O_2scl_part - Particle Sub-Library for O_2scl

Version 0.805

Contents

1 Main Page

1.1 User's guide

- Particles
- Atomic nuclei
- Example source code
- Bibliography

1.2 Particles

These classes in the library `o2scl_part` calculate the thermodynamic properties of interacting and non-interacting quantum and classical particles.

The class part is the basic structure for a particle:

- m - mass
- g - degeneracy factor (e.g. $2j+1$)
- n - number density
- ns - scalar number density
- ed - energy density
- pr - pressure
- en - entropy
- ms - effective mass
- nu - effective chemical potential
- bool inc_rest_mass - True if the rest mass is included
- bool non_interacting - False if the particle includes interactions
- std::string name - a 'name' field to distinguish different particles of the same class

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.

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

Typically, if the particle is non-interacting, then `mu` and `m` are copied to `nu` and `ms`, computations are performed with `nu` and `ms`, and then, if necessary, the result for `nu` is copied back to `mu`.

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 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 `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 `o2scl_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 `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. quark is a descendant of the fermion class which contains extra data members for the quark condensate and the contribution to the bag constant. 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 is used in rel_boson and rel_fermion, but these are necessarily quite a bit slower.

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 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 consistent with the above, i.e. $[n] = [T]^3 = [m]^3 = [P]^{4/3}$, etc.

1.3 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 mnmsk_mass provides the mass formula from Moller95, and the class ame_mass provides the mass formula from Audi95 or Audi03. 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.

1.4 Example source code

Example list

- Particle example

1.4.1 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);

  // Create two different electrons, one using the exact method from
  // rel_fermion, and the other from the 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 compute
  eff_fermion e(o2scl_fm::mass_electron,2.0);
  rel_fermion e2(o2scl_fm::mass_electron,2.0);
  classical 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;
  e.calc_density(10.0/o2scl_const::hc_mev_fm);
  e2.n=0.0001;
  e2.calc_density(10.0/o2scl_const::hc_mev_fm);
  e3.n=0.0001;
  e3.calc_density(10.0/o2scl_const::hc_mev_fm);

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

  // 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;
  e.calc_density(10.0/o2scl_const::hc_mev_fm);
  e2.n=0.0001;
  e2.calc_density(10.0/o2scl_const::hc_mev_fm);
  cout << e.pr << " " << e2.pr << endl;

  // Now add the contribution to the pressure from positrons using the
  // implmentation of part::pair_density()
  e.n=0.0001;
  e.pair_density(10.0/o2scl_const::hc_mev_fm);
  e2.n=0.0001;
  e2.pair_density(10.0/o2scl_const::hc_mev_fm);
  cout << e.pr << " " << e2.pr << endl;

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

1.5 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. A729 (2003) 337.

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

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

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

2 Ideas for future development

Class ame_mass 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.

Class ame_mass Use the atomic mass unit and other constants defined in the evaulation

Class eff_fermion Use bracketing to speed up one-dimensional **root** finding.

Class fermion Consider putting a parent version of calc_e and calc_p, or in part or fermion which automatically solves like eff_fermion::calc_density()?

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

Class nuclear_mass It might be useful to consider a fudge factor to ensure no problems with finite precision arithmetic when converting `double` to `int`.

Global nuclear_mass::parse_elstring(std::string ela, int &Z, int &N, int &A) Properly ignore dashes and other non-alphanumeric characters.

Class rel_fermion Allow the user to change the upper limit on the degenerate integration and the hard-coded value of 200 in the integrands.

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

3 Bug List

Global classical::pair_mu(const double temper) This function probably doesn't work (3/6/08). See end of eff_fermion_ts.cpp.

Global classical::pair_density(const double temper) This function probably doesn't work (3/6/08). See end of eff_fermion_ts.cpp.

4 Todo List

Class eff_boson Better documentation (see eff_fermion)

Class eff_boson Remove native error codes

Global eff_fermion::calc_mu(const double temper) Should see if the function actually works if $(\mu - m)/T = -199$.

Class eff_quark Add testing.

Global fermion::massless_pair_density(const double temper) Comment here about the precision of the expansions and allow the user to control how they are used if necessary.

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

Class nonrel_fermion I think calc_mu_zerot() and calc_density_zerot() are missing the proper dependence on the degeneracy, g. (8/20/07) (I think this is fixed now, but should be tested, 8/22/07)

Class nonrel_fermion 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 nuclear_mass Make the treatment of the electron binding energy contribution more consistent.

Class rel_boson Testing not completely finished.

Class simple_dist This takes a nuclear_mass pointer as input. It should probably be a reference instead? (3/18/08 - Yeah, I think it should be a reference.)

Class simple_dist Add error checking in constructors and set functions

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 $m^* \approx m$.

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

Class sn_fermion This class will have difficulty with extremely degenerate or extremely non-degnerate systems.

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

5 Data Structure Documentation

5.1 ame_entry Struct Reference

`#include <nuclear_mass.h>`

5.1.1 Detailed Description

Atomic mass entry structure.

Definition at line 444 of file nuclear_mass.h.

Data Fields

- int NMZ
  N-Z.
- int N
  Neutron number.
- int Z
  Proton number.
- int A
  Atomic number.
- std::string el
  Element name.
- std::string orig
  Data origin.
- double mass
  Mass excess.
- double dmass
  Mass excess uncertainty.
- double be
  Binding energy (given in the '95 data).
- double dbe
  Binding energy uncertainty (given in the '95 data).
- double beoa
  Binding energy / A (given in the '03 data).
- double dbeoa
  Binding energy / A uncertainty (given in the '03 data).
- std::string bdmode
  Beta decay mode.
- double bde
  Beta-decay energy.
- double dbde
  Beta-decay energy uncertainty.
- int A2
  ?
- double amass
  Atomic mass.
- double damass
  Atomic mass uncertainty.

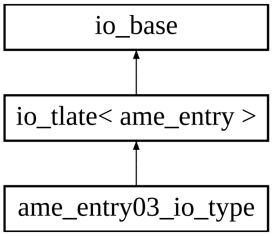
The documentation for this struct was generated from the following file:

- nuclear_mass.h

5.2 ame_entry03_io_type Class Reference

`#include <nuclear_mass.h>`

Inheritance diagram for ame_entry03_io_type::



5.2.1 Detailed Description

A support class for I/O of the 2003 AME data.

Definition at line 513 of file nuclear_mass.h.

Public Member Functions

- int **input** (**cinput** ∗co, **in_file_format** ∗ins, ame_entry ∗t)
- int **output** (**coutput** ∗co, **out_file_format** ∗outs, ame_entry ∗t)
- virtual const char ∗ **type** ()

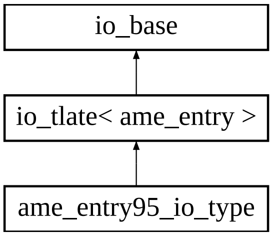
The documentation for this class was generated from the following file:

- nuclear_mass.h

5.3 ame_entry95_io_type Class Reference

`#include <nuclear_mass.h>`

Inheritance diagram for ame_entry95_io_type::



5.3.1 Detailed Description

A support class for I/O of the 1995 AME data.

Definition at line 504 of file nuclear_mass.h.

Public Member Functions

- int **input** (**cinput** ∗co, **in_file_format** ∗ins, ame_entry ∗t)
- int **output** (**coutput** ∗co, **out_file_format** ∗outs, ame_entry ∗t)
- virtual const char ∗ **type** ()

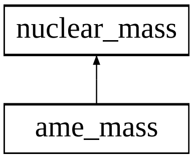
The documentation for this class was generated from the following file:

- nuclear_mass.h

5.4 ame_mass Class Reference

`#include <nuclear_mass.h>`

Inheritance diagram for ame_mass::



5.4.1 Detailed Description

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

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

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.

Note that blank entries in the original **table** that correspond to columns represented by the type `double` are set to zero arbitrarily.

Note that all uncertainties are 1 sigma uncertainties.

Warning:

> There are strict 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.

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

Use the atomic mass unit and other constants defined in the evaulation

Definition at line 563 of file nuclear_mass.h.

Public Member Functions

- ame_mass (std::string version="")
  Create a **collection** *specified by* `version`*.*
- 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 the constructor succesfully loaded the data.

Data Fields

- int n
  The number of entries (about 3000).
- std::string ∗ short_names
  The short names of the columns (length 16).
- std::string ∗ col_names
  The long names of the columns (length 16).
- std::string reference
  The reference for the original data.
- ame_entry ∗ mass
  The array containing the mass data of length ame::n.

Protected Attributes

- bool loaded
  True if loading the data was successful.

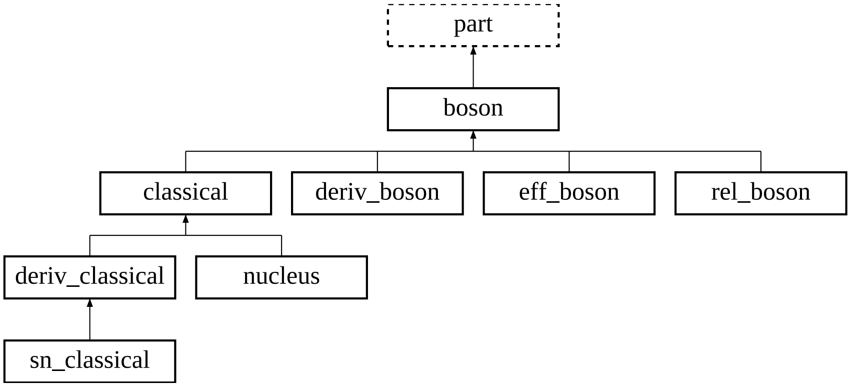
The documentation for this class was generated from the following file:

- nuclear_mass.h

5.5 boson Class Reference

`#include <boson.h>`

Inheritance diagram for boson::



5.5.1 Detailed Description

Boson class.

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 int calc_mu (const double temper)
  Calculate properties as function of chemical potential.
- virtual int calc_density (const double temper)
  Calculate properties as function of density.
- virtual int pair_mu (const double temper)
  Calculate properties with antiparticles as function of chemical potential.
- virtual int pair_density (const double temper)
  Calculate properties with antiparticles as function of density.
- virtual int massless_calc_mu (const double temper)
  Calculate properties of massless bosons.
- virtual const char ∗ type ()
  Return string denoting type ("boson").

Data Fields

- double co
  The condensate.

5.5.2 Member Function Documentation

5.5.2.1 virtual int massless_calc_mu (const double *temper*) `[virtual]`

Calculate properties of massless bosons.

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

5.5.3 Field Documentation

5.5.3.1 double co

The condensate.

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

Definition at line 76 of file boson.h.

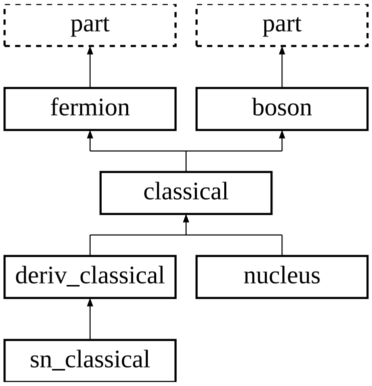
The documentation for this class was generated from the following file:

- boson.h

5.6 classical Class Reference

`#include <classical.h>`

Inheritance diagram for classical::



5.6.1 Detailed Description

Classical particle class.

Classical particles, because they inherit from both fermion and boson, have a fermi momentum, and a condensate, but these variables are ignored.

Definition at line 48 of file classical.h.

Public Member Functions

- classical (double m=0.0, double g=0.0)
  Create a classical particle with mass `m` *and degeneracy* `g`*.*
- virtual int calc_mu (const double temper)
  Calculate properties as function of chemical potential.

- virtual int calc_density (const double temper)
  Calculate properties as function of density.
- virtual int pair_mu (const double temper)
  Calculate properties with antiparticles as function of chemical potential.
- virtual int pair_density (const double temper)
  Calculate properties with antiparticles as function of density.
- virtual const char $*$ type ()
  Return string denoting type ("classical").

5.6.2 Member Function Documentation

5.6.2.1 virtual int pair_mu (const double *temper*) `[virtual]`

Calculate properties with antiparticles as function of chemical potential.

Bug

This function probably doesn't work (3/6/08). See end of eff_fermion_ts.cpp.

Reimplemented from boson.

Reimplemented in sn_classical.

5.6.2.2 virtual int pair_density (const double *temper*) `[virtual]`

Calculate properties with antiparticles as function of density.

Bug

This function probably doesn't work (3/6/08). See end of eff_fermion_ts.cpp.

Reimplemented from boson.

Reimplemented in sn_classical.

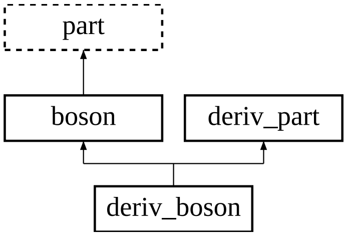
The documentation for this class was generated from the following file:

- classical.h

5.7 deriv_boson Class Reference

`#include <deriv_part.h>`

Inheritance diagram for deriv_boson::



5.7.1 Detailed Description

Boson with derivatives.

Definition at line 162 of file deriv_part.h.

Public Member Functions

- **deriv_boson** (double mass, double dof)

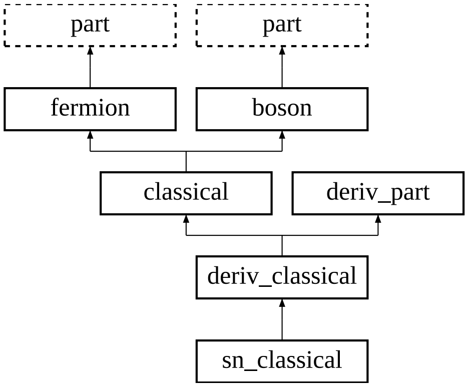
The documentation for this class was generated from the following file:

- deriv_part.h

5.8 deriv_classical Class Reference

`#include <deriv_part.h>`

Inheritance diagram for deriv_classical::



5.8.1 Detailed Description

Classical particle with derivatives.

Definition at line 170 of file deriv_part.h.

Public Member Functions

- **deriv_classical** (double mass, double dof)

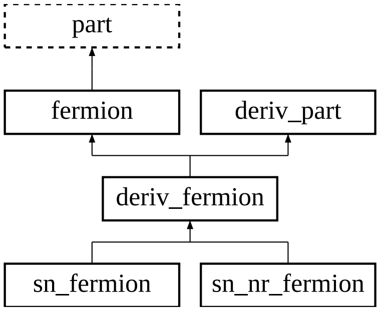
The documentation for this class was generated from the following file:

- deriv_part.h

5.9 deriv_fermion Class Reference

`#include <deriv_part.h>`

Inheritance diagram for deriv_fermion::



5.9.1 Detailed Description

Fermion with derivatives.

Definition at line 153 of file deriv_part.h.

Public Member Functions

- **deriv_fermion** (double mass, double dof)

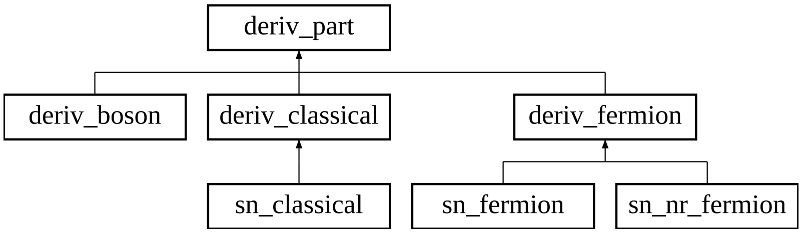
The documentation for this class was generated from the following file:

- deriv_part.h

5.10 deriv_part Class Reference

`#include <deriv_part.h>`

Inheritance diagram for deriv_part::



5.10.1 Detailed Description

Storage for deriviatives wrt μ and T.

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

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

respectively.

All other derivatives can be expressed simply in terms of these three.

Derivatives wrt to chemical potential and temperature:

There is a Maxwell relation

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

The pressure derivatives are trivial

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

The energy density derivatives are related through the thermodynamic identity:

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

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

Other derivatives:

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

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

To compute the specific heat 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

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

We can then construct a function

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

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

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

and the Maxwell relation above to give

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

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

Note that this is the specific heat per particle, and has no units. If specific heat per unit volume is required, you must multiply by the number density.

Definition at line 132 of file deriv_part.h.

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.

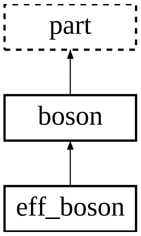
The documentation for this class was generated from the following file:

- deriv_part.h

5.11 eff_boson Class Reference

`#include <eff_boson.h>`

Inheritance diagram for eff_boson::



5.11.1 Detailed Description

Boson class from fitting method.

The constructor loads the coefficients from the file `boselat3` by default. If this is not successful, then is_loaded() will return false.

Todo

Better documentation (see eff_fermion)

Todo

Remove native error codes

Definition at line 59 of file eff_boson.h.

Public Member Functions

- eff_boson (double m=0.0, double g=0.0)
  Create a boson with mass `m` *and degeneracy* `g`*.*
- virtual int calc_mu (const double temper)
  Calculate properties as function of chemical potential.
- virtual int calc_density (const double temper)
  Calculate properties as function of density.
- virtual int pair_mu (const double temper)
  Calculate properties with antiparticles as function of chemical potential.
- virtual int pair_density (const double temper)
  Calculate properties with antiparticles as function of density.
- int set_psi_root (**root**< void ∗, **funct**< void ∗ > > &rp)
  Set the solver for use in calculating ψ.
- int set_density_mroot (**mroot**< void ∗, **mm_funct**< void ∗ > > &rp)
  Set the solver for use in calculating the chemical potential from the density.

- int set_meth2_root (**root**< void ∗, **funct**< void ∗ > > &rp)
  Set the solver for use in calculating the chemical potential from the density (meth2=true).
- virtual const char ∗ type ()
  Return string denoting type ("boson").

Static Public Member Functions

- static int loadcoeff (std::string bfile)
  Load coefficients for finite-temperature approximation.

Data Fields

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

Protected Member Functions

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

Protected Attributes

- **mroot**< void ∗, **mm_funct**< void ∗ > > ∗ density_mroot
  The solver for calc_density().
- **root**< void ∗, **funct**< void ∗ > > ∗ psi_root
  The solver to compute h from ψ.
- **root**< void ∗, **funct**< void ∗ > > ∗ meth2_root
  The solver for calc_density().

Static Protected Attributes

- static double ∗∗ Pmnb
  The coefficients.
- static int sizem
  The number of coefficient rows.
- static int sizen
  The number of coefficient columns.
- static double parma
  The parameter, a.
- static double fix_density
  Temporary storage.
- static double stat_temper
  Temporary storage.
- static bool loaded
  True if coefficients have been loaded.

5.11.2 Member Function Documentation

5.11.2.1 static int loadcoeff (std::string *bfile*) `[static]`

Load coefficients for finite-temperature approximation.

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

boselat3

```
double a 1.038
double[][] Pmn 4 4
1.68123 5.17553 5.66067 2.16447
6.72492 20.70212 22.64268 8.65788
8.51035 27.1555 30.5655 11.8288
3.47086 11.7253 13.8574 5.53865
```

bosejel21

```
double a 0.978
double[][] Pmn 3 2
1.63146 2.11571
4.89438 6.34713
3.31275 5.15372
```

bosejel22

```
double a 0.914
double[][] Pmn 3 3
1.68131 3.47558 2.16582
5.04393 10.42674 6.49746
3.25053 7.82859 5.19126
```

bosejel34

```
double a 1.029
double[][] Pmn 4 5
1.68134 6.85070 10.8537 7.81843 2.16461
6.72536 27.4028 43.4148 31.2737 8.65844
8.49651 35.6058 57.7134 42.3593 11.8199
3.45614 15.1152 25.5254 19.2745 5.51757
```

bosejel34cons

```
double a 1.040
double[][] Pmn 4 5
1.68130 6.85060 10.8539 7.81762 2.16465
6.72520 27.40240 43.4156 31.27048 8.65860
8.51373 35.6576 57.7975 42.4049 11.8321
3.47433 15.1995 25.6536 19.3811 5.54423
```

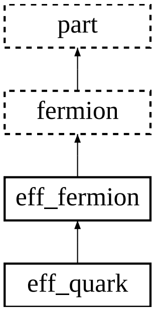
The documentation for this class was generated from the following file:

- eff_boson.h

5.12 eff_fermion Class Reference

```
#include <eff_fermion.h>
```

Inheritance diagram for eff_fermion::



5.12.1 Detailed Description

Fermion class from fitting method.

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

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

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

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

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.

- We can use the density to solve for f
- We can 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. The density is solved for the effective chemical potential instead. The initial guess is just taken from the present value of part::nu.

The constructor uses the coefficients from the file `fermilat3` by default.

It is important that if the **inte** and **root** objects are not thread safe, different instances should not use the same instance of a **inte** or **root** object simultaneously.

Idea for future

Use bracketing to speed up one-dimensional **root** finding.

Definition at line 94 of file eff_fermion.h.

Load coefficients for finite-temperature approximation

`ctype` Should be one of the constants below: cf_fermilat3, cf_fermijel2, cf_fermijel3, or cf_fermijel3cons.

Todo

Document these coefficients better

- static const int **cf_fermilat3** = 1
- static const int **cf_fermijel2** = 2
- static const int **cf_fermijel3** = 3
- static const int **cf_fermijel3cons** = 4
- static int **load_coefficients** (int ix)

Public Member Functions

- eff_fermion (double mass=0.0, double dof=0.0)
  Create a fermion with mass `mass` *and degeneracy* `dof`.
- virtual int calc_mu (const double temper)
  Calculate thermodynamic properties as function of chemical potential.
- virtual int calc_density (const double temper)
  Calculate thermodynamic properties as function of density.
- virtual int pair_mu (const double temper)
  Calculate thermodynamic properties with antiparticles as function of chemical potential.
- virtual int pair_density (const double temper)
  Calculate thermodynamic properties with antiparticles as function of density.
- int set_psi_root (**root**< void ∗, **funct**< void ∗ > > &rp)
  Set the solver for use in calculating ψ.
- int set_density_root (**root**< void ∗, **funct**< void ∗ > > &rp)
  Set the solver for use in calculating the chemical potential from the density with meth2=true.
- 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**< void ∗, **funct**< void ∗ > > def_psi_root
  The default solver for ψ.
- **cern_mroot_root**< void ∗, **funct**< void ∗ > > 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

- int solve_fun (double x, double &y, void ∗&pa)
  The function which solves for f *from* ψ.
- int density_fun (double x, double &y, void ∗&pa)
  Fix density for calc_density().
- int pair_density_fun (double x, double &y, void ∗&pa)
  Fix density for pair_density().

Protected Attributes

- **root**$<$ void $*$, **funct**$<$ void $*>>*$ psi_root
  The solver for ψ.
- **root**$<$ void $*$, **funct**$<$ void $*>>*$ density_root
  The other solver for calc_density().

Static Protected Attributes

- static double $**$ Pmnf
  The matrix of coefficients.
- static double parma
  The parameter a.
- static int sizem
  The array row size.
- static int sizen
  The array column size.

5.12.2 Member Function Documentation

5.12.2.1 virtual int calc_mu (const double *temper*) `[virtual]`

Calculate thermodynamic properties as function of chemical potential.

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 $(\mu-m)/T=-199$.

Reimplemented from fermion.

Reimplemented in eff_quark.

5.12.2.2 virtual int calc_density (const double *temper*) `[virtual]`

Calculate thermodynamic properties as function of density.

Warning:

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

Reimplemented from fermion.

Reimplemented in eff_quark.

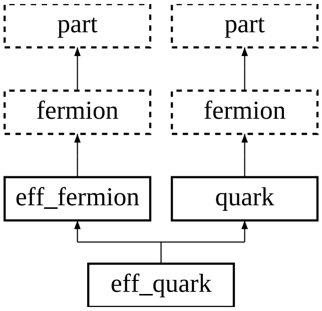
The documentation for this class was generated from the following file:

- eff_fermion.h

5.13 eff_quark Class Reference

`#include <eff_quark.h>`

Inheritance diagram for eff_quark::



5.13.1 Detailed Description

Quark class from fitting method.

Todo

Add testing.

Definition at line 45 of file eff_quark.h.

Public Member Functions

- eff_quark (double m=0.0, double g=0.0)
  Create a quark with mass `m` *and degeneracy* `g`*.*
- virtual int calc_mu (const double temper)
  Calculate thermodynamic properties as function of chemical potential.
- virtual int calc_density (const double temper)
  Calculate thermodynamic properties as function of density.
- virtual int pair_mu (const double temper)
  Calculate thermodynamic properties with antiparticles as function of chemical potential.
- virtual int pair_density (const double temper)
  Calculate thermodynamic properties with antiparticles as function of density.
- virtual const char $*$ type ()
  Return string denoting type ("eff_quark").

5.13.2 Member Function Documentation

5.13.2.1 virtual int calc_mu (const double *temper*) `[virtual]`

Calculate thermodynamic properties as function of chemical potential.

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 $(\mu - m)/T = -199$.

Reimplemented from eff_fermion.

5.13.2.2 virtual int calc_density (const double *temper*) `[virtual]`

Calculate thermodynamic properties as function of density.

Warning:

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

Reimplemented from eff_fermion.

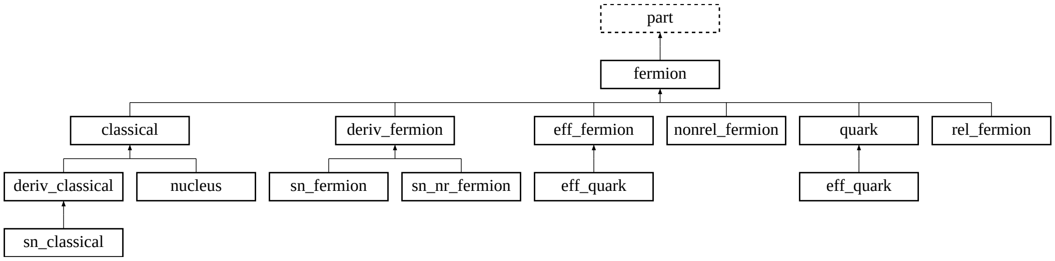
The documentation for this class was generated from the following file:

- eff_quark.h

5.14 fermion Class Reference

`#include <fermion.h>`

Inheritance diagram for fermion::



5.14.1 Detailed Description

Fermion class.

This is a base class for the computation of fermionic thermodynamics. This class includes the computations of zero-temperature (possibly massive) and massless fermions at zero or finite temperature. The more general case of fermions with both finite mass and a finite temperature are taken care of by the functions calc_mu(), calc_density(), pair_mu(), and pair_density(). These functions are not implemented in this base class (see eff_fermion, rel_fermion, nonrel_fermion, sn_fermion, and sn_nr_fermion).

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

> Consider putting a parent version of calc_e and calc_p, or in part or fermion which automatically solves like eff_fermion::calc_density()?

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 86 of file fermion.h.

Public Member Functions

- fermion (double mass=0, double dof=0)
  Create a fermion with mass `mass` *and degeneracy* `dof`.

- virtual int calc_mu (const double temper)
  Calculate properties as function of chemical potential.
- virtual int calc_density (const double temper)
  Calculate properties as function of density.
- virtual int pair_mu (const double temper)
  Calculate properties with antiparticles as function of chemical potential.
- virtual int pair_density (const double temper)
  Calculate properties with antiparticles as function of density.
- int set_massless_root (**root**< void ∗, **funct**< void ∗ > > &rp)
  Set the solver for use in massless_root_density().
- double deg_specific_heat (double T)
  Degenerate expansion for specific heat.
- virtual const char ∗ type ()
  Return string denoting type ("fermion").

Zero-temperature fermions

- int kffromden ()
  Calculate the Fermi momentum from the density.
- int sden ()
  Scalar number density at T=0 from kf and ms.
- int eden ()
  Energy density at T=0 from kf and ms.
- int pres ()
  Pressure at T=0 from kf and ms.
- virtual int calc_mu_zerot ()
  Zero temperature fermions from nu and ms.
- virtual int calc_density_zerot ()
  Zero temperature fermions from n and ms.

Massless fermions

- virtual int massless_calc_mu (const double temper)
  Finite temperature massless fermions.
- virtual int massless_calc_density (const double temper)
  Finite temperature massless fermions.
- int massless_pair_mu (const double temper)
  Finite temperature massless fermions and antifermions.
- int massless_pair_density (const double temper)
  Finite temperature massless fermions and antifermions.

Data Fields

- double kf
  Fermi momentum.
- double del
  Gap.
- **cern_mroot_root**< void ∗, **funct**< void ∗ > > def_massless_root
  The default solver for massless_calc_mu().

5.14.2 Member Function Documentation

5.14.2.1 int kffromden ()

Calculate the Fermi momentum from the density.

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

5.14.2.2 int sden ()

Scalar number density at T=0 from kf and ms.

Calculates the integral

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

5.14.2.3 int eden ()

Energy density at T=0 from kf and ms.

Calculates the integral

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

5.14.2.4 int pres ()

Pressure at T=0 from kf and ms.

Calculates the integral

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

5.14.2.5 virtual int calc_mu_zerot () `[virtual]`

Zero temperature fermions from nu and ms.

This function always returns `gsl_success`.

Reimplemented in nonrel_fermion.

5.14.2.6 virtual int calc_density_zerot () `[virtual]`

Zero temperature fermions from n and ms.

This function always returns `gsl_success`.

Reimplemented in nonrel_fermion.

5.14.2.7 int massless_pair_density (const double *temper*)

Finite temperature massless fermions and antifermions.

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

Todo

> Comment here about the precision of the expansions and allow the user to control how they are used if necessary.

5.14.2.8 double deg_specific_heat (double *T*) `[inline]`

Degenerate expansion for specific heat.

This is a temporary location and is also unchecked.

Definition at line 212 of file fermion.h.

5.14.3 Field Documentation

5.14.3.1 cern_mroot_root<void *,funct<void *> > def_massless_root

The default solver for massless_calc_mu().

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

Definition at line 232 of file fermion.h.

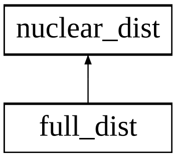
The documentation for this class was generated from the following file:

- fermion.h

5.15 full_dist Class Reference

`#include <nuclear_dist.h>`

Inheritance diagram for full_dist::



5.15.1 Detailed Description

Full distribution including all nuclei from a discrete mass formula.

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 264 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.

5.15.2 Member Function Documentation

5.15.2.1 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`.

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

5.16 mass_fit Class Reference

`#include <mass_fit.h>`

5.16.1 Detailed Description

Fit a nuclear mass formula.

Todo

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

Definition at line 43 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**< void *, **multi_funct**< void * > > &umm)
  Change the minimizer for use in the fit.
- int set_dist (nuclear_dist &und)
  Set the distribution of nuclei to fit.
- int set_masses (nuclear_mass &uexp)
  Set the experimental values to fit to.

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_simp**< void *, **multi_funct**< void * > > 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.

5.16.2 Field Documentation

5.16.2.1 gsl_mmin_simp<void *,multi_funct<void *> > def_mmin

The default minimizer.

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 74 of file mass_fit.h.

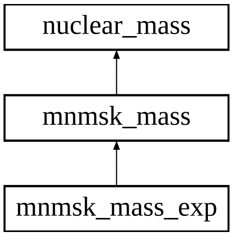
The documentation for this class was generated from the following file:

- mass_fit.h

5.17 mnmsk_mass Class Reference

`#include <nuclear_mass.h>`

Inheritance diagram for mnmsk_mass::



5.17.1 Detailed Description

Mass formula from Moller, Nix, Myers and Swiatecki.

The data containing an object of type moller_mass_entry for 8979 nuclei is automatically loaded by the constructor. If the file (nndc/moller.dat) 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 761 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.
- bool is_loaded ()
  Verify that the constructor properly loaded the **table**.
- 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}.

Data Fields

- int n
  The number of **table** *entries.*
- mnmsk_mass_entry ∗ mass
  The array containing the **table**.

Protected Attributes

- bool loaded
  True if the **table** *was successfully loaded.*
- int last
  The last **table** *index for caching.*

5.17.2 Member Function Documentation

5.17.2.1 mnmsk_mass_entry get_ZN (int *l_Z*, int *l_N*)

Get the entry for the specified proton and neutron number.

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

5.18 mnmsk_mass_entry Struct Reference

`#include <nuclear_mass.h>`

5.18.1 Detailed Description

Mass formula entry structure for Moller, et al.

Definition at line 627 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.
- std::string spinp
  Spin and pairity of odd proton.
- std::string spinn
  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.

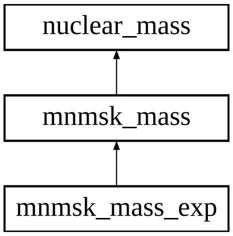
The documentation for this struct was generated from the following file:

- nuclear_mass.h

5.19 mnmsk_mass_exp Class Reference

`#include <nuclear_mass.h>`

Inheritance diagram for mnmsk_mass_exp::



5.19.1 Detailed Description

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

Definition at line 831 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.*

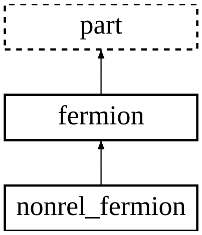
The documentation for this class was generated from the following file:

- nuclear_mass.h

5.20 nonrel_fermion Class Reference

`#include <nonrel_fermion.h>`

Inheritance diagram for nonrel_fermion::



5.20.1 Detailed Description

Nonrelativistic fermion class.

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's "Thermodynamics and an introduction to thermostatistics", 2nd edition, pg. 403 or Landau and Lifshitz, Stat. Phys. 3rd edition, part 1, pg. 164.

Note that the energy density integral can be rescaled:

$$\varepsilon = \frac{T^{5/2}(2m^*)^{3/2}}{\pi^2} \int_0^\infty du \frac{u^{3/2}}{1+\exp(u-y)}$$

where $u = k^2/2/m^*/T$ and $y = \mu/T$.

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

Todo

I think calc_mu_zerot() and calc_density_zerot() are missing the proper dependence on the degeneracy, `g`. (8/20/07) (I think this is fixed now, but should be tested, 8/22/07)

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 to invert the density integral so that we don't need to use a solver.

Definition at line 83 of file nonrel_fermion.h.

Public Member Functions

- nonrel_fermion (double m=0.0, double g=0.0)
  Create a nonrelativistic fermion with mass 'm' and degeneracy 'g'.
- virtual int calc_mu_zerot ()
  Zero temperature fermions.
- virtual int calc_density_zerot ()
  Zero temperature fermions.

- virtual int calc_mu (const double temper)
  Calculate properties as function of chemical potential.
- virtual int calc_density (const double temper)
  Calculate properties as function of density.
- virtual int nu_from_n (const double temper)
  Calculate effective chemical potential from density.
- int set_density_root (**root**< double, **funct**< double > > &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**< double, **funct**< double > > def_density_root
  The default solver for calc_density().

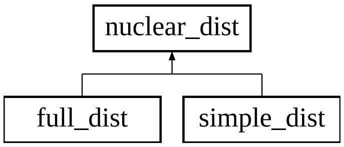
The documentation for this class was generated from the following file:

- nonrel_fermion.h

5.21 nuclear_dist Class Reference

`#include <nuclear_dist.h>`

Inheritance diagram for nuclear_dist::



5.21.1 Detailed Description

A distribution of nuclei.

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 41 of file nuclear_dist.h.

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.

Data Structures

- class iterator
  An iterator for the nuclear distribution.

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

- nuclear_dist.h

5.22 nuclear_dist::iterator Class Reference

`#include <nuclear_dist.h>`

5.22.1 Detailed Description

An iterator for the nuclear distribution.

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 70 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
  Dereference the iterator.

Protected Attributes

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

Friends

- int operator== (const nuclear_dist::iterator &i1, const nuclear_dist::iterator &i2)
  Compare two nuclei.
- int operator!= (const nuclear_dist::iterator &i1, const nuclear_dist::iterator &i2)
  Compare two nuclei.

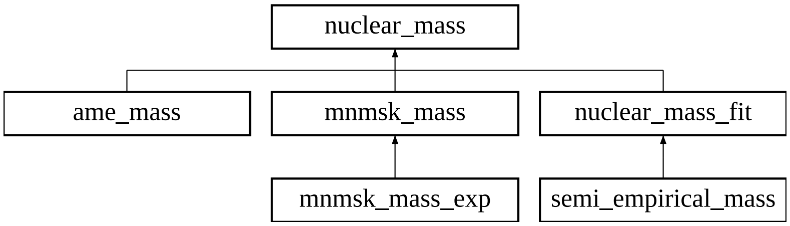
The documentation for this class was generated from the following file:

- nuclear_dist.h

5.23 nuclear_mass Class Reference

`#include <nuclear_mass.h>`

Inheritance diagram for nuclear_mass::



5.23.1 Detailed Description

Nuclear mass formula base.

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 112-118 are named "Uub", "Uut", "Uuq", "Uup", "Uuh", "Uus", and "Uuo", respectively.

The binding energy is defined by

$$\mathrm{BE} = Zm_H + Nm_n - m_{\mathrm{nuclide}}$$

where m_{nuclide} is the mass of the nucleus including the mass of the electrons. The mass excess is defined by

$$m_{\mathrm{excess}} = m_{\mathrm{nuclide}} - Am_u$$

For example, for U^{238}, the binding energy is 1801.695 MeV, the mass excess is 47.30366 MeV, and m_{nuclide} is 221742.9 MeV. This is consistent with the above, as m_H is 938.7830 MeV, m_n is 939.5650 MeV, and m_u is 931.494 MeV.

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.

This class is designed so that a child only needs to reimplement either mass_excess() (if `form_type==disc_type`) or mass_excess_d() (if `form_type==cont_type`), and the remaining functions should work then automatically.

Warning:

> The spin degeneracy is not handled particularly intelligently. The get_nucleus() function simply assumes 0 spin for even A and spin 1/2 for odd A.
> The treatment of the electron binding energy contribution is not necessarily consistent at present.

Some common reaction Q-values and separation energies:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Note:

All descendants ought to set `form_type` to indicate whether the mass formula is discrete or continuous

Todo

Make the treatment of the electron binding energy contribution more consistent.

Idea for future

It might be useful to consider a fudge factor to ensure no problems with finite precision arithmetic when converting `double` to `int`.

Definition at line 127 of file nuclear_mass.h.

Indicate whether or not the mass formula is

discrete or continuous

- static const int **cont_type** = 1
- static const int **disc_type** = 2
- int **form_type**

Public Member Functions

- virtual bool is_included (int Z, int N)
  Return false if the mass formula does not include specified nucleus.
- 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)
  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.*
- 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.
- int eltoZ (std::string el)
  Return Z given the element name.
- std::string Ztoel (size_t Z)
  Return the element name given Z.
- int parse_elstring (std::string ela, int &Z, int &N, int &A)
  Parse a string of the form "Pb208".

Protected Types

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

Protected Attributes

- std::map< std::string, int, string_less_than > 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).

Data Structures

- struct string_less_than
  String comparison operator for element_table.

5.23.2 Member Function Documentation

5.23.2.1 int get_nucleus (int *Z*, int *N*, nucleus & *n*) `[inline]`

Fill `n` with the information from nucleus with the given neutron and proton number.

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.

Definition at line 151 of file nuclear_mass.h.

5.23.2.2 virtual double binding_energy (int *Z*, int *N*) `[inline, virtual]`

Return the binding energy in MeV.

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 198 of file nuclear_mass.h.

5.23.2.3 virtual double binding_energy_d (double *Z*, double *N*) `[inline, virtual]`

Return the binding energy in MeV.

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 212 of file nuclear_mass.h.

5.23.2.4 int eltoZ (std::string *el*) `[inline]`

Return Z given the element name.

Properly handle errors

Definition at line 242 of file nuclear_mass.h.

5.23.2.5 std::string Ztoel (size_t Z) `[inline]`

Return the element name given Z.

Note that `Ztoel(0)` returns `"n"` indicating the neutron and if the argument `Z` is greater than 118, an empty string is returned.

Definition at line 253 of file nuclear_mass.h.

5.23.2.6 int parse_elstring (std::string *ela*, int & *Z*, int & *N*, int & *A*) `[inline]`

Parse a string of the form "Pb208".

Note that this does not correctly interpret dashes, e.g. "Pb-208". will not work.

Idea for future

> Properly ignore dashes and other non-alphanumeric characters.

Definition at line 270 of file nuclear_mass.h.

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

- nuclear_mass.h

5.24 nuclear_mass::string_less_than Struct Reference

`#include <nuclear_mass.h>`

5.24.1 Detailed Description

String comparison operator for element_table.

Definition at line 302 of file nuclear_mass.h.

Public Member Functions

- bool **operator()** (const std::string s1, const std::string s2) const

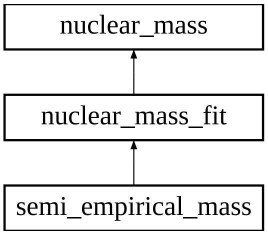
The documentation for this struct was generated from the following file:

- nuclear_mass.h

5.25 nuclear_mass_fit Class Reference

`#include <nuclear_mass.h>`

Inheritance diagram for nuclear_mass_fit::



5.25.1 Detailed Description

Fittable mass formula.

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

Definition at line 340 of file nuclear_mass.h.

Public Member Functions

- virtual int fit_fun (size_t nv, const **ovector_view** &x)
  Fix parameters from an array for fitting.
- virtual int guess_fun (size_t nv, **ovector_view** &x)
  Fill array with guess from present values for fitting.

Data Fields

- size_t nfit
  Number of fitting parameters.

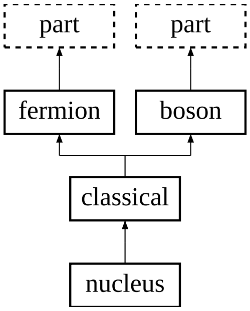
The documentation for this class was generated from the following file:

- nuclear_mass.h

5.26 nucleus Class Reference

`#include <nucleus.h>`

Inheritance diagram for nucleus::



5.26.1 Detailed Description

A simple nucleus class.

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 51 of file nucleus.h.

Data Fields

- int Z
  Proton number.
- int N
  Neutron number.
- int A
  Atomic number.
- double mex
  Mass excess.
- double be
  Binding energy (with a minus sign for bound nuclei).

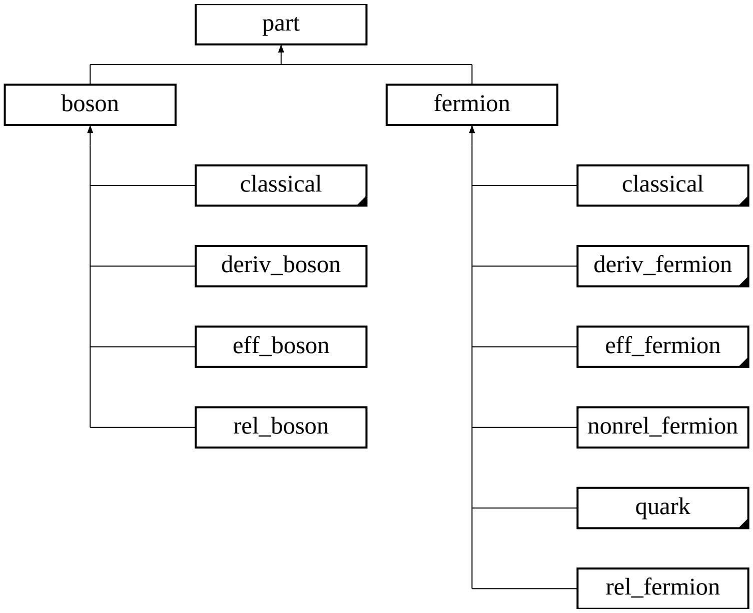
The documentation for this class was generated from the following file:

- nucleus.h

5.27 part Class Reference

`#include <part.h>`

Inheritance diagram for part::



5.27.1 Detailed Description

Particle base class.

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 100 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 calc_mu (const double temper)
  Calculate properties as function of chemical potential.
- virtual int calc_density (const double temper)
  Calculate properties as function of density.
- virtual int pair_mu (const double temper)
  Calculate properties with antiparticles as function of chemical potential.
- virtual int pair_density (const double temper)
  Calculate properties with antiparticles as function of density.
- 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
  When this is true, nu and ms are set equal to mu and m by calc_mu(), etc.. (default true).
- std::string name
  The name usually defaults to the class name.

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

- part.h

5.28 part_ioc Class Reference

`#include <part_ioc.h>`

5.28.1 Detailed Description

Setup I/O for particle classes.

Definition at line 44 of file part_ioc.h.

Data Fields

- **part_io_type** $*$ **part_io**
- **thermo_io_type** $*$ **thermo_io**
- **quark_io_type** $*$ **quark_io**
- rel_boson_io_type $*$ **rel_boson_io**
- **rel_fermion_io_type** $*$ **rel_fermion_io**
- **boson_io_type** $*$ **boson_io**
- **classical_io_type** $*$ **classical_io**
- **eff_boson_io_type** $*$ **eff_boson_io**
- **eff_fermion_io_type** $*$ **eff_fermion_io**
- **eff_quark_io_type** $*$ **eff_quark_io**
- **fermion_io_type** $*$ **fermion_io**
- **nonrel_fermion_io_type** $*$ **nonrel_fermion_io**

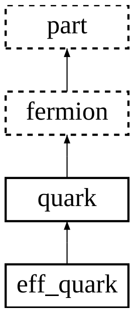
The documentation for this class was generated from the following file:

- part_ioc.h

5.29 quark Class Reference

`#include <quark.h>`

Inheritance diagram for quark::



5.29.1 Detailed Description

Quark base.

Definition at line 43 of file quark.h.

Public Member Functions

- quark (double mass=0.0, double dof=0.0)
  Create a quark with mass `m` *and degeneracy* `g`*.*
- virtual int calc_mu (const double temper)
  Calculate properties as function of chemical potential.
- virtual int calc_density (const double temper)
  Calculate properties as function of density.
- virtual int pair_mu (const double temper)
  Calculate properties with antiparticles as function of chemical potential.
- virtual int pair_density (const double temper)
  Calculate properties with antiparticles as function of density.
- virtual const char ∗ type ()
  Return string denoting type ("quark").

Data Fields

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

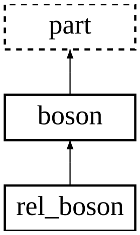
The documentation for this class was generated from the following file:

- quark.h

5.30 rel_boson Class Reference

`#include <rel_boson.h>`

Inheritance diagram for rel_boson::



5.30.1 Detailed Description

Equation of state for a relativistic boson.

Todo

Testing not completely finished.

Definition at line 48 of file rel_boson.h.

Public Member Functions

- rel_boson (double m=0.0, double g=0.0)
  Create a boson with mass `m` *and degeneracy* `g`.
- virtual int calc_mu (const double temper)
  Calculate properties as function of chemical potential.
- virtual int calc_density (const double temper)
  Calculate properties as function of density.
- virtual int pair_mu (const double temper)
  Calculate properties with antiparticles as function of chemical potential.
- virtual int nu_from_n (const double temper)
  Calculate effective chemical potential from density.
- int set_inte (**inte**< void ∗, **funct**< void ∗ > > &l_nit, **inte**< void ∗, **funct**< void ∗ > > &l_dit)
  Set **inte** *object.*
- int set_density_root (**root**< void ∗, **funct**< void ∗ > > &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**< void ∗, **funct**< void ∗ > > def_density_root
  The default solver for calc_density().
- **gsl_inte_qagiu**< void ∗, **funct**< void ∗ > > def_nit
  Default nondegenerate integrator.
- **gsl_inte_qag**< void ∗, **funct**< void ∗ > > def_dit
  Default degenerate integrator.

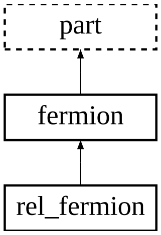
The documentation for this class was generated from the following file:

- rel_boson.h

5.31 rel_fermion Class Reference

`#include <rel_fermion.h>`

Inheritance diagram for rel_fermion::



5.31.1 Detailed Description

Equation of state for a relativistic fermion.

This implements an equation of state for a relativistic fermion using direct integration. 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. The upper limit on the degenerate integration is given by

$$\sqrt{(20T + \nu)^2 - m^{*,2}}$$

The default integrators are **gsl_inte_qag** (for degenerate particles) and **gsl_inte_qagiu** (for non-degenerate particles)

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, in which case the reported uncertainties are not necessarily correct.

Note:

This does not work with inc_rest_mass=false

Idea for future

Allow the user to change the upper limit on the degenerate integration and the hard-coded value of 200 in the integrands.

Idea for future

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

Definition at line 98 of file rel_fermion.h.

Public Member Functions

- rel_fermion (double m=0.0, double g=0.0)
  Create a fermion with mass `m` *and degeneracy* `g`*.*
- virtual int calc_mu (const double temper)
  Calculate properties as function of chemical potential.
- virtual int calc_density (const double temper)
  Calculate properties as function of density.
- virtual int pair_mu (const double temper)

Calculate properties with antiparticles as function of chemical potential.
- virtual int pair_density (const double temper)
  Calculate properties with antiparticles as function of density.
- virtual int nu_from_n (const double temper)
  Calculate effective chemical potential from density.
- int set_inte (**inte**< void ∗, **funct**< void ∗ > > &non_it, **inte**< void ∗, **funct**< void ∗ > > &deg_it)
  Set integrators.
- int set_density_root (**root**< void ∗, **funct**< void ∗ > > &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

- double deg_limit
  The critical degeneracy at which to switch integration techniques.
- fermion unc
  Storage for the uncertainty.
- 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**< void ∗, **funct**< void ∗ > > def_density_root
  The default solver for calc_density().
- **gsl_inte_qag**< void ∗, **funct**< void ∗ > > def_dit
  The default integrator for degenerate fermions.
- **gsl_inte_qagiu**< void ∗, **funct**< void ∗ > > def_nit
  The default integrator for non-degenerate fermions.

Protected Member Functions

- double density_fun (double u, void ∗&pa)
  The integrand for the density for non-degenerate fermions.
- double energy_fun (double u, void ∗&pa)
  The integrand for the energy density for non-degenerate fermions.
- double entropy_fun (double u, void ∗&pa)
  The integrand for the entropy density for non-degenerate fermions.
- double deg_density_fun (double u, void ∗&pa)
  The integrand for the density for degenerate fermions.
- double deg_energy_fun (double u, void ∗&pa)
  The integrand for the energy density for degenerate fermions.
- double deg_entropy_fun (double u, void ∗&pa)
  The integrand for the entropy density for degenerate fermions.
- int solve_fun (double x, double &yy, void ∗&pa)
  Solve for the chemical potential given the density.
- int pair_fun (double x, double &yy, void ∗&pa)
  Solve for the chemical potential given the density wiht antiparticles.

Protected Attributes

- **inte**< void ∗, **funct**< void ∗ > > ∗ nit
  The non-degenerate integrator.
- **inte**< void ∗, **funct**< void ∗ > > ∗ dit
  The degenerate integrator.
- **root**< void ∗, **funct**< void ∗ > > ∗ density_root
  The solver for calc_density().

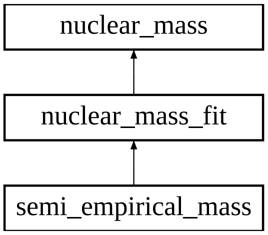
The documentation for this class was generated from the following file:

- rel_fermion.h

5.32 semi_empirical_mass Class Reference

`#include <nuclear_mass.h>`

Inheritance diagram for semi_empirical_mass::



5.32.1 Detailed Description

Semi-empirical mass formula.

A simple semi-empirical mass formula of the form

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

where

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

Note:

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

Definition at line 383 of file nuclear_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_view** &x)
  Fix parameters from an array for fitting.
- virtual int guess_fun (size_t nv, **ovector_view** &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).

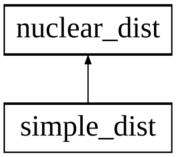
The documentation for this class was generated from the following file:

- nuclear_mass.h

5.33 simple_dist Class Reference

`#include <nuclear_dist.h>`

Inheritance diagram for simple_dist::



5.33.1 Detailed Description

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

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;
}
```

Todo

This takes a nuclear_mass pointer as input. It should probably be a reference instead? (3/18/08 - Yeah, I think it should be a reference.)

Todo

Add error checking in constructors and set functions

Definition at line 166 of file nuclear_dist.h.

Public Member Functions

- 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.

5.33.2 Constructor & Destructor Documentation

5.33.2.1 simple_dist (int *minZ*, int *maxZ*, int *minA*[ ], int *maxA*[ ], nuclear_mass ∗ *nm*)

Create a distribution from ranges in A specified for each Z.

The length of the arrays minA and maxA should be exactly $\mathrm{maxZ} - \mathrm{minZ} + 1$.

5.33.3 Member Function Documentation

5.33.3.1 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.

The length of the arrays minA and maxA should be exactly $\mathrm{maxZ} - \mathrm{minZ} + 1$.

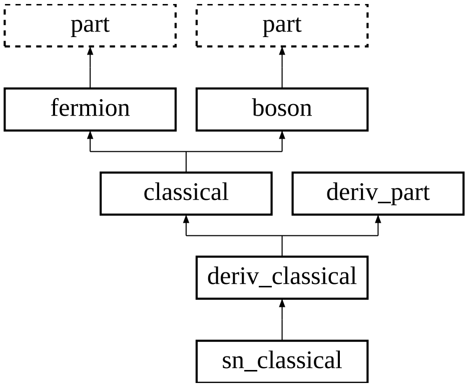
The documentation for this class was generated from the following file:

- nuclear_dist.h

5.34 sn_classical Class Reference

`#include <sn_classical.h>`

Inheritance diagram for sn_classical::



5.34.1 Detailed Description

Equation of state for a classical particle with derivatives.

Todo

This does not work with inc_rest_mass=true

Definition at line 42 of file sn_classical.h.

Public Member Functions

- sn_classical (double m=0.0, double g=0.0)
  Create a fermion with mass `m` *and degeneracy* `g`*.*
- virtual int calc_mu (const double temper)
  Calculate properties as function of chemical potential.
- virtual int calc_density (const double temper)
  Calculate properties as function of density.
- virtual int pair_mu (const double temper)
  Calculate properties with antiparticles as function of chemical potential.
- virtual int pair_density (const double temper)
  Calculate properties with antiparticles as function of density.
- virtual const char ∗ type ()
  Return string denoting type ("sn_classical").

5.34.2 Member Function Documentation

5.34.2.1 virtual int pair_mu (const double *temper*) `[virtual]`

Calculate properties with antiparticles as function of chemical potential.

Bug

This function probably doesn't work (3/6/08). See end of eff_fermion_ts.cpp.

Reimplemented from classical.

5.34.2.2 virtual int pair_density (const double *temper*) `[virtual]`

Calculate properties with antiparticles as function of density.

Bug

This function probably doesn't work (3/6/08). See end of eff_fermion_ts.cpp.

Reimplemented from classical.

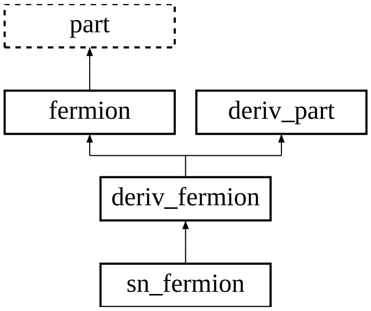
The documentation for this class was generated from the following file:

- sn_classical.h

5.35 sn_fermion Class Reference

`#include <sn_fermion.h>`

Inheritance diagram for sn_fermion::



5.35.1 Detailed Description

Equation of state for a relativistic fermion.

Note:

This class does not work with inc_rest_mass=true.

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. Typical choices are Gauss-Legendre integration for the degenerate regime and Gauss-Laguerre integration for the non-degenerate regime. The upper limit on the degenerate integration is given by the solution of

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

which is

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

where $\mathcal{L} \equiv \mathrm{flimit} \times T + \nu$.

In the non-degenerate regime, we make the substitution $u = k/T$ to ensure that the variable of integration does not have units.

Uncertainties are given in unc.

Todo

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

Todo

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

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

where the entropy density is

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

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

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

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

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

The dsdT integration doesn't work well if the system is very degenerate. When method is byparts, the integral involves a large cancellation between the regions from $k \in (0, \mathrm{ulimit}/2)$ and $k \in (\mathrm{ulimit}/2, \mathrm{ulimit})$. 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 is probably better than trying to perform the integration.

Todo

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

Todo

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

Definition at line 225 of file sn_fermion.h.

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.

Public Member Functions

- sn_fermion (double m=0.0, double g=0.0)
  Create a fermion with mass `m` *and degeneracy* `g`.
- virtual int calc_mu (const double temper)
  Calculate properties as function of chemical potential.
- virtual int calc_density (const double temper)
  Calculate properties as function of density.
- virtual int pair_mu (const double temper)

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

Data Fields

- double deg_limit
  The critical degeneracy at which to switch integration techniques (default 2.0).
- double flimit
  The limit for the Fermi functions (default 20.0).
- deriv_fermion unc
  Storage for the most recently calculated uncertainties.
- **gsl_inte_qagiu**< void ∗, **funct**< void ∗ > > def_nit
  The default integrator for the non-degenerate regime.
- **gsl_inte_qag**< void ∗, **funct**< void ∗ > > def_dit
  The default integrator for the degenerate regime.
- **cern_mroot_root**< void ∗, **funct**< void ∗ > > def_density_root
  The default solver for npen_density() and pair_density().

5.35.2 Member Function Documentation

5.35.2.1 int set_inte (inte< void ∗, funct< void ∗ > > & *unit*, inte< void ∗, funct< void ∗ > > & *udit*)

Set **inte** objects.

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.

5.35.3 Field Documentation

5.35.3.1 double flimit

The limit for the Fermi functions (default 20.0).

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

Definition at line 244 of file sn_fermion.h.

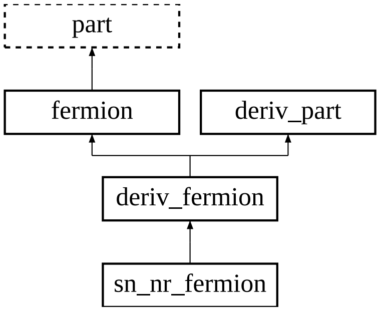
The documentation for this class was generated from the following file:

- sn_fermion.h

5.36 sn_nr_fermion Class Reference

`#include <sn_nr_fermion.h>`

Inheritance diagram for sn_nr_fermion::



5.36.1 Detailed Description

Equation of state for a nonrelativistic fermion.

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

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

where the entropy density is

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

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

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

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,

$$\frac{g}{2\pi^2}\int_0^\infty h(k) f(1-f) dk = \frac{g}{2\pi^2}\int_0^\infty f \frac{T m^*}{k}\left[h^\prime - \frac{h}{k}\right] dk$$

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

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

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

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

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

$$k dk = m^* T du$$

or

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

1) The derivative of the density wrt the chemical potential

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

2) The derivative of the density wrt the temperature

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

4) The derivative of the entropy wrt the temperature

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

5) The derivative of the density wrt the effective mass

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

Definition at line 221 of file sn_nr_fermion.h.

Public Member Functions

- sn_nr_fermion (double m=0.0, double g=0.0)
  Create a fermion with mass `m` *and degeneracy* `g`*.*
- virtual int calc_mu (const double temper)
  Calculate properties as function of chemical potential.
- virtual int calc_density (const double temper)
  Calculate properties as function of density.
- virtual int pair_mu (const double temper)
  Calculate properties with antiparticles as function of chemical potential.
- virtual int pair_density (const double temper)
  Calculate properties with antiparticles as function of density.
- virtual int nu_from_n (const double temper)
  Calculate effective chemical potential from density.
- int set_density_root (**root**< void ∗, **funct**< void ∗ > > &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).
- deriv_fermion 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**< void ∗, **funct**< void ∗ > > def_density_root
  The default solver for npen_density() and pair_density().

Protected Member Functions

- int solve_fun (double x, double &yy, void ∗&pa)
  Function to compute chemical potential from density.
- int pair_fun (double x, double &yy, void ∗&pa)
  Function to compute chemical potential from density when antiparticles are included.

Protected Attributes

- **root**< void ∗, **funct**< void ∗ > > ∗ density_root
  Solver to compute chemical potential from density.

5.36.2 Field Documentation

5.36.2.1 double flimit

The limit for the Fermi functions (default 20.0).

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

Definition at line 235 of file sn_nr_fermion.h.

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

- sn_nr_fermion.h

5.37 thermo Class Reference

`#include <part.h>`

5.37.1 Detailed Description

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

Definition at line 46 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

6 File Documentation

6.1 part.h File Reference

6.1.1 Detailed Description

File for definitions for thermo and part.

Definition in file part.h.

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

Data Structures

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

Typedefs

- typedef **io_tlate**< thermo > **thermo_io_type**
- typedef **io_tlate**< part > **part_io_type**

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.

Index