

LIBRARY
Michigan State
University

This is to certify that the

thesis entitled

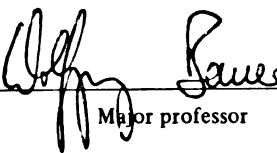
Numerical Study of Rotating
Core Collapse Supernovae

presented by

Mark Tobias Bollenbach

has been accepted towards fulfillment
of the requirements for

M.S. degree in Physics


Major professor

Date July 22, 2002

PLACE IN RETURN BOX to remove this checkout from your record.
TO AVOID FINES return on or before date due.
MAY BE RECALLED with earlier due date if requested.

DATE DUE	DATE DUE	DATE DUE

NUMERICAL STUDY OF ROTATING CORE COLLAPSE SUPERNOVAE

By

Mark Tobias Bollenbach

A THESIS

Submitted to
Michigan State University
in partial fulfillment of the requirements
for the degree of

MASTER OF SCIENCE

Department of Physics and Astronomy

2002

ABSTRACT

NUMERICAL STUDY OF ROTATING CORE COLLAPSE SUPERNOVAE

By

Mark Tobias Bollenbach

The explosion mechanism of core collapse supernovae is still far from being understood. In this work, an overview of the current understanding of core collapse supernovae and the history of numerical simulations that helped develop it is given. While it is widely believed that neutrino heating and convection above the neutrino sphere are the key processes to revive a stalled shock and thus obtain successful explosions, it is still possible that other phenomena are crucial for the explosion mechanism. For example, recent observations of the polarization of the light emitted by supernova explosions indicate that there are large deviations from spherical symmetry in the very heart of the explosion.

In contrast to most of the previous simulations which were performed in one or two dimensions, we use the different approach of a three dimensional test particle based simulation. The underlying microphysics is crudely simplified to make this computationally possible.

A systematic study of the influence of rotation mainly during the infall phase of the collapse of a typical iron core is performed. Different equations of state and initial conditions are used. Indications for significant deviations from spherical symmetry are found in our simulations.

ACKNOWLEDGMENTS

Most importantly, I would like to thank Professor Wolfgang Bauer for giving me the opportunity to work at the National Superconducting Cyclotron Laboratory and for being my advisor for this research project during this fantastic year at Michigan State University. I enjoyed the relaxed work environment and the beauty of this place very much.

Further, I would like to thank the German National Scholarship Foundation, the Studienstiftung des Deutschen Volkes, for the support I obtained during this year and for an unforgettable meeting in Washington, D.C. in April 2002.

Finally, I thank my family for supporting me throughout the years and thus making all this possible.

Contents

LIST OF TABLES	vii
LIST OF FIGURES	viii
1 Introduction	1
2 Overview of Type II Supernovae	3
2.1 Presupernova Stellar Evolution and Supernova Progenitor for a $15M_{\odot}$ Star	4
2.1.1 Stellar Evolution	5
2.1.2 Supernova Progenitor	6
2.2 Explosion Mechanism of Type II Supernovae	10
2.2.1 Collapse	10
2.2.2 Core Bounce	14
2.2.3 Prompt Shock Mechanism	15
2.2.4 Delayed Shock Mechanism	17
2.3 Recent Numerical Studies of Core Collapse Supernovae	22
2.3.1 Equation of State	23
2.3.2 One Dimensional Simulations	26
2.3.3 Two Dimensional Simulations	29
2.3.4 Three Dimensional Simulations	33
3 Our Three Dimensional Simulation	35
3.1 Test Particle Method	36
3.2 Grid	37
3.2.1 Cells and Boundaries	37
3.2.2 Grid Coordinates	38
3.2.3 Calculation of Densities	40
3.2.4 Calculation of Derivatives	42

3.3	Equation of State	44
3.3.1	Cold Nuclear EOS and Polytrope EOS	44
3.3.2	Calculation of Thermodynamic Quantities other than Density	45
3.3.3	Helmholtz EOS and Lattimer & Swesty EOS	47
3.3.4	How the EOS Affects the Dynamics	49
3.4	Symmetry Assumptions, Boundary Conditions, and Numerical Problems	52
3.4.1	Symmetry Assumptions	52
3.4.2	Derivatives at Grid Boundaries	53
3.4.3	Background Density	54
3.4.4	Singularity Treatment	54
3.5	Time Development	54
3.6	Calculation of Observables	55
3.7	Advantages and Weaknesses of this Method	56
3.8	Computational Requirements	57
3.9	Output Possibilities	57
3.9.1	Density Output	58
3.9.2	Test Particle Output	58
4	Numerical Results and Interpretation	59
4.1	Angular Momentum Conservation	59
4.2	Results of Simulation Runs	59
4.2.1	Cold Nuclear EOS without Electron Contribution	60
4.2.2	Cold Core with Polytrope Electron Contribution	66
4.2.3	Combination of Helmholtz EOS and Lattimer & Swesty EOS	72
4.3	Possible Implications of these Results	77
5	Summary and Conclusion	79
A	Source Code of the Simulation Program	83
A.1	<code>suno.cpp</code> and <code>suno.h</code>	83
A.2	<code>testparticle.cpp</code> and <code>testparticle.h</code>	125
A.3	<code>vector_and_spherical.cpp</code> and <code>vector_and_spherical.h</code>	128
B	Source Code of the Output Programs	133
B.1	<code>output.vbp</code>	133
B.2	<code>Suno Density Output.vbp</code>	140
C	Abbreviations and Symbols	148

List of Tables

2.1	Iron core masses of different supernova progenitors (taken from [56], all masses in units of M_{\odot})	9
4.1	Simulation runs using the cold nuclear EOS. The abbreviations are explained in the text.	61
4.2	Key for figure 4.3. t_a through t_e are the times (in ms) corresponding to the density profiles labeled (a) through (e) in the figure. r_a through r_e are the corresponding radii (in km) of the shown density plots. . .	64
4.3	Simulation runs using the combination of the cold nuclear EOS and the polytrope EOS for the electron gas. The abbreviations are explained in the text.	68
4.4	Key for figure 4.6. t_a through t_e are the times (in ms) corresponding to the density profiles labeled (a) through (e) in the figure. r_a through r_e are the corresponding radii (in km) of the shown density plots. . .	70
4.5	Simulation runs using the combination of the Helmholtz and the LS EOS. The abbreviations are explained in the text.	72
4.6	Key for figure 4.8. t_a through t_e are the times (in ms) corresponding to the density profiles labeled (a) through (e) in the figure. $\rho_{sc.min}$ and $\rho_{sc.max}$ are the densities (in g/cm^3) corresponding to the bottom end (color: light yellow) and top end (color: black) of the density key. . .	74

List of Figures

2.1	Structure of a $15M_{\odot}$ star at the onset of core collapse (taken from [56])	7
2.2	Composition of a $15M_{\odot}$ star at the onset of core collapse (taken from [56])	8
2.3	Infall velocity of the core matter V and local sound velocity A approximately one millisecond before core bounce (taken from [4] who used the results of [1] to create this plot).	13
2.4	Shock revival by neutrino heating after the shock wave has stalled (taken from [24]). The symbols are explained in the text.	18
2.5	Angular velocity profile used by Fryer and Heger [18] compared to that of different models used by Mönchmeyer and Müller (labeled “MM89”) [35]	30
3.1	Smearing of test particles for density calculation	41
3.2	Cold nuclear EOS and polytrope EOS for electron gas	45
3.3	$T(\rho)$ for the infall phase	47
3.4	$Y_e(\rho)$ for the infall phase	48
3.5	Combination of Helmholtz EOS and LS EOS used in our simulation .	49
3.6	Test particle j traveling from one grid cell to another	51
4.1	Angular momentum as a function of time for a typical simulation run	60
4.2	Time development of the different energies in the late stages of simulation CNEOS05	62
4.3	Mass density in a slice in the x-z-plane at five different times for models CNEOS03 (left), CNEOS05 (center), and CNEOS07 (right). Events at the respective times: (a) onset of simulation, (b) clumping and presence of centrifugal forces become apparent, (c) formation of “vortices” in CNEOS07, (d) core bounce, (e) shortly after core bounce. Note that the radii of the shown density profiles vary (see table 4.2 for more data). The black lines below each plot indicate a length of 43.5 km. The length scale for the plots labeled by (a) is much larger than this. Images in this thesis are presented in color.	65

4.4	Initial mass density as a function of the radius calculated by Woosley and Weaver [56] (solid line) and the variant contracted by a factor of 5 we use in our simulation (dashed line). The location at which the enclosed mass is $0.7M_{\odot}$ (edge of the inner core) is indicated for both distributions on the abscissa.	67
4.5	Time development of the different energies in simulation PCNEOS19	69
4.6	Mass density in a slice in the x-z-plane at five different times for models PCNEOS07 (left), PCNEOS13 (center), and PCNEOS19 (right). Events at the respective times: (a) onset of simulation, (b) after 2 ms, (c) presence of centrifugal forces and vortices become apparent, (d) core bounce, (e) shortly after core bounce. Note that the plots have different radii. The black line below each plot indicates a length of 56.1 km. See table 4.4 for more data. Images in this thesis are presented in color.	71
4.7	Time development of the different energies in simulation HLSEOS13.	73
4.8	Mass density in a slice in the x-z-plane at five different times for models HLSEOS07 (left), HLSEOS16 (center), and HLSEOS22 (right). Events at the respective times: (a) onset of simulation, (b) after 2 ms, (c) presence of centrifugal forces becomes apparent (after 3 ms), (d) core bounce, (e) shortly after core bounce. The density scale is only approximately valid: the highest density (indicated by the colors black and dark red) decreases from left to right. All plots have the same radius (123.42 km) indicated by the black line in the top left. See table 4.6 for more data. Images in this thesis are presented in color.	75
4.9	Density depletion along the z-axis after bounce in model HLSEOS22 at $t = 5.21\text{ms}$. The radius in this plot (measured from the center horizontally to the right) is 51.43km. Images in this thesis are presented in color.	76

Chapter 1

Introduction

Supernova explosions are among the most spectacular phenomena that we know of. It seems that the supernova explosion is also one of the most challenging phenomena as far as the understanding of the underlying physics is concerned.

The most famous recent supernova was observed in 1987 (therefore labeled SN 1987A). It could be seen with the naked eye and is particularly interesting because it was the first supernova whose progenitor star had been observed before the explosion. While such observations of supernovae (that we know of) date back almost 2000 years, the theory of core collapse supernovae has rapidly developed in the last decades. Several different explosion mechanisms have been suggested throughout the years. Most of these turned out to be fundamentally wrong. Apart from being remarkable optical events, supernovae are believed to play an important role in the synthesis of heavier elements in the mass range $16 \leq A \leq 60$ — core collapse supernovae are especially relevant for the synthesis of oxygen. They are possibly also the origin of the mysterious *γ -ray bursts* [8].

In chapter 2 of this work the state of the art of the knowledge of the physics of core collapse supernova explosions will be briefly reviewed. The relevant concepts and processes will be explained. Due to the complexity of the problem numerical studies

play an outstanding role in the understanding of supernova events. An overview of the history of these studies with a special focus on recent developments will be given. Special attention will also be paid to the role of rotation in supernovae.

Our new approach to simulate core collapse supernovae will be presented in chapter 3. The numerical techniques used, the necessary approximations and assumptions, the implementation, and the differences of this method in comparison to previous simulations will be explained. The strengths and weaknesses of this technique will be pointed out. At the current state of the art our method can be considered a good model of reality only during the infall phase, i.e. until core bounce (these terms will be explained in chapter 2).

In chapter 4 the results of several simulation runs (using different equations of state for the core matter and a variety of initial conditions) that were performed using the technique described in chapter 3 will be presented. Conclusions about the effects rotation may have on supernovae in reality will carefully be drawn. Possible improvements to our model will be suggested and discussed.

This whole work will be summarized and reflected in chapter 5.

A typographical convention will be to write important terminology in *italic type* at the place where it occurs for the first time in this work.

We will further use standard symbols like c for the speed of light or M_{\odot} for the solar mass in text and equations. Other abbreviations and symbols will be introduced and used throughout this work. For convenience a summary of all these is given in appendix C.

Images in this thesis are presented in color.

Chapter 2

Overview of Type II Supernovae

In this chapter the physics of *type II supernovae* will be briefly reviewed. Supernova explosions are labeled either *type I* or *type II* (yet both with several subdivisions). This distinction was established due to observable differences: the most important difference between the two is the absence of hydrogen lines in the spectrum of type Is while these are present in the case of type IIs. The *light curves*¹ of the two phenomena also differ significantly.

From a theoretical point of view, type Is and IIs are almost utterly distinct phenomena. While the power source of type Is is believed to be thermonuclear burning initiated by the exceeding of a *white dwarf's*² *Chandrasekhar mass*³ due to the accretion of matter from a *companion star*, type IIs are powered by the gravitational energy released during the collapse of a star's iron core (note, however, that there are also subclasses of type Is that are powered by core collapse).

We will only deal with type IIs in this work and most of the time only with the typical case of a star in the *ZAMS*⁴ mass region $\approx 15M_{\odot}$. For more information about type Is one may refer to [56, 11, 4]. Most of what will be said here remains

¹i.e. the luminosity of the emitted light as a function of time

²a very dense star with a mass of approximately $1M_{\odot}$ but only the size of earth

³the maximum mass a white dwarf can have without collapsing ($\approx 1.4M_{\odot}$)

⁴Zero Age Main Sequence: denotes the star's properties at the beginning of its stellar evolution

valid for stars with a ZAMS mass between $\approx 11M_{\odot}$ and $\approx 40M_{\odot}$. This separation is necessary because the properties of stars during and at the end of their stellar evolution are essentially determined by their ZAMS mass (see [11, 4, 55, 20, 52] for details on stellar evolution). A very light star like our sun for example will never develop an iron core during its evolution and thus never produce a core collapse supernova. Also note that for stars in the ZAMS mass region $8M_{\odot} < M_{star} < 11M_{\odot}$ type II supernovae are believed to occur but the details of these stars' presupernova evolution and the supernova event itself depend sensitively on their ZAMS mass and are somewhat different than what will be dealt with in the rest of this work (see e.g. [56]). Finally, stars more massive than $\approx 40M_{\odot}$ are believed to lose their hydrogen mantle before the end of their life which disqualifies them as type IIs.

For simplicity, we will refer to a star with ZAMS mass $\dots M_{\odot}$ as just a $\dots M_{\odot}$ star from now on.

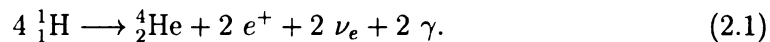
2.1 Presupernova Stellar Evolution and Supernova Progenitor for a $15M_{\odot}$ Star

As already mentioned the evolution of a star during its so-called *main sequence*⁵ is mainly determined by its ZAMS mass. It is way beyond the scope of this work to give a detailed account of the events and processes during the main sequence. Instead we will briefly follow the evolution of a $15M_{\odot}$ star to the point shortly before core collapse occurs.

⁵the long phase (possibly lasting billions of years) of the star's life during which hydrogen-burning is its dominant power source

2.1.1 Stellar Evolution

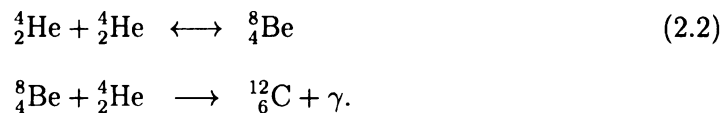
Just like our sun all stars “burn” hydrogen from the beginning of their main sequence, i.e. sequences of nuclear reactions take place which ultimately lead to the fusion of hydrogen to helium, e.g. by the *proton-proton chain* which results in the reaction



The intermediate steps have been omitted here. Other reaction chains also contribute to the conversion of hydrogen to helium.

It is well known that during this *fusion* of hydrogen a lot of energy is released because (for nuclei lighter than iron) the binding energy per nucleon increases with increasing mass number of the nucleus. However, for fusion to occur the Coulomb barrier of the (positively charged) participating nuclei (e.g. two protons) has to be overcome which requires them to have large kinetic energies. Such energies are available in the interior of stars where the temperature during the main sequence is of the order of magnitude 10^7K .

After significant amounts of helium have been produced and only if the temperature in the core of the star has become sufficiently high helium itself starts “burning” by the *triple alpha process*:



Similarly, given the needed prerequisites, “burning” of (most importantly) H, He, C, Ne, O, and Si occurs successively in the center of the star – sometimes even in its shells. Because of the increasing Coulomb barriers higher and higher temperatures (e.g. of the order of $4 \times 10^9\text{K}$ for Si-burning) are needed for these fusion reactions

and less and less energy is gained per participating nucleon. After carbon ignition the energy losses of the star are huge (compared to the previous stages) and dominated by neutrino emission since then the temperatures are high enough for the occurrence of (among other processes) the *electron capture* reaction:

$$p^+ + e^- \longrightarrow n + \nu_e. \quad (2.3)$$

The neutrinos created in this reaction hardly interact with the matter of the star and just radiate away. This way, energy is “carried” out of the star.

Thus the star develops a gigantic power using a much less effective power source than before. Consequently the time scale of the burning stages becomes smaller with increasing charge number of the fuel. For example, in a $20M_\odot$ star the H burning lasts for approximately 10^7 years, He burning 10^6 years, C burning 300 years, O burning 200 days, and Si burning merely 2 days (these numbers were taken from [11])!

The final product of these nuclear burning processes is either ^{56}Ni or ^{54}Fe after which no more energy can be gained by fusion.

2.1.2 Supernova Progenitor

It shall be mentioned that the knowledge of main sequence stellar evolution is very sophisticated and in fact much more profound than that of the supernova phenomenon itself. An incredible effort has been put into numerical simulations of stellar evolution and there is wide consensus on the development which stars pass through during their main sequence. These efforts converged to the initial conditions for the core collapse event of a $15M_\odot$ star illustrated in figures 2.1 and 2.2 and calculated by the simulations of Woosley and Weaver [52, 56].

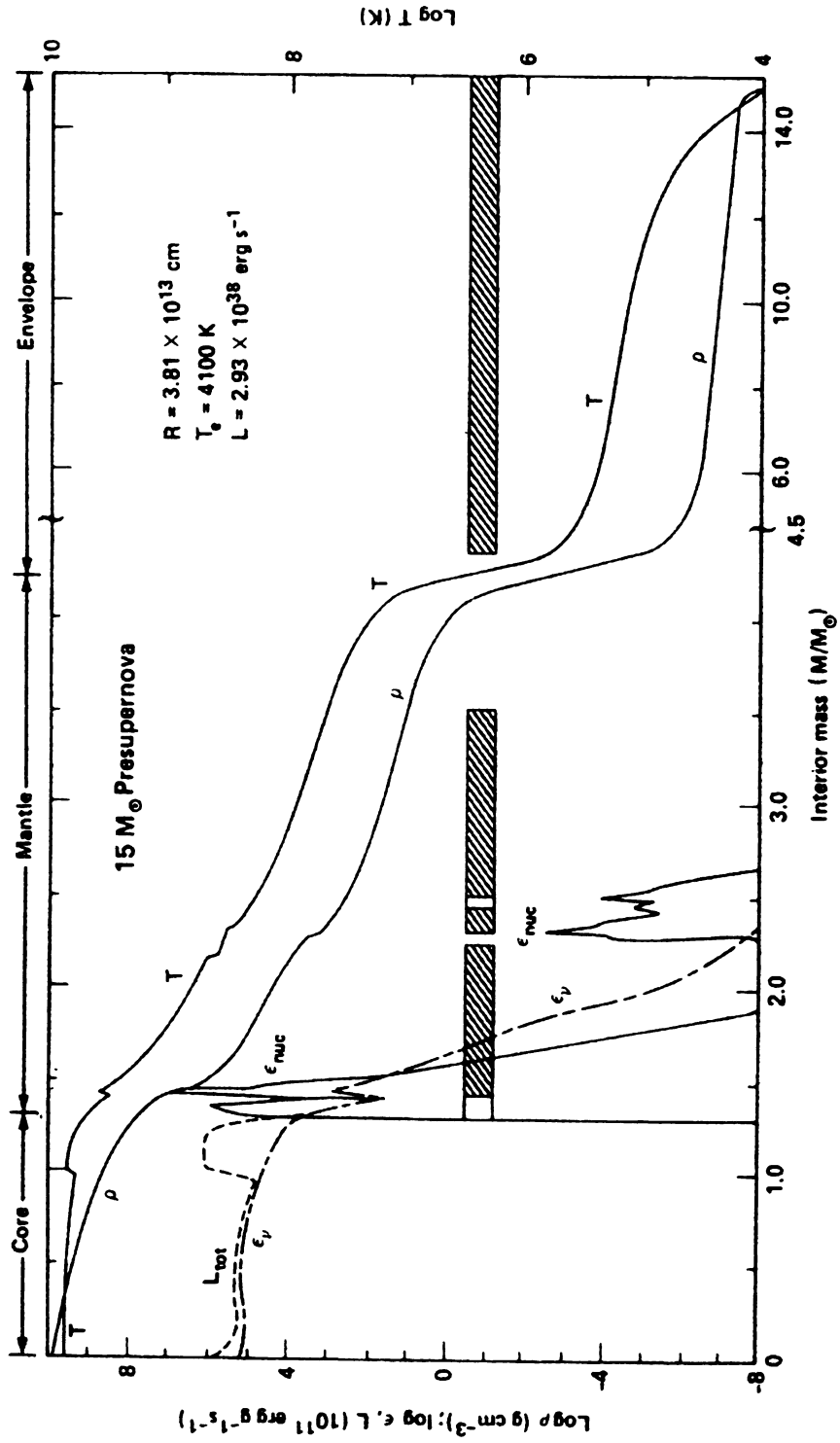


Figure 2.1: Structure of a $15M_{\odot}$ star at the onset of core collapse (taken from [56])

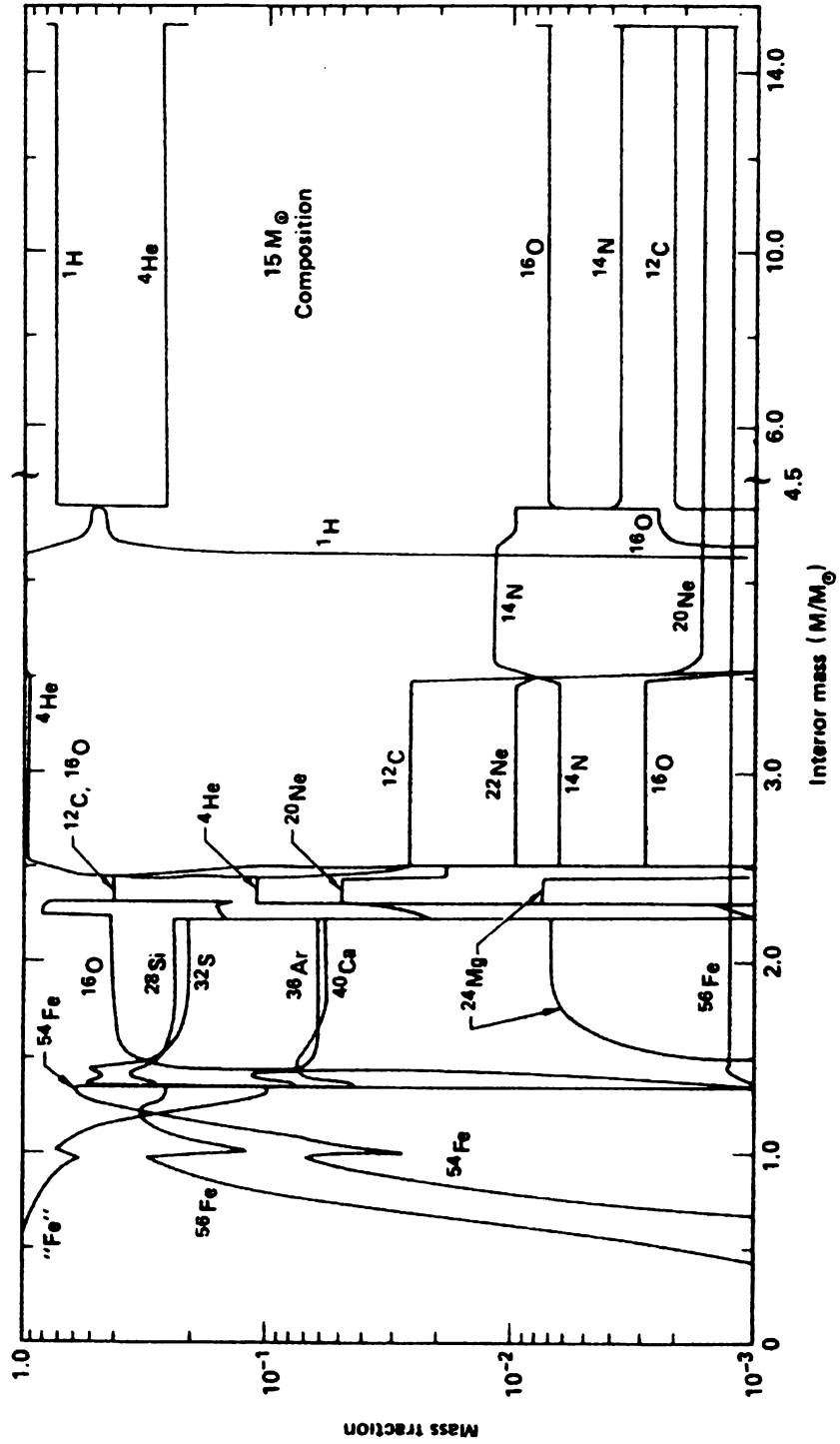


Figure 2.2: Composition of a $15M_{\odot}$ star at the onset of core collapse (taken from [56])

Note that their progenitor was calculated using a one dimensional model assuming spherical symmetry and neglecting possible effects due to rotation, which is the case for almost all stellar evolution simulations (an exception including the effects of rotation is e.g. [20]). Instead of the distance $r = |\vec{x}|$ from the center of the star, the *interior mass* $M(r)$ in units of M_\odot is used on the abscissa. If $\rho(r)$ is the mass density of the star as a function of the distance from its center

$$M(r) := \int_{|\vec{x}| \leq r} d^3x \rho(\vec{x}) = 4\pi \int_0^r dr' r'^2 \rho(r'). \quad (2.4)$$

Note that the *iron core* consisting of all the heavy nuclei ($48 \leq A \leq 65$) formed in the nuclear burning processes mentioned in 2.1.1 (denoted by “Fe” and ^{54}Fe in figure 2.2) whose collapse will get the main focus in this work ends in a relatively abrupt way at an interior mass of approximately $1.35M_\odot$ – a typical value as most stellar evolution simulations for stars of various ZAMS masses greater than $11M_\odot$ yield iron core masses between $\approx 1.1M_\odot$ and $\approx 2.5M_\odot$ (see table 2.1). Apart from the heavy elements (“Fe”) very small amounts of free neutrons, protons and alpha-particles are present in the iron core. Above the iron core, lighter elements are stratified in an “onion-skin” structure.

Before collapse the iron core is almost a perfect $\gamma = \frac{4}{3}$ *polytrope*, i.e. the pressure p inside the core is only a function of the density ρ satisfying

$$p \propto \rho^\gamma \quad (2.5)$$

ZAMS mass	12	15	20	25	35	50	100
Iron core mass	1.31	1.33	1.70	2.05	1.80	2.45	2.3

Table 2.1: Iron core masses of different supernova progenitors (taken from [56], all masses in units of M_\odot)

with $\gamma = \frac{4}{3}$. This is just due to the fact that the pressure comes mainly from the gas of relativistic, degenerate electron gas inside the core for which it is known from statistical mechanics that the pressure is independent of the temperature and follows equation 2.5 with $\gamma = \frac{4}{3}$ (γ is usually called *adiabatic index*, $n := \frac{1}{\gamma-1}$ is called *polytropic index*).

As pressure comes mainly from the electrons it is important to mention that the *electron fraction*⁶ Y_e in the iron core has dropped (mainly by reaction 2.3) to about 0.44 at the onset of collapse. Reaction 2.3 also leads to a drastic decrease of the entropy per baryon in the iron core region where it is roughly just $1k_B$ (where k_B denotes Boltzmann’s constant) directly before collapse compared to a value of about $23k_B$ at the beginning of the main sequence. Entropy is “carried” from the iron core to the envelope by neutrinos where an entropy per baryon of $\approx 40k_B$ is typical directly before collapse (all these entropy values are taken from [56]).

2.2 Explosion Mechanism of Type II Supernovae

In this section the predominating theories for the explosion mechanism of a typical core collapse supernova will be described.

2.2.1 Collapse

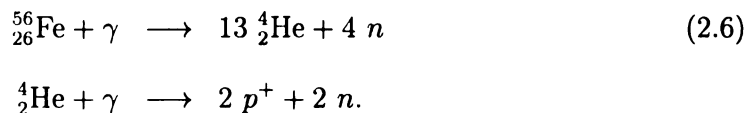
During Si burning the iron core obviously gains mass (as iron is produced in this reaction). This essentially goes on until it reaches a mass that results in gravitational forces which can no longer be supported by the pressure of the present degenerate electron gas. Also note that the pressure at the edge of the iron core is not zero but the outer layers (*mantle* and *envelope*) of the star help “squeezing” its core.

⁶the number of electrons per baryon where “baryon” is used here as a generic term for protons and neutrons

Electron Capture and Photodisintegration

As soon as collapse begins, two instabilities are of importance. Ongoing electron capture reduces the electron fraction in the core thus obviously further reducing the pressure created by the electron gas. The neutrinos created in the electron capture reactions are radiated away almost freely (at least before densities high enough for the occurrence of *neutrino trapping* are reached) which ultimately results in a reduction of entropy in the core – a phenomenon known as *neutrino cooling*. This helps the ongoing collapse even further as a reduction of entropy leads to a reduction of temperature which on its part implies a pressure decrease.

The second instability is due to a process called *photodisintegration* that is possible at the extremely high temperatures now present in the core: heavy nuclei are fragmented to their constituents by extremely high energetic photons, for example (and most importantly) by the following reactions:



These photodisintegration processes require a lot of energy (as they reverse the nuclear “burning” reactions by which the star was powered during its whole life). Therefore the temperature increase due to the increase of density in the core during its collapse is intensely weakened resulting again in a pressure decrement: gravity can no longer be compensated by pressure. In stars with ZAMS mass greater than $\approx 20M_{\odot}$ photodisintegration is considered to be the dominant cause of collapse while for lighter stars electron capture dominates.

Inner Core, Outer Core, and Neutrino Trapping

Once core collapse has begun, things develop very rapidly. The iron core matter falls almost (never quite though) at free-fall velocity towards the center of the star. Hence, the time scale for collapse is merely $\approx 100\text{ms}$ what justifies the assumption of an approximately adiabatic process.

Two regions in the iron core must be separated: the *inner core* and the *outer core*. In the inner core, the infall velocity of the matter is proportional to the distance from the center at any given time which obviously causes all the matter in the inner core to finally arrive at the center of the star simultaneously. The collapse of the inner core is homologous in the sense that the radial distributions of all important quantities (like density, temperature, electron fraction etc.) remain similar to themselves during collapse – just the respective scales change. A typical mass for the inner core is 0.6 to $0.8M_{\odot}$, i.e. the iron core is almost equally split [56]. The inner core ends at the distance where the infall velocity of the matter exceeds the local sound velocity. Figure 2.3 illustrates this showing the infall velocity of the matter and the local sound velocity as a function of the radius approximately one millisecond before core bounce.

Beyond that (obviously time-dependent) distance the outer core falls towards the center at supersonic velocities. It has decoupled from the inner core and arrives at the center of the star later.

Contrary to what was believed before 1979 when the large heat capacities of excitations of heavy nuclei were found which cause the matter to remain relatively cool, this collapse does not stop before nuclear density is reached. It actually goes on till the nuclei touch each other and even beyond that: it is believed that roughly three times the density of isospin symmetric nuclear matter (i.e. $3 \times 2.4 \times 10^{14} \frac{\text{g}}{\text{cm}^3} \approx$

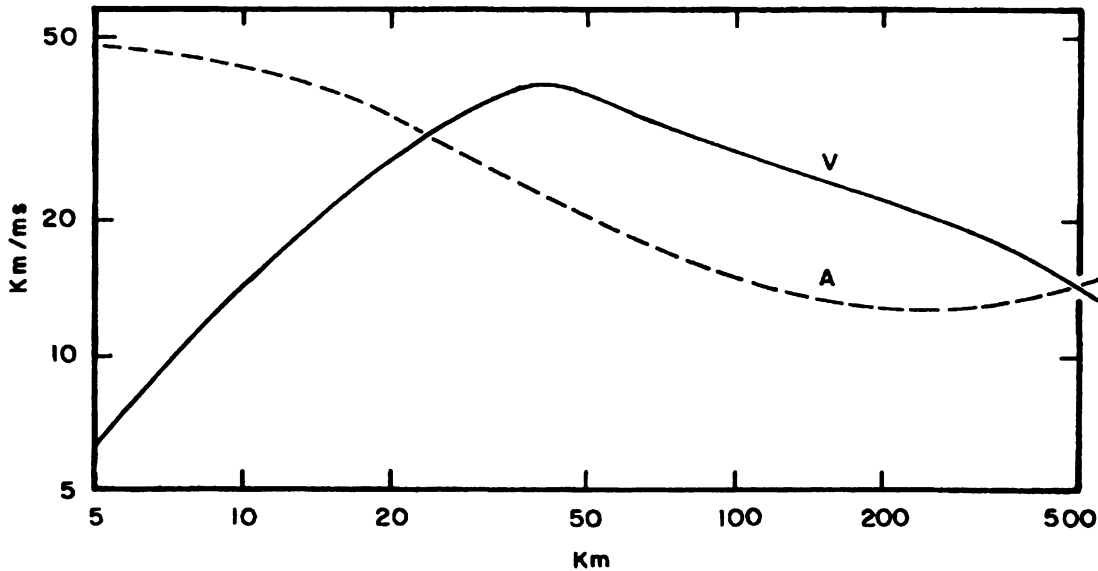


Figure 2.3: Infall velocity of the core matter V and local sound velocity A approximately one millisecond before core bounce (taken from [4] who used the results of [1] to create this plot).

$7 \times 10^{14} \frac{\text{g}}{\text{cm}^3}$) is reached at maximum compression of the core⁷ [4, 3]. Note that the matter is not isospin symmetric in the present situation: significant amounts of protons have been converted to neutrons by electron capture.

However, due to a phenomenon called *neutrino trapping* the matter is not as asymmetric as one might expect at first. Neutrino trapping occurs at densities higher than $\approx 10^{11} \frac{\text{g}}{\text{cm}^3}$. As suggested by its name, it means that neutrinos can no longer escape the core freely but are trapped in there since at these densities elastic scattering of the neutrinos by the nuclei becomes relevant. The mean free path for the neutrinos is so small now that they can only diffuse through this high density region. For the density mentioned above the time scale for the diffusion of neutrinos out of the core clearly exceeds the collapse time scale meaning that the neutrinos cannot get out of the core fast enough – they are trapped.

⁷This value is just the result of most numerical simulations.

Note that because of the very low neutrino mass the presence of the trapped neutrinos hardly affects the nuclei at all in the sense that their pressure contribution is negligible. However, they are able to heat the matter by neutrino-electron scattering events. The cross section for these is (under the given conditions) roughly two orders of magnitude smaller than that for neutrino scattering by heavy nuclei but the electrons are so highly degenerate that they can virtually only gain energy in these events (hence the matter can only be heated).

As electron capture results in the production of neutrinos and these cannot escape anymore, their chemical potential rises rapidly thus obstructing further electron capture. In fact reaction 2.3 occurs in both directions until (dynamic) equilibrium is accomplished:



Note that there are also positrons present which enable the reaction



also contributing to the neutrino production. After that, the total *lepton fraction*⁸ Y_L remains essentially constant (till *core bounce*, the end of collapse) at a value $Y_L \approx 0.36$ (according to [2]). A realistic value for the electron fraction in the core's center at that time is $Y_e \approx 0.3$ (according to [13]).

2.2.2 Core Bounce

As soon as a density greater than (the isospin dependent) nuclear matter saturation density is accomplished, the strong interaction between the nuclei becomes repulsive as a consequence of the Pauli principle for neutrons (which are fermions). As mentioned above, at a density of typically $\approx 7 \times 10^{14} \frac{\text{g}}{\text{cm}^3}$ this repulsion becomes so

⁸the number of leptons (here mainly electrons and neutrinos) per baryon

strong that the matter suddenly stiffens, the collapse is halted, and the inner core rebounds – a little bit like a spring that is first compressed and then released. In doing so, shockwaves are sent out to the infalling outer core. This event is known as *core bounce*.

Theoretical predictions of the maximum density reached at bounce and its vigorousness (i.e. the energy of the created shockwave) are certainly dependent on factors like the inner core mass, temperature, electron fraction and others but most important is the *equation of state* (EOS) of nuclear matter above nuclear density. The nuclear matter EOS in the density and temperature region present at core bounce is still unknown since it is extremely difficult to mimic these conditions experimentally. However, an enormous theoretical effort has been invested in finding the correct EOS and several theoretical predictions exist (see e.g. [3, 28, 43, 41, 39, 30]). Numerical simulations have shown that a “softer” nuclear EOS results in the achievement of higher densities at bounce and more vigorous shock waves in comparison to those computed using a “stiffer” EOS – a somewhat intuitive result.

2.2.3 Prompt Shock Mechanism

During collapse an enormous amount of gravitational energy is released. The main question is by which mechanism even a small fraction of this energy can be coupled to the mantle and the envelope of the star in order to eject them. The gravitational energy released is so immense that the coupling of just $\approx 1\%$ of it would be ample to eject the outer layers of the star and explain the observed supernova explosion energies of $\approx 10^{51} \text{erg} =: 1 \text{foe}$.

An appealing coupling mechanism, called the *prompt shock mechanism* was favored till the mid 1980s: after core bounce the created shockwave moves outward through the infalling matter of the outer core. Analytical arguments and numerical

simulations suggest that the energy of this shockwave is ≈ 4 to 7×10^{51} erg [26] – completely sufficient to power the explosion. So, after this shockwave has reached the outer layers of the star, these get enough kinetic energy to escape, are ejected, and the star explodes.

Unfortunately, things turned out to be not quite this simple: the problem is that the shock loses enormous amounts of energy while it beats its way through the infalling matter of the outer core. The densities and temperatures in the shock region are so high that again electron capture resulting in neutrino losses (at least when these can diffuse away fast enough) and photodisintegration occur massively. Large amounts of heavy nuclei are broken up completely to free nucleons. This costs gigantic amounts of energy ($\approx 1.5 \times 10^{51}$ erg for every $0.1M_{\odot}$ thus converted) that is taken from the shock energy. Consequently, if the outer core is sufficiently large, the shock eventually completely stalls (approximately at a distance of 100 to 300km from the center) and never even reaches the outer layers – the explosion has failed. Note that possible energy losses of the shock wave in the outer layers of the star are a lot smaller than those suffered inside the core because densities and temperatures are too low there for photodisintegration to occur massively. This is why a successful explosion can be anticipated as soon as the shock wave managed to leave the iron core.

The success of the prompt shock mechanism depends crucially on the iron core mass: only if the outer core is small enough (that is essentially the case if the total core mass is small enough) can the prompt explosion work. Numerical simulations indicate that prompt explosions can only occur in stars with iron core masses smaller than $\approx 1.25M_{\odot}$. Stellar structure calculations indicate that there are stars with such light iron cores. The majority of type II progenitors however has iron cores too massive for a successful prompt explosion.

The prompt mechanism is more likely to succeed if a “softer” nuclear EOS is used as it results in greater shock energies.

2.2.4 Delayed Shock Mechanism

So what happens once the shock has stalled? There is still an enormous amount of energy in the core, mainly in the form of thermal excitations and neutrino and electron chemical potentials. Due to large number density gradients the neutrinos diffuse outward. As they reach regions with lower densities their mean free path becomes larger and larger until they can finally radiate away almost freely.

Consequently, the whole matter above the *neutrino sphere*⁹ (located at $\approx 40\text{km}$ from the center) is bathed in a very intense neutrino flux. This can actually help to revive the stalled shock that, by now, has turned into a so-called *accretion shock* (caused by the infalling outer core matter) in the following way (called the *delayed shock mechanism*): the shocked matter above the neutrino sphere now contains a lot of free nucleons which can absorb some of the neutrinos (the cross section for these processes is small but not negligible). This ultimately leads to the heating and expansion of the matter (*neutrino heating*) so that the shock can resume its way outward. It shall be mentioned that neutrinos can also be absorbed by nucleons inside nuclei but (for reasons that will not be discussed here in more detail) are re-emitted shortly afterwards so that these events do not help in reviving the shock.

⁹the distance at which the optical depth for neutrinos is 1. The neutrinos can be considered to radiate away freely approximately from there. More precisely, all this is dependent on the energy of the neutrinos.

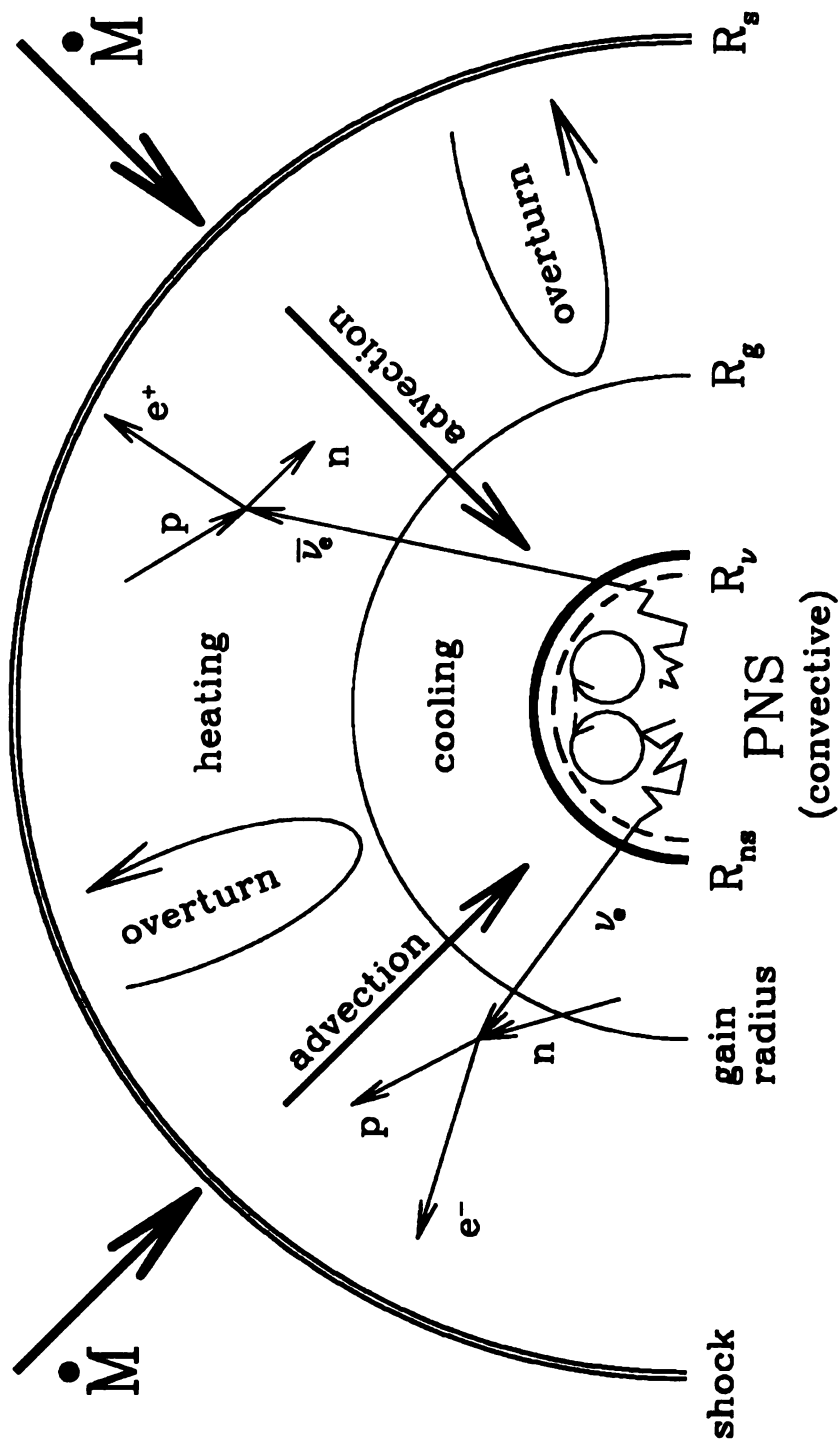


Figure 2.4: Shock revival by neutrino heating after the shock wave has stalled (taken from [24]). The symbols are explained in the text.

Figure 2.4 schematically shows the situation after the shock wave has stalled: the *gain radius* is the distance from the center at which neutrino cooling is dominated by neutrino heating (i.e. more neutrinos are captured than emitted), *overturn* means essentially the same as convection, \dot{M} denotes the infalling mass of the outer layers, R_{ns} is the radius of the *proto-neutron star*¹⁰ (“PNS”), and R_ν denotes the neutrino sphere.

This delayed mechanism was first suggested by Wilson in 1985 [53]. Today it is the commonly accepted theory for core collapse supernova explosions. Still, there is a lot of trouble with it: it depends crucially on the cross sections for neutrino capture on nucleons, the neutrino production rates, the details of the neutrino transport, and other factors many of which are not known with great certainty. In particular, the simulation of neutrino transport in the “gray” region, i.e. at densities where neutrinos are neither trapped nor able to freely radiate away is most problematic. Depending on these input parameters, simulations of the delayed mechanism (performed by several different groups) yielded successful explosions even for stars with very massive iron cores while in other cases the explosions “fizzled” again. A more detailed overview of these simulations will be given in 2.3. A very good summary of the delayed mechanism and the skepticism about its correctness was recently done by Janka [24].

Convection

As just mentioned, the delayed mechanism is most likely not a completely satisfactory explanation for supernova explosions as it has lead to failures in several (mostly one dimensional) simulations that were performed since Wilson suggested it. That is why the quest for a reliable explosion mechanism went on.

Given the more powerful computers that became available in the 1990s, it was

¹⁰the innermost part of the core that is believed to form a neutron star later

an obvious approach to go to higher dimensions: several two dimensional simulations have been carried out (see for example [23, 18, 22]) in which especially the impact of *convection* has been studied. Convection is going to occur for sure in the region between the shock and the neutrino sphere where the shock leaves behind an entropy profile that is instable to convection. This convection takes place very rapidly: large amounts of matter move up and down between the hot proto-neutron star and the colder outer regions – a little bit similar to the convection occurring when cold water in a pot is put on a hot plate, yet on entirely different scales. A detailed discussion of convection is clearly beyond the scope of this work. It shall be mentioned, however, that it is widely believed today that convection helps the success of the explosion by transporting energy from above the neutrino sphere (where the matter can be heated by neutrino absorption in a relatively easy way) to the outer layers of the star. Thus a larger fraction of the gravitational energy that was released during collapse is made available for the explosion. Two dimensional simulations done by Herant et al. [23, 22] for example resulted in successful explosions while their one dimensional ones, that did not include convection but used the same microphysics otherwise, failed.

Apart from the convection between the neutrino sphere and the stalled shock there is a possibility that convection in the proto-neutron star may help the explosion by intensifying the neutrino luminosities [24].

Still, this is not necessarily the final solution to the supernova problem. In particular, it is conceivable that phenomena that can only occur in three dimensions play a role.

Rotation

It is well known that stars carry angular momentum. The impact of this rotation on the core collapse and the explosion mechanism has been studied relatively little

(note however that studies including possible effects of rotation date back to 1970). On the other hand, in the beginning there were ideas (now considered wrong) that the explosion might actually be powered by the conversion of rotational energy via magnetic fields [29].

As simplifications are almost always a necessity to model real phenomena, it was a most reasonable approach to neglect rotation for the beginning. Another reason why relatively little attention has been given to rotation so far is a lack of computer power that is needed because it can only be simulated sensibly in more than one dimension. Simulating convection and neutrino transport is computationally so expensive that simulations including these effects are still restricted to one and two dimensions nowadays. There is also a lack of rotating progenitor models, so the precise initial conditions for simulations of rotating core collapse are relatively uncertain. Some studies were made, however, that will be presented in 2.3.

Indications have been found by several groups that rotation does not affect the explosion mechanism dramatically (see for example [18, 57]). It appears that explosions are slightly delayed and a little bit weaker if rotation is included.

It must be mentioned, however, that often rotation is mimicked in questionable ways necessary because most simulations are performed in two dimensions. Well known phenomena from earth such as vortices in flowing liquids or tornadoes in the atmosphere could hardly occur in less than three dimensions. We would also like to point out that rotation can become extremely rapid in the late stages of core collapse because the inner core contracts more and more. This enforces higher angular velocities as its angular momentum is essentially conserved which can at smaller radii and equal mass only be assured by faster rotation.

Another argument for studying the impact of rotation are recent observations

of the polarization of the light emitted by supernovae made by Wang and Wheeler [50, 49, 51]. Using a method in which they observe the polarization and the wavelength of the light simultaneously (“*spectropolarimetry*”), they found that the light emitted by type II supernovae (and actually all core collapse supernovae which also include e.g. types Ib and Ic) is significantly polarized while this is not (or not nearly as strong) the case for supernovae not powered by core collapse. They draw the conclusion that there must be large asymmetries in the explosion mechanism to cause this polarization. They also found that the polarization of the light gets greater the closer to the center of the explosion they measure it. This gives even more support to the conclusion that the asymmetry originates in the very heart of the explosion.

Rotation is a very good candidate (and the only obvious one) to explain such deviations from spherical symmetry as it defines a direction – that of the rotation axis.

2.3 Recent Numerical Studies of Core Collapse Supernovae

The separate treatment of previous numerical studies of supernovae is a bit artificial as it was hopefully made clear in the previous sections that the currently predominating theories for the explosion mechanism were chiefly developed by the excessive use of numerical simulations.

This section will mainly be focused on the techniques, assumptions, and simplifications used in specific recent simulations (performed in the last ten years) while the previous sections primarily dealt with their results. Here, this can hardly be done in great detail for which one may refer to the respective citations. This is not meant to be a complete overview of the work that was done in the last decade, we just want to

exemplify different approaches.

It shall be mentioned that there are also some attempts to deal with the supernova problem on a more analytical level (not completely relying on numeric). The works of Burrows and Goshy [9] or the very recent one of Janka [24] are examples for this.

2.3.1 Equation of State

An EOS describing the thermodynamic properties of the matter is a necessary input for all supernova simulations. The purpose of an EOS is to make use of the fact that the star is in (local) thermodynamic equilibrium. Thus only three thermodynamic quantities (in core collapse simulations usually the temperature T , density ρ , and electron fraction Y_e) have to be calculated locally, all other quantities (the most important among these the pressure and the specific internal energy) are given by the EOS.

A great theoretical effort has been invested into finding a correct EOS, especially the correct nuclear part of it (see for example [28, 42, 41, 43, 44]). The theory for this is virtually a whole separate science (like several aforementioned aspects of supernova theory). So we will again just give a brief account of the state of the art in this field.

Electron-Positron EOS

The electron-positron part of the EOS is less problematic than the nuclear part. Its contribution is often just modeled by the EOS for a relativistic degenerate ideal fermion gas, neglecting the Coulomb interaction, for which statistical mechanics predicts the temperature-independent pressure-density relation

$$p(\rho, Y_e) = \frac{3}{4} \left(\frac{\pi}{3} \right)^{\frac{2}{3}} \hbar c \left(\frac{Y_e}{m_B} \right)^{\frac{4}{3}} \rho^{\frac{4}{3}}, \quad (2.9)$$

where m_B is the baryon mass [59]. If an adiabatic change ($\delta Q = 0$) is assumed (which is reasonable at least for the infall phase) and the conversion of particle species is also neglected ($dN_i = 0$, where N_i denotes the number of particles of species i), the first law of thermodynamics reduces to

$$dU = \delta Q - pdV + \sum_i \mu_i dN_i = -pdV \quad (2.10)$$

and an expression for the internal energy density $u_{int}^{(V)} = \frac{U_{int}}{V}$ can easily be derived from equation 2.9 by integration:

$$u_{int}^{(V)}(\rho, Y_e) = \frac{9}{4} \left(\frac{\pi}{3} \right)^{\frac{2}{3}} \hbar c \left(\frac{Y_e}{m_B} \right)^{\frac{4}{3}} \rho^{\frac{4}{3}}. \quad (2.11)$$

However, this is a crude approximation. There is a variety of more sophisticated EsOS for the electron-positron gas which cannot be expressed by such simple analytic expressions as equations 2.9 and 2.11. They are available in the form of data tables or computer programs. Five different ones of these are compared in [47]. Sophistication is achieved by adding effects due to such phenomena like the presence of antiparticles (i.e. positrons), the interaction of the electrons with the ionized nuclei present in the matter, and radiation pressure p_{rad} . The latter is usually done using the well-known blackbody approximation that yields for the radiation pressure:

$$p_{rad} = \frac{4}{3} \frac{\sigma}{c} T^4, \quad (2.12)$$

where σ denotes the Stefan-Boltzmann constant. The consideration of the former effects is more complicated and will not be dealt with here.

In recent simulations, EsOS can be called over 10^9 times during a simulation run. Hence, it is important for the applicability of these more sophisticated EsOS (more precisely: the programs that calculate their output values or interpolate the tabulated data) that they work reasonably fast. This is true for both the electron EOS and the

nuclear EOS. As far as the electron EOS is concerned the so-called *Helmholtz EOS* developed by Timmes and Swesty [48] can be considered a great compromise between execution speed, thermodynamic consistency, and accuracy.

Nuclear EOS

As mentioned in 2.2.3, the nuclear EOS (in particular at densities above the nuclear matter saturation density) is of crucial importance for key features of the explosion mechanism – especially for the vigorousness of core bounce.

Mainly for the purpose of saving CPU time, various extremely simplified analytical EsOS have been proposed by different groups. All these are not rigorously derived from theory but simply mimic essential key features such as the drastic rise in pressure and specific internal energy above nuclear saturation density. For example, Zwerger and Müller [59], Yamada and Sato [57], and Bonazzola and Marck [5] use a simple polytrope EOS (see equation 2.5) imitating the stiffening above nuclear matter density with a large adiabatic index like e.g. $\gamma = 2.5$.

A very similar example is the high density EOS introduced by Baron et al. [3]:

$$p(\rho) = \frac{K_0 \rho_0}{9\gamma} \left[\left(\frac{\rho}{\rho_0} \right)^\gamma - 1 \right], \quad (2.13)$$

where K_0 is the *compression modulus*, ρ_0 the nuclear matter density, and γ a parameter. All these parameters are isospin dependent in their model.

Just as in the case of the electron EOS much more sophisticated EsOS are available in the form of computer programs (e.g. [27, 40]). In their derivation one has to deal with a lot of difficulties: at lower densities during core collapse the matter forms essentially a two phase system – a “gas” phase of free nucleons and alpha particles and a “condensed” phase consisting of heavier nuclei. At densities around one-half ρ_0 a so called “Swiss cheese” phase is formed: the nuclei touch each other uniformly

filling large space regions except for certain holes. Finally, at very high densities, the nuclei are “squeezed” beyond their saturation density. Thus, the nuclear matter goes through phase transitions which does not make things easier. These more sophisticated EsOS also rely on certain parameters (such as the *nuclear incompressibility modulus* or the *nuclear surface tension*) for which no exact values (for the densities and temperatures of relevance here) could be obtained from experiments so far.

The grand masters of these EsOS are Lattimer and Swesty whose EOS (from now on referred to as “LS EOS”) [27] has been used in many recent simulations (see e.g. [18, 31, 34, 32, 23, 33, 7, 37] just to name a few) and can be considered state of the art. It is valid only when the matter is in *nuclear statistical equilibrium* which is achieved at the high temperatures in a supernova core. The assumptions and techniques used in the derivation of the LS EOS are described in great detail in [28].

2.3.2 One Dimensional Simulations

One dimensional simulations in which spherical symmetry is assumed still play an important role in the process of better understanding the supernova explosion mechanism. They are still the only way to simulate the underlying microphysical processes in great detail. However, while one dimensional simulations became more and more sophisticated over the years, most of them failed to produce explosions. It thus became clear that phenomena that can only appear in higher dimensions, such as convection or rotation, are very likely to play an important role in the explosion mechanism.

Mezzacappa, Liebendörfer et al. [34, 31]

A very recent example of such a “fizzle” are the state of the art one dimensional simulations performed by Mezzacappa, Liebendörfer et al. [34, 31]. They follow the

whole explosion process of a $13M_{\odot}$ star using Newtonian [34] and general relativistic [31] gravity, explicitly not including convection (that is sometimes effectively included in one dimensional models) and paying special attention to the details of neutrino transport in order to make sure that the failure of previous simulations is not due to approximations in this area. They use a method called *multigroup flux-limited diffusion* including effects due to different neutrino flavors and energies which can be important because the cross sections for the interactions of neutrinos with the matter strongly depend on these. The LS EOS (see 2.3.1) is used in their hydrodynamic simulation that uses an adaptive grid. The latter means that the size of the cells for which they calculate quantities like mass density, temperature, or neutrino density is variable. This is used to obtain a preferably higher resolution in regions where one or more of these quantities change rapidly by going to smaller cell sizes there.

As mentioned above, their main result is that they do not obtain an explosion thus giving further support to the importance of higher dimensional effects.

Herant et al. [23]

Most of the recent work has been done in two dimensions. Quite often, however, results from one dimensional simulations are used to simplify the two dimensional ones by mapping data obtained from the former ones on the latter ones. One dimensional simulations are also performed in order to compare their results to those achieved with two dimensional ones.

An example for this is the simulation by Herant et al. [23]. They use a second order Runge-Kutta hydrodynamic grid-based code with adaptive cell size without including effects of convection in their one dimensional model. The way they model neutrino transport is strongly simplified: cells in their grid are either labeled optically “thick” or “thin” depending on the density currently available. In the “thick” regions

neutrino transport is approximated by diffusion (see below) and in the “thin” regions a so-called “central light bulb approximation” is used. This means that the matter in these regions is bathed in a neutrino flux originating from the center of the star, the magnitude of which is essentially determined by neutrino production rates in the “thin” regions and the rate at which neutrinos diffuse from “thick” to “thin” regions. They use the neutrino production rates calculated by Takahashi et al. [45].

For the diffusion they use a so-called “flux-limited” method. This means that

$$\frac{dn}{dt} = \min\left(\frac{c}{3}|\nabla n|, \nabla \cdot D\nabla n\right), \quad (2.14)$$

where n denotes the neutrino concentration in a cell and D is a diffusion coefficient. This obviously sets an upper bound for the flux thus limiting it. If neutrino concentrations are very high, diffusion may be obstructed by their fermi properties which disallow them to share identical quantum states. This effect is also included in their model of neutrino transport.

Their one dimensional model does not explode.

Burrows et al. [10]

Finally, the work of Burrows et al. [10] shall be mentioned. They also ran both one and two dimensional simulations using monopole Newtonian gravity in a hydrodynamic grid-based code in which (contrary to the previously mentioned works where the grid is fixed to the mass elements) the cells are fixed in space, which forces them to use a higher cell resolution the closer they get to the center in order to get an appropriate resolution of core bounce and shockwave formation. Neutrino transport is approximated by assuming diffusion. Three different neutrino species (ν_e , $\bar{\nu}_e$ and ν_μ where ν_μ represents all remaining neutrino species) are treated separately. The neutrino absorption and emission rates in the “gray” (semi translucent) regions are

approximated by relatively simple analytic expressions. The EOS they use is a slightly modified version of the LS EOS.

The results of their simulations will be discussed in 2.3.3.

2.3.3 Two Dimensional Simulations

Fryer and Heger [18]

The work of Fryer and Heger [18] is one of the most interesting with respect to our own simulation as they study the core collapse of the rotating progenitor calculated by Heger [20] and try to find possible explanations for the polarization measurements by Wang and Wheeler [50, 49, 51].

The angular velocity profile they use is shown in fig. 2.5. The discontinuities of the angular velocity are located at the boundaries of well-defined regions in the progenitor model in which angular velocity is equilibrated by convection. They use a numerical technique called *smooth particle hydrodynamics* [19, 21, 22, 23, 12] an essential feature of which is that the star's mass is represented by discrete particles. Neutrino transport is treated in the same way as by Herant et al. (see 2.3.2). A patchwork of different EsOS for the different density regions is used, among these the LS EOS for high densities. Gravity is treated using general relativity and assuming spherical symmetry. Cylindrical symmetry around the rotation axis but no equatorial symmetry is assumed for the rest (the former is a requirement to reduce the number of dimensions from three to two). Angular momentum conservation is enforced for each particle separately.

They find that rotation limits the convection overall and restricts it to the polar regions. This ultimately delays the explosion and also weakens it. Further, asymmetries due to rotation are apparent: the mean velocity of the matter in the polar

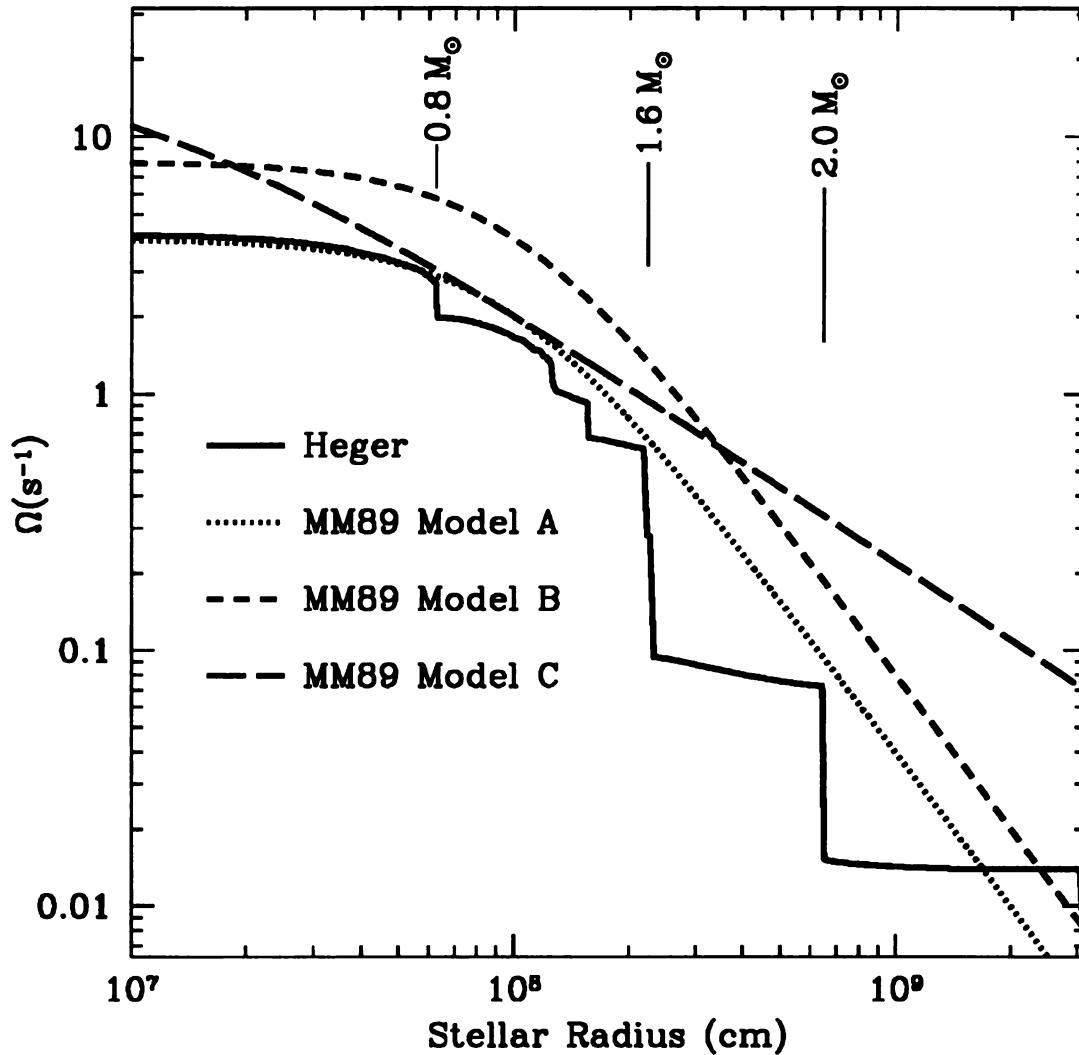


Figure 2.5: Angular velocity profile used by Fryer and Heger [18] compared to that of different models used by Mönchmeyer and Müller (labeled “MM89”) [35]

regions is larger by a factor of 2 compared to the equatorial region. They speculate that these asymmetries might explain the polarization of the emitted light.

Herant et al. [23]

The two dimensional model by Herant et al. [23] is very similar to the one of Fryer and Heger just described (the latter can actually be considered an advancement of the

former). Both use essentially the same numerical technique. Herant et al., however, approximate gravity using the Newtonian limit assuming spherical symmetry. More importantly, the effects of rotation are not studied in their work. They use a (non-rotating) progenitor calculated by Woosley and Weaver.

Their main result is that convection above the proto-neutron star is the key to obtaining an efficient and robust explosion mechanism. Their model turned out to be insensitive to microphysical details like neutrino cross sections (for which they used a variety of values). They compare the star in the phase after the shock has stalled to a thermodynamic engine in which the convective matter transports energy from the hot reservoir (the proto-neutron star) to the cold reservoir (the star's envelope) thus powering the explosion. Due to the large temperature difference between these two reservoirs the efficiency of this "Carnot engine" is high. This leads to a self-regulated explosion mechanism: the envelope is just heated enough to explode, which is a possible explanation for the fact that all observed supernova explosion energies are approximately between 1 and 2×10^{51} erg. However, they admit that there are certain subtleties in simulating convection in two dimensions and call the possibility that rotation may be important "a very real one".

Burrows et al. [10]

Most of what was said (e.g. the way neutrino transport is treated) in 2.3.2 about the one dimensional model of Burrows et al. remains valid for their two dimensional one. Actually, they also use the one dimensional simulation to simplify the two dimensional one by mapping data from the former on the latter. For example, they just use the information for the inner core obtained from the one dimensional model in the two dimensional one.

Just like the two groups just mentioned, they also find that convection is helpful

to obtain a successful explosion. However, unlike Herant et al. they draw the conclusion that the mechanism still significantly depends on the way neutrino transport is modeled.

Yamada and Sato [57]

Yamada and Sato [57] follow a completely different approach than all the aforementioned works. They do not deal with neutrino transport, a realistic EOS, convection, or other microphysical details but study the impact of rotation in a simple two dimensional model. Essentially, a polytrope EOS (equation 2.5) is used in which they make the adiabatic index γ a function of density: at low densities ($\rho < 4.0 \times 10^9 \frac{g}{cm^3}$) and at densities at which neutrino trapping occurs but the nuclear forces can still be neglected ($10^{12} \frac{g}{cm^3} < \rho < 2.8 \times 10^{14} \frac{g}{cm^3}$), $\gamma = \frac{4}{3}$ for the relativistic degenerate electron gas is used. At intermediate densities ($4 \times 10^9 \frac{g}{cm^3} < \rho < 10^{12} \frac{g}{cm^3}$) the pressure deficit due to electron capture and photodisintegration is mimicked by a drop of the adiabatic index below $\frac{4}{3}$ (e.g. $\gamma = 1.3$). Finally, at densities greater than nuclear ($\rho > 2.8 \times 10^{14} \frac{g}{cm^3}$) the stiffening of the nuclear matter is modeled by $\gamma = 2.5$. Their EOS also contains an ideal-gas-like temperature dependence.

The main purpose of their model is to find deviations from spherical symmetry in a two dimensional hydrodynamic simulation . They study the impact of rotation on the collapse of the iron core by artificially adding angular velocity to a (non-rotating) $15M_{\odot}$ progenitor calculated by Woosley and Weaver.

According to their results, the maximum density at core bounce, the explosion energy, and the ejected mass all decrease monotonically with increasing angular momentum of the core.

Zwerger and Müller [59]

The approach of Zwerger and Müller [59] is very similar to that of Yamada and Sato as far as the EOS and the treatment (or rather “non-treatment”) of neutrino transport is concerned. However, as initial models Zwerger and Müller use $\gamma = \frac{4}{3}$ - *polytropes*¹¹ in rotational equilibrium whose collapse is initiated by suddenly decreasing the adiabatic index below $\frac{4}{3}$ (to values between 1.325 and 1.28).

Their main focus is on the gravitational wave signal of the supernova event, a topic also studied by several other groups (e.g. [5, 58, 38]) that will not be discussed here any further. An interesting result is that in some of their rapidly rotating models core bounce occurs due to huge centrifugal forces even before nuclear density is reached.

This work must be seen in the context of a series of similar studies (among these [38, 15, 25]) performed by different people at the Max Planck Institut für Astrophysik in Garching, Germany.

2.3.4 Three Dimensional Simulations

Very few core collapse simulations have been done in three dimensions. The ones we know of mainly deal with the gravitational wave signal [5, 38] emitted by the explosion and use the simplified polytropic EOS described above (mimicking key features by manipulating parameters in one way or another). The effects of rotation are included in these models. To our best knowledge, there is currently no three dimensional simulation of a core collapse supernova in which a realistic EOS, a realistic progenitor model, or even neutrino transport (to say nothing of convection) is included. This situation is naturally due to the immense computational requirements of such a simulation.

¹¹equilibrium solutions of the polytrope EOS 2.5

It shall be mentioned, however, that Chris Fryer is currently working on such a model [16, 17] but has not published any results yet.

Chapter 3

Our Three Dimensional Simulation

As pointed out in 2.3.4 there were hardly any core collapse simulations in three dimensions so far. The basic idea of our own simulation is to simplify the underlying microphysics, mainly by using input from one and two dimensional simulations, in order to be able to follow the core's dynamics in three dimensions during collapse and bounce. The main focus is put on the impact of rotation during collapse. Our hope was to thus find deviations from spherical symmetry that are so significant that they may deliver alternatives to the currently favored complicated convection-driven explosion mechanism. After all, it is well known that much weaker pressure gradients (than those present during core collapse) and slower rotation can lead to very major phenomena like tornadoes in the atmosphere or vortices in flowing liquids (yet on a much larger time scale).

We think that the effects of rotation can truly only be studied in three dimensions because adding rotation in a two dimensional model is always somewhat artificial. A three dimensional approach is further motivated by the aforementioned polarization observations by Wang and Wheeler [50, 49, 51] for which no satisfactory explanation could be obtained from two dimensional simulations so far. The dependence of many one and two dimensional models on such (uncertain) parameters as the viscosity of

the matter and neutrino cross sections also raised our hope that there might be a solution to the supernova problem that is largely independent of these details.

3.1 Test Particle Method

The method we use is similar to the so called *test particle method* or *pseudoparticle method* that has been used extensively in nuclear physics [54]. Its original purpose is to approximate the time-dependent Hartree-Fock equations in a way that allows classical interpretation.

While in nuclear physics usually several thousand test particles represent one nucleon, things are vice versa in our model for the collapse of an iron core with a mass around $1.3M_\odot$: (assuming that we use 10^6 test particles) one test particle represents a mass of $1.3 \times 10^{-6} M_\odot \approx 2.5857 \times 10^{24} \text{kg}$ – just a little less than the mass of the entire earth!

In our model all N_{tp} particles have the same mass $M_{IC}/N_{tp} =: m_{tp}$, where M_{IC} denotes the mass of the whole iron core. For each individual particle, position \vec{r}_j and momentum \vec{p}_j are tracked (as classical three vectors). The equations of motion for the test particles are the relativistic versions of the Newtonian ones known from classical mechanics:

$$\dot{\vec{r}}_j = \frac{d\vec{r}_j}{dt} = \frac{\vec{p}_j}{\sqrt{m_{tp}^2 + (\frac{\vec{p}_j}{c})^2}} \quad (3.1)$$

$$\begin{aligned} \dot{\vec{p}}_j &= \frac{d\vec{p}_j}{dt} = \vec{F}_{G,j}(\vec{r}_1, \dots, \vec{r}_{N_{tp}}) + \vec{F}_{EOS,j}(\vec{r}_j) \\ j &= 1, \dots, N_{tp}, \end{aligned} \quad (3.2)$$

where $\vec{F}_{G,j}$ denotes the force on particle j due to gravity and $\vec{F}_{EOS,j}$ the force due to the equation of state. Gravity is modeled assuming spherical symmetry by using

the Newtonian monopole approximation. This means that $\vec{F}_{G,j}$ is the gravitational force test particle j would experience if the mass of all test particles located closer to the center of the iron core (which is identical to the origin of the coordinates) was combined in a point mass located at the center:

$$\vec{F}_{G,j} = -G \frac{m_{tp}^2 \# \left\{ i \in \{1, \dots, N_{tp}\} : |\vec{r}_i| < |\vec{r}_j| \right\}}{|\vec{r}_j|^3} \vec{r}_j. \quad (3.3)$$

This approximation is only good as long as the deviations from spherical symmetry (of the mass distribution) are sufficiently small. The calculation of $\vec{F}_{EOS,j}$ is a bit more complicated and will be explained in 3.3.4.

3.2 Grid

In order to be able to locally define thermodynamic quantities, most notably the local mass density, we introduced a spherical coordinate grid.

3.2.1 Cells and Boundaries

The boundaries between the cells of this grid are defined by the surfaces of constant r , constant ϕ , and constant θ in spherical coordinates (r, ϕ, θ) using the standard notation ($r = |\vec{x}|$, $\phi =$ azimuth angle, $\theta =$ polar angle). For the ϕ coordinate the boundaries are located at

$$0, 1 \times \frac{2\pi}{N_\phi}, 2 \times \frac{2\pi}{N_\phi}, \dots, (N_\phi - 1) \times \frac{2\pi}{N_\phi},$$

where N_ϕ denotes the total number of boundaries (for the ϕ coordinate). For the θ coordinate the boundaries are chosen so that the difference between $\cos \theta$ of two arbitrary neighboring boundaries is constant, i.e.

$$|\cos \theta_1 - \cos \theta_2| = \text{const.}$$

if θ_1, θ_2 are neighboring boundaries (for the θ coordinate). This is made so to make the volume of a grid cell independent of θ . Finally, for the r coordinate the boundaries are usually chosen equidistant so there are boundaries at

$$1 \times \frac{R}{N_r}, 2 \times \frac{R}{N_r}, \dots, N_r \times \frac{R}{N_r} = R,$$

where R denotes the radial location of the edge of the spherical grid and N_r the number of boundaries for the r coordinate. Note that this way the volume of the cells depends only on r (not on ϕ or θ) and increases with increasing distance from the center.

Radial Cell Width Feature

However, an option which enables us to use non-equidistant r -boundaries is included in our code: a monotonic growing function $f(y) : [0 : 1] \rightarrow [0 : 1]$, $y \mapsto f(y)$ satisfying $f(0) = 0$ and $f(1) = 1$ can be defined. The boundaries are then located wherever $(f(\frac{r}{R}) \times N_r) \in \mathbb{N}$. For example, $f(y) = y^{0.9}$ yields a higher grid resolution near the center. $f(y) = \text{id}(y) = y$ gives equidistant r -boundaries as before.

3.2.2 Grid Coordinates

Recapitulating, we have a spherical grid consisting of $N_{\text{cells}} := N_r \times N_\phi \times N_{\cos \theta}$ cells. These are conveniently labeled by three integers $n_r \in \{1, \dots, N_r\}$, $n_\phi \in \{1, \dots, N_\phi\}$, $n_{\cos \theta} \in \{1, \dots, N_{\cos \theta}\}$ where increasing n_X corresponds to increasing X for $X \in \{r, \phi, \cos \theta\}$ ¹. These three integers $(n_r, n_\phi, n_{\cos \theta})$ will be referred to as *grid coordinates* from now on.

Given a point inside the grid in spherical coordinates (r, ϕ, θ) , we can find its grid

¹In the C++ code, these integers run from 0 to $N_X - 1$.

coordinates using the following trivial relations:

$$n_r = \left[\frac{r}{R} N_r \right] + 1 \quad (3.4)$$

$$n_\phi = \left[\frac{\phi}{2\pi} N_\phi \right] + 1 \quad (3.5)$$

$$n_{\cos \theta} = \left[\frac{\cos \theta + 1}{2} N_{\cos \theta} \right] + 1, \quad (3.6)$$

where the squared brackets $[Y]$ denote truncation of the quantity Y . Note that increasing $n_{\cos \theta}$ corresponds to increasing $\cos \theta$ but to decreasing θ .

Cell Volume

Using this, we can now explicitly give the volume of a grid cell as a function of its grid coordinates:

$$Vol(n_r, n_\phi, n_{\cos \theta}) = Vol(n_r) = \frac{4\pi R^3}{3N_\phi N_{\cos \theta}} \left\{ f^{-1}\left(\frac{n_r}{N_r}\right)^3 - f^{-1}\left(\frac{n_r - 1}{N_r}\right)^3 \right\}, \quad (3.7)$$

or for the most important case $f(y) = f^{-1}(y) = \text{id}(y) = y$:

$$Vol(n_r) = \frac{4\pi R^3}{3N_\phi N_{\cos \theta}} \left\{ \left(\frac{n_r}{N_r}\right)^3 - \left(\frac{n_r - 1}{N_r}\right)^3 \right\}. \quad (3.8)$$

Innermost Cell

The volume of the cells with $n_r = 1$ is usually very small. Numerical problems also arise from the “wedge shape” of these cells. These problems will become more transparent in 3.2.3 and 3.2.4. Anyway, all these cells at the center are treated as one (spherical) cell.

Grid Scaling

During a core collapse simulation an enormous change in the length scale occurs: the inner iron core contracts roughly by a factor of 10^2 to 10^3 . Even if N_r is chosen very

large² (e.g. $N_r = 500$) the inner core at core bounce would still be almost completely in the innermost cell if the grid were fixed in space (as long as $f(y) = \text{id}(y)$ is chosen). This way an appropriate resolution of core bounce and other phenomena would be destroyed. That is why we chose to scale down the grid simultaneously with the core. More precisely the size of the grid (which is given by R , the distance of its outer edge from the center) is modified during collapse as follows: in every time step, the distance of the $\frac{N_{tp}}{100}$ th outermost test particle is determined and multiplied by a factor slightly larger than 1 (e.g. 1.12). The advantage of this apparently odd procedure is that it proved to be very functional: almost all test particles remain in the grid during collapse and fill a reasonable portion of it, and if a few particles are “lost” (what this means more precisely and how it can happen will be illuminated in 3.4) the simulation is not ruined (which can be the case if just the outermost test particle is used to define the size of the grid).

3.2.3 Calculation of Densities

A mass density $\rho(n_r, n_\phi, n_{\cos\theta})$ can be calculated for each cell of the grid using the following evident way. The number of test particles in the cell $N_{tp}(n_r, n_\phi, n_{\cos\theta})$ is counted, multiplied by the mass of a test particle m_{tp} , and divided by the cell volume $Vol(n_r)$:

$$\rho(n_r, n_\phi, n_{\cos\theta}) = \frac{N_{tp}(n_r, n_\phi, n_{\cos\theta}) m_{tp}}{Vol(n_r)}. \quad (3.9)$$

In order to minimize errors due to fluctuations and to smooth the density distribution, we decided to use a slightly more sophisticated method in which the test particles’ mass is smeared over the cell it is in and the seven (well-defined) neighboring cells which are located nearest to the test particle. Figure 3.1 illustrates this method for the two dimensional case: first, the cells which are closest to the test particle are

²Note that the total number of cells is clearly limited by the condition $N_{cells} \ll N_{tp}$.

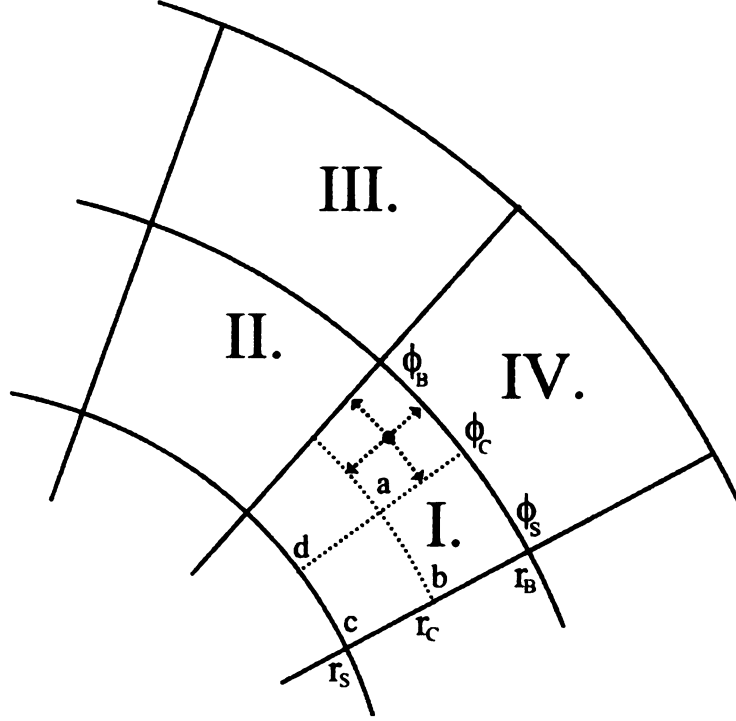


Figure 3.1: Smearing of test particles for density calculation

determined by detecting in which of the sub cells I.a through I.d it is located. Then the number fractions χ of the test particle these cells “get” are calculated as follows:

$$\chi_I = \left(1 - \frac{1}{2} \frac{r - r_C}{r_B - r_C}\right) \left(1 - \frac{1}{2} \frac{\phi - \phi_C}{\phi_B - \phi_C}\right) \quad (3.10)$$

$$\chi_{II} = \left(1 - \frac{1}{2} \frac{r - r_C}{r_B - r_C}\right) \left(\frac{1}{2} \frac{\phi - \phi_C}{\phi_B - \phi_C}\right) \quad (3.11)$$

$$\chi_{III} = \left(\frac{1}{2} \frac{r - r_C}{r_B - r_C}\right) \left(\frac{1}{2} \frac{\phi - \phi_C}{\phi_B - \phi_C}\right) \quad (3.12)$$

$$\chi_{IV} = \left(\frac{1}{2} \frac{r - r_C}{r_B - r_C}\right) \left(1 - \frac{1}{2} \frac{\phi - \phi_C}{\phi_B - \phi_C}\right), \quad (3.13)$$

where r and ϕ are the coordinates of the test particle and r_B, r_C, ϕ_B and ϕ_C denote the coordinates of the boundaries of cell I.a (with $r_B > r_C$ and $\phi_B > \phi_C$). $\sum_{i=I}^{IV} \chi_i = 1$ is ensured. It is obvious how this generalizes to three dimensional spherical coordinates.

The method yields for example that a test particle that is located exactly in

the corner of a cell is uniformly shared among the eight adjacent cells (in the three dimensional case), and belongs completely to the cell it is located in if it is exactly in its center.

The procedure may seem artificial and unjustified because if you visualize the smearing as being caused by giving the test particles a finite size, the technique just described implies that the size of the test particles depends on the distance from the center (as the cell size depends on it). But the use of the grid is only sensible if the grid resolution can be chosen high enough for the variations of the density (and other quantities defined for each cell) in neighboring cells to be small. Thus the smearing is just a technique to smooth possible fluctuations the details of which should not significantly affect our results.

3.2.4 Calculation of Derivatives

To follow the dynamics of core collapse, the calculation of (spacial) derivatives of thermodynamic quantities is necessary. More precisely, this is needed to calculate $\vec{F}_{EOS,j}$ from equation 3.2 which will be explained in 3.3.4.

Let Ω be a thermodynamic quantity defined on the grid (meaning that for each cell $(n_r, n_\phi, n_{\cos\theta})$ a real number $\Omega(n_r, n_\phi, n_{\cos\theta}) \in \mathbb{R}$ is defined). To approximate the gradient $\nabla\Omega(r, \phi, \theta)$, we use a modification of the standard technique for calculating numerical derivatives described in [36] and the well known expression for the gradient in spherical coordinates [6]:

$$\nabla\Omega(r, \phi, \theta) = \frac{\partial\Omega}{\partial r}\Big|_{(r,\phi,\theta)} \vec{e}_r + \frac{1}{r \sin\theta} \frac{\partial\Omega}{\partial\phi}\Big|_{(r,\phi,\theta)} \vec{e}_\phi + (-\sin\theta) \frac{1}{r} \frac{\partial\Omega}{\partial(\cos\theta)}\Big|_{(r,\phi,\theta)} \vec{e}_\theta, \quad (3.14)$$

where $\vec{e}_r, \vec{e}_\phi, \vec{e}_\theta$ denote the orthonormal basis vectors for spherical coordinates. These are dependent on the coordinates (r, ϕ, θ) . What remains to be done is the numerical

definition of

$$\frac{\partial \Omega}{\partial r} \Big|_{(r,\phi,\theta)}, \quad \frac{\partial \Omega}{\partial \phi} \Big|_{(r,\phi,\theta)}, \quad \text{and} \quad \frac{\partial \Omega}{\partial (\cos \theta)} \Big|_{(r,\phi,\theta)}.$$

Linear Interpolation of Derivatives

We will exemplarily describe our technique for this for $\frac{\partial \Omega}{\partial r}$. Let (r, ϕ, θ) be located in the cell with grid coordinates $(n_r, n_\phi, n_{\cos \theta})$. Then two obvious approximations for $\frac{\partial \Omega}{\partial r}$ are

$$\left(\frac{\partial \Omega}{\partial r} \Big|_{(r,\phi,\theta)} \right)_{right} = \frac{\Omega(n_r + 1, n_\phi, n_{\cos \theta}) - \Omega(n_r, n_\phi, n_{\cos \theta})}{R \times \left\{ f^{-1} \left(\frac{n_r + \frac{1}{2}}{N_r} \right) - f^{-1} \left(\frac{n_r - \frac{1}{2}}{N_r} \right) \right\}} \quad (3.15)$$

$$\left(\frac{\partial \Omega}{\partial r} \Big|_{(r,\phi,\theta)} \right)_{left} = \frac{\Omega(n_r, n_\phi, n_{\cos \theta}) - \Omega(n_r - 1, n_\phi, n_{\cos \theta})}{R \times \left\{ f^{-1} \left(\frac{n_r - \frac{1}{2}}{N_r} \right) - f^{-1} \left(\frac{n_r - \frac{3}{2}}{N_r} \right) \right\}}, \quad (3.16)$$

where the denominators are just the distances in the r coordinate between the centers of the neighboring cells ($= \frac{R}{N_r}$ for $f(y) = \text{id}(y)$). We decided to linearly interpolate the derivative between these two values:

$$\frac{\partial \Omega}{\partial r} \Big|_{(r,\phi,\theta)} = \frac{r - r_S}{r_B - r_S} \times \left(\frac{\partial \Omega}{\partial r} \Big|_{(r,\phi,\theta)} \right)_{right} + \frac{r_B - r}{r_B - r_S} \times \left(\frac{\partial \Omega}{\partial r} \Big|_{(r,\phi,\theta)} \right)_{left}, \quad (3.17)$$

where r_S and r_B denote the r -coordinates of the cell boundaries of cell $(n_r, n_\phi, n_{\cos \theta})$ with $r_S \leq r \leq r_B$ (just as in figure 3.1). $\frac{\partial \Omega}{\partial \phi}$ and $\frac{\partial \Omega}{\partial \cos \theta}$ are in principle interpolated the same way. For $\frac{\partial \Omega}{\partial \cos \theta}$ note that

$$\frac{\partial \Omega}{\partial \cos \theta} \Big|_{(r,\phi,\theta)} = \frac{-1}{\sin \theta} \frac{\partial \Omega}{\partial \theta} \Big|_{(r,\phi,\theta)}$$

which was already used in equation 3.14.

Problems at Grid Boundaries

At the boundaries of the grid and the z -axis, boundary conditions for the derivatives have to be fixed. To a certain extent these are always arbitrary and physical reasoning

is needed. Other problems may arise from the fact that test particles can be located outside the grid. We will explain in 3.4 (when the physical meaning of the spatial derivatives will have become clear) how these problems are dealt with.

3.3 Equation of State

Once we have defined the needed thermodynamic quantities on the grid, an appropriate EOS can be used to obtain remaining ones.

3.3.1 Cold Nuclear EOS and Polytrope EOS

As a first approximation to a realistic EOS for supernova matter we used the following well-known parameterization for the energy per baryon of cold (isospin symmetric) nuclear matter [30] which only depends on the density:

$$u_{nuc}(\rho) = a \frac{\rho}{\rho_0} + b \left(\frac{\rho}{\rho_0} \right)^\sigma \quad (3.18)$$

with $\rho_0 = 2.4 \times 10^{14} \frac{\text{g}}{\text{cm}^3}$ = nuclear matter density, $a = -218\text{MeV}$, $b = 164\text{MeV}$, and $\sigma = \frac{4}{3}$. This EOS yields such realistic features as a minimum of the energy per baryon at ρ_0 , a slight attraction for the matter at densities below ρ_0 , and a strong repulsion at densities higher than ρ_0 .

Once an electron fraction is defined (see 3.3.2), the presence of electrons can be modeled by adding equation 2.11 (multiplied by $\frac{m_B}{\rho}$ to get the internal energy per baryon) to equation 3.18:

$$u_{int}(\rho, Y_e) = a \frac{\rho}{\rho_0} + b \left(\frac{\rho}{\rho_0} \right)^\sigma + \frac{9}{4} \left(\frac{\pi}{3} \right)^{\frac{2}{3}} \hbar c Y_e^{\frac{4}{3}} \left(\frac{\rho}{m_B} \right)^{\frac{1}{3}}. \quad (3.19)$$

Note that an electron fraction other than 0.5 implies that an asymmetry energy term should be added to the nuclear part. As the correct shape of this term does not stand firm yet [30] and the isospin asymmetry of the matter remains relatively small (as

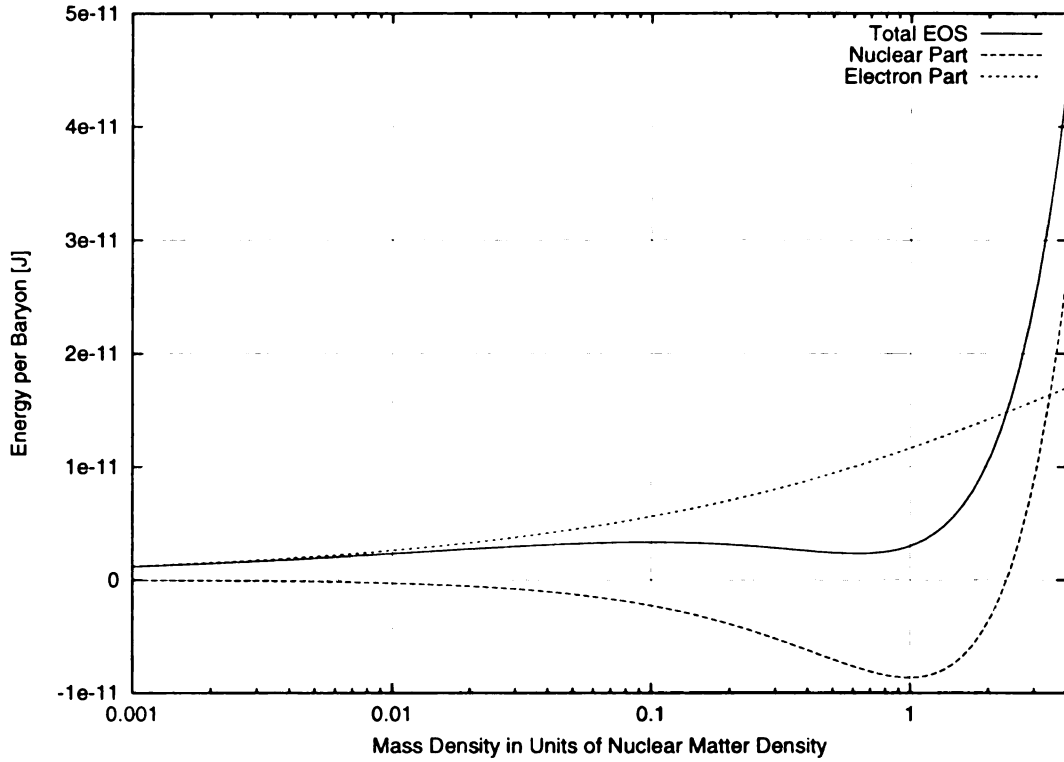


Figure 3.2: Cold nuclear EOS and polytrope EOS for electron gas

described in 2.2.1), it is neglected here. Figure 3.2 shows the different contributions to this EOS (using an electron fraction as defined in 3.3.2).

Equation 3.19 is obviously a crude approximation because (among other things) temperature does not even appear.

3.3.2 Calculation of Thermodynamic Quantities other than Density

More sophisticated EsOS generally need more input than just the density of the matter. The ones that will be used in this work will be described in 3.3.3 and need the temperature T , electron fraction Y_e , and the composition of the matter (the latter is given by the average charge and mass numbers of the present nuclei). A realistic calculation of these quantities is not nearly as simple as that of the density: for the

electron fraction, for example, one has to use proper electron capture rates (such as the ones calculated in [45]) and once the produced neutrinos are trapped, neutrino diffusion has to be dealt with.

Knowing that in a three dimensional simulation including all these detailed calculations is not yet feasible, we decided to circumvent these difficulties by mapping data from one dimensional simulations on our model. More precisely, the data calculated by Cooperstein and Wambach [14, 13] was used. Their tabulated data enables us to define the temperature $T(\rho)$ and electron fraction $Y_e(\rho)$ as a function of density and should yield a good approximation for the state of the matter until core bounce³.

Temperature

The tabulated data obtained from Cooperstein's work was interpolated using appropriate fit functions. The relation $T(\rho)$ that was finally used in our simulations is shown in figure 3.3 together with Cooperstein's values.

Electron Fraction

The electron fraction as a function of density $Y_e(\rho)$ was obtained from Cooperstein's data and interpolated just like the temperature. The resulting relation for our simulation is shown in figure 3.4. Note that $Y_e(\rho)$ is continued relatively arbitrarily for $\rho > 10^{12} \frac{\text{g}}{\text{cm}^3}$, yet the convergence of Y_e to a value $Y_e \approx 0.31$ for very high densities is realistic as described in 2.2.1.

Further Thermodynamic Quantities

More thermodynamic quantities such as the entropy per baryon or the electron chemical potential could be defined using Cooperstein's data but this is not necessary

³Our simulation can currently only be considered realistic until core bounce as will be further explained in 3.7.

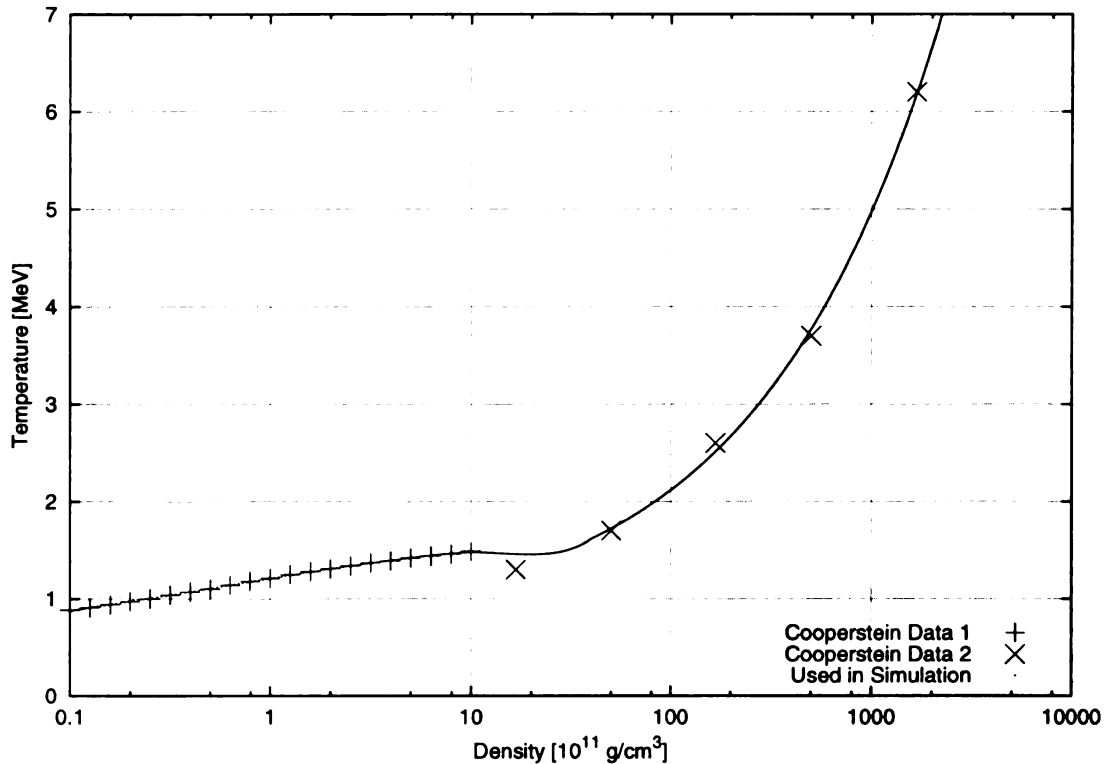


Figure 3.3: $T(\rho)$ for the infall phase

because the EsOS we use do not need more input. Au contraire – virtually any imaginable quantity of interest can be *returned* by the EsOS.

3.3.3 Helmholtz EOS and Lattimer & Swesty EOS

As a more realistic EOS (than the one presented in 3.3.1) we used a combination of the nuclear EOS by Lattimer & Swesty and the Helmholtz EOS by Timmes (mentioned in 2.3.1). The former is used for $\rho \geq 10^{11} \frac{\text{g}}{\text{cm}^3}$, the latter for $\rho < 10^{11} \frac{\text{g}}{\text{cm}^3}$ where the nuclear contribution to the pressure is negligible. Note that the LS EOS also includes an electron gas contribution. Moreover, there are different parameter sets for the LS EOS which mainly affect the behavior of the matter at densities above nuclear. We used the parameter set in which the nuclear compressibility is $K = 180 \text{ MeV}$.

Both EsOS are available as **FORTTRAN**-programs [27, 46]. The input for the LS EOS

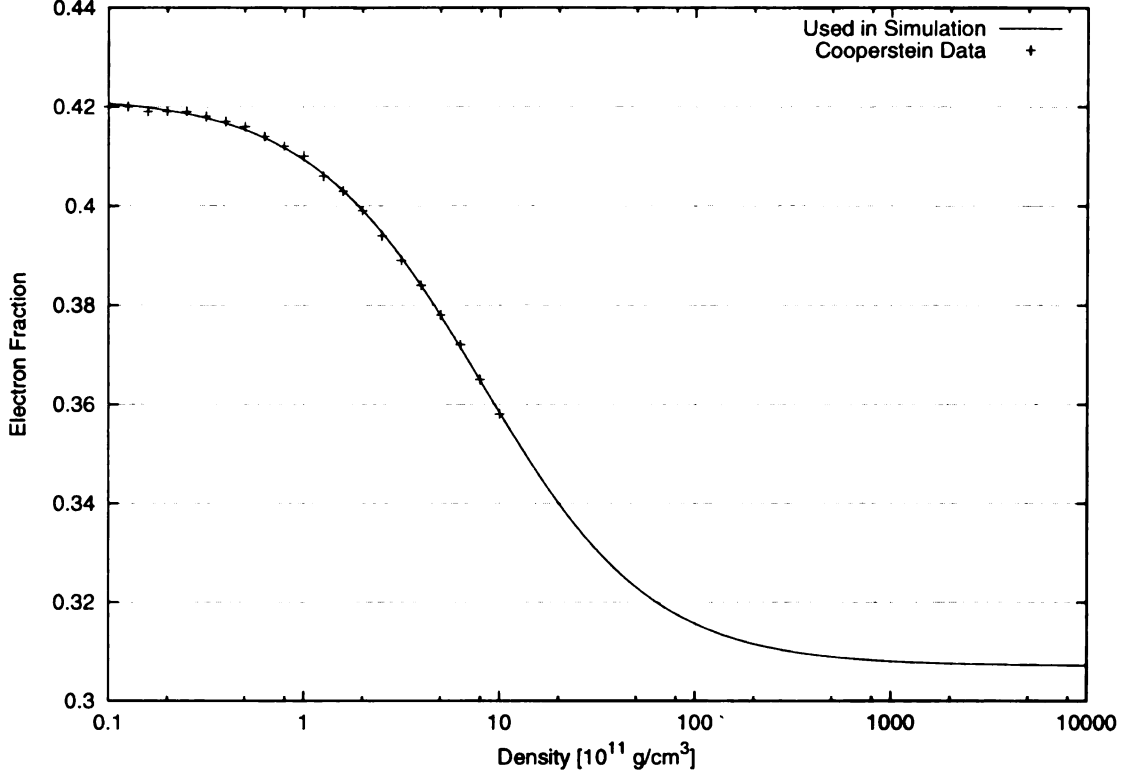


Figure 3.4: $Y_e(\rho)$ for the infall phase

is ρ , T , and Y_e . The Helmholtz EOS gets the composition of the matter instead of Y_e . This composition is defined by the mass and charge numbers of an arbitrary number of different ions present in the matter and their respective number fractions. We used a small constant hydrogen fraction of 2.69×10^{-4} (taken from [14]). The rest of the matter got a charge number $Z = 26$ and a mass number $A(\rho) = Z/Y_e(\rho)$ mimicking the change of the matter composition due to ongoing electron capture while taking into account that most of the nuclei present are iron isotopes. Note that this approximation does not lead to absurd isotopes (with extreme neutron excess) because the Helmholtz EOS is only used for densities at which $Y_e \gtrsim 0.41$ which yields $A \lesssim 63$.

The internal energy per baryon $u_{int}(\rho) = u_{int}(\rho, T(\rho), Y_e(\rho))$ this ultimately yields using $T(\rho)$ and $Y_e(\rho)$ from 3.3.2 is shown in figure 3.5. There is no longer a density

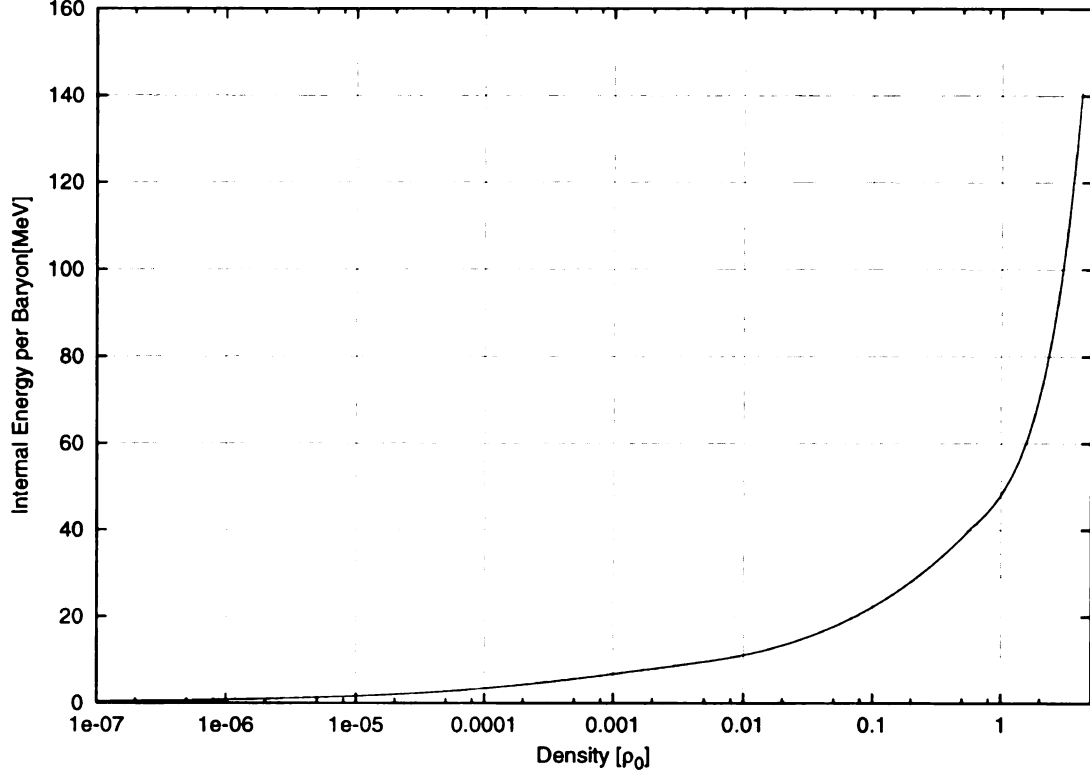


Figure 3.5: Combination of Helmholtz EOS and LS EOS used in our simulation

region in which the EOS is attractive.

3.3.4 How the EOS Affects the Dynamics

One major question still needs to be answered: how does the EOS affect the dynamics of our core collapse simulation? This happens through $\vec{F}_{EOS,j}$, the force due to the EOS on test particle j as introduced in equation 3.2.

The EOS is used to calculate the internal energy per baryon u_{int} for every grid cell. This is easily converted into an internal energy per test particle by multiplying with $m_{tp}/m_B =: \nu$, the number of baryons per test particle. A gradient of this quantity at the location of test particle j is calculated using the technique described in 3.2.4. Then

$$\vec{F}_{EOS,j} = -\nu \nabla u_{int} \Big|_{(r,\phi,\theta)_j} \quad (3.20)$$

where $(r, \phi, \theta)_j$ are the spherical coordinates of test particle j 's position.

This technique is justified by energy conservation: during the infall phase it is a good approximation to neglect energy losses due to neutrinos (and photons) radiating away from the core because the magnitude of these losses is relatively small. Thus the sum of the core's kinetic energy E_{kin} , gravitational energy E_G , and thermal (internal) energy E_{int}

$$E_{tot} = E_{kin} + E_G + E_{int} \quad (3.21)$$

should be approximately constant during collapse. After bounce this is no longer valid as neutrino losses cannot be neglected anymore.

We will now explain how our method of calculating $\vec{F}_{EOS,j}$ implies the (approximate) conservation of E_{tot} . For simplicity, assume the situation shown in figure 3.6: two neighboring cells $(n_r, n_\phi, n_{\cos\theta})$ and $(n_r + 1, n_\phi, n_{\cos\theta})$ with the respective internal energies per test particle u_1 and u_2 . Moreover let $u_2 > u_1$, a test particle j located in cell 1, and the internal energy per test particle in all remaining neighboring cells of cell 1 be equal to u_1 so that $\vec{F}_{EOS,j}$ only gets a contribution from the gradient between u_1 and u_2 . Let gravity be “turned off” for the moment. Then, the force on j due to the EOS is approximately

$$\vec{F}_{EOS,j} = -\nabla u \approx -\frac{u_2 - u_1}{\Delta r} \vec{e}_r$$

where Δr is the radial cell width. Now assume that j travels from cell 1 to cell 2 as indicated by the dashed arrow. In doing so, due to the action of $\vec{F}_{EOS,j}$, its kinetic energy is reduced by

$$\Delta E_{kin} = \int_1^2 \vec{F}_{EOS,j} \cdot d\vec{r} \approx -\frac{u_2 - u_1}{\Delta r} \Delta r = u_1 - u_2.$$

Thus, the internal energy j gains (it is now located in a cell with a higher internal energy per test particle) is (approximately) compensated by its loss of kinetic energy.

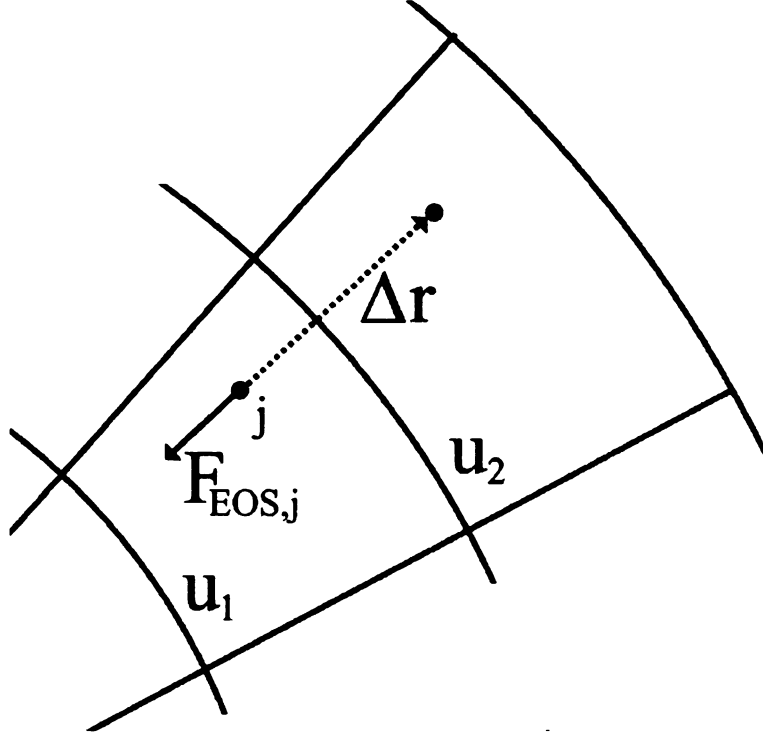


Figure 3.6: Test particle j traveling from one grid cell to another

In this derivation, the way gradients are calculated was simplified and the fact that the appearance of j in cell 2 (and its disappearance in cell 1) affects u_1 and u_2 was neglected.

One might consider using the pressure as returned by the EOS used (instead of the internal energy) to calculate $\vec{F}_{EOS,j}$. However, there is no evident way of doing this in our model without making further assumptions about the size and shape of the test particles. We wanted to avoid this because the test particles are after all completely imaginary objects.

In our simulations, energy (and angular momentum) is kept track of to assure that the method just described works properly and the conservation of these quantities is not destroyed by numerical limitations, a too large time step size, a too low grid resolution, or other effects.

3.4 Symmetry Assumptions, Boundary Conditions, and Numerical Problems

Like in almost all numerical simulations there are certain subtleties in ours that require special attention. It is also desirable to make simplifications wherever possible to invest the available CPU time where it does most good.

3.4.1 Symmetry Assumptions

The total number of grid cells N_{cells} is limited by the condition $N_{cells} \ll N_{tp}$ because otherwise there are inevitably many cells which contain very few, just one, or no test particles. This leads to unphysical density fluctuations on a length scale essentially given by the cell size. Most of the time we used $N_{tp} = 10^6$ and found that this kind of trouble occurs when $N_{cells} \gtrsim 2 \times 10^4$ is used. To get a good resolution of interesting features, it is thus desirable to make use of symmetries. Two symmetries can be anticipated almost a priori:

- equatorial symmetry: there is virtually no imaginable reason (other than fluctuations or numerical errors) why equatorial symmetry should be broken in our model. Nevertheless, we ran several simulations in which it was not enforced without finding significant deviations therefrom. Therefore equatorial symmetry is enforced by averaging the thermodynamic quantities in grid cells mirror symmetric about the equatorial plane.
- cylindrical symmetry: exactly the same way, it was found that there are no significant deviations from cylindrical symmetry (about the rotation axis) in our model. Hence this symmetry is enforced by setting $N_\phi = 1$.

Note that no symmetry whatsoever was enforced for the positions and momenta of the test particles (i.e. these are not located in positions mirror symmetric about the equatorial plane or something of the kind).

3.4.2 Derivatives at Grid Boundaries

z-Axis

The $\frac{\partial \Omega}{\partial \theta}$ derivative at the z-axis is simply set to zero. This causes the test particles initially located near the z-axis to tend to “stick” to it. This is not good and only done this way because we could not think of a better way to deal with it. Note, however, that this affects just a small fraction of the test particles. Those particles near the z-axis are initially rotating very slowly around this axis so that there is at least no obvious reason why they would move away from it. This treatment can also be seen as a consequence of assuming cylindrical symmetry.

Grid Surface

At the radial edge of the grid, $\frac{\partial \Omega}{\partial r}$ is calculated by assuming that the density of the matter outside the grid has a value ρ_{min} slightly lower than that present at the outermost layer of the iron core at the onset of the simulation. Thus the presence of the star’s mantle and envelope beyond the surface of the iron core is imitated in a simple way.

Center

The derivative $\frac{\partial \Omega}{\partial r}$ at the center is also set to zero. This is reasonable because matter should not traverse the core’s center in large amounts.

3.4.3 Background Density

As described in 3.2.2 the grid is usually chosen a bit larger than the space region in which the vast majority of the test particles is located. Hence, empty cells may occur. To assume $\rho = 0$ in these would be completely unrealistic as even outside the iron core (in the star’s mantle) densities are high. Thus a minimum density ρ_{min} (the magnitude of which is determined as described in 3.4.2) is enforced throughout the grid by setting ρ to ρ_{min} wherever $\rho < \rho_{min}$.

3.4.4 Singularity Treatment

The force on particle j due to gravity $\vec{F}_{G,j}$ calculated as in equation 3.3 has a (numerical) singularity at $|\vec{r}_j| = 0$. This can cause numerical trouble for very small $|\vec{r}_j|$ which yield extremely large forces. Thus these particles can be vigorously accelerated and more or less “shot” out of the core’s center. This is unrealistic and not desirable. Therefore we decided to “switch off” gravity for the $\frac{N_{tp}}{10^4}$ or $\frac{N_{tp}}{10^3}$ innermost particles. Note that $|\vec{F}_{G,j}|$ is relatively small for the innermost particles anyway because the mass they enclose is comparatively small.

3.5 Time Development

The time development of the collapsing core is essentially obtained by numerically integrating the system of $2 \times 3 \times N_{tp}$ coupled first-order ordinary differential equations 3.1 and 3.2. This is done using a *fourth-order Runge-Kutta* algorithm, a standard method precisely described e.g. in [36]. The whole simulation program (except for the EsOS mentioned in 3.3.3 which are written in **FORTRAN**) is written in **C++**.

This algorithm as such uses a constant time step size. As it is clear that the core’s matter will be continuously accelerated during collapse we implemented a very simple

way to adapt the step size nonetheless: once the core has contracted by a certain factor (e.g. 10) the step size is suddenly made sufficiently smaller (e.g. divided by 10) to guarantee a good resolution of the further time development. Depending on the initial conditions used in the simulation this step size modification can be done several times.

3.6 Calculation of Observables

Several physical observables are kept track of during the time development of the collapsing core. Among these are the total kinetic energy E_{kin} , (Newtonian) gravitational energy E_G , internal energy E_{int} , and the total angular momentum \vec{L} of the core. It is important to do this in order to make sure that the core's total energy and angular momentum are (at least approximately) conserved. It is also instructive to follow the conversion of the different energy types into one another. The observables are calculated as follows:

- E_G is calculated assuming spherical symmetry:

$$E_G = - \sum_{j=1}^{N_{tp}} G \frac{\# \left\{ i \in \{1, \dots, N_{tp}\} \mid |\vec{r}_i| < |\vec{r}_j| \right\} m_{tp}^2}{|\vec{r}_j|}. \quad (3.22)$$

- For E_{kin} the well-known expression for the relativistic energy is used:

$$E_{kin} = \sum_{j=1}^{N_{tp}} \left(\sqrt{m_{tp}^2 c^4 + \vec{p}_j^2 c^2} - m_{tp} c^2 \right) = \sum_{j=1}^{N_{tp}} \sqrt{m_{tp}^2 c^4 + \vec{p}_j^2 c^2} - N_{tp} m_{tp} c^2 \quad (3.23)$$

- E_{int} is evidently calculated by summing up the internal energies of all particles:

$$E_{int} = \sum_{n_r=1}^{N_r} \sum_{n_\phi=1}^{N_\phi} \sum_{n_{\cos\theta}=1}^{N_{\cos\theta}} u_{int}^{(tp)}(n_r, n_\phi, n_{\cos\theta}) \times N_{tp}(n_r, n_\phi, n_{\cos\theta}), \quad (3.24)$$

where $u_{int}^{(tp)}$ now denotes the internal energy per *test particle* not per baryon.

- The calculation of \vec{L} is also obvious:

$$\vec{L} = \sum_{j=1}^{N_{tp}} \vec{r}_j \times \vec{p}_j. \quad (3.25)$$

3.7 Advantages and Weaknesses of this Method

A big advantage of the method described in the preceding sections is its simplicity: it enables us to simulate the collapse of a rotating iron core in three space dimensions until core bounce on a quite ordinary home computer. Our code also conserves angular momentum (apart from tiny numerical errors) which was verified in countless different simulation runs using different initial conditions and EsOS. Angular momentum conservation is even good if cylindrical symmetry is not assumed. If the step size is chosen sufficiently small, energy conservation is given at least approximately (this will be deepened in 4.2.1, 4.2.2, and 4.2.3).

Yet, one has to keep in mind that it is a model. There is no way whatsoever to guarantee that the results our simulation yields are completely realistic (this is more or less the case for all core collapse simulations). Possible new phenomena found in such models may however guide our intuition in understanding the supernova explosion mechanism.

A weakness of our model at this time is that there are no collisions between the test particles. This is the main reason why our simulation is currently limited to the infall phase: after bounce the test particles that rebound from the core just fly through the infalling outer core matter without resistance. No shockwave is created making this a completely unrealistic scenario. Even during the infall phase some trouble arises from the lack of collisions (that will be described in chapter 4).

The simplicity of the model mentioned above can also be considered a weakness:

the use of data obtained from lower dimensional simulations certainly restricts the possibilities for the occurrence of new phenomena.

3.8 Computational Requirements

The computationally most expensive operations will be depicted now. In order to calculate $\vec{F}_{G,j}$ the particles essentially have to be sorted by their distance from the center. Moreover, to calculate $\vec{F}_{EOS,j}$ the EOS must be evaluated for every cell of the grid. Both things need to be done four times in every time step (due to the way the fourth-order Runge-Kutta algorithm works).

A typical simulation run uses 10^6 test particles, approximately 12×10^3 grid cells, and about 10^3 time steps. Therefore the *quicksort* algorithm as described in [36] (which is used to sort the particles) is called 4000 times. Most expensive, however, are the EOS calls: $4 \times 12 \times 10^3 \times 10^3 \approx 5 \times 10^7$ of these occur in a typical run. The time needed to calculate one time step⁴ can become up to $\approx 4\text{min}$ when the LS EOS is used, leading to a total time of roughly $4000\text{min} \approx 3\text{days}$.

3.9 Output Possibilities

Two different output programs were written to visualize the data created by the simulation program. For simplicity, **Microsoft Visual Basic**® was used for these programs which involve a lot of graphical output. The source code for both programs is reproduced in appendix B.

⁴All these times are valid for a 1GHz Intel Pentium® III with 512 MB RAM that was used throughout this project.

3.9.1 Density Output

The mass density in a slice through the core that includes the rotational axis (the z-axis) can be plotted for every time step. The different densities are indicated by colors. Due to the large density changes a logarithmic density scale had to be used.

3.9.2 Test Particle Output

Another program shows (up to) 2000 test particles in a (pseudo) three dimensional picture⁵. The motion of these individual particles can be followed during collapse. The perspective can be changed at any time. Apart from giving a good impression of the collapse dynamics, this program is very useful to identify errors like those arising from a too large number of grid cells.

⁵Showing more particles is not sensible because separate test particles could than hardly be discerned.

Chapter 4

Numerical Results and Interpretation

We will now describe some results obtained using the code described in chapter 3.

4.1 Angular Momentum Conservation

In all simulation runs angular momentum conservation was almost perfect. This should not come as a surprise because cylindrical symmetry about the rotation axis was assumed. The plots of the total angular momentum $|\vec{L}|$ as a function of time are often just a horizontal line. A typical angular momentum evolution (for the initial conditions and EOS that will be described in 4.2.1) is shown in figure 4.1. Note that during the fluctuations at the end of the simulation run $|\vec{L}|$ varies by less than 0.01%.

4.2 Results of Simulation Runs

We performed three series of simulations in which different EsOS were used.

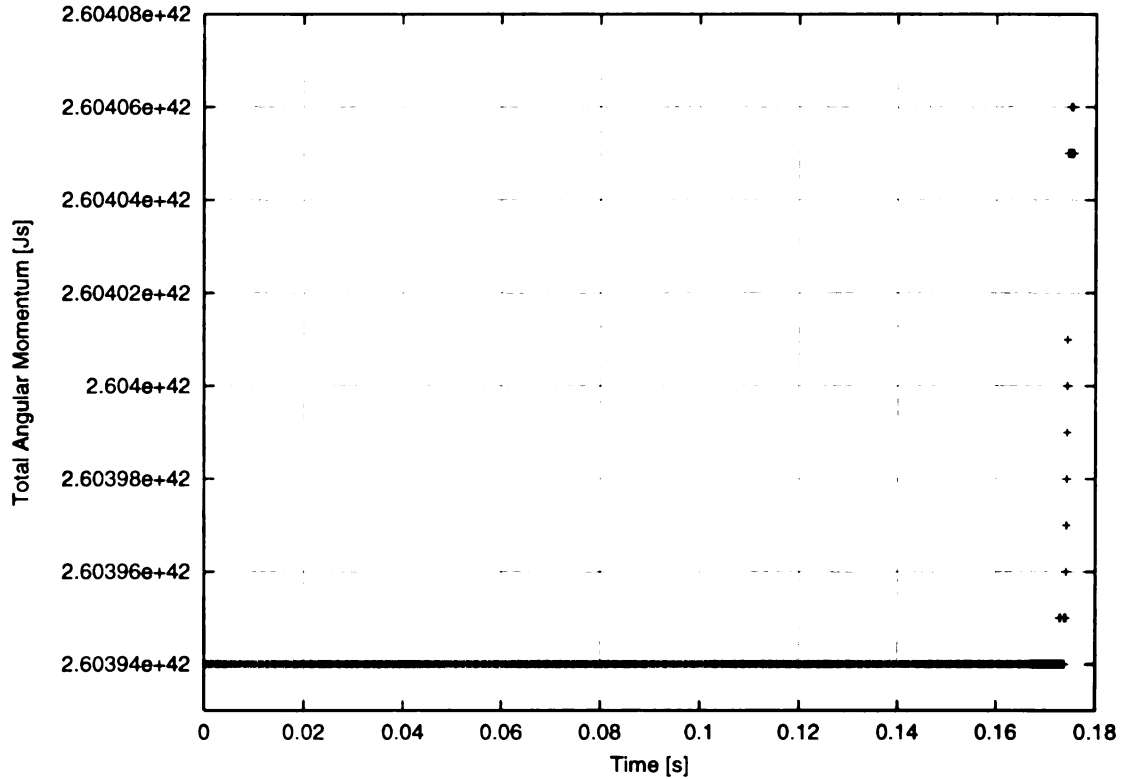


Figure 4.1: Angular momentum as a function of time for a typical simulation run

4.2.1 Cold Nuclear EOS without Electron Contribution

The first series is more of a toy model we created as a first test for the code: we only used the cold nuclear contribution (as in equation 3.18) to the EOS. Nevertheless this model exhibits some realistic features. The iron core mass was chosen to be $1.5M_{\odot}$ and the background density set to $\rho_{min} = 0$. To achieve a homologous collapse, a spherical homogeneous mass distribution had to be chosen: the test particles were randomly distributed in a sphere with radius $1.676307 \times 10^6 \text{m}$ which is about 100 times the radius this mass would have at nuclear matter density (if also homogeneously distributed in a sphere). This yields an (approximately) homologous collapse because the cold nuclear EOS hardly affects the matter at all at densities $\rho \ll \rho_0$ and the collapse of a homogeneous self-gravitating sphere is homologous.

Name of Run	ω_0 [$\frac{\text{rad}}{\text{s}}$]	$ \vec{L}_0 $ [10^{42} Js]	$ \frac{E_{\text{rot}}}{E_G} _{\text{init}}$	t_{bounce} [ms]	ρ_{max} [ρ_0]
CNEOS00	0.0	0.0	0.0%	173.8	9.5
CNEOS03	0.3	0.977	0.073%	174.1	3.7
CNEOS04	0.4	1.30	0.13%	174.2	5.1
CNEOS05	0.5	1.63	0.20%	174.5	3.5
CNEOS06	0.6	1.95	0.29%	174.8	3.6
CNEOS07	0.7	2.28	0.40%	175.0	3.0
CNEOS08	0.8	2.60	0.52%	175.4	2.8

Table 4.1: Simulation runs using the cold nuclear EOS. The abbreviations are explained in the text.

In this series the cores were started rotating like a rigid body around the z-axis. Different initial angular velocities ω_0 were used. The grid parameters $N_r = 120$, $N_\phi = 1$, and $N_{\cos\theta} = 100$ were applied. 10^6 test particles were employed. All runs of the series are summarized in table 4.1.

Energy Conservation

Energy conservation is only fulfilled approximately. It is very good before the EOS starts to strongly affect the dynamics at high densities. Figure 4.2 shows a typical energy development. The time on the abscissa is measured from the onset of the simulation. At earlier times (than shown in figure 4.2), only E_{kin} and E_G contribute significantly to E_{tot} . Although the total energy is obviously not conserved at all times it is good to see that the energy deficit that arose shortly before core bounce is ultimately corrected.

Time of Core Bounce and Maximum Density

Note that a time t_{bounce} at which core bounce occurs can be identified in the energy graph as a minimum of E_G . Another common observable in supernova simulations is the maximum density ρ_{max} achieved at core bounce. We decided to calculate ρ_{max}

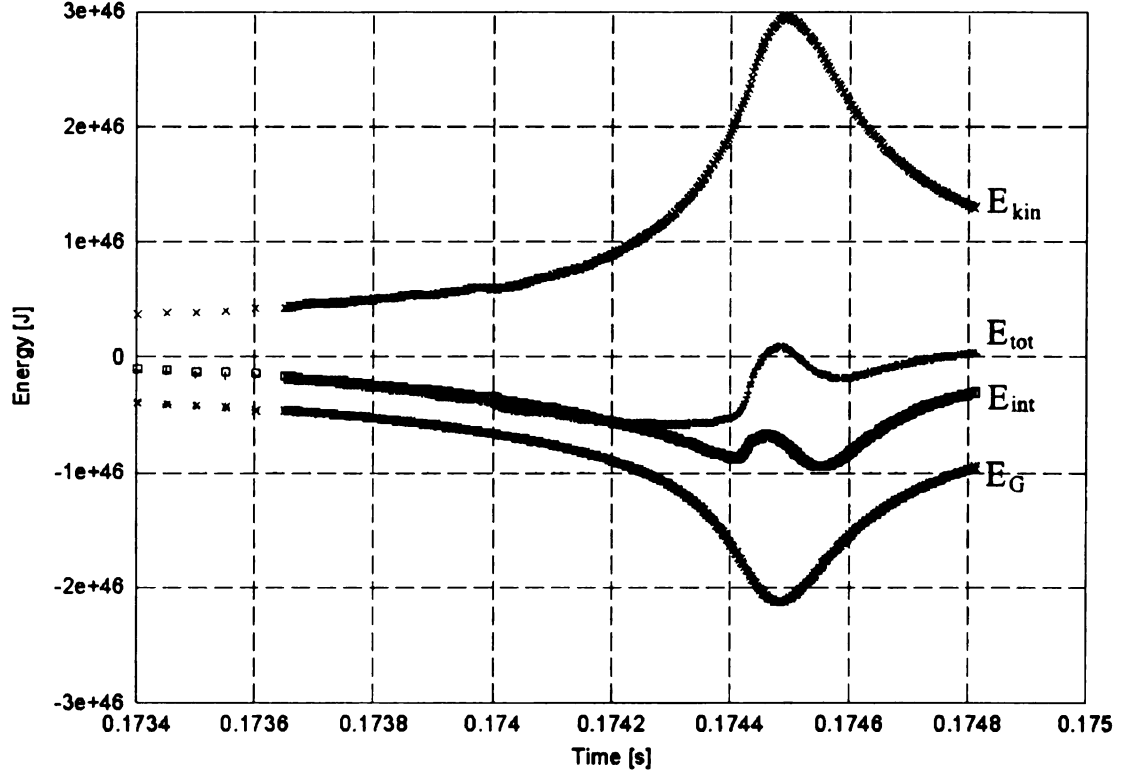


Figure 4.2: Time development of the different energies in the late stages of simulation CNEOS05

by averaging over a certain number of highest densities encountered in the grid cells during the simulation to avoid possible errors due to density fluctuations¹.

The values for t_{bounce} and ρ_{max} obtained from the different simulation runs of this series are shown in table 4.1. Obviously, this data confirms that a greater initial angular momentum leads to a later core bounce. There is also a clear tendency that rotation causes lower densities at core bounce (although model CNEOS04 unfortunately breaks ranks without identifiable reason).

$|\frac{E_{rot}}{E_G}|_{init}$ is the ratio of the initial rotational energy to the initial gravitational energy. It is common to use this ratio as a measure for how rapidly a supernova core is rotating.

¹Usually the roughly 200 highest densities encountered during the whole time development were used for this.

Deviations from Spherical Symmetry

Figure 4.3 shows the development of the density distribution during the infall phase till shortly after bounce for models CNEOS03 (slowly rotating), CNEOS05 (moderately rotating), and CNEOS07 (rapidly rotating).

The clumping of the matter that becomes obvious in the plots labeled with (b) is due to the wide attractive density region of the EOS: once there is a small density fluctuation somewhere, it is amplified until nuclear density is reached because this goes along with a decrease of the total energy. This leads to clumps of matter at nuclear density whose size is essentially determined by the size of the grid cells. This is obviously not a realistic feature and should be ignored.

A trivial feature of all three models is the oblate shape of the mass distribution during the late stages of the infall phase (b). In the slowly rotating model a prolate shape is apparent after bounce (e) – a feature that appeared in other simulations before (e.g. [57]). In the rapidly rotating model two other interesting deviations from spherical symmetry can be seen: during the late stages of the infall phase (c) the infall velocity of the matter along the rotation axis becomes very large which cannot just be due to the lack of centrifugal forces there because it leads to a slight “doughnut” shape of the mass distribution. This higher infall velocity also leads to a significantly larger density along the z-axis (the vertical in the plots) close to the center. One might envision that along the rotation axis vortices are formed which help the matter swirl to the center faster.

The assumption of spherically symmetric gravity is no longer completely justified when such large deviations from spherical symmetry as in model CNEOS07 occur. Note, however, that this assumption tends to stabilize the core against large scale deviations from spherical symmetry.

	CNEOS03	CNEOS05	CNEOS07
t_a	0	0	0
t_b	173.352	173.745	174.305
t_c	173.907	174.287	174.645
t_d	174.102	174.5	175.000
t_e	174.417	174.812	175.205
r_a	1844	1844	1844
r_b	87.81	83.06	75.57
r_c	58.72	53.60	53.29
r_d	115.2	57.99	43.49
r_e	148.7	95.05	54.23

Table 4.2: Key for figure 4.3. t_a through t_e are the times (in ms) corresponding to the density profiles labeled (a) through (e) in the figure. r_a through r_e are the corresponding radii (in km) of the shown density plots.

In model CNEOS07 shortly after core bounce (e) the mass density in the equatorial plane is much higher than that along the z-axis. A slight tendency towards this feature can already be seen in the moderately rotating model CNEOS05.

In the three core bounce plots (d) and after bounce (e) it can be seen how rotation leads to a more diffuse mass distribution. Keep in mind, however, that the radius in plot (d) for model CNEOS03 is roughly two times larger than in the other two plots.

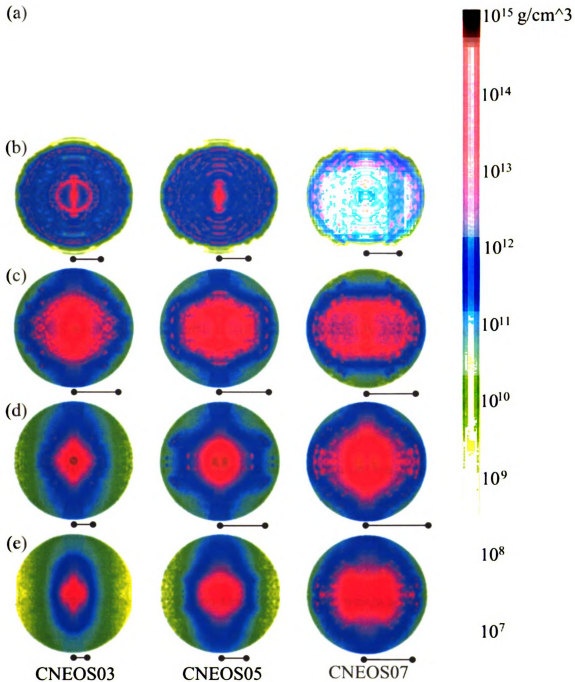


Figure 4.3: Mass density in a slice in the x - z -plane at five different times for models CNEOS03 (left), CNEOS05 (center), and CNEOS07 (right). Events at the respective times: (a) onset of simulation, (b) clumping and presence of centrifugal forces become apparent, (c) formation of “vortices” in CNEOS07, (d) core bounce, (e) shortly after core bounce. Note that the radii of the shown density profiles vary (see table 4.2 for more data). The black lines below each plot indicate a length of 43.5 km. The length scale for the plots labeled by (a) is much larger than this. Images in this thesis are presented in color.

4.2.2 Cold Core with Polytrope Electron Contribution

It is desirable to use a realistic (inhomogeneous) density profile of a presupernova core like that shown in figure 2.1 as initial condition. Unfortunately, this turned out to be impossible in our model. It is quite important that the inner core matter (or in our simulation the test particles representing it) arrive at the center of the core simultaneously. Great accuracy is needed to achieve this given a change in the length scale by a factor of about 10^2 during collapse.

Many other groups have tripped over this problem [5, 59]. One remedy (followed e.g. by [59]) is to use (artificial) initial conditions that are an equilibrium solution of the EOS used and then initiate collapse by suddenly reducing the electron fraction (or the adiabatic index if a polytropic EOS is used) a little bit.

As the effects of rotation should mainly play a role during the late stages of collapse (when angular momentum conservation can lead to very high angular velocities) we decided to use the following approach: the simulation is started when collapse is already going on and the (inner) core has already contracted by a factor of 5. Thus the problems arising from the scale change are simplified. The fact that the matter is already falling inward is mimicked by imprinting an initial velocity profile $v(r) = k_v r$ (in addition to the rotation) on the test particles, where a realistic value $k_v = 89.1 \text{ s}^{-1}$ (estimated using the results of previous simulations [4]) is chosen. This is a good model only for the inner core, so the mass is chosen to be $0.7M_\odot$.

Since centrifugal forces are relatively weak during the early collapse stages the initial mass distribution is still assumed to be spherically symmetric. The initial mass density as a function of the radius calculated by Woosley and Weaver² [56] and the variant contracted by a factor of 5 we used in our simulation are shown in figure

²Strictly speaking, Woosley and Weaver's data was approximated (extremely accurately) by the analytic expression $\rho(r) = \frac{1}{k_1 r^3 + k_2}$ with appropriate parameters k_1 and k_2 for figure 4.4.

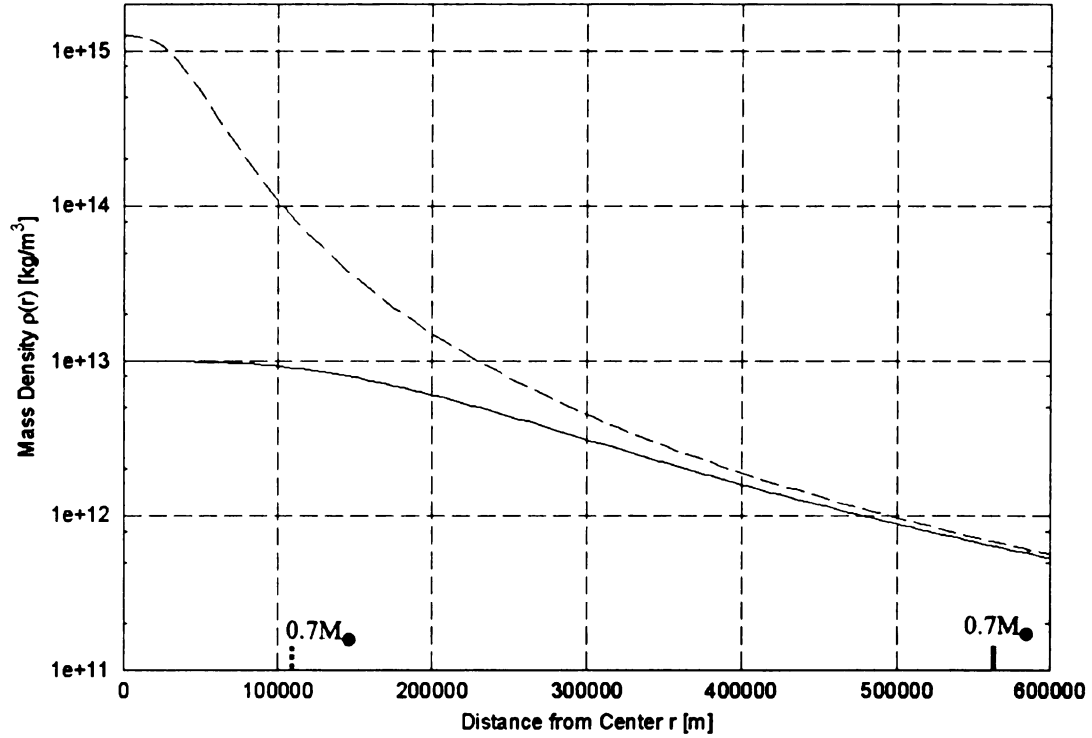


Figure 4.4: Initial mass density as a function of the radius calculated by Woosley and Weaver [56] (solid line) and the variant contracted by a factor of 5 we use in our simulation (dashed line). The location at which the enclosed mass is $0.7M_{\odot}$ (edge of the inner core) is indicated for both distributions on the abscissa.

4.4.

If collisions between the test particles were included in our code the use of fully realistic initial conditions might become possible because these collisions should lead to a local averaging of the infall velocity thus facilitating the simultaneous arrival of most test particles at the center.

The angular velocity profile of Heger's (rotating) progenitor (figure 2.5) indicates that it is an excellent approximation to assume that the inner core (initially) rotates like a rigid body. Therefore we used a constant angular velocity ω_0 as initial rotation. However, some simulation runs were performed using a parameterized r -dependent

Name of Run	ω_0 [$\frac{\text{rad}}{\text{s}}$]	$ \vec{L}_0 $ [10^{41} Js]	$ \frac{E_{\text{rot}}}{E_G} _{\text{init}}$	t_{bounce} [ms]	ρ_{max} [ρ_0]
PCNEOS01	10	0.467	0.027%	3.01	5.68
PCNEOS07	70	3.27	1.3%	3.20	2.93
PCNEOS13	130	6.06	4.5%	3.53	2.09
PCNEOS19	190	8.86	9.6%	3.86	1.66

Table 4.3: Simulation runs using the combination of the cold nuclear EOS and the polytrope EOS for the electron gas. The abbreviations are explained in the text.

initial angular velocity profile [57]:

$$\omega_0(r) = \omega_c \frac{r_0^2}{r^2 + r_0^2}. \quad (4.1)$$

This yields a very good approximation of the angular velocity of Heger’s progenitor if the parameters ω_c (the central angular velocity) and r_0 (which defines the degree of differential rotation) are chosen suitably. As anticipated no significant differences to the rigidly rotating models occurred.

In this series, the simple polytrope electron contribution was added to the nuclear part of the EOS (see equation 3.19). The parameters $N_{tp} = 10^6$, $N_r = 120$, $N_{\cos\theta} = 100$, and $\rho_{\text{min}} = 1.3 \times 10^{11} \frac{\text{g}}{\text{cm}^3}$ were applied.

Table 4.3 shows the data for all runs of this series.

Energy Conservation

Again, energy conservation is approximately fulfilled. Figure 4.5 shows a typical energy development. We will not conceal, however, that in some of the other runs of this series (especially PCNEOS01 and PCNEOS07), energy conservation was not as good as in figure 4.5, most likely because of a bad choice for the initial time step size.

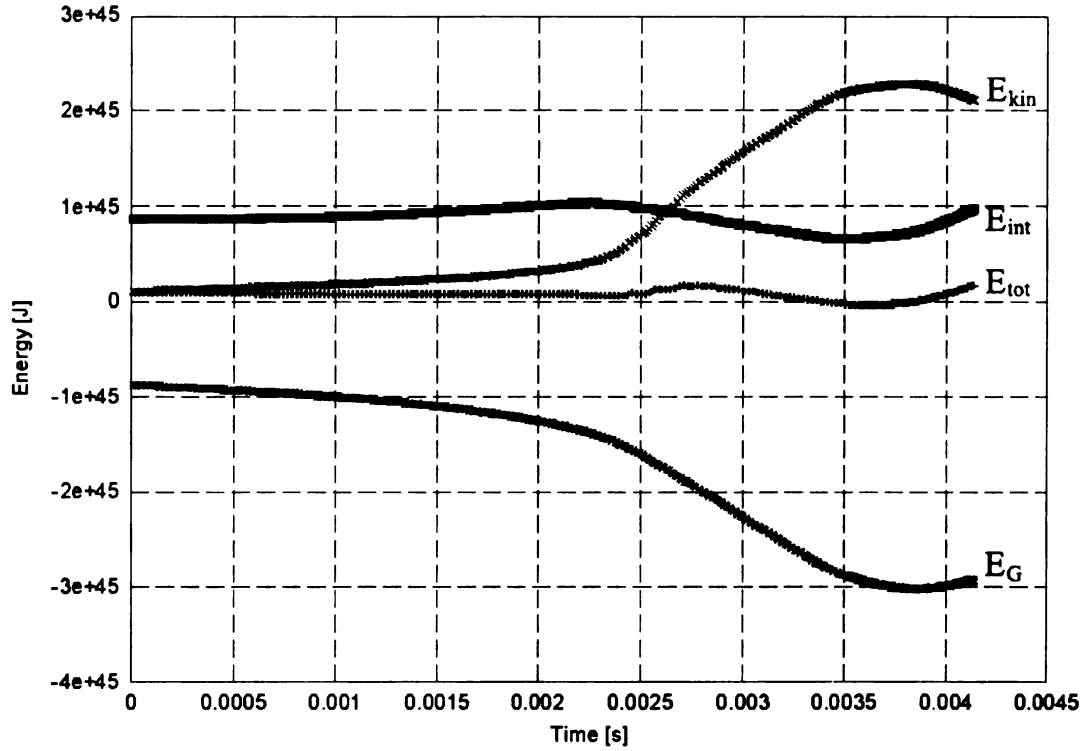


Figure 4.5: Time development of the different energies in simulation PCNEOS19

Deviations from Spherical Symmetry

Figure 4.6 shows the development of the mass density for the three simulation runs PCNEOS07 (slowly rotating), PCNEOS13 (moderately rotating), and PCNEOS19 (rapidly rotating).

The peculiar clumping of the matter that was apparent in the previous series does not occur in these models as can be seen in the plots labeled by (b) and (c). The oblate shape of the density distributions is again visible in plots (b) and (c) and obviously more pronounced in the more rapidly rotating models. The slowly rotating model PCNEOS07 shows a slight prolate shape shortly after core bounce in plots (d) and (e).

The formation of “vortices” along the rotation axis (the vertical in the plots)

	PCNEOS07	PCNEOS13	PCNEOS19
t_a	0	0	0
t_b	2.00	2.00	2.00
t_c	3.38	2.82	2.53
t_d	3.86	3.53	3.19
t_e	4.14	3.93	3.64
r_a	123.4	123.4	123.4
r_b	102.9	102.9	102.9
r_c	56.10	56.10	56.10
r_d	56.10	56.10	56.10
r_e	68.57	68.57	68.57

Table 4.4: Key for figure 4.6. t_a through t_e are the times (in ms) corresponding to the density profiles labeled (a) through (e) in the figure. r_a through r_e are the corresponding radii (in km) of the shown density plots.

during the infall phase can be seen in the two more rapidly rotating models in plots (c) and (d). It is also quite obvious in plots (e) of these models that most of the mass rebounds from the core in the equatorial plane.

The core bounce plots (d) show how rotation leads to a drastic deformation of the core at bounce.

A curiosity occurs in model PCNEOS07: in plot (c) a slight prolate shape can be seen which is quite odd during the infall phase. This is most likely a freak due to the (aforementioned) bad time step size used in this simulation run.

It is interesting to note that almost all of the features that appeared in the toy model using the cold nuclear EOS reappeared in this much more realistic one.

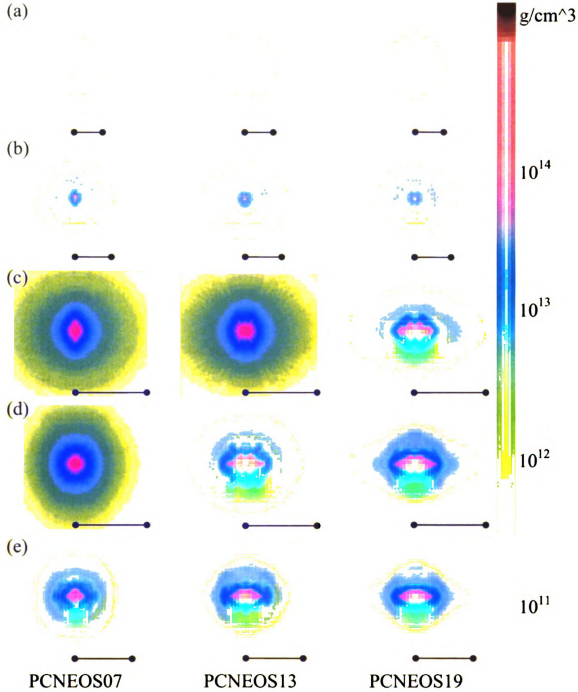


Figure 4.6: Mass density in a slice in the x-z-plane at five different times for models PCNEOS07 (left), PCNEOS13 (center), and PCNEOS19 (right). Events at the respective times: (a) onset of simulation, (b) after 2 ms, (c) presence of centrifugal forces and vortices become apparent, (d) core bounce, (e) shortly after core bounce. Note that the plots have different radii. The black line below each plot indicates a length of 56.1 km. See table 4.4 for more data. Images in this thesis are presented in color.

Name of Run	ω_0 [$\frac{\text{rad}}{\text{s}}$]	$ \vec{L}_0 $ [10^{41} Js]	$ \frac{E_{\text{rot}}}{E_G} _{\text{init}}$	t_{bounce} [ms]	ρ_{max} [ρ_0]
HLSEOS01	10	0.466	0.027%	3.44	2.99
HLSEOS07	70	3.26	1.3%	3.61	2.35
HLSEOS10	100	4.67	2.7%	3.76	1.63
HLSEOS13	130	6.06	4.5%	3.89	1.03
HLSEOS16	160	7.46	6.8%	4.03	0.56
HLSEOS19	190	8.86	9.6%	4.17	0.34
HLSEOS22	220	10.3	13%	4.31	0.21

Table 4.5: Simulation runs using the combination of the Helmholtz and the LS EOS. The abbreviations are explained in the text.

4.2.3 Combination of Helmholtz EOS and Lattimer & Swesty EOS

In this series the same initial conditions as in 4.2.2 were used. But as EOS the most realistic approach to actual core collapse conditions in this work was implemented – the combination of the Helmholtz EOS for the e^-/e^+ gas at lower densities and the LS EOS at higher densities. Both EsOS were described in 3.3.3. The parameters $N_{tp} = 10^6$, $N_r = 110$, $N_{\cos\theta} = 100$, and $\rho_{\min} = 1.3 \times 10^{11} \frac{\text{g}}{\text{cm}^3}$ were applied.

Table 4.5 shows the data for all runs of the series. With this realistic EOS the time of core bounce t_{bounce} and the maximum density ρ_{max} obviously depend on the initial angular momentum in a strictly monotonous way.

Energy Conservation

Figure 4.7 shows a typical energy development for the models of this series. Again, energy conservation is only fulfilled approximately. Temporary violations appear especially near core bounce. However, the conversion of gravitational energy into kinetic and internal energy during collapse is rendered quite well.

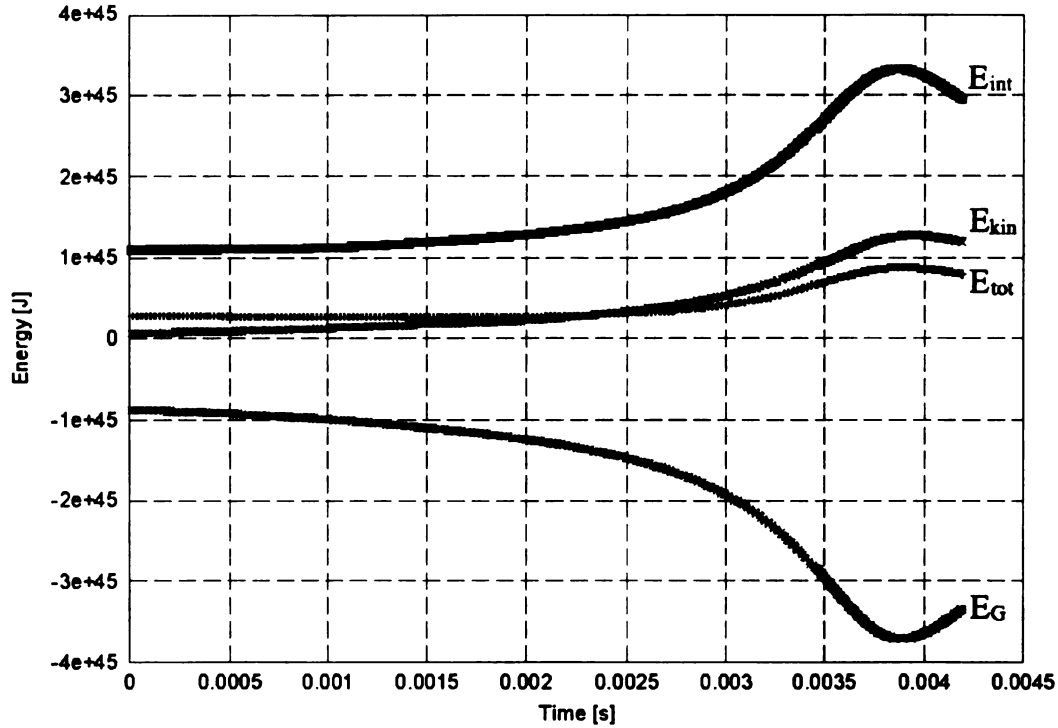


Figure 4.7: Time development of the different energies in simulation HLSEOS13.

Deviations from Spherical Symmetry

Figure 4.8 shows the development of the mass density distribution during the infall phase till shortly after bounce for models HLSEOS07 (slowly rotating), HLSEOS16 (moderately rotating), and HLSEOS22 (rapidly rotating).

The apparently different sizes of the density profiles are due to the grid scaling described in 3.2.2 (the same radius is used in all plots in this figure): the abrupt change between the light yellow and the white regions is located at the radial edge of the grid. Again, the oblate shape of the more rapidly rotating models is obvious in (b) and (c). A “vortex” formation along the z-axis during the infall phase is not identifiable in any of the models. It is obvious that core bounce gets more diffuse with increasing rotation (d). A slightly prolate shape shortly after bounce is apparent in

Tab
the
den
end

mo
alo
thi

	HLSEOS07	HLSEOS16	HLSEOS22
t_a	0	0	0
t_b	2.00	2.00	2.00
t_c	3.00	3.00	3.00
t_d	3.61	4.03	4.31
t_e	3.80	4.51	5.33
$\rho_{sc.min}$	5.13×10^{10}	5.30×10^{10}	5.30×10^{10}
$\rho_{sc.max}$	6.15×10^{14}	1.48×10^{14}	5.93×10^{13}

Table 4.6: Key for figure 4.8. t_a through t_e are the times (in ms) corresponding to the density profiles labeled (a) through (e) in the figure. $\rho_{sc.min}$ and $\rho_{sc.max}$ are the densities (in g/cm³) corresponding to the bottom end (color: light yellow) and top end (color: black) of the density key.

model HLSEOS07. In the very rapidly rotating model HLSEOS22 a density depletion along the z-axis (vertical in the plot) can be seen shortly after bounce (figure 4.9 shows this in a magnified way).

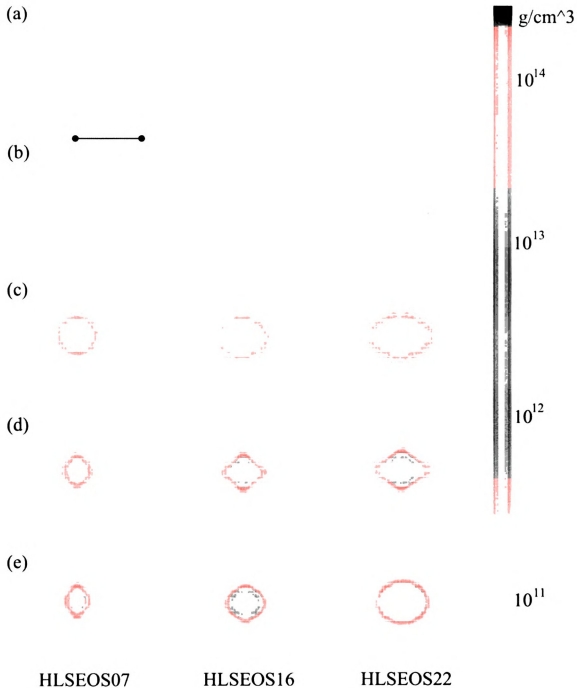


Figure 4.8: Mass density in a slice in the x-z-plane at five different times for models HLSEOS07 (left), HLSEOS16 (center), and HLSEOS22 (right). Events at the respective times: (a) onset of simulation, (b) after 2 ms, (c) presence of centrifugal forces becomes apparent (after 3 ms), (d) core bounce, (e) shortly after core bounce. The density scale is only approximately valid: the highest density (indicated by the colors black and dark red) decreases from left to right. All plots have the same radius (123.42 km) indicated by the black line in the top left. See table 4.6 for more data. Images in this thesis are presented in color.

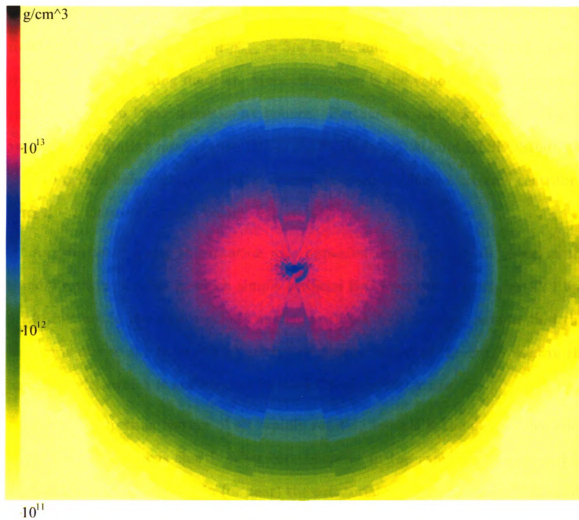


Figure 4.9: Density depletion along the z -axis after bounce in model HLSEOS22 at $t = 5.21\text{ms}$. The radius in this plot (measured from the center horizontally to the right) is 51.43km . Images in this thesis are presented in color.

4.3 Possible Implications of these Results

Our simulation succeeds in reproducing several results of previous ones which used different techniques of modeling rotating core collapse: we can confirm that core bounce occurs later the faster the core is rotating initially. Also the ballpark for the maximum density at bounce of $\rho_{max} \approx 3\rho_0$ is substantiated by our results (obtained from the slowly rotating models). More interesting is the validation that increasing rotation leads to lower densities at bounce. Further, we can acknowledge that in very rapidly rotating models (like HLSEOS220, HLSEOS190, and HLSEOS160) core bounce occurs before nuclear matter density is reached (due to extreme centrifugal forces) – a feature observed before (e.g. in [59]).

As “vortices” along the rotation axis appeared during the infall phase in the rapidly rotating models in which simple artificial EsOS were used (sections 4.2.1 and 4.2.2) but not in the most realistic model (section 4.2.3), one may assume that this feature is due to the (unrealistic) attractive density region of the EsOS used in the former models which is not existent in the EOS of the latter one.

A very interesting feature of all rapidly rotating models is the density depletion along the rotation axis shortly after core bounce. In the rapidly rotating models, most of the mass is shot out in (or near) the equatorial plane after bounce³. A highly speculative conclusion out of this is that the shockwave created after core bounce might be much more vigorous in the equatorial plane. Thus a prompt shock might succeed there: the star might explode in the equatorial plane first ejecting a ring of matter in this plane.

The density depletion along the rotation axis is extremely relevant for neutrino

³It is nearly impossible to see this in figures 4.3 and 4.8. But in the movies showing the full motion of the density distributions it is a quite obvious feature.

transport. Even though neutrinos are not dealt with in our simulation at all, it is evident that they would be able to escape from the central core much easier by diffusing through the low density region along the rotation axis. The peculiar concave shape of this low density region might actually “focus” a beam of neutrinos along the rotation axis creating such a large neutrino flux there that neutrino heating could lead to a jet-like explosion along this axis.

However, a lot more work needs to be done to underpin these speculations. Of particular importance is the implementation of collisions between individual test particles in our model or some other way that actually yields the formation of a shockwave after bounce. The inclusion of neutrino transport (at least in some simplified parameterized way) in our model is also most desirable if serious conclusions about the effects of neutrinos shall be drawn. After bounce our simple definitions of the temperature and electron fraction as functions of the mass density break down – a more sophisticated way of mapping data obtained from lower dimensional simulations on our model is needed to remedy this.

Both phenomena we just suggested (the prompt explosion in the equatorial plane and the delayed jet-like explosion along the rotation axis) are possible explanations for the polarization observations by Wang and Wheeler [50, 49, 51] because it is known that the light scattered from or through asymmetric surfaces is likely to be polarized (see [50] for more detail).

Chapter 5

Summary and Conclusion

First of all, the currently favored explosion mechanism of core collapse supernovae was described. It was made clear that this mechanism is complicated and could not be verified with great certainty so far since several different numerical simulations of supernovae performed by different groups all over the world yielded both successful explosions and failures. These deviating results are naturally due to uncertainties in the input physics (e.g. neutrino transport, convection) and to different numerical techniques used in the respective works.

The possible importance of rotation for the explosion mechanism was pointed out. The most relevant indication for the breakdown of spherical symmetry is the strong polarization of the light emitted by core collapse supernovae. Rotation is a very appealing candidate to explain the large deviations from spherical symmetry necessary to create such polarizations. This possible importance of rotation clearly motivates a three dimensional simulation.

Thus, a simplified three dimensional simulation we introduced to simulate the collapse of a supernova core during the infall phase until shortly after core bounce was presented in detail. In this model the core's mass is represented by discrete test particles. A spherical coordinate grid is used to locally define thermodynamic quanti-

ties. Input from other simulations is used to reduce the computational requirements of this simulation program to a minimum. The equations of motion for the test particles were derived making use of the assumption of approximate energy conservation during the infall phase. This assumption is justified by the short time scale of collapse which disables the neutrinos to carry energy out of the core in large amounts. A weakness of our technique (at the current state of the art) is the lack of collisions between the test particles which makes the formation of a shock wave after bounce impossible.

Finally, our simulation program was used with different EsOS and initial conditions to study the impact of rotation. Several (mostly predictable) effects of rotation found in previous works could be validated. Two interesting new features appeared in some of our rapidly rotating models:

1. A slight density depletion along the z-axis shortly after core bounce occurred in all rapidly rotating models independent of the EOS used. At the same time significantly more mass rebounded from the core in the equatorial plane than along the z-axis.
2. During the late stages of the infall phase the formation of “vortices” along the rotation axis was observed. However, this feature was extremely weak (hardly visible) in the model using the most realistic EOS.

Careful conclusions about the possible effects these features may have for the explosion mechanism were drawn. We suggested that the explosion might be initiated in the equatorial plane and along the rotation axis. This would deliver a possible explanation for the light polarization. However, it was noted that several improvements (such as the implementation of neutrino transport) should be made in our model to assert the validity of these new features.

To conclude, let us state that the supernova problem remains unsolved. Great theoretical challenges still lie ahead. It seems highly unlikely that computers powerful enough to make a three dimensional simulation including realistic neutrino transport and convection possible will become available in the near future. But probably not before then will there be certainty about the explosion mechanism. If nothing else, the present work has shown that the possible impact of rotation may have been underestimated for a long time.

APPENDIX

Appendix A

Source Code of the Simulation Program

In this appendix, the source code used for the simulation program which is written in C++ will be reproduced.

A.1 suno.cpp and suno.h

These two files contain all main parts of the simulation C++ code. Most important is the large C++ class **Star** which contains the memory structures and functions that deliver all information about the supernova core. The access functions for the **FORTRAN** routines for the EsOS are members of **Star**. Such trivial things as physical constants and functions for unit conversions are also declared and defined in these files. The whole code is extensively commented.

```
suno.h:

////////////////////////////////////
// suno.h
// Header file for suno.cpp
////////////////////////////////////

#ifndef SUNO_H
#define SUNO_H
```

```

#include "fortran.h"
#include <stdio.h>
#include <iostream.h>
#include <time.h>
#include <stdlib.h>
#include <math.h>
#include <fstream.h>
#include <vector>
#include "vector_and_spherical.h"
#include "testparticle.h"
#include <string>
#include <strstream>

const double c = 299792458; // global constant: speed of light [m/s]
const double G = 6.67259e-11; // gl. const.: gravitation constant [m^3/kg/s^2]
const double K_B = 1.3807e-23; // gl. const.: Boltzmann constant [J/K]
const double MASS_NEUTRON = 1.6749286e-27; //kg
const double HBAR = 6.6262e-34 / 2.0 / PI; // global constant: h bar [Js]
//const double G = 0; // "switch off" gravity

// declarations of global functions
int compare_tp_dist(const void* a, const void* b);

// functions for unit conversions
double mev_joule(double in_mev);
double joule_mev(double in_joule);
double mev_kelvin(double in_mev);
double kelvin_mev(double in_kelvin);
double baryonperfm3_kgperm3(double in_bpf3);
double kgperm3_baryonperfm3(double in_kpm3);

// FORTRAN-function for electron gas equation of state
extern "C" void FORTRAN_NAME(helmholtzeos)(const double* xmass, const
double* aion, const double* zion, const integer* ionmax, const double*
temp, const double* den, const double* energs, const double*
pressure);

// FORTRAN-function: initialization stuff for Lattimer & Swesty EOS
extern "C" void FORTRAN_NAME(loadmx)();

// FORTRAN-function for L&S EOS
extern "C" void FORTRAN_NAME(lseos)(const double* ipvar, const double* t_old,
const double* y_e, const double* brydns, const integer* iflag,
const integer* eosflg, const integer* forflg, const integer* sf,
const double* xprev, const double* pprev, const double* ls_out);

// forward declarations
class Star;
class TMap;
class TPDoubeMap;
#endif

```

suno.cpp:

```

#include "suno.h"

// global functions for unit conversions
double mev_joule(double in_mev){
    return in_mev * 1.602e-13;}

double joule_mev(double in_joule){
    return in_joule / 1.602e-13;}

double mev_kelvin(double in_mev){
    return mev_joule(in_mev)/K_B;}

```

```

double kelvin_mev(double in_kelvin){
    return joule_mev(in_kelvin*K_B);}

double baryonperfm3_kgperm3(double in_bpf3){
    return MASS_NEUTRON*1e45*in_bpf3;}

double kgperm3_baryonperfm3(double in_kpm3){
    return in_kpm3/MASS_NEUTRON*1e-45;}

////////////////////////////////////
// class from which class TPDoubeMap is derived
////////////////////////////////////
class TMap
{
protected:
    int i_r_max, i_phi_max, i_cos_theta_max;
    double r_max; // largest r coordinate of all partilcles to be saved
public:
    bool SetIRMax(int nmax)
    {
        if(nmax>0)
        {
            i_r_max=nmax;
            return false;
        }
        else
        {
            i_r_max=0;
            cout << "Error in TMap::SetIRMax(): i_r_max not positive.\n";
            return true;
        }
    }
    bool SetIPhiMax(int nmax)
    {
        if(nmax>0)
        {
            i_phi_max=nmax;
            return false;
        }
        else
        {
            i_phi_max=0;
            cout << "Error in TMap::SetIPhiMax(): i_phi_max not positive.\n";
            return true;
        }
    }
    bool SetICosThetaMax(int nmax)
    {
        if(nmax>0)
        {
            i_cos_theta_max=nmax;
            return false;
        }
        else
        {
            i_cos_theta_max=0;
            cout << "Error in TMap::SetICosThetaMax():
i_cos_theta_max not positive.\n";
            return true;
        }
    }
    bool SetRMax(double nr_max)
    {
        if(nr_max>=0)
        {
            r_max = nr_max;
            return false;
        }
    }
}

```

```

        }
        else
        {
            r_max=1;
            cout << "Error in TMap::SetRMax(): r_max
not positive.\n";
            return true;
        }
    }
    // access functions (contd..)
    int GetIRMax()
    {
        return i_r_max;
    }
    int GetIPhiMax()
    {
        return i_phi_max;
    }
    int GetICosThetaMax()
    {
        return i_cos_theta_max;
    }
    double GetRMax()
    {
        return r_max;
    }
    //virtual void SetAll2Zero()=0;

    // calculates the volume of one box
    double VolOfBox(int i_r);

    // given a point as a Vector object, calculates
    // i_r index of the box the point is in
    int GetIR(Spherical);

    // given a point as a Vector object, calculates
    // i_phi index of the box the point is in
    int GetIPhi(Spherical);

    // given a point as a Vector object, calculates
    // i_cos_theta index of the box the point is in
    int GetICosTheta(Spherical);

    // function defining box length in r direction
    double F_R(double rorm);

    // inverse function of F_R()
    double F_R_Inv(double iroirm);
};

/*****\
* given a point as a Spherical object, calculates
* i_r index of the box the point is in.
* i_r may be greater than i_r_max.
* Gets: - Spherical spos
* Returns: - i_r index of box as int
*****/
int TMap::GetIR(Spherical spos)
{
    int i_r;

    i_r = (int)((i_r_max) * F_R(spos.GetR()/r_max));
    // makes sure that i_r<i_r_max (appears obsolete)
    //i_r = i_r < i_r_max ? i_r : i_r_max - 1;
    // changed in order to get zero potetial gradients
    // for distant particles

    return i_r;
}

```

```

}

/*****\
* given a point as a Spherical object, calculates
* i_phi index of the box the point is in
* Gets: - Spherical spos
* Returns: - i_phi index of box as int
*****/
int TMap::GetIPhi(Spherical spos)
{
    int i_phi;

    i_phi = (int)((i_phi_max) * spos.GetPhi()/2.0/PI);
    // makes sure that i_phi < i_phi_max (appears obsolete)
    i_phi = i_phi < i_phi_max ? i_phi : i_phi_max - 1;

    return i_phi;
}

/*****\
* given a point as a Spherical object, calculates
* i_cos_theta index of the box the point is in
* Gets: - Spherical spos
* Returns: - i_cos_theta index of box as int
*****/
int TMap::GetICosTheta(Spherical spos)
{
    int i_cos_theta;

    i_cos_theta = (int)((i_cos_theta_max) * (cos(spos.GetTheta()+1)/2.0);
    i_cos_theta = i_cos_theta < i_cos_theta_max ?
        i_cos_theta : i_cos_theta_max-1;

    return i_cos_theta;
}

/*****\
* Function defining box length in r direction by the
* relation F_R(r/r_max) = i_r/i_r_max
* for future experiments
* Gets: - double r/r_max
* Returns: - double F_R(r/r_max)
*****/
double TMap::F_R(double rorm)
{
    return rorm;
    //return pow(rorm,0.7);
    //return pow(rorm,0.3);
}

/*****\
* Inverse Function of F_R
*****/
double TMap::F_R_Inv(double iroirm)
{
    return iroirm;
    //return pow(iroirm,1.0/0.7);
    //return pow(iroirm,1.0/0.3);
}

/*****\
* Calculates the volume of one box.
* This obviously depends on r respectively i_r.
* Gets: - i_r
* Returns: - Volume of all boxes on the shell labeled
*           by i_r:
*           Vol=Vol(shell)/(i_phi_max*i_cos_theta_max)
*****/

```



```

double TMap::VolOfBox(int i_r)
{
    double vol;
    double r0, r1; // radius of inner/outer shell boundary

    // calculate inner/outer radii
    r0 = r_max * F_R_Inv((double)i_r/(double)i_r_max);
    r1 = r_max * F_R_Inv((double)(i_r+1)/(double)i_r_max);

    // volume of shell divided by no. of boxes
    vol = 4.0*PI/3.0*(pow(r1,3)-pow(r0,3)) / i_phi_max / i_cos_theta_max;

    return vol;
}

////////////////////////////////////
// class for storing mass density of test particles or (smeared)
// particle numbers in certain boxes
////////////////////////////////////
class TPDoublMap:public TMap
{
private:
    double*** valueinbox; // " the mass density in the boxes
public:
    // constructor 1
    TPDoublMap()
    {
        SetIRMax(1);
        SetIPhiMax(1);
        SetICosThetaMax(1);
    }

    // constructor 2
    TPDoublMap(int ni_r_max, int ni_phi_max, int ni_cos_theta_max)
    {
        ReserveMemory(ni_r_max, ni_phi_max, ni_cos_theta_max);
    }

    // destructor
    ~TPDoublMap()
    {
        int i=0, j=0;
        // free memory
        for(i=0; i<i_r_max; i++)
            for(j=0; j<i_phi_max; j++)
                delete[] valueinbox[i][j];
        for(i=0; i<i_r_max; i++)
            delete[] valueinbox[i];
        delete[] valueinbox;
    }

    // reserves memory for the array containing the densities
    void ReserveMemory(int ni_r_max, int ni_phi_max, int ni_cos_theta_max);

    // sets test particle mass density in box (i_r, i_phi, i_cos_theta)
    void SetValueInBox(int i_r, int i_phi, int i_cos_theta, double dens)
    {
        valueinbox[i_r][i_phi][i_cos_theta] = dens;
        return;
    }

    // returns mass density in box (i_r, i_phi, i_cos_theta)
    double GetValueInBox(int i_r, int i_phi, int i_cos_theta)
    {
        return valueinbox[i_r][i_phi][i_cos_theta];
    }

    // sets all entries in tpinbox[][] to 0
    void SetAll2Zero()

```

```

{
    int i=0,j=0,k=0;
    for(i=0; i<i_r_max; i++)
        for(j=0; j<i_phi_max; j++)
            for(k=0; k<i_cos_theta_max; k++)
                valueinbox[i][j][k]=0;
}

// adds 1.0 to valueinbox in the box in which the
// TestParticle tp is located
void AddTP(Spherical spos);

// adds a total of 1.0 to the box in which the test particle is located
// and the neighboring boxes depending on the distance between the
// test particle's position and the boundaries of the box
void AddTPSmear(Spherical spos);

// function needed by AddTPSmear
double Smear(double x, double x_0, double x_bound);

// sets values in boxes (i_r, i_phi, i_cos_theta) and
// (" ", i_cos_theta_max-1 - i_cos_theta) to the common average
void AverageCosThetaBoxes();
};

/*****\
* Function of x: 0.5 if x==x_bound, 0 if
* x==x_c, linearly interpolated in between.
* Gets: - doubles x, x_c, x_bound
* Returns: - value as described above
\*****/
double TPDoubleMap::Smear(double x, double x_c, double x_bound)
{
    return 0.5*(x-x_c)/(x_bound-x_c);
}

/*****\
* Smears a test particle that is located in a certain
* box of the TPDoubleMap over the eight closest
* neighboring boxes. The part of the test particle (i.e.
* a fraction of 1.0) distributed to each box depends
* linearly on the distance between the particle and
* the boundary of the box (in spherical coord's).
* For example: the box in which the test particle is
* located gets at least 1/8 if the particle is on a
* corner point of the box and 1.0 if the
* particle is in its very center.
* Gets: - Spherical spos (location of test particle)
* Returns: nothing
\*****/
void TPDoubleMap::AddTPSmear(Spherical spos)
{
    // indices of the box in which tp is located
    int i_r, i_phi, i_cos_theta;
    // indices of box whose value is to be modified (see below)
    int i_r_mod, i_phi_mod, i_cos_theta_mod;
    int k,m,n; // loop counters
    double r,phi,theta; // tp's position in spherical coord's
    double r_c, phi_c, theta_c; // center of tp's box in s/c
    double r_s, phi_s, theta_s; // lower boundaries of box
    double r_b, phi_b, theta_b; // upper bounds of box

    // don't add particle if out of bounds of map
    if(spos.GetR()>r_max)return;

    // shorter names for variables...
    r = spos.GetR();
    phi = spos.GetPhi();

```

```

    theta = spos.GetTheta();

    // calculate the indices of the box in which
    // the test particle is located
    i_r = GetIR(spos);
    i_phi = GetIPhi(spos);
    i_cos_theta = GetICosTheta(spos);

    // calculates boundaries of box in s/c
    r_b = r_max * F_R_Inv((double)(i_r+1) / i_r_max);
    r_s = r_max * F_R_Inv((double)i_r / i_r_max);

    phi_b = 2*PI* (double)(i_phi+1)/i_phi_max;
    phi_s = 2*PI* (double)i_phi/i_phi_max;

    theta_b = acos(2.0*(double)i_cos_theta/i_cos_theta_max-1.0);
    theta_s = acos(2.0*(double)(i_cos_theta+1)/i_cos_theta_max-1.0);

    // calculates center of box in s/c
    r_c = 0.5*(r_b+r_s);
    phi_c = 0.5*(phi_b+phi_s);
    theta_c = 0.5*(theta_b+theta_s);

    // actually smears test particle over 8 closest neighboring boxes
    for(k=0; k<2; k++)
    {
        i_r_mod = i_r+k*(r>r_c?1:-1);
        // makes sure that i_r_mod >=0 and <i_r_max
        i_r_mod = i_r_mod<0?0:(i_r_mod<i_r_max?i_r_mod:i_r_max-1);
        r_b = r>r_c?r_b:r_s;
        for(m=0; m<2; m++)
        {
            i_phi_mod = i_phi+m*(phi>phi_c?1:-1);
            // makes sure that i_phi_mod >=0 and <i_phi_max
            i_phi_mod = i_phi_mod<0?i_phi_max-1:
                (i_phi_mod<i_phi_max?i_phi_mod:0);
            phi_b = phi>phi_c?phi_b:phi_s;
            for(n=0; n<2; n++)
            {
                i_cos_theta_mod = i_cos_theta+n*(theta>theta_c?1:-1);
                // makes sure that i_cos_theta_mod >=0
                // and <i_cos_theta_max
                i_cos_theta_mod = i_cos_theta_mod<0?0:
                    (i_cos_theta_mod<i_cos_theta_max?
                        i_cos_theta_mod:i_cos_theta_max-1);
                theta_b = theta>theta_c?theta_b:theta_s;

                valueinbox[i_r_mod][i_phi_mod][i_cos_theta_mod]+=
                    (1.0-k*(k==0?-1:1)*Smear(r,r_c,r_b))*
                    (1.0-m*(m==0?-1:1)*Smear(phi,phi_c,phi_b))*
                    (1.0-n*(n==0?-1:1)*Smear(theta,theta_c,theta_b));
            }
        }
    }

    return;
}

/*****\
* Adds 1.0 to valueinbox in the box in which
* the test particle at position spos is located.
* (valueinbox = #test particles)
* Gets: - Spherical spos (location of test particle)
* Returns: - nothing
\*****/
void TPDoubleMap::AddTP(Spherical spos)
{
    // indices of the box in which tp is located

```

```

    int i_r, i_phi, i_cos_theta;

    // don't add particle if out of bounds of map
    if(spos.GetR()>r_max)return;

    // calculate the indices of the box in
    // which the test particle is located
    i_r = GetIR(spos);
    i_phi = GetIPhi(spos);
    i_cos_theta = GetICosTheta(spos);

    // actually adds the test particle
    valueinbox[i_r][i_phi][i_cos_theta]++;
    return;
}

/*****\
* Reserves memory for the requested number
* of boxes and sets all entries in valueinbox[][] to
* zero.
* Gets: - three ints indicating the number of
*         separations in the three coordinates
* Returns: nothing
\*****/
void TPDouMap::ReserveMemory(int ni_r_max, int ni_phi_max,
                             int ni_cos_theta_max)
{
    int i=0, j=0;

    // memory reservation
    if(SetIRMax(ni_r_max) || SetIPhiMax(ni_phi_max) ||
       SetICosThetaMax(ni_cos_theta_max))
    {
        return;
    }
    else
    {
        valueinbox = new double**[ni_r_max];
        for(i=0; i<ni_r_max; i++)
            valueinbox[i] = new double*[ni_phi_max];
        for(i=0; i<ni_r_max; i++)
            for(j=0; j<ni_phi_max; j++)
                valueinbox[i][j] = new double[ni_cos_theta_max];

        SetAll12Zero();
    }
}

/*****\
* Sets values in boxes (i_r, i_phi, i_cos_theta) and
* (~,~, i_cos_theta_max-1 - i_cos_theta) to their
* common average. Sensible if mirror symmetry about
* the equatorial plane is assumed.
* Gets: nothing
* Returns: nothing
\*****/
void TPDouMap::AverageCosThetaBoxes()
{
    double av_val=0; // average value for current pair of boxes

    // average all the boxes...
    for(int i=0; i<ni_r_max; i++)
        for(int j=0; j<ni_phi_max; j++)
            for(int k=0; k<ni_cos_theta_max/2; k++)
            {
                av_val = 0.5 * (valueinbox[i][j][k] +
                               valueinbox[i][j][ni_cos_theta_max-1-k]);
                valueinbox[i][j][k] = av_val;
            }
}

```

```

        valueinbox[i][j][i_cos_theta_max-1-k] = av_val;
    }

    return;
}

////////////////////////////////////
// main class of program
////////////////////////////////////
class Star
{
private:
    TestParticle* tp;
    int number_tp;
    double radius;          // radius of star [m]
    double m_star; // mass of star [kg]
    double m_tp; // mass of test particle [kg] (==m_star/number_tp)
    double A; // parameter in nuclear eqn. of state (EOS): -3.49236e-11 J
    double SIGMA; // parameter in nuclear EOS: 4.0/3.0
    double B; // parameter in nuclear EOS: 2.62728e-11 J
    double DENS_MIN; // minimum density enforced
    // tp_id[i] is the ID of test particle i
    int* tp_id;
    // id_tp[i] is the test particle number of the t/p with ID i
    int* id_tp;
    // test particle mass density map (using a spherical coordinate grid)
    TPDoubeMap dmap;
    // potential map, potential due to EOS
    TPDoubeMap potmap;
    //flag indicating if thermalization is activated
    bool flag_thermalize;
    // flag indicating averaging of mass density in cos(theta)-direction
    bool flag_average_theta;
    // map containing indices of particles located in each box
    vector<int>*** box_tp;
    // input variables for LS EOS, saved for faster convergence of EOS
    double**** lseos_ipvar;
    // another input variable for LS EOS
    double*** lseos_pprev;
    // flag indicating if "energy method" is used instead of "force method"
    bool flag_energy_method;
    double time_passed; // time in seconds

public:
    // constructor
    Star(int number, double radius, double mass, int i_r_max,
        int i_phi_max, int i_cos_theta_max, bool thermalize,
        bool average_theta, double newdens_min, bool energy_method)
    {
        number_tp = number;
        // reserve memory for test particles
        tp = new TestParticle [number_tp];
        tp_id = new int [number_tp];
        id_tp = new int [number_tp];
        // set test particle IDs
        for(int i=0; i<number_tp; i++)
        {
            tp[i].SetID(i);
        }
        SetRadius(radius);
        SetMass(mass);

        RefreshTP();

        dmap.ReserveMemory(i_r_max, i_phi_max, i_cos_theta_max);
        potmap.ReserveMemory(i_r_max, i_phi_max, i_cos_theta_max);

        // reserves mem for box_tp only if thermalization is activated

```

```

        if((flag_thermalize==thermalize) == true)
            ReserveMemory4BoxTP(i_r_max, i_phi_max, i_cos_theta_max);

        // reserves mem for lseos_ipvar and lseos_pprev
        ReserveMemory4LSEOSInput(i_r_max, i_phi_max, i_cos_theta_max);

        flag_average_theta = average_theta;

        // sets parameters for EOS (old)
        A=-3.49236e-11;
        SIGMA=4.0/3.0;
        B=2.62728e-11;
        DENS_MIN = newdens_min;

        // sets flag showing if energy or force method is used
        flag_energy_method = energy_method;

        time_passed = 0;
    }

    // destructor
    ~Star()
    {
        // free memory reserved for test particles
        delete[] tp;
        delete[] tp_id;
        delete[] id_tp;

        // delete box_tp array if thermalization is activated
        if(flag_thermalize == true)
            DeleteBoxTP();

        // delete arrays lseos_ipvar and lseos_pprev
        DeleteLSEOSInput();
    }

    // refreshes tp_id and id_tp
    void RefreshTP();

    // sets radius of star
    void SetRadius(double newr)
    {
        radius = newr;
        return;
    }

    // sets the star's mass
    void SetMass(double newmass)
    {
        m_star = newmass;
        m_tp = m_star/number_tp;
        return;
    }

    // returns the mass of a test particle
    double GetMTP()
    {
        return m_tp;
    }

    // returns radius of star
    double GetRadius()
    {
        return radius;
    }

    // returns the number of test particles
    int GetNumberTP()

```

```

{
    return number_tp;
}

// returns test particle i
TestParticle GetTP(int i)
{
    if(i>=0 && i<number_tp)
        return tp[i];
    else{
        cout << "Error in class Star function GetTP():
Bad test particle number.\n";
        return tp[0];} // return garbage
    }

// sets test particle i
void SetTP(int i, TestParticle newtp)
{
    tp[i] = newtp;
}

// returns the test particle mass density map
TPDoubleMap GetTPDensityMap()
{
    return dmap;
}

// distributes test particles in position space
void DistributeTPPos(int no_shells);

// puts test particle i to a random position in a sphere of radius r
bool PutTPInSphere(int i, double r);

// mass density as a function of distance from
// the center (initial condition)
double initial_mass_density(double r, double parameter_d);

// distributes test particles in momentum space imposing rigid
// body rotation
void DistributeTPMom(Vector omega, double r_0, bool diff_rot);

// determines no. of test particles with smaller distance
// to the origin than current t/p
int CountTPInside(int current_tp);

// calculates & returns the derivatives of position
// and momentum of particle i
TPChange Deriv(int i);

// calculates the next step in the star's time-development
// using the Euler method
void NextStepEuler(double stepsize);

// calculates the next step in the star's time-development
// using a 4th order Runge Kutta algorithm
double NextStepRK4(double h, bool modify_stepsize_avvel);

// calculates the current TPChange for all test particles
void CalculateK(TPChange* k, double dt);

// moves all test particles to *initial + kfactor * *k
void MoveTPs(TestParticle* initial, TPChange* k, double kfactor);

// performs a certain number of steps
void Stepper(double stepsize, int number_of_steps,
    int points_per_step_save, bool modify_stepsize,
    bool modify_stepsize_avvel);

```

```

// modifies stepsize used by Star::Stepper()
// (based on density distribution)
double ModifyStepsize(double dt, bool& stepsize_modified,
    bool& stepsize_modified_twice);

// modifies stepsize (based on averag velocity of t/ps)
double ModifyStepsizeAvVel(double dt, double av_disp);

// sorts the test particles by their distance from the origin
void SortTP();

// saves the coordinates of every (number_tp/100)th test particle
// in a file readable by SuNoOutput.exe created with Visual Basic
void SaveCoordinatesVB(int points_per_file);

// saves the test particle mass density in a slice through the star
void SaveTPMDensityVB();

// saves the whole data (positions and momenta) for one time step
void SaveAllData(double max_radius, int timestep);

// reads the whole data (positions and momenta) for use as
// initial cconditions
void ReadAllData();

// calculates the TPSmearedNumberMap for the current config of the star
void Star::CalculateTPSmearedNumberMap(TPDoubleMap& nmap);

// calculates the mass density map
void CalculateTPMDensityMap();

// calculates the potential map, potential due to EOS
void CalculateEOSPotentialMap();

// calculates the temperature map
void CalculateTemperatureMap();

// calculates the temperature in MeV as a function of the density
double CalculateTemperature(double rho);

// calculates the gradient of the EOS potential  $\nabla V$ 
// at the location of test particle i
Vector GetGradEOSPotential(int i);

// calculates the derivative of the EOS potential in the
// r-direction in a more sophisticated way
double CalculateInterpolatedDVDR(int i_r, int i_phi,
    int i_cos_theta, double r);

// calculates the derivative of the EOS potential in the
// phi direction in a more sophisticated way
double CalculateInterpolatedDVDPHI(int i_r, int i_phi,
    int i_cos_theta, double phi);

// calculates the derivative of the EOS potential in the
// "cos(theta)-direction" in a more sophisticated way
double CalculateInterpolatedDVDCosTheta(int i_r, int i_phi,
    int i_cos_theta, double costheta);

// returns the force on test particle i
Vector GetForceOnTP(int i);

// equation of state, returns potential in box i_r, i_phi, ...
double CalculateEOSPotential(int i_r, int i_phi, int i_cos_theta);

// returns the flag indicating if thermalization is activated
bool GetFlagThermalize(){
    return flag_thermalize;}

```



```

// randomizes momenta of test particles in box (i_r,i_phi,i_cos_theta)
void ThermalizeBox(int i_r, int i_phi, int i_cos_theta);

// reserves memory for the box_tp
void ReserveMemory4BoxTP(int i_r_max, int i_phi_max, int i_cos_theta_max)
{
    int i=0, j=0;
    box_tp = new vector<int>** [i_r_max];
    for(i=0; i<i_r_max; i++){
        box_tp[i] = new vector<int>* [i_phi_max];
        for(i=0; i<i_r_max; i++){
            for(j=0; j<i_phi_max; j++){
                box_tp[i][j] = new vector<int> [i_cos_theta_max];
            }
        }
    }
    return;
}

// reserves mem for lseos_ipvar and lseos_pprev, sets initial guesses
void ReserveMemory4LSEOSInput(int i_r_max, int i_phi_max,
    int i_cos_theta_max)
{
    lseos_pprev = new double** [i_r_max];
    lseos_ipvar = new double*** [i_r_max];
    for(int i=0; i<i_r_max; i++){
        lseos_pprev[i] = new double* [i_phi_max];
        lseos_ipvar[i] = new double** [i_phi_max];
    }
    for(int i=0; i<i_r_max; i++){
        for(int j=0; j<i_phi_max; j++){
            lseos_pprev[i][j] = new double [i_cos_theta_max];
            lseos_ipvar[i][j] = new double* [i_cos_theta_max];
        }
    }
    for(int i=0; i<i_r_max; i++){
        for(int j=0; j<i_phi_max; j++){
            for(int k=0; k<i_cos_theta_max; k++){
                lseos_ipvar[i][j][k] = new double [4];
            }
        }
    }

    double pprev = DENS_MIN/MASS_NEUTRON*1e-45 * 0.3;
    // set values for initial guesses
    for(int i=0; i<i_r_max; i++){
        for(int j=0; j<i_phi_max; j++){
            for(int k=0; k<i_cos_theta_max; k++){
                lseos_ipvar[i][j][k][1]=0.155;
                lseos_ipvar[i][j][k][2]=-15.0;
                lseos_ipvar[i][j][k][3]=-10.0;
                lseos_pprev[i][j][k]= pprev;
            }
        }
    }
    return;
}

// deletes box_tp array
void DeleteBoxTP()
{
    int i=0, j=0;
    for(i=0; i<dmap.GetIRMax(); i++){
        for(j=0; j<dmap.GetIPhiMax(); j++){
            delete[] box_tp[i][j];
        }
    }
    delete[] box_tp;
    return;
}

// deletes arrays lseos_ipvar and lseos_pprev
void DeleteLSEOSInput()
{
    for(int i=0; i<dmap.GetIRMax(); i++){
        for(int j=0; j<dmap.GetIPhiMax(); j++){
            for(int k=0; k<dmap.GetICosThetaMax(); k++){
                delete[] lseos_ipvar[i][j][k];
            }
        }
    }
    for(int i=0; i<dmap.GetIRMax(); i++){

```

```

        for(int j=0; j<dmap.GetIPhiMax(); j++){
            delete[] lseos_pprev[i][j];
            delete[] lseos_ipvar[i][j];}
    for(int i=0; i<dmap.GetIRMax(); i++){
        delete[] lseos_pprev[i];
        delete[] lseos_ipvar[i];}
    delete[] lseos_pprev;
    delete[] lseos_ipvar;
    return;
}

// deletes all entries in box_tp array
void ClearBoxTP(){
    int i,j,k;
    for(i=0; i<dmap.GetIRMax(); i++)
        for(j=0; j<dmap.GetIPhiMax(); j++)
            for(k=0; k<dmap.GetICosThetaMax(); k++)
                box_tp[i][j][k].clear();

    return;}

// measures the total angular momentum of the star
Vector MeasureTotalAngularMomentum();

// measures the kinetic energy of the star
double MeasureKineticEnergy(bool relativistic);

// measures the gravitational energy of the star
double MeasureGravitationalEnergy();

// measures the internal energy of the star
double MeasureInternalEnergy();

// internal energy per baryon via Helmholtz EOS from Frank Timmes
double CalculateInternalEnergyHelmholtz(const double* xmass,
    const double* aion, const double* zion, integer ionmax,
    double temp, double den, const double* pressure);

// internal energy (and other thermodynamic qutts.)
// via Lattimer & Svesty EOS
double CalculateLSEOS(double* ipvar, double y_e, double density,
    double* pprev, const double* ls_out);

// modifies the momenta of all t/ps conserving the total energy
void ModifyMomentaConservingEnergy();

// sets the initial internal energies for the t/ps
void InitializeInternalEnergies();

// gives t/ps a momentum towards the center of the core
void KickTPInside(double max_velocity);

// calculates and returns electron fraction as function of density
double CalculateYE(double rho);
};

/*****\
* Calculates and returns electron fraction as function of
* density using the values from cooperstein.
* This is a good approximation for
* (and only for) the infall phase till core-bounce. The
* data is taken from Cooperstein & Wambach, Nucl. Phys. A
* 420, p.591 , 1984 and Cooperstein, Nucl. Phys. A 438,
* p.722, 1985 and was approximated by analytic expressions
* using the gnuplot fit function.
* Gets: - mass density rho in kg/m^3 as double
* Returns: - electron fraction as double
*****/
double Star::CalculateYE(double rho)

```

```

{
    double dens = rho/1e14;
    return 0.921883/(8.02162+dens)+0.307112;
}

/*****\
* Calculates and returns the internal energy using the
* EOS by Lattimer and Svesty which includes a nuclear
* contributions and is thus good at high densities.
* FORTRAN_NAME(loadmx)() must be called before this
* function can be used. Parameters from bound.atb and
* maxwel.atb are used (see EOS documentation for details),
* so these files have to be in the directory of suno.exe.
* Gets: - inputvariables ipvar as double array.
*       ipvar[0] == temperature in MeV
*       ipvar[1] == guess at nuclear density in fm^-3
*       ipvar[2] == guess at proton eta
*       ipvar[3] == guess at neutron eta
*       (save old values of these box for every new call)
*       - electron fraction y_e as double
*       - baryon mass density (in kg/m^3) dens as double
*       - pprev as double reference, this is the guess at
*       the exterior proton fraction
*       - ls_out[4] as double array. Here pressure[MeV/fm^3],
*       internal energy/baryon[MeV], entropy/baryon [k_B]
*       and free energy/baryon [?] are stored (in this order).
* Returns: - internal energy per baryon [J] as double
* Note: The LS EOS is able to calculate a LOT more thermodynamic
* quantities than just these.
\*****/
double Star::CalculateLSEOS(double* ipvar, double y_e, double density,
    double* pprev, const double* ls_out)
{
    static bool firstcall=true;
    // initial guess for temperature (only used if
    //temperature is not input variable)
    static double t_old;
    // needed by LS EOS, can be ignored
    static double xprev;
    // indicates ipvar[0]: 1=temperature, 2=internal energy, 3=entropy
    static integer iflag=1;
    // indicates which EOS was used
    static integer eosflg;
    // forces use of EOS indicated by eosflg if set to 1 (don't...)
    static integer forflg=0;
    // sf=1 means successful EOS call, everything else not
    static integer sf;
    // conversion kg/m^3 --> fm^-3
    double dens = kgperm3_baryonperfm3(density);

    // initialization if this is first call of function
    if(firstcall)
    {
        FORTRAN_NAME(loadmx)();
        firstcall = false;
    }

    // check if density is ok, set initial guesses for input
    // variables if not
    if(density > 1e17){
        ipvar[1]=0.155;
        ipvar[2]=-15.0;
        ipvar[3]=-10.0;
        *pprev=DENS_MIN/MASS_NEUTRON*1e-45 * 0.3;
    }

    FORTRAN_NAME(lseos)(ipvar, &t_old, &y_e, &dens, &iflag, &eosflg,
        &forflg, &sf, &xprev, pprev, ls_out);
}

```

```

        return mev_joule(ls_out[1]); // conversion MeV --> J
    }

/*****\
* Calculates and returns the internal energy using the
* Helmholtz EOS from Frank Timmes which does NOT include
* a nuclear contribution.
* Gets: - ionmax as integer. This is the number of
*        different ion types in the matter
*        - xmass as double array containing the mass
*          fractions of the different ions
*        - aion as double array containing the mass numbers
*          of the ions (=number of nucleons)
*        - zion as double array containing the charge
*          numbers of the ions (=number of protons)
*        - temperature temp in Kelvin as double
*        - mass density den in kg/m^3 as double
* Returns: - specific internal energy [J/kg]
* Note: - the limits of the EOS are:
*         1e4 <= temp <= 1e11, 1e-7 <= y_e*den <= 1e14
*         - the Helmholtz EOS is able to calculate a LOT
*         more thermodynamic quantities than just this
\*****/
double Star::CalculateInternalEnergyHelmholtz(const double* xmass,
        const double* aion, const double* zion, integer ionmax,
        double temp, double den, const double* pressure)
{
    double energ=0;

    // check if temperature and density are ok
    if(den < 1e-7 || den > 1e14 || temp < 1e4 || temp > 1e11)
    {
        cout << "Error in Star::CalculateInternalEnergyHelmholtz():
bad value for density or temperature.\n";
    }

    // convert den to g/cm^3
    den *= 1e-3;

    // calls the Helmholtz EOS, saves specific internal energy in energ
    FORTRAN_NAME(helmholtzeos)(xmass,aion,zion,&ionmax,&temp,&den,
        &energ,pressure);

    // convert erg/g to J/kg
    energ*=1e-4;

    return energ;
}

/*****\
* Calculates and returns the temperature as a function
* of the mass density. This is a good approximation for
* (and only for) the infall phase till core-bounce. The
* data is taken from Cooperstein & Wambach, Nucl. Phys. A
* 420, p.591, 1984 and Cooperstein, Nucl. Phys. A 438,
* p.722, 1985 and was approximated by analytic expressions
* using the gnuplot fit function.
* Gets: - mass density rho in kg/m^3 as double
* Returns: - temperature in K as double
\*****/
double Star::CalculateTemperature(double rho)
{
    double t=0;

    // converts rho from kg/m^3 to 1e11 g/cm^3
    rho*=1e-14;

```

```

        if(rho<=1e1)
            t=1.48962*pow(rho,0.182321)-0.280647*pow(rho,0.446682);
        else
            if(rho>1e1 && rho<=4e1) // interpolates between the two data sets
                t=(4e1-rho)/3e1*(1.48962*pow(rho,0.182321)
                    -0.280647*pow(rho,0.446682))
                    + (rho-1e1)/3e1*(0.148522*pow(rho,0.485484)+0.728965);
            else
                if(rho>4e1)
                    t=0.148522*pow(rho,0.485484)+0.728965;

        return mev_kelvin(t);
    }

    /*****\
    * Calculates and returns the total internal energy (i.e.
    * the energy due to the EOS) of the star.
    * Gets: - nothing
    * Returns: - total internal energy as double
    \*****/
    double Star::MeasureInternalEnergy()
    {
        double e_int=0;

        for(int i=0; i<potmap.GetIRMax(); i++)
            for(int j=0; j<potmap.GetIPhiMax(); j++)
                for(int k=0; k<potmap.GetICosThetaMax(); k++)
                {
                    e_int+=potmap.GetValueInBox(i,j,k)*
                        dmap.GetValueInBox(i,j,k)/m_tp*dmap.VolOfBox(i);
                    // could be simplified if numbermap was saved
                }
        /* // alternative method
            for(int i=0; i<number_tp; i++)
                e_int += tp[i].GetInternalEnergy();
        */

        return e_int;
    }

    /*****\
    * Calculates and returns the gravitational energy of the
    * star (using a crude approximation).
    * Gets: nothing
    * Returns: - gravitational energy as double
    \*****/
    double Star::MeasureGravitationalEnergy()
    {
        double e_grav=0;

        SortTP();
        for(int i=0; i<number_tp; i++)
            e_grav += -G*m_tp*i*m_tp/tp[i].GetPos().GetNorm();

        return e_grav;
    }

    /*****\
    * Calculates and returns the total kinetic energy of the
    * star. The flag relativistic indicates if calculation is
    * to be performed relativistically or not.
    * Gets: - relativistic as bool
    * Returns: - total kinetic energy as double
    \*****/
    double Star::MeasureKineticEnergy(bool relativistic)
    {
        double e_kin=0;

```

```

        if(relativistic)
        {
            for(int i=0; i<number_tp; i++)
                e_kin += sqrt(pow(m_tp,2)*pow(c,4)+
                    tp[i].GetMom().GetNormSquared()*pow(c,2))-m_tp*pow(c,2);
        }
        else
        {
            for(int i=0; i<number_tp; i++)
                e_kin+=tp[i].GetMom().GetNormSquared() / 2.0 / m_tp;
        }

        return e_kin;
    }

    /*****\
    * Calculates and returns the total angular momentum
    * vector of the star.
    * Gets: nothing
    * Returns: - angular momentum as Vector
    \*****/
    Vector Star::MeasureTotalAngularMomentum()
    {
        Vector angmom;

        for(int i=0; i<number_tp; i++)
            angmom = angmom + (tp[i].GetPos()*tp[i].GetMom());

        return angmom;
    }

    /*****\
    * Averages the motion of the test particles in box
    * (i_r, i_phi, i_cos_theta) not (!) conserving kinetic
    * energy, but consering angular momentum.
    * Only the projections of the momenta of the t/ps in the
    * radial direction are modified.
    * Gets: - subscripts of box
    * Returns: nothing
    \*****/
    void Star::ThermalizeBox(int i_r, int i_phi, int i_cos_theta)
    {
        Vector new_mom; // new momentum vector
        int n=0;
        int no_tp_box=0; // number of test particles in current box

        no_tp_box = box_tp[i_r][i_phi][i_cos_theta].size();

        if(no_tp_box)
        {
            Vector e_r; // unit vector in spherical coordinates
            double p_r=0; // projection of moentum on e_r

            for(n=0; n<no_tp_box; n++)
            {
                // calculates e_r for current test particle
                e_r = tp[box_tp[i_r][i_phi][i_cos_theta][n]].GetPos()
                    * (-1/tp[box_tp[i_r][i_phi][i_cos_theta][n]].GetPos().GetNorm());
                // calculates and adds p_r of the particles
                p_r+=tp[box_tp[i_r][i_phi][i_cos_theta][n]].GetMom()^e_r;
            }

            // calculates average p_r
            p_r /= no_tp_box;

            for(n=0; n<no_tp_box; n++)
            {
                e_r = tp[box_tp[i_r][i_phi][i_cos_theta][n]].GetPos()

```

```

        * (-1/tp[box_tp[i_r][i_phi][i_cos_theta][n]].GetPos().GetNorm());
new_mom = tp[box_tp[i_r][i_phi][i_cos_theta][n]].GetMom();
// subtracts old radial part
new_mom = new_mom - (new_mom*e_r)*e_r;
// adds new radial part
new_mom = new_mom + p_r*e_r;
tp[box_tp[i_r][i_phi][i_cos_theta][n]].SetMom(new_mom);
    }
}

return;
}

/*****\
* Calculates and returns the gradient of the potential
* due to the EOS at the position of test
* particle i as a Vector.
* TPDouMap potmap must be calculated before this works
* in a reasonable way.
* Gets: - int i identifying the test particle
* Returns: - gradient Vector
\*****/
Vector Star::GetGradEOSPotential(int i)
{
    // for the resulting gradient
    Vector grad;
    Spherical spos;
    // indices of the box test particle i is in
    int i_r=0, i_phi=0, i_cos_theta=0;
    //derivatives
    double dV_dr=0, dV_dphi=0, dV_dcstheta=0;
    // unit vectors for spherical coordinates
    Vector e_r, e_phi, e_theta;

    // converts the test particle's position vector to spherical coord's
    spos = tp[i].GetPos().GetSpherical();

    // locates test particle in TMap
    i_r = potmap.GetIR(spos);
    i_phi = potmap.GetIPhi(spos);
    i_cos_theta = potmap.GetICosTheta(spos);

    // calculates gradient only if test particle is in map,
    // i.e. i_r<i_r_max
    if(i_r<potmap.GetIRMax())
    {
        // calculates derivatives in spherical coordinates
        dV_dr = CalculateInterpolatedDVDR(i_r, i_phi,
            i_cos_theta, spos.GetR());
        //dV_dr = CalculateDVDR(i_r, i_phi, i_cos_theta);
        dV_dphi = CalculateInterpolatedDVDPHI(i_r, i_phi,
            i_cos_theta, spos.GetPhi());
        //dV_dphi = CalculateDVDPHI(i_r, i_phi, i_cos_theta);
        dV_dcstheta = CalculateInterpolatedDVDCosTheta(i_r,
            i_phi, i_cos_theta, cos(spos.GetTheta()));
        //dV_dcstheta = CalculateDVDCosTheta(i_r, i_phi, i_cos_theta);

        // sets cartesian basis vectors for spherical coordinates
        e_r = tp[i].GetPos() * (1/tp[i].GetPos().GetNorm());

        e_phi.SetCoords(-sin(spos.GetTheta())*sin(spos.GetPhi()),
            sin(spos.GetTheta())*cos(spos.GetPhi()), 0);

        e_theta.SetCoords(cos(spos.GetTheta())*cos(spos.GetPhi()),
            cos(spos.GetTheta())*sin(spos.GetPhi()),
            -sin(spos.GetTheta()));

        // calculates gradient in cartesian coordinates

```

```

        grad = dV_dr * e_r + (1/sin(spos.GetTheta())/spos.GetR())
            * dV_dphi * e_phi + (-sin(spos.GetTheta()))*
            (dV_dcstheta/spos.GetR()) * e_theta;

        return grad;
    }
    else
        return grad; // grad is null vector in this control path
}

/*****\
* Calculates the derivative of the potential
* due to the eqn. of state in the r-direction at
* a radial position r located in the box
* i_r, i_phi, i_cos_theta. The derivative is
* linearly interpolated depending on r.
* Gets: - ints i_r, i_phi, i_cos_theta, double r
* Returns: - derivative
\*****/
double Star::CalculateInterpolatedDVDR(int i_r, int i_phi,
    int i_cos_theta, double r)
{
    // derivative
    double dV_dr;
    // potential in the neighboring box with greater r
    double v_r_b;
    // potential in current box
    double v_r_c;
    // potential in neighboring box with smaller r
    double v_r_s;
    // radial distance between center of current box
    // and that of box with greater r
    double dr_b;
    // radial distance between center of current box
    // and that of box with smaller r
    double dr_s;
    // r value of smaller boundary of current box
    double r_s;
    // r value of greater boundary of current box
    double r_b;

    // calculates/sets values as described above
    v_r_c = potmap.GetValueInBox(i_r, i_phi, i_cos_theta);
    r_b = potmap.GetRMax() * potmap.F_R_Inv((double)(i_r+1)/
        (potmap.GetIRMax()));
    dr_b = potmap.GetRMax() * (potmap.F_R_Inv((double)(i_r+1.5)/
        (potmap.GetIRMax())) - potmap.F_R_Inv((double)(i_r+0.5)/
        (potmap.GetIRMax())));
    dr_s = potmap.GetRMax() * (potmap.F_R_Inv((double)(i_r+0.5)/
        (potmap.GetIRMax()))
        - potmap.F_R_Inv((double)(i_r-0.5>0?i_r-0.5:0)/
        (potmap.GetIRMax())));
    r_s = potmap.GetRMax() * potmap.F_R_Inv((double)i_r/
        (potmap.GetIRMax()));

    // sets v_r_b equal to DENS_MIN if current box is outermost one
    // (assuming mass density is low there)
    v_r_b = i_r<potmap.GetIRMax()-1?
        potmap.GetValueInBox(i_r+1, i_phi, i_cos_theta):DENS_MIN;

    // sets v_r_s equal to v_r_c if current box is innermost one
    // this sets derivative at center to zero
    v_r_s = i_r>0? potmap.GetValueInBox(i_r-1, i_phi, i_cos_theta):v_r_c;

    // linear interpolation between the two derivatives
    dV_dr = (r-r_s)/(r_b-r_s)*(v_r_b-v_r_c)/dr_b + (r_b-r)/(r_b-r_s)*
        (v_r_c-v_r_s)/dr_s;

```



```

        return dV_dr;
    }

/*****\
* Calculates the derivative of the potential
* due to the eqn. of state in the phi-direction at
* an angular position phi located in the box
* i_r, i_phi, i_cos_theta. The derivative is
* linearly interpolated depending on phi.
* Gets: - ints i_r, i_phi, i_cos_theta, double phi
* Returns: - derivative
\*****/
double Star::CalculateInterpolatedDVDPHi(int i_r, int i_phi,
    int i_cos_theta, double phi)
{
    double dV_dphi;// derivative
    double v_phi_b;// potential in the neighboring box with greater phi
    double v_phi_c;// potential in current box
    double v_phi_s;// potential in neighboring box with smaller phi
    double dphi;// angular distance between neighboring boxes
    double phi_s;// phi value of smaller boundary of current box
    double phi_b;// phi value if greater boundary of current box

    // calculates values as described above
    v_phi_c = potmap.GetValueInBox(i_r, i_phi, i_cos_theta);
    phi_s = (double)i_phi/potmap.GetIPhiMax()*2*PI;
    phi_b = (double)(i_phi+1)/potmap.GetIPhiMax()*2*PI;
    dphi = 2.0*PI/(double)potmap.GetIPhiMax();

    // makes sure i_phi is in correct boundaries
    v_phi_b = i_phi<potmap.GetIPhiMax()-1?
        potmap.GetValueInBox(i_r, i_phi+1, i_cos_theta):
        potmap.GetValueInBox(i_r, 0, i_cos_theta);
    v_phi_s = i_phi>0?potmap.GetValueInBox(i_r, i_phi-1, i_cos_theta):
        potmap.GetValueInBox(i_r, potmap.GetIPhiMax()-1, i_cos_theta);

    // linearly interpolates between the two derivatives at the boundaries
    dV_dphi = (phi-phi_s)/(phi_b-phi_s)*(v_phi_b-v_phi_c)/dphi +
        (phi_b-phi)/(phi_b-phi_s)*(v_phi_c-v_phi_s)/dphi;

    return dV_dphi;
}

/*****\
* Calculates the derivative of the potential
* due to the eqn. of state in the "cos(theta)-direction" at
* a position costheta located in the box
* i_r, i_phi, i_cos_theta. The derivative is
* linearly interpolated depending on cos(theta).
* Gets: - ints i_r, i_phi, i_cos_theta, double costheta
* Returns: - derivative
\*****/
double Star::CalculateInterpolatedDVDCosTheta(int i_r, int i_phi,
    int i_cos_theta, double costheta)
{
    double dV_dct;// derivative
    double v_ct_b;// potential in the neighboring box with box with greater r
    double v_ct_c;// potential in current box
    double v_ct_s;// potential in neighboring box with smaller r
    double dct;// difference of cos(theta) for two neighbouring boxes
    double ct_s;// cos(theta) value of smaller boundary of current box
    double ct_b;// cos(theta) value of greater boundary of current box

    // calculates derivative only if there are separations at all
    if(potmap.GetICosThetaMax(>1)
    {
        // sets values as described above
        v_ct_c = potmap.GetValueInBox(i_r, i_phi, i_cos_theta);

```

```

        dct = 2.0/(double)potmap.GetICosThetaMax();
        ct_s = 2.0 * (double)i_cos_theta/potmap.GetICosThetaMax()-1.0;
        ct_b = 2.0 * (double)(i_cos_theta+1)/potmap.GetICosThetaMax()-1.0;

        // makes sure i_cos_theta is in correct boundaries.
        // if current box is at a boundary the potential
        // for the "missing" next box is set to the value
        // of the potential in the current box
        // HIGHLY ARBITRARY!!
        v_ct_b = i_cos_theta<potmap.GetICosThetaMax()-1?
            potmap.GetValueInBox(i_r, i_phi, i_cos_theta+1):
            v_ct_c;
        v_ct_s = i_cos_theta>0?
            potmap.GetValueInBox(i_r, i_phi, i_cos_theta-1):
            v_ct_c;

        // linearly interpolates between the two derivatives
        // at the boundaries
        dV_dct = (costheta-ct_s)/(ct_b-ct_s)*(v_ct_b-v_ct_c)/dct +
            (ct_b-costheta)/(ct_b-ct_s)*(v_ct_c-v_ct_s)/dct;
    }
    else
        dV_dct=0;

    return dV_dct;
}

/*****\
* Refreshs the information stored in tp_id[] and
* id_tp[].
* Gets: nothing
* Returns: nothing
\*****/
void Star::RefreshTP()
{
    int i=0;

    for(i=0; i<number_tp; i++)
    {
        tp_id[i] = tp[i].GetID();
        id_tp[tp[i].GetID()] = i;
    }
    return;
}

/*****\
* saves the coordinates of every
* (number_tp/points_per_step)th test particle in a file
* ("suno_all.dat") readable by output.exe
* Needs: - function Star::RefreshTP()
* Gets: - int points_per_step, i.e. no. of points to be
*         saved per time step
* Returns: nothing
\*****/
void Star::SaveCoordinatesVB(int points_per_step)
{
    int j,k; // loop counters
    int skip; // number of points to be skipped

    // Update arrays tp_id and id_tp
    RefreshTP();

    // open output file
    ofstream sunoall ("suno_all.dat", ios::app);

    // calculates skip

```

```

        skip = number_tp/points_per_step ? number_tp/points_per_step : 1;

// save coordinates in output file
if (sunoall.is_open())
    for(j=0; j<number_tp; j+=skip)
    {
        for(k=0; k<3; k++)
        {
            sunoall << tp[id_tp[j]].GetPos().GetCoord(k) << endl;
        }
    }

// set end-of-timestep mark
sunoall << "end of timestep\n";

// close output file
sunoall.close();

// the following is just for error diagnosis
// begins here////////////////////////////////////
ofstream denstp ("dens_tp.dat", ios::app);
if (denstp.is_open())
    for(j=0; j<3*skip; j+=skip)
    {
        Spherical spos;
        spos=tp[id_tp[j]].GetPos().GetSpherical();
        int i_r=dmap.GetIR(spos);
        if(i_r<dmap.GetIRMax())
        {
            denstp << "Particle #" << j << ": Density: " <<
                dmap.GetValueInBox(i_r,dmap.GetIPhi(spos),
                dmap.GetICosTheta(spos)) <<
                ", Force due to EOS:" <<
                GetGradEOSPotential(id_tp[j]).GetNorm() <<
                ", Force due to gravity:" <<
                G*m_tp*m_tp*id_tp[j]/
                tp[id_tp[j]].GetPos().GetNormSquared()
                << endl;
        }
    }
    denstp.close();
// ends here////////////////////////////////////

return;
}

```

```

/*****\
* Saves all the positions and momenta of all test
* particles inside a sphere of radius max_radius to
* a binary data file. Other parameters such as the
* number of test particles saved, the total mass of the
* saved star etc. are also saved.
* Gets: - max_radius as double
*        - timestep as int (used for filename)
* Returns: nothing
/*****/
void Star::SaveAllData(double max_radius, int timestep)
{
    double temp=0; // for temporary storage
    int max_i=0; // number of last particle saved
    ostream s;
    s << timestep << ends;
    string timestepstring = s.str(); // convert timestep to string
    string filenamestr = "suno_full_data";
    string filenameext = ".dat";
    filenamestr = filenamestr + timestepstring + filenameext;
    const char* filename = filenamestr.c_str();

```

```

    ofstream out_bin (filename, ios::binary);

    // I'm not sure if this is necessary but t/ps are sorted
    SortTP();

    // finds last test particle within sphere of radius max_radius
    for(max_i=0; tp[max_i].GetPos().GetNorm() < max_radius &&
        max_i<number_tp; max_i++);
    cout << "Number of t/ps saved: " << max_i << endl;

    // calculates the total mass of the saved particles
    double new_m_star = m_tp * max_i;

    // save all the (more or less...) relevant parameters
    out_bin.write(&max_i, sizeof(int));
    out_bin.write(&max_radius, sizeof(double));
    out_bin.write(&new_m_star, sizeof(double));
    out_bin.write(&m_tp, sizeof(double));
    /*out_bin.write(&GAMMA, sizeof(double));
    out_bin.write(&Y_E, sizeof(double));
    out_bin.write(&flag_thermalize, sizeof(bool));
    out_bin.write(&flag_average_theta, sizeof(bool));*/

    // save all the positions and momenta
    for(int i=0; i<number_tp; i++)
    {
        for(int j=0; j<3; j++){
            temp = tp[i].GetPos().GetCoord(j);
            out_bin.write(&temp, sizeof(double));
        }
        for(int j=0; j<3; j++){
            temp = tp[i].GetMom().GetCoord(j);
            out_bin.write(&temp, sizeof(double));
        }
    }

    // close filestream
    out_bin.close();

    return;
}

/*****\
* Reads the whole data (positions, momenta and parameters)
* saved with Star::SaveAllData() for use as initial
* conditions of a new simulation.
* Gets: nothing
* Returns: nothing
\*****/
void Star::ReadAllData()
{
    double temp=0; // temporary
    Vector vec; // also just temporary
    ifstream in_bin ("suno_full_data.dat", ios::binary);

    // read all the (more or less...) relevant parameters
    in_bin.read(&number_tp, sizeof(int));
    in_bin.read(&radius, sizeof(double));
    in_bin.read(&m_star, sizeof(double));
    in_bin.read(&m_tp, sizeof(double));
    /*    in_bin.read(&GAMMA, sizeof(double));
        in_bin.read(&Y_E, sizeof(double));
        in_bin.read(&flag_thermalize, sizeof(bool));
        in_bin.read(&flag_average_theta, sizeof(bool));*/

    // read all the positions and momenta
    for(int i=0; i<number_tp; i++)
    {

```

```

        for(int j=0; j<3; j++)
        {
            in_bin.read(&temp, sizeof(double));
            vec.SetCoord(j, temp);
        }
        tp[i].SetPos(vec);
        for(int j=0; j<3; j++)
        {
            in_bin.read(&temp, sizeof(double));
            vec.SetCoord(j, temp);
        }
        tp[i].SetMom(vec);
    }

    // close file stream
    in_bin.close();

    return;
}

/*****\
 * Saves a slice through the density map in the x-z-plane
 * to "suno_dens.dat". This file is readable by "SuNo
 * Density Output.exe".
 * Gets: nothing
 * Returns: nothing
 \*****/
void Star::SaveTPMDensityVB()
{
    int i_r=0, i_cos_theta=0, i=0; // loop counters

    ofstream tpmdensity("suno_dens.dat", ios::app);

    tpmdensity << dmap.GetRMax() << endl;
    tpmdensity << time_passed << endl;
    for(i_r=0; i_r<dmap.GetIRMax(); i_r++)
        for(i=0; i<2; i++)
            for(i_cos_theta=0; i_cos_theta<dmap.GetICosThetaMax();
                i_cos_theta++)
                tpmdensity << dmap.GetValueInBox(i_r,
                    i*dmap.GetIPhiMax()/2,i_cos_theta)
                    << endl;

    tpmdensity.close();

    return;
}

/*****\
 * Compares the distance from the origin of two test particles. This function
 * is needed to use qsort() from stdlib.h.
 * Gets: - pointers to the two test particles to be compared
 * Returns: - 1 if a is farther away from origin than b, -1 otherwise
 \*****/
int compare_tp_dist(const void* a, const void* b)
{
    return ((TestParticle*)a->GetPos().GetNormSquared() >
        ((TestParticle*)b->GetPos().GetNormSquared())? 1 : -1;
}

/*****\
 * Sorts the test particles by their distance from the origin.
 * Needs: stdlib.h, global function compare_tp_dist
 * Gets: nothing
 * Returns: nothing
 \*****/
void Star::SortTP()

```

```

{
    qsort(tp, number_tp, sizeof(TestParticle),compare_tp_dist);
}

/*****\
* Calculates & returns the time derivatives of test particle
* i's position and momentum, those are its velocity
* and the force on it.
* TEST PARTICLES MUST BE SORTED!
* Gets: - number of test particle i
* Returns: - derivatives as a TestParticle object
*****/
TPChange Star::Deriv(int i)
{
    TPChange derivs;

    // stores derivatives of position and momentum of particle i in derivs
    derivs.SetPosChg(tp[i].GetMom()*(1/
        sqrt(m_tp*m_tp+tp[i].GetMom().GetNormSquared()/(c*c))));
    // the long expression above is the relativistic velocity
    // of test particle i
    derivs.SetMomChg(GetForceOnTP(i));

    return derivs;
}

/*****\
* Calculates the next step in the star's time development using the Euler
* method.
* Uses function Star::Deriv().
* Gets: - time stepsize dt for the step
*****/
void Star::NextStepEuler(double dt)
{
    int i=0;
    TestParticle* initial;

    initial = new TestParticle [number_tp];

    // sorts the test particles
    SortTP();
    RefreshTP();

    for(i=0; i<number_tp; i++)
        initial[i] = tp[i];

    for(i=0; i<number_tp; i++)
    {
        tp[i] = initial[i] + dt*Deriv(i);
        /*      tp[i].SetMom(initial[i].GetMom() + Deriv(i).GetMom() * dt);
           tp[i].SetPos(initial[i].GetPos() + Deriv(i).GetPos() * dt);*/
    }

    delete[] initial;

    return;
}

/*****\
* Calculates number_of_steps steps with stepsize dt. Density of test
* particles is written into a file ("suno_out.txt").
* Coordinates of (some) test particles are saved using SaveCoordinates().
*
* Gets: - time stepsize dt for the step
*        - number of steps as int
*        - number of points to be saved to the output file by function
*        Star::SaveCoordinatesVB() per time step as int

```

```

*      points_per_step_save
*      - bool modify_stepsize which indicates if the stepsize may be
*        modified during the calculation
* Returns: nothing
*
*****/
void Star::Stepper(double dt, int number_of_steps, int points_per_step_save,
    bool modify_stepsize, bool modify_stepsize_avvel)
{
    int i=0;
    bool stepsize_modified=false; // indicates if stepsize already modified
    bool stepsize_modified_twice=false;
    double av_disp=0; // average position change of t/ps in last step

    // deletes old file created with Star::SaveCoordinatesVB() and
    // writes the number of time steps to the 1st line of the output file
    // open output file
    ofstream sunoall ("suno_all.dat");

    // write number of stime steps to file
    //( +1 because the initial configuration is also saved)
    sunoall << number_of_steps + 1 << endl;

    // close output stream
    sunoall.close();

    // deletes old file created with Star::SaveTPMDensityVB() and
    // writes the no. of time steps, i_r_max, i_cos_theta_max and
    // F_R_Inv(i) to the new output file
    ofstream tpmdensity ("suno_dens.dat");

    tpmdensity << number_of_steps + 1 << endl;
    tpmdensity << dmap.GetIRMax() << endl;
    tpmdensity << dmap.GetICosThetaMax() << endl;
    for(i=0; i<dmap.GetIRMax(); i++)
        tpmdensity << dmap.F_R_Inv((double)i/dmap.GetIRMax()) << endl;

    tpmdensity.close();

    // files to save energies and angular momentum
    ofstream sunoenergy ("suno_energy.dat");
    ofstream sunoangmom ("suno_angmom.dat");

    // the following is just for diagnosis
    // begins here////////////////////////////////////
    // deletes old dens_tp.dat - file
    ofstream denstp ("dens_tp.dat");
    denstp.close();
    // ends here////////////////////////////////////

    SortTP();
    CalculateTPMDensityMap();
    SaveTPMDensityVB();

    // saves initial configuration
    //      SaveCoordinates(i);
    SaveCoordinatesVB(points_per_step_save);

    // set internal energies for test particles if "energy method" is used
    if(flag_energy_method)
        InitializeInternalEnergies();

    for(i=0; i<number_of_steps; i++)
    {
        // sets new time
        time_passed += dt;

        // unnecessary output

```

```

        cout << "Calculating time step " << i+1 << ".\n";
        if(modify_stepsize && !stepsize_modified_twice)
            dt = ModifyStepsize(dt, stepsize_modified,
                                stepsize_modified_twice);
        av_disp = NextStepRK4(dt, modify_stepsize_avvel);
        // enforce energy conservation by modifying momentum
        // so that the change in kinetic energy compensates
        // the change in internal energy
        if(flag_energy_method)
            ModifyMomentaConservingEnergy();

        if(i>5 && modify_stepsize_avvel)
            dt = ModifyStepsizeAvVel(dt, av_disp);

        SaveCoordinatesVB(points_per_step_save);
        SaveTPMDensityVB();

        // save energies
        sunoenergy << time_passed << "\t" << MeasureKineticEnergy(true)
            + MeasureGravitationalEnergy() + MeasureInternalEnergy() <<
            "\t " << MeasureKineticEnergy(true) << "\t" <<
            MeasureGravitationalEnergy() <<
            "\t" << MeasureInternalEnergy() << endl;

        // save angular momentum
        sunoangmom << time_passed << "\t" <<
            MeasureTotalAngularMomentum().GetNorm() << endl;
    }

    // close file streams
    sunoenergy.close();
    sunoangmom.close();

    return;
}

/*****\
* Modifies the stepsize used by Star::Stepper(). Checks the
* average displacement of all t/ps in last time step. If too
* big, stepsize is divided by 2.
* Gets: - current stepsize dt as double
*        - average displacement av_disp as double
* Returns: - new stepsize as double
*****/
double Star::ModifyStepsizeAvVel(double dt, double av_disp)
{
    if(av_disp>radius/6e2)
    {
        dt /= 2.0;
        cout << "Step size divided by 2.\n";
        cout << "Average displacement of t/ps was: " << av_disp << "m.\n";
    }
    else
        if(av_disp<radius/6e3)
        {
            dt *= 2.0;
            cout << "Step size multiplied by 2.\n";
            cout << "Average displacement of t/ps was: " <<
                av_disp << "m.\n";
        }

    return dt;
}

/*****\
* Modifies the stepsize used by Star::Stepper(). Stepsize is
* divided by 10 if 70% of the test particles are in a shell
* with 1/4 the radius of the initial star.

```



```

* Gets: - currently used stepsize dt as double
* Returns: - new stepsize as double
\*****/
double Star::ModifyStepsize(double dt, bool& stepsize_modified,
    bool& stepsize_modified_twice)
{
    double new_dt=0;

    // check if 70% of t/ps are in the 1/4 - radius - shell
    if(!stepsize_modified)
        new_dt= (stepsize_modified =
            (tp[(int)(0.7 * number_tp)].GetPos().GetNormSquared()
            < radius*radius/16)) ? dt/10 : dt;

    /*
    // check if 70% of t/ps are in the 1/5 - radius - shell
    if(!stepsize_modified)
        new_dt= (stepsize_modified =
            (tp[(int)(0.7 * number_tp)].GetPos().GetNormSquared()
            < radius*radius/25)) ? dt/10 : dt;

    */
    else
        // check if 70% of t/ps are in a 1/25 - radius - shell
        if(!stepsize_modified_twice)
            new_dt= (stepsize_modified_twice =
                (tp[(int)(0.7 * number_tp)].GetPos().GetNormSquared()
                < radius*radius/625)) ? dt/20 : dt;

    return new_dt;
}

\*****/
* mass density as a function of distance from the center
* (initial condition)
* Gets: - distance r from center as double
* - parameter_d as double
* Returns: - density at that distance
\*****/
double Star::initial_mass_density(double r, double parameter_d)
{
    double rho; // mass density

    // realistic
    //rho = 1.0/(4.0*PI/3.0*parameter_d*pow(r*10,3)+1.005e-13);

    // realistic inner core 0.7 M_\odot contracted by 5
    rho = pow(5.0,3)/(8.2436e-30*pow(r*5,3)+1.005e-13);

    // homogeneous
    //rho = m_star/4.0/pow(radius,3);

    // ^bonazzola fig. 1 c
    //rho = 2.4e17 * 0.08 * exp(-8e-10*pow(r,2));

    return rho;
}

\*****/
* Sets test particle tp[i] to a random position in a sphere
* of radius r around the star's center.
* Random number generator should be initialized before
* calling this function.
* Gets: - int i identifying the t/p
* - radius r of sphere as double
* Returns: - true if an error occurs, false otherwise
\*****/
bool Star::PutTPInSphere(int i, double r)

```

```

{
    Vector vec; // for temporary use

    // check if argument values are ok
    if(i<0)
    {
        cout << "Error: argument i<0 in function PutTPInSphere().\n";
        return true;
    }
    else
        if(i>=number_tp)
        {
            cout << "Error: argument i>=number_tp in function
PutTPInSphere().\n";
            return true;
        }
    else
        if(r <= 0)
        {
            cout << "Error: argument r<=0 in function PutTPInSphere().\n";
            return true;
        }

    // finds random vector
    do
        for(int j=0; j<3; j++)
            vec.SetCoord(j, 2.0*rand()/RAND_MAX-1.0);
    while(vec.GetNormSquared()>1); //rejection method(see Numerical Recipies)

    // sets test particle i to the random position
    tp[i].SetPos(vec*r);

    return false;
}

/*****\
* Distributes test particles in position space. Uses function
* Star::initial_mass_density(), normalizes it (so that it
* gives the correct total mass of the star) and distributes
* test particles accordingly.
* Needs: - number of shells with different density as int
* Returns: nothing
\*****/
void Star::DistributeTPPos(int no_shells)
{
    // shell thickness
    double delta_r=0;
    // array containing radii of spheres to be filled
    double r[no_shells];
    // mass density for current shell
    double rho=0;
    // array containing the numbers of test particles distributed
    int no_tps_sphere[no_shells];
    int no_tps_current_sphere=0;
    // total number of t/ps already distributed
    int tps_dist=0;
    // bulk mass of core resulting from density distribution
    double bulk_mass=0;
    // parameter for initial density distribution
    double par_d=1.5e-30;

    // unnecessary output
    cout << "Position:";

    // initializes random generator
    srand(time(NULL));

    // calculates shell thickness

```

```

delta_r = radius / no_shells;

// calculate the bulk mass resulting from initial_mass_density()
// in order to normalize density distribution so that m_star comes
// out correctly. also vary parameter par_d to get correct mass
//do{
    bulk_mass = 0;
    for(int k=0; k<no_shells; k++)
    {
        r[k] = radius-delta_r*k;
        rho = initial_mass_density(r[k] - delta_r/2, par_d);
        bulk_mass += rho*4.0*PI/3.0*(pow(r[k],3)-pow(r[k]-
            delta_r,3));
    }
    //par_d +=0.0001e-30;
    //}while(bulk_mass>m_star);

    cout << "bulk mass: " << bulk_mass << endl;

    // assuming that density increases when going inward, distribute
    // test particles in shells (starting outside) in a way that
    // creates the correct densities
    for(int k=0; k<no_shells; k++)
    {
        // get desired mass density for current shell
        rho = m_star/bulk_mass * initial_mass_density(r[k] -
            delta_r/2, par_d);
        // calculate how many test particles are needed in
        // current sphere (not shell!!)
        // always too small due to round-off
        no_tps_current_sphere = (int)(rho*4.0/3.0*PI*pow(r[k],3)/m_tp);
        no_tps_current_sphere = no_tps_current_sphere<number_tp?
            no_tps_current_sphere:number_tp;
        // subtract the (approximate) number of particles that
        // is already in the sphere
        for(int n=0; k-n>0; n++)
        {
            no_tps_current_sphere -= (int)(no_tps_sphere[n]*
                pow(r[k]/r[n],3)+1); //always subtract to much
        }
        no_tps_current_sphere = no_tps_current_sphere>0?
            no_tps_current_sphere:0;
        no_tps_sphere[k] = no_tps_current_sphere;

        // finally put the corrected number of test particles
        // in the sphere
        for(int j=0; j<no_tps_current_sphere; j++)
        {
            if(tps_dist+j>=number_tp)
            {
                cout << "Ran out of test particles
while creating initial position space configuration.\n";
                break;
            }
            PutTPInSphere(tps_dist+j, r[k]);

            // output of progress (unnecessary)
            if((tps_dist+j)%(number_tp/10)==0)
                cout << ".";
        }
        // count the distributed test particles
        tps_dist+=no_tps_current_sphere;
    }

    cout << "\nResidual test particles: " << number_tp-tps_dist << endl;

    // distribute residual test particles
    // completely arbitrary but shouldn't matter

```

```

        for(int j=tps_dist; j<number_tp; j++)
            PutTPInSphere(j, radius/3);

    return;
}

/*****\
* Distributes test particles in momentum space imposing a "rigid body rotation"
* at angular velocity omega if diff_rot = false.
* Distributes test particles in momentum space using a radial
* dependence of the angular velocity
*      omega_0 * r_0^2
*      omega = -----
*                r^2 + r_0^2
* if diff_rot = true.
*
* Gets: - Vector omega (corresponds to omega_0 in the above formula)
*       - r_0 as double
*       - diff_rot as bool
* Returns: nothing
\*****/
void Star::DistributeTPMom(Vector omega, double r_0, bool diff_rot)
{
    Vector v; // for temporary storage / local velocity
    double max=0; // maximum velocity squared
    double vc=0;

    // unnecessary output of rotation type used
    switch(diff_rot)
    {
        case true:
            cout << "Using differential rotation with r_0 = "
                  << r_0 << "m.\n";
            break;
        default:
            cout << "Using rigid body rotation.\n";
    }

    // unnecessary output
    cout << "Momentum:";

    for(int i=0; i<number_tp; i++)
    {
        switch(diff_rot)
        {
            case true:
                v = r_0*r_0/(tp[i].GetPos().GetNormSquared() +
                           r_0*r_0) * (omega * tp[i].GetPos());
                break;
            default:
                v = omega * tp[i].GetPos();
        }
        //v = v + (-1) * tp[i].GetPos() *5;// infall
        if((vc=v.GetNormSquared())>max)max=vc;
        // set relativistic momentum
        tp[i].SetMom(m_tp * v * (1/sqrt(1-(v^v)/c/c)));
        // output of progress (unnecessary)
        if(i%(number_tp/10)==0)
            cout << ".";
    }

    // Warning if velocities are close to c (or greater than c!)
    if(max>0.81*c*c && max < c*c) cout
        << "Warning: initial velocities greater than 0.9c.\n";
    if(max > c*c) cout
        << "Warning: initial velocity greater than c found!\n";

    cout << endl;
}

```

```

        return;
    }

    /*****\
    * Gives the t/ps a momentum towards the center of the core. This
    * momentum increases linearly with the t/p's distance from the center.
    * Gets: - double max_velocity indicating the maximum velocity (for the
    *         outermost t/p
    * Returns: nothing
    \*****/
    void Star::KickTPInside(double max_velocity)
    {
        Vector newmom;

        for(int i=0; i<number_tp; i++)
        {
            newmom = tp[i].GetMom() + tp[i].GetPos() *
                    (-1/radius) * m_tp * max_velocity;
            tp[i].SetMom(newmom);
        }

        return;
    }

    /*****\
    * Calculates the next step in the star's time development using a
    * fourth order Runge-Kutta-Algorithm
    *
    * Gets: - time stepsize dt for the step
    *        - flag modify_stepsize_avvel as bool indicating if if the
    *          average velocity of all t/ps shall be calculated
    \*****/
    double Star::NextStepRK4(double dt, bool modify_stepsize_avvel)
    {
        int i;
        // for saving the initial configuration
        TestParticle *initial;
        // changes of test particle momentum and positions
        TPChange *k1, *k2, *k3, *k4, total_k;
        // average position change of all test particles in this step
        double av_pos_chg=0;

        // memory reservation
        initial = new TestParticle[number_tp];
        k1 = new TPChange[number_tp];
        k2 = new TPChange[number_tp];
        k3 = new TPChange[number_tp];
        k4 = new TPChange[number_tp];

        // 1st step
        CalculateK(k1,dt);
        // copies initial configuration
        // (after 1st step because now test particles are sorted)
        for (i=0 ;i<number_tp ;i++)
            initial[tp_id[i]] = tp[i];
        // moves system to 1st intermediate position
        MoveTPs(initial, k1, 0.5);

        // 2nd step
        CalculateK(k2,dt);
        // moves system to new intermediate position
        MoveTPs(initial, k2, 0.5);

        // 3rd step
        CalculateK(k3, dt);
        // moves system to final position
        MoveTPs(initial, k3, 1);
    }

```

```

// 4th step and performing overall step
CalculateK(k4,dt);
for (i=0; i<number_tp ;i++)
{
    total_k = k1[i]*(1.0/6.0) + k2[i]*(1.0/3.0)
              + k3[i]*(1.0/3.0) + k4[i]*(1.0/6.0);
    tp[id_tp[i]] = initial[i] + total_k;
    // calculate average displacement only if needed
    if(modify_stepsize_avvel)
        av_pos_chg += total_k.GetPosChg().GetNorm();
}

// free memory
delete[] initial;
delete[] k1;
delete[] k2;
delete[] k3;
delete[] k4;

// normalize av_pos_chg
av_pos_chg /= number_tp;

return av_pos_chg;
}

/*****\
* Calculates the current TPChange for all test particles and
* saves it in the TPChange array k. dt is the step size.
* Gets: - a pointer to an array of number_tp TPChange objects
*        - desired step size
* Returns: nothing, but changes *k
\*****/
void Star::CalculateK(TPChange* k, double dt)
{
    int i=0;

    SortTP();
    RefreshTP();
    CalculateEOSPotentialMap();
    for (i=0 ;i<number_tp ;i++)
        k[tp_id[i]] = Deriv(i) * dt; // calculates k

    return;
}

/*****\
* Moves all of the star's test particles to *initial+*k * kfactor
* Gets: - a pointer to an array of number_tp TestParticles initial
*        - a pointer to an array of number_tp TPChanges k
*        - a double kfactor
* Returns: nothing, but changes *tp
\*****/
void Star::MoveTPs(TestParticle* initial, TPChange* k, double kfactor)
{
    int i;

    for (i=0 ;i<number_tp ;i++)
        tp[id_tp[i]] = initial[i] + k[i]*kfactor;

    return;
}

/*****\
* Modifies the momenta of all test particles to enforce
* energy conservation. Should be called after every time
* step. Previous internal energy of the test particle is
* compared to its current internal energy. The difference

```

```

* is subtracted from its kinetic energy so that also
* the total angular momentum is conserved.
* Gets: nothing
* Returns: nothing
\*****/
void Star::ModifyMomenaConservingEnergy()
{
    Spherical spos;
    int i_r=0, i_phi=0, i_cos_theta=0;
    double e_kin_change=0, momentum_change=0,
           p=0, p_r=0, newenergy=0, temp=0, real_e_kin_change=0;
    Vector newmom, e_r;

    CalculateEOSPotentialMap(); // might be obsolete
    for (int i=0 ;i<number_tp ;i++)
    {
        // converts position to spherical coords
        spos = tp[i].GetPos().GetSpherical();
        // r-direction unit vector
        e_r = tp[i].GetPos() * (-1/spos.GetR());
        // indices of box in which t/p is located
        i_r = potmap.GetIR(spos);
        i_phi = potmap.GetIPhi(spos);
        i_cos_theta = potmap.GetICosTheta(spos);
        if(i_r<potmap.GetIRMax()) // t/p inside map?
            newenergy = potmap.GetValueInBox(i_r,i_phi,i_cos_theta);
        else
            newenergy = tp[i].GetInternalEnergy();//new energy==old energy
        e_kin_change = newenergy-tp[i].GetInternalEnergy();
        p=tp[i].GetMom().GetNorm(); // momentum of t/p

        p_r=tp[i].GetMom()^e_r; // radial proj. of momentum

        temp=p_r*p_r-p*p+pow(e_kin_change/c+
            sqrt(pow(m_tp*c,2)+p*p),2)-pow(m_tp*c,2);
        if(temp>=0)
        {
            momentum_change = p_r - sqrt(temp);
            if(momentum_change < -p_r)
                momentum_change = -p_r; // stop test particle
        }
        else
        {
            // arbitrary, doesn't matter very much if p is very small
            momentum_change = 0;
        }

        // calculates new momentum
        newmom = tp[i].GetMom() + momentum_change * e_r;
        // sets new momentum
        tp[i].SetMom(newmom);

        real_e_kin_change = sqrt(pow(m_tp*c,2)+
            newmom.GetNormSquared()*c*c)-
            sqrt(pow(m_tp*c,2)+pow(p,2)*c*c);
        // sets new internal energy for t/p, allowing "energy debt"
        tp[i].SetInternalEnergy(newenergy-e_kin_change-real_e_kin_change);

        //tp[i].SetInternalEnergy(newenergy); // no "energy debt"
    }

    return;
}

\*****/
* Sets the initial values for the internal energies of
* all test particles. Should be called before the beginning
* of the simulation.

```

```

* Gets: nothing
* Returns: nothing
\*****/
void Star::InitializeInternalEnergies()
{
    Spherical spos;
    int i_r=0, i_phi=0, i_cos_theta=0;

    CalculateEOSPotentialMap();
    for (int i=0 ;i<number_tp ;i++)
    {
        // get t/p's position in spherical coordinates
        spos = tp[i].GetPos().GetSpherical();
        // indices of box in which t/p is located
        i_r = potmap.GetIR(spos);
        // make sure t/p is in map
        i_r = i_r < potmap.GetIRMax() ? i_r : potmap.GetIRMax()-1;
        i_phi = potmap.GetIPhi(spos);
        i_cos_theta = potmap.GetICosTheta(spos);
        // set internal energy value
        tp[i].SetInternalEnergy(potmap.GetValueInBox(i_r,
            i_phi,i_cos_theta));
    }
    return;
}

\*****/
* Calculates the TPSmearedNumberMap for the current configuration
* of the star.
* Gets: - a reference to a TPDoubleMap object
* Returns: nothing
\*****/
void Star::CalculateTPSmearedNumberMap(TPDoubleMap& nmap)
{
    int i=0;
    Spherical spos;
    double new_r_max; // new value for r_max, calculated below

    //SortTP(); // might become obsolete

    // sets r_max to 10.1/9.0 times the distance from the
    // origin of the (number_tp/100)th most distant test
    // particle (in order to include all "important" particles
    // that do not escape due to fluctuations)
    // r_max is conjectured to be smaller than 1.1*radius
    new_r_max = tp[(int)(0.99*(number_tp-1))].GetPos().GetNorm()*10.1/9.0;
    nmap.SetRMax(new_r_max < 1.1*radius ? new_r_max : 1.1*radius);

    // deletes entries in box_tp array if thermalization
    // is activated
    if(GetFlagThermalize())
        ClearBoxTP();

    // adds all test particles to the number map
    for(i=0; i<number_tp; i++)
    {
        // convert tp's position vector to spherical coordinates
        spos = tp[i].GetPos().GetSpherical();
        nmap.AddTPSmeared(spos); // nmap.AddTP() for unsmeared
        // add test particle's index to box_tp if
        // thermalization is activated
        if(GetFlagThermalize())
        {
            // add index to box_tp if test particle in map
            int i_r=nmap.GetIR(spos);
            if(i_r<nmap.GetIRMax())
                box_tp[i_r][nmap.GetIPhi(spos)]
                    [nmap.GetICosTheta(spos)].push_back(i);
        }
    }
}

```



```

    }

    return;
}

/*****\
 * Calculates the mass density map.
 * Gets: - nothing
 * returns: nothing
 \*****/
void Star::CalculateTPMDensityMap()
{
    TPDoubleMap nmap(dmap.GetIRMax(), dmap.GetIPhiMax(),
                     dmap.GetICosThetaMax());
    double vol, dens;
    int i_r=0, i_phi=0, i_cos_theta=0;
    double tpincsph=0;
    static const double RHO_1=0.0; //2.4e17 , density for thermalization

    // calculates number map first
    CalculateTPSmearedNumberMap(nmap);

    // sets r_max to the value calculated by CalculateTPNumberMap()
    dmap.SetRMax(nmap.GetRMax());

    // set the density in the central sphere to an averaged value
    // (independent of i_phi and i_theta)
    vol = dmap.VolOfBox(0) * dmap.GetIPhiMax() *
          dmap.GetICosThetaMax(); // volume of central sphere
    // counts all particles in central sphere
    for(i_phi=0; i_phi<dmap.GetIPhiMax(); i_phi++)
        for(i_cos_theta=0; i_cos_theta<dmap.GetICosThetaMax(); i_cos_theta++)
            tpincsph += nmap.GetValueInBox(0,i_phi,i_cos_theta);
    // sets the averaged value
    dens = m_tp * tpincsph/vol;
    dens = dens > DENS_MIN ? dens : DENS_MIN;
    for(i_phi=0; i_phi<dmap.GetIPhiMax(); i_phi++)
        for(i_cos_theta=0; i_cos_theta<dmap.GetICosThetaMax(); i_cos_theta++)
            dmap.SetValueInBox(0,i_phi,i_cos_theta, dens);

    // calculates mass density map by dividing number map by volume of
    // corresponding boxes and multiplying with mass of a test particle
    // (i_r=0 if separations shall be made for the central sphere)
    for(i_r=1; i_r<dmap.GetIRMax(); i_r++)
    {
        vol = dmap.VolOfBox(i_r);
        for(i_phi=0; i_phi<dmap.GetIPhiMax(); i_phi++)
            for(i_cos_theta=0; i_cos_theta<dmap.GetICosThetaMax();
                i_cos_theta++)
            {
                dens = m_tp * (double)nmap.GetValueInBox(i_r,
                    i_phi,i_cos_theta)/vol;
                dens = dens > DENS_MIN ? dens : DENS_MIN;
                dmap.SetValueInBox(i_r,i_phi,i_cos_theta, dens);
                // thermalize test particles in box
                // if density is greater than RHO_1
                if(dmap.GetValueInBox(i_r, i_phi, i_cos_theta)
                    >RHO_1 && GetFlagThermalize())
                    ThermalizeBox(i_r,i_phi,i_cos_theta);
            }
    }

    // averages the density values of the cos_theta-boxes (if flag is true)
    // which are mirror-symmetric about the equatorial plane
    if(flag_average_theta)
        dmap.AverageCosThetaBoxes();
}

```

```

        return;
    }

    /*****\
    * Calculates the EOS potential (=specific internal energy)
    * map.
    * Gets: - nothing
    * returns: nothing
    \*****/
    void Star::CalculateEOSPotentialMap()
    {
        int i_r=0, i_phi=0, i_cos_theta=0;

        // calculates density map
        CalculateTPMDensityMap();

        // sets r_max to the value calculated by CalculateTPNumberMap()
        potmap.SetRMax(dmap.GetRMax());

        // calculates EOS potential map
        for(i_r=0; i_r<potmap.GetIRMax(); i_r++)
            for(i_phi=0; i_phi<potmap.GetIPhiMax(); i_phi++)
                for(i_cos_theta=0; i_cos_theta<potmap.GetICosThetaMax();
                    i_cos_theta++)
                {
                    potmap.SetValueInBox(i_r, i_phi, i_cos_theta,
                        CalculateEOSPotential(i_r, i_phi, i_cos_theta));
                }

        return;
    }

    /*****\
    * Equation of state for the star. Calculates and
    * returns the internal energy (due to the EOS)
    * of a test particle (not a nucleon!) in a certain
    * box of the map.
    * Gets: - integers i_r, i_phi, i_cos_theta which
    *         specify the box for which the potential
    *         energy is calculated and returned
    * Returns: - potential energy in the box as double
    \*****/
    double Star::CalculateEOSPotential(int i_r, int i_phi, int i_cos_theta)
    {
        // nucleons per test particle, neutron mass is used as nucleon mass
        const static double nucleons_per_tp = m_tp/MASS_NEUTRON;
        // density in kg/m^3
        double density = dmap.GetValueInBox(i_r, i_phi, i_cos_theta);
        double potential=0;
        double y_e=CalculateYE(density);

        // flag indicating if this is the first function call
        static bool firstcall=true;
        // transition density between the two EOS
        static double density_at_gap = 1e14;
        static double potential_gap=0;
        double pressure[1];
        // temperature in K
        double temperature = CalculateTemperature(density);

        // stuff for LS EOS
        double ls_out[4];

        // stuff for Helmholtz EOS
        integer ionmax=2; // fortran int, number of different ions
        double xmass[2], aion[2], zion[2]; // composition of matter
        xmass[0]=2.69e-4; // low density value from cooperstein
        xmass[1]=1-xmass[0]; // should be made a function of density

```

```

aion[0]=1;
aion[1]=26/y_e;
zion[0]=1; // use hydrogen and
zion[1]=26; // "iron" only

// calculate the gap at the transition density between the
// two EOS in first call
if(firstcall)
{
    double temperature_at_gap = CalculateTemperature(density_at_gap);
    lseos_ipvar[i_r][i_phi][i_cos_theta][0] =
        kelvin_mev(temperature_at_gap);

    potential_gap = m_tp * CalculateInternalEnergyHelmholtz(xmass,
        aion, zion, ionmax, temperature_at_gap, density_at_gap, pressure)
        - nucleons_per_tp *
        CalculateLSEOS(lseos_ipvar[i_r][i_phi][i_cos_theta],
            y_e, density_at_gap, &(lseos_pprev[i_r][i_phi][i_cos_theta]),
            ls_out);

    firstcall=false;
}

if(density<density_at_gap)
{
    // use Helmholtz EOS
    // contribution only from electron-positron-gas for
    // densities below 1e14
    potential = m_tp * CalculateInternalEnergyHelmholtz(xmass,
        aion, zion, ionmax, temperature, density, pressure);
}
else
{
    // use Lattimer and Swesty EOS
    // set temperature input variable
    // temperature in MeV
    lseos_ipvar[i_r][i_phi][i_cos_theta][0] = kelvin_mev(temperature);
    // nuclear and electron-positron contribution at densities above 1e14
    potential = nucleons_per_tp *
        CalculateLSEOS(lseos_ipvar[i_r][i_phi][i_cos_theta], y_e,
            density, &(lseos_pprev[i_r][i_phi][i_cos_theta]),ls_out)
        + potential_gap;
}

/*
// nuclear contribution (old way) and polytrope EOS for electron gas
//static const double GAMMA=1.33;
//static double K=0.75 * pow(PI/3.0,2.0/3.0) * HBAR * c
* pow(y_e/MASS_NEUTRON,4.0/3.0);
//static double E=1.0/(GAMMA-1.0)*K;
const static double RHO_0=2.4e17; // nuclear density [kg/m^3] 2.4e17

potential = nucleons_per_tp * (A * density / RHO_0
+ B * pow(density/RHO_0, SIGMA));

//potential += E*pow(dmap.GetValueInBox(i_r,
i_phi, i_cos_theta),GAMMA-1.0) * m_tp;
*/

return potential;
}

/*****\
* Returns the force on test particle i. Test
* particles must be sorted, TPMDensityMap dmap must
* be calculated for proper results!
* Gets: - int i identifying test particle
* Returns: - force Vector
*****/

```

```

Vector Star::GetForceOnTP(int i)
{
    Vector f(0,0,0); // null vector
    double r; // distance of test particle i from origin

    // force due to gravity
    // if test particles are unsorted, i must be replaced by CountTPInside(i)
    r = tp[i].GetPos().GetNorm();
    if(i>number_tp/10000) // singularity treatment (very, very simple...)
        f = tp[i].GetPos()*(-G*m_tp*m_tp*i/pow(r,3));

    // force due to equation of state
    if(!flag_energy_method) // add only if "force method" is used
        f = f - GetGradEOSPotential(i);

    return f;
}

/*****\
* Reads the parameters for the simulation from a
* datafile (filename).
* Gets: - name of the datafile as char[]
*        - a whole bunch of references to the parameters
*        to be read
* Returns: - true, if an error occurs
*           - false, if all goes fine
*****/
bool read_parameters(char* filename, int& number_tp, int& i_r_max,
    int& i_phi_max, int& i_cos_theta_max, double& radius, double& mass,
    double& omega_norm, double& stepsize, int& numb_part_save,
    bool& modify_stepsize, bool& thermalize, bool& diff_rotation,
    double& r_0, bool& average_theta, bool& modify_stepsize_avvel,
    double& dens_min, bool& energy_method, double& kick)
{
    ifstream parameters (filename);

    if(parameters)
    {
        parameters >> number_tp;
        parameters >> i_r_max;
        parameters >> i_phi_max;
        parameters >> i_cos_theta_max;
        parameters >> radius;
        parameters >> mass;
        parameters >> omega_norm;
        parameters >> stepsize;
        parameters >> numb_part_save;
        parameters >> modify_stepsize;
        parameters >> modify_stepsize_avvel;
        parameters >> thermalize;
        parameters >> diff_rotation;
        parameters >> r_0;
        parameters >> average_theta;
        parameters >> dens_min;
        parameters >> energy_method;
        parameters >> kick;
    }
    else
    {
        cout << "Error opening parameter file.\n";
        return true;
    }

    parameters.close();

    return false;
}

```

```

int main()
{
    int number_tp=1000, i_r_max=40, i_phi_max=1, i_cos_theta_max=40;
    double radius=1.256e6, mass=2e30, omega_norm=2.1, stepsize=5e-4,
        r_0=1.256e5;
    double dens_min=1e1, kick=0;
    bool modify_stepsize=true, thermalize=false, diff_rotation=false,
        average_theta=false;
    bool modify_stepsize_avvel, new_simulation=true,
        energy_method=false;
    int steps=0; // number of steps
    int numb_part_save=2000; // number of particles saved in every time step
    char* parameter_file="suno_parameters_new.dat";

    cout << "SuNo Version 0.3\n-----\n";
    cout << "Use saved data (=0) or start new simulation (=1)? ";
    cin >> new_simulation;

    if(!new_simulation)
        parameter_file="suno_parameters_ctd.dat";

    // reads the parameters from a data file
    if(read_parameters(parameter_file, number_tp, i_r_max, i_phi_max,
        i_cos_theta_max, radius, mass, omega_norm, stepsize, numb_part_save,
        modify_stepsize, thermalize, diff_rotation, r_0, average_theta,
        modify_stepsize_avvel, dens_min, energy_method, kick))
        return 1; // exit with error code 1 if error in
    //read_parameters() occurs

    // creates a Star object (number of test particles, radius, mass, i_r_max,
    // i_phi_max, i_cos_theta_max, thermalization flag)
    Star star(number_tp, radius, mass, i_r_max, i_phi_max, i_cos_theta_max,
        thermalize, average_theta, dens_min, energy_method);
    Vector omega(0,0,omega_norm);//init. angular velocity of star

    if(new_simulation)
    {
        // distribute test particles in position space
        // (argument=number of shells)
        star.DistributeTPPos(i_r_max);

        // distribute test particles in momentum space
        star.DistributeTPMom(omega, r_0, diff_rotation);

        cout << "Total angular momentum: " <<
            star.MeasureTotalAngularMomentum().GetNorm()
            << " Js" <<endl;
        cout << "Ratio |E_rot/E_pot|: " <<
            fabs(star.MeasureKineticEnergy(true)/
                star.MeasureGravitationalEnergy()) << endl;

        // give t/ps a kick towards the center of the core
        star.KickTPInside(kick);
    }
    else
        star.ReadAllData(); // changes some parameters read from the file

    cout << "Enter number of time steps: ";
    if(cin >> steps)
    {
        // calculate time development: step size, number of steps
        // , ... , flags indicating if stepsize
        // may be modified during the calculation
        star.Stepper(stepsize, steps, numb_part_save,
            modify_stepsize, modify_stepsize_avvel);
    }

    // saves final configuration if this is a new simulation

```

```

        if(new_simulation)
            star.SaveAllData(radius/1, steps);

        cout << "Bye.\n";

        getchar();

        return 0;
    }

```

A.2 testparticle.cpp and testparticle.h

These files contain the C++ classes `TestParticle` and `TPChange`. All functions and data directly related to individual test particles (like their position and momentum vectors) are included in these.

testparticle.h:

```

////////////////////////////////////
// testparticle.h
// Header file for testparticle.cpp
////////////////////////////////////

#ifndef TESTPARTICLE_H
#define TESTPARTICLE_H

#include "vector_and_spherical.h"

// forward decleration
class TPChange;

class TestParticle
{
private:
    Vector pos;           // position vector
    Vector mom;           // momentum vector
    int id;               // identification no. of test particle
    double internal_energy; // internal energy of this test particle
public:
    TestParticle() // constructor
    {
        Vector nullvec(0,0,0);

        pos = nullvec;
        mom = nullvec;
        id = 0;
        internal_energy=0;
    }
    double GetInternalEnergy()
    {
        return internal_energy;
    }
}

```

```

void SetInternalEnergy(double newinterg)
{
    internal_energy = newinterg;
    return;
}
int GetID() // access function for ID
{
    return id;
}
void SetID(int newid) // set new ID
{
    id = newid;
    return;
}
Vector GetPos() // access function for position
{
    return pos;
}
Vector GetMom() // access function for momentum
{
    return mom;
}
void SetPos(Vector newpos) // set new position
{
    pos = newpos;
    return;
}
void SetMom(Vector newmom) // set new momentum
{
    mom = newmom;
    return;
}
// binary operator for addition of a TPChange object
friend TestParticle operator + (TestParticle tp, TPChange tpc);
};

////////////////////////////////////
// this class is simply used to perform a position and a momentum
// change simultaneously
////////////////////////////////////
class TPChange
{
private:
    Vector pos_chg;
    Vector mom_chg;
public:
    void SetPosChg(Vector newpc)
    {
        pos_chg = newpc;
        return;
    }
    Vector GetPosChg()
    {
        return pos_chg;
    }
    void SetMomChg(Vector newmc)
    {
        mom_chg = newmc;
        return;
    }
    Vector GetMomChg()
    {
        return mom_chg;
    }
    // binary operators for ...
    friend TestParticle operator + (TestParticle tp, TPChange tpc);
    friend TPChange operator * (TPChange tpc, double x);
    friend TPChange operator * (double x, TPChange tpc);
};

```

```

        friend TestParticle operator + (TPChange tpc, TestParticle tp);
        friend TPChange operator + (TPChange tpc1, TPChange tpc2);
};

#endif

testparticle.cpp:

////////////////////
// test particle class
////////////////////

// operators to multiply both the momentum and the position of a TPChange
// object by a constant
TPChange operator * (TPChange tpc, double x)
{
    TPChange temp;

    temp.SetPosChg(tpc.GetPosChg() * x);
    temp.SetMomChg(tpc.GetMomChg() * x);

    return temp;
}

TPChange operator * (double x, TPChange tpc)
{
    TPChange temp;

    temp.SetPosChg(tpc.GetPosChg() * x);
    temp.SetMomChg(tpc.GetMomChg() * x);

    return temp;
}

// binary operators for addition of a TPChange object to a TestParticle obj
TestParticle operator + (TestParticle tp, TPChange tpc)
{
    TestParticle temp;
    temp.SetPos(tp.GetPos()+tpc.GetPosChg());
    temp.SetMom(tp.GetMom()+tpc.GetMomChg());
    temp.SetID(tp.GetID());
    temp.SetInternalEnergy(tp.GetInternalEnergy());

    return temp;
}

TestParticle operator + (TPChange tpc, TestParticle tp)
{
    TestParticle temp;
    temp.SetPos(tp.GetPos()+tpc.GetPosChg());
    temp.SetMom(tp.GetMom()+tpc.GetMomChg());
    temp.SetID(tp.GetID());
    temp.SetInternalEnergy(tp.GetInternalEnergy());

    return temp;
}

// binary operator for adding two TPChange objects
TPChange operator + (TPChange tpc1, TPChange tpc2)
{
    TPChange temp;
    temp.SetPosChg(tpc1.GetPosChg()+tpc2.GetPosChg());
    temp.SetMomChg(tpc1.GetMomChg()+tpc2.GetMomChg());

    return temp;
}

```


A.3 vector_and_spherical.cpp and vector_and_spherical.h

These files contain the C++ classes `Vector` and `Spherical`. These are needed to store vectors in cartesian or spherical coordinates. Functions for operations like vector addition, cross and dot products, or the conversion from cartesian to spherical coordinates (i.e. `Vector` to `Spherical` objects) are also implemented here.

vector_and_spherical.h:

```
#ifndef VECTOR_AND_SPHERICAL_H
#define VECTOR_AND_SPHERICAL_H
#include <iostream.h>
#define IOSTREAM_H
#endif
#include <math.h>
#define MATH_H
#endif

// forward declaration
class Spherical;

////////////////////
// Vector with three coordinates
////////////////////
class Vector
{
private:
    double coord[3];
public:
    // constructor
    Vector (double x, double y, double z)
    {
        SetCoords(x,y,z);
    }
    Vector ()
    {
        SetCoords(0,0,0);
    }

    // returns the coordinates of the vector
    double GetCoord(int i)
    {
        if(i>=0 && i<3)
            return coord[i];
        else
            cout << "Error in Vector::GetCoord: invalid index." << endl;
            return 0;
    }
}
```

```

// sets the coordinates
void SetCoord(int i, double val)
{
    if(i>=0 && i<3)
    {
        coord[i] = val;
        return;
    }
    else
        cout << "Error in Vector::SetCoord" << endl;
}

// sets all coordinates at once
void SetCoords(double x, double y, double z)
{
    coord[0] = x, coord[1] = y, coord[2] = z;
    return;
}

// returns the squared norm of the vector
double GetNormSquared()
{
    double normsq=0.0;      // squared norm of vector

    for(int i=0; i<3;i++)
        normsq+=coord[i]*coord[i];

    return normsq;
}

// returns the norm of the vector
double GetNorm()
{
    return sqrt(GetNormSquared());
}

// converts to a Spherical object
Spherical GetSpherical();

friend Vector operator * (double, Vector);
friend double operator ^ (Vector avec, Vector bvec);
friend Vector operator * (Vector avec, double lambda);
friend Vector operator * (Vector bvec, Vector avec);
friend Vector operator + (Vector avec, Vector bvec);
friend Vector operator - (Vector avec, Vector bvec);
};

////////////////////////////////////
// 3 component Vector in spherical coordinates
////////////////////////////////////
class Spherical
{
private:
    double r, phi, theta;
public:
    // constructors
    Spherical()
    {
        r=0, phi=0, theta=0;
    }
    Spherical(double nr, double nphi, double ntheta)
    {
        SetR(nr);
        SetPhi(nphi);
        SetTheta(ntheta);
    }
    // access functions
    double GetR()

```

```

{
    return r;
}
double GetPhi()
{
    return phi;
}
double GetTheta()
{
    return theta;
}
// sets r. r must be >=0, returns true if error occurs
bool SetR(double nr)
{
    if(nr>=0)
    {
        r=nr;
        return false;
    }
    else
    {
        cout << "Error in Spherical::SetR(): Invalid value for r: "
              << nr << endl;
        r=0;
        return true;
    }
}
// sets phi, nphi is adjusted so that 0<=phi<=2PI
void SetPhi(double nphi)
{
    phi=nphi>=0?nphi-(int)(nphi/(2*PI))*2*PI:
        nphi-(int)(nphi/(2*PI)-1)*2*PI;
    return;
}
// sets theta. theta must be >=0 and <= PI, returns true if error occurs
bool SetTheta(double ntheta)
{
    if(ntheta>=0 && ntheta <= PI)
    {
        theta = ntheta;
        return false;
    }
    else
    {
        cout << "Error in Spherical::SetTheta(): invalid value
              for theta.\n";
        theta = 0;
        return true;
    }
}
// returns vector in cartesian coordinates
Vector GetCartesian();
};

#define VECTOR_AND_SPHERICAL_H
#endif

```

vector_and_spherical.cpp:

```

#include "vector_and_spherical.h"

// binary operator for scalar product
double operator ~ (Vector avec, Vector bvec)
{
    double temp = 0;

```

```

        for(int i=0; i<3; i++)
        {
            temp+=bvec.GetCoord(i)*avec.GetCoord(i);
        }

        return temp;
    }

    // s-multiplication (right)
    Vector operator * (Vector avec, double lambda)
    {
        Vector temp;

        for(int i=0; i<3; i++)
            temp.SetCoord(i,lambda*avec.GetCoord(i));

        return temp;
    }

    // s-multiplication (left)
    Vector operator * (double lambda, Vector avec)
    {
        Vector temp;

        for(int i=0; i<3; i++)
            temp.SetCoord(i,lambda*avec.GetCoord(i));

        return temp;
    }

    // binary operator for vector product ("cross")
    Vector operator * (Vector bvec, Vector avec)
    {
        Vector temp;

        for(int i=0; i<3; i++)
            temp.SetCoord(i,bvec.GetCoord((i+1)%3)*avec.GetCoord((i+2)%3) -
            bvec.GetCoord((i+2)%3)*avec.GetCoord((i+1)%3));

        return temp;
    }

    // binary operator for vector addition
    Vector operator + (Vector avec, Vector bvec)
    {
        Vector temp;

        for(int i=0; i<3; i++)
            temp.SetCoord(i,avec.GetCoord(i)+bvec.GetCoord(i));

        return temp;
    }

    // binary operator for vector subtraction
    Vector operator - (Vector avec, Vector bvec)
    {
        Vector temp;

        for(int i=0; i<3; i++)
            temp.SetCoord(i,avec.GetCoord(i)-bvec.GetCoord(i));

        return temp;
    }

    /*****\
    * Converts a Spherical object (r,phi,theta) to a Vector

```

```

* object (cartesian coordinates).
* Needs: - math.h
* Gets: - nothing
* Returns: - Vector
\*****/
Vector Spherical::GetCartesian()
{
    Vector cart;

    cart.SetCoord(0, r*sin(theta)*cos(phi));
    cart.SetCoord(1, r*sin(theta)*sin(phi));
    cart.SetCoord(2, r*cos(theta));

    return cart;
}

\*****\
* Converts a Vector object to a Spherical object.
* Needs: - math.h
* Gets: - nothing
* Returns: - Spherical
\*****/
Spherical Vector::GetSpherical()
{
    Spherical spher;
    double temp;

    temp = atan2(this->GetCoord(1),this->GetCoord(0));

    spher.SetR(this->GetNorm());
    spher.SetPhi(temp>=0?temp:temp+2*PI);
    spher.SetTheta(acos(this->GetCoord(2)/spher.GetR()));

    return spher;
}

```

Appendix B

Source Code of the Output Programs

The source code of the two output programs written in Microsoft Visual Basic[®] will be reproduced in this part of the appendix.

B.1 output.vbp

output.vbp is a Microsoft Windows[®] based program that reads and displays the whole time development of the positions of up to 2000 test particles during a simulation run from a data file created by the simulation program. The particles are shown in a pseudo three dimensional plot for each time step. One can zoom in and out and rotate the particles around three axes. To better visualize the dynamics a line indicating the particle's velocity can be shown in its stead.

output.vbp:

```
Dim pointsperstep As Integer ' number of points per time step
Dim pointshow As Integer ' number of points actually shown
Dim step() As Double ' array in which all points will be saved
Dim xax(2) As Double ' screen coordinates for x axis
Dim yax(2) As Double ' screen coordinates for y axis
Dim zax(2) As Double ' screen coordinates for z axis
```

```

Public stepno As Integer ' number of steps
Public flag1 As Boolean ' flag indicating if files point have been loaded yet
Public nofs As Integer 'Number of time steps
Public lambda As Double ' scale factor for points
Dim gamma As Double 'scale factor for velocity lines
Public longestaxis As Double ' longest axis (on screen)
Public maxi As Double ' largest coordinate of all points' coord.s
Public mfx As Double ' middle of picture1 (horizontal)
Public mfy As Double ' middle of picture1 (vertical)
Const pi = 3.14159265358979 'guess what...

' calculates and returns minimum of two doubles
Function minimum(a As Double, b As Double) As Double
If a < b Then
minimum = a
Else
minimum = b
End If
End Function
'calculates and returns maximum of three doubles
Function maximum(a As Double, b As Double, c As Double) As Double
If a > b Then
temp = a
Else
temp = b
End If
If temp > c Then
maximum = temp
Else
maximum = c
End If
End Function
' show / don't show axis
Private Sub CheckAxis_Click()
Call draw_main
End Sub

' show / don't show velocity lines
Private Sub CheckVlines_Click()
Call draw_main
End Sub

' draw all pictures
Private Sub Command1_Click()
Call draw_main
End Sub

Private Sub Form_Load()
' auto redraw form and pictures when form is scaled...
Form1.AutoRedraw = True
Picture1.AutoRedraw = True

' use full form for picture1
Picture1.Height = Form1.Height - 620
Picture1.Width = Form1.Width - 3420

pointsperstep = find_pointsperstep 'finds points per step
pointshow = pointsperstep 'show all points by default
nofs = find_number_of_steps 'finds number of steps
gamma = 3# 'set scale factor for velocity lines to 3.0

' set flag indicating that data was not loaded yet
flag1 = False

' set maximum value for time step scrollbar

```

```

ScrollStep.Max = nofs - 1

' dimensionates array for saving points
ReDim step(nofs, pointsperstep - 1, 2)

End Sub
'resize picture1 if form is resized
Private Sub Form_Resize()
If Form1.Height - 620 > 0 Then
Picture1.Height = Form1.Height - 620
End If
If Form1.Width - 3420 > 0 Then
Picture1.Width = Form1.Width - 3420
End If
Call draw_main
End Sub

Private Sub HScrollgamma_Change()
gamma = 16# * HScrollgamma.Value / HScrollgamma.Max
Call draw_main
End Sub

Private Sub HScrollgamma_Scroll()
Call HScrollgamma_Change
End Sub

Private Sub ScrollPhi_Change()
Call draw_main
End Sub

Private Sub ScrollPhi_Scroll()
Call draw_main
End Sub

Private Sub ScrollPsi_Change()
Call draw_main
End Sub

Private Sub ScrollPsi_Scroll()
Call draw_main
End Sub

Private Sub ScrollShowPoints_Change()
Call draw_main
End Sub

Private Sub ScrollShowPoints_Scroll()
Call draw_main
End Sub

Private Sub ScrollStep_Change()
stepno = ScrollStep.Value
Call draw_main
End Sub

Private Sub ScrollStep_Scroll()
stepno = ScrollStep.Value
Call draw_main
End Sub

Private Sub ScrollTheta_Change()
Call draw_main
End Sub

Private Sub ScrollTheta_Scroll()
Call draw_main
End Sub
' loads all points from "suno_all.dat" and saves 'em to array step(,,)

```



```

Function load_points()
'On Error Resume Next
Dim fs, a 'filesystem object, filestream
Dim ret As String 'for temporary storage of read lines
Dim i As Integer 'point index
Dim s As Integer 'step index
Dim c As Integer 'coordinate index

' opens file stream
Set fs = CreateObject("Scripting.FileSystemObject")
Set a = fs.OpenTextFile("suno_all.dat", 1, False)
If Err Then
Exit Function
End If

' skip 1st line (containing the number of time steps)
a.SkipLine

' reads and saves the points
s = 0

Do While s < nofs
For i = 0 To pointsperstep - 1
    For c = 0 To 2
        ret = a.ReadLine
        step(s, i, c) = CDBl(ret) 'stores read components to array step
        ' check if current maxi is exceeded
        If Abs(step(s, i, c)) > maxi Then
            maxi = Abs(step(s, i, c))
        End If
    Next c
Next i

s = s + 1
a.SkipLine 'skips line containing the "end of time step" string
Loop
a.Close 'close file stream

End Function

' calculates mfx and mfy
Function calculate_mfx_mfy()
mfx = Picture1.ScaleWidth / 2
mfy = Picture1.ScaleHeight / 2
End Function

' calculates and draws axes, necessary before calling function draw_main()
Function draw_axis()
theta = pi * (-0.5 + ScrollTheta.Value / ScrollTheta.Max) '1st rotation around z
phi = pi * (-0.5 + ScrollPhi.Value / ScrollPhi.Max) '1st rotation around x
psi = pi * ScrollPsi.Value / ScrollPsi.Max '2nd rotation around z

' rotation z -> x -> z by theta, phi, psi
xax(0) = Cos(psi) * Cos(theta) - Sin(psi) * Cos(phi) * Sin(theta)
xax(1) = Sin(psi) * Cos(theta) + Cos(psi) * Cos(phi) * Sin(theta)
xax(2) = Sin(phi) * Sin(theta)
yax(0) = -Cos(psi) * Sin(theta) - Sin(psi) * Cos(phi) * Cos(theta)
yax(1) = -Sin(psi) * Sin(theta) + Cos(psi) * Cos(phi) * Cos(theta)
yax(2) = Sin(phi) * Cos(theta)
zax(0) = Sin(psi) * Sin(phi)
zax(1) = -Cos(psi) * Sin(phi)
zax(2) = Cos(phi)

' set longestaxis to the length of the longest axis on screen display
longestaxis = Sqr(maximum(xax(0) ^ 2 + yax(1) ^ 2, yax(0) ^ 2 +
yax(1) ^ 2, zax(0) ^ 2 + zax(1) ^ 2))

' scale factor for axes, makes sure axes are completely visible
axfac = minimum(Picture1.ScaleWidth, Picture1.ScaleHeight) / 2.1 / longestaxis

```

```

' draws and labels coordinate axes
If CheckAxis.Value = 1 Then
Picture1.Line (mfx - xax(0) * axfac, mfy + xax(1) * axfac)-(mfx + xax(0) *
axfac, mfy - xax(1) * axfac)
Picture1.PSet (mfx + xax(0) * axfac, mfy - xax(1) * axfac), RGB(255, 0, 0)
Picture1.Print "x"
Picture1.Line (mfx - yax(0) * axfac, mfy + yax(1) * axfac)-(mfx + yax(0) *
axfac, mfy - yax(1) * axfac)
Picture1.PSet (mfx + yax(0) * axfac, mfy - yax(1) * axfac), RGB(255, 0, 0)
Picture1.Print "y"
Picture1.Line (mfx - zax(0) * axfac, mfy + zax(1) * axfac)-(mfx + zax(0) *
axfac, mfy - zax(1) * axfac)
Picture1.PSet (mfx + zax(0) * axfac, mfy - zax(1) * axfac), RGB(255, 0, 0)
Picture1.Print "z"

'displays units, scale and tics for axis
'positive x-axis
lambda = minimum(Picture1.ScaleWidth, Picture1.ScaleHeight) / 2 / maxi
/ longestaxis * ScrollZoom / ScrollZoom.Max * 200
plotx = mfx + lambda * maxi * xax(0)
ploty = mfy - lambda * maxi * xax(1)
plotxend = mfx + lambda * maxi * xax(0) + axfac / 20 * yax(0)
plotyend = mfy - lambda * maxi * xax(1) - axfac / 20 * yax(1)
Picture1.Line (plotx, ploty)-(plotxend, plotyend)
Picture1.PSet (plotxend, plotyend)
Picture1.Print "+" & maxi & "[m]"
' negative x-axis
plotx = mfx - lambda * maxi * xax(0)
ploty = mfy + lambda * maxi * xax(1)
plotxend = mfx - lambda * maxi * xax(0) + axfac / 20 * yax(0)
plotyend = mfy + lambda * maxi * xax(1) - axfac / 20 * yax(1)
Picture1.Line (plotx, ploty)-(plotxend, plotyend)
Picture1.PSet (plotxend, plotyend)
Picture1.Print "-" & maxi & "[m]"
'positive y-axis
plotx = mfx + lambda * maxi * yax(0)
ploty = mfy - lambda * maxi * yax(1)
plotxend = mfx + lambda * maxi * yax(0) + axfac / 20 * xax(0)
plotyend = mfy - lambda * maxi * yax(1) - axfac / 20 * xax(1)
Picture1.Line (plotx, ploty)-(plotxend, plotyend)
Picture1.PSet (plotxend, plotyend)
Picture1.Print "+" & maxi & "[m]"
'negative y-axis
plotx = mfx - lambda * maxi * yax(0)
ploty = mfy + lambda * maxi * yax(1)
plotxend = mfx - lambda * maxi * yax(0) + axfac / 20 * xax(0)
plotyend = mfy + lambda * maxi * yax(1) - axfac / 20 * xax(1)
Picture1.Line (plotx, ploty)-(plotxend, plotyend)
Picture1.PSet (plotxend, plotyend)
Picture1.Print "-" & maxi & "[m]"
' positive z-axis
plotx = mfx + lambda * maxi * zax(0)
ploty = mfy - lambda * maxi * zax(1)
plotxend = mfx + lambda * maxi * zax(0) + axfac / 20 * xax(0)
plotyend = mfy - lambda * maxi * zax(1) - axfac / 20 * xax(1)
Picture1.Line (plotx, ploty)-(plotxend, plotyend)
Picture1.PSet (plotxend, plotyend)
Picture1.Print "+" & maxi & "[m]"
' negative z-axis
plotx = mfx - lambda * maxi * zax(0)
ploty = mfy + lambda * maxi * zax(1)
plotxend = mfx - lambda * maxi * zax(0) + axfac / 20 * xax(0)
plotyend = mfy + lambda * maxi * zax(1) - axfac / 20 * xax(1)
Picture1.Line (plotx, ploty)-(plotxend, plotyend)
Picture1.PSet (plotxend, plotyend)
Picture1.Print "-" & maxi & "[m]"

'displays tics on half axes

```

```

'positive x-axis
lambda = minimum(Picture1.ScaleWidth, Picture1.ScaleHeight) / 2
/ maxi / longestaxis * ScrollZoom / 50
plotx = mfx + 0.5 * lambda * maxi * xax(0)
ploty = mfy - 0.5 * lambda * maxi * xax(1)
plotxend = mfx + 0.5 * lambda * maxi * xax(0) + 0.8 * axfac / 20 * yax(0)
plotyend = mfy - 0.5 * lambda * maxi * xax(1) - 0.8 * axfac / 20 * yax(1)
Picture1.Line (plotx, ploty)-(plotxend, plotyend)

' negative x-axis
plotx = mfx - 0.5 * lambda * maxi * xax(0)
ploty = mfy + 0.5 * lambda * maxi * xax(1)
plotxend = mfx - 0.5 * lambda * maxi * xax(0) + 0.8 * axfac / 20 * yax(0)
plotyend = mfy + 0.5 * lambda * maxi * xax(1) - 0.8 * axfac / 20 * yax(1)
Picture1.Line (plotx, ploty)-(plotxend, plotyend)

'positive y-axis
plotx = mfx + 0.5 * lambda * maxi * yax(0)
ploty = mfy - 0.5 * lambda * maxi * yax(1)
plotxend = mfx + 0.5 * lambda * maxi * yax(0) + 0.8 * axfac / 20 * xax(0)
plotyend = mfy - 0.5 * lambda * maxi * yax(1) - 0.8 * axfac / 20 * xax(1)
Picture1.Line (plotx, ploty)-(plotxend, plotyend)

'negative y-axis
plotx = mfx - 0.5 * lambda * maxi * yax(0)
ploty = mfy + 0.5 * lambda * maxi * yax(1)
plotxend = mfx - 0.5 * lambda * maxi * yax(0) + 0.8 * axfac / 20 * xax(0)
plotyend = mfy + 0.5 * lambda * maxi * yax(1) - 0.8 * axfac / 20 * xax(1)
Picture1.Line (plotx, ploty)-(plotxend, plotyend)

' positive z-axis
plotx = mfx + 0.5 * lambda * maxi * zax(0)
ploty = mfy - 0.5 * lambda * maxi * zax(1)
plotxend = mfx + 0.5 * lambda * maxi * zax(0) + 0.8 * axfac / 20 * xax(0)
plotyend = mfy - 0.5 * lambda * maxi * zax(1) - 0.8 * axfac / 20 * xax(1)
Picture1.Line (plotx, ploty)-(plotxend, plotyend)

' negative z-axis
plotx = mfx - 0.5 * lambda * maxi * zax(0)
ploty = mfy + 0.5 * lambda * maxi * zax(1)
plotxend = mfx - 0.5 * lambda * maxi * zax(0) + 0.8 * axfac / 20 * xax(0)
plotyend = mfy + 0.5 * lambda * maxi * zax(1) - 0.8 * axfac / 20 * xax(1)
Picture1.Line (plotx, ploty)-(plotxend, plotyend)
End If

End Function
Function draw_points()

'scale factor for points
lambda = minimum(Picture1.ScaleWidth, Picture1.ScaleHeight) / 2 / maxi
/ longestaxis * ScrollZoom / 50

'scale factor for distz
sc_distz = 127 / maxi / longestaxis / 2

For i = 0 To pointshow - 1
' calculates actual screen locations for plotting points
plotx = mfx + lambda * (step(stepno, i, 0) * xax(0) + step(stepno, i, 1)
* yax(0) + step(stepno, i, 2) * zax(0))
ploty = mfy - lambda * (step(stepno, i, 0) * xax(1) + step(stepno, i, 1)
* yax(1) + step(stepno, i, 2) * zax(1))

' calculates distance orthogonal to the screen from point to screen
distz% = 127 + sc_distz * (step(stepno, i, 0) * xax(2) + step(stepno, i, 1)
* yax(2) + step(stepno, i, 2) * zax(2))

' displays velocity lines if corresponding box is checked
If CheckVlines.Value = 1 And stepno > 0 Then

```

```

plotxold = mfx + lambda * (step(stepno - 1, i, 0) * xax(0) +
step(stepno - 1, i, 1) * yax(0) + step(stepno - 1, i, 2) * zax(0))
plotyold = mfy - lambda * (step(stepno - 1, i, 0) * xax(1) +
step(stepno - 1, i, 1) * yax(1) + step(stepno - 1, i, 2) * zax(1))
Picture1.Line (plotx, ploty)-(plotx + gamma * (plotxold - plotx), ploty
+ gamma * (plotyold - ploty)), RGB(255 - distz%, 255 - distz%, 255 - distz%)
Else
' displays dots otherwise, dots are bigger and darker
' the closer they are to the screen
Picture1.DrawWidth = distz% / 80 + 1
Picture1.PSet (plotx, ploty), RGB(255 - distz%, 255 - distz%, 255 - distz%)
'Picture1.Circle (plotx, ploty), distz% / 5, RGB(255, 0, 0)
'Picture1.Line (plotx, ploty)-(plotx + 20, ploty + 20), , BF
End If
Next i
Picture1.DrawWidth = 1
End Function

Private Sub ScrollZoom_Change()
Call draw_main
End Sub

Private Sub ScrollZoom_Scroll()
Call draw_main
End Sub
' finds and returns the number of points per time step
Function find_pointsperstep() As Integer

Dim fs, a 'filesystem object, file stream
Dim ret As String
Dim i As Integer 'counts number of lines

' open file stream
Set fs = CreateObject("Scripting.FileSystemObject")
Set a = fs.OpenTextFile("suno_all.dat", 1, False)
If Err Then
Exit Function
End If

' skip 1st line (containing the total number of steps)
a.SkipLine

i = -1
ret = ""

' counts number of lines until end of timestep is reached
Do While ret <> "end of timestep"
ret = a.ReadLine
i = i + 1
Loop

' 3 lines = one point
find_pointsperstep = i / 3
a.Close
End Function

Function draw_main()
' loads data if this is the first call of the function
If flag1 = False Then
schrott = MsgBox("All data will be loaded now. This may take a while...",
vbOKOnly)

' save points for all time steps in array step(,,)
Call load_points
flag1 = True
End If

```

```

' calculate number of points to be shown
pointshow = pointstep * (ScrollShowPoints.Value / ScrollShowPoints.Max)

' display how many points are shown
Label7.Caption = "Showing " & CStr(pointshow) & " pts."

' cleans old picture, draws new one
Picture1.Cls
Call calculate_mfx_mfy
Call draw_axis
Call draw_points

' shows number of time step
Form1.Label4 = stepno

End Function
' reads the total number of time steps from the 1st line of suno_all.dat
Function find_number_of_steps() As Integer
Dim fs, a
Dim x As Integer

Set fs = CreateObject("Scripting.FileSystemObject")
Set a = fs.OpenTextFile("suno_all.dat", 1, False)
If Err Then
Exit Function
End If

' read 1st line of file, convert to integer
x = a.readline

find_number_of_steps = x
a.Close
End Function

```

B.2 Suno Density Output.vbp

Suno Density Output.vbp is the second Microsoft Windows[®] based program that reads and displays the whole time development of the mass density in a slice through the x-z-plane of the core from a data file created by the simulation program. The densities are indicated by different colors. A logarithmic density scale is implemented to cover the full density range occurring in our simulations. A “cutoff” density below which the color corresponding to the lowest density is always taken can be adjusted. One can also zoom in and out.

Suno Density Output.vbp:

```

Dim big_r_max As Double 'greatest r_max value
Dim r_max() As Double 'array containing r_max-values for all time steps
Dim time() As Double 'array containing time for all time steps
Dim dens() As Double 'array containing densities for all boxes and time steps
Dim max_dens As Double 'maximum density in all time steps and boxes
Dim r_sep() As Double 'actual r-positions of seperations in r-direction
Dim theta_sep() As Double 'theta positions of separators in theta-direction
Dim i_cos_theta_max As Integer 'number of seperations in theta-direction
Dim i_r_max As Integer 'number of seperations in r-direction
Dim step As Integer ' current time step
Dim PI As Double
Dim centerx As Integer
Dim centery As Integer
Dim max_radius As Integer
Dim nofs As Integer 'number of time steps

```

```

Function load_data()

```

```

Dim fs, a 'filesystem object, filestream
Dim ret As String 'for temprary storage of read lines
Dim i_r As Integer
Dim i_cos_theta As Integer
Dim i As Integer
Dim s As Integer 'step index

```

```

' opens file stream
Set fs = CreateObject("Scripting.FileSystemObject")
Set a = fs.OpenTextFile("suno_dens.dat", 1, False)
If Err Then
Exit Function
End If

```

```

' reads number of time steps from file
ret = a.readline
nofs = CInt(ret)

```

```

'reads i_r_max and i_cos_theta_max from file
ret = a.readline
i_r_max = CInt(ret)
ret = a.readline
i_cos_theta_max = CInt(ret)

```

```

' reads r-positions of seperations in r-direction from file
ReDim r_sep(i_r_max)
For i_r = 0 To i_r_max - 1
ret = a.readline
r_sep(i_r) = CDBl(ret)
Next i_r
r_sep(i_r_max) = 1

```

```

'dimensionates array for theta-positions of separators in theta-direction
ReDim theta_sep(i_cos_theta_max)

```

```

'reads density maps and r_max for all steps from file
ReDim r_max(nofs - 1)
ReDim time(nofs - 1)
ReDim dens(nofs - 1, i_r_max - 1, 1, i_cos_theta_max - 1)

```

```

For s = 0 To nofs - 1
ret = a.readline
r_max(s) = CDBl(ret)
ret = a.readline
time(s) = CDBl(ret)
If r_max(s) > big_r_max Then
big_r_max = r_max(s)
End If
For i_r = 0 To i_r_max - 1

```

```

For i = 0 To 1
For i_cos_theta = 0 To i_cos_theta_max - 1
ret = a.readline
dens(s, i_r, i, i_cos_theta) = CDBl(ret)
If dens(s, i_r, i, i_cos_theta) > max_dens Then
max_dens = dens(s, i_r, i, i_cos_theta)
End If
Next i_cos_theta
Next i
Next i_r
Next s

a.Close

End Function

Private Sub Command1_Click()
If Check1.Value = 1 Then
VScrollZoom.Value = 5 'set standard zoom factor
End If

For i% = 0 To nofs - 1
HScrollTimeStep.Value = i%
' modify zoomfactor if necessary and check1 checked
If Check1.Value = 1 Then
If r_max(step) / big_r_max * VScrollZoom.Value / VScrollZoom.Max * 200 < 0.5 Then
VScrollZoom.Value = VScrollZoom.Value * 2
Else
If r_max(step) / big_r_max * VScrollZoom.Value / VScrollZoom.Max * 200 > 1.1 Then
VScrollZoom.Value = VScrollZoom.Value / 2
End If
End If
End If

Call HScrollTimeStep_Change
SavePicture Picture1.Image, "SuNo" & get4digitnumber(i%) & ".bmp"
Next i%
End Sub

Private Sub Command2_Click()
SavePicture Picture1.Image, "SuNo" & get4digitnumber(HScrollTimeStep.Value)
& ".bmp"
End Sub

Private Sub Form_Load()
' auto redraw form and pictures when form is scaled...
PI = 3.141592
Form1.AutoRedraw = True
Picture1.AutoRedraw = True
Picture1.FillStyle = 1

big_r_max = 0#
step = 0
maxdens = 0#

' use full form for picture1
Picture1.Height = Form1.Height - 620
Picture1.Width = Form1.Width - 3420

MsgBox ("All data will be loaded now. This may take a while...")
Call load_data

HScrollTimeStep.Max = nofs - 1
HScrollTimeStep.Value = 0

Call draw_density_key

Label2.Caption = CStr(max_dens) & " kg/m^3"

```

```

Label3.Caption = CStr(Exp(HScrollDens.Value / HScrollDens.Max *
(Log(max_dens) - 1))) & " kg/m^3"

Label4.Caption = "Radius in plot: " & CStr(big_r_max) & " m"
End Sub

Private Sub Form_Resize()
'resize picture1 if form is resized
Picture1.Cls

If Form1.Height - 620 > 0 Then
Picture1.Height = Form1.Height - 620
End If
If Form1.Width - 3420 > 0 Then
Picture1.Width = Form1.Width - 3420
End If
Call calculate_drawseps
Call draw_densities

If Form1.Height - 685 > 0 Then
Label3.Top = Form1.Height - 685
End If
Call draw_density_key

End Sub

Function calculate_drawseps()
Dim i_cos_theta As Integer
Dim costheta As Double

' calculates center of picture1
centerx = Picture1.ScaleWidth / 2
centery = Picture1.ScaleHeight / 2

' set maximum radius for const.-r-circles
max_radius = minimum(CInt(centerx), CInt(centery))

' calculates theta-seperators
For i_cos_theta = 0 To i_cos_theta_max
costheta = (2# * CDbl(i_cos_theta) / CDbl(i_cos_theta_max) - 1#)
theta_sep(i_cos_theta) = arccos(costheta)
Next i_cos_theta

End Function

Function minimum(a As Integer, b As Integer) As Integer
If a > b Then
minimum = b
Else
minimum = a
End If

End Function

Function draw_densities()
Picture1.Cls
Dim i_r, i, i_cos_theta As Integer
Dim nextdraw, rdraw As Long
Dim color As Long
Dim costheta, sintheta, nextcostheta, nextsintheta As Double
Dim rfactor As Double
Dim temp As Double

Picture1.DrawWidth = VScrollRes.Value + 1

rfactor = r_max(step) * max_radius / big_r_max * VScrollZoom.Value

```



```

/ VScrollZoom.Max * 200
For i = 0 To 1
For i_r = 0 To i_r_max - 1
nextdraw = r_sep(i_r + 1) * rfactor
For i_cos_theta = 0 To i_cos_theta_max - 1
rdraw = r_sep(i_r) * rfactor
'color = CInt(dens(step, i_r, i, i_cos_theta) / (max_dens *
(1 - HScrollDens.Value / (HScrollDens.Max + 1))) * 255)
'If color > 255 Then
'color = 255
'End If

If i = 0 Then
theta_b = mod2pi(-theta_sep(i_cos_theta + 1) + 5 * PI / 2)
theta_s = mod2pi(-theta_sep(i_cos_theta) + 5 * PI / 2)
Else
theta_s = theta_sep(i_cos_theta + 1) + PI / 2
theta_b = theta_sep(i_cos_theta) + PI / 2
End If

'color = CLng(dens(step, i_r, i, i_cos_theta) / max_dens /
(1 - HScrollDens.Value / (HScrollDens.Max + 1)) * 1020)
If dens(step, i_r, i, i_cos_theta) <> 0 Then
temp = HScrollDens.Value / HScrollDens.Max * (Log(max_dens) - 1)
color = CLng((Log(dens(step, i_r, i, i_cos_theta)) - temp) /
(Log(max_dens) - temp) * 1275)
Else
color = 0
End If

If color > 1275 Then
color = 1275
Else
If color < 0 Then
color = 0
End If
End If

Do While rdraw < nextdraw
If color <= 255 Then
Picture1.Circle (centerx, centery), rdraw, RGB(255, 255, 255 - color),
theta_s, theta_b
Else
If color > 255 And color <= 511 Then
Picture1.Circle (centerx, centery), rdraw, RGB(511 - color, 255, 0),
theta_s, theta_b
Else
If color > 511 And color <= 767 Then
Picture1.Circle (centerx, centery), rdraw, RGB(0, 767 - color, color -
511),
theta_s, theta_b
Else
If color > 767 And color <= 1023 Then
Picture1.Circle (centerx, centery), rdraw, RGB(color - 768, 0, 1023 -
color),
theta_s, theta_b
Else
If color > 1023 Then
Picture1.Circle (centerx, centery), rdraw, RGB((1 - ((color - 1024) /
255) ^ 4) * 255, 0, 0),
theta_s, theta_b
End If
End If
End If
End If
End If

rdraw = rdraw + VScrollRes.Value

```

```

Loop
Next i_cos_theta
Next i_r
Next i

Call draw_density_key

End Function

Private Sub HScrollDens_Change()
'Label2.Caption = CStr(max_dens / (10 ^ (HScrollDens.Value / 10))) & "[kg/m^3]"
Label3.Caption = CStr(Exp(HScrollDens.Value / HScrollDens.Max *
(Log(max_dens) - 1)))
& " kg/m^3"
Call draw_densities
End Sub

Private Sub HScrollDens_Scroll()
'Label2.Caption = CStr(max_dens / (10 ^ (HScrollDens.Value / 10))) & "[kg/m^3]"
Label3.Caption = CStr(Exp(HScrollDens.Value / HScrollDens.Max *
(Log(max_dens) - 1)))
& " kg/m^3"
Call draw_densities
End Sub

Private Sub HScrollTimeStep_Change()

Picture1.DrawWidth = 1
step = HScrollTimeStep.Value
Label1.Caption = "Time step: " & CStr(step)
Call draw_densities
End Sub

Function draw_density_key()
'draws density key
'For i% = 0 To 255
'Picture2.PSet (Picture2.Width / 2, Picture2.Height / 255 * i%), RGB(255 - i%, i%, 0)
'Next i%

Picture1.DrawWidth = 20

' show radius in plot
Picture1.CurrentX = Picture1.ScaleWidth - 180
Picture1.CurrentY = 1
Picture1.Print Label4.Caption

Picture1.CurrentX = 25
Picture1.CurrentY = 1
Picture1.Print Label2.Caption

' show time in plot
Picture1.CurrentX = Picture1.ScaleWidth - 120
Picture1.CurrentY = Picture1.ScaleHeight - 15

Picture1.Print CStr(time(step)) & "s"

' labels for scale
Picture1.CurrentX = 25
Picture1.CurrentY = Picture1.ScaleHeight - 15
Picture1.Print Label3.Caption

Picture1.CurrentX = 25
Picture1.CurrentY = 1
Picture1.Print Label2.Caption

temp = HScrollDens.Value / HScrollDens.Max * (Log(max_dens) - 1)
For denslabel = 1 To 17
If 10 ^ denslabel < max_dens Then

```

```

labelpos = Picture1.ScaleHeight * (1 - (denslabel * Log(10) - temp)
/ (Log(max_dens) - temp))
Picture1.CurrentX = 20
Picture1.CurrentY = labelpos
Picture1.Print "1e" & CStr(denslabel)
End If
Next denslabel

For i% = 0 To 255
'Picture2.PSet (Picture2.Width / 2, Picture2.Height - Picture2.Height
/ 1278 * i%), RGB(255, 255, 255 - i%)
Picture1.PSet (10, Picture1.ScaleHeight - Picture1.ScaleHeight / 1278
* i%), RGB(255, 255, 255 - i%)
Next i%

For i% = 256 To 511
'Picture2.PSet (Picture2.Width / 2, Picture2.Height - Picture2.Height
/ 1278 * i%), RGB(511 - i%, 255, 0)
Picture1.PSet (10, Picture1.ScaleHeight - Picture1.ScaleHeight
/ 1278 * i%), RGB(511 - i%, 255, 0)
Next i%

For i% = 512 To 767
'Picture2.PSet (Picture2.Width / 2, Picture2.Height - Picture2.Height
/ 1278 * i%), RGB(0, 767 - i%, i% - 511)
Picture1.PSet (10, Picture1.ScaleHeight - Picture1.ScaleHeight
/ 1278 * i%), RGB(0, 767 - i%, i% - 511)
Next i%

For i% = 768 To 1023
'Picture2.PSet (Picture2.Width / 2, Picture2.Height - Picture2.Height
/ 1278 * i%), RGB(i% - 768, 0, 1023 - i%)
Picture1.PSet (10, Picture1.ScaleHeight - Picture1.ScaleHeight
/ 1278 * i%), RGB(i% - 768, 0, 1023 - i%)
Next i%

For i% = 1023 To 1278
'Picture2.PSet (Picture2.Width / 2, Picture2.Height - Picture2.Height
/ 1278 * i%), RGB((1 - ((i% - 1023) / 255) ^ 4) * 255, 0, 0)
Picture1.PSet (10, Picture1.ScaleHeight - Picture1.ScaleHeight
/ 1278 * i%), RGB((1 - ((i% - 1023) / 255) ^ 4) * 255, 0, 0)
Next i%

End Function

Function arcsin(x As Double) As Double
If x <> 1 And x <> -1 Then
arcsin = Atn(x / Sqr(-x * x + 1))
ElseIf x = 1 Then
arcsin = PI / 2
ElseIf x = -1 Then
arcsin = -PI / 2
End If
End Function

Function arccos(x As Double) As Double
If x <> 1 And x <> -1 Then
arccos = Atn(-x / Sqr(-x * x + 1)) + 2 * Atn(1)
ElseIf x = 1 Then
arccos = 0
ElseIf x = -1 Then
arccos = PI
End If
End Function

Private Sub VScrollRes_Change()
Picture1.Cls

```

```

Call draw_densities
End Sub

Private Sub VScrollRes_Scroll()
Picture1.Cls
Call draw_densities
End Sub

Private Sub VScrollZoom_Change()
Label4.Caption = "Radius in plot: " & CStr(big_r_max /
(VScrollZoom.Value
/ VScrollZoom.Max * 200)) & " m"
Picture1.Cls
Call draw_densities
End Sub

Private Sub VScrollZoom_Scroll()
Label4.Caption = "Radius in plot: " & CStr(big_r_max /
(VScrollZoom.Value
/ VScrollZoom.Max * 200)) & " m"
Picture1.Cls
Call draw_densities
End Sub

'converts an arbitrary angle to a number between 0 and 2pi
Function mod2pi(x As Double) As Double
Dim n As Integer
n = CInt(x / (2 * PI) - 0.5)
mod2pi = x - n * 2 * PI
End Function

' converts as integer > 0 into a 4 digit string by adding zeros in front
Function get4digitnumber(i As Integer) As String
If i < 10 Then
get4digitnumber = "000" & CStr(i)
Else
If i < 100 Then
get4digitnumber = "00" & CStr(i)
Else
If i < 1000 Then
get4digitnumber = "0" & CStr(i)
Else
If i < 10000 Then
get4digitnumber = CStr(i)
Else
Form1.Print "Error"
Exit Function
End If
End If
End If
End If
End Function

```

Appendix C

Abbreviations and Symbols

A	mass number of a nucleus
c	speed of light $\approx 2.99792458 \times 10^8 \frac{\text{m}}{\text{s}}$
E_G	gravitational energy of core
E_{int}	(total) internal energy of core
E_{kin}	(total) kinetic energy of core
E_{tot}	total energy of core
EOS	equation of state
EsOS	equations of state
f	function defining the locations of the grid boundaries for the r coordinate
$\vec{F}_{EOS,j}$	force on test particle j due to the EOS
$\vec{F}_{G,j}$	force on test particle j due to gravity
G	gravitation constant $\approx 6.67259 \times 10^{-11} \frac{\text{m}^3}{\text{kg s}^2}$
\hbar	Planck's constant $= \frac{h}{2\pi} \approx \frac{6.6262}{2\pi} \times 10^{-34} \text{Js}$
k_B	Boltzmann's constant $\approx 1.3807 \times 10^{-23} \frac{\text{J}}{\text{K}}$
LS EOS	Lattimer & Swesty EOS
\vec{L}	(total) angular momentum of core
M_\odot	solar mass $\approx 1.989 \times 10^{30} \text{kg}$
m_B	baryon mass $\approx 1.6749286 \times 10^{-27} \text{kg}$
M_{IC}	mass of the iron core of the star
m_{tp}	mass of a test particle
M_{star}	mass of the entire star
N_{cells}	number of grid cells
$(n_r, n_\phi, n_{\cos \theta})$	grid coordinates (integers defining a grid cell)

N_ϕ	number of grid boundaries for the ϕ coordinate
$N_{\cos \theta}$	number of grid boundaries for the θ coordinate
N_r	number of grid boundaries for the r coordinate
N_{tp}	number of test particles
p	pressure
\vec{p}_j	momentum vector of test particle j
r	distance from the center of the star (or coordinate system)
R	radial location of the edge of the grid
\vec{r}_j	position vector of test particle j
ρ	(mass) density
ρ_0	saturation density of (isospin symmetric) nuclear matter
ρ_{min}	minimum density enforced
ρ_{max}	maximum density in a simulation run
T	temperature
t_{bounce}	time of core bounce in a simulation run
u_{int}	internal energy per baryon
$Vol(n_r, n_\phi, n_{\cos \theta}), Vol(n_r)$	volume of the grid cell $(n_r, n_\phi, n_{\cos \theta})$
Y_e	electron fraction
Y_L	lepton fraction
Z	charge number of a nucleus
ZAMS	zero age main sequence

BIBLIOGRAPHY

Bibliography

- [1] W.D. Arnett. Neutrino trapping during gravitational collapse of stars. *Astrophys. J.*, 218:815–833, December 1977.
- [2] E. Baron and J. Cooperstein. The effect of iron core structure on supernovae. *Astrophys. J.*, 353:597–611, April 1990.
- [3] E. Baron, J. Cooperstein, and S. Kahana. Supernovae and the nuclear equation of state at high densities. *Nucl. Phys. A*, 440:744–754, 1985.
- [4] H.A. Bethe. Supernova mechanisms. *Rev. Mod. Phys.*, 62(4):801–866, October 1990.
- [5] S. Bonazzola and J.A. Marck. Efficiency of gravitational radiation from axisymmetric and 3d stellar collapse. *Astron. Astrophys.*, 267:623–633, 1993.
- [6] I.N. Bronstein, K.A. Semendjajew, G. Musiol, and H. Mühlig. *Taschenbuch der Mathematik*. Verlag Harri Deutsch, Frankfurt am Main, third edition, 1997.
- [7] S.W. Bruenn, K.R. De Nisco, and A. Mezzacappa. General relativistic effects in the core collapse supernova mechanism. *Astrophys. J.*, 560:326–338, October 2001.
- [8] A. Burrows. Supernova explosions in the universe. *Nature*, 403:727–733, February 2000.
- [9] A. Burrows and J. Goshy. A theory of supernova explosions. *Astrophys. J.*, 416:L75–L78, October 1993.
- [10] A. Burrows, J. Hayes, and B. Fryxell. On the nature of core-collapse supernova explosions. *Astrophys. J.*, 450:830–850, September 1995.
- [11] B.W. Carrol and D.A. Ostlie. *Modern Astrophysics*. Addison-Wesley, 1996.
- [12] C.L. Fryer. Mass limits for black hole formation. *Astrophys. J.*, 522:413–418, September 1999.
- [13] J. Cooperstein. The equation of state in supernovae. *Nucl. Phys. A*, 438:722–739, 1985.
- [14] J. Cooperstein and J. Wambach. Electron capture in stellar collapse. *Nucl. Phys. A*, 420:591–620, 1984.

- [15] H. Dimmelmeier, J.A. Font, and E. Mueller. Gravitational waves from relativistic rotational core collapse. *Astrophys. J.*, 560:L163–L166, October 2001.
- [16] C.L. Fryer. 3d sph core collapse simulations. <http://qso.lanl.gov/clf/main.html>, January 2002.
- [17] C.L. Fryer. private communication, January 2002.
- [18] C.L. Fryer and A. Heger. Core-collapse simulations of rotating stars. *Astrophys. J.*, 541:1033–1050, October 2000.
- [19] R.A. Gingold and J.J. Monaghan. Smoothed particle hydrodynamics: theory and application to non-spherical stars. *Mon. Not. R. astr. Soc.*, 181:375–389, 1977.
- [20] A. Heger, N. Langer, and S.E. Woosley. Presupernova evolution of rotating massive stars. i. numerical method and evolution of the internal stellar structure. *Astrophys. J.*, 528:368–396, January 2000.
- [21] M. Herant and W. Benz. Postexplosion hydrodynamics of sn 1987a. *Astrophys. J.*, 387:294–308, March 1992.
- [22] M. Herant, W. Benz, and S. Colgate. Postcollapse hydrodynamics of sn1987a: Two-dimensional simulations of the early evolution. *Astrophys. J.*, 395:642–653, August 1992.
- [23] M. Herant, W. Benz, W.R. Hix, C.L. Fryer, and S.A. Colgate. Inside the supernova: A powerful convective engine. *Astrophys. J.*, 435:339–361, November 1994.
- [24] H.-Th. Janka. Conditions for shock revival by neutrino heating in core-collapse supernovae. *Astron. Astrophys.*, 368:527–560, 2001.
- [25] H.-Th. Janka, Th. Zwerger, and R. Mönchmeyer. Does artificial viscosity destroy prompt type-ii supernova explosions ? *Astron. Astrophys.*, 268:360–368, 1993.
- [26] J. Lattimer, A. Burrows, and A. Yahil. Type ii supernova energetics. *Astrophys. J.*, 288:644–652, January 1985.
- [27] J.M. Lattimer and F.D. Swesty. Equation of state version 2.7 (ls eos v2.7). <http://www.ess.sunysb.edu/dswesty/lseos.html>.
- [28] J.M. Lattimer and F.D. Swesty. A generalized equation of state for hot, dense matter. *Nucl. Phys. A*, 535:331–367, 1991.
- [29] J.M. LeBlanc and J.R. Wilson. A numerical example of the collapse of a rotating magnetized star. *Astrophys. J.*, 161:541–551, August 1970.
- [30] B.-A. Li, C.M. Ko, and W. Bauer. Isospin physics in heavy-ion collisions at intermediate energies. *Int. J. Mod. Phys.*, 7(2):147–229, April 1998.
- [31] M. Liebendörfer, A. Mezzacappa, F.-K. Thielemann, O.E.B. Messer, W.R. Hix, and S.W. Bruenn. Probing the gravitational well: No supernova explosion in

- spherical symmetry with general relativistic boltzmann neutrino transport. *Phys. Rev. D*, 63(103004), May 2001.
- [32] O.E.B. Messer, A. Mezzacappa, S.W. Bruenn, and M.W. Guidry. A comparison of boltzmann and multigroup flux-limited diffusion neutrino transport during the postbounce shock reheating phase in core-collapse supernovae. *Astrophys. J.*, 507:353–360, November 1998.
 - [33] A. Mezzacappa, A.C. Calder, S.W. Bruenn, J.M. Blondin, M.W. Guidry, M.R. Strayer, and A.S. Umar. The interplay between proto-neutron star convection and neutrino transport in core-collapse supernovae. *Astrophys. J.*, 493:848–862, February 1998.
 - [34] A. Mezzacappa, M. Liebendörfer, O.E.B. Messer, W.R. Hix, F.-K. Thielemann, and S.W. Bruenn. Simulation of the spherically symmetric stellar core collapse, bounce, and postbounce evolution of a star of 13 solar masses with boltzmann neutrino transport, and its implications for the supernova mechanism. *Phys. Rev. Lett.*, 86(10):1935, March 2001.
 - [35] R. Mönchmeyer and E. Müller. *Timing Neutron Stars*, volume 262 of *NATO ASI Ser. C*. ASI, New York, 1989.
 - [36] W.H. Press, S.A. Teukolsky, W.T. Vetterling, and B.P. Flannery. *NUMERICAL RECIPES in C*. Press Syndicate of the University of Cambridge, Cambridge, UK, second edition, 1988.
 - [37] M. Rampp and H.-T. Janka. Spherically symmetric simulation with boltzmann neutrino transport of core collapse and postbounce evolution of a $15m_{\odot}$ star. *Astrophys. J.*, 539:L33–L36, August 2000.
 - [38] M. Rampp, E. Müller, and M. Ruffert. Simulations of non-axisymmetric rotational core collapse. *Astron. Astrophys.*, 332:969–983, 1998.
 - [39] H. Shen, H. Toki, K. Oyamatsu, and K. Sumiyoshi. Relativistic equation of state of nuclear matter for supernova and neutron star. *Nucl. Phys. A*, 637(3):435–450, May 1998.
 - [40] H. Shen, H. Toki, K. Oyamatsu, and K. Sumiyoshi. Relativistic eos table. <http://physics.senkou.numazu-ct.ac.jp/sumi/eos/>, March 2001.
 - [41] K. Sumiyoshi, H. Shen, K. Oyamatsu, M. Terasawa, H. Suzuki, S. Yamada, and H. Toki. Unstable nuclei and eos table for supernova explosion and r-process in relativistic many body approach. *RIKEN Review*, 26, January 2000.
 - [42] K. Sumiyoshi, M. Terasawa, H. Suzuki, S. Yamada, H. Toki, G.J. Mathews, and T. Kajino. Relativistic simulations of supernovae and the r-process; a new relativistic eos and nuclear reaction network. *Nucl. Phys. A*, 688:478c–480c, 2001.
 - [43] K. Sumiyoshi and H. Toki. Relativistic equation of state of nuclear matter for the supernova explosion and the birth of neutron stars. *Astrophys. J.*, 422:700–718, February 1994.

- [44] F.D. Swesty, J.M. Lattimer, and E.S. Myra. The role of the equation of state in the "prompt phase" of type ii supernovae. *Astrophys. J.*, 425:195–204, April 1994.
- [45] K. Takahashi, M.F. El Eid, and W. Hillebrandt. Beta transition rates in hot and dense matter. *Astron. Astrophys.*, 67:185–197, 1978.
- [46] F.X. Timmes. The helmholtz eos. <http://flash.uchicago.edu/fxt/eos.shtml>.
- [47] F.X. Timmes and D. Arnett. The accuracy, consistency, and speed of five equations of state for stellar hydrodynamics. *Astrophys. J. Suppl. S.*, 125:277–294, November 1999.
- [48] F.X. Timmes and F.D. Swesty. The accuracy, consistency, and speed of an electron-positron equation of state based on table interpolation of the helmholtz free energy. *Astrophys. J. Suppl. S.*, 126:501–516, February 2000.
- [49] L. Wang, D.A. Howell, P. Höflich, and J.C. Wheeler. Bipolar supernova explosions. *Astrophys. J.*, 550:1030–1035, April 2001.
- [50] L. Wang and J.C. Wheeler. Supernovae are not round. *Sky and Telescope*, January 2002.
- [51] L. Wang, J.C. Wheeler, Z. Li, and A. Clocchiatti. Broadband polarimetry of supernovae: Sn 1994d, sn 1994y, sn 1994ae, sn 1995d, and sn 1995 h. *Astrophys. J.*, 467:435–445, August 1996.
- [52] T.A. Weaver, G.B. Zimmermann, and S.E. Woosley. Presupernova evolution of massive stars. *Astrophys. J.*, 225:1021–1029, November 1978.
- [53] J.R. Wilson. *Numerical Astrophysics*, page 422. Jones and Bartlett, Boston, 1985.
- [54] C.-Y. Wong. Dynamics of nuclear fluid. vii. time-dependent hartree-fock approximation from a classical point of view. *Phys. Rev. C*, 25(3):1460–1475, March 1982.
- [55] S.E. Woosley, N. Langer, and T.A. Weaver. The evolution of massive stars including mass loss: Presupernova models and explosion. *Astrophys. J.*, 411:823–839, July 1993.
- [56] S.E. Woosley and T.A. Weaver. The physics of supernova explosions. *Ann. Rev. Astron. Astrophys.*, 24:205–253, 1986.
- [57] S. Yamada and K. Sato. Numerical study of rotating core collapse in supernova explosions. *Astrophys. J.*, 434:268–276, October 1994.
- [58] S. Yamada and K. Sato. Gravitational radiation from rotational collapse of a supernova core. *Astrophys. J.*, 450:245–252, September 1995.
- [59] T. Zwerger and E. Müller. Dynamics and gravitational wave signature of axisymmetric rotational core collapse. *Astron. Astrophys.*, 320:209–227, 1997.

MICHIGAN STATE UNIVERSITY LIBRARIES



3 1293 02328 8222