O_2scl_eos - Equation of State Sub-Library for O_2scl

Version 0.805

Contents

1 Main Page

All equations of state inherit from eos (except for the TOV solver tov_solve and cold_nstar). The `o2scl_eos` library contains all of the classes mentioned in this documentation.

1.1 User's Guide

1.2 Hadronic equations of state

The hadronic equations of state are all inherited from hadronic_eos: schematic_eos, skyrme_eos, rmf_eos, apr_eos, and gen_potential_eos.

hadronic_eos includes several methods that can be used to calculate the saturation properties of nuclear matter. These methods are sometimes overloaded in descendants when exact formulas are available.

There is also a set of classes to modify the quartic term of the symmetry energy: rmf4_eos, apr4_eos, skyrme4_eos, and mdi4_eos all based on sym4_eos_base which can be used in sym4_eos.

1.3 Equations of state of quark matter

The equations of state of **quark** matter are all inherited from quark_eos: bag_eos is a simple bag model, nambujl_eos is the Nambu–Jona-Lasinio model.

1.4 Solution of the Tolman-Oppenheimer-Volkov equations

The class tov_solve provide a solution to the Tolman-Oppenheimer-Volkov (TOV) equations given an equation of state. This is particularly useful for static neutron star structure: given any equation of state one can calculate the mass vs. radius curve and the properties of any star of a given mass. An adaptive integration is employed and calculates the gravitational mass, the baryonic mass (if the baryon density is supplied), and the gravitational potential. The remaining columns is the equation of state are also interpolated into the solution, e.g. if a chemical potential is given, then the radial dependence of the chemical potential for a 1.4 solar mass star can be automatically computed. The equation of state may be specified in arbitrary units so long as an appropriate conversion factor is supplied. An equation of state for low densities (baryon density < 0.08 fm^{-3}) is provided and can be automatically appended to the user-defined equation of state.

This is still experimental.

1.5 Naive Cold Neutron Stars

There is also a class to calculate zero-temperature neutron stars: cold_nstar. It uses tov_solve to compute the structure, given a hadronic equation of state (of type hadronic_eos). It also computes the adiabatic index, the speed of sound, and determines the possibility of the direct Urca process as a function of density or radius.

This is still experimental.

1.6 Non-relativistic Finite Temperature Approximations

This is taken from the Prakash87.

The entropy is

$$s = -\sum_k \left[n_k \ln n_k + (1 - n_k) \ln (1 - n_k) \right]$$

The low-temperature (degenerate) approximation to the entropy is

$$s = \pi^2 / 3 N(0) T$$

where the density of states at the Fermi surface is

$$N(0) = \sum_k \delta(\epsilon_k - \mu) = \frac{3\rho}{k_F v_F}$$

where the Fermi velocity is

$$v_F = \left. \frac{\partial \epsilon_k}{\partial k} \right|_{k_F} = \frac{k_F}{m^*}$$

The latter equation defines the effective mass. The level density parameter is given by

$$a = \frac{\pi^2 N(0)}{6\rho}$$

Defining the Fermi temperature:

$$T_F = \frac{1}{2} k_F v_F = k_F^2 / 2 / m^*$$

another expression for the entropy is

$$s = \frac{\pi^2}{2} \rho (T / T_F)$$

Expressions for the remaining quantities are

$$P = P(T = 0) + \frac{\rho}{3} a T^2 \left(1 + \frac{d \ln v_F}{d \ln k_F} \right)$$

$$E/A = E/A(T = 0) + a T^2$$

$$\mu = \mu(T = 0) - \frac{1}{3} a T^2 \left(2 - \frac{d \ln v_F}{d \ln k_F} \right)$$

Typically, the leading correction to

$$s = \frac{\pi^2}{2} \rho (T / T_F)$$

is of order $(T/T_F)^2$ unless soft collective modes give rise to a $(T/T_F)^3 \ln(T/T_F)$ correction.

At high temperature (non-degenerate approximation), a stationary phase approximation gives

$$\begin{aligned}\rho(T) &\sim \frac{\gamma}{2\pi^2} e^{\mu/T} \cdot k^2 e^{-\epsilon_k/T} \sqrt{2\pi} \left[\frac{2}{k^2} + \frac{1}{T} \frac{\partial v_k}{\partial k} \right]^{-1/2} \\ &= e^{\mu/T} \left. f(T) \right|_{k=k_\rho}\end{aligned}$$

where γ is the spin and isospin degeneracy and the velocity function is $v_k = \partial \epsilon_k / \partial k$. The function $f(T)$ is evaluated at momentum k_ρ which is obtained by solving $T - k v_k / 2 = 0$. The chemical potential is obtained by inverting the above relation for $\rho(T)$:

$$\mu \sim T \ln \rho - T \ln f(T)$$

From this value of μ we can derive the entropy density using

$$Ts \sim \sum_k n_k \epsilon_k + \rho T - \mu \rho$$

Using the stationary phase method:

$$\begin{aligned}\sum_k n_k \epsilon_k &\sim \frac{\gamma}{2\pi^2} e^{\mu/T} \cdot k^2 \epsilon_k e^{-\epsilon_k/T} \sqrt{2\pi} \left[\frac{2}{k^2} - \left(\frac{1}{\epsilon_k} - \frac{1}{T} \right) \frac{\partial v_k}{\partial k} + \left(\frac{v_k}{\epsilon_k} \right)^2 \right]^{-1/2} \\ &= e^{\mu/T} \left. g(T) \right|_{k=k_E}\end{aligned}$$

where k_E is the solution of

$$\frac{2}{k} + v_k \left(\frac{1}{\epsilon_k} - \frac{1}{T} \right) = 0$$

This provides a first approximation to the energy and together with the thermodynamic identity gives the pressure.

1.7 Example source code

Example list

- Cold neutron star example

1.7.1 Cold neutron star example

```
/* Example: ex_cold_nstar.cpp
   -------------------------------------------------------------------
   This example solves the TOV equations using class cold_nstar using a
   relativistic mean-field EOS from class rmf_eos.
*/

#include <o2scl/collection.h>
#include <o2scl/text_file.h>
#include <o2scl/cold_nstar.h>
#include <o2scl/rmf_eos.h>
#include <o2scl/test_mgr.h>

using namespace std;
using namespace o2scl;

// For hc_mev_fm
using namespace o2scl_const;

int main(void) {

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

```
  test_mgr t;
  t.set_output_level(1);

  cold_nstar nst;

  // Initialize EOS
  rmf_eos rmf;

  rmf.load("NL3");

  rmf.saturation();
  cout << "Saturation density: " << rmf.n0 << endl;
  cout << "Binding energy: " << rmf.eoa*hc_mev_fm << endl;
  cout << "Effective mass: " << rmf.msom << endl;
  cout << "Symmetry energy: " << rmf.esym*hc_mev_fm << endl;
  cout << "Compressibility: " << rmf.comp*hc_mev_fm << endl;

  // Compute EOS, include muons
  nst.include_muons=true;
  nst.set_eos(rmf);
  nst.calc_eos();
  table &te=nst.get_eos_results();

  // Output EOS results to a file
  collection co;
  text_out_file *tof=new text_out_file("ex_cold_nstar.out");
  co.out_one(tof,"table","tov",&te);
  delete tof;

  // Compute mass vs. radius
  nst.calc_nstar();
  table &tr=nst.get_tov_results();
  cout << "Maximum mass: " << tr.max("gm") << endl;
  cout << "Radius of maximum mass star: "
       << tr.get("r",tr.lookup("gm",tr.max("gm"))) << endl;
  cout << "Central baryon density of maximum mass star: ";
  cout << tr.get("nb",tr.lookup("gm",tr.max("gm"))) << endl;

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

1.8 Other Todos

Todo

Right now, the equation of state classes depend on the user to input the correct value of `non_interacting` for the particle inputs. This is not very graceful...

Todo

Document the "n15" models. What where they for?

1.9 Bibliography

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

Akmal98: Akmal, Pandharipande, and Ravenhall, Phys. Rev. C **58**, 1805 (1998).

Bartel79: J. Bartel, P. Quentin, M. Brack, C. Guet, and Håkansson, Nucl. Phys. A **386** (1982) 79.

Baym71: G. Baym, C. Pethick, and P. Sutherland, Astrophys. J. **170** (1971) 299.

Beiner75: M. Beiner, H. Flocard, Nguyen van Giai, and P. Quentin, Nucl. Phys. A **238** (1975) 29.

Bernard88: V. Bernard, R. L. Jaffe, and U.-G. Meissner, Nucl. Phys. B **308** (1988) 753.

Bombaci01: I. Bombaci, "Equation of State for Dense Isospin Asymmetric Nuclear Matter for Astrophysical Applications Equation of State for Isospin-Asymmetric Nuclear Matter and Neutron Star Properties", in "Isospin physics in heavy-ion collisions at intermediate energies" ed. by B-A. Li and W. U. Schröder (2001) Nova Science, New York.

Brack85: M. Brack, C. Guet, and H.-B. Håkansson, Phys. Rep. **123** (1985) 275.

Buballa99: M. Buballa and M. Oertel, Phys. Lett. B **457** (1999) 261.

Buballa04: M. Buballa, Phys. Rep. **407** (2005) 205-376.

Chabanat95: E. Chabanat, Ph. D. Thesis 1995

Chabanat97: E. Chabanat, J. Meyer, P. Bonche, R. Schaeffer, and P. Haensel, Nucl. Phys. A **627** (1997) 710.

Das03: C.B. Das, S. Das Gupta, C. Gale, and B.-A. Li, Phys. Rev. C **67** (2003) 034611.

Dobaczewski94: J. Dobaczewski, H. Flocard, and J. Treiner, Nucl. Phys. A **422** (1984) 103.

Dutta86: A.K.Dutta, J.-P.Arcoragi, J.M.Pearson, R.Behrman, F.Tondeur, Nucl. Phys. A **458** (1986) 77.

Friedrich86: J. Friedrich, and P.-G. Reinhard, Phys. Rev. C **33** (1986) 335.

Gale87: C. Gale, G. Bertsch, and S. Das Gupta, Phys. Rev. C **35** (1986) 1666.

Hatsuda94: T. Hatsuda and T. Kunihiro, Phys. Rep. **247** (1994) 221.

Horowitz01: Horowitz and Piekarewicz, Phys. Rev. Lett. **86** (2001) 5647.

Margueron02: J. Margueron, J. Navarro, and N. V. Giai, Phys. Rev. C **66** (2002) 014303.

Muller96: H. Muller and B. D. Serot, Nucl. Phys. A **606** (1996), 508.

Osni94: M. Onsi, H. Przysiezniak, J.M. Pearson, Phys. Rev. C **50** (1994) 460.

Pethick95: C. J. Pethick, D. G. Ravenhall, and C. P. Lorenz, Nucl. Phys. A **584** (1995) 675.

Prakash87: M. Prakash, T. L. Ainsworth, J. P. Blaizot, and H. Wolter, in Windsurfing the Fermi Sea, Volume II edited by T. T. S. Kuo and J. Speth, Elsevier 1987, pg. 357.

Prakash88: M. Prakash, T. L. Ainsworth, and J. M. Lattimer, Phys. Rev. Lett. **61** (1988) 2518.

Prakash97: Madappa Prakash, I. Bombaci, Manju Prakash, P. J. Ellis, J. M. Lattimer, and R. Knorren, Phys. Rep. **280** (1997) 1.

Reinhard95: P.-G. Reinhard and H. Flocard, Nucl. Phys. A **584** (1995) 467.

Reinhard99: P.-G. Reinhard, D.J. Dean, W. Nazarewicz, J. Dobaczewski, J.A. Maruhn, M.R. Strayer, Phys. Rev. C **60**, (1999) 014316.

Shapiro83: S. L. Shapiro and S. A. Teukolsky, "Black Holes, White Dwarfs, and Neutron Stars: The Physics of Compact Objects", John Wiley and Sons, New York, 1983.

Tondeur84: F. Tondeur, M. Brack, M. Farine, and J.M. Pearson, Nucl. Phys. A **420** (1984) 297.

Typel99: S. Typel and H. H. Wolter, Nucl. Phys. A **656 (1999) 331**.

VanGiai81: Nguyen van Giai and H. Sagawa, Phys. Lett. B **106** (1981) 379.

Zimanyi90: Zimanyi and Moszkowski, Phys. Rev. C **42** (1990) 1416.

2 Todo List

page Main Page Right now, the equation of state classes depend on the user to input the correct value of `non_interacting` for the particle inputs. This is not very graceful...

page Main Page Document the "n15" models. What where they for?

Class apr_eos Improve testing

Class bps_eos Can the pressure be made to match more closely?

Global cfl6_eos_old::eigenvalues(double mom, ovector_view &egv, double mu3, double mu8) This function may make some inappropriate assumptions on the vector egv.

Class cfl_njl_eos Improve documentation (Note that it appears to report the member functions twice?)

Class cfl_njl_eos Allow user to change derivative object? This isn't possible right now because the stepsize parameter of the derivative object is used.

Class cfl_njl_eos This class internally mixes ovector, omatrix, gsl_vector and gsl_matrix objects in a confusing and non-optimal way. Fix this.

Global cfl_njl_eos::calc_eq_temp_p(quark &u, quark &d, quark &s, double &qq1, double &qq2, double &qq3, double &gap1, double &
It surprises me that n3 is not -res[11]. Is there a sign error in the color densities?

Global cfl_njl_eos::gapped_eigenvalues(double m1, double m2, double lmom, double mu1, double mu2, double tdelta, double lam[4], do
Only the "ms" **part** of the quarks is referenced, so we should rewrite to use only double's as function arguments, and avoid passing pointers to **quark** objects.

Global cfl_njl_eos_old::gapped_eigenvalues(quark *q1, quark *q2, double mom, double mu1, double mu2, double delta, ovector_cx_vie
Only the "ms" **part** of the quarks is referenced, so we should rewrite to use only double's as function arguments, and avoid passing pointers to **quark** objects.

Class cold_nstar Ensure that the adiabatic index of the central density is greater than 4/3?

Class cold_nstar Implement more stability criteria

Class cold_nstar Warn if the EOS becomes pure neutron matter?

Class ddc_eos Finite temperature

Global ddc_eos::calc_eq_e(fermion &neu, fermion &p, double sig, double ome, double rho, double &f1, double &f2, double &f3, therm Is the thermodynamic identity is satisfied even when the field equations are not solved? Check this.

Class gen_potential_eos Calculate the chemical potentials analytically

Class hadronic_eos Need to consider using denpar versus n_baryon and proton_frac. Maybe some of this can be simplified or improved?

Class rmf_delta_eos Finish finite temperature

Class rmf_eos
- The number of couplings is getting unmanagable, maybe new organization is required.
- Check the formulas in the "Background" section
- Overload hadronic_eos::fcomp() with an exact version
- Fix calc_p() to be better at guessing
- There are two calc_e() functions that solve. One is specially designed to work without a good initial guess. Possible the other calc_e() function should be similarly designed?
- Make sure that this class properly handles particles for which inc_rest_mass is true/false
- It might be nice to remove explicit reference to the meson masses in functions which only compute nuclear matter since they are unnecessary. This might, however, demand redefining some of the couplings.

Global rmf_eos::calc_e(fermion &ne, fermion &pr, thermo &lth) This doesn't always work well when the proton density is zero.

Global rmf_eos::calc_e(fermion &ne, fermion &pr, thermo &lth, double &sig, double &ome, double &rho) This doesn't always work well when the proton density is zero.

Global rmf_eos::calc_e(fermion &ne, fermion &pr, thermo &lth, double &sig, double &ome, double &rho) Rename this function to distinguish between calc_e()'s

Global rmf_eos::n_charge Should use hadronic_eos::proton_frac instead?

Global rmf_eos::fix_saturation(double guess_cs=4.0, double guess_cw=3.0, double guess_b=0.001, double guess_c=-0.001)
- Fix this for zm_mode=true
- Ensure solver is more robust

Global rmf_eos::fkprime_fields(double sig, double ome, double nb, double &k, double &kprime) Does this work? Fix fkprime_fields() if it does not.

Global rmf_eos::check_naturalness(rmf_eos &re) I may have ignored some signs in the above, which are unimportant for this application, but it would be good to fix them for posterity.

Global schematic_eos::set_a_from_mstar(double u_msom, double mnuc) This was computed in schematic_sym.nb, which might be added to the documentation?

Class skyrme_eos
- Make sure that this class properly handles particles for which inc_rest_mass is true/false
- What about the spin-orbit units?
- Need to write a function that calculates saturation density?
- Remove use of mnuc in calparfun()?
- The compressibility could probably use some simplification
- Make sure the finite-temperature **part** is properly tested

Global skyrme_eos::calpar(double gt0=-10.0, double gt3=70.0, double galpha=0.2, double gt1=2.0, double gt2=-1.0) Does this work for both 'a' and 'b' non-zero?

Global skyrme_eos::calpar(double gt0=-10.0, double gt3=70.0, double galpha=0.2, double gt1=2.0, double gt2=-1.0) Compare to similar formulae from Margueron02

Global skyrme_eos::landau_nuclear(double n0, double m, double &f0, double &g0, double &f0p, double &g0p, double &f1, double &g1 This needs to be checked.

Global skyrme_eos::landau_neutron(double n0, double m, double &f0, double &g0, double &f1, double &g1) This needs to be checked

Class tov_buchdahl_eos Fix the reference above

Class tov_solve
- The error handler is called in tov_solve()derivs for pressures less than the minimum even in normal circumstances. This should be fixed, so that errors are more rare
- baryon mass doesn't work for fixed() (This may be fixed. We should make sure it's tested.)
- Combine maxoutsize and kmax?
- Document column naming issues

3 Ideas for future development

Class bps_eos Convert to a hadronic_eos object and offer an associated interface?

Class cfl_njl_eos_old This class internally mixes ovector, omatrix, gsl_vector and gsl_matrix objects in a confusing and non-optimal way. Fix this.

Class cfl_njl_eos_old Allow user to change derivative object? This isn't possible right now because the stepsize parameter of the derivative object is used.

Class tov_buchdahl_eos Figure out what to do with the buchfun() function

Class tov_solve
- Turn numbers in max() function into variables

4 Bug List

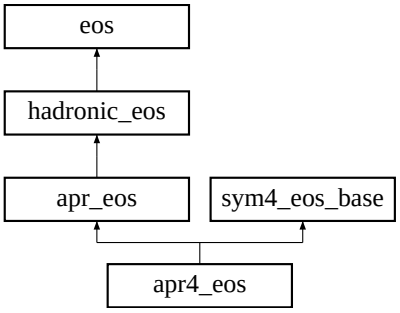
Class gen_potential_eos The BGBD eos doesn't work and the effective mass for the GBD eos doesn't work

5 Data Structure Documentation

5.1 apr4_eos Class Reference

`#include <sym4_eos.h>`

Inheritance diagram for apr4_eos::



5.1.1 Detailed Description

A version of apr_eos to separate potential and kinetic contributions.

Definition at line 97 of file sym4_eos.h.

Public Member Functions

- virtual int calc_e_sep (**fermion** &ne, **fermion** &pr, double &ed_kin, double &ed_pot, double &mu_n_kin, double &mu_p_kin, double &mu_n_pot, double &mu_p_pot)
  Compute the potential and kinetic parts separately.

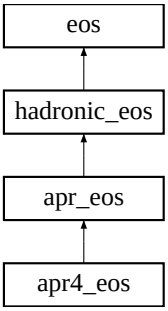
The documentation for this class was generated from the following file:

- sym4_eos.h

5.2 apr_eos Class Reference

`#include <apr_eos.h>`

Inheritance diagram for apr_eos::



5.2.1 Detailed Description

EOS from Akmal, Pandharipande, and Ravenhall (zero temperature).

Taken from Akmal98.

The chemical potentials include the rest mass energy and the energy density includes the rest mass energy density.

Note that APR seems to have been designed to be used with non-relativistic neutrons and protons with equal masses of 939 MeV. This gives a saturation density very close to 0.16.

The Hamiltonian is:

$$\mathcal{H}_{APR} = \mathcal{H}_{kin} + \mathcal{H}_{pot}$$

$$\mathcal{H}_{kin} = \left(\frac{\hbar^2}{2m} + \left(p_3 + (1-x)\, p_5\right) n e^{-p_4 n} \right) \tau_n + \left(\frac{\hbar^2}{2m} + \left(p_3 + x p_5\right) n e^{-p_4 n} \right) \tau_p$$

$$\mathcal{H}_{pot} = g_1 \left(1 - (1-2x)^2 \right) + g_2 \left(1-2x\right)^2$$

The following are definitions for g_i in the low-density phase (LDP) or the high-density phase (HDP):

$$g_{1,LDP} = -n^2 \left(p_1 + p_2 n + p_6 n^2 + \left(p_{10} + p_{11} n\right) e^{-p_9^2 n^2} \right)$$

$$g_{2,LDP} = -n^2 \left(p_{12}/n + p_7 + p_8 n + p_{13} e^{-p_9^2 n^2} \right)$$

$$g_{1,HDP} = g_{1,LDP} - n^2 \left(p_{17} \left(n - p_{19}\right) + p_{21} \left(n - p_{19}\right)^2 e^{p_{18}(n - p_{19})} \right)$$

$$g_{2,HDP} = g_{2,LDP} - n^2 \left(p_{15} \left(n - p_{20}\right) + p_{14} \left(n - p_{20}\right)^2 e^{p_{16}(n - p_{20})} \right)$$

The variables ν_n and ν_p contain the expressions $(-\mu_n + V_n)/T$ and $(-\mu_p + V_p)/T$ respectively, where V is the potential **part** of the single particle energy for particle i (i.e. the derivative of the Hamiltonian w.r.t. density while energy density held constant). Equivalently, ν_n is just $-k_{F_n}^2/2m^*$.

The finite temperature approximations from Prakash97 (see also the "Windsurfing the Fermi Sea" proceedings) are used in testing.

Note:

Since this EOS uses the effective masses and chemical potentials in the **fermion** class, the values of **part::non_interacting** for neutrons and protons are set to false in many of the functions.

Todo

Improve testing

Definition at line 112 of file apr_eos.h.

Choose phase

0 = use phase which minimizes energy (default)

1 = LDP (no pion condensation)

2 = HDP (pion condensation)

- static const int **best** = 0
- static const int **ldp** = 1
- static const int **hdp** = 2
- int **pion**

Public Member Functions

- virtual int calc_e (**fermion** &n, **fermion** &p, **thermo** &th)
  Equation of state as a function of density.
- virtual int calc_temp_e (**fermion** &n, **fermion** &pr, const double temper, **thermo** &th)
  Equation of state as a function of densities.
- int last_phase ()
  Return the phase of the most recent call to calc_e().
- double fcomp (double nb)
  Compute the compressibility.
- double fesym_diff (double nb)
  Calculate symmetry energy of matter as energy of neutron matter minus the energy of nuclear matter.
- void select (int model_index)
  Select model.
- int gradient_qij2 (double nn, double np, double &qnn, double &qnp, double &qpp, double &dqnndnn, double &dqnndnp, double &dqnpdnn, double &dqnpdnp, double &dqppdnn, double &dqppdnp)
  Calculate Q's for semi-infinite nuclear matter.
- double get_par (int n)
  Get the value of one of the parameters.
- int set_par (int n, double x)
  Set the value of one of the parameters.
- virtual const char * type ()
  Return string denoting type ("apr_eos").

Data Fields

- **nonrel_fermion** def_nr_neutron
  Default nonrelativistic neutron.
- **nonrel_fermion** def_nr_proton
  Default nonrelativistic proton.
- bool parent_method
  If true, use the methods from hadronic_eos for fcomp().

Protected Attributes

- double * par
  Storage for the parameters.
- int lp
  An integer to indicate which phase was used in calc_e().
- int choice
  The variable indicating which parameter set is to be used.

5.2.2 Member Function Documentation

5.2.2.1 double fcomp (double *nb*) [virtual]

Compute the compressibility.

See general notes at hadronic_eos::fcomp(). This computes the compressibility (at fixed proton fraction = 0.5) exactly, unless parent_method is true in which case the derivative is taken numerically in hadronic_eos::fcomp().

Reimplemented from hadronic_eos.

5.2.2.2 double fesym_diff (double *nb*) [virtual]

Calculate symmetry energy of matter as energy of neutron matter minus the energy of nuclear matter.

This function returns the energy per baryon of neutron matter minus the energy per baryon of nuclear matter. This will deviate significantly from the results from fesym() only if the dependence of the symmetry energy on δ is not quadratic.

Reimplemented from hadronic_eos.

5.2.2.3 void select (int *model_index*)

Select model.

Valid values for model_index are:

1 - A18+UIX*+deltav (preferred by Akmal, et. al. - this is the default)

2 - A18+UIX*

3 - A18+deltav

4 - A18

If any other integer is given, A18+UIX*+deltav is assumed.

5.2.2.4 int gradient_qij2 (double *nn*, double *np*, double & *qnn*, double & *qnp*, double & *qpp*, double & *dqnndnn*, double & *dqnndnp*, double & *dqnpdnn*, double & *dqnpdnp*, double & *dqppdnn*, double & *dqppdnp*)

Calculate Q's for semi-infinite nuclear matter.

For general discussion, see the documentation to hadronic_eos::qs().

For APR, we set $x_1 = x_2 = 0$ so that $Q_i = P_i/2$ and then

$$\begin{aligned} P_1 &= \left(\frac{1}{2}p_3 - p_5\right) e^{-p_4 n} \\ P_2 &= \left(\frac{1}{2}p_3 + p_5\right) e^{-p_4 n} \end{aligned}$$

This gives

$$Q_{nn} = \frac{1}{4} e^{-p_4 \rho} \left[-6p_5 - p_4(p_3 - 2p_5)(n_n + 2n_p)\right]$$

$$
\begin{aligned}
Q_{np} &= \frac{1}{8} e^{-p_4 \rho} \left[4(p_3 - 4p_5) - 3p_4(p_3 - 2p_5)(n_n + n_p) \right] \\
Q_{pp} &= \frac{1}{4} e^{-p_4 \rho} \left[-6p_5 - p_4(p_3 - 2p_5)(n_p + 2n_n) \right]
\end{aligned}
$$

See the Mathematica notebook

```
doc/o2scl/extras/apr_eos.nb
doc/o2scl/extras/apr_eos.ps
```

5.2.3 Field Documentation

5.2.3.1 bool parent_method

If true, use the methods from hadronic_eos for fcomp().

This can be set to true to check the difference in the compressibility wbetween the exact expressions and the numerical values from class hadronic_eos.

Definition at line 267 of file apr_eos.h.

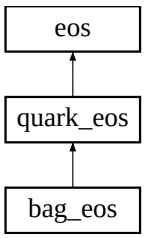
The documentation for this class was generated from the following file:

- apr_eos.h

5.3 bag_eos Class Reference

```
#include <bag_eos.h>
```

Inheritance diagram for bag_eos::



5.3.1 Detailed Description

Naive bag model EOS.

An equation of state with $P = -B + P_{u,FG} + P_{d,FG} + P_{s,FG}$ where $P_{i,FG}$ is the Fermi gas contribution from particle i and B is a density- and temperature-independent bag constant.

The finite temperature functions run the zero temperature code if the temperature is less than or equal to 0.

Definition at line 47 of file bag_eos.h.

Public Member Functions

- virtual int calc_p (**quark** &u, **quark** &d, **quark** &s, **thermo** &th)
  Calculate equation of state as a function of chemical potentials.
- virtual int calc_e (**quark** &u, **quark** &d, **quark** &s, **thermo** &th)
  Calculate equation of state as a function of density.

- virtual int calc_temp_p (**quark** &u, **quark** &d, **quark** &s, const double temper, **thermo** &th)
  Calculate equation of state as a function of the chemical potentials.
- virtual int calc_temp_e (**quark** &u, **quark** &d, **quark** &s, const double temper, **thermo** &th)
  Calculate equation of state as a function of the densities.
- virtual const char * type ()
  Return string denoting type ("bag_eos").

Data Fields

- double bag_constant
  The bag constant in fm^{-4} *(default* $200/(\hbar c)$*).*

5.3.2 Member Function Documentation

5.3.2.1 virtual int calc_temp_p (quark & *u*, quark & *d*, quark & *s*, const double *temper*, thermo & *th*) [virtual]

Calculate equation of state as a function of the chemical potentials.

This function returns zero (success) unless the call to **quark::pair_mu()** fails.

Reimplemented from quark_eos.

5.3.2.2 virtual int calc_temp_e (quark & *u*, quark & *d*, quark & *s*, const double *temper*, thermo & *th*) [virtual]

Calculate equation of state as a function of the densities.

This function returns zero (success) unless the call to **quark::pair_density()** fails.

Reimplemented from quark_eos.

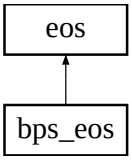
The documentation for this class was generated from the following file:

- bag_eos.h

5.4 bps_eos Class Reference

`#include <bps_eos.h>`

Inheritance diagram for bps_eos::



5.4.1 Detailed Description

Baym-Pethick-Sutherland equation of state.

This calculates the equation of state of electrons and nuclei using the approach of Baym71 (based on the discussion in Shapiro83) between about 8×10^6 g/cm^3 and 4.3×10^{11} g/cm^3. Below these densities, more complex Coulomb corrections need to be considered, and above these densities, neutron drip is important.

The default mass formula is semi-empirical

$$\begin{aligned} M(A,Z) &= (A-Z)m_n + Z(m_p+m_e) - 15.76A - 17.81A^{2/3} \\ &\quad -0.71Z^2/A^{1/3} - 94.8/A\,(A/2-Z)^2 + E_{\rm pair} \end{aligned}$$

where

$$E_{\rm pair} = \pm 39/A^{3/4}$$

if the **nucleus** is odd-odd (plus sign) or even-even (minus sign) and $E_{\rm pair}$ is zero for odd-even and even-odd nuclei. The nuclei are assumed not to contribute to the pressure. The electronic contribution to the pressure is assumed to be equal to the Fermi gas contribution plus a "lattice" contribution

$$\varepsilon_L = -1.444 Z^{2/3} e^2 n_e^{4/3}$$

This is Eq. 2.7.2 in Shapiro83. The rest mass energy of the nucleons is included in the energy density.

The original results from Baym71 are stored as a **table** in bps.eos. The testing code for this class compares the calculations to the **table** and matches to within .2 percent for the energy density and 9 percent for the pressure (for a fixed baryon number density).

Todo

Can the pressure be made to match more closely?

Idea for future

Convert to a hadronic_eos object and offer an associated interface?

Definition at line 84 of file bps_eos.h.

Public Member Functions

- virtual int calc_density (double barn, **thermo** &th, int &Z, int &A)
  Calculate the equation of state as a function of the baryon number density `barn`.
- virtual int calc_pressure (**thermo** &th, double &barn, int &Z, int &A)
  Calculate the equation of state as a function of the pressure.
- virtual double lattice_energy (int Z)
  The electron lattice energy.
- virtual **fermion** * get_electron ()
  Get a pointer to the electron.
- virtual double mass_formula (int Z, int A)
  The mass formula.
- virtual const char * type ()
  Return string denoting type ("bps_eos").
- int set_mass_formula (**nuclear_mass** &nm)
  Set the nuclear mass formula to be used.
- int calc_density_fixedA (double barn, **thermo** &th, int &Z, int A)
  Compute the ground state assuming a fixed atomic number.

Data Fields

- **semi_empirical_mass** def_mass
  Default mass formula.

Protected Member Functions

- virtual int eq274 (size_t nv, const **ovector_view** &nx, **ovector_view** &ny, void *&pa)
  Solve Equation 2.7.4 for a given pressure.
- double gibbs (int Z, int A)
  The Gibbs free energy.
- double energy (double barn, int Z, int A)
  The energy density.

Protected Attributes

- **gsl_mroot_hybrids**< void ∗, **mm_funct**< void ∗ > > gs
  A solver to solve Eq. 2.7.4.
- **fermion** e
  The electron thermodynamics.
- **nuclear_mass** ∗ nmp
  The nuclear mass formula.

5.4.2 Member Function Documentation

5.4.2.1 virtual int calc_density (double *barn*, thermo & *th*, int & *Z*, int & *A*) [virtual]

Calculate the equation of state as a function of the baryon number density `barn`.

This calculates the equation of state as a function of the baryon number density in fm^{-3}, returning the representative **nucleus** with proton number `Z` and atomic number `A`. The pressure and energy density are returned in `th` in fm^{-4}.

5.4.2.2 virtual int calc_pressure (thermo & *th*, double & *barn*, int & *Z*, int & *A*) [virtual]

Calculate the equation of state as a function of the pressure.

This calculates the equation of state as a function of the pressure, returning the representative **nucleus** with proton number `Z` and atomic number `A` and the baryon number density `barn` in fm^{-3}. The energy density is also returned in fm^{-4} in `th`.

5.4.2.3 virtual double mass_formula (int *Z*, int *A*) [virtual]

The mass formula.

The nuclear mass without the contribution of the rest mass of the electrons. The electron rest mass energy is included in the electron thermodynamics elsewhere.

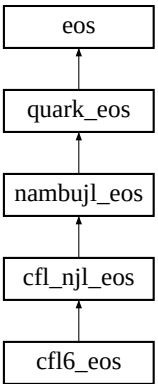
The documentation for this class was generated from the following file:

- bps_eos.h

5.5 cfl6_eos Class Reference

`#include <cfl6_eos.h>`

Inheritance diagram for cfl6_eos::



5.5.1 Detailed Description

CFL NJL EOS with a color-superconducting 't Hooft interaction.

Beginning with the Lagrangian:

$$\mathcal{L} = \mathcal{L}_{Dirac} + \mathcal{L}_{NJL} + \mathcal{L}_{'tHooft} + \mathcal{L}_{SC} + \mathcal{L}_{SC6}$$

$$\mathcal{L}_{Dirac} = \bar{q}\left(i\partial - m - \mu\gamma^0\right) q$$

$$\mathcal{L}_{NJL} = G_S \sum_{a=0}^{8} \left[\left(\bar{q}\lambda^a q\right)^2 - \left(\bar{q}\lambda^a\gamma^5 q\right)^2\right]$$

$$\mathcal{L}_{'tHooft} = G_D \left[\det{}_f \bar{q}\left(1-\gamma^5\right) q + \det{}_f \bar{q}\left(1+\gamma^5\right) q\right]$$

$$\mathcal{L}_{SC} = G_{DIQ}\left(\bar{q}_{i\alpha} i\gamma^5 \varepsilon^{ijk}\varepsilon^{\alpha\beta\gamma} q^C_{j\beta}\right)\left(\bar{q}_{\ell\delta} i\gamma^5 \epsilon^{\ell mk}\epsilon^{\delta\varepsilon\gamma} q^C_{m\varepsilon}\right)$$

$$\mathcal{L}_{SC6} = K_D\left(\bar{q}_{i\alpha} i\gamma^5 \varepsilon^{ijk}\varepsilon^{\alpha\beta\gamma} q^C_{j\beta}\right)\left(\bar{q}_{\ell\delta} i\gamma^5 \epsilon^{\ell mn}\epsilon^{\delta\varepsilon\eta} q^C_{m\varepsilon}\right)\left(\bar{q}_{k\gamma} q_{n\eta}\right)$$

We can simplify the relevant terms in $\mathcal{L}_{NJL}$:

$$\mathcal{L}_{NJL} = G_S\left[(\bar{u}u)^2 + \left(\bar{d}d\right)^2 + (\bar{s}s)^2\right]$$

and in $\mathcal{L}_{'tHooft}$:

$$\mathcal{L}_{NJL} = G_D\left(\bar{u}u\bar{d}d\bar{s}s\right)$$

Using the definition:

$$\Delta^{k\gamma} = \left\langle \bar{q} i\gamma^5 \epsilon\epsilon q^C \right\rangle$$

and the ansatzes:

$$(\bar{q}_1 q_2)(\bar{q}_3 q_4) \rightarrow \bar{q}_1 q_2 \left\langle \bar{q}_3 q_4\right\rangle + \bar{q}_3 q_4 \left\langle \bar{q}_1 q_2\right\rangle - \left\langle \bar{q}_1 q_2\right\rangle \left\langle \bar{q}_3 q_4\right\rangle$$

$$(\bar{q}_1 q_2)(\bar{q}_3 q_4)(\bar{q}_5 q_6) \rightarrow \bar{q}_1 q_2 \left\langle \bar{q}_3 q_4\right\rangle \left\langle \bar{q}_5 q_6\right\rangle + \bar{q}_3 q_4 \left\langle \bar{q}_1 q_2\right\rangle \left\langle \bar{q}_5 q_6\right\rangle + \bar{q}_5 q_6 \left\langle \bar{q}_1 q_2\right\rangle \left\langle \bar{q}_3 q_4\right\rangle - 2\left\langle \bar{q}_1 q_2\right\rangle \left\langle \bar{q}_3 q_4\right\rangle \left\langle \bar{q}_5 q_6\right\rangle$$

for the mean field approximation, we can rewrite the Lagrangian

$$\mathcal{L}_{NJL} = 2G_S\left[\left(\bar{u}u\right)\left\langle \bar{u}u\right\rangle + \left(\bar{d}d\right)\left\langle \bar{d}d\right\rangle + \left(\bar{s}s\right)\left\langle \bar{s}s\right\rangle - \left\langle \bar{u}u\right\rangle^2 - \left\langle \bar{d}d\right\rangle^2 - \left\langle \bar{s}s\right\rangle^2\right]$$

$$\mathcal{L}_{'tHooft} = -2G_D\left[\left(\bar{u}u\right)\left\langle \bar{u}u\right\rangle\left\langle \bar{s}s\right\rangle + \left(\bar{d}d\right)\left\langle \bar{u}u\right\rangle\left\langle \bar{s}s\right\rangle + \left(\bar{s}s\right)\left\langle \bar{u}u\right\rangle\left\langle \bar{d}d\right\rangle - 2\left\langle \bar{u}u\right\rangle\left\langle \bar{d}d\right\rangle\left\langle \bar{s}s\right\rangle\right]$$

$$\mathcal{L}_{SC} = G_{DIQ}\left[\Delta^{k\gamma}\left(\bar{q}_{\ell\delta} i\gamma^5 \epsilon^{\ell mk}\epsilon^{\delta\varepsilon\gamma} q^C_{m\varepsilon}\right) + \left(\bar{q}_{i\alpha} i\gamma^5 \varepsilon^{ijk}\varepsilon^{\alpha\beta\gamma} q^C_{j\beta}\right)\Delta^{k\gamma\dagger} - \Delta^{k\gamma}\Delta^{k\gamma\dagger}\right]$$

$$\mathcal{L}_{SC6} = K_D\left[\left(\bar{q}_{m\varepsilon} q_{n\eta}\right)\Delta^{k\gamma}\Delta^{m\varepsilon\dagger} + \left(\bar{q}_{i\alpha} i\gamma^5 \varepsilon^{ijk}\varepsilon^{\alpha\beta\gamma} q^C_{j\beta}\right)\Delta^{m\varepsilon\dagger}\left\langle \bar{q}_{m\varepsilon} q_{n\eta}\right\rangle\right]$$
$$+K_D\left[\Delta^{k\gamma}\left(\bar{q}_{\ell\delta} i\gamma^5 \epsilon^{\ell mn}\epsilon^{\delta\varepsilon\eta} q^C_{m\varepsilon}\right)\left\langle \bar{q}_{m\varepsilon} q_{n\eta}\right\rangle - 2\Delta^{k\gamma}\Delta^{m\varepsilon\dagger}\left\langle \bar{q}_{m\varepsilon} q_{n\eta}\right\rangle\right]$$

If we make the definition $\tilde{\Delta} = 2G_{DIQ}\Delta$

Definition at line 183 of file cfl6_eos.h.

Public Member Functions

- virtual int calc_eq_temp_p (**quark** &u, **quark** &d, **quark** &s, double &qq1, double &qq2, double &qq3, double &gap1, double &gap2, double &gap3, double mu3, double mu8, double &n3, double &n8, **thermo** &qb, const double temper)
  Calculate the EOS.
- virtual int integrands (double p, double res[ ])
  The momentum integrands.
- virtual int test_derivatives (double lmom, double mu3, double mu8, **test_mgr** &t)
  Check the derivatives specified by eigenvalues().
- virtual int eigenvalues6 (double lmom, double mu3, double mu8, double egv[36], double dedmuu[36], double dedmud[36], double dedmus[36], double dedmu[36], double dedmd[36], double dedms[36], double dedu[36], double dedd[36], double deds[36], double dedmu3[36], double dedmu8[36])

Calculate the energy eigenvalues and their derivatives.

- virtual int make_matrices (double lmom, double mu3, double mu8, double egv[36], double dedmuu[36], double dedmud[36], double dedmus[36], double dedmu[36], double dedmd[36], double dedms[36], double dedu[36], double dedd[36], double deds[36], double dedmu3[36], double dedmu8[36])
  Construct the matrices, but don't solve the eigenvalue problem.
- virtual const char * type ()
  Return string denoting type ("cfl6_eos").

Data Fields

- double KD
  The color superconducting 't Hooft coupling (default 0).
- double kdlimit
  The absolute value below which the CSC 't Hooft coupling is ignored(default 10^{-6}*).*

Protected Member Functions

- int set_masses ()
  Set the **quark** *effective masses from the gaps and the condensates.*

Protected Attributes

- **omatrix_cx** iprop6
  Storage for the inverse propagator.
- **omatrix_cx** eivec6
  The eigenvectors.
- **omatrix_cx** dipdgapu
  The derivative wrt the ds gap.
- **omatrix_cx** dipdgapd
  The derivative wrt the us gap.
- **omatrix_cx** dipdgaps
  The derivative wrt the ud gap.
- **omatrix_cx** dipdqqu
  The derivative wrt the up **quark** *condensate.*
- **omatrix_cx** dipdqqd
  The derivative wrt the down **quark** *condensate.*
- **omatrix_cx** dipdqqs
  The derivative wrt the strange **quark** *condensate.*
- **ovector** eval6
  Storage for the eigenvalues.
- gsl_eigen_hermv_workspace * w6
  GSL workspace for the eigenvalue computation.

Static Protected Attributes

- static const int mat_size = 36
  The size of the matrix to be diagonalized.

Private Member Functions

- **cfl6_eos** (const cfl6_eos &)
- cfl6_eos & **operator=** (const cfl6_eos &)

5.5.2 Member Function Documentation

5.5.2.1 virtual int calc_eq_temp_p (quark & *u*, quark & *d*, quark & *s*, double & *qq1*, double & *qq2*, double & *qq3*, double & *gap1*, double & *gap2*, double & *gap3*, double *mu3*, double *mu8*, double & *n3*, double & *n8*, thermo & *qb*, const double *temper*) [virtual]

Calculate the EOS.

Calculate the EOS from the **quark** condensates. Return the mass gap equations in `qq1`, `qq2`, `qq3`, and the normal gap equations in `gap1`, `gap2`, and `gap3`.

Using `fromqq=true` as in nambujl_eos and nambujl_temp_eos does not work here and will return an error.

If all of the gaps are less than gap_limit, then the nambujl_temp_eos::calc_temp_p() is used, and `gap1`, `gap2`, and `gap3` are set to equal `u.del`, `d.del`, and `s.del`, respectively.

Reimplemented from cfl_njl_eos.

5.5.2.2 virtual int eigenvalues6 (double *lmom*, double *mu3*, double *mu8*, double *egv*[36], double *dedmuu*[36], double *dedmud*[36], double *dedmus*[36], double *dedmu*[36], double *dedmd*[36], double *dedms*[36], double *dedu*[36], double *dedd*[36], double *deds*[36], double *dedmu3*[36], double *dedmu8*[36]) [virtual]

Calculate the energy eigenvalues and their derivatives.

Given the momentum `mom`, and the chemical potentials associated with the third and eighth gluons (`mu3` and `mu8`), this computes the eigenvalues of the inverse propagator and the assocated derivatives.

Note that this is not the same as cfl_njl_eos::eigenvalues() which returns `dedmu` rather `dedqqu`.

5.5.2.3 virtual int make_matrices (double *lmom*, double *mu3*, double *mu8*, double *egv*[36], double *dedmuu*[36], double *dedmud*[36], double *dedmus*[36], double *dedmu*[36], double *dedmd*[36], double *dedms*[36], double *dedu*[36], double *dedd*[36], double *deds*[36], double *dedmu3*[36], double *dedmu8*[36]) [virtual]

Construct the matrices, but don't solve the eigenvalue problem.

This is used by check_derivatives() to make sure that the derivative entries are right.

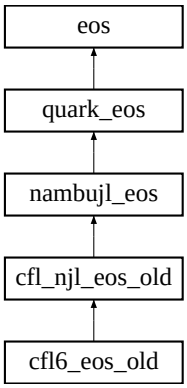
The documentation for this class was generated from the following file:

- cfl6_eos.h

5.6 cfl6_eos_old Class Reference

`#include <cfl6_eos_old.h>`

Inheritance diagram for cfl6_eos_old::



5.6.1 Detailed Description

CFL NJL EOS with a color-superconducting 't Hooft interaction.

Beginning with the Lagrangian:

$$\mathcal{L} = \mathcal{L}_{Dirac} + \mathcal{L}_{NJL} + \mathcal{L}_{'tHooft} + \mathcal{L}_{SC} + \mathcal{L}_{SC6}$$

$$\mathcal{L}_{Dirac} = \bar{q}\left(i\partial - m - \mu\gamma^0\right) q$$

$$\mathcal{L}_{NJL} = G_S \sum_{a=0}^{8} \left[\left(\bar{q}\lambda^a q\right)^2 - \left(\bar{q}\lambda^a\gamma^5 q\right)^2\right]$$

$$\mathcal{L}_{'tHooft} = G_D \left[\det{}_f \bar{q}\left(1-\gamma^5\right) q + \det{}_f \bar{q}\left(1+\gamma^5\right) q\right]$$

$$\mathcal{L}_{SC} = G_{DIQ}\left(\bar{q}_{i\alpha} i\gamma^5 \varepsilon^{ijk}\varepsilon^{\alpha\beta\gamma} q^C_{j\beta}\right)\left(\bar{q}_{\ell\delta} i\gamma^5 \epsilon^{\ell mk}\epsilon^{\delta\varepsilon\gamma} q^C_{m\varepsilon}\right)$$

$$\mathcal{L}_{SC6} = K_D\left(\bar{q}_{i\alpha} i\gamma^5 \varepsilon^{ijk}\varepsilon^{\alpha\beta\gamma} q^C_{j\beta}\right)\left(\bar{q}_{\ell\delta} i\gamma^5 \epsilon^{\ell mn}\epsilon^{\delta\varepsilon\eta} q^C_{m\varepsilon}\right)\left(\bar{q}_{k\gamma} q_{n\eta}\right)$$

We can simplify the relevant terms in $\mathcal{L}_{NJL}$:

$$\mathcal{L}_{NJL} = G_S\left[(\bar{u}u)^2 + \left(\bar{d}d\right)^2 + (\bar{s}s)^2\right]$$

and in $\mathcal{L}_{'tHooft}$:

$$\mathcal{L}_{NJL} = G_D\left(\bar{u}u\bar{d}d\bar{s}s\right)$$

Using the definition:

$$\Delta^{k\gamma} = \left\langle \bar{q} i\gamma^5 \epsilon\epsilon q^C\right\rangle$$

and the ansatzes:

$$(\bar{q}_1 q_2)(\bar{q}_3 q_4) \rightarrow \bar{q}_1 q_2 \left\langle \bar{q}_3 q_4\right\rangle + \bar{q}_3 q_4 \left\langle \bar{q}_1 q_2\right\rangle - \left\langle \bar{q}_1 q_2\right\rangle \left\langle \bar{q}_3 q_4\right\rangle$$

$$(\bar{q}_1 q_2)(\bar{q}_3 q_4)(\bar{q}_5 q_6) \rightarrow \bar{q}_1 q_2 \left\langle \bar{q}_3 q_4\right\rangle \left\langle \bar{q}_5 q_6\right\rangle + \bar{q}_3 q_4 \left\langle \bar{q}_1 q_2\right\rangle \left\langle \bar{q}_5 q_6\right\rangle + \bar{q}_5 q_6 \left\langle \bar{q}_1 q_2\right\rangle \left\langle \bar{q}_3 q_4\right\rangle - 2\left\langle \bar{q}_1 q_2\right\rangle \left\langle \bar{q}_3 q_4\right\rangle \left\langle \bar{q}_5 q_6\right\rangle$$

for the mean field approximation, we can rewrite the Lagrangian

$$\mathcal{L}_{NJL} = 2G_S\left[\left(\bar{u}u\right)\left\langle \bar{u}u\right\rangle + \left(\bar{d}d\right)\left\langle \bar{d}d\right\rangle + \left(\bar{s}s\right)\left\langle \bar{s}s\right\rangle - \left\langle \bar{u}u\right\rangle^2 - \left\langle \bar{d}d\right\rangle^2 - \left\langle \bar{s}s\right\rangle^2\right]$$

$$\mathcal{L}_{'tHooft} = -2G_D\left[\left(\bar{u}u\right)\left\langle \bar{u}u\right\rangle\left\langle \bar{s}s\right\rangle + \left(\bar{d}d\right)\left\langle \bar{u}u\right\rangle\left\langle \bar{s}s\right\rangle + \left(\bar{s}s\right)\left\langle \bar{u}u\right\rangle\left\langle \bar{d}d\right\rangle - 2\left\langle \bar{u}u\right\rangle\left\langle \bar{d}d\right\rangle\left\langle \bar{s}s\right\rangle\right]$$

$$\mathcal{L}_{SC} = G_{DIQ}\left[\Delta^{k\gamma}\left(\bar{q}_{\ell\delta} i\gamma^5 \epsilon^{\ell mk}\epsilon^{\delta\varepsilon\gamma} q^C_{m\varepsilon}\right) + \left(\bar{q}_{i\alpha} i\gamma^5 \varepsilon^{ijk}\varepsilon^{\alpha\beta\gamma} q^C_{j\beta}\right)\Delta^{k\gamma\dagger} - \Delta^{k\gamma}\Delta^{k\gamma\dagger}\right]$$

$$\mathcal{L}_{SC6} = K_D\left[\left(\bar{q}_{m\varepsilon} q_{n\eta}\right)\Delta^{k\gamma}\Delta^{m\varepsilon\dagger} + \left(\bar{q}_{i\alpha} i\gamma^5 \varepsilon^{ijk}\varepsilon^{\alpha\beta\gamma} q^C_{j\beta}\right)\Delta^{m\varepsilon\dagger}\left\langle \bar{q}_{m\varepsilon} q_{n\eta}\right\rangle\right]$$
$$+K_D\left[\Delta^{k\gamma}\left(\bar{q}_{\ell\delta} i\gamma^5 \epsilon^{\ell mn}\epsilon^{\delta\varepsilon\eta} q^C_{m\varepsilon}\right)\left\langle \bar{q}_{m\varepsilon} q_{n\eta}\right\rangle - 2\Delta^{k\gamma}\Delta^{m\varepsilon\dagger}\left\langle \bar{q}_{m\varepsilon} q_{n\eta}\right\rangle\right]$$

If we make the definition $\tilde{\Delta} = 2G_{DIQ}\Delta$

Definition at line 184 of file cfl6_eos_old.h.

Public Member Functions

- virtual int calc_eq_temp_p (**quark** &u, **quark** &d, **quark** &s, double &qq1, double &qq2, double &qq3, double &gap1, double &gap2, double &gap3, double mu3, double mu8, double &n3, double &n8, **thermo** &qb, const double temper)
  Calculate the EOS.
- virtual double thd_potential (**quark** &u, **quark** &d, **quark** &s, double mu3, double mu8, const double ltemper)
  Direct calculation of the thermodynamic potential.
- virtual int eigenvalues (double mom, **ovector_view** &egv, double mu3, double mu8)
  Calculate the energy eigenvalues as a function of the momentum.
- virtual const char * type ()
  Return string denoting type ("cfl6_eos_old").

Data Fields

- double KD
  The color superconducting 't Hooft coupling (default 0).
- double kdlimit
  The absolute value below which the CSC 't Hooft coupling is ignored(default 10^{-6}).

Protected Member Functions

- virtual double tpot (double var, void *&pa)
  The function used to take derivatives of the thermodynamic potential (used by calc_eq_temp_p()).

Protected Attributes

- gsl_complex zero
  To clear all of the matrix entries.
- **omatrix_cx** m6
  Storage for the inverse propagator.
- **ovector** eval6
  Storage for the eigenvalues.
- gsl_eigen_herm_workspace * w6
  GSL workspace for the eigenvalue computation.

Static Protected Attributes

- static const int size = 36
  The size of the matrix to be diagonalized.

Private Member Functions

- **cfl6_eos_old** (const cfl6_eos_old &)
- cfl6_eos_old & **operator=** (const cfl6_eos_old &)

5.6.2 Member Function Documentation

5.6.2.1 virtual int calc_eq_temp_p (quark & *u*, quark & *d*, quark & *s*, double & *qq1*, double & *qq2*, double & *qq3*, double & *gap1*, double & *gap2*, double & *gap3*, double *mu3*, double *mu8*, double & *n3*, double & *n8*, thermo & *qb*, const double *temper*) [virtual]

Calculate the EOS.

Calculate the EOS from the **quark** condensates. Return the mass gap equations in `qq1`, `qq2`, `qq3`, and the normal gap equations in `gap1`, `gap2`, and `gap3`.

Using `fromqq=true` as in nambujl_eos and nambujl_temp_eos does not work here and will return an error. The quarks must be set through quark_eos::quark_set() before use.

If all of the gaps are less than gap_limit, then the nambujl_temp_eos::calc_temp_p() is used, and `gap1`, `gap2`, and `gap3` are set to equal `u.del`, `d.del`, and `s.del`, respectively.

Reimplemented from cfl_njl_eos_old.

5.6.2.2 virtual int eigenvalues (double *mom*, ovector_view & *egv*, double *mu3*, double *mu8*) [virtual]

Calculate the energy eigenvalues as a function of the momentum.

Given the momentum 'mom', and the chemical potentials associated with the third and eighth gluons ('mu3' and 'mu8'), the energy eigenvalues are computed in egv[0] ... egv[35]. No space is allocated for the array by the function.

Todo

This function may make some inappropriate assumptions on the vector egv.

Reimplemented from cfl_njl_eos_old.

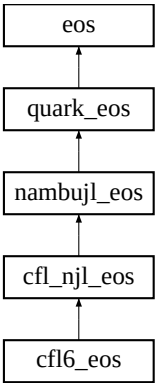
The documentation for this class was generated from the following file:

- cfl6_eos_old.h

5.7 cfl_njl_eos Class Reference

`#include <cfl_njl_eos.h>`

Inheritance diagram for cfl_njl_eos::



5.7.1 Detailed Description

Nambu Jona-Lasinio model with a schematic CFL di-quark interaction at finite temperature.

The variable B0 must be set before use.

The original Lagrangian is

$$\mathcal{L} = \mathcal{L}_{\rm Dirac} + \mathcal{L}_{4-{\rm fermion}} + \mathcal{L}_{6-{\rm fermion}} + \mathcal{L}_{CSC1} + \mathcal{L}_{CSC2}$$

$$\mathcal{L}_{\rm Dirac} = \bar{q}_{i\alpha} \left(i\partial \delta_{ij} \delta_{\alpha\beta} - m_{ij} \delta_{\alpha\beta} - \mu_{ij,\ \alpha\beta} \gamma^0 \right) q_{j\beta}$$

$$\mathcal{L}_{4-{\rm fermion}} = G_S \sum_{a=0}^{8} \left[\left(\bar{q} \lambda_f^a q \right)^2 + \left(\bar{q} i \gamma_5 \lambda_f^a q \right)^2 \right]$$

$$\mathcal{L}_{6-{\rm fermion}} = -G_D \left[{\rm det}_{ij}\ \bar{q}_{i\alpha} \left(1 + i\gamma_5 \right) q_{j\beta} + {\rm det}_{ij}\ \bar{q}_{i\alpha} \left(1 - i\gamma_5 \right) q_{j\beta} \right] \delta_{\alpha\beta}$$

$$\mathcal{L}_{CSC1} = G_{DIQ} \sum_k \sum_\gamma \left[\left(\bar{q}_{i\alpha} \epsilon_{ijk} \epsilon_{\alpha\beta\gamma} q^C_{j\beta} \right) \left(\bar{q}^C_{i'\alpha'} \epsilon_{i'j'k} \epsilon_{\alpha'\beta'\gamma} q_{j'\beta'} \right) \right]$$

$$\mathcal{L}_{CSC2} = G_{DIQ} \sum_k \sum_\gamma \left[\left(\bar{q}_{i\alpha} i\gamma_5 \epsilon_{ijk} \epsilon_{\alpha\beta\gamma} q^C_{j\beta} \right) \left(\bar{q}^C_{i'\alpha'} i\gamma_5 \epsilon_{i'j'k} \epsilon_{\alpha'\beta'\gamma} q_{j'\beta'} \right) \right] ,$$

where μ is the **quark** number chemical potential. couplings G_S, G_D, and G_{DIQ} ultra-violet three-momentum cutoff, Λ

The thermodynamic potential is

$$\Omega(\mu_i, \langle \bar{q}q \rangle_i, \langle qq \rangle_i, T) = \Omega_{\rm vac} + \Omega_{\rm stat} + \Omega_0$$

where i runs over all nine (three colors times three flavors) quarks. We assume that the condensates are independent of color and that the **quark** chemical potentials are of the form $\mu_Q = \mu_{\rm Flavor(Q)} + \mu_{\rm Color(Q)}$ with

$$\mu_{\rm red} = \mu_3 + \mu_8/\sqrt{3} \quad \mu_{\rm green} = -\mu_3 + \mu_8/\sqrt{3} \quad \mu_{\rm blue} = -2\mu_8/\sqrt{3}$$

With these assumptions, the thermodynamic potential as given by the function thd_potential(), is a function of 12 variables

$$\Omega(\mu_u, \mu_d, \mu_s, \mu_3, \mu_8, \langle \bar{u}u \rangle, \langle \bar{d}d \rangle, \langle \bar{s}s \rangle, \langle ud \rangle, \langle us \rangle, \langle ds \rangle, T)$$

The individual terms are

$$\Omega_{\rm stat} = -\frac{1}{2} \int \frac{d^3p}{(2\pi)^3} \sum_{i=1}^{72} \left[\frac{\lambda_i}{2} + T \ln \left(1 + e^{-\lambda_i/T} \right) \right]$$

$$\Omega_{\rm vac} = -2G_S \sum_{i=u,d,s} \langle \bar{q}_i q_i \rangle^2 + 4G_D \langle \bar{u}u \rangle \langle \bar{d}d \rangle \langle \bar{s}s \rangle + \sum_k \sum_\gamma \frac{\left| \Delta^{k\gamma} \right|^2}{4G_{DIQ}}$$

where λ_i are the eigenvalues of the (72 by 72) matrix (calculated by the function eigenvalues())

$$D = \begin{bmatrix} -\gamma^0 \vec{\gamma} \cdot \vec{p} - M_i \gamma^0 + \mu_{i\alpha} & \Delta i \gamma^0 \gamma_5 C \\ i\Delta \gamma^0 C \gamma_5 & -\gamma^0 \vec{\gamma}^T \cdot \vec{p} + M_i \gamma^0 - \mu_{i\alpha} \end{bmatrix}$$

and C is the charge conjugation matrix (in the Dirac representation).

The values of the various condensates are usually determined by the condition

$$\frac{\partial \Omega}{\langle \bar{q}q \rangle_i} = 0 \quad \frac{\partial \Omega}{\langle qq \rangle_i} = 0$$

Note that setting fixed_mass to `true` and setting all of the gaps to zero when `gap_limit` is less than zero will reproduce an analog of the bag model with a momentum cutoff.

The variable nambujl_eos::fromqq is automatically set to true in the constructor, as computations with `fromqq=false` are not implemented.

Todo

Improve documentation (Note that it appears to report the member functions twice?)

Todo

Allow user to change derivative object? This isn't possible right now because the stepsize parameter of the derivative object is used.

Todo

This class internally mixes ovector, omatrix, gsl_vector and gsl_matrix objects in a confusing and non-optimal way. Fix this.

Definition at line 205 of file cfl_njl_eos.h.

Public Member Functions

- virtual int set_parameters (double lambda=0.0, double fourferm=0.0, double sixferm=0.0, double fourgap=0.0)
  Set the parameters and the bag constant 'B0'.
- virtual int calc_eq_temp_p (**quark** &u, **quark** &d, **quark** &s, double &qq1, double &qq2, double &qq3, double &gap1, double &gap2, double &gap3, double mu3, double mu8, double &n3, double &n8, **thermo** &qb, const double temper)
  Calculate the EOS.
- virtual int test_derivatives (double lmom, double mu3, double mu8, **test_mgr** &t)
  Check the derivatives specified by eigenvalues().
- virtual int eigenvalues (double lmom, double mu3, double mu8, double egv[36], double dedmuu[36], double dedmud[36], double dedmus[36], double dedmu[36], double dedmd[36], double dedms[36], double dedu[36], double dedd[36], double deds[36], double dedmu3[36], double dedmu8[36])
  Calculate the energy eigenvalues as a function of the momentum.
- int set_quartic (**quartic_real_coeff** &q)
  Set the routine for solving quartics.
- int test_integration (**test_mgr** &t)
  Test the integration routines.
- int test_normal_eigenvalues (**test_mgr** &t)
  Test the routine to compute the eigenvalues of non-superfluid fermions.
- int test_gapped_eigenvalues (**test_mgr** &t)
  Test the routine to compute the eigenvalues of superfluid fermions.
- virtual const char * type ()
  Return string denoting type ("cfl_njl_eos").

Data Fields

- double eq_limit
  The equal mass threshold.
- bool integ_test
  Set to true to test the integration (default false).
- **cern_quartic_real_coeff** def_quartic
  The default quartic routine.
- double gap_limit
  Smallest allowable gap (default 0.0).
- bool zerot
  If this is true, then finite temperature corrections are ignored (default false).
- bool fixed_mass
  Use a fixed **quark** *mass and ignore the* **quark** *condensates.*
- bool color_neut
  If true, then ensure color neutrality.
- double GD
  Diquark coupling constant (default 3 G/4).
- double inte_epsabs
  The absolute precision for the integration (default 10^{-4}).
- double inte_epsrel
  The relative precision for the integration (default 10^{-4}).
- size_t inte_npoints
  The number of points used in the last integration (default 0).

Protected Member Functions

- virtual int integrands (double p, double res[ ])
  The integrands.
- int normal_eigenvalues (double m, double lmom, double mu, double lam[2], double dldmu[2], double dldm[2])
  Compute ungapped eigenvalues and the appropriate derivatives.
- int gapped_eigenvalues (double m1, double m2, double lmom, double mu1, double mu2, double tdelta, double lam[4], double dldmu1[4], double dldmu2[4], double dldm1[4], double dldm2[4], double dldg[4])

Treat the simply gapped quarks in all cases gracefully.

For the integration

- double rescale_error (double err, const double result_abs, const double result_asc)
  The error scaling function for integ_err.
- int integ_err (double a, double b, const size_t nr, **ovector** &res, double &err2)
  A new version of **gsl_inte_qng** *to integrate several functions at the same time.*

Protected Attributes

- double temper
  Temperature.
- double smu3
  3rd gluon chemical potential
- double smu8
  8th gluon chemical potential

Numerical methods

- **quartic_real_coeff** * quartic
  The routine to solve quartics.

For computing eigenvalues

- **omatrix_cx** iprop
  Inverse propagator matrix.
- **omatrix_cx** eivec
  The eigenvectors.
- **omatrix_cx** dipdgapu
  The derivative of the inverse propagator wrt the ds gap.
- **omatrix_cx** dipdgapd
  The derivative of the inverse propagator wrt the us gap.
- **omatrix_cx** dipdgaps
  The derivative of the inverse propagator wrt the ud gap.
- **ovector** eval
  The eigenvalues.
- gsl_eigen_hermv_workspace * w
  Workspace for eigenvalue computation.

Private Member Functions

- **cfl_njl_eos** (const cfl_njl_eos &)
- cfl_njl_eos & **operator=** (const cfl_njl_eos &)

5.7.2 Member Function Documentation

5.7.2.1 virtual int set_parameters (double *lambda* = `0.0`, double *fourferm* = `0.0`, double *sixferm* = `0.0`, double *fourgap* = `0.0`) `[virtual]`

Set the parameters and the bag constant 'B0'.

This function allows the user to specify the momentum cutoff, `lambda`, the four-fermion coupling `fourferm`, the six-fermion coupling from the 't Hooft interaction `sixferm`, and the color-superconducting coupling, `fourgap`. If 0.0 is given for any of the values, then the default is used ($\Lambda = 602.3/(\hbar c), G = 1.835/\Lambda^2, K = 12.36/\Lambda^5$).

If the four-fermion coupling that produces a gap is not specified, it is automatically set to 3/4 G, which is the value obtained from the Fierz transformation.

The value of the shift in the bag constant nambujl_eos::B0 is automatically calculated to ensure that the vacuum has zero energy density and zero pressure. The functions set_quarks() and set_thermo() must be used before hand to specify the **quark** and **thermo** objects.

5.7.2.2 virtual int calc_eq_temp_p (quark & *u*, quark & *d*, quark & *s*, double & *qq1*, double & *qq2*, double & *qq3*, double & *gap1*, double & *gap2*, double & *gap3*, double *mu3*, double *mu8*, double & *n3*, double & *n8*, thermo & *qb*, const double *temper*) [virtual]

Calculate the EOS.

Calculate the EOS from the **quark** condensates in `u.qq`, `d.qq` and `s.qq`. Return the mass gap equations in `qq1`, `qq2`, `qq3`, and the normal gap equations in `gap1`, `gap2`, and `gap3`.

Using `fromqq=false` as in nambujl_eos and nambujl_eos does not work here and will return an error. Also, the quarks must be set through quark_eos::quark_set() before use.

If all of the gaps are less than gap_limit, then the nambujl_eos::calc_temp_p() is used, and `gap1`, `gap2`, and `gap3` are set to equal `u.del`, `d.del`, and `s.del`, respectively.

Todo

It surprises me that n3 is not -res[11]. Is there a sign error in the color densities?

Reimplemented in cfl6_eos.

5.7.2.3 virtual int eigenvalues (double *lmom*, double *mu3*, double *mu8*, double *egv*[36], double *dedmuu*[36], double *dedmud*[36], double *dedmus*[36], double *dedmu*[36], double *dedmd*[36], double *dedms*[36], double *dedu*[36], double *dedd*[36], double *deds*[36], double *dedmu3*[36], double *dedmu8*[36]) [virtual]

Calculate the energy eigenvalues as a function of the momentum.

Given the momentum `mom`, and the chemical potentials associated with the third and eighth gluons (`mu3` and `mu8`), the energy eigenvalues are computed in egv[0] ... egv[35].

5.7.2.4 virtual int integrands (double *p*, double *res*[ ]) [protected, virtual]

The integrands.

- res[0] is the thermodynamic potential, Ω
- res[1] is $d - \Omega/dT$
- res[2] is $d\Omega/d\mu_u$
- res[3] is $d\Omega/d\mu_d$
- res[4] is $d\Omega/d\mu_s$
- res[5] is $d\Omega/dm_u$
- res[6] is $d\Omega/dm_d$
- res[7] is $d\Omega/dm_s$
- res[8] is $d\Omega/d\Delta_{ds}$
- res[9] is $d\Omega/d\Delta_{us}$
- res[10] is $d\Omega/d\Delta_{ud}$
- res[11] is $d\Omega/d\mu_3$
- res[12] is $d\Omega/d\mu_8$

Reimplemented in cfl6_eos.

5.7.2.5 int gapped_eigenvalues (double *m1*, double *m2*, double *lmom*, double *mu1*, double *mu2*, double *tdelta*, double *lam*[4], double *dldmu1*[4], double *dldmu2*[4], double *dldm1*[4], double *dldm2*[4], double *dldg*[4]) [protected]

Treat the simply gapped quarks in all cases gracefully.

This function uses the quarks `q1` and `q2` to construct the eigenvalues of the inverse propagator, properly handling the either zero or finite **quark** mass and either zero or finite **quark** gaps. In the case of finite **quark** mass and finite **quark** gaps, the quartic solver is used.

The chemical potentials are separated so we can add the color chemical potentials to the **quark** chemical potentials if necessary.

This function is used by eigenvalues(). It does not work for the "ur-dg-sb" set of quarks which are paired in a non-trivial way.

Todo

Only the "ms" **part** of the quarks is referenced, so we should rewrite to use only double's as function arguments, and avoid passing pointers to **quark** objects.

5.7.3 Field Documentation

5.7.3.1 cern_quartic_real_coeff def_quartic

The default quartic routine.

Slightly better accuracy (with slower execution times) can be achieved using **gsl_poly_real_coeff** which polishes the roots of the quartics. For example

```
cfl_njl_eos cfl;
gsl_poly_real_coeff gp;
cfl.set_quartic(gp);
```

Definition at line 315 of file cfl_njl_eos.h.

5.7.3.2 double gap_limit

Smallest allowable gap (default 0.0).

If any of the gaps are below this value, then it is assumed that they are zero and the equation of state is simplified accordingly. If all of the gaps are less than gap_limit, then the results from nambujl_eos are used in calc_eq_temp_p(), calc_temp_p() and thd_potential().

Definition at line 339 of file cfl_njl_eos.h.

5.7.3.3 bool zerot

If this is true, then finite temperature corrections are ignored (default false).

This implements some simplifications in the momentum integration that are not possible at finite temperature.

Definition at line 348 of file cfl_njl_eos.h.

5.7.3.4 double GD

Diquark coupling constant (default 3 G/4).

The default value is the one derived from a Fierz transformation. (Buballa04)

Definition at line 365 of file cfl_njl_eos.h.

5.7.3.5 double inte_epsabs

The absolute precision for the integration (default 10^{-4}).

This is analogous to gsl_inte::epsabs

Definition at line 373 of file cfl_njl_eos.h.

5.7.3.6 double inte_epsrel

The relative precision for the integration (default 10^{-4}).

This is analogous to gsl_inte::epsrel

Definition at line 381 of file cfl_njl_eos.h.

5.7.3.7 size_t inte_npoints

The number of points used in the last integration (default 0).

This returns 21, 43, or 87 depending on the number of function evaluations needed to obtain the desired precision. If it the routine failes to obtain the desired precision, then this variable is set to 88.

Definition at line 392 of file cfl_njl_eos.h.

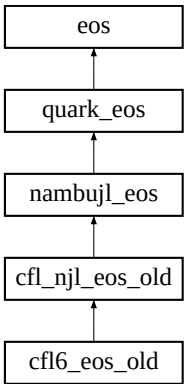
The documentation for this class was generated from the following file:

- cfl_njl_eos.h

5.8 cfl_njl_eos_old Class Reference

`#include <cfl_njl_eos_old.h>`

Inheritance diagram for cfl_njl_eos_old::



5.8.1 Detailed Description

Nambu Jona-Lasinio model with a schematic CFL di-quark interaction at finite temperature.

The variable B0 must be set before use.

The original Lagrangian is

$$\mathcal{L} = \mathcal{L}_{\rm Dirac} + \mathcal{L}_{4-{\rm fermion}} + \mathcal{L}_{6-{\rm fermion}} + \mathcal{L}_{CSC1} + \mathcal{L}_{CSC2}$$

$$\mathcal{L}_{\rm Dirac} = \bar{q}_{i\alpha}\left(i\partial\delta_{ij}\delta_{\alpha\beta} - m_{ij}\delta_{\alpha\beta} - \mu_{ij,\ \alpha\beta}\gamma^0\right)q_{j\beta}$$

$$\mathcal{L}_{4-\mathrm{fermion}} = G_S \sum_{a=0}^{8} \left[\left(\bar{q} \lambda_f^a q \right)^2 + \left(\bar{q} i \gamma_5 \lambda_f^a q \right)^2 \right]$$

$$\mathcal{L}_{6-\mathrm{fermion}} = -G_D \left[\mathrm{d}et_{ij} \, \bar{q}_{i\alpha} \left(1 + i\gamma_5 \right) q_{j\beta} + \mathrm{d}et_{ij} \, \bar{q}_{i\alpha} \left(1 - i\gamma_5 \right) q_{j\beta} \right] \delta_{\alpha\beta}$$

$$\mathcal{L}_{CSC1} = G_{DIQ} \sum_k \sum_\gamma \left[\left(\bar{q}_{i\alpha} \epsilon_{ijk} \epsilon_{\alpha\beta\gamma} q^C_{j\beta} \right) \left(\bar{q}^C_{i'\alpha'} \epsilon_{i'j'k} \epsilon_{\alpha'\beta'\gamma} q_{j'\beta'} \right) \right]$$

$$\mathcal{L}_{CSC2} = G_{DIQ} \sum_k \sum_\gamma \left[\left(\bar{q}_{i\alpha} i\gamma_5 \epsilon_{ijk} \epsilon_{\alpha\beta\gamma} q^C_{j\beta} \right) \left(\bar{q}^C_{i'\alpha'} i\gamma_5 \epsilon_{i'j'k} \epsilon_{\alpha'\beta'\gamma} q_{j'\beta'} \right) \right] ,$$

where μ is the **quark** number chemical potential. couplings G_S, G_D, and G_{DIQ} ultra-violet three-momentum cutoff, Λ

The thermodynamic potential is

$$\Omega(\mu_i, \langle \bar{q} q \rangle_i , \langle qq \rangle_i , T) = \Omega_{\mathrm{vac}} + \Omega_{\mathrm{stat}} + \Omega_0$$

where i runs over all nine (three colors times three flavors) quarks. We assume that the condensates are independent of color and that the **quark** chemical potentials are of the form $\mu_Q = \mu_{\mathrm{Flavor(Q)}} + \mu_{\mathrm{Color(Q)}}$ with

$$\mu_{\mathrm{red}} = \mu_3 + \mu_8/\sqrt{3} \quad \mu_{\mathrm{green}} = -\mu_3 + \mu_8/\sqrt{3} \quad \mu_{\mathrm{blue}} = -2\mu_8/\sqrt{3}$$

With these assumptions, the thermodynamic potential as given by the function thd_potential(), is a function of 12 variables

$$\Omega(\mu_u, \mu_d, \mu_s, \mu_3, \mu_8, \langle \bar{u} u \rangle , \langle \bar{d} d \rangle , \langle \bar{s} s \rangle , \langle ud \rangle , \langle us \rangle , \langle ds \rangle , T)$$

The individual terms are

$$\Omega_{\mathrm{stat}} = -\frac{1}{2} \int \frac{d^3 p}{(2\pi)^3} \sum_{i=1}^{72} \left[\frac{\lambda_i}{2} + T \ln \left(1 + e^{-\lambda_i / T} \right) \right]$$

$$\Omega_{\mathrm{vac}} = -2G_S \sum_{i=u,d,s} \langle \bar{q}_i q_i \rangle^2 + 4G_D \langle \bar{u} u \rangle \langle \bar{d} d \rangle \langle \bar{s} s \rangle + \sum_k \sum_\gamma \frac{\left| \Delta^{k\gamma} \right|^2}{4G_{DIQ}}$$

where λ_i are the eigenvalues of the (72 by 72) matrix (calculated by the function eigenvalues())

$$D = \begin{bmatrix} -\gamma^0 \vec{\gamma} \cdot \vec{p} - M_i \gamma^0 + \mu_{i\alpha} & \Delta i \gamma^0 \gamma_5 C \\ i \Delta \gamma^0 C \gamma_5 & -\gamma^0 \vec{\gamma}^T \cdot \vec{p} + M_i \gamma^0 - \mu_{i\alpha} \end{bmatrix}$$

and C is the charge conjugation matrix (in the Dirac representation).

The values of the various condensates are usually determined by the condition

$$\frac{\partial \Omega}{\langle \bar{q} q \rangle_i} = 0 \quad \frac{\partial \Omega}{\langle qq \rangle_i} = 0$$

Note that setting fixed_mass to `true` and setting all of the gaps to zero when `gap_limit` is less than zero will reproduce an analog of the bag model with a momentum cutoff.

The variable nambujl_eos::fromqq is automatically set to true in the constructor, as computations with `fromqq=false` are not implemented.

Idea for future

This class internally mixes ovector, omatrix, gsl_vector and gsl_matrix objects in a confusing and non-optimal way. Fix this.

Idea for future

Allow user to change derivative object? This isn't possible right now because the stepsize parameter of the derivative object is used.

Definition at line 202 of file cfl_njl_eos_old.h.

Numerical methods

- **gsl_mmin_conf**< void ∗, **multi_funct**< void ∗ > > def_min1
  The first default minimizer.
- **gsl_mmin_conf**< void ∗, **multi_funct**< void ∗ > > def_min2
  The second default minimizers.
- **cern_quartic_real_coeff** def_quartic
  The default quartic routine.
- int set_minimizers (**multi_min**< void ∗, **multi_funct**< void ∗ > > &mi1, **multi_min**< void ∗, **multi_funct**< void ∗ > > &mi2)
  Set the minimizers for the thermodynamic potential.
- int set_quartic (**quartic_real_coeff** &q)
  Set the routine for solving quartics.

Public Member Functions

- virtual int set_parameters (double lambda=0.0, double fourferm=0.0, double sixferm=0.0, double fourgap=0.0)
  Set the parameters and the bag constant 'B0'.
- virtual int calc_temp_p (**quark** &u, **quark** &d, **quark** &s, const double temper, **thermo** &th)
  Calculate equation of state as a function of chemical potentials.
- virtual int calc_eq_temp_p (**quark** &u, **quark** &d, **quark** &s, double &qq1, double &qq2, double &qq3, double &gap1, double &gap2, double &gap3, double mu3, double mu8, double &n3, double &n8, **thermo** &qb, const double temper)
  Calculate the EOS.
- virtual int eigenvalues (double mom, **ovector_view** &egv, double mu3, double mu8)
  Calculate the energy eigenvalues as a function of the momentum.
- virtual double thd_potential (**quark** &u, **quark** &d, **quark** &s, double mu3, double mu8, const double ltemper)
  Direct calculation of the thermodynamic potential.
- virtual const char ∗ type ()
  Return string denoting type ("cfl_njl_eos_old").
- virtual double tpot (double var, void ∗&pa)
  The function used to take derivatives of the thermodynamic potential (used by calc_temp_p()).

Data Fields

- double gap_limit
  Smallest allowable gap (default 0.0).
- bool zerot
  If this is true, then finite temperature corrections are ignored (default false).
- bool fixed_mass
  Use a fixed **quark** *mass and ignore the* **quark** *condensates.*
- bool color_neut
  If true, then ensure color neutrality.
- double GD
  Diquark coupling constant (default 3 G/4).
- bool faster

Faster eigenvalues.

- double solver_qual
  The quality of the most recently obtained solution.
- bool use_solver
  If true, use the solver, otherwise, use the minimizers.

Protected Member Functions

- int solve_fun_p (size_t nv, const **ovector_view** &x, **ovector_view** &y, void *&vp)
  Function specifying gap equations to solve.
- int min_fun2 (size_t nv, const **ovector_view** &x, double &y, void *&vp)
  Outer minimization function for color neutrality.
- int min_fun (size_t nv, const **ovector_view** &x, double &y, void *&vp)
  Inner minimization function for mass and gap equations.
- double derivfn (double *dvar)
  A function that takes the derivative of the thermodynamic potential with respect to the variable whose address is 'dvar'.
- double tpot_integrand (double p, void *&pa)
  The integrand of the thermodynamic potential.
- double entropy_integrand (double p, void *&pa)
  The integrand of the entropy.
- int gapped_eigenvalues (**quark** *q1, **quark** *q2, double mom, double mu1, double mu2, double delta, **ovector_cx_view** &ev)
  Treat the simply gapped quarks in all cases gracefully.

Protected Attributes

- double temper
  Temperature.
- double smu3
  3rd gluon chemical potential
- double smu8
  8th gluon chemical potential
- double * var
  Pointer to the variable to be differentiated by.

Numerical methods

- **quartic_real_coeff** * quartic
  The routine to solve quartics.
- bool quartic_set
  true if the quartic routine has been set
- **gsl_deriv**< void *, **funct**< void * > > de
  The derivative routine.
- **multi_min**< void *, **multi_funct**< void * > > * min1
  Pointer to inner minimizer.
- **multi_min**< void *, **multi_funct**< void * > > * min2
  Pointer to outer minimizer.

For computing eigenvalues

- **omatrix_cx** m
  Inverse propagator matrix.
- **omatrix_cx** m2
  Inverse propagator matrix.
- **ovector** eval
  Storage for eigenvalues.
- **ovector** eval2

Storage for eigenvalues.

- gsl_eigen_herm_workspace * w
  Workspace for eigenvalue computation.
- gsl_eigen_herm_workspace * w2
  Workspace for eigenvalue computation.

- bool ugap_solve
  Switches for solving the gap equations.
- bool **dgap_solve**
- bool **sgap_solve**

Private Member Functions

- **cfl_njl_eos_old** (const cfl_njl_eos_old &)
- cfl_njl_eos_old & **operator=** (const cfl_njl_eos_old &)

5.8.2 Member Function Documentation

5.8.2.1 virtual int set_parameters (double *lambda* = `0.0`, double *fourferm* = `0.0`, double *sixferm* = `0.0`, double *fourgap* = `0.0`) `[virtual]`

Set the parameters and the bag constant 'B0'.

This function allows the user to specify the momentum cutoff, `lambda`, the four-fermion coupling `fourferm`, the six-fermion coupling from the 't Hooft interaction `sixferm`, and the color-superconducting coupling, `fourgap`. If 0.0 is given for any of the values, then the default is used ($\Lambda = 602.3/(\hbar c), G = 1.835/\Lambda^2, K = 12.36/\Lambda^5$).

If the four-fermion coupling that produces a gap is not specified, it is automatically set to 3/4 G, which is the value obtained from the Fierz transformation.

The value of the shift in the bag constant nambujl_eos::B0 is automatically calculated to ensure that the vacuum has zero energy density and zero pressure. The functions set_quarks() and set_thermo() must be used before hand to specify the **quark** and **thermo** objects.

5.8.2.2 virtual int calc_temp_p (quark & *u*, quark & *d*, quark & *s*, const double *temper*, thermo & *th*) `[virtual]`

Calculate equation of state as a function of chemical potentials.

If any of the gaps given in **quark::del** are greater than gap_limit, then the respective gap equations are solved. If fixed_mass is false, then the **quark** condensates are solved for. If color_neut is true, then color neutrality is enforced.

Reimplemented from nambujl_eos.

5.8.2.3 virtual int calc_eq_temp_p (quark & *u*, quark & *d*, quark & *s*, double & *qq1*, double & *qq2*, double & *qq3*, double & *gap1*, double & *gap2*, double & *gap3*, double *mu3*, double *mu8*, double & *n3*, double & *n8*, thermo & *qb*, const double *temper*) `[virtual]`

Calculate the EOS.

Calculate the EOS from the **quark** condensates in `u.qq`, `d.qq` and `s.qq`. Return the mass gap equations in `qq1`, `qq2`, `qq3`, and the normal gap equations in `gap1`, `gap2`, and `gap3`.

Using `fromqq=false` as in nambujl_eos and nambujl_eos does not work here and will return an error. Also, the quarks must be set through quark_eos::quark_set() before use.

If all of the gaps are less than gap_limit, then the nambujl_eos::calc_temp_p() is used, and `gap1`, `gap2`, and `gap3` are set to equal `u.del`, `d.del`, and `s.del`, respectively.

Reimplemented in cfl6_eos_old.

5.8.2.4 virtual int eigenvalues (double *mom*, ovector_view & *egv*, double *mu3*, double *mu8*) [virtual]

Calculate the energy eigenvalues as a function of the momentum.

Given the momentum mom, and the chemical potentials associated with the third and eighth gluons (mu3 and mu8), the energy eigenvalues are computed in egv[0] ... egv[35]. No space is allocated for the array by the function.

Reimplemented in cfl6_eos_old.

5.8.2.5 virtual double tpot (double *var*, void *& *pa*) [virtual]

The function used to take derivatives of the thermodynamic potential (used by calc_temp_p()).

To use this function just to compute the thermodynamic potential, make sure var is 0 (or NULL) and use

```
tpot(0.0,0);
```

since the arguments are both ignored.

Reimplemented in cfl6_eos_old.

5.8.2.6 int gapped_eigenvalues (quark * *q1*, quark * *q2*, double *mom*, double *mu1*, double *mu2*, double *delta*, ovector_cx_view & *ev*) [protected]

Treat the simply gapped quarks in all cases gracefully.

This function uses the quarks q1 and q2 to construct the eigenvalues of the inverse propagator, properly handling the either zero or finite **quark** mass and either zero or finite **quark** gaps. In the case of finite **quark** mass and finite **quark** gaps, the quartic solver is used.

The chemical potentials are separated so we can add the color chemical potentials to the **quark** chemical potentials if necessary.

This function is used by eigenvalues(). It does not work for the "ur-dg-sb" set of quarks which are paired in a non-trivial way.

Todo

Only the "ms" **part** of the quarks is referenced, so we should rewrite to use only double's as function arguments, and avoid passing pointers to **quark** objects.

5.8.3 Field Documentation

5.8.3.1 cern_quartic_real_coeff def_quartic

The default quartic routine.

Slightly better accuracy (with slower execution times) can be achieved using **gsl_poly_real_coeff** which polishes the roots of the quartics. For example

```
cfl_njl_eos_old cfl;
gsl_poly_real_coeff gp;
cfl.set_quartic(gp);
```

Definition at line 324 of file cfl_njl_eos_old.h.

5.8.3.2 double gap_limit

Smallest allowable gap (default 0.0).

If any of the gaps are below this value, then it is assumed that they are zero and the equation of state is simplified accordingly. If all of the gaps are less than gap_limit, then the results from nambujl_eos are used in calc_eq_temp_p(), calc_temp_p() and thd_potential().

Definition at line 336 of file cfl_njl_eos_old.h.

5.8.3.3 bool zerot

If this is true, then finite temperature corrections are ignored (default false).

This implements some simplifications in the momentum integration that are not possible at finite temperature.

Definition at line 344 of file cfl_njl_eos_old.h.

5.8.3.4 double GD

Diquark coupling constant (default 3 G/4).

The default value is the one derived from a Fierz transformation. (Buballa04)

Definition at line 360 of file cfl_njl_eos_old.h.

5.8.3.5 bool faster

Faster eigenvalues.

This assumes that the strange **quark** mass can be used as an effective chemical potential $\mu_s' = \mu_s - m_s^2/2/\mu_s$ and calculates the eigenvalues to first order in the four variables

$$\alpha = \frac{\mu_d - \mu_u}{\mu_u}$$

$$\beta = \frac{\mu_s - \mu_u}{\mu_u}$$

$$\gamma = \frac{\Delta_{us} - \Delta_{ud}}{\Delta_{ud}}$$

$$\delta = \frac{\Delta_{ds} - \Delta_{ud}}{\Delta_{ud}}$$

Definition at line 384 of file cfl_njl_eos_old.h.

5.8.3.6 bool quartic_set [protected]

true if the quartic routine has been set

This is used for the I/O.

Definition at line 480 of file cfl_njl_eos_old.h.

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

- cfl_njl_eos_old.h

5.9 cold_nstar Class Reference

`#include <cold_nstar.h>`

5.9.1 Detailed Description

Naive static cold neutron star.

This uses hadronic_eos::calc_e() to compute the equation of state of zero-temperature beta-equilibrated neutron star matter and tov_solve::mvsr() to compute the mass versus radius curve.

The electron and muon are given masses o2scl_fm_const::mass_electron and o2scl_fm_const::mass_muon, respectively.

The energy density and pressure are both calculated in units fm^{-4} and the baryon density in fm^{-3}

The condition for Urca is the area of the triangle formed by the neutron, proton, and electron Fermi momenta.

Using the definition of the semi-perimeter,

$$s \equiv \left(k_{F,n} + k_{F,p} + k_{F,e}\right)/2$$

Heron's formula gives the triangle area as

$$a = \sqrt{s(s-k_{F,n})(s-k_{F,p})(s-k_{F,e})}$$

The column in the eos **table** labeled `urca` is a^2 . If this quantity is positive, then direct Urca is allowed.

The squared speed of sound (in units of c) is calculated by

$$c_s^2 = \frac{dP}{d\varepsilon}$$

and this is placed in the column labeled `cs2`.

The adabatic index is calculated by

$$\Gamma = \frac{d\ln P}{d\ln \varepsilon}$$

Note that Γ must be greater than $4/3$ at the center of the neutron star for stability. (This is a necessary, but not sufficient condition.)

Note that if the speed of sound is non-monotonic, then calc_eos() will only record the lowest density for which the EOS becomes acausal.

There is an example for the usage of this class given in `examples/ex_cold_nstar.cpp`.

Todo

Ensure that the adiabatic index of the central density is greater than 4/3?

Todo

Implement more stability criteria

Todo

Warn if the EOS becomes pure neutron matter?

Definition at line 100 of file cold_nstar.h.

Output

- double min_bad
  The smallest baryon density where the pressure decreases.
- double allow_urca
  The smallest density where Urca becomes allowed.
- double deny_urca
  The smallest density where Urca becomes disallowed.
- double acausal
  The density at which the EOS becomes acausal.
- double acausal_pr
  The pressure at which the EOS becomes acausal.
- double acausal_ed
  The energy density at which the EOS becomes acausal.
- double solver_tol
  Solver tolerance (default 10^{-4}).
- **table** & get_eos_results ()
  Get the eos **table** *(after having called calc_eos()).*
- **table** & get_tov_results ()
  Get the results from the TOV (after having called calc_nstar()).

The thermodynamic information

- **thermo hb**
- **thermo h**
- **thermo l**

Basic operation

- int set_eos (hadronic_eos &he)
  Set the equation of state.
- int calc_eos (double np_0=0.0)
  Calculate the given equation of state.
- double calc_urca (double np_0=0.0)
  Compute the density at which the direct Urca process is allowed.
- int calc_nstar ()
  Calculate the M vs. R curve.

Public Member Functions

- int set_n_and_p (**fermion** &n, **fermion** &p)
  Set the neutron and proton.
- int set_tov (tov_solve &ts)
  Specify the object for solving the TOV equations.
- int set_root (**root**< void ∗, **funct**< void ∗ > > &rf)
  Set the equation solver for the EOS.

Data Fields

- double nb_start
  The starting baryon density (default 0.05).
- double nb_end
  The final baryon density (default 2.0).
- double dnb
  The baryon density stepsize (default 0.01).
- bool include_muons
  If true, include muons (default false).
- **fermion** def_n
  The default neutron.
- **fermion** def_p
  The default proton.
- tov_solve def_tov
  The default TOV equation solver.
- **cern_mroot_root**< void ∗, **funct**< void ∗ > > def_root
  The default equation solver for the EOS.
- tov_interp_eos def_tov_eos
  Default EOS object for the TOV solver.

Protected Member Functions

- int solve_fun (double x, double &y, void ∗&vp)
  Solve to ensure zero charge in β-equilibrium.

Protected Attributes

- bool eos_set
  True if equation of state has been set.
- **fermion** e
  The electron.
- **fermion** mu
  The muon.
- hadronic_eos * hep
  A pointer to the equation of state.
- **fermion** * np
  A pointer to the neutron.
- **fermion** * pp
  A pointer to the proton.
- tov_solve * tp
  A pointer to the TOV object.
- **root**< void *, **funct**< void * > > * rp
  A pointer to the solver.
- **table** eost
  Storage for the EOS **table**.
- double barn
  The baryon density.

5.9.2 Member Function Documentation

5.9.2.1 int set_eos (hadronic_eos & *he*) [inline]

Set the equation of state.

This should be set before calling calc_eos().

Definition at line 112 of file cold_nstar.h.

5.9.2.2 double calc_urca (double *np_0* = 0.0)

Compute the density at which the direct Urca process is allowed.

This is faster than using calc_eos() since it does nothing other than computes the critical density. It does not store the equation of state.

5.9.2.3 int set_n_and_p (fermion & *n*, fermion & *p*) [inline]

Set the neutron and proton.

The default objects are of type **fermion**, with mass o2scl_fm_const::mass_neutron and o2scl_fm_const::mass_proton. These defaults will give incorrect results for non-relativistic equations of state.

Definition at line 234 of file cold_nstar.h.

5.9.2.4 int set_tov (tov_solve & *ts*) [inline]

Specify the object for solving the TOV equations.

The default uses the low-density equation of state with tov::verbose=0. In calc_nstar(), the units are set by calling tov_solve::set_units().

Definition at line 253 of file cold_nstar.h.

5.9.3 Field Documentation

5.9.3.1 double min_bad

The smallest baryon density where the pressure decreases.

If this is zero after calling calc_eos(), then the pressure does not decrease in the specified range of baryon density

Definition at line 162 of file cold_nstar.h.

5.9.3.2 double allow_urca

The smallest density where Urca becomes allowed.

If this is zero after calling calc_eos(), then direct Urca is never allowed.

Definition at line 170 of file cold_nstar.h.

5.9.3.3 double deny_urca

The smallest density where Urca becomes disallowed.

If this is zero after calling calc_eos(), then direct Urca is not disallowed at a higher density than it becomes allowed.

Definition at line 179 of file cold_nstar.h.

5.9.3.4 double acausal

The density at which the EOS becomes acausal.

If this is zero, then the EOS is causal at all baryon densities in the specified range

Definition at line 187 of file cold_nstar.h.

5.9.3.5 double acausal_pr

The pressure at which the EOS becomes acausal.

If this is zero, then the EOS is causal at all baryon densities in the specified range

Definition at line 195 of file cold_nstar.h.

5.9.3.6 double acausal_ed

The energy density at which the EOS becomes acausal.

If this is zero, then the EOS is causal at all baryon densities in the specified range

Definition at line 203 of file cold_nstar.h.

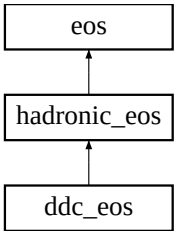
The documentation for this class was generated from the following file:

- cold_nstar.h

5.10 ddc_eos Class Reference

`#include <ddc_eos.h>`

Inheritance diagram for ddc_eos::



5.10.1 Detailed Description

Relativistic mean field EOS with density dependent couplings.

Based on Typel99.

Todo

Finite temperature

Definition at line 48 of file ddc_eos.h.

Public Member Functions

- virtual int calc_eq_e (**fermion** &neu, **fermion** &p, double sig, double ome, double rho, double &f1, double &f2, double &f3, **thermo** &th)
  Equation of state and meson field equations as a function of the density.
- virtual const char * type ()
  Return string denoting type ("ddc_eos").

Data Fields

- double **rho0**

Masses

- double mnuc
  nucleon mass
- double ms
  ϕ mass (in fm^{-1})
- double mw
  A_ω mass (in fm^{-1})
- double mr
  A_ρ mass (in fm^{-1})

Parameters for couplings

- double Gs
  The coupling $\Gamma_\sigma(\rho_{\mathrm{sat}})$.
- double Gw
  The coupling $\Gamma_\omega(\rho_{\mathrm{sat}})$.
- double Gr
  The coupling $\Gamma_\rho(\rho_{\mathrm{sat}})$.
- double as
  a_σ
- double aw
  a_ω

- double ar
  a_ρ
- double bs
  b_σ
- double bw
  b_ω
- double cs
  c_σ
- double cw
  c_ω
- double ds
  d_σ
- double dw
  d_ω

5.10.2 Member Function Documentation

5.10.2.1 virtual int calc_eq_e (fermion & *neu*, fermion & *p*, double *sig*, double *ome*, double *rho*, double & *f1*, double & *f2*, double & *f3*, thermo & *th*) [virtual]

Equation of state and meson field equations as a function of the density.

This calculates the pressure and energy density as a function of $\mu_n, \mu_p, \phi, A_\omega, A_\rho$. When the field equations have been solved, `f1`, `f2`, and `f3` are all zero.

Todo

Is the thermodynamic identity is satisfied even when the field equations are not solved? Check this.

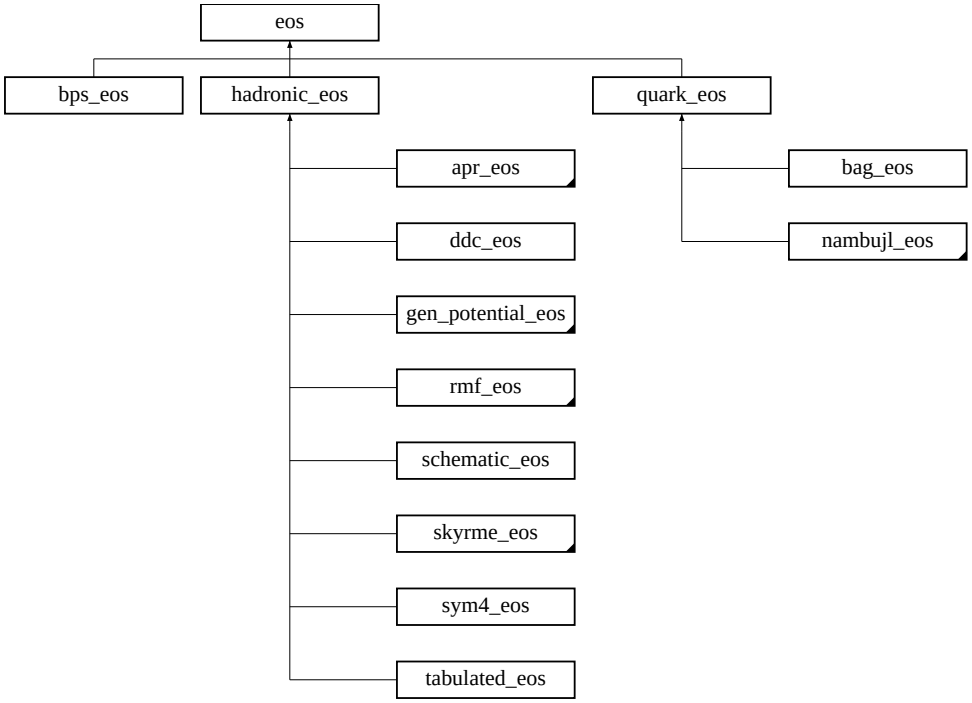
The documentation for this class was generated from the following file:

- ddc_eos.h

5.11 eos Class Reference

`#include <eos.h>`

Inheritance diagram for eos::



5.11.1 Detailed Description

Equation of state base.

A base class for the computation of an equation of state

Definition at line 41 of file eos.h.

Public Member Functions

- virtual int set_thermo (**thermo** &th)
  Set class **thermo** *object.*
- virtual int get_thermo (**thermo** *&th)
  Get class **thermo** *object.*
- virtual const char * type ()
  Return string denoting type ("eos").

Data Fields

- **thermo** def_thermo
  The default **thermo** *object.*

Protected Attributes

- **thermo** * eos_thermo
  A pointer to the **thermo** *object.*

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

- eos.h

5.12 eos_ioc Class Reference

`#include <eos_ioc.h>`

5.12.1 Detailed Description

Setup I/O objects for equation of state classes.

Definition at line 40 of file eos_ioc.h.

Data Fields

- **apr_eos_io_type** * **apr_eos_io**
- **bag_eos_io_type** * **bag_eos_io**
- **gen_potential_eos_io_type** * **gen_potential_eos_io**
- **eos_io_type** * **eos_io**
- **hadronic_eos_io_type** * **hadronic_eos_io**
- **nambujl_eos_io_type** * **nambujl_eos_io**
- **quark_eos_io_type** * **quark_eos_io**
- **rmf_eos_io_type** * **rmf_eos_io**
- **schematic_eos_io_type** * **schematic_eos_io**
- **skyrme_eos_io_type** * **skyrme_eos_io**

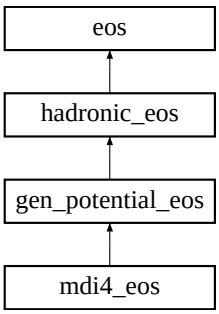
The documentation for this class was generated from the following file:

- eos_ioc.h

5.13 gen_potential_eos Class Reference

`#include <gen_potential_eos.h>`

Inheritance diagram for gen_potential_eos::



5.13.1 Detailed Description

Generalized potential model equation of state.

The single particle energy is defined by the functional derivative of the energy density with respect to the distribution function

$$e_\tau = \frac{\delta \varepsilon}{\delta f_\tau}$$

The effective mass is defined by

$$\frac{m^*}{m} = \left(\frac{m}{k}\frac{de_\tau}{dk}\right)^{-1}_{k=k_F}$$

In all of the models, the kinetic energy density is $\tau_n + \tau_p$ where

$$\tau_i = \frac{2}{(2\pi)^3}\int d^3k \left(\frac{k^2}{2m}\right) f_i(k,T)$$

and the number density is

$$\rho_i = \frac{2}{(2\pi)^3}\int d^3k\, f_i(k,T)$$

When form == mdi_form or gbd_form, the potential energy density is given by Das03 :

$$V(\rho,\delta) = \frac{Au}{\rho_0}\rho_n\rho_p + \frac{A_l}{2\rho_0}\left(\rho_n^2+\rho_p^2\right) + \frac{B}{\sigma+1}\frac{\rho^{\sigma+1}}{\rho_0^\sigma}\left(1-x\delta^2\right) + V_{mom}(\rho,\delta)$$

where $\delta = 1 - 2\rho_p/(\rho_n+\rho_p)$. If form == mdi_form, then

$$V_{mom}(\rho,\delta) = \frac{1}{\rho_0}\sum_{\tau,\tau'} C_{\tau,\tau'}\int\int d^3k d^3k' \frac{f_\tau(\vec{k}) f_{\tau'}(\vec{k}')}{1+(\vec{k}-\vec{k}')^2/\Lambda^2}$$

where $C_{1/2,1/2} = C_{-1/2,-1/2} = C_\ell$ and $C_{1/2,-1/2} = C_{-1/2,1/2} = C_u$. Otherwise if form == gbd_form, then

$$V_{mom}(\rho,\delta) = \frac{1}{\rho_0}\left[C_\ell\left(\rho_n g_n + \rho_p g_p\right) + C_u\left(\rho_n g_p + \rho_p g_n\right)\right]$$

where

$$g_i = \frac{\Lambda^2}{\pi^2}\left[k_{F,i} - \Lambda\tan^{-1}\left(k_{F,i}/\Lambda\right)\right]$$

Otherwise, if form == bgbd_form, bpalb_form or sl_form, then the potential energy density is given by Bombaci01 :

$$V(\rho,\delta) = V_A + V_B + V_C$$

$$V_A = \frac{2A}{3\rho_0}\left[\left(1+\frac{x_0}{2}\right)\rho^2 - \left(\frac{1}{2}+x_0\right)\left(\rho_n^2+\rho_p^2\right)\right]$$

$$V_B = \frac{4B}{3\rho_0^\sigma}\frac{T}{1+4B'T/\left(3\rho_0^{\sigma-1}\rho^2\right)}$$

where

$$T = \rho^{\sigma-1}\left[\left(1+\frac{x_3}{2}\right)\rho^2 - \left(\frac{1}{2}+x_3\right)\left(\rho_n^2+\rho_p^2\right)\right]$$

The term V_C is:

$$V_C = \sum_{i=1}^{i_{\max}}\frac{4}{5}\left(C_i+2z_i\right)\rho(g_{n,i}+g_{p,i}) + \frac{2}{5}\left(C_i-8z_i\right)\left(\rho_n g_{n,i}+\rho_p g_{p,i}\right)$$

where

$$g_{\tau,i} = \frac{2}{(2\pi)^3}\int d^3k f_\tau(k,T) g_i(k)$$

For form == bgbd_form or form == bpalb_form, the form factor is given by

$$g_i(k) = \left(1+\frac{k^2}{\Lambda_i^2}\right)^{-1}$$

while for form == sl_form, the form factor is given by

$$g_i(k) = 1 - \frac{k^2}{\Lambda_i^2}$$

where Λ_1 is specified in the parameter `Lambda` when necessary.

See Mathematica notebook at

```
doc/o2scl/extras/gen_potential_eos.nb
doc/o2scl/extras/gen_potential_eos.ps
```

Bug

The BGBD eos doesn't work and the effective mass for the GBD eos doesn't work

Todo

Calculate the chemical potentials analytically

Definition at line 172 of file gen_potential_eos.h.

The mode for the energy() function [protected]

- int **mode**
- static const int **nmode** = 1
- static const int **pmode** = 2
- static const int **normal** = 0

Public Member Functions

- virtual int calc_e (**fermion** &ne, **fermion** &pr, **thermo** &lt)
  Equation of state as a function of density.
- int set_mu_deriv (**deriv**< void ∗, **funct**< void ∗ > > &de)
  Set the derivative object to calculate the chemical potentials.
- virtual const char ∗ type ()
  Return string denoting type ("gen_potential_eos").

Data Fields

- int form
  Form of potential.
- **gsl_deriv**< void ∗, **funct**< void ∗ > > def_mu_deriv
  The default derivative object for calculating chemical potentials.
- **nonrel_fermion** def_nr_neutron
  Default nonrelativistic neutron.
- **nonrel_fermion** def_nr_proton
  Default nonrelativistic proton.

The parameters for the various interactions

- double **x**
- double **Au**
- double **Al**
- double **rho0**
- double **B**
- double **sigma**
- double **Cl**
- double **Cu**
- double **Lambda**
- double **A**
- double **x0**
- double **x3**
- double **Bp**
- double **C1**
- double **z1**
- double **Lambda2**
- double **C2**
- double **z2**
- double **bpal_esym**
- int **sym_index**

Static Public Attributes

- static const int mdi_form = 1
  The "momentum-dependent-interaction" form.
- static const int bgbd_form = 2
  The modifed GBD form.
- static const int bpalb_form = 3
  The form from Prakash88 as formulated in Bombaci01.
- static const int sl_form = 4
  The "SL" form. See Bombaci01.
- static const int gbd_form = 5
  The Gale, Bertsch, Das Gupta from Gale87.
- static const int bpal_form = 6
  The form from Prakash88.

Protected Member Functions

- double mom_integral (double pft, double pftp)
  Compute the momentum integral for mdi_form.
- double energy (double x)
  Compute the energy.

Protected Attributes

- bool mu_deriv_set
  True of the derivative object has been set.
- **deriv**< void *, **funct**< void * > > * mu_deriv_ptr
  The derivative object.

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

- gen_potential_eos.h

5.14 had_gibbs_nseos Class Reference

```
#include <had_gibbs_nseos.h>
```

5.14.1 Detailed Description

Create a Gibbs phase transition between two hadronic EOSs to create neutron star matter.

This computes neutron star matter assuming a Gibbs phase transition between two hadronic_eos 's. When possible it generally uses hadronic_eos::calc_e() over hadronic_eos::calc_p().

This class is not very sophisticated and the functions calc_e() and fixed_chi() will fail if the initial guess is not good enough.

Definition at line 46 of file had_gibbs_nseos.h.

Public Member Functions

- virtual int calc_e (double nb, **fermion** &n1, **fermion** &p1, **thermo** &th1, **fermion** &n2, **fermion** &p2, **thermo** &th2, double &chi, int type)
  Compute the properties of neutron star matter at.
- virtual int fixed_chi (double chi, **fermion** &n1, **fermion** &p1, **thermo** &th1, **fermion** &n2, **fermion** &p2, **thermo** &th2, double &nb, int type)

Compute the properties of neutron star matter in the.

- int set_hadronic_eos1 (hadronic_eos &he)
  Set the Phase 1 hadronic equation of state.
- int set_hadronic_eos2 (hadronic_eos &he)
  Set the Phase 2 hadronic equation of state.
- int set_solver (**mroot**< void ∗, **mm_funct**< void ∗ > > &mr)
  Change the solver.

Data Fields

- **gsl_mroot_hybrids**< void ∗, **mm_funct**< void ∗ > > def_solver
  The default solver.
- int phase_guess
  The guess for the phase of the next call to calc_e() (default: `phase1`*).*

Static Public Attributes

Possible values for phase_guess

- static const int **phase1** = 1
- static const int **phase2** = 2
- static const int **mixed_phase** = 3

Possible values for type

- static const int **nuclear_matter** = 1
- static const int **neutron_matter** = 2
- static const int **nstar_matter** = 3

Protected Types

- typedef struct had_gibbs_nseos::solvepar_s **solvepar**
- typedef struct had_gibbs_nseos::chipar_s **chipar**

Protected Member Functions

- int phase1_nstar (size_t nv, const **ovector_view** &x, **ovector_view** &y, void ∗&pa)
  Solve for Phase 1 matter.
- int phase2_nstar (size_t nv, const **ovector_view** &x, **ovector_view** &y, void ∗&pa)
  Solve for Phase 2 matter.
- int gibbs_nstar (size_t nv, const **ovector_view** &x, **ovector_view** &y, void ∗&pa)
  Solve for mixed phase matter.
- int fixed_chi_nstar (size_t nv, const **ovector_view** &x, **ovector_view** &y, void ∗&pa)
  Solve for mixed phase matter at fixed chi.
- int gibbs_nuc (size_t nv, const **ovector_view** &x, **ovector_view** &y, void ∗&pa)
  Solve for mixed phase matter.
- int fixed_chi_nuc (size_t nv, const **ovector_view** &x, **ovector_view** &y, void ∗&pa)
  Solve for mixed phase matter at fixed chi.
- int gibbs_neut (size_t nv, const **ovector_view** &x, **ovector_view** &y, void ∗&pa)
  Solve for mixed phase matter.
- int fixed_chi_neut (size_t nv, const **ovector_view** &x, **ovector_view** &y, void ∗&pa)
  Solve for mixed phase matter at fixed chi.

Protected Attributes

- hadronic_eos * hep1
  The hadronic EOS for Phase 1.
- hadronic_eos * hep2
  The hadronic EOS for Phase 2.
- **fermion** elec
  The electron.
- **fermion** mu
  The muon.
- **mroot**< void *, **mm_funct**< void * > > * solverp
  The solver.

Storage for the solution vectors

- **ovector gx**
- **ovector px**

Data Structures

- struct chipar_s
  Useful structure for parameters to had_gibbs_nseos::fixed_chi_eqs() [protected].
- struct solvepar_s
  Useful structure for parameters to phase1_eqs(), phase2_eqs(), and gibbs_eqs() [protected].

5.14.2 Member Function Documentation

5.14.2.1 virtual int calc_e (double *nb*, fermion & *n1*, fermion & *p1*, thermo & *th1*, fermion & *n2*, fermion & *p2*, thermo & *th2*, double & *chi*, int *type*) [virtual]

Compute the properties of neutron star matter at.

fixed density

5.14.2.2 virtual int fixed_chi (double *chi*, fermion & *n1*, fermion & *p1*, thermo & *th1*, fermion & *n2*, fermion & *p2*, thermo & *th2*, double & *nb*, int *type*) [virtual]

Compute the properties of neutron star matter in the.

mixed phase for fixed chi

5.14.2.3 int set_solver (mroot< void *, mm_funct< void * > > & *mr*) [inline]

Change the solver.

In order to work properly, the solver needs to gracefully recover from a function which returns a non-zero value. If a **gsl_mroot_hybrids** object is used, then **gsl_mroot_hybrids::shrink_step** should be set to `true`. This is done automatically for the default solver.

Definition at line 94 of file had_gibbs_nseos.h.

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

- had_gibbs_nseos.h

5.15 had_gibbs_nseos::chipar_s Struct Reference

`#include <had_gibbs_nseos.h>`

5.15.1 Detailed Description

Useful structure for parameters to had_gibbs_nseos::fixed_chi_eqs() [protected].

Definition at line 140 of file had_gibbs_nseos.h.

Data Fields

- **fermion** ∗ **n1**
- **fermion** ∗ **p1**
- **fermion** ∗ **n2**
- **fermion** ∗ **p2**
- **thermo** ∗ **th1**
- **thermo** ∗ **th2**
- double **chi**

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

- had_gibbs_nseos.h

5.16 had_gibbs_nseos::solvepar_s Struct Reference

`#include <had_gibbs_nseos.h>`

5.16.1 Detailed Description

Useful structure for parameters to phase1_eqs(), phase2_eqs(), and gibbs_eqs() [protected].

Definition at line 127 of file had_gibbs_nseos.h.

Data Fields

- **fermion** ∗ **n1**
- **fermion** ∗ **p1**
- **fermion** ∗ **n2**
- **fermion** ∗ **p2**
- **thermo** ∗ **th1**
- **thermo** ∗ **th2**
- double **nb**

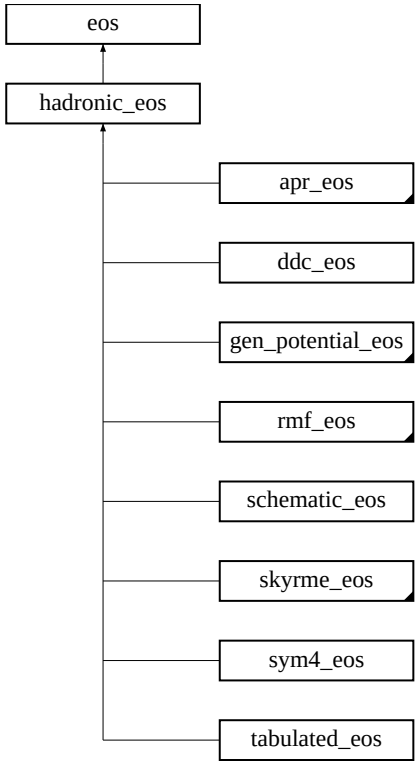
The documentation for this struct was generated from the following file:

- had_gibbs_nseos.h

5.17 hadronic_eos Class Reference

`#include <hadronic_eos.h>`

Inheritance diagram for hadronic_eos::



5.17.1 Detailed Description

Hadronic equation of state.

In the method documentation below, n is baryon number density, ϵ is energy density, and P is pressure.

See more about Svprime in the Mathematica notebook at

```
doc/o2scl/extras/hadronic_eos.nb
doc/o2scl/extras/hadronic_eos.ps
```

Todo

Need to consider using denpar versus n_baryon and proton_frac. Maybe some of this can be simplified or improved?

Definition at line 67 of file hadronic_eos.h.

Switch for overloading [protected]

The integer `'provides'` needs to be set by children corresponding to which calc() functions are overloaded. Use the above integers in combination with the bitwise or '|' operator. For example, in the constructor of a descendant, you might set

```
provides=(provides_calc_e | provides_calc_temp_e);
```

for a class which does both zero and finite temperature calculations as a function of density.

- int **provides**
- static const int **provides_calc_e** = 1
- static const int **provides_calc_p** = 2
- static const int **provides_calc_temp_e** = 4
- static const int **provides_calc_temp_p** = 8

Public Member Functions

- int gradient_qij (**fermion** &n, **fermion** &p, **thermo** &th, double &qnn, double &qnp, double &qpp, double &dqnndnn, double &dqnndnp, double &dqnpdnn, double &dqnpdnp, double &dqppdnn, double &dqppdnp)
  Calculate coefficients for **gradient part** *of Hamiltonian.*
- virtual const char * type ()
  Return string denoting type ("hadronic_eos").

Equation of state

- virtual int calc_p (**fermion** &n, **fermion** &p, **thermo** &th)
  Equation of state as a function of the chemical potentials.
- virtual int calc_temp_p (**fermion** &n, **fermion** &p, const double T, **thermo** &th)
  Equation of state as a function of the chemical potentials at finite temperature.
- virtual int calc_e (**fermion** &n, **fermion** &p, **thermo** &th)
  Equation of state as a function of density.
- virtual int calc_temp_e (**fermion** &n, **fermion** &p, const double T, **thermo** &th)
  Equation of state as a function of densities at finite temperature.

Physical properties

- virtual double fcomp (const double nb)
  Calculate compressibility of nuclear matter using calc_e().
- virtual double feoa (const double nb, const double pf=0.5)
  Calculate binding energy using calc_e().
- virtual double fesym (const double nb, const double pf=0.5)
  Calculate symmetry energy of matter using calc_e().
- virtual double fesym_slope (const double nb, bool alt_sym=false)
  The symmetry energy slope parameter.
- virtual double fesym_diff (const double nb)
  Calculate symmetry energy of matter as energy of neutron matter minus the energy of nuclear matter.
- virtual double fsprime (const double nb, const double pf=0.5)
  Calculate S' in matter using calc_e().
- virtual double fkprime (const double nb)
  Calculate skewness of nuclear matter using calc_e().
- virtual double fmsom (const double nb, const double pf=0.5)
  Calculate reduced neutron effective mass using calc_e().
- virtual double fn0 (const double protfrac, double &leoa)
  Calculate saturation density using calc_e().
- virtual int saturation ()
  Calculates all of the properties at the saturation density.

Functions for calculating physical properties

- double calc_pressure (double nb, void *&pa)
  Calculate pressure as a function of baryon density.
- double calc_press_on2 (double nb, void *&pa)
  Calculate pressure / baryon density squared as a function of baryon density.
- double calc_edensity (double delta, void *&pa)
  Calculate energy density as a function of 'delta'.
- double calc_esym (double delta, void *&pa)
  Calculate symmetry energy as a function of 'delta'.
- int saturation_matter_e (double x, double &y, void *&pa)
  Solve for zero pressure as a function of baryon density.

Other functions

- int nuc_matter_p (size_t nv, const **ovector_view** &x, **ovector_view** &y, void *&pa)
  Nucleonic matter from calc_p().
- int nuc_matter_e (size_t nv, const **ovector_view** &x, **ovector_view** &y, void *&pa)
  Nucleonic matter from calc_e().

Set auxilliary objects

- virtual int set_mroot (**mroot**< void ∗, **mm_funct**< void ∗ > > &mr)
  Set class **mroot** *object for use in calculating chemical potentials from densities.*
- virtual int set_sat_root (**root**< void ∗, **funct**< void ∗ > > &mr)
  Set class **mroot** *object for use calculating saturation density.*
- virtual int set_sat_deriv (**deriv**< void ∗, **funct**< void ∗ > > &de)
  Set **deriv** *object to use to find saturation properties.*
- virtual int set_sat_deriv2 (**deriv**< bool, **funct**< bool > > &de)
  Set **deriv** *object to use to find saturation properties.*
- virtual int set_n_and_p (**fermion** &n, **fermion** &p)
  Set neutron and proton.

Data Fields

- double eoa
  Binding energy.
- double comp
  Compressibility.
- double esym
  Symmetry energy.
- double n0
  Saturation density.
- double msom
  Effective mass.
- double kprime
  Skewness.
- double sprime
  Symmetry energy derivative.
- **gsl_deriv**< void ∗, **funct**< void ∗ > > def_deriv
  The default object for derivatives.
- **gsl_deriv**< bool, **funct**< bool > > def_deriv2
  The second default object for derivatives.
- **fermion** def_neutron
  The defaut neutron.
- **fermion** def_proton
  The defaut proton.
- **gsl_mroot_hybrids**< void ∗, **mm_funct**< void ∗ > > def_mroot
  The default solver.
- **cern_mroot_root**< void ∗, **funct**< void ∗ > > def_sat_root
  The default solver for calculating the saturation density.

Protected Types

- typedef struct hadronic_eos::mupar_s **mupar**
- typedef struct hadronic_eos::denpar_s **denpar**

Protected Member Functions

- double t1_fun (double barn, void ∗&vp)
  Compute t1 for gradient_qij().
- double t2_fun (double barn, void ∗&vp)
  Compute t2 for gradient_qij().
- double calc_esym (double nb, bool &alt)
  Return the symmetry energy at density `nb`.

Protected Attributes

- **mroot**< void ∗, **mm_funct**< void ∗ > > ∗ eos_mroot
  The EOS solver.
- **root**< void ∗, **funct**< void ∗ > > ∗ sat_root
  The solver to compute saturation properties.
- **deriv**< void ∗, **funct**< void ∗ > > ∗ sat_deriv
  The derivative object for saturation properties.
- **deriv**< bool, **funct**< bool > > ∗ sat_deriv2
  The second derivative object for saturation properties.
- **fermion** ∗ neutron
  The neutron object.
- **fermion** ∗ proton
  The proton object.
- double proton_frac
  Temporary proton fraction.
- double n_baryon
  Temporary baryon number.

Data Structures

- struct denpar_s
  Density parameters [protected].
- struct mupar_s
  Chemical potential parameters [protected].

5.17.2 Member Function Documentation

5.17.2.1 virtual double fcomp (const double *nb*) [virtual]

Calculate compressibility of nuclear matter using calc_e().

The compression modulus is defined here by: $\chi = -1/V(dV/dP) = 1/n(dP/dn)^{-1}$ It is customary to use the incompressibility modulus $K = 9/(n\chi)$. This is the value denoted `comp` in the code and can be written: $K = 9nd^2\epsilon/(dn^2) = 9dP/(dn)$. It is often referred to as the "compressibility" and is about 220 MeV at saturation density. (Taken from Chabanat, et. al. NPA 627 (1997) 710.) Note that this differs from $K_2 = 9n^2d^2(\epsilon/n)/(dn^2)$ by $18P/n$ at any density except the saturation density.

Reimplemented in apr_eos, and skyrme_eos.

5.17.2.2 virtual double feoa (const double *nb*, const double *pf* = 0.5) [virtual]

Calculate binding energy using calc_e().

`eoa` = (energy density/baryon number density-nucleon mass) at $n = n_0$. $E_b \approx -16/(\hbar c)$

5.17.2.3 virtual double fesym (const double *nb*, const double *pf* = 0.5) [virtual]

Calculate symmetry energy of matter using calc_e().

`esym` =

$$\left(\frac{1}{2n}\frac{d^2\epsilon}{d\delta^2}\right)_{n=n_B,\delta=\delta_0}$$

where $\delta = 1 - 2x$, $\delta_0 = 1 - 2x$ and x is the proton fraction (for x=0.5 at saturation density, esym $\approx 32/\hbar c$)

Reimplemented in skyrme_eos.

5.17.2.4 virtual double fesym_slope (const double *nb*, bool *alt_sym* = `false`) `[virtual]`

The symmetry energy slope parameter.

This returns the value of the "slope parameter" of the symmetry energy

$$L = 3n_B \left(\frac{\partial E_{sym}}{n_B} \right)$$

in inverse Fermis.

where n_B is the baryon density. This ranges between about zero and 200 MeV for many EOSs. If `alt_sym` is false (the default), then fesym() is used to compute the symmetry energy, otherwise fesym_diff() is used.

5.17.2.5 virtual double fesym_diff (const double *nb*) `[virtual]`

Calculate symmetry energy of matter as energy of neutron matter minus the energy of nuclear matter.

This function returns the energy per baryon of neutron matter minus the energy per baryon of nuclear matter. This will deviate significantly from the results from fesym() only if the dependence of the symmetry energy on δ is not quadratic.

Reimplemented in apr_eos.

5.17.2.6 virtual double fsprime (const double *nb*, const double *pf* = `0.5`) `[virtual]`

Calculate S' in matter using calc_e().

`sprime` =

$$\left[n \frac{d}{dn} \left(\frac{1}{2n} \frac{d^2\epsilon}{d\delta^2} \right) \right]_{n=n_B, \delta=\delta_0}$$

where $\delta = 1 - 2x$, $\delta_0 = 1 - 2(\mathrm{pf})$ and x is the proton fraction

5.17.2.7 virtual double fkprime (const double *nb*) `[virtual]`

Calculate skewness of nuclear matter using calc_e().

The skewness is defined to be $27n^3 d^3(\epsilon/n)/(dn^3) = 27n^3 d^2(P/n^2)/(dn^2)$

and is denoted 'kprime'. This definition seems to be ambiguous for densities other than the saturation density and is not quite analogous to the compressibility.

Reimplemented in skyrme_eos.

5.17.2.8 virtual double fmsom (const double *nb*, const double *pf* = `0.5`) `[virtual]`

Calculate reduced neutron effective mass using calc_e().

Neutron effective mass (n.ms) divided by vacuum mass (n.m) in nuclear matter at saturation density. Note that this simply uses the value of n.ms from calc_e(), so that this effective mass could be either the Landau or Dirac mass depending on the context. Note that this may not be equal to the reduced proton effective mass.

5.17.2.9 virtual double fn0 (const double *protfrac*, double & *leoa*) `[virtual]`

Calculate saturation density using calc_e().

This function finds the density for which the pressure vanishes in matter with $n_n = n_p$.

n_0 = baryon number density at which $P = 0, n_0 \approx 0.16$

5.17.2.10 int gradient_qij (fermion & *n*, fermion & *p*, thermo & *th*, double & *qnn*, double & *qnp*, double & *qpp*, double & *dqnndnn*, double & *dqnndnp*, double & *dqnpdnn*, double & *dqnpdnp*, double & *dqppdnn*, double & *dqppdnp*)

Calculate coefficients for **gradient part** of Hamiltonian.

Note:

This is still somewhat experimental.

We want the **gradient part** of the Hamiltonian in the form

$$\mathcal{H}_{\rm grad} = \frac{1}{2} \sum_{i=n,p} \sum_{j=n,p} Q_{ij} \vec{\nabla} n_i \cdot \vec{\nabla} n_j$$

The expression for the **gradient** terms from Pethick95 is

$$\begin{aligned} \mathcal{H}_{\rm grad} &= -\frac{1}{4}\left(2P_1 + P_{1;f} - P_{2;f}\right) \\ &\quad +\frac{1}{2}\left(Q_1 + Q_2\right)\left(n_n \nabla^2 n_n + n_p \nabla^2 n_p\right) \\ &\quad +\frac{1}{4}\left(Q_1 - Q_2\right)\left[\left(\nabla n_n\right)^2 + \left(\nabla n_p\right)^2\right] \\ &\quad +\frac{1}{2}\frac{dQ_2}{dn}\left(n_n \nabla n_n + n_p \nabla n_p\right) \cdot \nabla n \end{aligned}$$

This can be rewritten

$$\begin{aligned} \mathcal{H}_{\rm grad} &= \frac{1}{2}\left(\nabla n\right)^2 \left[\frac{3}{2}P_1 + n\frac{dP_1}{dn}\right] \\ &\quad -\frac{3}{4}\left[\left(\nabla n_n\right)^2 + \left(\nabla n_p\right)^2\right] \\ &\quad -\frac{1}{2}[] \cdot \nabla n \frac{dQ_1}{dn} \\ &\quad -\frac{1}{4}\left(\nabla n\right)^2 P_2 - \frac{1}{4}\left[\left(\nabla n_n\right)^2 + \left(\nabla n_p\right)^2\right] Q_2 \end{aligned}$$

or

$$\begin{aligned} \mathcal{H}_{\rm grad} &= \frac{1}{4}\left(\nabla n\right)^2 \left[3P_1 + 2n\frac{dP_1}{dn} - P_2\right] \\ &\quad -\frac{1}{4}\left(3Q_1 + Q_2\right)\left[\left(\nabla n_n\right)^2 + \left(\nabla n_p\right)^2\right] \\ &\quad -\frac{1}{2}\frac{dQ_1}{dn}\left[n_n \nabla n_n + n_p \nabla n_p\right] \cdot \nabla n \end{aligned}$$

or

$$\begin{aligned} \mathcal{H}_{\rm grad} &= \frac{1}{4}\left(\nabla n\right)^2 \left[3P_1 + 2n\frac{dP_1}{dn} - P_2\right] \\ &\quad -\frac{1}{4}\left(3Q_1 + Q_2\right)\left[\left(\nabla n_n\right)^2 + \left(\nabla n_p\right)^2\right] \\ &\quad -\frac{1}{2}\frac{dQ_1}{dn}\left[n_n \left(\nabla n_n\right)^2 + n_p \left(\nabla n_p\right)^2 + n \nabla n_n \cdot \nabla n_p\right] \end{aligned}$$

Generally, for Skyrme-like interactions

$$\begin{aligned} P_i &= \frac{1}{4} t_i \left(1 + \frac{1}{2} x_i\right) \\ Q_i &= \frac{1}{4} t_i \left(\frac{1}{2} + x_i\right) . \end{aligned}$$

for $i = 1, 2$.

This function uses the assumption $x_1 = x_2 = 0$ to calculate t_1 and t_2 from the neutron and proton effective masses assuming the Skyrme form. The values of Q_{ij} and their derivatives are then computed.

The functions set_n_and_p() and set_thermo() will be called by gradient_qij(), to facilitate the use of the `n`, `p`, and `th` parameters.

5.17.2.11 double calc_pressure (double *nb*, void *&* *pa*)

Calculate pressure as a function of baryon density.

Used by fcomp().

5.17.2.12 double calc_press_on2 (double *nb*, void *&* *pa*)

Calculate pressure / baryon density squared as a function of baryon density.

Used by fkprime().

5.17.2.13 double calc_edensity (double *delta*, void *&* *pa*)

Calculate energy density as a function of 'delta'.

Used by fesym().

5.17.2.14 double calc_esym (double *delta*, void *&* *pa*)

Calculate symmetry energy as a function of 'delta'.

Used by fsprime().

5.17.2.15 int saturation_matter_e (double *x*, double & *y*, void *&* *pa*)

Solve for zero pressure as a function of baryon density.

Used by fn0().

5.17.2.16 double calc_esym (double *nb*, bool & *alt*) [protected]

Return the symmetry energy at density `nb`.

Used by fesym_slope().

5.17.3 Field Documentation

5.17.3.1 gsl_deriv<void *,funct<void *> > def_deriv

The default object for derivatives.

The value of **gsl_deriv::h** is set to 10^{-3} in the hadronic_eos constructor.

Definition at line 409 of file hadronic_eos.h.

5.17.3.2 gsl_deriv<bool,funct<bool> > def_deriv2

The second default object for derivatives.

The value of **gsl_deriv::h** is set to 10^{-3} in the hadronic_eos constructor.

Definition at line 417 of file hadronic_eos.h.

5.17.3.3 gsl_mroot_hybrids<void *,mm_funct<void *> > def_mroot

The default solver.

Used by calc_e() and calc_temp_e() to solve nuc_matter_p() (2 variables) and by calc_p() and calc_temp_p() to solve nuc_matter_e() (2 variables).

Definition at line 432 of file hadronic_eos.h.

5.17.3.4 cern_mroot_root<void *,funct<void *> > def_sat_root

The default solver for calculating the saturation density.

Used by fn0() (which is called by saturation()) to solve saturation_matter_e() (1 variable).

Definition at line 440 of file hadronic_eos.h.

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

- hadronic_eos.h

5.18 hadronic_eos::denpar_s Struct Reference

`#include <hadronic_eos.h>`

5.18.1 Detailed Description

Density parameters [protected].

Definition at line 517 of file hadronic_eos.h.

Data Fields

- double **nn**
- double **np**
- double **T**

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

- hadronic_eos.h

5.19 hadronic_eos::mupar_s Struct Reference

`#include <hadronic_eos.h>`

5.19.1 Detailed Description

Chemical potential parameters [protected].

Definition at line 511 of file hadronic_eos.h.

Data Fields

- double **mun**
- double **mup**

- double **T**

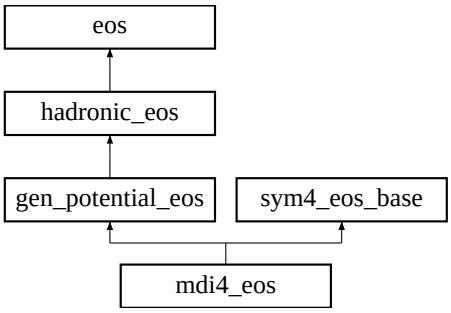
The documentation for this struct was generated from the following file:

- hadronic_eos.h

5.20 mdi4_eos Class Reference

`#include <sym4_eos.h>`

Inheritance diagram for mdi4_eos::



5.20.1 Detailed Description

A version of gen_potential_eos to separate potential and kinetic contributions.

Definition at line 125 of file sym4_eos.h.

Public Member Functions

- virtual int calc_e_sep (**fermion** &ne, **fermion** &pr, double &ed_kin, double &ed_pot, double &mu_n_kin, double &mu_p_kin, double &mu_n_pot, double &mu_p_pot)
  Compute the potential and kinetic parts separately.
- virtual int test_separation (**fermion** &ne, **fermion** &pr, **test_mgr** &t)
  Test the separation of the potential and kinetic energy parts.

Protected Member Functions

- double energy_kin (double var)
  Compute the kinetic **part** *of the energy density.*
- double energy_pot (double var)
  Compute the potential **part** *of the energy density.*

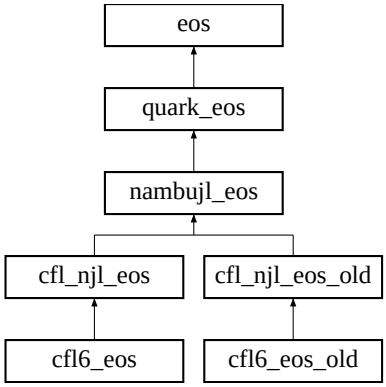
The documentation for this class was generated from the following file:

- sym4_eos.h

5.21 nambujl_eos Class Reference

`#include <nambujl_eos.h>`

Inheritance diagram for nambujl_eos::



5.21.1 Detailed Description

Nambu Jona-Lasinio EOS at zero temperature.

Calculates everything from the **quark** condensates ([uds].qq) and the chemical potentials ([uds].mu). If "fromqq" is set to false, then instead it calculates everything from the dynamical masses ([uds].ms) and the chemical potentials. L, G, K, and B0 are fixed constants. [uds].pr returns the pressure due to the Fermi-gas contribution plus the bag pressure contribution. [uds.ed] is the energy density for each **quark** so that e.g. u.ed+u.pr=u.mu∗u.n. B0 should be fixed using calc_B0() beforehand to ensure that the energy density and pressure of the vacuum is zero.

The functions set_parameters() should be called first.

The code is based on Buballa99.

The Lagrangian is

$$\mathcal{L} = \bar{q}(i\partial\!\!\!/ - \hat{m}_0)q \; + \; G\sum_{k=0}^{8}[\,(\bar{q}\lambda_k q)^2 + (\bar{q}i\gamma_5\lambda_k q)^2\,] + \mathcal{L}_6$$

$$\mathcal{L}_6 = -K\,[\,\det{}_f(\bar{q}(1+\gamma_5)q) + \det{}_f(\bar{q}(1-\gamma_5)q)\,]\,.$$

And the corresponding thermodynamic potential is

$$\Omega = \Omega_{FG} + \Omega_{Int}$$

where Ω_{FG} is the Fermi gas contribution and

$$\frac{\Omega_{\rm Int}}{V} = -2N_c \sum_{i=u,d,s} \int \frac{d^3p}{(2\pi)^3}\sqrt{m_i^2+p^2} + \frac{\Omega_V}{V}$$

$$\frac{\Omega_V}{V} = \sum_{i=u,d,s} 2G\langle\bar{q}_i q_i\rangle^2 - 4K\langle\bar{q}_u q_u\rangle\langle\bar{q}_d q_d\rangle\langle\bar{q}_s q_s\rangle + B_0\,.$$

where B_0 is a constant defined to ensure that the energy density and the pressure of the vacuum is zero.

Unlike Buballa99, the bag constant, Ω_{Int}/V is defined without the term

$$\sum_{i=u,d,s} 2N_C \int_0^\Lambda \frac{d^3p}{(2\pi)^3}\sqrt{m_{0,i}^2+p^2}\;dp$$

since this allows an easier comparison to the finite temperature EOS. The constant B_0 in this case is therefore significantly larger, but the energy density and pressure are still zero in the vacuum.

The Feynman-Hellman theorem (Bernard88), gives

$$\langle\bar{q}q\rangle = \frac{\partial m^*}{\partial m}$$

The functions calc_e() and calc_p() never return a value other than zero, but will give nonsensical results for nonsensical inputs.

Finite T documentation

Calculates everything from the **quark** condensates ([uds].qq) and the chemical potentials ([uds].mu). If "fromqq" is set to false, then instead it calculates everything from the dynamical masses ([uds].ms) and the chemical potentials. L, G, K, and B0 are fixed constants. [uds].pr returns the pressure due to the Fermi-gas contribution plus the bag pressure contribution. [uds.ed] is the energy density for each **quark** so that e.g. u.ed+u.pr=u.mu∗u.n. B0 is fixed to ensure that the energy density and pressure of the vacuum is zero.

This implementation includes contributions from antiquarks.

References: Buballa99, Hatsuda94.

Definition at line 125 of file nambujl_eos.h.

Public Member Functions

- virtual int set_parameters (double lambda=0.0, double fourferm=0.0, double sixferm=0.0)
  Set the parameters and the bag constant `B0`.
- virtual int calc_p (**quark** &u, **quark** &d, **quark** &s, **thermo** &lth)
  Equation of state as a function of chemical potentials.
- virtual int calc_temp_p (**quark** &u, **quark** &d, **quark** &s, const double T, **thermo** &th)
  Equation of state as a function of chemical potentials at finite temperature.
- virtual int calc_eq_p (**quark** &u, **quark** &d, **quark** &s, double &gap1, double &gap2, double &gap3, **thermo** &lth)
  Equation of state and gap equations as a function of chemical potential.
- virtual int calc_eq_e (**quark** &u, **quark** &d, **quark** &s, double &gap1, double &gap2, double &gap3, **thermo** &lth)
  Equation of state and gap equations as a function of the densities.
- int calc_eq_temp_p (**quark** &tu, **quark** &td, **quark** &ts, double &gap1, double &gap2, double &gap3, **thermo** &qb, const double temper)
  Equation of state and gap equations as a function of chemical potentials.
- int gapfunms (size_t nv, const **ovector_view** &x, **ovector_view** &y, void ∗&pa)
  Calculates gap equations in `y` *as a function of the constituent masses in* `x`.
- int gapfunqq (size_t nv, const **ovector_view** &x, **ovector_view** &y, void ∗&pa)
  Calculates gap equations in `y` *as a function of the* **quark** *condensates in* `x`.
- int gapfunmsT (size_t nv, const **ovector_view** &x, **ovector_view** &y, void ∗&pa)
  Calculates gap equations in `y` *as a function of the constituent masses in* `x`.
- int gapfunqqT (size_t nv, const **ovector_view** &x, **ovector_view** &y, void ∗&pa)
  Calculates gap equations in `y` *as a function of the* **quark** *condensates in* `x`.
- int set_quarks (**quark** &u, **quark** &d, **quark** &s)
  Set the **quark** *objects to use.*
- virtual const char ∗ type ()
  Return string denoting type ("nambujl_eos").
- virtual int set_solver (**mroot**< void ∗, **mm_funct**< void ∗ > > &s)
  Set solver to use in set_parameters().
- virtual int set_inte (**inte**< void ∗, **funct**< void ∗ > > &i)
  Set integration object.

Data Fields

- double limit
  Accuracy limit for Fermi integrals for finite temperature.
- bool fromqq
  Calculate from **quark** *condensates if true (default true).*
- double L
  The momentum cutoff.
- double G
  The four-fermion coupling.
- double K
  The 't Hooft six-fermion interaction coupling.

- double B0
  The bag constant.
- **gsl_mroot_hybrids**< void ∗, **mm_funct**< void ∗ > > def_solver
  The default solver.

The default quark masses

These are the values from Buballa99 which were used to fix the pion and kaon decay constants, and the pion, kaon, and eta prime masses. They are set in the constructor and are in units of fm^{-1} .

- double **up_default_mass**
- double **down_default_mass**
- double **strange_default_mass**

The default quark objects

The masses are automatically set in the constructor to `up_default_mass`, `down_default_mass`, *and* `strange_default_mass.c`

- **quark def_up**
- **quark def_down**
- **quark def_strange**

Protected Types

- typedef struct nambujl_eos::njtp_s **njtp**

Protected Member Functions

- int B0fun (size_t nv, const **ovector_view** &x, **ovector_view** &y, void ∗&pa)
  Used by calc_B0() to compute the bag constant.
- void njbag (**quark** &q)
  Calculates the contribution to the bag constant from **quark** q.
- double iqq (double x, void ∗&pa)
  The integrand for the **quark** *condensate.*
- double ide (double x, void ∗&pa)
  The integrand for the density.
- double ied (double x, void ∗&pa)
  The integrand for the energy density.
- double ipr (double x, void ∗&pa)
  The integrand for the pressure.

Protected Attributes

- **gsl_inte_qag**< void ∗, **funct**< void ∗ > > def_it
  The default integrator.
- **inte**< void ∗, **funct**< void ∗ > > ∗ it
  The integrator for finite temperature integrals.
- **mroot**< void ∗, **mm_funct**< void ∗ > > ∗ solver
  The solver to use for set_parameters().
- **quark** ∗ up
  The up **quark**.
- **quark** ∗ down
  The down **quark**.
- **quark** ∗ strange
  The strange **quark**.
- double cp_temp
  The temperature for calc_temp_p().

Data Structures

- struct njtp_s
  A structure for passing parameters to the integrands [protected].

5.21.2 Member Function Documentation

5.21.2.1 virtual int set_parameters (double *lambda* = 0.0, double *fourferm* = 0.0, double *sixferm* = 0.0) [virtual]

Set the parameters and the bag constant B0.

This function allows the user to specify the momentum cutoff, lambda, the four-fermion coupling fourferm and the six-fermion coupling from the 't Hooft interaction sixferm . If 0.0 is given for any of the values, then the default is used ($\Lambda = 602.3/(\hbar c)$, $G = 1.835/\Lambda^2$, $K = 12.36/\Lambda^5$).

The value of the shift in the bag constant B0 is automatically calculated to ensure that the energy density and the pressure of the vacuum are zero. The functions set_quarks() and set_thermo() can be used before hand to specify the **quark** and **thermo** objects.

5.21.2.2 virtual int calc_p (quark & *u*, quark & *d*, quark & *s*, thermo & *lth*) [virtual]

Equation of state as a function of chemical potentials.

This function automatically solves the gap equations

Reimplemented from quark_eos.

5.21.2.3 virtual int calc_temp_p (quark & *u*, quark & *d*, quark & *s*, const double *T*, thermo & *th*) [virtual]

Equation of state as a function of chemical potentials at finite temperature.

This function automatically solves the gap equations

Reimplemented from quark_eos.

Reimplemented in cfl_njl_eos_old.

5.21.2.4 int gapfunms (size_t *nv*, const ovector_view & *x*, ovector_view & *y*, void *& *pa*)

Calculates gap equations in y as a function of the constituent masses in x.

The function utilizes the **quark** objects which can be specified in set_quarks() and the **thermo** object which can be specified in eos::set_thermo().

5.21.2.5 int gapfunqq (size_t *nv*, const ovector_view & *x*, ovector_view & *y*, void *& *pa*)

Calculates gap equations in y as a function of the **quark** condensates in x.

The function utilizes the **quark** objects which can be specified in set_quarks() and the **thermo** object which can be specified in eos::set_thermo().

5.21.2.6 int gapfunmsT (size_t *nv*, const ovector_view & *x*, ovector_view & *y*, void *& *pa*)

Calculates gap equations in y as a function of the constituent masses in x.

The function utilizes the **quark** objects which can be specified in set_quarks() and the **thermo** object which can be specified in eos::set_thermo().

5.21.2.7 int gapfunqqT (size_t *nv*, const ovector_view & *x*, ovector_view & *y*, void *& *pa*)

Calculates gap equations in `y` as a function of the **quark** condensates in `x`.

The function utilizes the **quark** objects which can be specified in set_quarks() and the **thermo** object which can be specified in eos::set_thermo().

5.21.2.8 int set_quarks (quark & *u*, quark & *d*, quark & *s*)

Set the **quark** objects to use.

The **quark** objects are used in gapfunms(), gapfunqq(), gapfunmsT(), gapfunqqT(), and B0fun().

5.21.3 Field Documentation

5.21.3.1 double limit

Accuracy limit for Fermi integrals for finite temperature.

limit is used for the finite temperature integrals to ensure that no numbers larger than exp(limit) or smaller than exp(-limit) are avoided. (Default: 20)

Definition at line 153 of file nambujl_eos.h.

5.21.3.2 bool fromqq

Calculate from **quark** condensates if true (default true).

If this is false, then computations are performed using the effective masses as inputs

Definition at line 161 of file nambujl_eos.h.

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

- nambujl_eos.h

5.22 nambujl_eos::njtp_s Struct Reference

`#include <nambujl_eos.h>`

5.22.1 Detailed Description

A structure for passing parameters to the integrands [protected].

Definition at line 340 of file nambujl_eos.h.

Data Fields

- double **ms**
- double **m**
- double **mu**
- double **temper**
- double **limit**

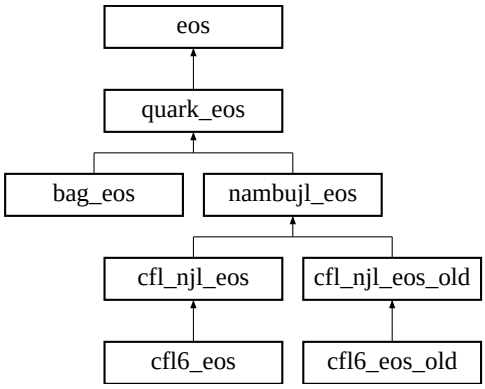
The documentation for this struct was generated from the following file:

- nambujl_eos.h

5.23 quark_eos Class Reference

`#include <quark_eos.h>`

Inheritance diagram for quark_eos::



5.23.1 Detailed Description

Quark matter equation of state base.

Definition at line 39 of file quark_eos.h.

Public Member Functions

- virtual int calc_p (**quark** &u, **quark** &d, **quark** &s, **thermo** &th)
  Calculate equation of state as a function of chemical potentials.
- virtual int calc_e (**quark** &u, **quark** &d, **quark** &s, **thermo** &th)
  Calculate equation of state as a function of density.
- virtual int calc_temp_p (**quark** &u, **quark** &d, **quark** &s, const double temper, **thermo** &th)
  Calculate equation of state as a function of chemical potentials.
- virtual int calc_temp_e (**quark** &u, **quark** &d, **quark** &s, const double temper, **thermo** &th)
  Calculate equation of state as a function of density.
- virtual const char * type ()
  Return string denoting type ("quark_eos").

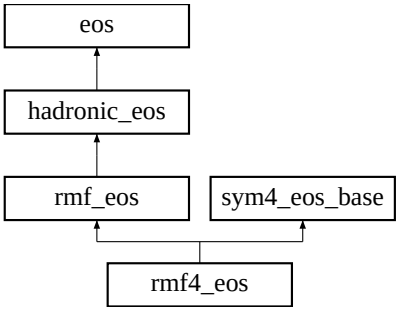
The documentation for this class was generated from the following file:

- quark_eos.h

5.24 rmf4_eos Class Reference

`#include <sym4_eos.h>`

Inheritance diagram for rmf4_eos::



5.24.1 Detailed Description

A version of rmf_eos to separate potential and kinetic contributions.

Definition at line 80 of file sym4_eos.h.

Public Member Functions

- virtual int calc_e_sep (**fermion** &ne, **fermion** &pr, double &ed_kin, double &ed_pot, double &mu_n_kin, double &mu_p_kin, double &mu_n_pot, double &mu_p_pot)
  Compute the potential and kinetic parts separately.

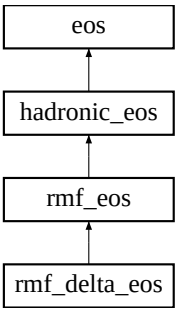
The documentation for this class was generated from the following file:

- sym4_eos.h

5.25 rmf_delta_eos Class Reference

`#include <rmf_delta_eos.h>`

Inheritance diagram for rmf_delta_eos::



5.25.1 Detailed Description

Field-theoretical EOS with scalar-isovector meson,.

δ.

This essentially follows the notation in PLB 399 (1997) 191, except that our definitions of $\mathtt{b}$ and $\mathtt{c}$ follow their $\bar{b}$ and $\bar{c}$, respectively.

Also discussed in NPA 732 (2004) 24, where they take $m_\delta = 980$ MeV.

The full Lagragian is:

$$\mathcal{L} = \mathcal{L}_{Dirac} + \mathcal{L}_{\sigma} + \mathcal{L}_{\omega} + \mathcal{L}_{\rho} + \mathcal{L}_{\delta}$$

$$\begin{aligned}
\mathcal{L}_{Dirac} &= \bar{\Psi}\left[i\partial\!\!\!/ - g_{\omega}\omega\!\!\!/ - \frac{g_{\rho}}{2}\vec{\rho}\!\!\!/\,\vec{\tau} - M + g_{\sigma}\sigma - \frac{e}{2}\left(1+\tau_3\right)A_{\mu}\right]\Psi \\
\mathcal{L}_{\sigma} &= \tfrac{1}{2}\left(\partial_{\mu}\sigma\right)^2 - \tfrac{1}{2}m_{\sigma}^2\sigma^2 - \frac{bM}{3}\left(g_{\sigma}\sigma\right)^3 - \frac{c}{4}\left(g_{\sigma}\sigma\right)^4 \\
\mathcal{L}_{\omega} &= -\tfrac{1}{4}f_{\mu\nu}f^{\mu\nu} + \tfrac{1}{2}m_{\omega}^2\omega^{\mu}\omega_{\mu} + \frac{\zeta}{24}g_{\omega}^4\left(\omega^{\mu}\omega_{\mu}\right)^2 \\
\mathcal{L}_{\rho} &= -\tfrac{1}{4}\vec{B}_{\mu\nu}\cdot\vec{B}^{\mu\nu} + \tfrac{1}{2}m_{\rho}^2\vec{\rho}^{\,\mu}\cdot\vec{\rho}_{\,\mu} + \frac{\xi}{24}g_{\rho}^4\left(\vec{\rho}^{\,\mu}\right)\cdot\vec{\rho}_{\,\mu} + g_{\rho}^2 f(\sigma,\omega)\vec{\rho}^{\,\mu}\cdot\vec{\rho}_{\,\mu}
\end{aligned}$$

where the additional terms are

$$\mathcal{L}_{\delta} = \bar{\Psi}\left(g_{\delta}\vec{\delta}\cdot\vec{\tau}\right)\Psi + \frac{1}{2}(\partial_{\mu}\vec{\delta})^2 - \frac{1}{2}m_{\delta}^2\vec{\delta}^{\,2}$$

The new field equation for the delta meson is

$$m_{\delta}^2\delta = g_{\delta}(n_{s,p} - n_{s,n})$$

Todo

Finish finite temperature

Definition at line 93 of file rmf_delta_eos.h.

Public Member Functions

- virtual int calc_e (**fermion** &ne, **fermion** &pr, **thermo** &lth)
  Equation of state as a function of density.
- virtual int calc_p (**fermion** &neu, **fermion** &p, double sig, double ome, double rho, double delta, double &f1, double &f2, double &f3, double &f4, **thermo** &th)
  Equation of state as a function of chemical potentials.
- int calc_temp_p (**fermion** &ne, **fermion** &pr, double temper, double sig, double ome, double lrho, double delta, double &f1, double &f2, double &f3, double &f4, **thermo** &lth)
  Finite temperature (unfinished).
- virtual int set_fields (double sig, double ome, double lrho, double delta)
  Set a guess for the fields for the next call to calc_e(), calc_p(), or saturation().
- virtual int saturation ()
  Calculate saturation properties for nuclear matter at the saturation density.

Data Fields

- double md
  The mass of the scalar-isovector field.
- double cd
  The coupling of the scalar-isovector field to the nucleons.
- double del
  The value of the scalar-isovector field.

Protected Member Functions

- virtual int calc_e_solve_fun (size_t nv, const **ovector_view** &ex, **ovector_view** &ey, void *&pa)
  The function for calc_e().
- virtual int zero_pressure (size_t nv, const **ovector_view** &ex, **ovector_view** &ey, void *&pa)
  Compute matter at zero pressure (for saturation()).

Private Member Functions

- virtual int set_fields (double sig, double ome, double lrho)
  Forbid setting the guesses to the fields unless all four fields are specified.

5.25.2 Member Function Documentation

5.25.2.1 virtual int saturation () [virtual]

Calculate saturation properties for nuclear matter at the saturation density.

This requires initial guesses to the chemical potentials, etc.

Reimplemented from rmf_eos.

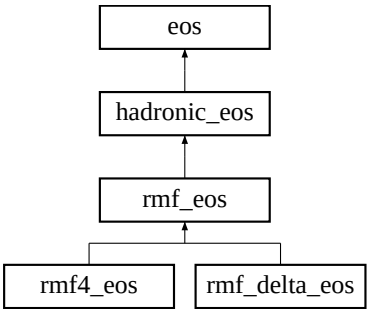
The documentation for this class was generated from the following file:

- rmf_delta_eos.h

5.26 rmf_eos Class Reference

`#include <rmf_eos.h>`

Inheritance diagram for rmf_eos::



5.26.1 Detailed Description

Relativistic mean field theory EOS.

Before sending neutrons and protons to these member functions, the masses should be set to their vacuum values and the degeneracy factor should be 2. If an internal model is used (using load()), then the neutron and proton masses should be set to mnuc.

It is important to point out that expressions for the energy densities are often simplified in the literature using the field equations. These expressions are not used in this code since they are only applicable in infinite matter where the field equations hold, and are not suitable for use in applications (such as to finite nuclei) where the spatial derivatives of the fields are non-zero. Notice that in the proper expressions for the energy density the similarity between terms in the pressure up to a sign. This procedure allows one to verify the thermodynamic identity even if the field equations are not solved and allows the user to add **gradient** terms to the energy density and pressure.

Note:

Since this EOS uses the effective masses and chemical potentials in the **fermion** class, the values of **part::non_interacting** for neutrons and protons are set to false in many of the functions.

Todo

- The number of couplings is getting unmanagable, maybe new organization is required.
- Check the formulas in the "Background" section
- Overload hadronic_eos::fcomp() with an exact version
- Fix calc_p() to be better at guessing
- There are two calc_e() functions that solve. One is specially designed to work without a good initial guess. Possible the other calc_e() function should be similarly designed?
- Make sure that this class properly handles particles for which inc_rest_mass is true/false
- It might be nice to remove explicit reference to the meson masses in functions which only compute nuclear matter since they are unnecessary. This might, however, demand redefining some of the couplings.

Background

The full Lagragian is:

$$\mathcal{L} = \mathcal{L}_{Dirac} + \mathcal{L}_{\sigma} + \mathcal{L}_{\omega} + \mathcal{L}_{\rho}$$

$$\begin{aligned}
\mathcal{L}_{Dirac} &= \bar{\Psi}\left[i\partial\!\!\!/ - g_{\omega}\omega\!\!\!/ - \frac{g_{\rho}}{2}\rho\!\!\!/\vec{\tau} - M + g_{\sigma}\sigma - \frac{e}{2}\left(1+\tau_3\right)A_{\mu}\right]\Psi \\
\mathcal{L}_{\sigma} &= \tfrac{1}{2}\left(\partial_{\mu}\sigma\right)^2 - \tfrac{1}{2}m_{\sigma}^2\sigma^2 - \frac{bM}{3}\left(g_{\sigma}\sigma\right)^3 - \frac{c}{4}\left(g_{\sigma}\sigma\right)^4 \\
\mathcal{L}_{\omega} &= -\tfrac{1}{4}f_{\mu\nu}f^{\mu\nu} + \tfrac{1}{2}m_{\omega}^2\omega^{\mu}\omega_{\mu} + \frac{\zeta}{24}g_{\omega}^4\left(\omega^{\mu}\omega_{\mu}\right)^2 \\
\mathcal{L}_{\rho} &= -\tfrac{1}{4}\vec{B}_{\mu\nu}\cdot\vec{B}^{\mu\nu} + \tfrac{1}{2}m_{\rho}^2\vec{\rho}^{\,\mu}\cdot\vec{\rho}_{\,\mu} + \frac{\xi}{24}g_{\rho}^4\left(\vec{\rho}^{\,\mu}\right)\cdot\vec{\rho}_{\,\mu} + g_{\rho}^2 f(\sigma,\omega)\vec{\rho}^{\,\mu}\cdot\vec{\rho}_{\,\mu}
\end{aligned}$$

The coefficients b and c are related to the somewhat standard κ and λ by:

$$\kappa = 2Mb \quad \lambda = 6c;$$

The function f is the coefficient of $g_{\tau}^2\rho^2$

$f(\sigma,\omega) = b_1\omega^2 + b_2\omega^4 + b_3\omega^6 + a_1\sigma + a_2\sigma^2 + a_3\sigma^3 + a_4\sigma^4 + a_5\sigma^5 + a_6\sigma^6$

where the notation from Horowitz01 is:

$f(\sigma,\omega) = \lambda_4 g_s^2\sigma^2 + \lambda_v g_w^2\omega^2$

This implies $b_1 = \lambda_v g_w^2$ and $a_2 = \lambda_4 g_s^2$

The field equations are:

$$0 = m_{\sigma}^2\sigma - g_{\sigma}\left(n_{sn} + n_{sp}\right) + bMg_{\sigma}^3\sigma^2 + cg_{\sigma}^4\sigma^3 - g_{\rho}^2\rho^2\frac{\partial f}{\partial\sigma}$$

$$0 = m_{\omega}^2\omega - g_{\omega}\left(n_n + n_p\right) + \frac{\zeta}{6}g_{\omega}^4\omega^3 g_{\rho}^2\rho^2\frac{\partial f}{\partial\omega}$$

$$0 = m_\rho^2 \rho + \frac{1}{2} g_\rho \left(n_n - n_p \right) + 2 g_\rho^2 \rho f + \frac{\xi}{6} g_\rho^4 \rho^3$$

When the variable zm_mode is true, the effective mass is fixed using the approach of Zimanyi90 .

Defining

$$U(\sigma) = \frac{1}{2} m_\sigma^2 \sigma^2 + \frac{bM}{3} (g_\sigma \sigma)^3 + \frac{c}{4} (g_\sigma \sigma)^4 \, ,$$

the binding energy per particle in symmetric matter at equilibrium is given by

$$\frac{E}{A} = \frac{1}{n_0} \left[U(\sigma_0) + \frac{1}{2} m_\omega \omega_0^2 + \frac{\zeta}{8} (g_\omega \omega_0)^4 + \frac{2}{\pi^2} \int_0^{k_F} dk k^2 \sqrt{k^2 + M^{*2}} \right] \, .$$

where the Dirac effective mass is $M^* = M - g_\sigma \sigma_0$. The compressibility is given by

$$K = 9 \frac{g_\omega^2}{m_\omega^2} n_0 + 3 \frac{k_F^2}{E_F^*} - 9 n_0 \frac{M^{*2}}{E_F^{*2}} \left[\left(\frac{1}{g_\sigma^2} \frac{\partial^2}{\partial \sigma_0^2} + \frac{3}{g_\sigma M^*} \frac{\partial}{\partial \sigma_0} \right) U(\sigma_0) - 3 \frac{n_0}{E_F^*} \right]^{-1} \, .$$

The symmetry energy of bulk matter is given by

$$E_{sym} = \frac{k_F^2}{6 E_F^*} + \frac{n}{8 \left(g_\rho^2 / m_\rho^2 + 2 f(\sigma_0, \omega_0) \right)}$$

In the above equations, the subscipt " 0 " denotes the mean field values of σ and ω . For the case $f = 0$, the symmetry energy varies linearly with the density at large densities. The function f permits variations in the density dependence of the symmetry energy above nuclear matter density.

See also Muller96, Zimanyi90.

Definition at line 212 of file rmf_eos.h.

Public Member Functions

- int load (std::string model, bool external=false)
  Load parameters for model named 'model'.
- int calc_temp_e (**fermion** &ne, **fermion** &pr, const double T, **thermo** &lth)
  Equation of state as a function of densities at finite temperature.
- int fix_saturation (double guess_cs=4.0, double guess_cw=3.0, double guess_b=0.001, double guess_c=-0.001)
  Calculate cs, cw, cr, b, and c from the saturation properties.
- virtual int saturation ()
  Calculate properties for nuclear matter at the saturation density.
- double fesym_fields (double sig, double ome, double nb)
  Calculate symmetry energy assuming the field equations have already been solved.
- double fcomp_fields (double sig, double ome, double nb)
  Calculate the compressibility assuming the field equations have already been solved.
- int fkprime_fields (double sig, double ome, double nb, double &k, double &kprime)
  Calculate compressibilty and `kprime` *assuming the field equations have already been solved.*
- int field_eqs (size_t nv, const **ovector_view** &x, **ovector_view** &y, void *&pa)
  A function for solving the field equations.
- int field_eqsT (size_t nv, const **ovector_view** &x, **ovector_view** &y, void *&pa)
  A function for solving the field equations at finite temperature.

- virtual int set_fields (double sig, double ome, double lrho)
  Set a guess for the fields for the next call to calc_e(), calc_p(), or saturation().
- int get_fields (double &sig, double &ome, double &lrho)
  Return the most recent values of the meson fields.
- virtual const char * type ()
  Return string denoting type ("rmf_eos").
- int check_naturalness (rmf_eos &re)
  Set the coefficients of a rmf_eos object to their limits from naturalness.
- int naturalness_limits (double value, rmf_eos &re)
  Provide the maximum values of the couplings assuming a limit on naturalness.
- virtual int set_sat_mroot (**mroot**< void *, **mm_funct**< void * > > &mrx)
  Set class **mroot** *object for use calculating saturation density.*

Compute EOS from densities at zero temperature

- virtual int calc_e (**fermion** &ne, **fermion** &pr, **thermo** &lth)
  Equation of state as a function of density.
- virtual int calc_e (**fermion** &ne, **fermion** &pr, **thermo** &lth, double &sig, double &ome, double &rho)
  Equation of state as a function of density.

Compute EOS from chemical at zero temperature

- virtual int calc_p (**fermion** &ne, **fermion** &pr, **thermo** &lth)
  Equation of state as a function of chemical potential.
- virtual int calc_eq_p (**fermion** &neu, **fermion** &p, double sig, double ome, double rho, double &f1, double &f2, double &f3, **thermo** &th)
  Equation of state and meson field equations as a function of chemical potentials.

Compute EOS from chemical potentials at finite temperature

- virtual int calc_eq_temp_p (**fermion** &ne, **fermion** &pr, const double temper, double sig, double ome, double rho, double &f1, double &f2, double &f3, **thermo** &th)
  Equation of state and meson field equations as a function of chemical potentials.
- virtual int calc_temp_p (**fermion** &ne, **fermion** &pr, const double T, **thermo** &lth)
  Equation of state as a function of chemical potential.

Data Fields

- bool zm_mode
  Modifies method of calculating effective masses.
- **gsl_mroot_hybrids**< void *, **mm_funct**< void * > > def_sat_mroot
  The default solver for calculating the saturation density.

Masses

- double mnuc
  nucleon mass
- double ms
  σ mass (in fm^{-1})
- double mw
  ω mass (in fm^{-1})
- double mr
  ρ mass (in fm^{-1})

Standard couplings (including nonlinear sigma terms)

- double **cs**
- double **cw**
- double **cr**

- double **b**
- double **c**

Non-linear terms for omega and rho.

- double **zeta**
- double **xi**

Additional isovector couplings

- double **a1**
- double **a2**
- double **a3**
- double **a4**
- double **a5**
- double **a6**
- double **b1**
- double **b2**
- double **b3**

Protected Member Functions

- int fix_saturation_fun (size_t nv, const **ovector_view** &x, **ovector_view** &y, void ∗&pa)
  The function for fix_saturation().
- virtual int zero_pressure (size_t nv, const **ovector_view** &ex, **ovector_view** &ey, void ∗&pa)
  Compute matter at zero pressure (for saturation()).
- virtual int calc_e_solve_fun (size_t nv, const **ovector_view** &ex, **ovector_view** &ey, void ∗&pa)
  The function for calc_e().
- virtual int calc_temp_e_solve_fun (size_t nv, const **ovector_view** &ex, **ovector_view** &ey, void ∗&pa)
  The function for calc_temp_e().
- int calc_cr (double sig, double ome, double nb)
  Calculate the `cr` *coupling given* `sig` *and* `ome` *at the density 'nb'.*

Protected Attributes

- double n_charge
  Temporary charge density.
- double fe_temp
  Temperature for solving field equations at finite temperature.
- bool ce_neut_matter
  For calc_e(), if true, then solve for neutron matter.
- bool ce_prot_matter
  For calc_e(), if true, then solve for proton matter.
- bool guess_set
  True if a guess for the fields has been given.
- **mroot**< void ∗, **mm_funct**< void ∗ > > ∗ sat_mroot
  The solver to compute saturation properties.
- double **ce_temp**

The meson fields

- double **sigma**
- double **omega**
- double **rho**

5.26.2 Member Function Documentation

5.26.2.1 int load (std::string *model*, bool *external* = `false`)

Load parameters for model named 'model'.

Presently accepted values from file rmfdata/model_list:

```
word[] models 46
BMPI BMPII FPWC FSUGold L-BF L-HS L-W L-Z L1 L2 L3 NL-06 NL-065 NL-07
NL-075 NL-B1 NL-B2 NL-SH NL-Z NL1 NL2 NL3 NL4 PL-40 PL-Z RAPR RAPRhdp
S271 SR1 SR2 SR3 TM1 TM2 Z271 es25 es25n15 es275 es275n15 es30 es30n15
es325 es325n15 es35 es35n15 es25small es25new es275new es30new
#
# Comments:
# PL-40 and PL-Z have a special m_infinity parameter that is
# described in P.-G. Reinhard, Rep. Prog. Phys., 52 (1989) 439 that
# I don't quite understand. For spher1d, these need to be run manually
# using the input files in ~/spher1d/data.
#
#
```

In these files, the nucleon and meson masses are by default specified in MeV, and cs, cw, and cr are given in fm. The parameters b and c are both unitless. If the bool 'oakstyle' is true, then load() assumes that gs, gw, and gr have been given where gs and gw are as usual, but gr is a factor of two smaller than usual, and g2 and g3 have been given where g2 = -b M gs^3 and g3 = c gs^4. If tokistyle is true, then it is additionally assumed that c3 is given where c3=zeta/6∗gw^4.

If `external` is true, then model is the filename (relative to the current directory) of the file containing the model parameters. Otherwise, the model is assumed to be present in the O2scl library data directory.

5.26.2.2 virtual int calc_e (fermion & *ne*, fermion & *pr*, thermo & *lth*) `[virtual]`

Equation of state as a function of density.

Initial guesses for the chemical potentials are taken from the user-given values. Initial guesses for the fields can be set by set_fields(), or default values will be used. After the call to calc_e(), the final values of the fields can be accessed through get_fields().

This is a little more robust than the standard version in the parent hadronic_eos.

Todo

This doesn't always work well when the proton density is zero.

Reimplemented from hadronic_eos.

Reimplemented in rmf_delta_eos.

5.26.2.3 virtual int calc_e (fermion & *ne*, fermion & *pr*, thermo & *lth*, double & *sig*, double & *ome*, double & *rho*) `[virtual]`

Equation of state as a function of density.

Todo

This doesn't always work well when the proton density is zero.

Todo

Rename this function to distinguish between calc_e()'s

5.26.2.4 virtual int calc_p (fermion & *ne*, fermion & *pr*, thermo & *lth*) [virtual]

Equation of state as a function of chemical potential.

Solves for the field equations automatically.

Note:

This may not be too robust. Fix?

Reimplemented from hadronic_eos.

5.26.2.5 virtual int calc_eq_p (fermion & *neu*, fermion & *p*, double *sig*, double *ome*, double *rho*, double & *f1*, double & *f2*, double & *f3*, thermo & *th*) [virtual]

Equation of state and meson field equations as a function of chemical potentials.

This calculates the pressure and energy density as a function of $\mu_n, \mu_p, \sigma, \omega, rho$. When the field equations have been solved, `f1`, `f2`, and `f3` are all zero.

The thermodynamic identity is satisfied even when the field equations are not solved.

5.26.2.6 virtual int calc_temp_p (fermion & *ne*, fermion & *pr*, const double *T*, thermo & *lth*) [virtual]

Equation of state as a function of chemical potential.

Solves for the field equations automatically.

Reimplemented from hadronic_eos.

5.26.2.7 int fix_saturation (double *guess_cs* = `4.0`, double *guess_cw* = `3.0`, double *guess_b* = `0.001`, double *guess_c* = `-0.001`)

Calculate cs, cw, cr, b, and c from the saturation properties.

Note that the meson masses and mnuc must be specified before calling this function.

This function does not give correct results when bool zm_mode is true.

`guess_cs`, `guess_cw`, `guess_b`, and `guess_c` are initial guesses for `cs`, `cw`, `b`, and `c` respectively.

Todo

- Fix this for zm_mode=true
- Ensure solver is more robust

5.26.2.8 virtual int saturation () [virtual]

Calculate properties for nuclear matter at the saturation density.

This requires initial guesses to the chemical potentials, etc.

Reimplemented from hadronic_eos.

Reimplemented in rmf_delta_eos.

5.26.2.9 double fesym_fields (double *sig*, double *ome*, double *nb*)

Calculate symmetry energy assuming the field equations have already been solved.

This may only work at saturation density. Used by saturation().

5.26.2.10 double fcomp_fields (double *sig*, double *ome*, double *nb*)

Calculate the compressibility assuming the field equations have already been solved.

This may only work at saturation density.

5.26.2.11 int fkprime_fields (double *sig*, double *ome*, double *nb*, double & *k*, double & *kprime*)

Calculate compressibilty and `kprime` assuming the field equations have already been solved.

This may only work at saturation density. Used by saturation().

Todo

Does this work? Fix fkprime_fields() if it does not.

5.26.2.12 int field_eqs (size_t *nv*, const ovector_view & *x*, ovector_view & *y*, void *& *pa*)

A function for solving the field equations.

x[0], x[1], and x[2] should be set to σ, ω , and ρ on input (in fm^{-1}) and on exit, y[0], y[1] and y[2] contain the field equations and are zero when the field equations have been solved. The `pa` parameter is ignored.

5.26.2.13 int field_eqsT (size_t *nv*, const ovector_view & *x*, ovector_view & *y*, void *& *pa*)

A function for solving the field equations at finite temperature.

x[0], x[1], and x[2] should be set to σ, ω , and ρ on input (in fm^{-1}) and on exit, y[0], y[1] and y[2] contain the field equations and are zero when the field equations have been solved. The `pa` parameter is ignored.

5.26.2.14 int get_fields (double & *sig*, double & *ome*, double & *lrho*) [inline]

Return the most recent values of the meson fields.

This returns the most recent values of the meson fields set by a call to saturation(), calc_e(), or calc_p(**fermion** &, **fermion** &, **thermo** &).

Definition at line 456 of file rmf_eos.h.

5.26.2.15 int check_naturalness (rmf_eos & *re*) [inline]

Set the coefficients of a rmf_eos object to their limits from naturalness.

As given in muller and Serot, npa 606, 508

The definition of the vector-isovector field and coupling matches what is done here. Compare the Lagrangian above with Eq. 10 from the reference.

The following couplings should all be of the same size:

$$\frac{1}{2c_s^2M^2}, \frac{1}{2c_v^2M^2}\frac{1}{8c_\rho^2M^2}, \text{ and } \frac{\bar{a}_{ijk}M^{i+2j+2k-4}}{2^{2k}}$$

which are equivalent to

$$\frac{m_s^2}{2g_s^2M^2}, \frac{m_v^2}{2g_v^2M^2}\frac{m_\rho^2}{8g_\rho^2M^2}, \text{ and } \frac{a_{ijk}M^{i+2j+2k-4}}{g_s^i g_v^{2j} g_\rho^{2k} 2^{2k}}$$

The connection the a_{ijk} 's and the coefficients that are used here is

$$\frac{bM}{3}g_\sigma^3\sigma^3 \quad = \quad a_{300}\,\sigma^3$$

$$
\begin{aligned}
\frac{c}{4} g_\sigma^4 \sigma^4 &= a_{400}\, \sigma^4 \\
\frac{\zeta}{24} g_\omega^4 \omega^4 &= a_{020}\, \omega^4 \\
\frac{\xi}{24} g_\rho^4 \rho^4 &= a_{002}\, \rho^4 \\
b_1 g_\rho^2 \omega^2 \rho^2 &= a_{011}\, \omega^2 \rho^2 \\
b_2 g_\rho^2 \omega^4 \rho^2 &= a_{021}\, \omega^4 \rho^2 \\
b_3 g_\rho^2 \omega^6 \rho^2 &= a_{031}\, \omega^6 \rho^2 \\
a_1 g_\rho^2 \sigma^1 \rho^2 &= a_{101}\, \sigma^1 \rho^2 \\
a_2 g_\rho^2 \sigma^2 \rho^2 &= a_{201}\, \sigma^2 \rho^2 \\
a_3 g_\rho^2 \sigma^3 \rho^2 &= a_{301}\, \sigma^3 \rho^2 \\
a_4 g_\rho^2 \sigma^4 \rho^2 &= a_{401}\, \sigma^4 \rho^2 \\
a_5 g_\rho^2 \sigma^5 \rho^2 &= a_{501}\, \sigma^5 \rho^2 \\
a_6 g_\rho^2 \sigma^6 \rho^2 &= a_{601}\, \sigma^6 \rho^2
\end{aligned}
$$

Note that Muller and Serot use the notation

$$
\frac{\bar{\kappa} g_s^3}{2} = \frac{\kappa}{2} = b M g_s^3 \qquad \text{and} \qquad \frac{\bar{\lambda} g_s^4}{6} = \frac{\lambda}{6} = c g_s^4
$$

which differs slightly from the "standard" notation above.

We need to compare the values of

$$
\begin{gathered}
\frac{m_s^2}{2 g_s^2 M^2}, \frac{m_v^2}{2 g_v^2 M^2} \frac{m_\rho^2}{8 g_\rho^2 M^2}, \frac{b}{3}, \frac{c}{4} \\
\frac{\zeta}{24}, \frac{\xi}{384}, \frac{b_1}{4 g_\omega^2}, \frac{b_2 M^2}{4 g_\omega^4}, \frac{b_3 M^4}{4 g_\omega^6}, \frac{a_1}{4 g_\sigma M}, \\
\frac{a_2}{4 g_\sigma^2}, \frac{a_3 M}{4 g_\sigma^3}, \frac{a_4 M^2}{4 g_\sigma^4}, \frac{a_5 M^3}{4 g_\sigma^5}, \text{ and } \frac{a_6 M^4}{4 g_\sigma^6} .
\end{gathered}
$$

These values are stored in the variables cs, cw, cr, b, c, zeta, xi, b1, etc. in the specified rmf_eos object. All of the numbers should be around 0.001 or 0.002.

For the scale M, mnuc is used.

Todo

I may have ignored some signs in the above, which are unimportant for this application, but it would be good to fix them for posterity.

Definition at line 561 of file rmf_eos.h.

5.26.2.16 int naturalness_limits (double *value*, rmf_eos & *re*) [inline]

Provide the maximum values of the couplings assuming a limit on naturalness.

The limits for the couplings are function of the nucleon and meson masses, except for the limits on `b`, `c`, `zeta`, and `xi` which are independent of the masses because of the way that these four couplings are defined.

Definition at line 598 of file rmf_eos.h.

5.26.2.17 int calc_cr (double *sig*, double *ome*, double *nb*) [protected]

Calculate the `cr` coupling given `sig` and `ome` at the density 'nb'.

Used by fix_saturation().

5.26.3 Field Documentation

5.26.3.1 double mnuc

nucleon mass

This need not be exactly equal to the neutron or proton mass, but provides the scale for the coupling b.

Definition at line 227 of file rmf_eos.h.

5.26.3.2 gsl_mroot_hybrids<void *,mm_funct<void *> > def_sat_mroot

The default solver for calculating the saturation density.

Used by fn0() (which is called by saturation()) to solve saturation_matter_e() (1 variable).

Definition at line 640 of file rmf_eos.h.

5.26.3.3 double n_charge [protected]

Temporary charge density.

Todo

Should use hadronic_eos::proton_frac instead?

Definition at line 651 of file rmf_eos.h.

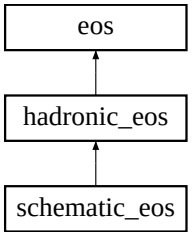
The documentation for this class was generated from the following file:

- rmf_eos.h

5.27 schematic_eos Class Reference

`#include <schematic_eos.h>`

Inheritance diagram for schematic_eos::



5.27.1 Detailed Description

Schematic hadronic equation of state.

Equation of state defined by the energy density:

$$\epsilon = n\left\{M + eoa + \frac{comp}{18}(n/n0-1)^2 + \frac{kprime}{162}(n/n0-1)^3 + \frac{kpp}{1944}(n/n0-1)^4 + (1-2x)^2\left[a\left(\frac{n}{n0}\right)^{2/3} + b\left(\frac{n}{n0}\right)^{\gamma}\right]\right\}$$

Symmetry energy at nuclear matter density is a+b.

Definition at line 49 of file schematic_eos.h.

Public Member Functions

- virtual int calc_e (**fermion** &ln, **fermion** &lp, **thermo** &lth)
  Equation of state as a function of density.
- virtual int set_kprime_zeroden ()
  Set kprime so that the energy per baryon of zero-density matter is zero.
- virtual int set_kpp_zeroden ()
  Set kpp so that the energy per baryon of zero-density matter is zero.
- virtual int set_a_from_mstar (double u_msom, double mnuc)
  Fix the kinetic energy symmetry coefficient from the nucleon effective mass and the saturation density.
- virtual double eoa_zeroden ()
  Return the energy per baryon of matter at zero density.
- virtual const char * type ()
  Return string denoting type ("schematic_eos").

Data Fields

- double a
  The kinetic energy symmetry coefficient in MeV (default 17).
- double b
  The potential energy symmetry coefficient in MeV (default 13).
- double kpp
  The coefficient of a density to the fourth term (default 0).
- double gamma
  The exponent of the high-density symmetry energy (default 1.0).

5.27.2 Member Function Documentation

5.27.2.1 virtual int set_a_from_mstar (double *u_msom*, double *mnuc*) [inline, virtual]

Fix the kinetic energy symmetry coefficient from the nucleon effective mass and the saturation density.

This assumes the nucleons are non-relativistic and that the neutrons and protons have equal mass. The relativistic corrections are around 1 **part** in 10^6.

Todo

This was computed in schematic_sym.nb, which might be added to the documentation?

Definition at line 105 of file schematic_eos.h.

5.27.2.2 virtual double eoa_zeroden () [inline, virtual]

Return the energy per baryon of matter at zero density.

This is inaccessible from calc_e() so is available separately here. Using set_kprime_zeroden() or set_kpp_zeroden() will fix kprime or kpp (respectively) to ensure that this is zero.

The result provided here does not include the nucleon mass and is given in fm^{-1}.

Definition at line 121 of file schematic_eos.h.

5.27.3 Field Documentation

5.27.3.1 double a

The kinetic energy symmetry coefficient in MeV (default 17).

The default value corresponds to an effective mass of about 0.7.

Definition at line 59 of file schematic_eos.h.

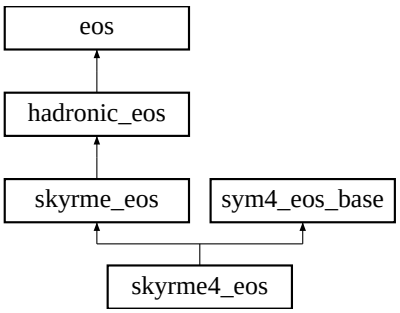
The documentation for this class was generated from the following file:

- schematic_eos.h

5.28 skyrme4_eos Class Reference

`#include <sym4_eos.h>`

Inheritance diagram for skyrme4_eos::



5.28.1 Detailed Description

A version of skyrme_eos to separate potential and kinetic contributions.

Definition at line 111 of file sym4_eos.h.

Public Member Functions

- virtual int calc_e_sep (**fermion** &ne, **fermion** &pr, double &ed_kin, double &ed_pot, double &mu_n_kin, double &mu_p_kin, double &mu_n_pot, double &mu_p_pot)
  Compute the potential and kinetic parts separately.

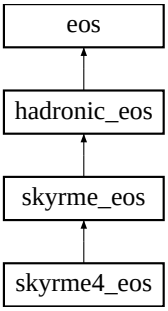
The documentation for this class was generated from the following file:

- sym4_eos.h

5.29 skyrme_eos Class Reference

`#include <skyrme_eos.h>`

Inheritance diagram for skyrme_eos::



5.29.1 Detailed Description

Skyrme hadronic equation of state at zero temperature.

Quantities which have units containing powers of energy are divided by $\hbar c$ to ensure all quantities are in units of fm. The x_i and α are unitless, while the original units of the t_i are:

- t_0 - MeV fm^3
- t_1 - MeV fm^5
- t_2 - MeV fm^5
- t_3 - MeV $\mathrm{fm}^{3(1+\alpha)}$

These are stored internally with units of:

- t_0 - fm^2
- t_1 - fm^4
- t_2 - fm^4
- t_3 - $\mathrm{fm}^{2+3\alpha}$

The class skyrme_eos_io uses **o2scl_const::hc_mev_fm** for I/O so that all files contain the parameters in the original units.

The chemical potentials do include the rest mass energy.

n.ed and p.ed contain tau_n and tau_p which are in fm^{-5}, not fm^{-4}.

The functions for the usual saturation properties are based partly on Brack85.

The Hamiltonian is defined below. $a = 0, b = 1$ gives the standard definition of the Skyrme Hamiltonian (ref?), while $a = 1, b = 0$ contains the modifications suggested by Osni94. Check also Dutta86.

$$\mathcal{H} = \mathcal{H}_{k1} + \mathcal{H}_{k2} + \mathcal{H}_{k3} + \mathcal{H}_{p1} + \mathcal{H}_{p2} + \mathcal{H}_{p3} + \mathcal{H}_{g1} + \mathcal{H}_{g2}$$

The kinetic terms are:

$$\mathcal{H}_{k1} = \frac{\tau_n}{2m_n} + \frac{\tau_p}{2m_p}$$

$$\mathcal{H}_{k2} = n\left(\tau_n + \tau_p\right)\left[\frac{t_1}{4}\left(1 + \frac{x_1}{2}\right) + \frac{t_2}{4}\left(1 + \frac{x_2}{2}\right)\right]$$

$$\mathcal{H}_{k3} = (\tau_n n_n + \tau_p n_p) \left[\frac{t_2}{4} \left(\frac{1}{2} + x_2 \right) - \frac{t_1}{4} \left(\frac{1}{2} + x_1 \right) \right]$$

The potential terms are:

$$\mathcal{H}_{p1} = \frac{t_0}{2} \left[\left(1 + \frac{x_0}{2} \right) n^2 - \left(\tfrac{1}{2} + x_0 \right) \left(n_n^2 + n_p^2 \right) \right]$$

$$\mathcal{H}_{p2} = \frac{a t_3}{6} \left[\left(1 + \frac{x_3}{2} \right) n^\alpha n_n n_p + 2^{\alpha-2} \left(1 - x_3 \right) \left(n_n^{\alpha+2} + n_p^{\alpha+2} \right) \right]$$

$$\mathcal{H}_{p3} = \frac{b t_3}{12} \left[\left(1 + \frac{x_3}{2} \right) n^{\alpha+2} - \left(\frac{1}{2} + x_3 \right) n^\alpha \left(n_n^2 + n_p^2 \right) \right]$$

The **gradient** terms are displayed here for completeness even though they are not computed in the code:

$$\mathcal{H}_{g1} = \frac{3}{32} \left[t_1 \left(1 - x_1 \right) - t_2 \left(1 + x_2 \right) \right] \left[\left(\nabla n_n \right)^2 + \left(\nabla n_p \right)^2 \right]$$

$$\mathcal{H}_{g2} = \frac{1}{8} \left[3 t_1 \left(1 + \frac{x_1}{2} \right) - t_2 \left(1 + \frac{x_2}{2} \right) \right] \nabla n_n \nabla n_p$$

A couple useful definitions:

$$t_3' = \left(a + b \right) t_3$$

$$C = \frac{3}{10 m} \left(\frac{3 \pi^2}{2} \right)^{2/3}$$

$$\beta = \frac{M}{2} \left[\frac{1}{4} \left(3 t_1 + 5 t_2 \right) + t_2 x_2 \right]$$

Models are taken from the references: Friedrich86, Dutta86, VanGiai81, Chabanat95, Chabanat97, Beiner75, Reinhard95, Bartel79, Dobaczewski94, Osni94, Tondeur84 and others.

See Mathematica notebook at

```
doc/o2scl/extras/skyrme_eos.nb
doc/o2scl/extras/skyrme_eos.ps
```

Finite temperature documentation

The finite temperature extension is performed using the method Prakash97 (see also the Windsurfing the Fermi Sea proceedings by Prakash).

Returned chemical potentials include the rest mass, and energy densities include the rest mass energy density.

The variables n.nu and p.nu contain the expressions (-mu_n+V_n)/temper and (-mu_p+V_p)/temper respectively, where V is the potential **part** of the single particle energy for particle i (i.e. the derivative of the Hamiltonian wrt density while energy density held constant). Equivalently, n.nu is just -n.kf^2/2/mstar.

For y>4 we use a non-degenerate expansion, and for y<-40 we use a Sommerfeld expansion. For y>50 the code fails, but this is well outside the regime of applicability of this model.

Runs the zero temperature code if temper<=0.0.

The finite temperature code does not include attempt to include antiparticles and uses **part::calc_density**().

Note:

Since this EOS uses the effective masses and chemical potentials in the **fermion** class, the values of **part::non_interacting** for neutrons and protons are set to false in many of the functions.

Todos

Todo

- Make sure that this class properly handles particles for which inc_rest_mass is true/false
- What about the spin-orbit units?
- Need to write a function that calculates saturation density?
- Remove use of mnuc in calparfun()?
- The compressibility could probably use some simplification
- Make sure the finite-temperature **part** is properly tested

Definition at line 224 of file skyrme_eos.h.

Public Member Functions

- skyrme_eos (std::string model)
  Load the model named 'model'.
- virtual int calc_temp_e (**fermion** &ne, **fermion** &pr, const double temper, **thermo** &th)
  Equation of state as a function of densities.
- virtual int calc_e (**fermion** &ne, **fermion** &pr, **thermo** &lt)
  Equation of state as a function of density.
- int calpar (double gt0=-10.0, double gt3=70.0, double galpha=0.2, double gt1=2.0, double gt2=-1.0)
  Calculate t_0, t_1, t_2, t_3 and α from the saturation properties.
- int load (std::string model, bool external=false)
  Load parameters from model 'model'.
- int check_landau (double nb, double m)
  Check the Landau parameters for instabilities.
- int landau_nuclear (double n0, double m, double &f0, double &g0, double &f0p, double &g0p, double &f1, double &g1, double &f1p, double &g1p)
  Calculate the Landau parameters for nuclear matter.
- int landau_neutron (double n0, double m, double &f0, double &g0, double &f1, double &g1)
  Calculate the Landau parameters for neutron matter.
- virtual const char * type ()
  Return string denoting type ("skyrme_eos").

Saturation properties

These calculate the various saturation properties exactly from the parameters at any density. These routines often assume that the neutron and proton masses are equal.

- virtual double feoa (double nb)
  Calculate binding energy.
- virtual double fmsom (double nb)
  Calculate effective mass.
- virtual double fcomp (double nb)

Calculate compressibility.

- virtual double fesym (const double nb, const double pf=0.5)
  Calculate symmetry energy.
- virtual double fkprime (double nb)
  skewness

Data Fields

- double **t0**
- double **t1**
- double **t2**
- double **t3**
- double **x0**
- double **x1**
- double **x2**
- double **x3**
- double **alpha**
- double **a**
- double **b**
- double W0
  Spin-orbit splitting.
- bool parent_method
  Use hadronic_eos methods for saturation properties.
- **nonrel_fermion** def_nr_neutron
  Default nonrelativistic neutron.
- **nonrel_fermion** def_nr_proton
  Default nonrelativistic proton.

5.29.2 Constructor & Destructor Documentation

5.29.2.1 skyrme_eos (std::string *model*)

Load the model named 'model'.

See comments under skyrme_eos::load().

5.29.3 Member Function Documentation

5.29.3.1 virtual double feoa (double *nb*) [virtual]

Calculate binding energy.

$$\frac{E}{A} = C n_B^{2/3} (1 + \beta n_B) + \frac{3 t_0}{8} n_B + \frac{t_3'}{16} n_B^{\alpha+1}$$

5.29.3.2 virtual double fmsom (double *nb*) [virtual]

Calculate effective mass.

$$M^*/M = (1 + \beta n_B)^{-1}$$

5.29.3.3 virtual double fcomp (double *nb*) [virtual]

Calculate compressibility.

$$K = 10Cn_B^{2/3} + \frac{27}{4}t_0 n_B + 40C\beta n_B^{5/3} + \frac{9t_3'}{16}\alpha\left(\alpha+1\right)n_B^{1+\alpha} + \frac{9t_3'}{8}\left(\alpha+1\right)n_B^{1+\alpha}$$

Reimplemented from hadronic_eos.

5.29.3.4 virtual double fesym (const double *nb*, const double *pf* = 0.5) [virtual]

Calculate symmetry energy.

If pf=0.5, then the exact expression below is used. Otherwise, the method from class hadronic_eos is used.

$$E_{sym} = \frac{5}{9}Cn^{2/3} + \frac{10Cm}{3}\left[\frac{t_2}{6}\left(1+\frac{5}{4}x_2\right) - \frac{1}{8}t_1x_1\right]n^{5/3} - \frac{t_3'}{24}\left(\tfrac{1}{2}+x_3\right)n^{1+\alpha} - \frac{t_0}{4}\left(\tfrac{1}{2}+x_0\right)n$$

Reimplemented from hadronic_eos.

5.29.3.5 virtual double fkprime (double *nb*) [virtual]

skewness

$$2Cn_B^{2/3}\left(9-5/M^*/M\right) + \frac{27t_3'}{16}n^{1+\alpha}\alpha\left(\alpha^2-1\right)$$

Reimplemented from hadronic_eos.

5.29.3.6 int calpar (double *gt0* = -10.0, double *gt3* = 70.0, double *galpha* = 0.2, double *gt1* = 2.0, double *gt2* = -1.0)

Calculate t_0, t_1, t_2, t_3 and α from the saturation properties.

In nuclear matter:

$E_b = E_b(n_0, M^*, t_0, t_3, \alpha)$

$P = P(n_0, M^*, t_0, t_3, \alpha)$

$K = K(n_0, M^*, t_3, \alpha)$ (the t_0 dependence vanishes)

$M^* = M^*(n_0, t_1, t_2, x_2)$ (the x_1 dependence cancels),

$E_{sym} = E_{sym}(x_0, x_1, x_2, x_3, t_0, t_1, t_2, t_3, \alpha)$

To fix the couplings from the saturation properties, we take n_0, M^*, E_b, K as inputs, and we can fix t_0, t_3, α from the first three relations, then use M^*, E_b to fix t_2 and t_1. The separation into two solution steps should make for better convergence. All of the x's are free parameters and should be set before the function call.

The arguments gt0, gt3, galpha, gt1, and gt2 are used as initial guesses for t0, t3, alpha, t1, and t2 respectively.

Todo

Does this work for both 'a' and 'b' non-zero?

Todo

Compare to similar formulae from Margueron02

5.29.3.7 int load (std::string *model*, bool *external* = `false`)

Load parameters from model 'model'.

Presently accepted values from file skdata/model_list:

```
word[] models 81
E Es FitA FitB FitK FitKs FitL Gs Ly5 NRAPR NRAPR2 PeEVs PeHF PeSIs
RATP Rs SGI SGII SI SII SIII SIIIs SIV SLy0 SLy1 SLy10 SLy2 SLy230a
SLy230b SLy3 SLy4 SLy5 SLy6 SLy7 SLy8 SLy9 SV SVI SVII SkI1 SkI2 SkI3
SkI4 SkI5 SkI6 SkM SkMDIx0 SkMDIx1 SkMDIxm1 SkMDIxm2 SkMP SkMs SkNF1
SkNF2 SkO SkOp SkP SkSC10 SkSC4 SkSC5 SkSC6 SkT SkT1 SkT1s SkT2 SkT3
SkT3s SkT4 SkT5 SkT6 SkT7 SkT8 SkT9 SkkT8 Z Zs Zss mst1 mst0.81 mst0.90
SIp
```

If `external` is true, then model is the filename (relative to the current directory) of the file containing the model parameters

5.29.3.8 int check_landau (double *nb*, double *m*)

Check the Landau parameters for instabilities.

This returns zero if there are no instabilities.

5.29.3.9 int landau_nuclear (double *n0*, double *m*, double & *f0*, double & *g0*, double & *f0p*, double & *g0p*, double & *f1*, double & *g1*, double & *f1p*, double & *g1p*)

Calculate the Landau parameters for nuclear matter.

Given `n0` and `m`, this calculates the Landau parameters in nuclear matter as given in Margueron02

Todo

This needs to be checked.

(Checked once on 11/05/03)

5.29.3.10 int landau_neutron (double *n0*, double *m*, double & *f0*, double & *g0*, double & *f1*, double & *g1*)

Calculate the Landau parameters for neutron matter.

Given 'n0' and 'm', this calculates the Landau parameters in neutron matter as given in Margueron02

Todo

This needs to be checked

(Checked once on 11/05/03)

5.29.4 Field Documentation

5.29.4.1 double W0

Spin-orbit splitting.

This is unused, but included for possible future use and present in the internally stored models.

Definition at line 235 of file skyrme_eos.h.

5.29.4.2 bool parent_method

Use hadronic_eos methods for saturation properties.

This can be set to true to check the difference between the exact expressions and the numerical values from class hadronic_eos.

Definition at line 382 of file skyrme_eos.h.

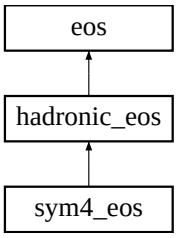
The documentation for this class was generated from the following file:

- skyrme_eos.h

5.30 sym4_eos Class Reference

`#include <sym4_eos.h>`

Inheritance diagram for sym4_eos::



5.30.1 Detailed Description

Construct an EOS with an arbitrary choice for the terms in the symmetry energy that are quartic in the isospin asymmetry.

Definition at line 155 of file sym4_eos.h.

Public Member Functions

- int set_base_eos (sym4_eos_base &seb)
  Set the base equation of state.
- virtual int test_eos (**fermion** &ne, **fermion** &pr, **thermo** &lth)
  Test the equation of state.
- virtual int calc_e (**fermion** &ne, **fermion** &pr, **thermo** &lth)
  Equation of state as a function of density.

Data Fields

- double alpha
  The strength of the quartic terms.

Protected Attributes

- sym4_eos_base * sp
  The base equation of state to use.

5.30.2 Member Function Documentation

5.30.2.1 virtual int test_eos (fermion & *ne*, fermion & *pr*, thermo & *lth*) [virtual]

Test the equation of state.

This compares the chemical potentials from calc_e_sep() to their finite-difference approximations in order to ensure that the separation into potential and kinetic parts is done properly.

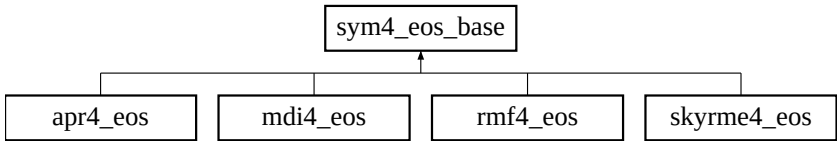
The documentation for this class was generated from the following file:

- sym4_eos.h

5.31 sym4_eos_base Class Reference

`#include <sym4_eos.h>`

Inheritance diagram for sym4_eos_base::



5.31.1 Detailed Description

A class to compute quartic contributions to the symmetry energy.

Definition at line 40 of file sym4_eos.h.

Public Member Functions

- virtual int calc_e_alpha (**fermion** &ne, **fermion** &pr, **thermo** &lth, double &alphak, double &alphap, double &alphat, double &diff_kin, double &diff_pot, double &ed_kin_nuc, double &ed_pot_nuc)
  Compute alpha at the specified density.
- virtual double calc_muhat (**fermion** &ne, **fermion** &pr)
  Compute $\hat{\mu}$, the out-of-whack parameter.
- virtual int calc_e_sep (**fermion** &ne, **fermion** &pr, double &ed_kin, double &ed_pot, double &mu_n_kin, double &mu_p_kin, double &mu_n_pot, double &mu_p_pot)
  Compute the potential and kinetic parts separately (to be overwritten in children).

Protected Attributes

- **fermion** e
  An electron for the computation of the $\hat{\mu}$.

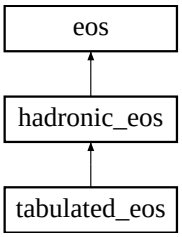
The documentation for this class was generated from the following file:

- sym4_eos.h

5.32 tabulated_eos Class Reference

`#include <tabulated_eos.h>`

Inheritance diagram for tabulated_eos::



5.32.1 Detailed Description

EOS from a **table**.

This assumes a symmetry energy which depends quadratically on the isospin asymmetry in order to construct an EOS from a **table** of baryon density and energy per baryon for both nuclear and pure neutron matter.

Note: If using a tabulated EOS to compute derivatives (like the compressibility which effectively requires a second derivative), it is important to tabulated the EOS precisely enough to ensure that the derivatives are accurate. In the case of ensuring that the compressibility at saturation density is well reproduced, I have needed the EOS to be specified with at least 6 digits of precision on a grid at least as small as 0.002 fm^{-3}.

Definition at line 53 of file tabulated_eos.h.

Public Member Functions

- int **free_table** ()
- virtual int calc_e (**fermion** &ne, **fermion** &pr, **thermo** &th)
  Equation of state as a function of density.
- template<class vec_t>
  int set_eos (size_t n, vec_t &rho, vec_t &Enuc, vec_t &Eneut)
  Set the EOS through vectors specifying the densities and energies.
- template<class vec_t>
  int set_eos (size_t n_nuc, vec_t &rho_nuc, vec_t &E_nuc, size_t n_neut, vec_t &rho_neut, vec_t &E_neut)
  Set the EOS through vectors specifying the densities and energies.
- **table** & get_nuc_table ()
  Return the internal **table**.
- **table** & get_neut_table ()
  Return the internal **table**.
- int read_file (std::string fname, std::string rho_str, std::string nuc_str, std::string neut_str)
  Set the EOS by specifying a file containing a **table** *object.*

Protected Attributes

- bool table_alloc
  True if the **table** *has been allocated.*
- bool one_table
  If true, then tnuc and tneut point to the same **table**.

The EOS tables

- **table ∗ tnuc**
- **table ∗ tneut**

Strings for the column names

- std::string **srho_nuc**
- std::string **srho_neut**
- std::string **snuc**
- std::string **sneut**

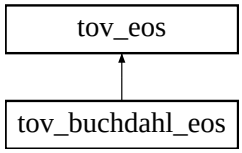
The documentation for this class was generated from the following file:

- tabulated_eos.h

5.33 tov_buchdahl_eos Class Reference

`#include <tov_eos.h>`

Inheritance diagram for tov_buchdahl_eos::



5.33.1 Detailed Description

The Buchdahl EOS for the TOV solver.

Given the eos

$$\rho = 12\sqrt{p_* P} - 5P$$

the TOV equation has an analytical solution

$$R = (1-\beta)\sqrt{\frac{\pi}{288 p_* G(1-2\beta)}}$$

where $\beta = GM/R$.

The central pressure and energy density are

$$P_c = 36 p_* \beta^2$$

$$\rho_c = 72 p_* \beta (1 - 5\beta/2)$$

Physical solutions are obtained only for $P < 25p_*/144$ and $\beta < 1/6$.

(Lattimer & Prakash, 2001)

Todo

Fix the reference above

Idea for future

Figure out what to do with the buchfun() function

Definition at line 321 of file tov_eos.h.

Public Member Functions

- virtual int get_eden (double P, double &e, double &nb)
  Given the pressure, produce the energy and number densities.
- virtual int get_aux (double P, size_t &np, **ovector_view** &auxp)
  Given the pressure, produce all the remaining quantities.
- virtual int get_names (size_t &np, std::vector< std::string > &pnames)
  Fill a list with strings for the names of the remaining quanities.

Data Fields

- double Pstar
  The parameter with units of pressure in units of solar masses per km cubed (default value 3.2×10^{-5} *).*

5.33.2 Member Function Documentation

5.33.2.1 virtual int get_eden (double *P*, double & *e*, double & *nb*) [inline, virtual]

Given the pressure, produce the energy and number densities.

If the baryon density is not specified, it should be set to zero or baryon_column should be set to false

Reimplemented from tov_eos.

Definition at line 344 of file tov_eos.h.

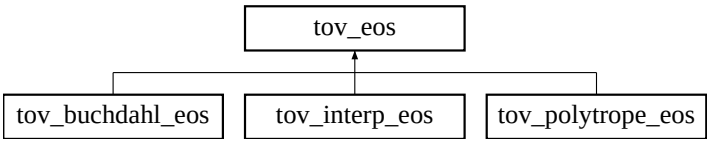
The documentation for this class was generated from the following file:

- tov_eos.h

5.34 tov_eos Class Reference

`#include <tov_eos.h>`

Inheritance diagram for tov_eos::



5.34.1 Detailed Description

A EOS base class for the TOV solver.

Definition at line 54 of file tov_eos.h.

Public Member Functions

- virtual int get_eden (double P, double &e, double &nb)
  Given the pressure, produce the energy and number densities.
- virtual int get_aux (double P, size_t &np, **ovector_view** &auxp)
  Given the pressure, produce all the remaining quantities.
- virtual int get_names (size_t &np, std::vector< std::string > &pnames)
  Fill a list with strings for the names of the remaining quanities.

Data Fields

- int verbose
  control for output (default 1)
- bool baryon_column
  Set to true if the baryon density is provided in the EOS (default false).

5.34.2 Member Function Documentation

5.34.2.1 virtual int get_eden (double *P*, double & *e*, double & *nb*) [inline, virtual]

Given the pressure, produce the energy and number densities.

If the baryon density is not specified, it should be set to zero or baryon_column should be set to false

Reimplemented in tov_interp_eos, tov_buchdahl_eos, and tov_polytrope_eos.

Definition at line 74 of file tov_eos.h.

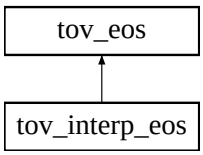
The documentation for this class was generated from the following file:

- tov_eos.h

5.35 tov_interp_eos Class Reference

`#include <tov_eos.h>`

Inheritance diagram for tov_interp_eos::



5.35.1 Detailed Description

An EOS for the TOV solver using simple linear interpolation and a default low-density EOS.

Internally, energy and pressure are stored in units of solar masses per cubic kilometer and baryon density is stored in units of fm^{-3}. The user-specified EOS **table** is left as is, and unit conversion is performed as needed in get_eden() and other functions so that results are returned in the units specified by set_units().

The function set_units() needs to be called before either of the functions get_eden() or get_eden_ld() are called. The function set_units() may be called after calling either the read_table() functions or the set_low_density_eos() function.

Definition at line 118 of file tov_eos.h.

Public Member Functions

- virtual int get_eden (double pres, double &ed, double &nb)
  Given the pressure, produce the energy and number densities.
- virtual int get_eden_user (double pres, double &ed, double &nb)
  Given the pressure, produce the energy and number densities from the user-specified EOS.
- virtual int get_eden_ld (double pres, double &ed, double &nb)
  Given the pressure, produce the energy and number densities from the low-density EOS.
- virtual int get_aux (double P, size_t &nv, **ovector_view** &auxp)

Given the pressure, produce all the remaining quantities.

- virtual int get_names (size_t &np, std::vector< std::string > &pnames)
  Fill a list with strings for the names of the remaining quanities.
- int read_table (**table** &eosat, std::string s_cole="ed", std::string s_colp="pr", std::string s_colnb="nb")
  Specify the EOS through a **table**.
- int read_table_file (std::string eosfn, std::string s_cole="ed", std::string s_colp="pr", std::string s_colnb="nb")
  Specify the EOS through a **table** *in a file.*
- int set_low_density_eos (bool s_ldeos, std::string s_nvpath, int s_nvcole=0, int s_nvcolp=1, int s_nvcolnb=2)
  Set the low-density EOS.
- int set_units (double s_efactor, double s_pfactor, double s_nfactor)
  Set the units of the user-specified EOS.
- int set_units (std::string leunits="", std::string lpunits="", std::string lnunits="")
  Set the units of the user-specified EOS.
- int get_transition (double &plow, double &ptrans, double &phi)
  Return limiting and transition pressures.
- int set_transition (double ptrans, double pw)
  Set the transition pressure and "width".

Protected Member Functions

- int check_eos ()
  Check that the EOS is valid.
- void interp (const **ovector_view** &x, const **ovector_view** &y, double xx, double &yy, int n1, int n2)
  Linear EOS interpolation.

Protected Attributes

- **base_ioc** bio
  A base I/O object for reading EOSs.

Low-density EOS

- bool ldeos
  true if we are using the low-density eos (false)
- bool ldeos_read
  Low-density EOS switch.
- std::string ldpath
  the path to the low-density EOS
- int ldcole
  column in low-density eos for energy density
- int ldcolp
  column in low-density eos for pressure
- int ldcolnb
  column in low-density eos for baryon density
- **table** * ld_eos
  file containing low-density EOS
- double presld
  highest pressure in low-density EOS
- double eld
  highest energy density in low-density EOS
- double nbld
  highest baryon density in low-density EOS
- double prest
  Transition pressure.
- double pwidth
  Transition width.

User EOS

- **table** * eost
  file containing eos
- int nfile
  number of lines in eos file
- int cole
  column for energy density in eos file
- int colp
  column for pressure in eos file
- int colnb
  column for baryon density in eos file
- bool eos_read
  True if an EOS has been specified.

Units

- std::string eunits
  Units for energy density.
- std::string punits
  Units for pressure.
- std::string nunits
  Units for baryon density.
- double efactor
  unit conversion factor for energy density (default 1.0)
- double pfactor
  unit conversion factor for pressure (default 1.0)
- double nfactor
  unit conversion factor for baryon density (default 1.0)

5.35.2 Member Function Documentation

5.35.2.1 int get_transition (double & *plow*, double & *ptrans*, double & *phi*)

Return limiting and transition pressures.

Returns, in order:

- the highest pressure in the low-density EOS
- the transition pressure
- the lowest pressure in the high-density EOS

5.35.2.2 int set_transition (double *ptrans*, double *pw*)

Set the transition pressure and "width".

Sets the transition pressure and the width (specified as a number greater than unity in `pw`) of the transition between the two EOSs. The transition is done smoothly using linear interpolation between $P = \mathrm{ptrans}/pmathrmpw$ and $P = \mathrm{ptrans} \times pmathrmpw$.

5.35.3 Field Documentation

5.35.3.1 bool ldeos_read [protected]

Low-density EOS switch.

This is `true` if the ldeos has been read by set_ldeos. This is useful, since then we know whether or not we need to free the memory for the LD EOS in the destructor

Definition at line 218 of file tov_eos.h.

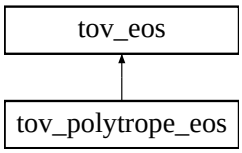
The documentation for this class was generated from the following file:

- tov_eos.h

5.36 tov_polytrope_eos Class Reference

`#include <tov_eos.h>`

Inheritance diagram for tov_polytrope_eos::



5.36.1 Detailed Description

Standard polytropic EOS $p = K\rho^{1+1/n}$.

Any units are permissible, but if this is to be used with tov_solve, then the units of K must be consistent with the units set in tov_solve::set_units().

Definition at line 388 of file tov_eos.h.

Public Member Functions

- virtual int get_eden (double P, double &e, double &nb)
  Given the pressure, produce the energy and number densities.
- virtual int get_aux (double P, size_t &np, **ovector_view** &auxp)
  Given the pressure, produce all the remaining quantities.
- virtual int get_names (size_t &np, std::vector< std::string > &pnames)
  Fill a list with strings for the names of the remaining quanities.

Data Fields

- double K
  Coefficient (default 1.0).
- double n
  Index (default 3.0).

5.36.2 Member Function Documentation

5.36.2.1 virtual int get_eden (double *P*, double & *e*, double & *nb*) `[inline, virtual]`

Given the pressure, produce the energy and number densities.

If the baryon density is not specified, it should be set to zero or baryon_column should be set to false

Reimplemented from tov_eos.

Definition at line 413 of file tov_eos.h.

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

- tov_eos.h

5.37 tov_solve Class Reference

`#include <tov_solve.h>`

5.37.1 Detailed Description

Class to solve the Tolman-Oppenheimer-Volkov equations.

Integrate Tolman-Oppenheimer-Volkov (TOV) equations

The present code, as demonstrated in the tests, gives the correct central pressure and energy density of the analytical solution by Buchdahl to within less than 1 **part** in 10^5.

The TOV equations (i.e. Einstein's equations for a static spherically symmetric object) are

$$\frac{dm}{dr} = 4\pi r^2 \varepsilon$$

$$\frac{dP}{dr} = -\frac{G\varepsilon m}{r^2}\left(1+\frac{P}{\varepsilon}\right)\left(1+\frac{4\pi P r^3}{m}\right)\left(1-\frac{2Gm}{r}\right)^{-1}$$

where r is the radial coordinate, $m(r)$ is the mass enclosed within a radius r, and $\varepsilon(r)$ and $P(r)$ are the energy density and pressure at r, and G is the gravitational constant. The boundary conditions are $m(r=0)=0$ the condition $P(r=R)=0$ for some fixed radius R. These boundary conditions give a series of solutions to the TOV equations as a function of the radius, although they do not necessarily have a solution for all radii.

The gravitational mass is given by

$$M_G = \int_0^R 4\pi r^2 \varepsilon dr$$

while the baryonic mass is given by

$$M_B = \int_0^R 4\pi r^2 n_B m_B \left(1-\frac{Gm}{r}\right)^{-1/2} dr$$

where $n_B(r)$ is the baryon number density at radius r and m_B is the mass of one baryon.

The gravitational potential, $\Phi(r)$ can be determined from

$$\frac{d\Phi}{dr} = -\frac{1}{\varepsilon}\frac{dP}{dr}\left(1+\frac{P}{\varepsilon}\right)^{-1}$$

The proper boundary condition for the gravitational potential is

$$\Phi(r=R) = \frac{1}{2}\ln\left(1-\frac{2GM}{R}\right)$$

The equation of state may be changed at any time.

Note that the pressure in the low-density eos is not strictly increasing! (see at P=4.3e-10)

Screen output

- `verbose=0` - Nothing (even if an error occurs).
- `verbose=1` - Some basic information and any errors or warnings.
- `verbose=2` - For each profile computation, information at every kilometer.
- `verbose=3` - Profile information every 20 grid points. A keypress is required after each profile.

Todo

- The error handler is called in tov_solve()derivs for pressures less than the minimum even in normal circumstances. This should be fixed, so that errors are more rare
- baryon mass doesn't work for fixed() (This may be fixed. We should make sure it's tested.)
- Combine maxoutsize and kmax?

- Document column naming issues

Idea for future

- Turn numbers in max() function into variables

Definition at line 113 of file tov_solve.h.

Public Member Functions

- int solution_check ()
  Check the solution (unfinished).
- int set_units (double s_efactor=1.0, double s_pfactor=1.0, double s_nbfactor=1.0)
  Set units.
- int set_units (std::string eunits="", std::string punits="", std::string nunits="")
  Set units.
- int set_kmax (int s_maxoutsize=400, int s_kmax=40000)
  Set maximum storage for integration.
- int set_eos (tov_eos &ter)
  Set the EOS to use.
- int set_mroot (**mroot**< void *, **mm_funct**< void * > > &s_mrp)
  Set solver.
- int set_minimize (**minimize**< void *, **funct**< void * > > &s_mp)
  Set minimizer.
- int set_stepper (**adapt_step**< void *, **ode_funct**< void * > > &sap)
  Set the adaptive stepper.

Results

- **table** & get_results ()
  Return the results data **table**.

Actual solution of equations

- int mvsr ()
  Calculate the mass vs. radius curve.
- int fixed (double d_tmass)
  Calculate the profile of a star with fixed mass.
- int max ()
  Calculate the profile of the maximum mass star.

Data Fields

- **gsl_min_brent**< void *, **funct**< void * > > def_min
  The default minimizer.
- **gsl_mroot_hybrids**< void *, **mm_funct**< void * > > def_solver
  The default solver.
- **gsl_astep**< void *, **ode_funct**< void * > > def_stepper
  The default adaptive stepper.
- bool compute_ang_vel
  If true, compute the angular velocity (default false).
- double cap_omega
  The angular velocity.
- double schwarz_km
  The schwarzchild radius in km.

Basic properties

- double mass
  mass
- double rad
  radius
- double bmass
  baryonic mass
- double gpot
  gravitational potential

Solution parameters

- bool generel
  Use general relativistic version (default true).
- bool calcgpot
  calculate the gravitational potential and the enclosed baryon mass (default false)
- double hmin
  smallest allowed radial stepsize (default 1.0e-4)
- double hmax
  largest allowed radial stepsize (default 0.05)
- double hstart
  initial radial stepsize (default 4.0e-3)
- int verbose
  control for output (default 1)
- double maxradius
  maximum radius for integration in km (default 60)

Mass versus radius parameters

- double prbegin
  Beginning pressure (default 7.0e-7).
- double prend
  Ending pressure (default 8.0e-3).
- double princ
  Increment for pressure (default 1.1).
- bool logmode
  Use 'princ' as a multiplier, not an additive increment (default true).
- double prguess
  Guess for central pressure in solar masses per km3 (default 5.2×10^{-5}*).*

Fixed mass parameters

- double tmass
  Target mass.

Protected Member Functions

- int make_unique_name (std::string &col, std::vector< std::string > &cnames)
  Ensure `col` *does not match strings in* `cnames`.
- int derivs (double x, size_t nv, const **ovector_view** &y, **ovector_view** &dydx, void *&pa)
  The ODE step function.
- int derivs_ang_vel (double x, size_t nv, const **ovector_view** &y, **ovector_view** &dydx, void *&pa)
  The ODE step function for the angular velocity.
- int profile_out (double xx)
  Output a stellar profile.
- double maxfun (double maxx, void *&pa)
  The minimizer function to compute the maximum mass.
- int starfun (size_t ndvar, const **ovector_view** &ndx, **ovector_view** &ndy, void *&pa)
  The solver function to compute the stellar profile.
- int ang_vel ()
  Compute the angular velocity.

Protected Attributes

- tov_eos * te
  The EOS.
- bool eos_set
  True if the EOS has been set.
- **base_ioc** bio
  Define some necessary I/O objects.
- int kmax
  maximum storage size (default 40000)
- int maxoutsize
  maximum size of output file (default 400)
- double presmin
  Smallest allowed pressure for integration (default: -100).
- **table** out_table
  The output **table**.
- **mroot**< void *, **mm_funct**< void * > > * mroot_ptr
  The solver.
- **minimize**< void *, **funct**< void * > > * min_ptr
  The minimizer.
- **adapt_step**< void *, **ode_funct**< void * > > * as_ptr
  The default adaptive stepper.
- **smart_interp** smi
  Interpolation object for derivs_ang_vel().

User EOS

- std::string eunits
  Units for energy density.
- std::string punits
  Units for pressure.
- std::string nunits
  Units for baryon density.
- double efactor
  unit conversion factor for energy density (default 1.0)
- double pfactor
  unit conversion factor for pressure (default 1.0)
- double nfactor
  unit conversion factor for baryon density (default 1.0)

Integration storage

- **ovector rky** [6]
- **ovector rkx**
- **ovector rkdydx** [6]

5.37.2 Member Function Documentation

5.37.2.1 int set_units (std::string *eunits* = "", std::string *punits* = "", std::string *nunits* = "")

Set units.

Valid entries for the units of energy density and pressure are:

- "g/cm^3"
- "erg/cm^3"
- "MeV/fm^3"

- "fm^-4"
- "Msun/km^3" (i.e. solar masses per cubic kilometer)

Valid entries for the units of baryon density are:

- "m^-3"
- "cm^-3"
- "fm^-3"

5.37.2.2 int set_kmax (int *s_maxoutsize* = 400, int *s_kmax* = 40000)

Set maximum storage for integration.

The variable s_kmax is the maximum number of radial integration stepsk while s_maxoutsize is the maximum number of points that will be output for any profile.

If s_kmax is less than zero, there is no limit on the number of radial steps.

5.37.3 Field Documentation

5.37.3.1 bool generel

Use general relativistic version (default true).

These parameters can be changed at any time.

Definition at line 134 of file tov_solve.h.

5.37.3.2 double prguess

Guess for central pressure in solar masses per km3 (default 5.2×10^{-5}).

This guess is used in the function fixed().

Definition at line 167 of file tov_solve.h.

5.37.3.3 double tmass

Target mass.

Use negative values to indicate a mass measured relative to the maximum mass. For example, if the EOS has a maximum mass of 2.0, then -0.15 will give the profile of a 1.85 solar mass star.

Definition at line 180 of file tov_solve.h.

5.37.3.4 double presmin [protected]

Smallest allowed pressure for integration (default: -100).

This can't be much smaller since we need to compute numbers near exp(-presmin)

Definition at line 338 of file tov_solve.h.

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

- tov_solve.h

Index