.

This implements an equation of state for a relativistic fermion using direct integration. After subtracting the rest mass from the chemical potentials, the distribution function is

$$\left\{1+\exp[(\sqrt{k^2+m^{*2}}-m-\nu)/T]\right\}^{-1}$$

where k is the momentum, ν is the effective chemical potential, m is the rest mass, and m^* is the effective mass. For later use, we define $E^*=\sqrt{k^2+m^{*2}}$. The degeneracy parameter is

$$\psi=(\nu+(m-m^*))/T$$

For ψ greater than deg_limit (degenerate regime), a finite interval integrator is used and for ψ less than deg_limit (non-degenerate regime), an integrator over the interval from $[0,\infty)$ is used. The upper limit on the degenerate integration is given by the value of the momentum k which is the solution of

$$(\sqrt{k^2+m^{*,2}}-m-\nu)/T=\text{flimit}$$

which is

$$\sqrt{(m+\mathscr{L})^2 - m^{*2}}$$

where $\mathscr{L} \equiv \text{flimit} \times T + \nu$.

For the entropy integration, we set the lower limit to

$$2\sqrt{\nu^2 + 2\nu m} - \text{upper limit}$$

since the only contribution to the entropy is at the Fermi surface.

In the non-degenerate regime, we make the substitution $u = k/T$ to ensure that the variable of integration scales properly.

Uncertainties are given in unc.

Todo This needs to be corrected to calculate $\sqrt{k^2 + m^{*2}} - m$ gracefully when $m^* \approx m << k$.

Call error handler if inc_rest_mass is true or update to properly treat the case when inc_rest_mass is true.

Evaluation of the derivatives

The relevant derivatives of the distribution function are

$$\frac{\partial f}{\partial T} = f(1-f)\frac{E^* - m - \nu}{T^2}$$

$$\frac{\partial f}{\partial \nu} = f(1-f)\frac{1}{T}$$

$$\frac{\partial f}{\partial k} = -f(1-f)\frac{k}{E^* T}$$

$$\frac{\partial f}{\partial m^*} = -f(1-f)\frac{m^*}{E^* T}$$

We also need the derivative of the entropy integrand w.r.t. the distribution function, which is

$$\mathscr{S} \equiv f \ln f + (1-f)\ln(1-f) \qquad \frac{\partial \mathscr{S}}{\partial f} = \ln\left(\frac{f}{1-f}\right) = \left(\frac{\nu - E^* + m}{T}\right)$$

where the entropy density is

$$s = -\frac{g}{2\pi^2}\int_0^\infty \mathscr{S} k^2 dk$$

The derivatives can be integrated directly (method = direct) or they may be converted to integrals over the distribution function through an integration by parts (method = byparts)

$$\int_a^b f(k)\frac{dg(k)}{dk}dk = f(k)g(k)|_{k=a}^{k=b} - \int_a^b g(k)\frac{df(k)}{dk}dk$$

using the distribution function for $f(k)$ and 0 and ∞ as the limits, we have

$$\frac{g}{2\pi^2}\int_0^\infty \frac{dg(k)}{dk} f dk = \frac{g}{2\pi^2}\int_0^\infty g(k) f(1-f)\frac{k}{E^* T}dk$$

as long as $g(k)$ vanishes at $k = 0$. Rewriting,

$$\frac{g}{2\pi^2}\int_0^\infty h(k) f(1-f)dk = \frac{g}{2\pi^2}\int_0^\infty f\frac{T}{k}\left[h' E^* - \frac{hE^*}{k} + \frac{hk}{E^*}\right]dk$$

as long as $h(k)/k$ vanishes at $k = 0$.

Explicit forms

1) The derivative of the density wrt the chemical potential

$$\left(\frac{dn}{d\mu}\right)_T = \frac{g}{2\pi^2}\int_0^\infty \frac{k^2}{T} f(1-f)dk$$

Using $h(k) = k^2/T$ we get

$$\left(\frac{dn}{d\mu}\right)_T = \frac{g}{2\pi^2}\int_0^\infty \left(\frac{k^2+E^{*2}}{E^*}\right) f dk$$

2) The derivative of the density wrt the temperature

$$\left(\frac{dn}{dT}\right)_\mu = \frac{g}{2\pi^2}\int_0^\infty \frac{k^2(E^*-m-\nu)}{T^2} f(1-f)dk$$

Using $h(k) = k^2(E^*-\nu)/T^2$ we get

$$\left(\frac{dn}{dT}\right)_\mu = \frac{g}{2\pi^2}\int_0^\infty \frac{f}{T}\left[2k^2+E^{*2}-E^*(\nu+m)-k^2\left(\frac{\nu+m}{E^*}\right)\right] dk$$

3) The derivative of the entropy wrt the chemical potential

$$\left(\frac{ds}{d\mu}\right)_T = \frac{g}{2\pi^2}\int_0^\infty k^2 f(1-f)\frac{(E^*-m-\nu)}{T^2} dk$$

This verifies the Maxwell relation

$$\left(\frac{ds}{d\mu}\right)_T = \left(\frac{dn}{dT}\right)_\mu$$

4) The derivative of the entropy wrt the temperature

$$\left(\frac{ds}{dT}\right)_\mu = \frac{g}{2\pi^2}\int_0^\infty k^2 f(1-f)\frac{(E^*-m-\nu)^2}{T^3} dk$$

Using $h(k) = k^2(E^*-\nu)^2/T^3$

$$\left(\frac{ds}{dT}\right)_\mu = \frac{g}{2\pi^2}\int_0^\infty \frac{f(E^*-m-\nu)}{E^*T^2}\left[E^{*3}+3E^*k^2-(E^{*2}+k^2)(\nu+m)\right] dk$$

5) The derivative of the density wrt the effective mass

$$\left(\frac{dn}{dm^*}\right)_{T,\mu} = -\frac{g}{2\pi^2}\int_0^\infty \frac{k^2 m^*}{E^*T} f(1-f)dk$$

Using $h(k) = -(k^2m^*)/(E^*T)$ we get

$$\left(\frac{dn}{dm^*}\right)_{T,\mu} = -\frac{g}{2\pi^2}\int_0^\infty m^* f dk$$

Note

The dsdT integration may fail if the system is very degenerate. When method is byparts, the integral involves a large cancellation between the regions from $k \in (0, \text{ulimit}/2)$ and $k \in (\text{ulimit}/2, \text{ulimit})$. Switching to method=direct and setting the lower limit to llimit, may help, but recent testing on this gave negative values for dsdT. For very degenerate systems, an expansion may be better than trying to perform the integration. The value of the integrand at k=0 also looks like it might be causing difficulties.

Idea for Future It might be worth coding up direct differentiation, or differentiating the eff results, as these may succeed more generally.

Idea for Future This class will have difficulty with extremely degenerate or extremely non-degnerate systems. Fix this.

Idea for Future Create a more intelligent method for dealing with bad initial guesses for the chemical potential in calc_density().

Definition at line 243 of file sn_fermion.h.

Public Member Functions

- sn_fermion ()
  Create a fermion with mass m and degeneracy g.
- virtual void calc_mu (fermion_deriv &f, double temper)
  Calculate properties as function of chemical potential.
- virtual void calc_density (fermion_deriv &f, double temper)
  Calculate properties as function of density.
- virtual void pair_mu (fermion_deriv &f, double temper)
  Calculate properties with antiparticles as function of chemical potential.
- virtual void pair_density (fermion_deriv &f, double temper)
  Calculate properties with antiparticles as function of density.
- virtual void nu_from_n (fermion_deriv &f, double temper)
  Calculate effective chemical potential from density.
- void set_inte (**inte**< **funct** > &unit, **inte**< **funct** > &udit)
  Set inte objects.
- void set_density_root (**root**< **funct** > &rp)
  Set the solver for use in calculating the chemical potential from the density.
- virtual const char * type ()
  Return string denoting type ("sn_fermion")
- double deriv_calibrate (fermion_deriv &f, int verbose, std::string fname="")
  Desc.

Data Fields

- double exp_limit
  Limit of arguments of exponentials for Fermi functions (default 200.0)
- double deg_limit
  The critical degeneracy at which to switch integration techniques (default 2.0)
- double upper_limit_fac
  The limit for the Fermi functions (default 20.0)
- fermion_deriv unc
  Storage for the most recently calculated uncertainties.
- **gsl_inte_qagiu**< **funct** > def_nit
  The default integrator for the non-degenerate regime.
- **gsl_inte_qag**< **funct** > def_dit
  The default integrator for the degenerate regime.
- **cern_mroot_root**< **funct** > def_density_root
  The default solver for npen_density() and pair_density()

Method of computing derivatives

- int method
  Method (default is byparts)
- static const int direct = 1
  In the form containing $f(1-f)$.
- static const int byparts = 2
  Integrate by parts.

8.47.2 Member Function Documentation

8.47.2.1 void sn_fermion::set_inte (**inte**< **funct** > & *unit*, **inte**< **funct** > & *udit*)

The first integrator is used for non-degenerate integration and should integrate from 0 to ∞ (like **gsl_inte_qagiu**). The second integrator is for the degenerate case, and should integrate between two finite values.

8.47.3 Field Documentation

8.47.3.1 double sn_fermion::upper_limit_fac

sn_fermion will ignore corrections smaller than about $\exp(-\text{flimit})$.

Definition at line 267 of file sn_fermion.h.

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

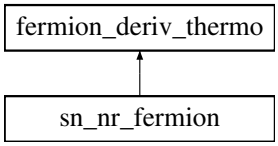
- sn_fermion.h

8.48 sn_nr_fermion Class Reference

Equation of state for a nonrelativistic fermion.

`#include <sn_nr_fermion.h>`

Inheritance diagram for sn_nr_fermion:



8.48.1 Detailed Description

This does not include the rest mass energy in the chemical potential or the rest mass energy density in the energy density to alleviate numerical precision problems at low densities

This implements an equation of state for a nonrelativistic fermion using direct integration. After subtracting the rest mass from the chemical potentials, the distribution function is

$$\left\{1+\exp\left[\left(\frac{k^2}{2m^*}-\nu\right)/T\right]\right\}^{-1}$$

where ν is the effective chemical potential, m is the rest mass, and m^* is the effective mass. For later use, we define $E^*=k^2/2/m^*$.

Uncertainties are given in unc.

Evaluation of the derivatives

The relevant derivatives of the distribution function are

$$\frac{\partial f}{\partial T}=f(1-f)\frac{E^*-\nu}{T^2}$$

$$\frac{\partial f}{\partial \nu}=f(1-f)\frac{1}{T}$$

$$\frac{\partial f}{\partial k}=-f(1-f)\frac{k}{m^*T}$$

$$\frac{\partial f}{\partial m^*}=f(1-f)\frac{k^2}{2m^{*2}T}$$

We also need the derivative of the entropy integrand w.r.t. the distribution function, which is quite simple

$$\mathscr{S}\equiv f\ln f+(1-f)\ln(1-f)\qquad \frac{\partial \mathscr{S}}{\partial f}=\ln\left(\frac{f}{1-f}\right)=\left(\frac{\nu-E^*}{T}\right)$$

where the entropy density is

$$s = -\frac{g}{2\pi^2}\int_0^\infty \mathscr{S} k^2 dk$$

The derivatives can be integrated directly or they may be converted to integrals over the distribution function through an integration by parts

$$\int_a^b f(k)\frac{dg(k)}{dk}dk = f(k)g(k)|_{k=a}^{k=b} - \int_a^b g(k)\frac{df(k)}{dk}dk$$

using the distribution function for $f(k)$ and 0 and ∞ as the limits, we have

$$\frac{g}{2\pi^2}\int_0^\infty \frac{dg(k)}{dk} f dk = \frac{g}{2\pi^2}\int_0^\infty g(k) f(1-f)\frac{k}{E^* T}dk$$

as long as $g(k)$ vanishes at $k = 0$. Rewriting,

$$\frac{g}{2\pi^2}\int_0^\infty h(k) f(1-f)dk = \frac{g}{2\pi^2}\int_0^\infty f\frac{T m^*}{k}\left[h' - \frac{h}{k}\right]dk$$

as long as $h(k)/k$ vanishes at $k = 0$.

Explicit forms

1) The derivative of the density wrt the chemical potential

$$\left(\frac{dn}{d\mu}\right)_T = \frac{g}{2\pi^2}\int_0^\infty \frac{k^2}{T} f(1-f)dk$$

Using $h(k) = k^2/T$ we get

$$\left(\frac{dn}{d\mu}\right)_T = \frac{g}{2\pi^2}\int_0^\infty m^* f dk$$

2) The derivative of the density wrt the temperature

$$\left(\frac{dn}{dT}\right)_\mu = \frac{g}{2\pi^2}\int_0^\infty \frac{k^2(E^*-\nu)}{T^2} f(1-f)dk$$

Using $h(k) = k^2(E^* - \nu)/T^2$ we get

$$\left(\frac{dn}{dT}\right)_\mu = \frac{g}{2\pi^2}\int_0^\infty \frac{f}{T}\left[m^*(E^*-\nu) - k^2\right]dk$$

3) The derivative of the entropy wrt the chemical potential

$$\left(\frac{ds}{d\mu}\right)_T = \frac{g}{2\pi^2}\int_0^\infty k^2 f(1-f)\frac{(E^*-\nu)}{T^2}dk$$

This verifies the Maxwell relation

$$\left(\frac{ds}{d\mu}\right)_T = \left(\frac{dn}{dT}\right)_\mu$$

4) The derivative of the entropy wrt the temperature

$$\left(\frac{ds}{dT}\right)_\mu = \frac{g}{2\pi^2}\int_0^\infty k^2 f(1-f)\frac{(E^*-\nu)^2}{T^3}dk$$

Using $h(k) = k^2(E^* - \nu)^2/T^3$

$$\left(\frac{ds}{dT}\right)_\mu = \frac{g}{2\pi^2}\int_0^\infty f\frac{m^*}{T^2}\left[(E^*-\nu)^2 + \frac{2k^2}{m^*}(E^*-\nu)\right]dk$$

5) The derivative of the density wrt the effective mass

$$\left(\frac{dn}{dm^*}\right)_{T,\mu} = \frac{g}{2\pi^2}\int_0^\infty \frac{k^2}{2m^{*2}T} f(1-f)k^2 dk$$

Using $h(k) = k^4/(2m^{*2}T)$ we get

$$\left(\frac{dn}{dm^*}\right)_{T,\mu} = \frac{g}{2\pi^2}\int_0^\infty f\frac{3k^2}{2m^*} dk$$

New section

$u = k^2/2/m^*/T$ and $y = \mu/T$, so

$$kdk = m^*Tdu$$

or

$$dk = \frac{m^*T}{\sqrt{2m^*Tu}}du = \sqrt{\frac{m^*T}{2u}}du$$

1) The derivative of the density wrt the chemical potential

$$\left(\frac{dn}{d\mu}\right)_T = \frac{gm^{*3/2}\sqrt{T}}{2^{3/2}\pi^2}\int_0^\infty u^{-1/2} f du$$

2) The derivative of the density wrt the temperature

$$\left(\frac{dn}{dT}\right)_\mu = \frac{gm^{*3/2}\sqrt{T}}{2^{3/2}\pi^2}\int_0^\infty f du\left[3u^{1/2} - yu^{-1/2}\right]$$

4) The derivative of the entropy wrt the temperature

$$\left(\frac{ds}{dT}\right)_\mu = \frac{gm^{*3/2}T^{1/2}}{2^{3/2}\pi^2}\int_0^\infty f\left[5u^{3/2} - 6yu^{1/2} + y^2u^{-1/2}\right] du$$

5) The derivative of the density wrt the effective mass

$$\left(\frac{dn}{dm^*}\right)_{T,\mu} = \frac{3gm*1/2T^{3/2}}{2^{3/2}\pi^2}\int_0^\infty u^{1/2} f du$$

Definition at line 221 of file sn_nr_fermion.h.

Public Member Functions

- sn_nr_fermion ()
  Create a fermion with mass m and degeneracy g.
- virtual void calc_mu (fermion_deriv &f, double temper)
  Calculate properties as function of chemical potential.
- virtual void calc_density (fermion_deriv &f, double temper)
  Calculate properties as function of density.
- virtual void pair_mu (fermion_deriv &f, double temper)
  Calculate properties with antiparticles as function of chemical potential.
- virtual void pair_density (fermion_deriv &f, double temper)
  Calculate properties with antiparticles as function of density.
- virtual void nu_from_n (fermion_deriv &f, double temper)
  Calculate effective chemical potential from density.
- void set_density_root (**root**< **funct** > &rp)
  Set the solver for use in calculating the chemical potential from the density.
- virtual const char * type ()
  Return string denoting type ("sn_nr_fermion")

Data Fields

- double flimit
  The limit for the Fermi functions (default 20.0)
- fermion_deriv unc
  Storage for the most recently calculated uncertainties.
- bool guess_from_nu
  If true, use the present value of the chemical potential as a guess for the new chemical potential.
- **cern_mroot_root**< **funct** > def_density_root
  The default solver for npen_density() and pair_density()

Protected Member Functions

- double solve_fun (double x)
  Function to compute chemical potential from density.
- double pair_fun (double x)
  Function to compute chemical potential from density when antiparticles are included.

Protected Attributes

- double T
  Desc.
- fermion_deriv * fp
  Desc.
- **root**< **funct** > * density_root
  Solver to compute chemical potential from density.

8.48.2 Field Documentation

8.48.2.1 double sn_nr_fermion::flimit

sn_nr_fermion will ignore corrections smaller than about $\exp(-\text{flimit})$.

Definition at line 234 of file sn_nr_fermion.h.

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

- sn_nr_fermion.h

8.49 thermo Class Reference

A class for the thermodynamical variables (energy density, pressure, entropy density)

`#include <part.h>`

8.49.1 Detailed Description

Definition at line 45 of file part.h.

Public Member Functions

- const char * type ()
  Return string denoting type ("thermo")

Data Fields

- double pr
  pressure
- double ed
  energy density
- double en
  entropy density

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

- part.h

9 File Documentation

9.1 hdf_nucmass_io.h File Reference

File for HDF input of the O_2scl ame_mass and mnmsk_mass data files.

`#include <hdf5.h> #include <hdf5_hl.h> #include <o2scl/constants.h> #include <o2scl/hdf_file.h> #include <o2scl/lib_settings.h> #include <o2scl/nuclear_mass.h>`

9.1.1 Detailed Description

Definition in file hdf_nucmass_io.h.

Functions

- int ame_load (ame_mass &ame, std::string version, string dir="")
  Read data for ame_mass from an HDF table.
- int mnmsk_load (mnmsk_mass &mnmsk, string dir="")
  Read data for mnmsk_mass from an HDF table.
- int hfb_load (hfb_mass &hfb, int model=14, string dir="")
  Read data for hfb_mass from an HDF table.
- int hfb_sp_load (hfb_sp_mass &hfb, int model=21, string dir="")
  Read data for hfb_mass from an HDF table.
- int **ame_load** (o2scl::ame_mass &ame, std::string version, std::string dir="")
- int **mnmsk_load** (o2scl::mnmsk_mass &mnmsk, std::string dir="")
- int **hfb_load** (o2scl::hfb_mass &hfb, int model=14, std::string dir="")
- int **hfb_sp_load** (o2scl::hfb_sp_mass &hfb, int model=21, std::string dir="")

9.1.2 Function Documentation

9.1.2.1 int hfb_load (hfb_mass & *hfb*, int *model* = `14`, string *dir* = " ")

Valid values of `model` at present are 2, 8, and 14, corresponding to the HFB2 (Goriely02), HFB8 (Samyn04), and HFB14 (-Goriely07). If a number other than these three is given, the error handler is called.

9.1.2.2 int hfb_sp_load (hfb_sp_mass & *hfb*, int *model* = `21`, string *dir* = " ")

Valid values of `model` at present are 17 and 21, corresponding to the HFB17 (Goriely02) and HFB21 (Samyn04). If a number other than these two is given, the error handler is called.

9.2 part.h File Reference

File for definitions for thermo and part.

`#include <string> #include <iostream> #include <cmath> #include <o2scl/constants.h> #include <o2scl/inte.h> #include <o2scl/funct.h> #include <o2scl/mroot.h>`

9.2.1 Detailed Description

Definition in file part.h.

Data Structures

- class thermo
  A class for the thermodynamical variables (energy density, pressure, entropy density)
- class part
  Particle base class.

Functions

- thermo operator+ (const thermo &left, const thermo &right)
  Addition operator.
- thermo operator- (const thermo &left, const thermo &right)
  Subtraction operator.
- thermo operator+ (const thermo &left, const part &right)
  Addition operator.
- thermo operator- (const thermo &left, const part &right)
  Subtraction operator.