Three-dimensional Parallel Hydrodynamics and Astrophysical Applications

Dissertation

zur Erlangung des Grades eines Doktors
der Naturwissenschaften
der Fakultät für Mathematik und Physik
der Eberhard-Karls-Universität zu Tübingen

vorgelegt von

Richard Günther

aus Tübingen
2005

Tag der mündlichen Prüfung:	22. April 2005
Dekan:	Prof. Dr. Peter Schmid
1. Berichterstatter:	Prof. Dr. Wilhelm Kley
2. Berichterstatter:	Prof. Dr. Hanns Ruder

Abstract

This work presents the newly developed three-dimensional parallel hydrodynamics code TraMP together with a generic C++ library for automatic parallelization of finite difference schemes on structured grids. Furthermore we present two astrophysical applications, namely the problem of oscillating n-dimensional polytropes and extensive studies on the evolution of circumbinary disks including the accretion process and spectral distribution of the emitted energy.

TraMP is based on the freely available POOMA library, which has been extensively extended and optimized for our needs. We were able to greatly improve parallel performance of the POOMA library by introducing native support for MPI and OpenMP. Extensive performance measurements have shown that the parallel performance of TraMP scales up to 128 processors and more. The implemented infrastructure for Cartesian, cylindrical and spherical coordinate systems allows for easy covariant implementation of the finite difference schemes in TraMP. In the appendix an introduction to the POOMA library is given and the C++ techniques used by POOMA are briefly presented.

The problem of n-dimensional polytropic oscillations in different geometries is tackled in an uniform description employing linear perturbation theory. With the resulting eigenfrequencies the results of non-linear multi-dimensional hydrodynamic calculations with the TraMP code are verified. This work is especially useful as it allows testing solving the Poisson equation with cheap two-dimensional simulations.

We study the evolution of circumbinary disks surrounding T Tauri binary systems using high resolution numerical simulations. For the first time we perform long-time integration of the complete system covering several hundred orbital periods of the binary and compare the properties of the evolved systems with observational data such as spectral energy distributions in the infrared and optical bands and accretion rates estimated from luminosities. These simulations include a detailed energy balance including viscous heating and radiative cooling. For close systems also irradiation from the stars is taken into account. A novel numerical approach using a parallelized Dual-Grid technique on two different coordinate systems has been implemented to solve the problem of not treating the interior of the binary orbit with the same resolution as the circumbinary disk.

Zusammenfassung

Die vorliegende Arbeit präsentiert das neuentwickelte dreidimensionale und parallele Hydrodynamik Programm TraMP zusammen mit einer generischen C++ Bibliothek zur automatischen Parallelisierung von Finite Differenzen Verfahren auf strukturierten Gittern. Zudem werden als astrophysikalische Anwendungen oszillierende n-dimensionale Polytropen und Arbeiten zur Entwicklung von zirkumbinären Scheiben, deren Akkretionsprozess und deren Abstrahlverhalten in Form von spektralen Energieverteilungen präsentiert.

TraMP basiert auf der frei verfügbaren POOMA Bibliothek, die für unsere Zwecke maßgeblich erweitert und optimiert worden ist. Durch eine neuentwickelte Parallelisierung basierend auf der MPI Bibliothek und dem OpenMP Standard wurde die Leistung der Parallelisierung deutlich verbessert. Ausführliche Messungen haben ergeben, dass die parallele Leistung von TraMP bis 128 und mehr Prozessoren skaliert. Die Infrastruktur für kartesische, zylindrische und sphärische Koordinaten ermöglicht eine einfache kovariante Formulierung der Finite Differenzen Algorithmen in TraMP. Im Anhang dieser Arbeit wird in die Benutzung der POOMA Bibliothek eingeführt und werden die wesentlichen Grundlagen der verwendeten C++ Techniken besprochen.

Das Problem der Oszillationen von n-dimensionalen Polytropen wird in einer generischen Beschreibung mittels linearer Störungstheorie analysiert. Mit Hilfe der so gewonnenen Eigenfrequenzen werden die Ergebnisse von nichtlinearen mehrdimensionalen hydrodynamischen Simulationen mit TraMP verifiziert. Diese Arbeit ist besonders nützlich für die Verifikation der Lösung der Poisson-Gleichung, da sie die Beschränkung auf zwei Dimensionen und somit weniger rechenaufwendige Simulationen ermöglicht.

Wir untersuchen die Entwicklung von zirkumbinären Scheiben um T Tauri Systeme mittels hochaufgelösten Simulationen. Zum ersten mal sind wir in der Lage, eine Langzeitintegration des kompletten Systems über mehrere hundert Orbitalperioden des Binärsystems durchzuführen und die Eigenschaften des so entwickelten Systems mit Beobachtungsdaten, wie zum Beispiel spektralen Energieverteilungen im infraroten und optischen, sowie aus Helligkeitsdaten abgeleiteten Akkretionsraten, zu vergleichen. Diese Simulationen beinhalten ein detailliertes Modell für eine Energiebilanz sowie die vertikale Heizung und Kühlung. Für enge Systeme wird zudem die Sterneinstrahlung auf die Akkretionsscheibe berücksichtigt. Für die Simulationen wurde eine neue Methode, die Dual-Grid Technik, entwickelt, um die innere Lücke im zylindrischen Koordinatensystem mit einem kartesischen System zu überdecken und somit eine gleichmäßig aufgelöste Simulation auch der inneren Region der Akkretionsscheibe zu ermöglichen.

Contents

1 Introduction **9**

2 Equations and Numerical Scheme **11**
2.1 Equations . . . 11
2.2 Operator Splitting . . . 12
2.3 Advection Scheme . . . 13
2.4 Artificial Viscosity . . . 14
2.5 Forces . . . 15
2.6 Coordinate Forces . . . 15
2.7 Viscosity . . . 15
2.8 Gravitation and N-Body . . . 16

3 Implementation and Performance **17**
3.1 Extending the POOMA Library . . . 19
3.1.1 Adding MPI and OpenMP Support . . . 19
3.1.2 HDF5 Parallel and Serial I/O . . . 20
3.1.3 Coordinate System Support . . . 21
3.1.4 Interfacing to PETSc . . . 22
3.2 Serial Performance . . . 23
3.3 Parallelization and Performance . . . 25
3.3.1 MPI Parallel Performance . . . 27
3.3.2 OpenMP Parallel Performance . . . 31
3.3.3 MPI and OpenMP Hybrid Parallel Performance . . . 32

4 Tests **35**
4.1 Shock Problems . . . 35
4.2 Accretion Disks . . . 35
4.3 Protoplanetary Disks . . . 39

5 Stellar Oscillations **45**
5.1 Equilibrium Structure . . . 46
5.2 Linear Analysis . . . 47
5.3 Boundary Conditions . . . 50
5.4 Testing the new Hydro Code . . . 50
5.4.1 One-dimensional Calculations . . . 50
5.4.2 Two- and Three-dimensional Calculations . . . 53
5.4.3 Testing the Poisson Solver . . . 55

6 Circumbinary Disks **59**
6.1 Introduction . . . 59
6.2 Physical Model . . . 60
6.2.1 General Layout . . . 60
6.2.2 Equations . . . 61
6.2.3 Viscosity and Disk Height . . . 61
6.2.4 Radiative Balance . . . 62

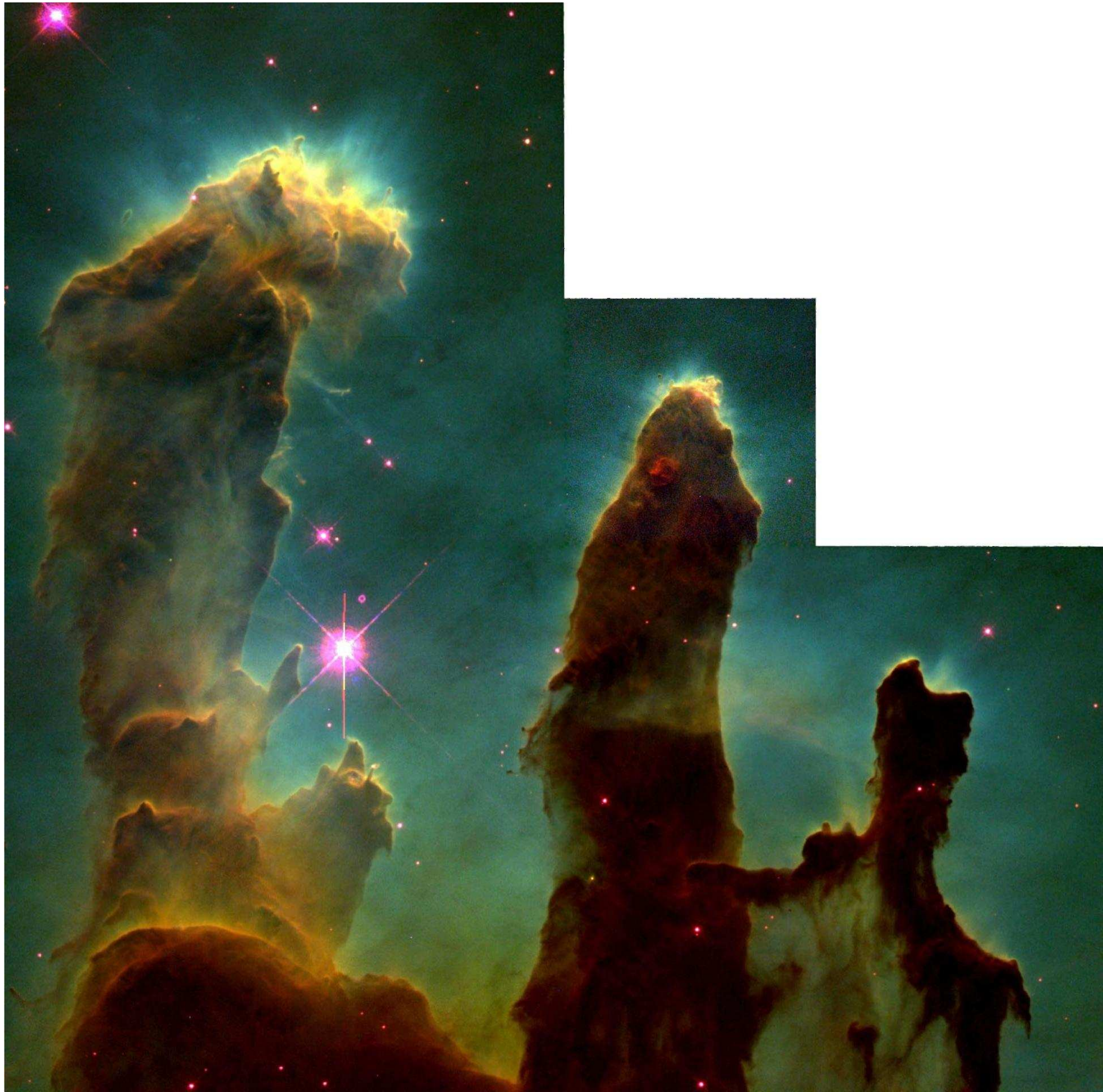
6.3 Spectra Generation . . . 63
6.4 Numerical Issues . . . 63
6.4.1 Irradiation . . . 64
6.4.2 Accretion onto the Stars . . . 64
6.4.3 Initial and Boundary Conditions . . . 65
6.4.4 Dual-Grid Technique . . . 67
6.5 Systems . . . 71
6.6 Results, General . . . 71
6.7 Results, Equal Mass Binary . . . 72
6.8 Results, Wide Systems . . . 76
6.9 Results, Spectroscopic Systems . . . 79
6.9.1 Accretion . . . 79
6.9.2 Irradiation Effects . . . 82
6.9.3 DQ Tau Circumbinary Disk . . . 83
6.9.4 AK Sco . . . 83
6.9.5 Circumstellar Disks . . . 85
6.10 Conclusion . . . 87

A Using Advanced C++ Techniques for CFD 93
A.1 Language and Library Features . . . 93
A.2 Object-Oriented Programming and Pitfalls . . . 94
A.3 Expression Templates . . . 96
A.4 Data-Flow Analysis and Out-of-order Execution . . . 98
A.5 C++ Compilers and their Problems . . . 99

B Introduction to the POOMA Library 101
B.1 Initialization and Overview . . . 101
B.2 Arrays . . . 103
B.2.1 Array Expressions . . . 103
B.2.2 Array Sub-Objects . . . 105
B.3 Parallel Setup . . . 106
B.4 Fields . . . 107
B.4.1 Meshes . . . 107
B.4.2 Materials and Centerings . . . 107
B.4.3 Relations . . . 108
B.4.4 Boundary Conditions . . . 108
B.5 Finite Differencing . . . 109
B.5.1 Array Stencils . . . 109
B.5.2 ScalarCode . . . 110
B.6 File I/O . . . 111

C Leafify Patch 113

D Supplements 117



1 Introduction

In the past years the resolution and quality of detectors at telescopes improved considerably. This made it possible to observe complex dynamical processes in the universe with better quality and resolution in all wavelengths. While most of the observations give only a snapshot at a given moment in time, it is possible to obtain a time evolution through sequential monitoring.

One goal of the theoretical astrophysics community is to describe and understand such time dependent processes by appropriate modeling and simulation. Due to the increased amount of data from the observation, models have become more realistic and now require computational resources nobody has imagined in the past.

Single computers are getting faster every year, still following Moore's law. At the same time, astrophysical problems are growing faster in demand for both computational power and memory. Both aspects can be addressed by using parallel computers. Large-scale parallel computing is now available even to undergraduate students as can be seen from the huge number of commodity PC-clusters at universities. Parallelization is the key-component of every computational fluid dynamics (CFD) application nowadays, as it allows to utilize the available computational power and often enables running a suitable simulation at all.

In the past, parallelization has been a hard and cumbersome task that only a few skilled programmers could apply. Now, with the introduction of C++ to the scientific community, parallelization can be done transparently for the programmer of a CFD application using a so called template class library. This allows rapid development and testing of new numerical techniques and physical models that need supercomputers from the start.

There are multiple approaches to CFD. In the engineering community finite element techniques are used to accurately model arbitrary shaped boundaries that occur in aeronautics and automotive engineering. For astrophysical applications there are particle based approaches such as n-body and smoothed particle hydrodynamics (SPH) methods used in cosmology and star formation theory. Those methods are suitable whenever there is a need for a dynamic computational domain or non-trivial shaped boundaries.

The method of choice for local problems and problems with simple geometry, like our studies on accretion disks, planet formation and stellar oscillations, is a finite volume method on a regular grid. This class of discretization techniques is well studied in the mathematical community and widely deployed not only in the astrophysical community. One solution technique is to compute the finite volume fluxes based on local differencing and appropriately limiting the gradients to improve the solution. Another way is to solve the Riemann problem for the set of equations which iteratively and possibly approximatively solves the initial value problem at the grid cell interfaces.

The goal of this work has been to create a new parallel CFD code based on the established RH2D (Kley 1988, 1989), TRAMP (Klahr 1998) and Nirvana (Ziegler & Yorke 1997) codes that are used at our institute. To avoid spending too much work on a parallel framework, multiple packages providing infrastructure for parallel grid-based CFD have been evaluated, among them ZeusMP (Norman 2000), Flash (Fryxell et al. 2000), Overture (Henshaw 1996), Chombo, Cactus (Allen 2000) and POOMA (Reynders et al. 1996). Most promising has been the POOMA library which is of high quality, extensively documented and allows interfacing to legacy Fortran code. Also this library uses modern C++ techniques and applies further high-level optimizations like scheduling based on data-flow

analysis. These make the POOMA library interesting from a research perspective, too.

In the following we introduce a library for development of parallel computational fluid dynamics applications, POOMA (appendix B), which is freely available. The library has been extended for the specific needs of astrophysical CFD and improved with regard to documentation, quality and parallel performance (sec. 3.1).

As an example application we present the development of a new three-dimensional parallel CFD code, TraMP, based on the serial RH2D and TRAMP codes (sec. 2-3). Applications of these codes include ongoing simulations of circumbinary disk systems with RH2D (sec. 6), three-dimensional calculations of planet-disk systems and stellar oscillation problems (sec. 5).

We start with presenting the equations of hydrodynamics in the form we are applying our discretization scheme to and continue with a detailed description of the scheme itself (sec. 2). Following this, some details are presented of the implementation of the new code and how we extended the POOMA library (sec. 3). As parallel performance is a key feature of TraMP, we follow up these tests with a section on parallelization and performance (sec. 3.3). Then we present numerical tests we have performed to validate the numerical scheme and its implementation (sec. 4).

Following this we present our work on stellar oscillations that have been used both as a test problem and real physical application. This includes a detailed linear analysis of the problem (sec. 5).

The next part is exclusively on one application, the dynamical evolution of circumbinary disks with the focus on accretion phenomena and matching synthetic spectra with observations (sec. 6). Here we were able to do high-resolution long-time integration of a binary system with a circumbinary disk including a detailed energy-balance and irradiation of the disk from the stars.

In the appendix you will find some notes on the use of advanced C++ techniques within the POOMA library (appendix A) and a tutorial introduction to the POOMA library and its features (appendix B).

2 Equations and Numerical Scheme

In this section we present the equations of non-ideal hydrodynamics, the so called *Navier-Stokes equations*. The discussion will focus on the specific formulation and splitting that is used in RH2D (Kley 1988, 1989), TRAMP (Klahr 1998) and the parallelized version TraMP presented in the following.

The numerical scheme is applied on a staggered mesh (fig. 1) which defines scalar quantities cell centered and vector quantities face centered. This has the advantage that finite differences are centered and hence automatically of second order in space. It is closely modeled following the Zeus code (Stone & Norman 1992), and uses an advection scheme introduced by van Leer (1977).

2.1 Equations

We solve the Navier-Stokes equations with gravitational forces. As primitive variables we choose the mass density ϱ, the components of the velocity $\mathbf{u}$ and the temperature T which is related to the specific internal energy $\varepsilon = C_{\mathrm{V}}T$.

For transparently handling of curvilinear coordinates we use a covariant notation with the metric tensor g_{ij}. We only consider orthogonal coordinates in three-space, so the metric tensor is diagonal and we can introduce the metric scale factors $h_i = g_{(i)(i)}^{1/2}$ and the determinant $g = \left|\Pi_i g_{(i)(i)}\right|$. In the following we use the notation $x_{,i}$ for partial derivatives, $x_{;i}$ for covariant derivatives and apply summation over repeated indices not written in brackets ($g_{(i)(i)}$ denotes a tensor component and i is not summed over).

The covariant divergence of a contravariant vector v^i and a symmetric tensor S^{ij} are

$$\begin{aligned} v^i_{;i} &= v^i_{,i} + \begin{Bmatrix} i \\ i\,j \end{Bmatrix} v^i = g^{-1/2}\left(g^{1/2} v^i\right)_{,i} \\ S^{ij}_{;j} &= S^{ij}_{,j} + \begin{Bmatrix} i \\ k\,j \end{Bmatrix} S^{kj} + \begin{Bmatrix} j \\ k\,j \end{Bmatrix} S^{ik} = g^{-1/2}\left(g^{1/2} S^{ij}\right)_{,j} + \begin{Bmatrix} i \\ j\,k \end{Bmatrix} S^{jk}, \end{aligned}$$

where $\left\{ {i \atop j\,k} \right\}$ are the Christoffel symbols of the second kind.

The equations can be written with a conservative part on the left hand sides resembling the Euler equations and extra forces on the right hand side of the equations. Appending the Poisson equation and an ideal gas equation of state, in a covariant form they read:

$$\varrho_{,t} + \left(\varrho u^i\right)_{;i} = 0 \tag{2.1}$$

$$\left(\varrho u_i\right)_{,t} + \left(\varrho u_i u^j\right)_{;j} = -p_{,i} + \sigma^{ij}_{;j} - \varrho\phi_{,i} \tag{2.2}$$

$$\left(\varrho\varepsilon\right)_{,t} + \left[\left(\varrho\varepsilon + p\right) u^i\right]_{;i} = \left(u_i \sigma^{ij}\right)_{;i} \tag{2.3}$$

$$\left(g^{ij}\phi_{,i}\right)_{;j} = 4\pi G\varrho \tag{2.4}$$

$$p = R\frac{1}{\mu}\varrho T\,. \tag{2.5}$$

The Poisson equation (2.4) relates the gravitational potential ϕ of the fluid to the mass density ϱ, the equation of state (2.5) closes the system by relating the gas pressure p to the mass density ϱ and the gas temperature T. In eq. (2.3) one can solve for the total specific

energy $\varepsilon_{\text{tot}} = \varepsilon + \frac{1}{2}u^2$, too. Viscosity is modeled using the viscous stress tensor σ, whose components are

$$\sigma_{ij} = \mu\left(u_{i;j} + u_{j;i}\right) + \left(\zeta - \frac{2}{3}\mu\right) u^k_{;k} g_{ij} = 2\mu E_{ij} + \left(\zeta - \frac{2}{3}\mu\right) u^k_{;k} g_{ij}\,, \tag{2.6}$$

with the coefficients of shear (μ) and bulk viscosity (ζ). The viscous dissipation $\psi = \left(u_i \sigma^{ij}\right)_{;j}$ can also be written as

$$\psi = 2\mu E_{ij} E^{ij} + \left(\zeta - \frac{2}{3}\mu\right) \left(u^i_{;i}\right)^2\,, \tag{2.7}$$

where $E_{ij} = \frac{1}{2}\left(u_{i;j} + u_{j;i}\right)$ is the deformation tensor.

2.2 Operator Splitting

We apply operator splitting to split solving of the equations (2.1)-(2.3) into an advection and a force part. In the advection step we solve the transport part of the equations (that is, equations (2.1)-(2.3) with zero right hand side). In the force step, we honor forces implied by the right hand sides of equations (2.2) and (2.3), namely pressure, gravitational and viscous forces.

Coordinate forces appearing in (2.2) are handled specially. Namely the Coriolis force is accounted for by using angular momentum for u_i (Kley 1998). The centrifugal force is applied explicitly in the force part of the operator split scheme. Respectively, we split eq. (2.2) along the following idea

$$\begin{aligned}
\left(\varrho g^{1/2} u_i\right)_{,t} + \left(\varrho g^{1/2} u_i u^j\right)_{,j} &= 0 \\
u_{i,t} &= -\begin{Bmatrix} i \\ j\,k \end{Bmatrix} \left(u_i u^j\right) - p_{,i} + \sigma^{ij}_{;j} - \phi_{,i}\,,
\end{aligned}$$

while applying simplifications in the actual implementation where possible, as we consider only Cartesian, cylindrical and spherical coordinates. Especially we preserve angular momentum by carefully rewriting these equations in the spherical and cylindrical case (Kley 1998). In the cylindrical case, the equation for $u_\varphi = r\omega = r\dot{\varphi} = ru_2$ is written as

$$\left(\varrho r^2(\omega + \Omega)\right)_{,t} + \left(\varrho r^2(\omega + \Omega)u^j\right)_{,j} = 0\,,$$

while in the spherical case it reads

$$\left(\varrho r^2 \sin^2\theta(\omega + \Omega)\right)_{,t} + \left(\varrho r^2 \sin^2\theta(\omega + \Omega)u^j\right)_{,j} = 0\,,$$

where all contributions of the coordinate forces are accounted for now. The remaining coordinate force terms in the other equations are specified in sec. 2.6.

In addition to splitting the equations into a advection and force part, we apply directional splitting to the advection.

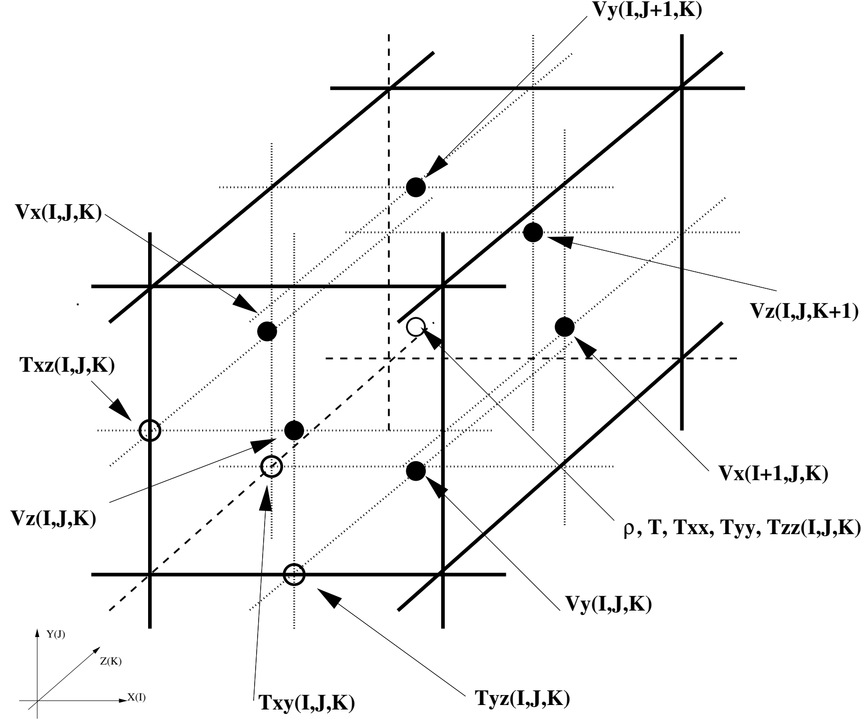


Figure 1: Variables on the staggered mesh.

2.3 Advection Scheme

The advection scheme is directional split, fully discrete and of 2nd-order accuracy in space and time. It is modeled after the second order scheme in van Leer (1977) (see also Hawley et al. 1984) on a staggered mesh using the *mono* limiter introduced there. The staggered mesh (fig. 1) defines scalar quantities cell centered and vector quantities face centered. This has the advantage that differences of vector quantities are centered and such of second order in space.

The equations are discretized in the finite volume description. For instance, eq. (2.1) in the x-direction can be discretized as

$$\varrho^{n+1}_{i+1/2} = \varrho^{n}_{i+1/2} + \Delta t \left(F_i - F_{i+1}\right) ,$$

where upper indices denote the iteration and lower indices grid vertex positions. So F_i and F_{i+1} are the fluxes through the cell $_{i+1/2}$'s faces. With such a discretization scheme, preserving of the advected quantity (here ϱ) is guaranteed.

We use an explicit second-order differencing in time which then leads to the following scheme for computing the fluxes F ($j = i$ for $v_i < 0$, else $j = i-1$) over the cell boundaries:

$$F_i = v_i \Delta t \left(w_{j+1/2} + \left(x_i - x_{j+1/2} - v_i \Delta t/2\right) (\nabla w)^{\rm mono}_{j+1/2} \right) ,$$

where w is the to be advected quantity. Note that for variables that are not cell-centered such as the velocity components the flux cell is shifted to center the component and additional interpolation is necessary.

The mono-limited gradient $(\nabla w)^{\mathrm{mono}}_{i+1/2}$ is expressed as the limited harmonic mean of the up- and down-wind differences $\Delta_i w/\Delta_i x$ and $\Delta_{i+1} w/\Delta_{i+1} x$:

$$(\nabla w)^{\mathrm{mono}}_{i+1/2} = \begin{cases} \dfrac{2\Delta_i w \, \Delta_{i+1} w}{\Delta_i w \Delta_{i+1} x + \Delta_{i+1} w \Delta_i x} & \text{if } \Delta_i w \, \Delta_{i+1} w > 0, \\ 0 & \text{otherwise,} \end{cases}$$

where $\Delta_i w = w_{i+1/2} - w_{i-1/2}$ and $_i$ indexed quantities are on the cell faces and $_{i+1/2}$ indexed quantities on the cell center.

The time-step Δt is limited by the Courant-Friedrichs-Levy (CFL) condition (Courant & Friedrichs 1948; Ritchmyer & Morton 1967) which limits propagation of information to one grid-cell per integration step:

$$\Delta t \le \min_i \frac{\Delta_i x}{|u_i| + C_a} ,$$

where u_i is the ith component of the local fluid velocity, C_a is the local adiabatic speed of sound, $\Delta_i x$ is the grid-spacing in direction i and the minimum is taken over all grid-cells and directions. For multiple dimensions the limit needs to be reduced by multiplying by a constant $0 < c < 1$. For the above advection scheme values of 0.8, 0.7 and 0.5 are appropriate for c in one-, two- and three-dimensional calculations.

2.4 Artificial Viscosity

Artificial viscosity can be applied as part of using a tensor viscosity (see sec. 2.7) or as contributing to the pressure term in the momentum and energy equation. In this case the sum of pressure p and artificial pressure q is computed and substituted for the pressure terms in the momentum and energy equations during calculation of the forces.

In the following we will use $a_{<0}$ as short-hand for $\min(a, 0)$. Then the artificial pressure q is computed as follows:

$$q = c\varrho(\Delta x)^2 \left[(\nabla\cdot u)_{<0}\right]^2$$

where Δx is a characteristic length and c is a constant factor of order unity determining the strength of the artificial viscosity.

For multi-dimensional accretion disks a slightly modified version of the momentum equation (2.2) with a non-isotropic artificial pressure has been found useful:

$$(\varrho u_i)_{,t} + \left(\varrho u_i u^j\right)_{;j} = -\left(p + q_j\right)_{,i} + \sigma^{ij}_{;j} - \phi_{,i} ,$$

where the direction dependent scalar artificial pressure q_i is either

$$q_i = c\varrho \Delta x^2 \left(\nabla\cdot u\right)_{<0} \left(\nabla u_i\right)_{<0} ,$$

or a slightly different version (Klahr, private communication):

$$q_i = c\varrho \left[(\nabla\cdot u)_{<0} \frac{\nabla u_i}{\sum_i (\nabla u_i)_{<0}}\right]^2 .$$

These formulations avoid invalid artificial viscosity in case of a fluid that is frequently compressed in one direction while expanding in another. Both ideas follow Caramana et al. (1998).

2.5 Forces

For the forces, which are essentially the right hand sides of eq. (2.2) and (2.3), we apply a finite difference discretization utilizing centered differences whenever possible. This results in forces computed 2nd-order accurate in space. The time accuracy is of 2nd-order only if the energy equation is solved. In this case, time-centered variables are used to compute the right-hand-sides of the equations as follows:

$$
\begin{aligned}
u_i^{n+1} &= u_i^n + \Delta t \left(-\frac{\nabla p^{n+1/2} + F_{\text{visc}}(u_i^n, \varrho^{n+1})}{\varrho^{n+1}} - \nabla \phi^n + F_{\text{coord}}(u_i^n) \right) \\
T^{n+1} &= T^n - \Delta t \frac{p^{n+1/2}}{\varrho^{n+1}} \nabla \cdot \mathbf{u}^{n+1/2} \,,
\end{aligned}
$$

where viscous and coordinate forces F_{visc} and F_{coord} are computed as described in sec. 2.7 and 2.6. $p^{n+1/2}$ is extrapolated using a simplified form of the energy equation and the equation of state, while $u_i^{n+1/2}$ is computed as the average of u_i^n and u_i^{n+1}.

In case the energy equation is not solved, the scheme is only first-order accurate in time as we substitute p^n for $p^{n+1/2}$.

2.6 Coordinate Forces

Coriolis forces in the φ-direction are handled by the advection scheme and appropriately choosing the primitive velocity variables (sec. 2.2). Centrifugal, remaining Coriolis forces and coordinate system rotation with Ω are accounted for in the forces step (sec. 2.5) and computed specifically for Cartesian (in case of a rotating coordinate system), cylindrical and spherical coordinates. In the cylindrical and spherical cases F_φ and F_z vanish, and for cylindrical coordinates there remains

$$
F_r = r \left(u_\varphi + \Omega \right)^2 \,,
$$

and for spherical coordinates

$$
\begin{aligned}
F_r &= r \left(u_\vartheta^2 + (u_\varphi + \Omega)^2 \sin^2 \vartheta \right) \\
F_\vartheta &= -r \left(u_\varphi + \Omega \right)^2 \cos \vartheta \sin \vartheta \,.
\end{aligned}
$$

For the rotating Cartesian case, there are

$$
\begin{aligned}
F_x &= u_y \Omega + x \Omega^2 \\
F_y &= -u_x \Omega + y \Omega^2 \,.
\end{aligned}
$$

2.7 Viscosity

For accretion disk modeling there is a need of angular momentum transport. This can be either driven by the MRI (Balbus & Hawley 1991) or by explicitly modeling a viscous transport. We model a tensor viscosity with a shear and a bulk part. The bulk or volume part of the viscosity can be used as an artificial viscosity to maintain correct shock solutions

(see also sec. 2.4), the shear part of the viscosity is the physical component that drives e.g. angular momentum transport.

Viscous forces are handled as part of the forces step. First the components of the viscous stress tensor σ_{ij} are calculated from eq. (2.6) in the respective coordinate system, adding artificial bulk viscosity, if requested. The calculation follows the scheme sketched in Mihalas & Mihalas (1984), applying $\zeta = 0$ as simplification. Then the forces resulting from the tensor are computed and applied in the momentum equation. Optionally the viscous dissipation ψ is calculated following eq. (2.7) and applied as a source in the energy equation.

2.8 Gravitation and N-Body

In case of a self-gravitating fluid the Poisson equation (2.4) is solved for the gravitational potential. Being an elliptic boundary value problem this is expensive. For discretization we use central differences, so in the one-dimensional Cartesian case the finite difference equations on an uniformly spaced grid read

$$\frac{\phi_{i-2} - 2\phi_i + \phi_{i+2}}{4\,(\Delta x)^2} = 4\pi G \varrho_i\,.$$

This can be generalized to multiple dimensions and the finite difference equations can then be written as a matrix equation for ϕ and ϱ like the following

$$\mathrm{A}\phi = 4\pi G \varrho\,.$$

With the state vectors ϕ and ϱ and the banded sparse matrix A. This equation can be solved numerically using an iterative solver for linear systems.

For optimal performance on a parallel machine we use the PETSc[1] library which provides a wide variety of MPI parallelized solvers for sparse systems of linear equations. We use the Conjugate Gradients (CG) Krylov subspace method which is suitable for positive semi-definite systems. To improve convergence we apply an incomplete LU decomposition (ILU) as preconditioner in serial computation mode and a block Jacobi preconditioner in the parallel case (these are the defaults chosen by the PETSc library).

To the gravitational potential ϕ of the fluid (in case of a non-self-gravitating fluid this is zero) we add the gravitational potential of all gravitating bodies, such as a central star or a planet:

$$\phi'(\mathbf{r}) = \phi(\mathbf{r}) + \sum_i \frac{G m_i}{|\mathbf{r} - \mathbf{r_i}|}\,.$$

The forces from the gravitational potential are then applied during the force step solving eq. (2.2).

For integrating the n-body paths in time we solve the Newtonian equations of motion for the bodies using a fourth order Runge-Kutta integrator, taking the gravitational potential ϕ of the fluid into account. Appropriate smoothing of ϕ in the vicinity of the bodies is required to keep integration stable. Integration of the bodies is done using the same time-step as determined by the CFL condition for the fluid.

[1] `http://www-unix.mcs.anl.gov/petsc/petsc-2/`

3 Implementation and Performance

Based on the equations, covariant formulation and numerical scheme introduced in chapter 2 we build a parallel three-dimensional hydrodynamics code using the C++ framework provided by the POOMA library (see also appendix A and B). In the following the structure of TraMP is sketched out briefly and the process of compilation and running the code is described. We continue elaborating on how we extended the POOMA library to suit our needs, thereby improving both serial and parallel performance. Last, we extensively evaluate serial and parallel performance achieved.

The source (which is available on the CD in the appendix) consists of the files listed in table 3 with a brief description of the contents. The main program is contained in the `tramp3d.cpp` file, while the advection routines are in `adv5.cpp`, the force routines in `forgas.cpp` and the viscous forces are handled in `visexp.cpp`.

Building of the code is separated into a configuration and a compilation step. First you need to create the configuration script `configure` by invoking the `autoconf` program from the GNU autoconf package. After creating and entering an object-file directory you should call this created `configure` script specifying the following parameters:

--with-pooma=	specify the pooma installation directory
--with-dim=	specify the dimensionality, 1, 2 or 3
--with-coord=	specify the coordinate system, Cartesian, Cylindrical or Spherical

file	contents
tramp3d.cpp	main program including command line processing, I/O routines and main iteration loop
rhalk.h	traits for the POOMA library
delta.h	dX, dY and dZ declarations
cfl2.h	computation of the time-step using a CFL condition
eos.h	different equations of state
adv5.cpp	advection routine
forgas.cpp	force routine
visexp.cpp	explicit viscous forces
checknan.cpp	routines for regularity checks
nbody.cpp	n-body integrator
poisson.h	interface to the poisson solver
poisson.cpp	poisson solver using the PETSc library
hacks.h	special boundary conditions and limiters
test.cpp	initial configuration generator for test problems
star.cpp	initial configuration generator for star problems
disk.cpp	initial configuration generator for disk problems
perturb.cpp	tool to perturb configurations
configure.ac	autoconf configuration file
makefile.in	autoconf makefile template

Table 1: Files of the TraMP code.

This will create a `makefile` in the current directory where you should now invoke the `make` command. The executable is named `tramp`. Parts of the physical and numerical parameters can be specified in the input file (see sec. 3.1.2), others on the `tramp` command-line, some in both ways. General `tramp` command-line parameters include

--help, -h, -?	print available command-line parameters and exit
--input, -i	input file [tramp3d.in]
--output, -o	output file, optional integer (out.%i) [tramp3d.out]
--checkpoint, -c	checkpoint file
--num-iter, -n	number of iterations to perform [Inf]
--end-t, -t	time to stop simulation [Inf]
--output-iter, -no	number of iterations to perform between outputs [Inf]
--output-dt	delta time to do outputs [Inf]
--checkpoint-iter, -nc	number of iterations between checkpointing [Inf]
--checkpoint-dt	CPU time between checkpointing [1h]
--dump-(pg,ph,T)	whether to dump extra variables [no,yes,yes]
--dump-not-(pg,ph,T)	whether to dump extra variables [no,yes,yes]
--dump-guards-(rh,v,T,pg,ph)	whether to dump guards [no,no,no,no,no]
--dump-debug	whether to do debugging dumps [no]
--cfl	cfl number [0.6]
--dt	constant time step, used with cfl number [off]
--min-dt	minimum time step allowed
--max-dt	maximum time step allowed
--blocks nx ny nz	processor setup [automatic]
--smooth rh w	smoothing params [off, try 1e-4 0.25]
--limitv f	v limiting to f*cs [off]
--cartvis nr av	artificial viscosity NR AV [off]
--vis eta	physical viscosity [off]
--selfgrav	self-gravitation [off]
--dump-points x y z ...	dump values ASCII to dump.txt, last param only [off]
--spherical-domain	assume spherical domain [off]
--bh-boundary r	black-hole boundary at r [off]
--eos n	choose eos
--rhomin rho	density floor [off]
--blocking-expressions	set blocking expressions [off]
--parallel-io	use parallel I/O [off]
--hdf5-debug	turn on error/debug messages from hdf5 [off]
--wait-for-gdb	raises SIGSTOP after parsing cmdline [off]

with values in brackets denoting parameter default values.

Input files for the simulations need to be generated using an external program. Three such programs are available, one for standard tests such as Sod and Sedov problems, one for accretion disks and one for stellar oscillation setups. They are in the `test.cpp`, `disk.cpp` and `star.cpp` files. An input file for a one-dimensional Sod problem can e.g. be constructed with

```
> ./test1d sod.in --verts 200 shock +x
```

Please refer to the source and the command-line help (--help) for more information.

3.1 Extending the POOMA Library

The POOMA library as of the 2.4.0 release lacks several features we would like to have for our parallel three-dimensional hydro code. One problem is the need of external libraries for MPI and thread level parallelization, additionally these libraries being hard to compile and install. Another problem is the lack of a proper on-disk format for the data and the lack of parallel I/O support.

Apart from these generic shortcomings of the POOMA library, for our specific implementation we like to have much better support for coordinate systems and covariant formulation of the finite differencing implementation. Also for solving the Poisson equation we need a parallel solver for sparse linear systems which the POOMA library does not provide.

During the task of implementing the hydro code we have also discovered shortcomings in the finite differencing abstraction provided and improved the available mechanisms. Finally, improving the performance of the parallelization has been important, in fact, optimizing the guard updates has improved performance a lot. To summarize, we have extended the POOMA library as follows:

- Array and Field I/O using the HDF5 library, both serial and parallel I/O.
- Interfacing to the PETSc library for the solving of large sparse linear systems from finite difference equations on structured grids.
- Support for non-uniform rectilinear meshes, for cylindrical and spherical coordinate systems and tools for easy covariant implementation of finite difference schemes.
- Much improved finite differencing abstraction (see also appendix B.5).
- Native MPI and OpenMP parallelization without the need of external libraries and with optimized guard updates.

3.1.1 Adding MPI and OpenMP Support

While the POOMA library already provides parallelization with both MPI and threads (and in fact some more parallelization libraries), parallelization is achieved by using a set of helper libraries. Unfortunately these are now unmaintained, outdated and partly unavailable. Also they do not achieve the maximum possible performance.

To overcome this limitations, removing the build dependency on these libraries and avoiding unnecessary indirection has been considered first priority. With the Message Passing Interface (MPI) library and the OpenMP standard there are two widely supported means of parallelization that seemed worthwhile.

The MPI standard (MPI Forum 1995, 1997) defines a set of communication primitives that can be used to explicitly manage communication on a distributed memory computer. Usually MPI is used to do parallelization of grid-based codes using domain decomposition. Hereby guard zones are added to the local grids to hide the fact of distributed memory from the algorithms. These guard zones are then communicated to neighbor nodes at appropriate times using MPI primitives.

In contrast to MPI, OpenMP (OpenMP Architecture Review Board 2002) parallelizes using threads on a shared memory computer. This is done by telling the compiler appropriate places to split the work to be done between multiple processors. A common place to do this is computation loops without data-dependencies between individual loop iterations (that is, the result is the same regardless of the order of evaluation of the loop iterations).

Native MPI parallelization in POOMA replaces the present message passing parallelization via the Cheetah library, which is now unavailable. The appropriate level to do this is the layout patch management code. There is already abstraction for a message passing parallelization provided which we modify for natively using MPI. We especially use the out-of-order iterate scheduler to allow for non-blocking MPI operation to reduce communication latency. We have been able to use data-flow analysis to remove most of the unnecessary guard updates that have been triggered previously. Also with using the native MPI code-path, memory-to-memory copies during communication have been reduced from three to one, which is the best one can do.

While in principle OpenMP parallelization using the domain decomposition and assigning different patches to different threads is possible, not to interfere with MPI parallelization and to allow efficient hybrid MPI and OpenMP parallelization, we choose to strive for loop-level parallelization of the expression kernels resulting from expression template expansion (see appendix A). These are nested loops over the computational domain composed at localized places inside the POOMA library. So only a small number of loop annotations are necessary to deploy OpenMP support. In fact, only reduction operations needed special care, because OpenMP does not understand C++ custom reduction operators, so the final reduction has to be done manually.

3.1.2 HDF5 Parallel and Serial I/O

For the on-disk format of the data we choose the HDF5 (Hierarchical Data Format) file format. The HDF5 library provides structured access to a hierarchy of objects, much like a file system does. The HDF5 library is widely deployed in the scientific and visualization communities. In addition to using the HDF5 file format one needs to define the actual layout of the hierarchy to store the data to.

The POOMA interface to the HDF5 library allows any hierarchy possible (see appendix B.6 for an introduction to this interface). Within the hydro code we use a flat hierarchy meaning that we store all datasets and attributes in the root group of the file. The hierarchy looks as follows (in no particular order):

/			root group
	rh		density dataset
	v		velocity dataset group
		centering0	x-component dataset of the velocity
		centering1	y-component dataset of the velocity
		centering2	z-component dataset of the velocity
	T		temperature dataset
	ph		gravitational potential dataset
	pg		pressure dataset

Attached to the root group there are several important attributes specifying grid and physical setup:

id	type	
dim	int	dimensionality of the datafile
domain	Loc<dim>	size of the computational vertex domain
origin	Vector<dim>	origin of the grid
spacings	Vector<dim>	spacing of the grid
periodicity	Vector<dim, int>	periodicity of the domain (0: not, 1: periodic)
time	double	current time
eos	int	equation of state to use (see eos.h)
nbody	int	number of bodies to integrate
created_with	string	command line this file has been produced with

Other attributes, such as those specifying parameters to the equation of state or the viscosity are listed in the main program (tramp3d.cpp) along with their default values used, if not specified in the file.

Boundary conditions used for the variables are stored in attributes attached to the data-object. Labels for these attributes are constructed by appending the face number to BC and an additional boundary condition specific identifier.

Generally the standard HDF5 tools for inspecting data files, h5ls and h5dump, can be used to extract information out of the files in alphanumeric format. On CD an OpenDX[2] import module and several visualization networks are provided for easy interactive visualization of the data.

3.1.3 Coordinate System Support

Present support for coordinate support in POOMA is limited to associate a mesh and coordinates with a field. While this is not inherently limited to Cartesian coordinates, all derived quantities assume that, so using non-Cartesian coordinates limits applicability of the whole mesh feature.

The task has been to tie coordinate system *type* information to the mesh classes provided by the field abstraction. Due to the explosion of the number of template parameters required for this, we chose to introduce the MeshTraits traits class to encapsulate all the tricky dependencies in the mesh classes hierarchy and to ease further extensions. An introduction on how to use the new mesh abstraction is given in appendix B.4.1.

The MeshTraits class creates a mesh class (see fig. 2 for the mesh inheritance hierarchy) by combining the following components:

MeshTraits ::Mesh_t	MeshTraits ::MeshData_t	MeshTraits ::CoordinateSystem_t
UniformRectilinearMesh	UniformRectilinearMeshData	CartesianURM
RectilinearMesh	RectilinearMeshData	CylindricalURM
		SphericalURM
		CartesianRM
		CylindricalRM
		SphericalRM

[2] http://www.opendx.org/

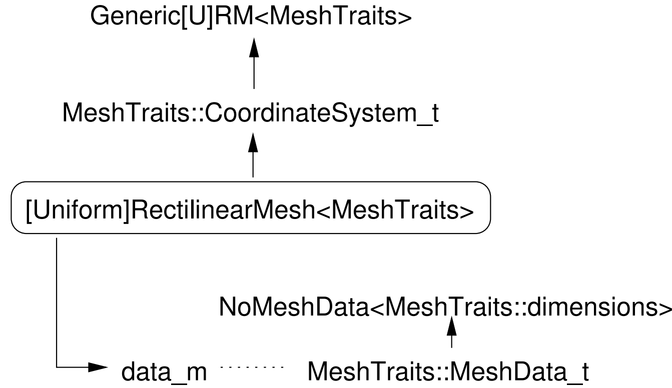


Figure 2: Mesh class inheritance diagram.

The mesh class provides the interface to the user, the data class manages the mesh storage and the coordinate class contains specializations for the geometry factors and coordinate transforms. The abstraction is designed such that the user can provide custom implementations of the mesh, its data container and the coordinate system abstraction by providing the appropriate MeshTraits specialization.

We introduce geometry factors to allow covariant formulation (in an orthogonal metric) of the discretized equations following the approach in Stone & Norman (1992) and (Klahr 1998). All symbolic factors come in two variations, namely positioned on the `A` and `B` grid. For instance there is both `VOLXA` and `VOLXB`, `VOLX` on the `A` and the `B` grid. In the following we omit the distinction between those two and just use `VOLX` referring to both of them.

First we introduce the decomposed metric scale-factors $h^i = \sqrt{g^{ii}}$ for doing covariant derivatives and for computing the line-element $\mathrm{d}s$. We assume $h^0 = 1$, $h^1 = \texttt{GEOXG}(x_0)$ and $h^2 = \texttt{GEOXH}(x_0)\,\texttt{GEOYH}(x_1)$ and no further dependencies. So $\mathrm{d}s = \mathrm{d}x_0\mathbf{\hat{x}_0} + \texttt{GEOXG}\,\mathrm{d}x_1\mathbf{\hat{x}_1} + \texttt{GEOXH}\,\texttt{GEOYH}\,\mathrm{d}x_2\mathbf{\hat{x}_2}$.

The general volume element $\mathrm{d}V = \mathrm{d}x_0\mathrm{d}x_1\mathrm{d}x_2$ is introduced via the symbolic names `VOLX`, `VOLY` and `VOLZ`. For Cartesian coordinates these are equal to the respective grid-spacings, in the cylindrical case the volume element is $\mathrm{d}V = r\mathrm{d}r\mathrm{d}\theta\mathrm{d}\varphi = \mathrm{d}\frac{r^2}{2}\mathrm{d}\theta\mathrm{d}\varphi$, for spherical coordinates the volume element is $\mathrm{d}V = \sin\theta\mathrm{d}\theta r^2\mathrm{d}r\mathrm{d}\varphi = \mathrm{d}\left(-\cos\theta\right)\mathrm{d}\frac{r^3}{3}\mathrm{d}\varphi$.

Similarly we introduce the area element as the derivative of the volume element so that for the area element of the r-θ plane we can use $\mathrm{d}A_{r\theta} = \mathrm{d}\left(-\cos\theta\right)\mathrm{d}\frac{r^3}{3}\partial_\varphi\mathrm{d}\varphi$ (or `VOLY VOLX SURZ`).

In table 2 the geometry factors are listed along with their values in the respective coordinate systems. The geometry factors are available in the respective mesh class specialization.

3.1.4 Interfacing to PETSc

We provide a tool to interface with the PETSc library for solving of large sparse linear systems. This tool is the `PoomaDA` class which translates between the parallel decompositions of the POOMA and the PETSc library.

Adapting the parallel decomposition of the POOMA library to the one of the PETSc

	Cartesian	Cylindrical	Spherical
GEOXG	1	1	r
GEOXH	1	r	r
GEOYH	1	1	$\sin\theta$
VOLX	Δx	$1/2\Delta r^2$	$1/3\Delta r^3$
VOLY	Δy	Δz	$-\Delta\cos\theta$
VOLZ	Δz	$\Delta\varphi$	$\Delta\phi$
SURX	1	r	r^2
SURY	1	1	$\sin\theta$
SURZ	1	1	1

Table 2: Geometry factors for Cartesian, cylindrical and spherical coordinates.

library is done by using the PETSc DA (Distributed Array) feature, which is initialized by instantiating a `PoomaDA` object (see reference documentation of the `Transform/PETSc.h` file). The `PoomaDA` object can be used in place of a PETSc DA object, initialization of the required PETSc vectors and matrices proceeds as with natively using PETSc. Extra functionality is provided only for assigning POOMA arrays and fields to and from PETSc `Vec`s via the overloaded `PoomaDA::assign` method. Solving of a linear system can be as easy as (PETSc setup stripped for brevity):

```
Array rh(layout), ph(layout);
PoomaDA da(layout);
KSP petsc_ksp;
Vec petsc_rh, petsc_ph;
...
KSPSetRhs(petsc_ksp, petsc_rh);
KSPSetSolution(petsc_ksp, petsc_ph);
...
da.assign(petsc_rh, rh);
KSPSolve(petsc_ksp);
da.assign(ph, petsc_ph);
```

Please refer to the PETSc documentation for further information and possible advanced uses.

3.2 Serial Performance

In appendix A.5 we discuss problems of todays compilers regarding to optimizing the expression template expander loops. The main problem is the lack of complete inlining and in turn the break-down of most loop optimizers.

With the availability of the source code of one of the most popular compilers, namely the C++ compiler of the GNU[3] compiler collection (GCC), we have been able to solve the inlining problem ourselves instead of relying on compiler vendors. That is, we can optimize the compiler for our use of C++ instead of adapting the use of C++ for the compiler (we do the latter as well, if possible).

[3] GNU is Not Unix

The solution to the inlining problem is to hint the compiler at completely inlining all calls inside the expression template expander loops. With the new tree-ssa technology emerging in recent GCC versions (4.0.0) this can be accomplished by adding a new function attribute (for GCC 3.3 and 3.4 this is possible, too, but more complicated and not without side-effects to inlining at unrelated call-sites).

Our new attribute has been named `leafify` as it turns the marked function into a leaf function of the call graph of the program. To avoid running in circles if having recursive calls inside a to be leafified function we need to find closed loops inside the call graph. After doing this we recursively mark any call inside the function body as to be inlined, stopping only at entry points to detected loops inside the call graph. For example, the expression expander is marked for leafification in the following way[4]:

```
template<>
struct KernelEvaluator<InlineKernelTag>
{
  /// Input an expression and cause it to be evaluated.
  /// All this template function does is extract the domain
  /// from the expression and call evaluate on that.

  template<class LHS, class Op, class RHS>
  static __attribute__((leafify))
  void evaluate(const LHS& lhs, const Op& op, const RHS& rhs)
  {
    typedef typename LHS::Domain_t Domain_t;
    evaluate(lhs, op, rhs, lhs.domain(),
             WrappedInt<Domain_t::dimensions>());
  }
...
  template<class LHS, class Op, class RHS, class Domain>
  inline static void evaluate(const LHS& lhs, const Op& op, const RHS& rhs,
                              const Domain& domain, WrappedInt<2>)
  {
    int e0 = domain[0].length();
    int e1 = domain[1].length();
    for (int i1=0; i1<e1; ++i1)
      for (int i0=0; i0<e0; ++i0)
        op(lhs(i0,i1), rhs.read(i0,i1));
  }
...
};
```

Here we show only the expander for the two-dimensional case. Note that only the dispatcher needs to be marked with the leafify attribute.

We compare the performance of a simulation of a three-dimensional Sedov explosion, first with the leafify function attribute turned on, and second with it turned off, using different compiler versions. The simulation is run for 50 iterations and the total runtime is measured. Base options used for compilations are `-O2 -funroll-loops -ffast-math -march=pentium4 -D__NO_MATH_INLINES`. All measurements have been performed on an Intel Pentium-4 machine. GCC 4.0.0 is not yet released, so a snapshot from CVS as of

[4]Code excerpt from `src/Evaluator/InlineEvaluator.h`, edited for brevity

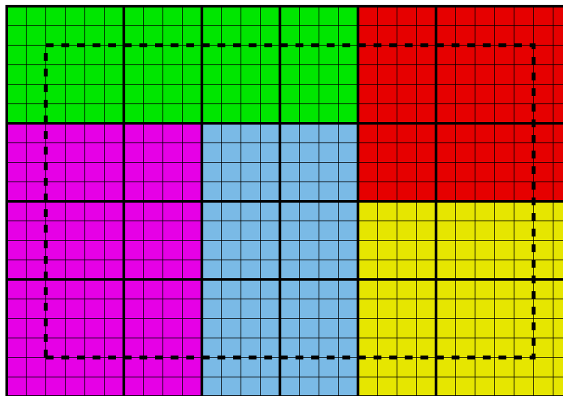


Figure 3: Domain decomposition. Straight thick lines mark the decomposition of the thin lined grid. Colors mark the sub-domains assigned to one processor. The dashed line denotes the computational boundary beyond which the external guard cells are located.

Nov 21th, 2004 has been used. As reference, numbers for the Intel compiler as of version 8.1 are provided (though without leafify, as that is unsupported). Optimization options for the Intel compiler were `-O2 -xW` (adding `-ip` for inter-procedural optimization causes the compiler to crash). The CFD program used for performance testing is also included on the supplemental CD.

Compiler	compile time	no leafify(1)	leafify(2)	ratio (2)/(1)
GCC 3.3.5	3m29s	4m6s	2m48s	0.68
GCC 3.4.2	1m36s	3m7s	1m28s	0.47
GCC 4.0.0	2m12s	9m16s	1m22s	0.15
Intel 8.1	1m43s	2m6s	-	-

As one can clearly see, using the `leafify` attribute to hint the compiler with its inlining improves code by up to 50% (ignore the baseline values of the 4.0.0 development snapshot, it is likely to improve to that of 3.4.2). It is also obvious that compilers are improving in the area of optimizing these kind of codes.

For reference the patch adding the leafify function attribute to the current mainline CVS of the GCC is reproduced in Appendix C. On the supplemental CD you can also find patches for the 3.3 and 3.4 series.

3.3 Parallelization and Performance

TraMP is parallelized both for a shared memory computer using OpenMP directives and for a distributed memory computer using the MPI library. Parallelization is on the level of the POOMA library (see appendix B for an introduction) and does not influence the implementation of the finite difference equations directly. This greatly simplifies the implementation of a finite difference scheme as one need not care about details of the parallelization.

Implementation of the parallelization is done on two levels. On the low level, the finite differencing loops are parallelized using OpenMP directives. On the high level, a domain

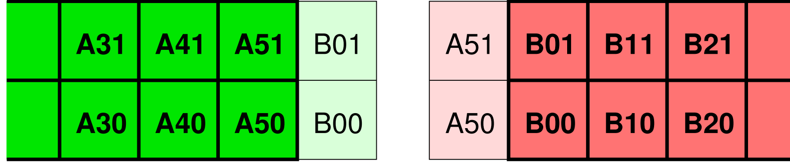


Figure 4: Guard zones (light area) are used to hide the fact of decomposition from the patches. They contain data from adjacent patches.

decomposition (fig. 3) into patches is performed and an approximately equal number of patches is assigned to each processor. Beyond the border of these patches guard regions are introduced that contain data from adjacent patches (fig. 4). Just like for boundary conditions that are implemented using external guard regions, this allows transparent treatment of the numerical coupling between patches. The size of these guard regions is dependent on the numerical scheme and its so called *stencil width*. In the case of TraMP we need two extra guard cells beyond the computational domain. Only at certain time during evaluating the finite difference equations these guards need to be updated using communication between neighboring patches.

Parallelization is done for two reasons. First, to reduce the runtime a simulation takes, second to allow bigger simulations than can be done on a single computer. We can expect the runtime to decrease as adding more computer nodes to the simulation, but not below a certain level, where the parallelization overhead outweighs the extra computational power. As opposed to this, the ability to scale to bigger simulations does not have a limit other than the available total memory and I/O bandwidth.

With an ideal parallel computer and implementation we expect the runtime of a simulation to decrease with T/N with N being the number of computing nodes and T being the serial runtime. We also expect a linear increase of the runtime with the size of the simulation. While the latter is usually matching real life experience, the former does not hold because of parallelization and communication overhead and due to Amdahl's law. This law from 1967 tells, that there is always a part of the program that is executed serially, and that the runtime of the parallel program cannot decrease below the runtime of the serial part of the program. Specifically, the speedup for N processors is limited to

$$\frac{1}{F + (1-F)/N} \stackrel{N\to\infty}{\longrightarrow} \frac{1}{F},$$

where F is the fraction of the calculation that is sequential. Suppose that F is 10%, the parallel program can be sped up by a maximum of a factor of 10 only.

For parallel testing we have two clusters of workstations available at the department. The *Kepler cluster* is composed of 96 nodes, each with two Pentium-III 650 MHz processors and 1 GB of memory, and 32 nodes, each with two Athlon-MP 1.6 GHz processors and 2 GB of memory. The nodes are connected with a fully switched Myrinet interconnect with a bandwidth of 2 Gbit/s (full-duplex) and a latency of about 7 μs.

The *Phoenix cluster* is composed of 16 nodes, each with two Athlon-MP 1.4 GHz processors and 2 GB of memory. The nodes are connected with a fully switched 100 Mbit Ethernet interconnect with a bandwidth of 0.1 Gbit/s (full-duplex) and a latency of about

Node setup	Domain decomp.	64^3 Athlon	64^3 Pentium3	128^3 Athlon	128^3 Pentium3	256^3 Athlon	256^3 Pentium3
1x1	1x1x1	86.1s	279.4s	667.1s	(2235s)-	(5337s)-	(17882s)-
1x2	2x1x1	57.5s	188.7s	513.0s	-	-	-
2x2	2x2x1	29.2s	91.2s	203.4s	-	-	-
4x2	2x2x2	16.6s	52.9s	108.6s	381.8s	822.0s	-
6x2	3x2x2	12.7s	38.0s	81.0s	263.9s		-
8x2	4x2x2	10.7s	31.9s	68.4s	225.8s	462.5s	(*)1659s
10x2	5x2x2	10.2s	29.5s	54.2s	172.2s		
12x2	4x3x2	8.9s	25.9s	46.6s	148.9s		
14x2	7x2x2	9.6s	26.2s	47.4s	135.9s		
16x2	4x4x2	9.3s	23.2s	37.5s	121.6s	261.1s	897s
20x2	5x4x2	8.2s		34.8s		215.4s	
24x2	4x4x3	7.4s	20.5s	30.8s	95.7s	183.3s	623s
32x2	4x4x4		22.9s		79.6s	145.9s	493s
48x2	6x4x4				66.7s		362s
64x2	8x4x4				63.2s		296s

Table 3: Three-dimensional Sedov explosion. Wall clock time for 50 iterations. Kepler cluster, Myrinet. Times marked with (*) are extrapolated from less iterations. Cells containing "-" denote configurations with insufficient memory, the time in parentheses is extrapolated from lower resolution.

80 μs.

For comparison, the theoretical bandwidth of dual-channel 266 MHz 64-bit DDR memory as used for the Athlon-MP parts of the above clusters is 17 Gbit/s, the latency is in the order of 0.2 μs.

3.3.1 MPI Parallel Performance

For evaluating MPI parallelization performance we have performed Sedov explosion (see sec. 4.1) simulations in two and three spatial dimensions with different grid-sizes on the two different PC-clusters. In the tables 3, 4 and 5 the raw data resulting from the simulations are displayed. The node setup is specified using two numbers, the numbers of computation nodes (each with two processors) used – Nx – and the actual number of processors participating on each node – x1 or x2. The domain decomposition is in blocks, i.e. it specifies the block subdivision, AxBxC specifies a grid of A times B times C blocks with an approximately equal number of grid cells.

The performance can be best evaluated by looking at double logarithmic plots of the runtime over the number of processors participating in the computation. We expect large simulations to run into the parallelization limit later than small simulations. This can be seen in figures 5-9. Here the run-times of the different sized simulations have been normalized to that of the smallest simulation to allow comparison. From the figures we can see that on the Kepler cluster simulations scale well up to at least eight to sixteen times the number of processors the simulation can be run on minimally. Even higher

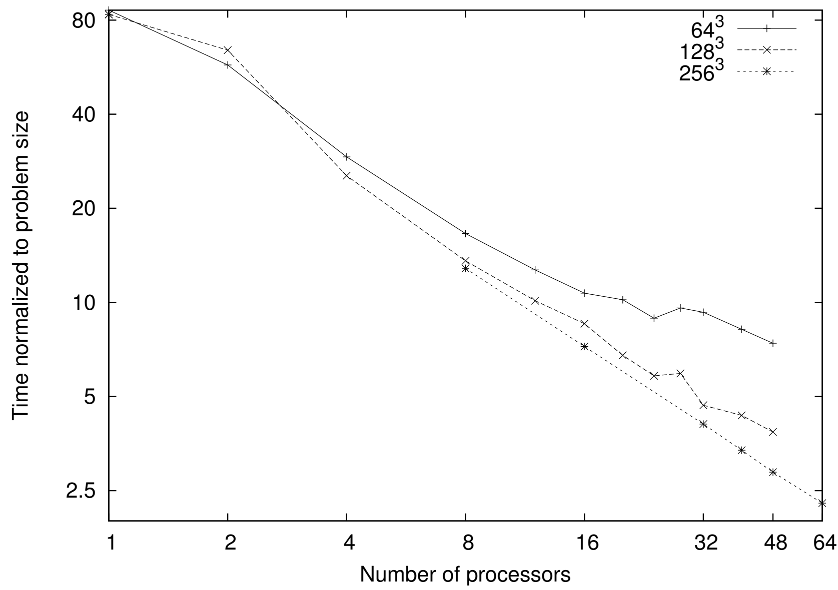


Figure 5: Three-dimensional Sedov explosion. Timings for 50 iterations over different numbers of processors. Kepler cluster, Athlon processors.

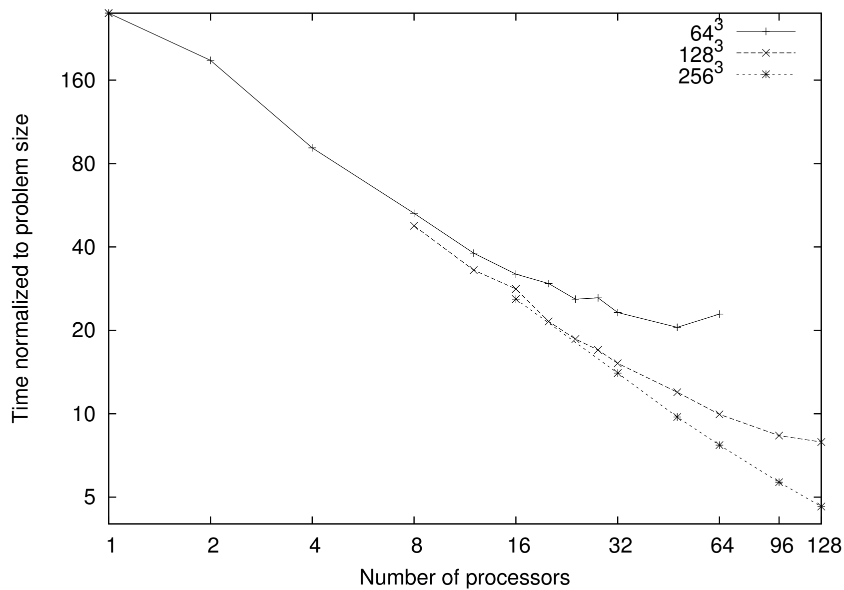


Figure 6: Three-dimensional Sedov explosion. Timings for 50 iterations over different numbers of processors. Kepler cluster, Pentium-III processors.

Node setup	Domain decomp.	64^3	128^3	256^3
1x1	1x1x1	99.4s	773.3s	(6186s) -
1x2	2x1x1	62.3s	485.3s	-
2x1	2x2x1	69.2s	522.5s	-
2x2	2x2x1	46.9s	318.0s	-
4x1	2x2x2	44.4s	307.6s	$^{(*)}2228s$
4x2	2x2x2	32.2s	185.7s	1265s
8x1	2x2x2	29.4s	175.7s	1234s
6x2	3x2x2	29.3s	142.9s	
8x2	4x2x2	27.1s	116.3s	731s
16x1	4x2x2	26.4s	110.1s	719s
10x2	5x2x2	29.0s	110.6s	
12x2	4x3x2	29.0s	99.5s	
14x2	7x2x2	31.2s	98.7s	
16x2	4x4x2	29.2s	75.4s	422s

Table 4: Three-dimensional Sedov explosion. Wall clock time for 50 iterations. Phoenix cluster, 100 Mbit ethernet. Times marked with $^{(*)}$ are extrapolated from less iterations. Cells containing "-" denote configurations with insufficient memory, the time in parentheses is extrapolated from lower resolution.

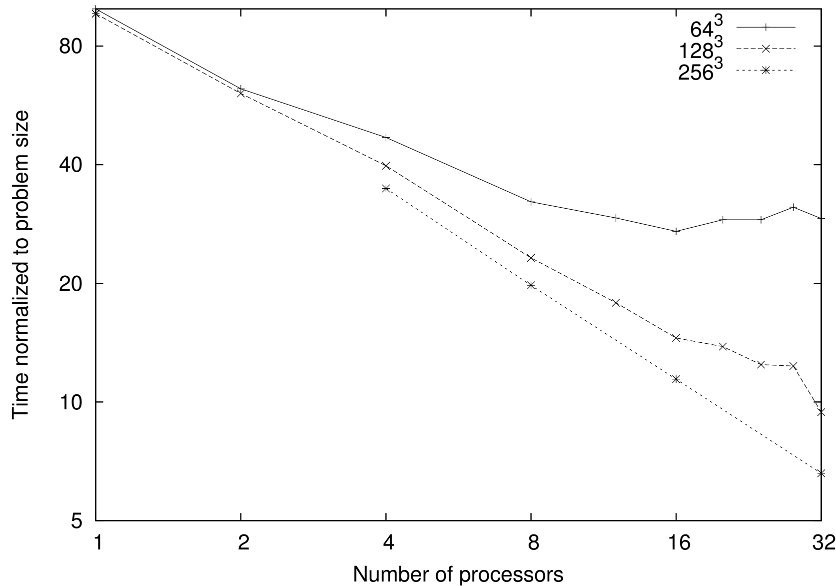


Figure 7: Three-dimensional Sedov explosion. Timings for 50 iterations over different numbers of processors. Phoenix cluster with Athlon processors.

Node setup	Domain decomp.	512^2 Athlon	512^2 Pentium3	4096^2 Athlon	4096^2 Pentium3
1x1	1x1	78.6s	266.4s	(5030s)-	(17050s)-
1x2	2x1	41.9s	154.9s	-	-
2x2	2x2	21.6s	79.0s	-	-
4x1	2x2			1584s	-
4x2	4x2	11.9s	44.5s	912s	-
8x1	4x4				3071s
6x2	4x3	8.7s			
8x2	4x4	7.1s	24.7s	489s	1768s
10x2	5x4	6.3s			
12x2	6x4	5.8s			
14x2	7x4	5.5s			
16x2	8x4	5.4s	16.9s	247s	877s
18x2	6x6	5.2s			
20x2	8x5	5.1s			
24x2	8x6	5.1s	15.4s	165s	582s
32x2	8x8		18.1s	96s	355s
56x2	14x8				219s
64x2	16x8				198s

Table 5: Two-dimensional Sedov explosion. Wall clock time for 100 iterations. Kepler cluster, Myrinet. Cells containing “-” denote configurations with insufficient memory, the time in parentheses is extrapolated from lower resolution.

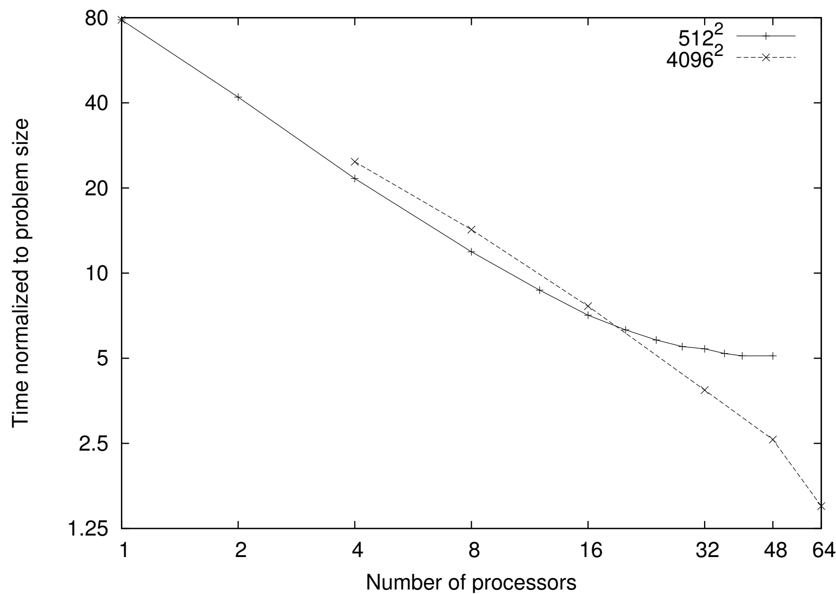


Figure 8: Two-dimensional Sedov explosion. Timings for 100 iterations over different numbers of processors. Kepler cluster, Athlon processors.

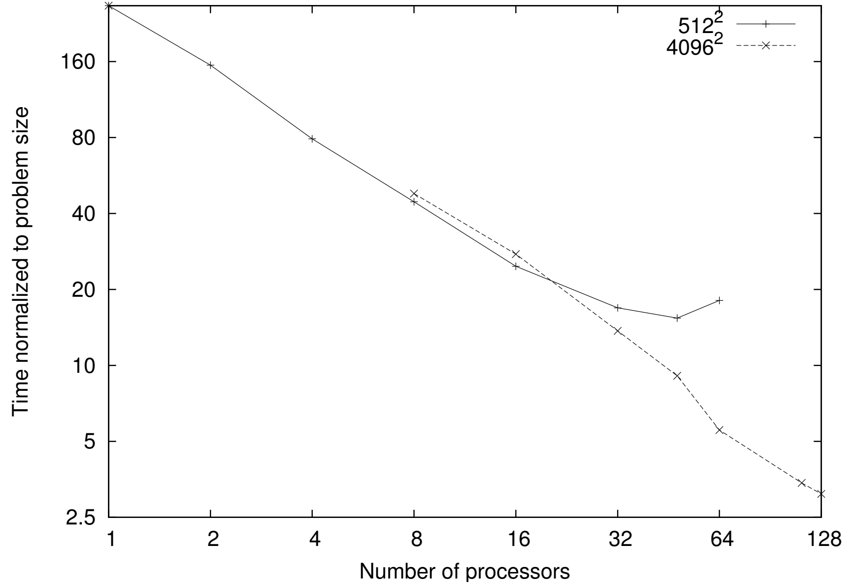


Figure 9: Two-dimensional Sedov explosion. Timings for 100 iterations over different numbers of processors. Kepler cluster, Pentium-III processors.

speedups are suggested by the larger simulations, but we were not able to do testing on more nodes. The numbers do not suggest differences between two- and three-dimensional simulations, so we may conclude that bandwidth is not the limiting part. The simulations on the Phoenix cluster confirm this, as does the comparison of the timings between different sized simulations. Also looking at the 64^3 simulation on the Pheonix cluster indeed suggests latency limited performance, as this effect should be exaggerated in small-sized simulations.

3.3.2 OpenMP Parallel Performance

For evaluating OpenMP parallelization performance we have been able to perform some measurements on a four-way Itanium-2 server using the OpenMP capable Intel 8.0 compiler. Direct comparisons using GCC cannot be done because the GNU compiler does not provide OpenMP support. Instead we compare with GCC using MPI parallelization to compare both technologies. Measuring parallelization efficiency using this setup is not very sophisticated, because four processors are not many but it serves as a starting point. Also four processors are a typical value for processors per node for a hybrid setup.

As test problem we use an isothermal accretion disk (see sec. 4.2) on a grid of 64x20x128 cells. In table 6 the timing results comparing the Intel compiler with OpenMP and the GCC compiler in two different versions with MPI are shown. CPU time is the aggregate time the CPUs have been working, wall time is the time the simulation takes to finish in real time. The speedup is the latter divided by the former.

Apart from measurement noise this is an ideal scale-up to up to four processors for the OpenMP parallelization. This suggests that the memory bandwidth is not the limiting factor for the simulation and hence, the quality of the compiled code is not good, which is

#proc	Compiler	CPU time	Wall time	Speedup
1	Intel, OpenMP	321.04s	321s	1.0
2		321.96s	161s	1.99
3		321.38s	107s	3.0
4		320.93s	80s	4.01
1	GCC 3.4.2, MPI	397.32s	397s	1.0
2			241s	1.65
4			137s	2.89
1	GCC 4.0.0, MPI	201.08s	201s	1.0
2			128s	1.57
4			71s	2.83

Table 6: Comparison of OpenMP (Intel compiler) and MPI (GCC) performance on a four processor Itanium machine.

Nodes	Processors	MPI	MPI & OpenMP	Speedup
1	2	33.7s	22.0s	1.53
2	4	21.6s	22.2s	0.97
4	8	13.7s	13.9s	0.99
8	16	9.9s	9.2s	1.08
12	24	8.4s	7.7s	1.09
16	32	7.1s	6.9s	1.03

Table 7: Hybrid parallelization performance compared to pure MPI on the Phoenix cluster which is composed of dual processor machines. These measurements were exclusively performed with the Intel compiler. Pure MPI timings were produced on a Nx2 setup, MPI & OpenMP timings on a Nx1 setup with two OpenMP threads running per node.

also hinted at by the fact that the GCC compiler is able to outperform the Intel compiler on the Itanium architecture.

Note that while the MPI scaling is not as good as the OpenMP scaling, using a GCC 4.0.0 prerelease with leafification enabled (see sec. 3.2) instead of the Intel compiler results in a speedup of 1.6. Also using MPI with GCC on four processors is faster than using the Intel compiler with OpenMP as can be seen in the results in table 6. Extrapolating this figure to a machine with more processors suggests that using OpenMP parallelization with the Intel compiler pays off starting from eight processors.

3.3.3 MPI and OpenMP Hybrid Parallel Performance

The OpenMP numbers hint at the possibility of a good hybrid parallelization performance. Exercising hybrid parallelization using OpenMP and MPI on the Phoenix cluster, which is a cluster of two-way Athlon-MP computing nodes, yields the results in table 7. These are unclear results, as only the timings on one node suggest a measurable speedup for using OpenMP instead of MPI. The other differences are in the noise and some times in favor of a hybrid and sometimes of a pure MPI setup.

Until tests are possible on e.g. a SX-6 parallel supercomputer, which is composed of eight-way shared memory nodes, with a compiler that can create fast code for serial operation like GCC with the leafify patch, hybrid parallelization performance can not be judged on.

4 Tests

To test the implementation of the numerical scheme, a number of test-calculations have been performed successfully. The standard Sod problem has been used to test correct integration of the Euler equations in all three spatial directions. The standard Sedov explosion has been used to check retaining spherical symmetry in two and three-dimensional Cartesian coordinates. Similar, radial stellar oscillations have been used to check correct operation of gravitational forces, including solving the Poisson equation (see sec. 5). A three-dimensional non-viscous isothermal accretion disk has been set up in cylindrical and spherical coordinates to test integration in cylindrical and spherical coordinates and the correct operation of the centrifugal and Coriolis forces. We also did tests embedding a protoplanet into the accretion disk.

4.1 Shock Problems

We tested the numerical scheme and its implementation on the standard Sod problem (Sod 1978), which is a time-evolution of a discontinuity in density and temperature. The test has been performed in one, two and three dimensions, in all directions, with and without artificial viscosity. The initial conditions are $v = 0.0$, $\varrho_{\mathrm{left}} = 1.0$, $\varrho_{\mathrm{right}} = 0.125$, $T_{\mathrm{left}} = 1.0/\varrho_{\mathrm{left}} \cdot 2.3$ and $T_{\mathrm{right}} = 0.1/\varrho_{\mathrm{right}} \cdot 2.3$. The Sod problem solution consists of a left rarefraction, a contact and a right shock.

For reference, in fig. 10 you can see the results for low resolution calculations with solving for the internal energy, with shocks going in positive and negative x-direction with and without artificial viscosity. In fig. 11 you can see high resolution calculations with solving for the total energy for shocks going in positive x-direction with and without artificial viscosity. Tests in the remaining directions show numerically identic results. One can clearly see the effect of artificial viscosity, which smears out the right shock, reduces overshoots at the discontinuities and helps maintaining the correct shock temperatures in the case of solving for the internal energy.

We used the Sedov explosion (Sedov 1977) to test both symmetry and periodic and reflecting boundaries in multiple dimensions on a Cartesian grid. The Sedov problem evolves a spherical deposition of energy on an ambient media in time. Initial conditions for the Sedov explosion are as follows (where placing of the explosion center $\mathbf{r_0}$ is variable): $v = 0$, $\varrho = 0.1$ and $T(\mathbf{r}) = 1.0 + 99.0/e^{1000 \cdot |\mathbf{r}-\mathbf{r_0}|^2}$.

In fig. 12 one can see different stages of a two-dimensional simulation of an off-center explosion with periodic boundaries. A simulation of an off-center explosion with reflecting boundaries can be seen in fig. 13. Visually one can confirm conservation of spherical symmetry as well as correct reflection and periodic behavior. Exact preserving of symmetry was confirmed for one-dimensional simulations. Also conservation of total mass is exact, as expected from the numerical scheme.

4.2 Accretion Disks

Full three-dimensional and two-dimensional axisymmetric configurations of an isothermal accretion disk have been used to test covariance and as such centrifugal and Coriolis force implementations for cylindrical and spherical coordinates in a stationary and a rotating frame. Following D'Angelo et al. (2003) we use an isothermal equation of state $p = \varrho c_{\mathrm{s}}^2$,

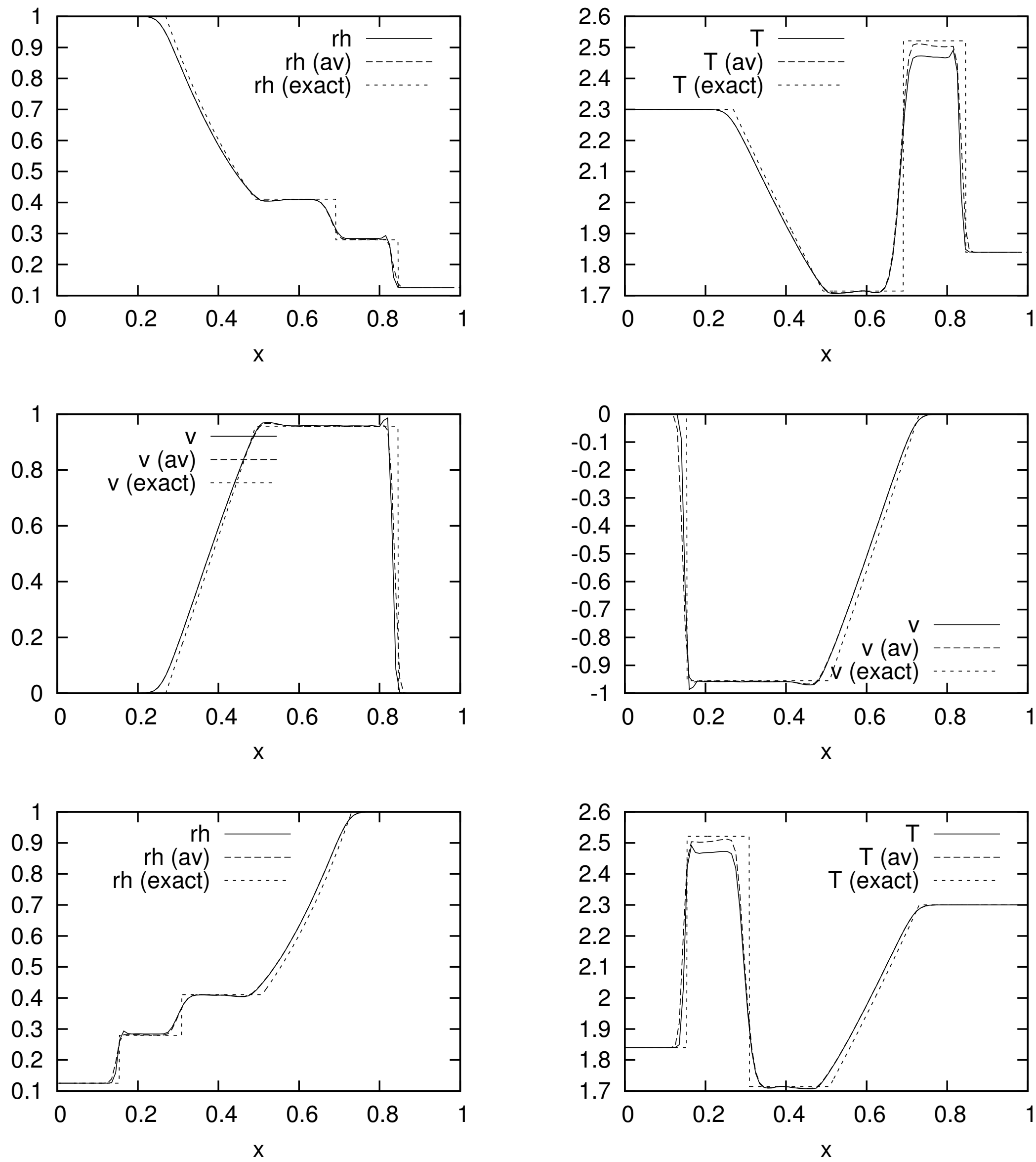


Figure 10: Sod problem in positive and negative x-direction solving for the internal energy, with (av) and without artificial viscosity. 100 grid-points, after 0.2s of integration. The short dashed line is the solution obtained with an exact Riemann solver.

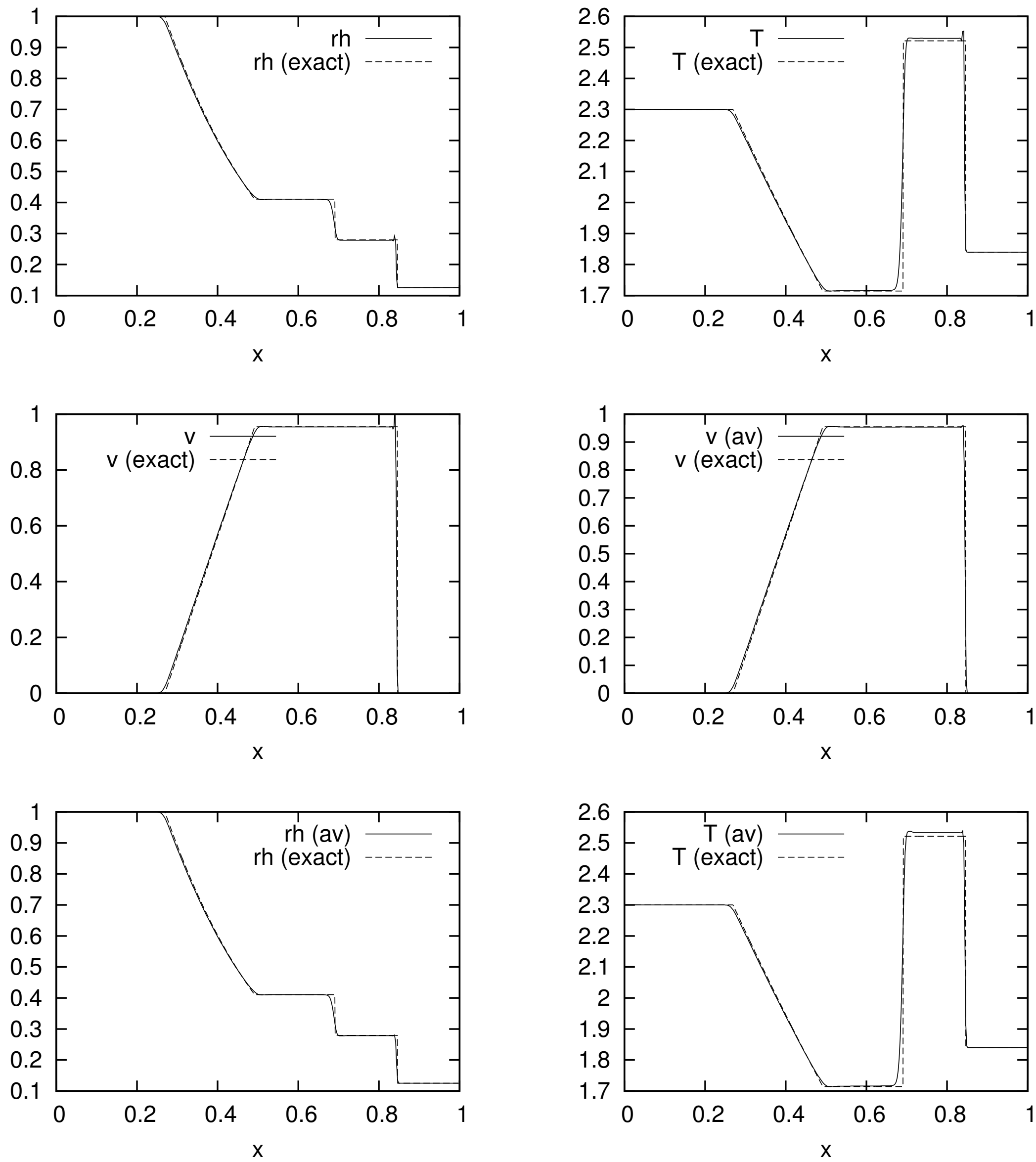


Figure 11: Sod problem in positive x-direction solving for the total energy, with (av) and without artificial viscosity. 400 grid-points, after 0.2s of integration. The dashed line is the solution obtained with an exact Riemann solver.

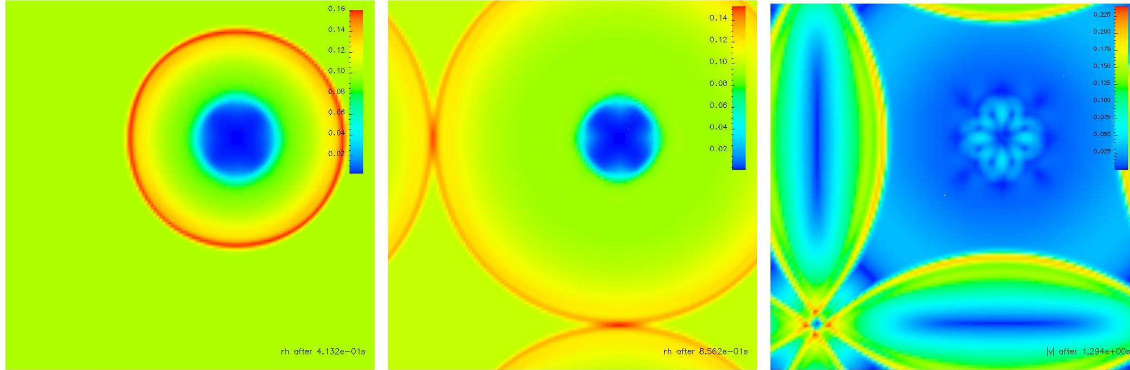


Figure 12: Off-center two-dimensional Sedov explosion in Cartesian coordinates with periodic boundary conditions on a 128 squared grid. From left to right: density after 200, density after 400 and velocity magnitude after 600 iterations.

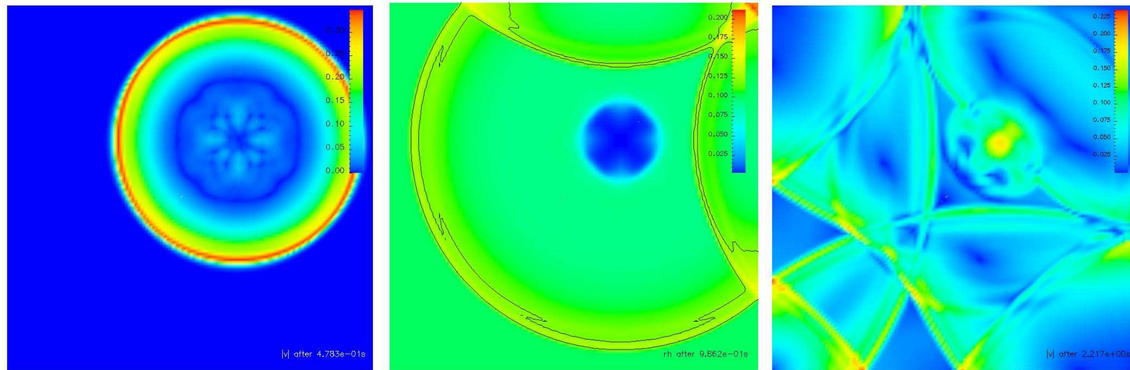


Figure 13: Off-center two-dimensional Sedov explosion in Cartesian coordinates with reflecting boundaries on a 128 squared grid. From left to right: velocity magnitude after 200 (similar to fig. 12), density (isosurface $\varrho = 0.1205$) after 400 and velocity magnitude after 900 iterations.

where the sound speed c_s equals the Keplerian velocity v_K times the disk aspect ratio $h = H/(r\sin\vartheta)$ which we choose to 0.05. Here H is the pressure scale height of the disk. As we assume h is constant, the disk is azimuthally and vertically isothermal. The initial density distribution is chosen as

$$\varrho_0 = (r\sin\vartheta)^{-3/2}\, e^{-\left(\frac{\cot\vartheta}{h}\right)^2} .$$

Boundaries are set up as reflecting.

Note that there is no stationary solution to the Euler equations in axisymmetry (spherical or cylindrical coordinates) for an isothermal or polytropic vertically extended disk. In fact, the momentum equations for u_r and u_θ can be used to derive a system of equations for u_φ and ϱ of the stationary solution (axisymmetric $\partial_\varphi = 0$, stationary $\partial_t = 0$, $v_r = v_\theta = 0$):

$$\begin{aligned} -\frac{\varrho}{r}u_\varphi^2 &= -\frac{\varrho}{r^2} - \frac{\partial p}{\partial r} \\ -\frac{\varrho}{r}u_\varphi^2 \cot\theta &= -\frac{1}{r}\frac{\partial p}{\partial \theta} . \end{aligned}$$

In the polytropic case $p = c\varrho$ (c =const.) this reduces to the following equation for $\varrho(r,\theta)$:

$$\varrho\left(\frac{1}{r} + c\frac{r}{\varrho}\frac{\partial\varrho}{\partial r}\right)\cot\theta = c\frac{\partial\varrho}{\partial\theta} ,$$

which has only the non-disk like solutions

$$\varrho(r,\vartheta) = F\left(\frac{1}{r\sin\vartheta}\right)e^{\frac{1}{cr}} ,$$

with F being any function that can be differentiated appropriately. For the isothermal case with $p = \varrho c_\mathrm{s}^2$, $c_\mathrm{s} = c/\sqrt{r}$ (c =const.), the equation for $\varrho(r,\theta)$ is

$$\varrho\left(\sqrt{r} - \frac{1}{2}c + c\frac{r}{\varrho}\frac{\partial\varrho}{\partial r}\right)\cot\theta = c\frac{\partial\varrho}{\partial\theta} ,$$

which results in solutions of the form

$$\varrho(r,\vartheta) = F\left(\frac{1}{r\sin\vartheta}\right)\sqrt{r}e^{-2\frac{\sqrt{r}}{c}} .$$

So one cannot expect our initial condition to be stationary, nor settle to a solution with $u_\theta = u_r = 0$, but rather to some quasi-stationary state.

In fig. 14 you can see a vertical slice of the density distribution after 80 orbital periods of the outer disk edge. This quasi-stationary configuration exhibits the radial and vertical structures visible in figs. 15 and 16.

4.3 Protoplanetary Disks

A non-viscous isothermal accretion disk with an embedded protoplanet has been evolved for an extended time to look for familiar features. The calculation has taken place on a spherical grid with a resolution of 128 cells in r, 24 cells in θ and 383 cells in φ. The computational domain radially extends from two to nine AU, the vertical opening angle of

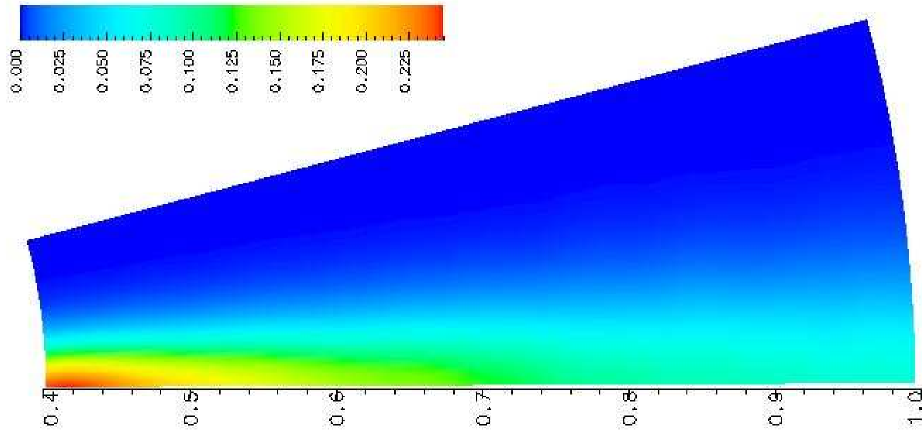


Figure 14: Isothermal accretion disk density distribution after 80 orbital periods of the outer disk edge. Two-dimensional axisymmetric calculation on a 204x28 ($r \,\mathrm{x}\, \theta$) grid with mirror-symmetry in the $\theta = \pi/2$ plane.

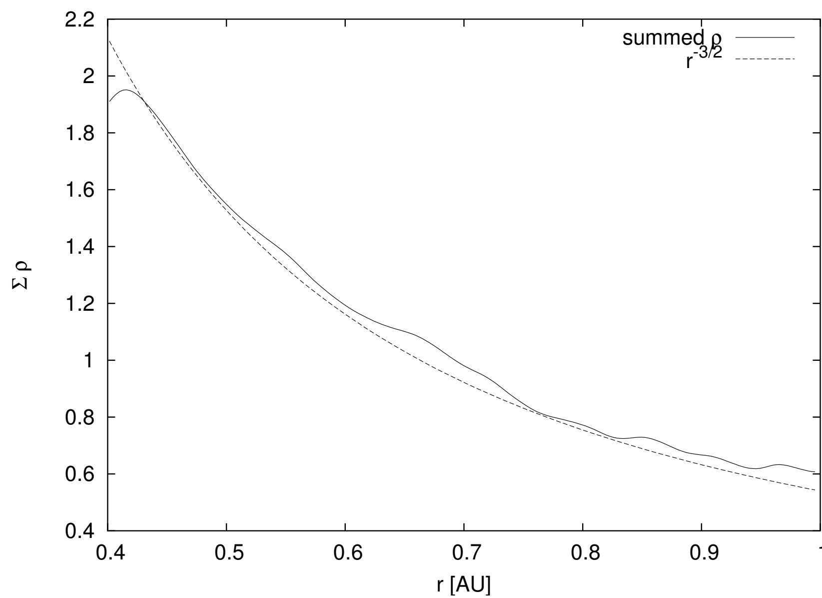


Figure 15: Isothermal accretion disk (fig. 14) radial density distribution summed vertically. The dashed curve is a $r^{-3/2}$ dependency (which is the initial condition) fitted to the summed density.

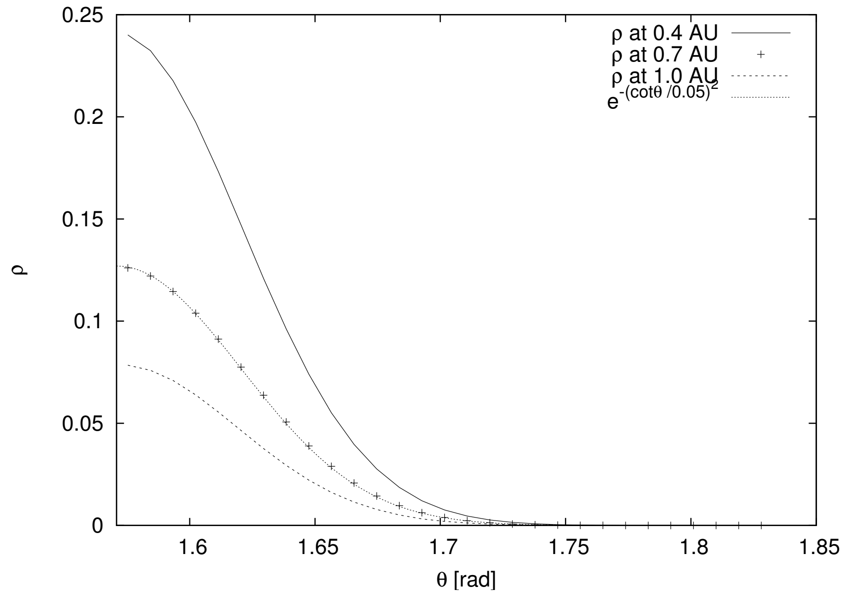


Figure 16: Isothermal accretion disk (fig. 14) vertical density distribution at different radii. The dotted curve is a Gaussian fit to the vertical distribution at 0.7 AU (crosses).

the disk is 30 degree while only half of the disk is calculated exploiting mirror symmetry. The protoplanet is Jupiter-like and has been placed into an unperturbed disk with initial conditions like the non-viscous accretion disks above. Boundaries are closed at all faces.

In fig. 17 you can see the density distribution in a slice through the midplane, fig. 18 is a vertical slice at the planet position, both after 150 orbital periods of the planet. In fig. 19 you can see the gap formed after this time illustrated by a radial plot of the vertically and azimuthally summed density. These results agree with those by D'Angelo (2003) and Schäfer et al. (2004).

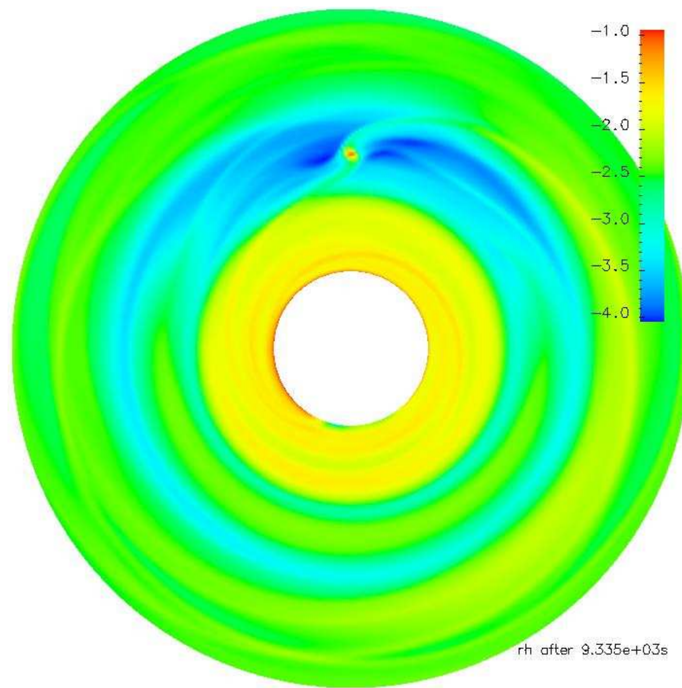


Figure 17: Protoplanetary disk logarithmic density distribution after 150 orbital periods of the planet. Slice through the midplane of the three-dimensional disk with a grid size of 128x24x384 ($r \,\mathrm{x}\, \theta \,\mathrm{x}\, \varphi$). The disk gap cleared by the Jupiter planet is clearly visible.

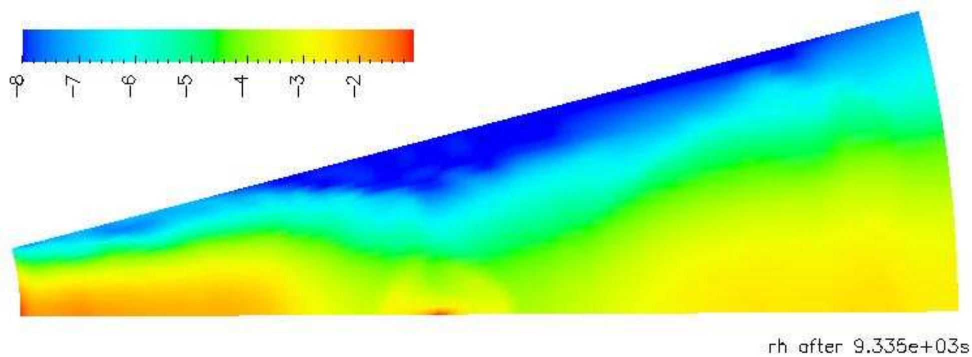


Figure 18: Protoplanetary disk logarithmic density distribution after 150 orbital periods of the planet. Vertical slice at the Jupiter planet position (fig. 17).

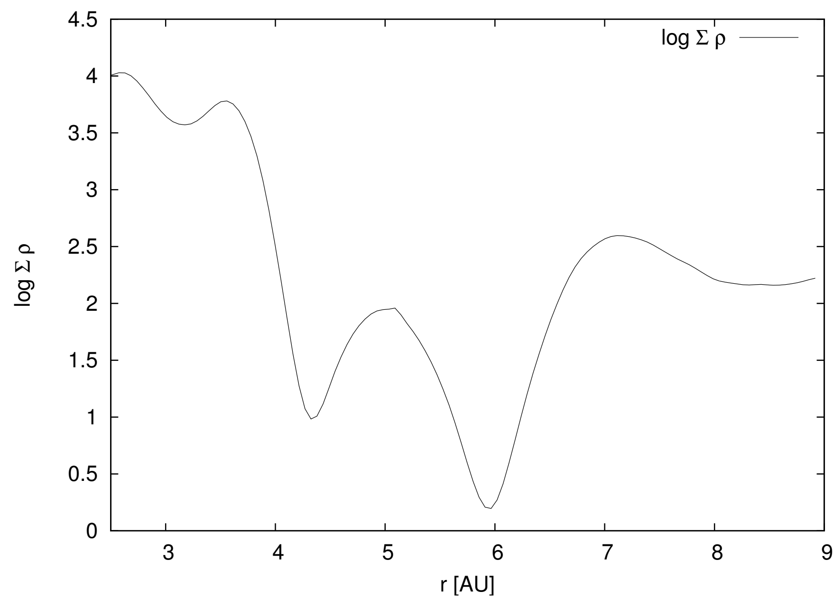


Figure 19: Protoplanetary disk gap after 150 orbital periods of the planet. Jupiter planet is at 5.1 AU. Logarithmic vertically and azimuthally summed radial density structure.

5 Stellar Oscillations

As part of writing a new three-dimensional hydrodynamics code there is a need for test problems with known (semi-)analytical results. Radial modes of stellar oscillations of spherically symmetric polytropes are such a problem. One-dimensional simulations in spherical symmetry are not interesing for testing the Poisson-solver. While three-dimensional simulations in Cartesian geometry are useful in this regard, they are computationally expensive. So our goal is to create a two-dimensional problem similar to that of stellar oscillations while being able to semi-analytically verify the results.

At this point we will look at the following three oscillator problems of different geometric structure and dimensionality:

1. Oscillations of a spherically symmetric (three-dimensional) polytrope.
2. Oscillations of an axisymmetric, two-dimensional cylindrical ($r - \varphi$) polytrope infinitely extended in the z-direction.
3. Oscillations of an one-dimensional wall-like (x) polytrope infinitely extended along the y-z-plane.

Only the first problem is a real physical system, the others are unphysical, but still useful for testing. For reference, fig. 20 sketches the configurations of these problems. The time-evolution equations for these problems are the Euler equations for a polytrope including the Poisson equation and the polytropic equation of state:

$$\frac{\partial \varrho}{\partial t} + \mathbf{u} \cdot \nabla \varrho = -\varrho \nabla \cdot \mathbf{u} \tag{5.1}$$

$$\frac{\partial \mathbf{u}}{\partial t} + (\mathbf{u} \cdot \nabla) \mathbf{u} = -\frac{1}{\varrho} \nabla p - \nabla \phi \tag{5.2}$$

$$\frac{\partial p}{\partial t} + \mathbf{u} \cdot \nabla p = -\gamma p \nabla \cdot \mathbf{u} \tag{5.3}$$

$$\nabla^2 \phi = 4\pi G \varrho \tag{5.4}$$

$$p = K \varrho^{1+\frac{1}{n}} \tag{5.5}$$

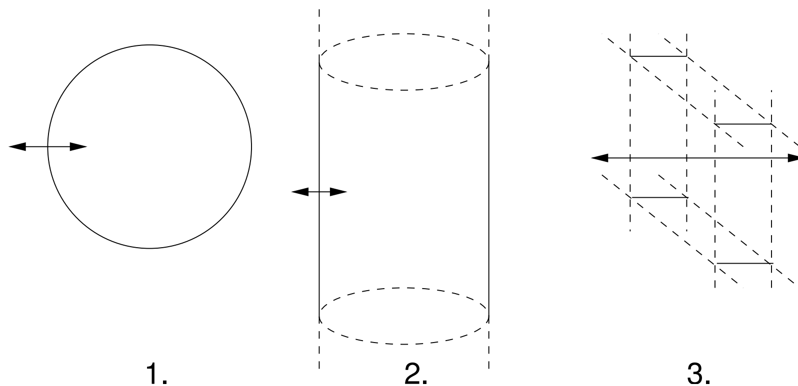


Figure 20: Configurations of polytropic oscillation problems. 1. three-dimensional sphere. 2. two-dimensional cylinder. 3. one-dimensional wall.

In the time-evolution the adiabatic index γ can be different from the polytropic index n of the equilibrium structure. Choosing the correct coordinate-system enables us to reduce the above equations to one dimension for all of the above problems. Namely spherical r for case one, cylindrical r for case two, and Cartesian x for case three.

All the above configurations can be looked at in two cases, first honouring the complete set of equations, and second, using the Cowling approximation. Cowling's approximation is to take the gravitational potential as constant in time and as such not solving the Poisson equation (5.4) but for the equilibrium structure.

In the rotating case such configurations have been studied in great detail, while non-rotating polytropes have not been dealt with in an unified description and linear analysis to our knowledge.

5.1 Equilibrium Structure

Following Kippenhahn & Weigert (1991) one can construct equilibrium density and gravitational potential distributions for polytropes known as the *Lane-Emden* polytropes. We start from radial hydrostatic equilibrium, the poisson equation and the polytropic closure:

$$\nabla\phi = -\frac{1}{\varrho}\nabla p \tag{5.6}$$

$$\nabla^2\phi = 4\pi G\varrho \tag{5.7}$$

$$p = K\varrho^{1+\frac{1}{n}}\,. \tag{5.8}$$

Substituting eqn. (5.8) into eqn. (5.6) and integrating (with $\phi = 0$ at the surface $\varrho = 0$), one gets

$$\varrho = \left(\frac{-\phi}{(n+1)\,K}\right)^n\,. \tag{5.9}$$

Introducing eqn. (5.9) into the right-hand side of eqn. (5.7) yields an ordinary differential equation for ϕ:

$$\nabla^2\phi = 4\pi G\left(\frac{-\phi}{(n+1)\,K}\right)^n\,.$$

One now defines the dimensionless variables z and w by

$$z = Ar\,, \qquad A^2 = \frac{4\pi G}{(n+1)\,K}\varrho_c^{\frac{n-1}{n}}\,, \qquad w = \frac{\phi}{\phi_c} = \left(\frac{\varrho}{\varrho_c}\right)^{\frac{1}{n}}$$

with ϕ_c and ϱ_c being appropriate values at the sphere center. This leads to the so-called Lane-Emden equation

$$\nabla^2 w + w^n = 0\,, \tag{5.10}$$

which describes the equilibrium structure of a Lane-Emden polytrope and can be numerically integrated. From eqn. (5.9) we see that the corresponding gravitational potential inside the gaseous structure is of the form

$$\phi = -\varrho^{\frac{1}{n}}\,(n+1)\,K\,, \tag{5.11}$$

and different outside, dependend on the geometry. Eqn. (5.9) in the case of vacuum ($\varrho = 0$) and spherical coordinates (case one) is $\frac{1}{r^2}\partial_r\left(r^2\partial_r\right)\phi = 0$ which leads to

$$\phi_{\rm outer} = -\frac{C_1}{r} + C_2\,, \tag{5.12a}$$

in cylindrical symmetry (case two), solving $\frac{1}{r}\partial_r\left(r\partial_r\right)\phi = 0$ yields

$$\phi_{\rm outer} = C_1 \ln r + C_2\,, \tag{5.12b}$$

and in the Cartesian case (case three), $\partial_x\partial_x\phi = 0$, the outer gravitational potential looks like

$$\phi_{\rm outer} = C_1 x + C_2\,. \tag{5.12c}$$

One needs to assure continuity between ϕ and $\phi_{\rm outer}$ by choosing C_1 and C_2. C_1 can generally be identified with the integral over the density ϱ, which is the stellar mass in the three-dimensional spherical case.

For three values of n there are analytic solutions for $w(z)$ in spherical coordinates, namely for $n = 0$: $w(z) = 1 - \frac{1}{6}z^2$, for $n = 1$: $w(z) = \frac{\sin z}{z}$ and for $n = 5$: $w(z) = \left(1 + z^3/3\right)^{-1/2}$. In the cylindrical case there is $n = 0$: $w(z) = 1 - \frac{1}{4}z^2$. In Cartesian geometry there are at least $n = 0$: $w(z) = 1 - \frac{1}{2}z^2$, and $n = 1$: $w(z) = \sin z$.

More details on the equilibrium structure of the infinite polytropic cylinder can be found in Ostriker (1964).

5.2 Linear Analysis

While in the spherically symmetric case the radial modes are well-known (e.g. Hurley et al. 1966), for the modified two- and one-dimensional cases we could not find sources with tabulated oscillation periods of radial modes. We can use linear perturbation theory as it has been done for the spherically symmetric case (e.g. Ledoux & Walraven 1958) to numerically find eingenmodes and eigenfunctions of the systems.

Looking at small perturbations from the equilibrium, the linearization of eqns. (5.1)–(5.4) in the perturbed quantities $\tilde{\varrho} = \varrho_0 + \varrho$, $\tilde{\mathbf{u}} = \mathbf{u}_0 + \mathbf{u}$, $\tilde{p} = p_0 + p$ and $\tilde{\phi} = \phi_0 + \phi$ leads to

$$\frac{\partial \varrho}{\partial t} + \mathbf{u}\cdot\nabla\varrho_0 = -\varrho_0\nabla\cdot\mathbf{u} \tag{5.13}$$

$$\frac{\partial \mathbf{u}}{\partial t} = -\frac{1}{\varrho_0}\nabla p + \frac{\varrho}{\varrho_0^2}\nabla p_0 - \nabla\phi \tag{5.14}$$

$$\frac{\partial p}{\partial t} + \mathbf{u}\cdot\nabla p_0 = -\gamma p_0\nabla\cdot\mathbf{u} \tag{5.15}$$

$$\nabla^2\phi = 4\pi G\varrho\,. \tag{5.16}$$

Using the following identity from combining the hydrostatic equilibrium in equation (5.2) with equation (5.4):

$$\frac{1}{\varrho_0}\nabla^2 p_0 - \frac{1}{\varrho_0^2}\nabla p_0\cdot\nabla\varrho_0 = \nabla\cdot\left(\frac{1}{\varrho_0}\nabla p_0\right) = -4\pi G\varrho_0\,, \tag{5.17}$$

and noting that $\partial_t \nabla\phi$ can be integrated from equation (5.16) using Gauss theorem like

$$\begin{aligned} \nabla\cdot\frac{\partial}{\partial t}\nabla\phi &= 4\pi G\frac{\partial\varrho}{\partial t} \\ \int\nabla\cdot\frac{\partial}{\partial t}\nabla\phi\,\mathrm{d}V &=_{(5.13)} -4\pi G\int\nabla\cdot(\varrho_0\mathbf{u})\,\mathrm{d}V \\ \oint\frac{\partial}{\partial t}\nabla\phi\,\mathrm{d}\mathbf{A} &= -4\pi G\oint\varrho_0\mathbf{u}\,\mathrm{d}\mathbf{A} \end{aligned}$$

$$\frac{\partial}{\partial t}\nabla\phi = -4\pi G\varrho_0\mathbf{u}\,, \tag{5.18}$$

while in case of applying the Cowling approximation $\partial_t\nabla\phi$ vanishes completely. Taking the derivative of equation (5.14) with respect to t, combining with equations (5.13) and (5.15) and using (5.17) and (5.18) finally leads to the following time-dependent equation for $\mathbf{u}$:

$$\frac{\partial^2\mathbf{u}}{\partial t^2} = \frac{\gamma}{\varrho_0}\nabla\left(p_0\nabla\cdot\mathbf{u}\right)\underline{-4\pi G\varrho_0\mathbf{u}}\,,$$

where the underlined term appears in case of using the Cowling approximation. Using the ansatz $\mathbf{u}(\mathbf{r},t) = \mathbf{r}W(\mathbf{r})e^{i\omega t}$ we can separate a time-independent equation for $W(\mathbf{r})$ and ω:

$$\frac{\gamma}{\varrho_0}\nabla\left[p_0\nabla\cdot(\mathbf{r}W(\mathbf{r}))\right] + \mathbf{r}W(\mathbf{r})\left(\omega^2\underline{-4\pi G\varrho_0}\right) = 0\,.$$

In the Cartesian case, the equation for the x component $W(x)$ and ω then is the following:

$$\frac{\mathrm{d}^2W}{\mathrm{d}x^2} + \left(\frac{2}{x}+\frac{1}{p_0}\frac{\mathrm{d}p_0}{\mathrm{d}x}\right)\frac{\mathrm{d}W}{\mathrm{d}x} + \frac{\varrho_0}{\gamma p_0}\left(\frac{\gamma}{x\varrho_0}\frac{\mathrm{d}p_0}{\mathrm{d}x} + \omega^2\underline{-4\pi G\varrho_0}\right)W = 0\,,$$

where again the underlined term appears in case of using the Cowling approximation. This ordinary differential equation is a standard Sturm-Liouville boundary value problem for the eigenvalue ω and the eigenfunction $W(x)$.

In the cylindrical case we apply similar mathematics and finally end up with the following equation for $W(r)$ and ω:

$$\frac{\mathrm{d}^2W}{\mathrm{d}r^2} + \left(\frac{3}{r}+\frac{1}{p_0}\frac{\mathrm{d}p_0}{\mathrm{d}r}\right)\frac{\mathrm{d}W}{\mathrm{d}r} + \frac{\varrho_0}{\gamma p_0}\left(\frac{1}{r\varrho_0}\frac{\mathrm{d}p_0}{\mathrm{d}r}(2\gamma-2) + \omega^2\underline{-4\pi G\varrho_0}\right)W = 0$$

and the equation in spherical symmetry looks like

$$\frac{\mathrm{d}^2W}{\mathrm{d}r^2} + \left(\frac{4}{r}+\frac{1}{p_0}\frac{\mathrm{d}p_0}{\mathrm{d}r}\right)\frac{\mathrm{d}W}{\mathrm{d}r} + \frac{\varrho_0}{\gamma p_0}\left(\frac{1}{r\varrho_0}\frac{\mathrm{d}p_0}{\mathrm{d}r}(3\gamma-4) + \omega^2\underline{-4\pi G\varrho_0}\right)W = 0\,,$$

where the underlined terms again appear in case of using Cowling's approximation.

These ordinary differential equations can now be numerically integrated for varying ω to obtain a series of eigenvalues and eigenfunctions of the oscillating linear system. One uses a so called *shooting technique* where one solves the boundary value problem by varying ω to make the integrated eigenfunction honor the boundary condition, which is a vanishing

n	γ	1st	2nd	3rd	4th	5th	6th
1.0	2.0	2.87 (2.15)	1.39 (1.27)	0.96 (0.91)	0.73 (0.71)	0.60 (0.58)	0.50 (0.50)
1.5	5/3	4.57 (3.07)	2.12 (1.87)	1.46 (1.36)	1.12 (1.07)	0.91 (0.89)	0.77 (0.76)
2.0	1.5	7.21 (4.29)	3.15 (2.69)	2.18 (1.99)	1.68 (1.58)	1.37 (1.32)	1.17 (1.13)

Table 8: Oscillation periods in normalized units of a spherical symmetric star for different n, γ. Values in brackets are for the Cowling approximation.

n	γ	1st	2nd	3rd	4th	5th	6th
1.0	2.0	2.13 (1.70)	1.05 (0.98)	0.72 (0.69)	0.55 (0.54)	0.44 (0.44)	0.37 (0.37)
1.5	5/3	2.78 (2.13)	1.39 (1.28)	0.97 (0.93)	0.75 (0.73)	0.61 (0.60)	0.52 (0.51)
2.0	1.5	3.39 (2.53)	1.75 (1.59)	1.23 (1.17)	0.96 (0.93)	0.79 (0.77)	0.67 (0.66)

Table 9: Oscillation periods in normalized units of a two-dimensional axisymmetric cylinder for different n, γ. Values in brackets are for the Cowling approximation.

n	γ	1st	2nd	3rd	4th	5th	6th
1.0	2.0	1.37 (1.20)	0.68 (0.66)	0.46 (0.46)	0.35 (0.35)	0.28 (0.28)	0.24 (0.24)
1.5	5/3	1.54 (1.33)	0.82 (0.78)	0.57 (0.56)	0.44 (0.43)	0.36 (0.35)	0.30 (0.30)
2.0	1.5	1.65 (1.43)	0.92 (0.88)	0.65 (0.64)	0.51 (0.50)	0.42 (0.41)	0.35 (0.35)

Table 10: Oscillation periods in normalized units of a one-dimensional Cartesian wall for different n, γ. Values in brackets are for the Cowling approximation.

Lagrangian pressure at the stellar surface ($\nabla^2 p = 0$). This technique is e.g. described in Bardeen et al. (1966).

Using appropriate normalization for the equations, namely $G = 1$, $\varrho_c = 1$ and $\phi_c = 1$ one can generate the equilibrium structure ϱ_0, p_0 and ϕ_0 by solving the Lane-Emden equation and then apply the linear analysis for different γ and n. Oscillation periods corresponding to the eigenvalues are tabulated for certain values of γ and n in tables 8–10. Non-linear one-dimensional simulations with small perturbations of the initial equilibrium models confirm these results.

5.3 Boundary Conditions

To simulate a star on a grid one needs proper boundary conditions for the stellar boundary and possibly its interior. As we consider a Cartesian grid where we can cover the whole stellar body, we only need to implement proper boundary conditions for the stellar surface. With an Eulerian approach, tracking the stellar surface is difficult, so we choose to approximately model the vacuum region outside of the star and impose proper boundary conditions on the computational domain instead.

Vacuum is represented using a minimum numerical density of e.g. 10^{-6} that is forced throughout the simulation. This helps to avoid too steep density gradients and possibly break-down of the numerical scheme. The gravitational potential outside of the equilibrium star is chosen to be the analytical solution, in case of not solving the poisson equation.

For the velocity we need to choose boundary conditions that do not destroy the spherical symmetry of the star, as due to the lack of an equilibrium model for the density outside of the star there is free-falling material coming down onto the star with very high velocities acting as an unwanted perturber. A convenient solution to this problem is to impose a spherical shell outside of the star where minimum-density material can start free-falling onto the star surface (fig. 24). The distance of the shell from the stellar boundary determines the maximum in-fall velocities. The spherical symmetry of the free-falling material does not destroy the shape of the star too much.

A steady shock is formed at the stellar boundary this way so we need to include an artificial viscosity to maintain the correct shock solution. In fig. 21 you can see the steady solution for the one-dimensional problem imposing the aforementioned boundary condition.

5.4 Testing the new Hydro Code

All of the following calculations use a von Neumann-Richtmyer artificial viscosity (Stone & Norman 1992) to stabilize the stellar boundary. Appropriate factors for the CFL condition are 0.6 for 2d, 0.5 for 3d and 0.8 for the 1d case. These were verified by comparing with calculations with a CFL factor of 0.1.

5.4.1 One-dimensional Calculations

One-dimensional unperturbed calculations show initial numerical noise decaying quickly and a constant mass accretion rate due to the boundary conditions. Fig. 22 shows the velocity magnitude and the density at the star center developing over time. After the plotted time the fluctuations will eventually start to grow again due to the accreting density profile not matching the constant-in-time potential.

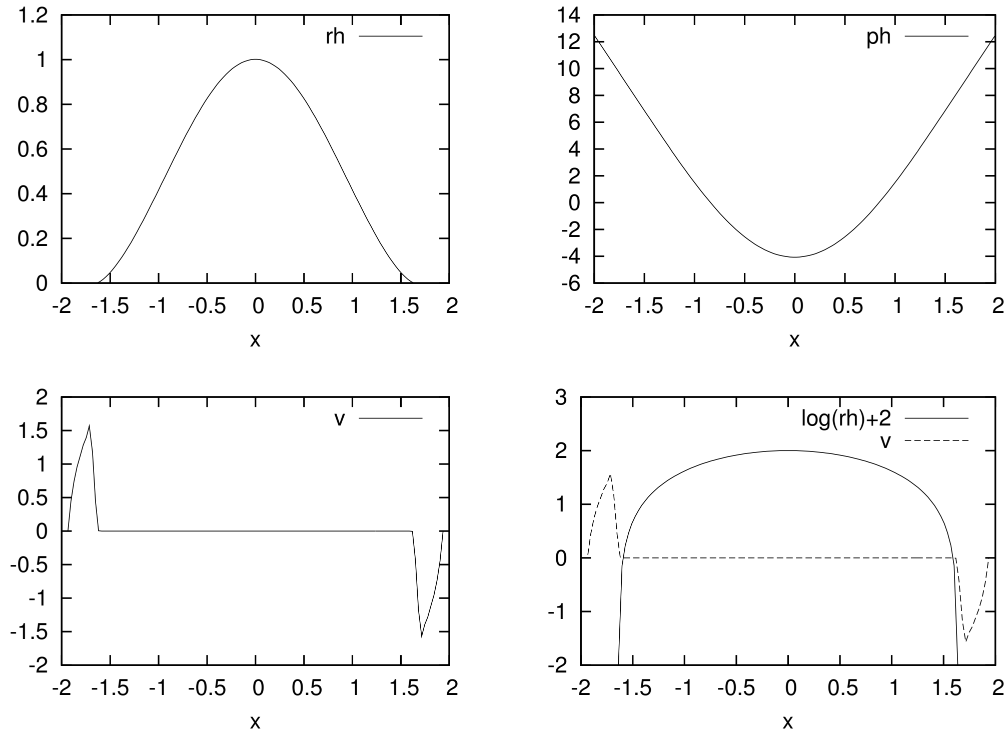


Figure 21: Steady shock solution of the unperturbed one-dimensional case.

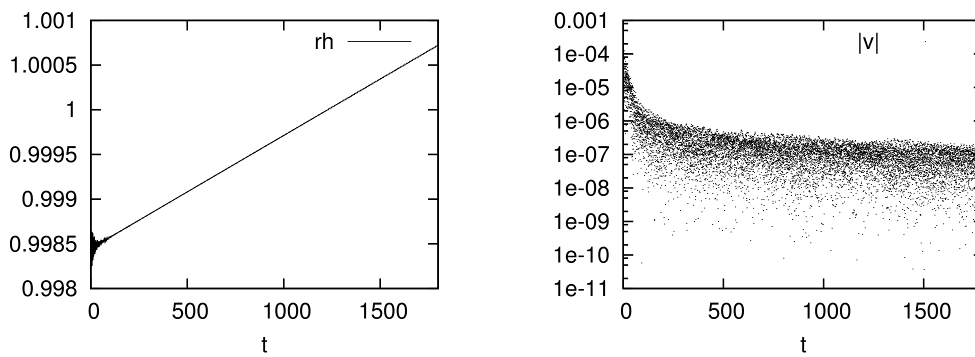


Figure 22: Unperturbed evolution of the velocity and density of a one-dimensional Cartesian wall for 1350 oscillation periods of the first mode.

n	γ	1st	2nd	3rd	4th
1.0	2.0	1.20	0.66	0.46	0.35
1.5	5/3	1.34	0.78	0.56	0.49
5/3	1.6	1.37	0.82	0.59	0.51
2.0	1.5	1.43	0.88	0.64	0.54

Table 11: Oscillation periods of a one-dimensional Cartesian wall for different n and γ.

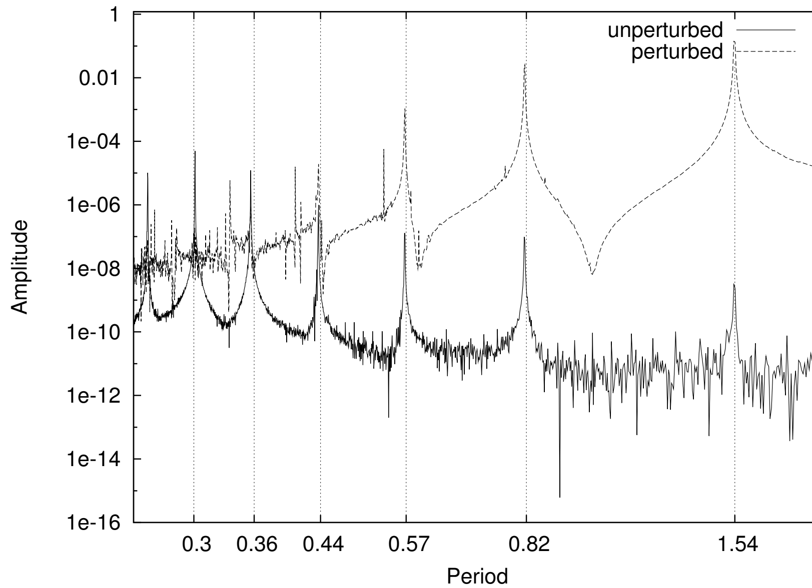


Figure 23: Spectrum of one-dimensional Cartesian oscillations in Cowling approximation obtained at the wall's center. Perturbed and unperturbed initial model. Vertical bars denote oscillation periods from the linear analysis. Extra peaks are due to aliasing artifiacts of the Fourier transformation.

We excite the stars by applying a density perturbation using spherical harmonics. For the fundamental radial mode we use a $\cos r$ dependence, correcting for eventual mass losses or gains. If we excite the density with the first fundamental radial mode of amplitude of 1% and apply Cowling's approximation, we can extract oscillation periods by means of a Fourier transformation of the density and velocity variations at different radii over time. In fig. 23 you can see the oscillation spectrum of this setup both in the perturbed and the unperturbed case. In the perturbed case amplitudes are much higher and the ground modes are stronger excited. In the unperturbed case perturbations are caused by numerical noise on a small scale, so higher modes get excited first and with higher amplitude. Looking for correlations between the periodic signals at different radii and between density and velocity perturbations we can derive the data in table 11. Higher order modes are in the noise in case of the perturbed calculation.

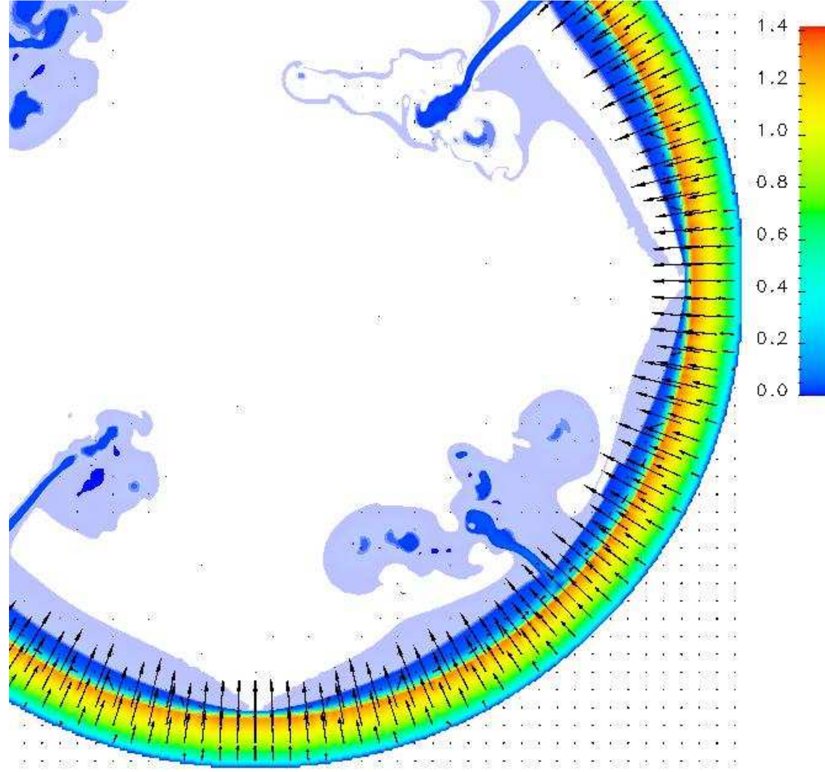


Figure 24: Flow pattern of a two-dimensional unperturbed cylinder in Cartesian coordinates. The high velocity region is the imposed shell-shaped boundary condition. Color coded velocity magnitude after 120 oscillations of the first radial mode. For emphasis reason blue (zero velocity) is faded to white.

5.4.2 Two- and Three-dimensional Calculations

For two- (and three-)dimensional simulations in Cartesian coordinates choosing appropriate boundary conditions as stated in section 5.3 is especially important. An unperturbed model with such boundary setup survives a long time without visible perturbation. In fig. 25 you can see the evolution of stellar mass, central and outer density of an equilibrium model during the time of 250 oscillation periods of the first radial mode. You can see the outer density fluctuating between the numerical minimal density of 10^{-6} and a value 0.1% higher. The central density is oscillating with a very small amplitude and linearily raising over time, as is the total stellar mass. The latter is because of the boundary condition imposing a constant flow of thin material onto the star. Fig. 24 shows the flow pattern after 250 oscillation periods of the first radial mode and the structure of the boundary condition. In fig. 26 you can see the difference in densities of the initial equilibrium model and the final state. Initial core density is 1 and minimum density is 10^{-6}. In the diagonal direction is the only place where one can see effects of the Cartesian geometry. Analyzing the oscillation of the density inside the unperturbed star shows the familiar radial modes with very low amplitudes as can be seen in fig. 27.

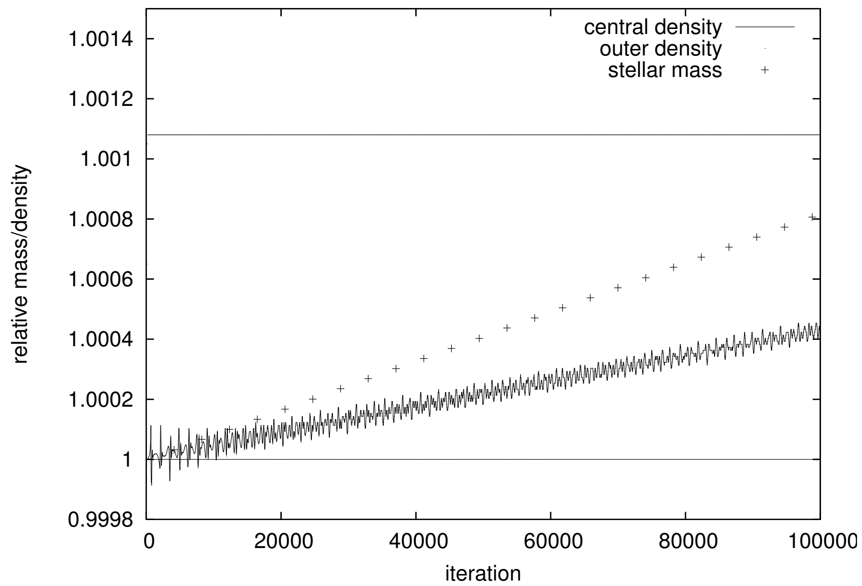


Figure 25: Mass and density evolution of an unperturbed star.

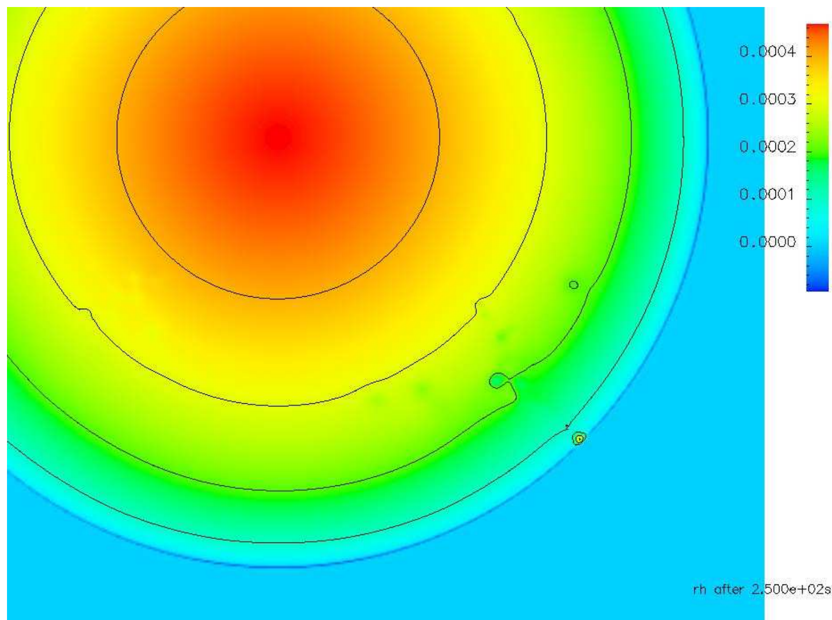


Figure 26: Density structure of an unperturbed star compared to its initial equilibrium model after 120 oscillation periods of the first radial mode. Linear difference is color-coded, iso-lines at $1 \cdot 10^{-4}$, $2 \cdot 10^{-4}$, $3 \cdot 10^{-4}$ and $4 \cdot 10^{-4}$.

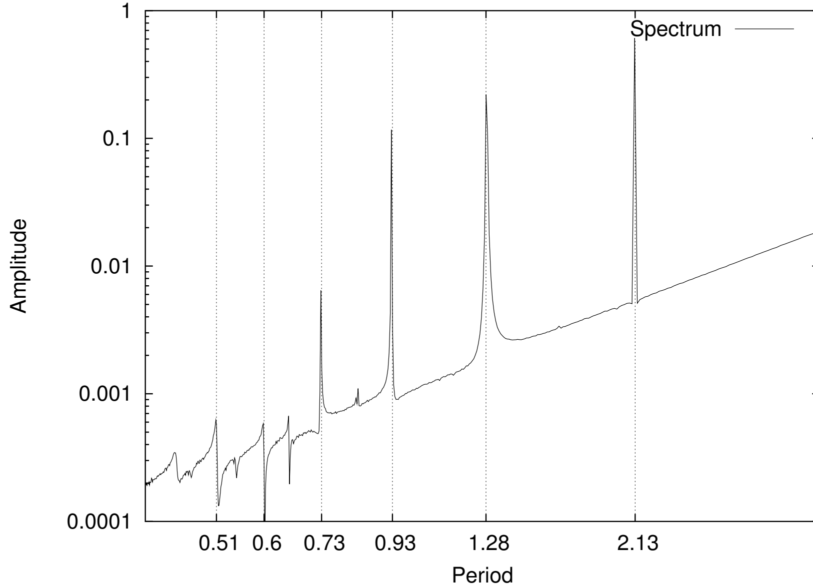


Figure 27: Spectrum of central density fluctuations of an unperturbed cylinder in Cowling approximation. Vertical bars denote oscillation periods from the linear analysis.

In three dimensions we can reproduce nicely the analytic radial modes exciting the first mode with an amplitude of 1% of the density while applying the Cowling approximation. In fig. 28 you can see the spectrum of the oscillating density in the center of the star resulting from a 128 cubed simulation.

5.4.3 Testing the Poisson Solver

To verify correct operation of the Poisson solver we have done truly self-gravitating calculations in two dimensions with a constant in time Dirichlet boundary condition for the gravitational potential. A comparison of the excited modes in this non-linear calculation with the radial linear analysis can be seen in fig. 29 where the Fourier spectrum of the non-linear calculation is plotted and the radial modes from the linear analysis are denoted as vertical lines. In addition to the radial modes we see excitement of a non-radial $l = 4$ mode looking at the density perturbation $(\varrho - \varrho_0)$ in fig. 30.

The excitement of non-radial modes is both a consequence of the Cartesian grid and of the boundary conditions for solving the Poisson equation. In the spectrum one can see that e.g. the radial mode at $p = 0.97$ is shifted compared to the linear analysis. We believe this is due to non-linear coupling between the radial and the non-radial modes.

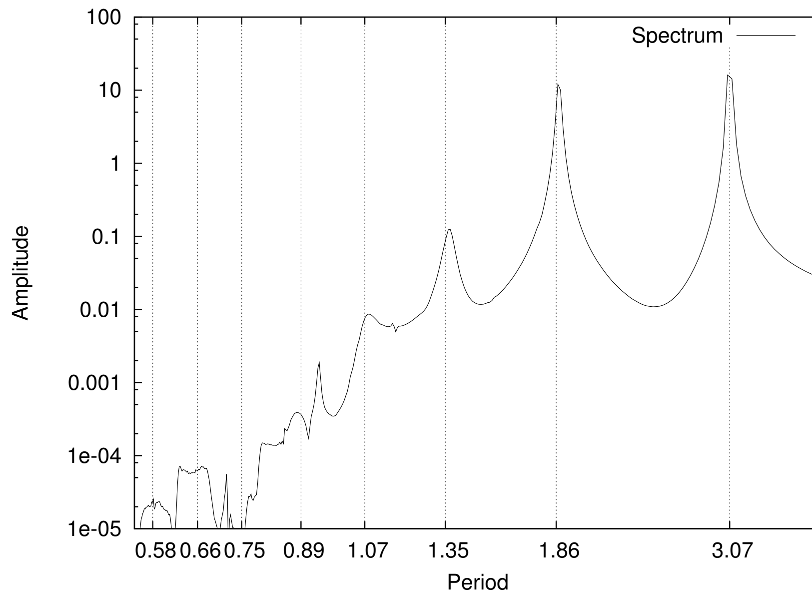


Figure 28: Spectrum of central density fluctuations of an excited three-dimensional star in Cowling approximation. Vertical bars denote oscillation periods for radial modes from the linear analysis.

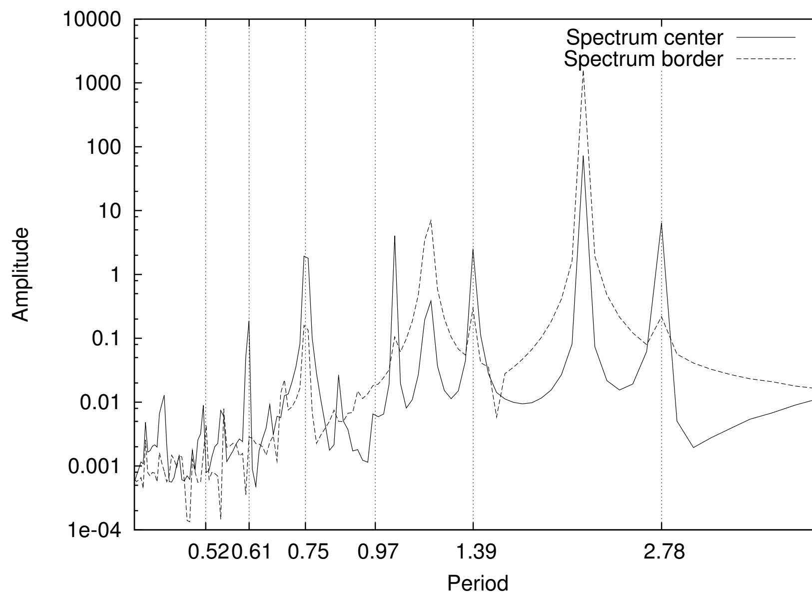


Figure 29: Spectrum of density fluctuations of an unperturbed cylinder with solving of the Poisson equation. Vertical bars denote oscillation periods for radial modes from the linear analysis.

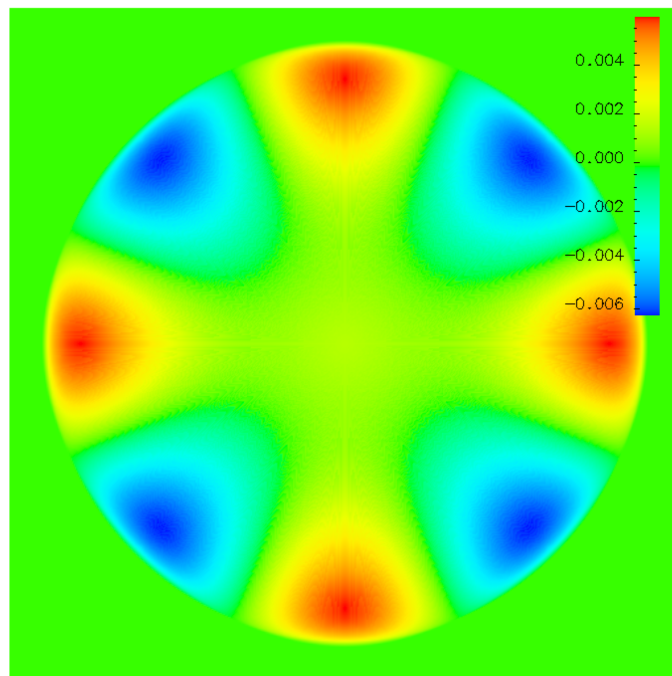


Figure 30: Density fluctuations $(\varrho - \varrho_0)$ of an unperturbed cylinder with solving of the Poisson equation after 16 oscillation periods of the first radial mode.

6 Circumbinary Disks

We study the evolution of circumbinary disks surrounding classical T Tauri binary systems. High resolution numerical simulations are employed to model a system consisting of a central eccentric binary star within an accretion disk. The disk is assumed to be infinitesimally thin, however a detailed energy balance including viscous heating and radiative cooling is applied. For close systems also irradiation from the stars is taken into account. This allows accurately calculating the emitted spectral energy distribution.

A novel numerical approach using a parallelized Dual-Grid technique on two different coordinate systems has been implemented to solve the problem of not treating the interior of the binary orbit with the same resolution as the circumbinary disk. In additional to the finite difference calculations verifications using the Smoothed Particle Hydrodynamics method are employed to compare the hydrodynamical features and strengthen our conclusions.

Physical parameters of the setup are chosen to model the close systems of DQ Tau and AK Sco, as well as the wider systems of GG Tau and UY Aur. Our main findings are for the tight binaries a substantial flow of material through the disk gap which is accreted onto the central stars in a phase dependent process (Günther & Kley 2002). We are able to constrain the parameters of the systems by matching both accretion rates and derived spectral energy distributions to observational data where available. We find that the incorporation of irradiation effects is necessary to obtain correct disk temperatures and spectra for the close systems (Günther et al. 2004).

6.1 Introduction

Observations of main sequence stars have shown that about 60% belong to binary or multiple systems (Duquennoy & Mayor 1991). However, recent observations of pre-main sequence stars seem to indicate that nearly all stars are born in multiple systems (Reviews in Mathieu et al. 2000; Zinnecker & Mathieu 2001).

Apart from many spectroscopic binary systems with surrounding disks like DQ Tau and AK Sco, there have been only two observations of directly imaged wide optical binary systems with circumbinary disks, namely GG Tau (Dutrey et al. 1994) and UY Aur (Duvert et al. 1998). The observations of the spectroscopic binaries usually include emission data from the infrared and optical part of the spectrum. Observations of the luminosities of the boundary layer are typically used to estimate accretion rates onto the binary stars (Gullbring et al. 1998; Hartigan et al. 1995).

Bate & Bonnell (1997) model the early phase of a molecular cloud collapsing onto a protobinary system with an SPH code and derive properties of the created circumbinary and circumstellar disks. We are interested in the following phase of the evolution when the envelope has been cleared. By modeling the evolution we compare fully developed circumbinary and circumstellar disks and their properties with observational data.

In a system consisting of a stellar binary surrounded by an accretion disk there is a continuous exchange of angular momentum between the binary and the disk. As the binary is orbiting with a shorter period than the disk material it is generating positive angular momentum waves in the disk which, upon dissipating, deposit their energy and angular momentum in the disk. Hence, there is a continuous transfer of angular momentum from

the binary to the outer disk. Upon receiving this angular momentum, the inner disk speeds up, and the material is receding from the binary, leaving a region of very low density around the central binary, a so called *inner gap*. At the same time, the loss of angular momentum leads to a secular shrinkage of the semi-major axis of the binary. The width and form of the gap depend on disk parameters such as temperature and viscosity and on the orbital parameters (eccentricity) of the binary as well (Artymowicz et al. 1991; Artymowicz & Lubow 1994).

First calculations dealing with the evolution of circumbinary disks so far have been using the numerical method of smoothed particle hydrodynamics. Applying such a scheme Artymowicz & Lubow (1996) could demonstrate that for sufficiently high viscosities and temperatures material from the disk is able to penetrate into the inner gap region of the disk to become eventually accreted by one of the binary stars. This process which is facilitated by a gas flow along the saddle points of the potential leads to a *pulsed accretion* whose magnitude and phase dependence for eccentric binaries has been estimated. However, when using only a small number of particles all having the same mass only a limited resolution is achievable.

Later Rozyczka & Laughlin (1997) used a finite difference method in attacking this problem. Using a constant kinematic viscosity coefficient and eccentric binaries they also found, in agreement with Artymowicz & Lubow (1996), a *pulsed accretion flow* across the gap, with most of the accretion occurring onto the lower mass secondary star.

Here we extend these computations and solve explicitly a time dependent energy equation which includes the effects of viscous heating and radiative cooling. We aim at the structure and dynamics of the disk as well as the gas flow in the close vicinity of the binary star. To that purpose we utilize a newly developed method which enables us to cover the whole spatial domain. For the first time we perform long-time integration of the complete system covering several hundred orbital periods of the binary and compare the properties of the evolved systems with observational data such as spectral energy distributions in the infrared and optical bands and accretion rates estimated from luminosities.

Furthermore we extend the model by accounting for irradiation effects which are important for passive accretion disks such as the disks of DQ Tau and AK Sco, according to observational data. Additionally, refined observational data is available from Alencar et al. (2003) for the AK Sco system which allows to better constrain the parameters of the system by our numerical simulations.

6.2 Physical Model

6.2.1 General Layout

The system consists of a stellar binary system and a surrounding circumbinary accretion disk. We assume a geometrically thin disk and describe the disk structure by means of a two-dimensional, infinitesimal thin model in the $z = 0$ plane with the origin at the center of mass of the binary. We use vertically averaged quantities, such as the surface mass density

$$\Sigma = \int_{-\infty}^{\infty} \rho dz,$$

where ρ is the regular three-dimensional density. We work with two different coordinate systems. Firstly, a cylindrical coordinate system (r, φ) centered on the center of mass of

the binary stars. As such a coordinate system does not allow for a full coverage of the plane, we overlay the center additionally with a Cartesian (x, y) grid (see sec. 6.4.4).

The gas in the disk is non-self-gravitating and is orbiting a binary system with stellar masses M_1 and M_2 which are modeled by the gravitational potential of two solid spheres. The binary is fixed on a Keplerian orbit with given semimajor axis a and eccentricity e. Hence, we neglect self-gravity and the gravitational back-reaction of the disk onto the stars. This is justified because the total mass of the disk M_d within the simulated region, which extends up to several tens of semi-major axis a of the binary, is typically much smaller than that of the central binary. Hence, the timescale for a change in a and e is indeed much longer than the timescales considered here (Artymowicz et al. 1991). For the majority of the computations we employ an inertial coordinate system, however test calculations have been performed in a system corotating with the binary as well.

6.2.2 Equations

The evolution of the disk is given by the two-dimensional (x, y or r, φ) evolutionary equations for the density Σ, the velocity field $\vec{u} \equiv (u_x, u_y) = (u_r, u_\varphi)$, and the temperature T. In a coordinate-free representation the equations read

$$\frac{\partial \Sigma}{\partial t} + \nabla \cdot (\Sigma \vec{u}) = 0 \tag{6.1}$$

$$\frac{\partial \Sigma \vec{u}}{\partial t} + \nabla \cdot (\Sigma \vec{u} \cdot \vec{u}) = -\nabla P - \Sigma \nabla \Phi + \nabla \cdot \mathbf{T} \tag{6.2}$$

$$\frac{\partial \Sigma \varepsilon}{\partial t} + \nabla \cdot [(\Sigma \varepsilon + P)\, \vec{u}] = -\Sigma \vec{u} \cdot \Phi - \nabla \cdot \left(\vec{F} - \vec{u} \cdot \mathbf{T} \right) \tag{6.3}$$

Here $\varepsilon = c_\mathrm{v} T + 1/2 u^2$ is the specific total energy, P is the vertically integrated (two-dimensional) pressure $P = R \Sigma T / \mu$ with the midplane temperature T and the mean molecular weight μ, which can be obtained by solving the Saha rate equations for Hydrogen dissociation and ionization and Helium ionization. $\vec{F}$ and $\mathbf{T}$ are the radiative flux and viscous stress tensor, respectively.
The gravitational potential Φ generated by the binary stars is given by

$$\Phi = -\sum_{i=1,2} \begin{cases} \dfrac{G\, M_i}{|\vec{r} - \vec{r}_i|} & \text{for } |\vec{r} - \vec{r}_i| > R_* \\[2ex] \dfrac{G\, M_i \left(3 R_*^2 - |\vec{r} - \vec{r}_i|^2\right)}{2\, R_*^3} & \text{for } |\vec{r} - \vec{r}_i| \le R_* \end{cases} \tag{6.4}$$

where $\vec{r}_i$ are the radius vectors to the two stars, and R_* is the stellar radius assumed to be identical for both stars. The second case in eq. (6.4) gives the potential inside a star with radius $R_\star$ having a homogeneous (constant) density.

6.2.3 Viscosity and Disk Height

The effects of viscosity are contained in the viscous stress tensor $\mathbf{T}$. Here we assume that the accretion disk may be described as a viscous medium driven by some internal turbulence which we approximate with a Reynolds ansatz for the stress tensor. The components of $\mathbf{T}$ in different coordinate systems are spelled out explicitly for example in Tassoul (1978).

A useful form for disk calculations considering angular momentum conservation is given in Kley (1999).

For the kinematic shear viscosity ν we assume in this work an α-model (Shakura & Sunyaev 1973) in the form

$$\nu = \alpha c_{\mathrm{s}} H \,, \tag{6.5}$$

where c_{s} and H are the local sound-speed and vertical disk height, respectively. The local vertical disk height $H(\vec{r})$ is computed from the vertical hydrostatic equilibrium

$$\frac{\partial \Phi}{\partial z} = \frac{1}{\rho}\frac{\partial p}{\partial z} \,, \tag{6.6}$$

where ρ and p are the standard three dimensional density and pressure. Substituting eq. (6.4) for the potential with $|\vec{r} - \vec{r}_i| > R_*$, assuming an ideal gas law $p = R\rho T/\mu$, and integrating over z yields

$$\rho = \rho_0 e^{-\left(\frac{1}{2}\frac{z^2}{H^2}\right)} \,, \tag{6.7}$$

where $\rho_0 = \Sigma/[(2\pi)^{1/2}H]$ is the midplane density, and H is the vertical disk height given by

$$H(\vec{r}) = \left(\sum_{i=1,2} \frac{GM_i}{c_{\mathrm{s}}^2 \, |\vec{r} - \vec{r}_i|^3} \right)^{-\frac{1}{2}} = \left(\sum_{i=1,2} H_i^{-2}(\vec{r}) \right)^{-\frac{1}{2}} \,. \tag{6.8}$$

This can be split into the single star disk heights given by $H_i(\vec{r}) = c_{\mathrm{s}}\sqrt{\frac{|\vec{r}-\vec{r}_i|^3}{GM_i}}$. The sound speed is given here by $c_{\mathrm{s}} = RT/\mu$. We assume a vanishing physical bulk viscosity ζ, but consider it in the artificial viscosity coefficient (see Kley 1999).

6.2.4 Radiative Balance

The influence of radiative and viscous effects on the disk temperature is treated in a local vertical balance. The balance equation for the internal and radiative energy considering these effects reads

$$\frac{\partial\left(\Sigma c_{\mathrm{V}} T\right)}{\partial t} = Q_{\mathrm{diss}} - Q_{\mathrm{rad}} + Q_{\mathrm{stars}} \,, \tag{6.9}$$

where Q_{diss} and Q_{rad} denote the viscous dissipation and radiative losses. Q_{stars} is the irradiation heating by the stars, which is considered for the optically thick regions of the circumbinary disk. This inclusion of irradiation effects is done in a simple geometric fashion due to the lack of explicit treatment of the vertical structure of the disk in these two-dimensional calculations. For the viscous dissipation we use the vertically averaged expression

$$Q_{\mathrm{diss}} = \vec{u} \cdot \nabla \mathbf{T} = \frac{1}{2\Sigma\nu} \mathrm{Tr}\left(\mathbf{T}^2\right) \,, \tag{6.10}$$

where $\mathbf{T}$ is the viscous stress tensor. For the radiative transport we use

$$Q_{\mathrm{rad}} = -\nabla \cdot \vec{F}_0 - \frac{\partial F_z}{\partial z} \,, \tag{6.11}$$

where $\vec{F_0}$ is the flux vector in the $z = 0$ plane. Here we consider only the losses in the vertical direction, i.e. $\vec{F_0} = 0$, a standard approximation in accretion disk theory. Integration over the vertical direction yields

$$Q_{\mathrm{rad}} = 2F_{\mathrm{rad}} = 2\sigma_{\mathrm{B}} T_{\mathrm{eff}}^4 \tag{6.12}$$

with the local effective (surface) temperature T_{eff}. Following Hubeny (1990) this temperature is related to the midplane temperature T by the LTE gray solution for the vertical structure of accretion disks. Simplifying this relation we get

$$T^4 = \frac{3}{4} T_{\mathrm{eff}}^4 \left[\frac{1}{2}\tau + \frac{1}{\sqrt{3}} + \frac{1}{3\tau} \right] . \tag{6.13}$$

The optical depth is calculated via (Ruden & Pollack 1991)

$$\tau = \frac{1}{2}\kappa\Sigma \tag{6.14}$$

using an interpolated opacity $\kappa(\rho, T)$ adapted from Lin & Papaloizou (1985).

6.3 Spectra Generation

Spectral energy distributions from the disk emission can be computed by integrating the flux of the blackbody radiation from the disk surface. This is done decoupled from the numerical simulation as a postprocessing step. We follow Adams et al. (1988), and compute the flux at the observer via

$$F_\nu = \frac{\cos i}{D^2} \int_{r_0}^{R_d} \int_0^{2\pi} B_\nu \left[T(r,\varphi) \right] \left(1 - e^{-\tau(r,\varphi)} \right) r \mathrm{d}\varphi \, \mathrm{d}r \tag{6.15}$$

where i is the inclination of the disk, D the distance to the observer and τ the line-of-sight optical depth through the disk, which can be approximated using the opacity κ, $\tau(r,\varphi) = \frac{\kappa\Sigma(r,\varphi)}{\cos i}$. The surface disk temperature T is estimated using the optical depth of the disk as in the radiative balance process.

This integration is limited to regions outside of the stellar cores as temperatures and densities inside the cores are not modeled correctly. The stellar emission is accounted for by adding two blackbody spectra for appropriate effective stellar temperatures. To compare with observational data all of the generated spectra are reddened according to an extinction A_v suggested by the cited references using the method from Cardelli et al. (1989).

6.4 Numerical Issues

In order to study the disk evolution, we utilize a finite difference method to solve the hydrodynamic equations outlined in the previous section. As we intend to model possible accretion flow onto the binary stars and to achieve a very good resolution in their vicinity, we use a *Dual-Grid*-technique.

The code is based on the hydrodynamics code RH2D suited to study general two-dimensional systems including radiative transport (Kley 1989). RH2D uses a spatially second-order accurate, mixed explicit and implicit method. Due to an operator-splitting method it is semi-second order accurate in time. The advection is computed by means

of the second order monotonic transport algorithm, introduced by van Leer (1977), which guarantees global conservation of mass and angular momentum. Advection and forces are solved explicitly, and the viscosity and radiation are treated implicitly. The formalism for application to thin disks in (r, φ) geometry has been described in detail in Kley (1998, 1999). Here this scheme is extended to allow for a Dual-Grid system (see 6.4.4) and a more detailed energy balance following eq. (6.9).

As the calculations have shown, the system behaves extremely dynamically with very strong, moving gradients at the inner edge of the disk. To model such a situation, the viscosity has to be treated implicitly to ensure numerical stability. The coupling of u_r and u_φ requires a solution method similar to that in Kley (1989). For stability purposes an additional implicit artificial viscosity has to be included in the computations (Kley 1999).

This code has been employed successfully in related simulations dealing with embedded planets in disks (Kley 1999, 2000). The numerical details of the present simulations are presented in Günther (2001).

6.4.1 Irradiation

For the heating from the stars which is considered for the close binary systems we use

$$Q_{\mathrm{stars}} = A \sigma_{\mathrm{B}} T_*^4 \sum_i \left(\frac{R_*}{r_i} \right)^2 , \tag{6.16}$$

where T_* is the effective blackbody temperature of the stars, R_* is the stellar radius and r_i are the distances of the stars to the local patch. The factor A honors the effective surface the flux operates on. This is obtained by reconstructing a surface from the radially varying pressure scale height. Self-shadowing is taken into account by setting A to zero for back-faces.

Numerical tests have shown that with the inclusion of irradiation effects we observe oscillatory behavior of the temperature near to the edge of the gap due to self-shadowing effects, as also found by Dullemond (2000). But we can overcome this specific instability by using appropriate time sub-stepping for the radiative balance. This suggests that if the radiative balance is imposed using a time relaxation process the self-shadowing instability does not occur. This may be either because the cooling timescale is not within the region of linear instability or because of non-linear effects exposed by the relaxation mechanism.

6.4.2 Accretion onto the Stars

As will be seen, matter flows from the disk through the gap and circulates the stars in small circumstellar disks. From those it can be accreted onto the stars. This stellar accretion is accounted for by removing some mass from a region close to the stars. This removal is performed on both grids dependent on the star positions. The removed mass is not added to the dynamical mass of the stars, but is just monitored.

The region R out of which matter is removed is given as a fraction (0.2) of the Roche lobe size for the wide systems and as the stellar cores for the tight spectroscopic systems. The accretion process typically involves only a few grid cells on each side of the stars for the wide systems and the whole resolved stellar core (up to 11x11 grid-cells) for the tight systems, making it a locally confined process.

Two methods for deciding whether a grid cell is targeted for accreting a part of its mass have been implemented. Accreting a constant fraction f per time of the mass exceeding a defined minimum density $\Sigma_{\rm min}$ leads to a very smooth accretion process. The accreted mass per iteration is computed by

$$M_{\rm acc} = \sum_{i,j \in R} \min(1.0, f\Delta t) \max(0.0, \Sigma_{ij} - \Sigma_{\rm min}) \Delta {\rm Vol}_{ij} \,. \tag{6.17}$$

Limiting the density in the accretion region R to an arbitrary value $\Sigma_{\rm max}$ results in a more realistic structure of the circumstellar disks and to more correct spectral energy distributions of the circumstellar disks:

$$M_{\rm acc} = \sum_{i,j \in R} \max(0.0, \Sigma_{ij} - \Sigma_{\rm max})\, \Delta {\rm Vol}_{ij} \,. \tag{6.18}$$

This accretion process is not smooth, but consists of accretion events because exceeding the maximum density is a discrete process and not distributed evenly over time. Nevertheless the time-averaged accretion rates are the same for both methods.

The inferred accretion rates in any event compare favorably with the average disk accretion rate, as estimated from the radial mass fluxes through different disk surfaces, and the evolution of the total mass found in the circumstellar disks.

6.4.3 Initial and Boundary Conditions

We start with an axisymmetric circumbinary disk around the binary system with an initial gap around the center of mass of the two stars, given by an approximate gap radius $r_{\rm gap}$ and a gap function

$$f_{\rm gap} := \left(e^{-\dfrac{r - r_{\rm gap}}{0.1 r_{\rm gap}}} + 1 \right)^{-1} . \tag{6.19}$$

Figure 32 shows a typical surface density at $t = 0$. The gap radius is determined by simulations including the gap formation process. This results in gap radii about three to four times the binary separation radius consistent with results from Artymowicz & Lubow (1994).

The initial density distribution is proportional to r^{-d}, where d is either set to 0.5 which is the equilibrium density distribution for an accretion disk around a central star having a constant kinematic viscosity, or given by a fit of the generated spectral energy distribution to the one observed by Mathieu et al. (1997) or Alencar et al. (2003). The initial temperature distribution is treated similarly but is relaxed to a midplane temperature satisfying eq. (6.13).

The stellar temperatures and radii specify the flux observed, and that transfered into the disk. These parameters cannot be fitted independently but need to be fixed for the irradiation of the disk. This is why we use stellar radii as suggested by Mathieu et al. (1997) and Alencar et al. (2003) and fit the effective temperatures T_* according to the observed spectral energy distributions by assuming simple blackbody spectra. Direct matching of the stellar spectra is a good approximation as can be seen in fig. 31, where we display the AK Sco disk with the binary stars projected using the estimated inclination of 63°. This demonstrates that occultation of the stars by the circumbinary disk is not to be expected.

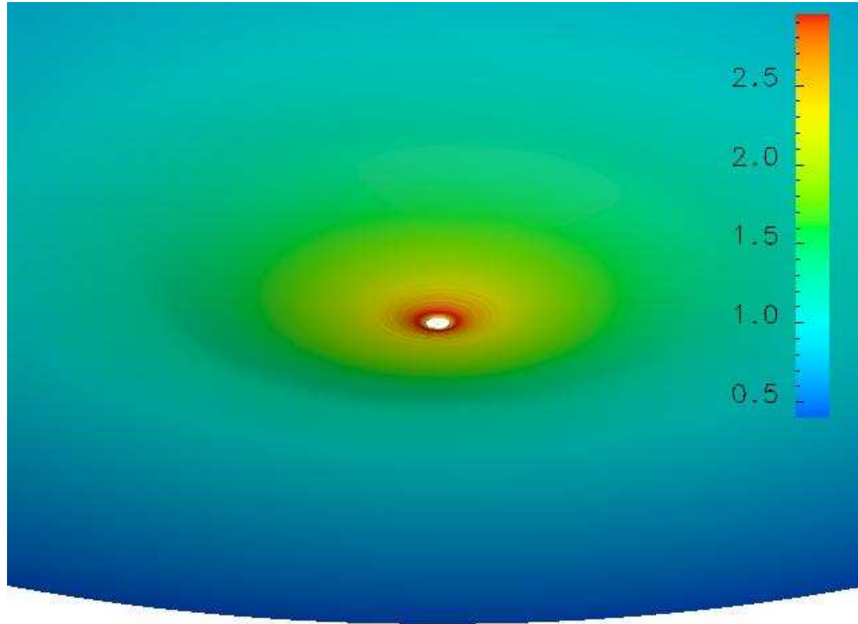


Figure 31: Visible setup of the AK Sco disk system using the estimated inclination of 63°. The disk shape is modeled according to pressure scale height, gray scale coding is logarithmic surface density in g/cm^2. The disk extent is 12 AU.

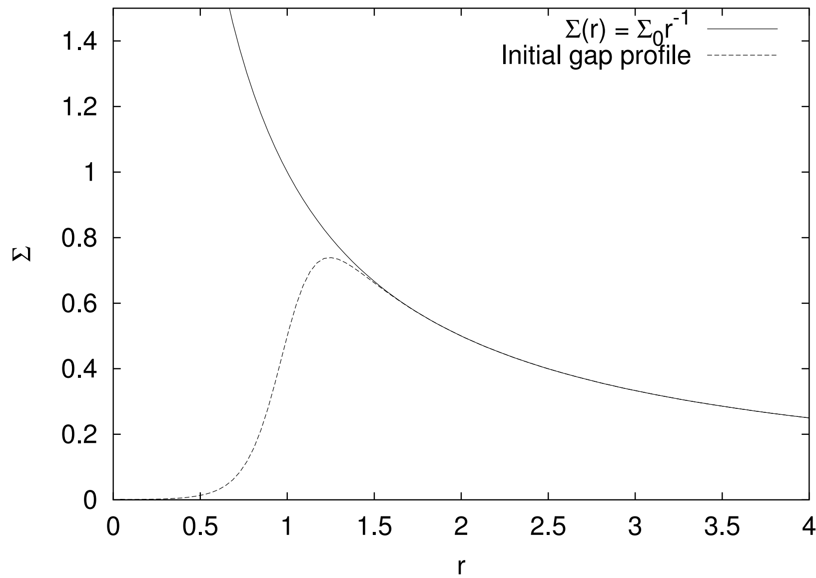


Figure 32: Initial gap profile composed out of $\Sigma_0 r^{-1}$ and the gap function eq. (6.19). The figure shows a gap at $r_{\rm gap} = 1$ with $\Sigma_0 = 1$.

The gravitational potential in eq. (6.4) is phased-in smoothly from an initial central potential for the total mass, $M_1 + M_2$, during an initialization phase lasting typically ten orbital periods. This allows the system to adjust smoothly from the semi-stationary setup of a single central star to the binary star situation.

For the boundary conditions, periodicity is imposed at $\varphi = 0$ and $\varphi = 2\pi$. We set outflowing boundary conditions at the outer radial border $r_{\rm out}$, i.e. the radial velocity gradient at $R_{\rm max}$ is zero, and the radial velocity is truncated to positive values. Angular velocity there equals the unperturbed initial Keplerian value.

For single grid calculations with a polar grid we use zero gradient boundary conditions for both velocity components and the density at the inner border.

The initial configuration is evolved for several orbital periods, so the circumstellar environment can form and the whole system settles into a quasi-periodic state.

6.4.4 Dual-Grid Technique

The *Dual-Grid* technique employed here combines two desirable features of thin disk computations: a) An ideally suited coordinate system (r, φ) in the outer bulk parts of the disk which ensures conservation of angular momentum, and b) Coverage of the region near the central binary star allowing the determination of the mass flow onto and between the stars.

In the main part the domain is covered by a cylindrical grid. On this (outer) grid the equations are formulated in polar coordinates, and the solution technique guarantees exact conservation of global angular momentum. The conservation property is absolutely crucial in obtaining physically reliable results for long term disk evolution (Kley 1998).

However, such a grid cannot be extended to the very center and leaves a hole in the middle. For typical computations this does not pose serious problems, but in this case we are dealing with eccentric binary systems in the middle. Hence, the stars or their circumstellar disks may easily cross the inner grid boundary causing strong irregularities. In any case a central hole does not allow for an accurate determination of the individual mass accretion rates onto or a possible overflow between the stars. Therefore, using a Cartesian grid centered on the origin enables an accurate calculation of the mass flows in the vicinity of the stars. Applying this technique, a high local resolution can be obtained in the center as well. An example of such a grid structure is displayed in fig. 33.

Related numerical schemes using a nested grid technique, though not in different coordinate systems, have been applied in astrophysical simulations by a number of authors (eg. Ruffert 1992; Yorke et al. 1993).

The necessary equations (in Cartesian and cylindrical coordinates) are then integrated, *independently* and *in parallel*, on the two grids, and after each time step the necessary information in the overlap region has to be exchanged. Restrictions are imposed on the time step for stability reasons, the Courant-Friedrichs-Lewy (CFL) condition must be fulfilled during each integration, on each grid.

Test calculations using a single Cartesian grid covering the whole domain have shown that the non-conservation of angular momentum yields serious deviations from known analytical results in particular for longer integration times.

As the grids possess a different geometry, exact conservation of momenta and energy cannot be guaranteed in the conversion/interpolation process which uses linear interpolation to exchange density, temperature and velocity fields. Exchanging density, internal

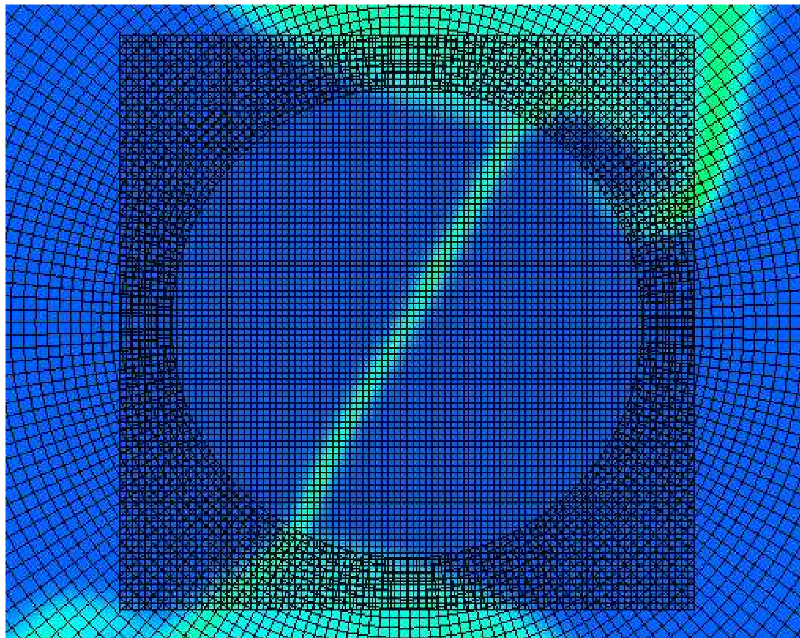


Figure 33: Grid system of a typical calculation. Shown is the grid structure of the Cartesian and cylindrical grid at the center of the computational domain. Circumstellar disks of the system can be seen at the top and bottom edges of the picture.

energy and momentum has been also implemented and leads to more correct conservation of momenta and energy. However, the advantage of a complete coverage of the physical space in between the two stars clearly outweighs this disadvantage.

We ran several tests to check the accuracy and numerical stability of the grid overlap region, including an infalling shock and a Sedov like explosion. For these tests no viscosity nor radiative effects are applied. All tests show that the grid overlap technique is sufficient for handling problems with radial symmetric features such as accretion disks.

The shock experiment initial setup is a step function of density and temperature with the floor starting at r_0 (about 10 grid cells outside of the inner boundary of the cylindrical grid) and covering the whole Cartesian grid:

$$\Sigma(r) = \Sigma_{\rm min} + \Sigma_0\, \Theta(r - r_0)\,. \tag{6.20}$$

In fig. 34 you can see the grid setup and the shock in the initial infalling phase just after crossing the grid boundaries. The shock front remains radially symmetric and no reflections are visible. After the shock compresses into a single point it switches to the outflowing phase. You can see the shock front propagating outwards just after crossing the grid boundaries in fig. 35. Apart from unavoidable grid effects near the center the shock front remains nicely circular and again neither disturbances nor reflections are visible at the grid interface.

We additionally have performed Smoothed Particle Hydrodynamics (SPH) test calculations which cover only the inner part of the simulation domain to corroborate the results of the Dual-Grid technique. The initial setup consists of two circumstellar disks in Keplerian

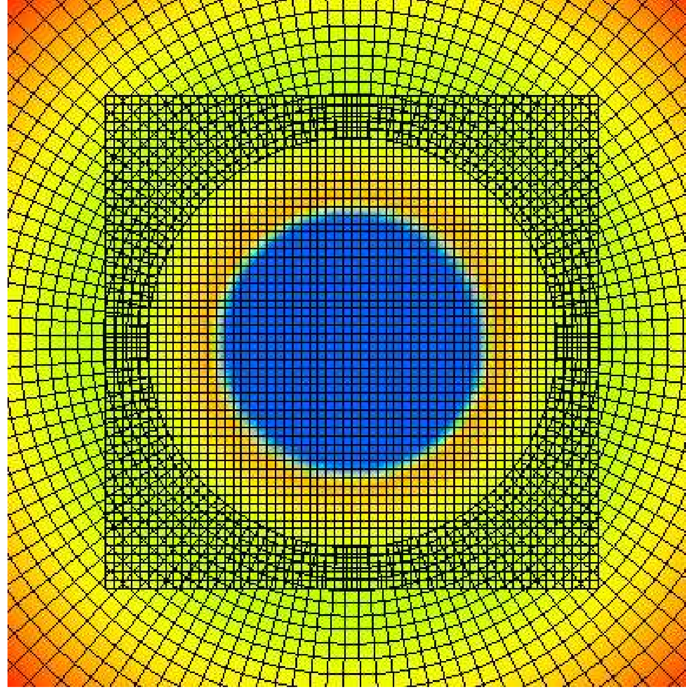


Figure 34: Radial shock in first (infalling) phase right after crossing the grid boundaries. Surface density is color coded, the grid structure is shown.

motion around the individual stars in apastron. Similar to the grid based calculations we evolve this system for a couple of orbital periods to reach a quasi-stationary state. For detailed descriptions of the SPH method see, e.g., Benz (1990) and Monaghan (1992). The SPH simulations include only radiative cooling and viscous heating and not the irradiation from the stars, as these calculations are primarily intended to support the results obtained by the Dual-Grid technique.

Simulations are set up to initially contain two circumstellar disks around the individual stars with a mass of $10^{-8}\,\mathrm{M}_\odot$ and a disk profile according to $\Sigma \propto r^{-3/2}$. This system is then evolved for a few orbital periods to truncate the disks to the right size and go into a quasi-stationary equilibrium. Dual-Grid calculations have been performed on a 180×247 sized $r - \varphi$ grid and a 101×101 sized Cartesian grid, while the SPH calculations instrumented 150 000 particles.

Fig. 36 shows the structure of the disks for Dual-Grid and SPH calculations after the same evolution time. In both simulations one can see similar circumstellar material features, namely

- spiral structure inside the circumstellar material,
- spiral waves going off the circumstellar region,
- a bar-like connection between both circumstellar regions.

Thus we can conclude that with the Dual-Grid calculations we see the same features across

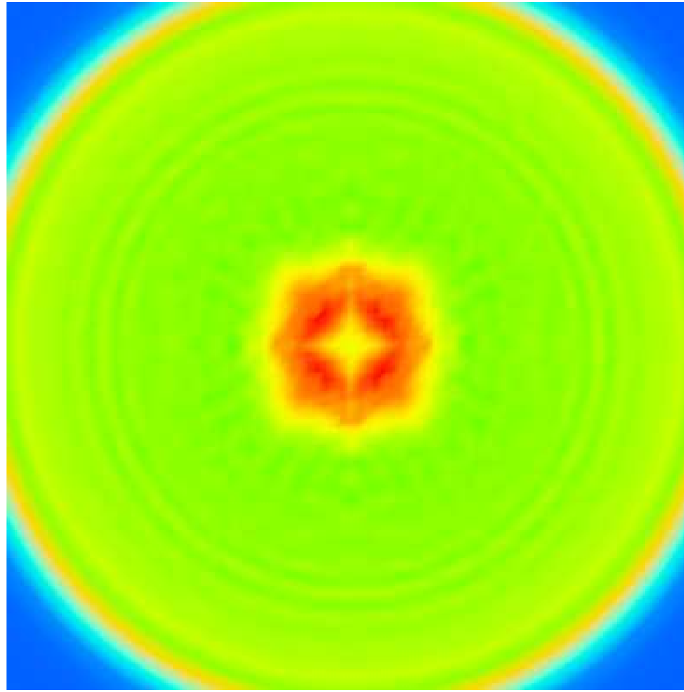


Figure 35: Radial shock in second (explosion) phase right after crossing the grid boundaries again. Surface density is color coded, the imaged domain is the same as in fig. 34.

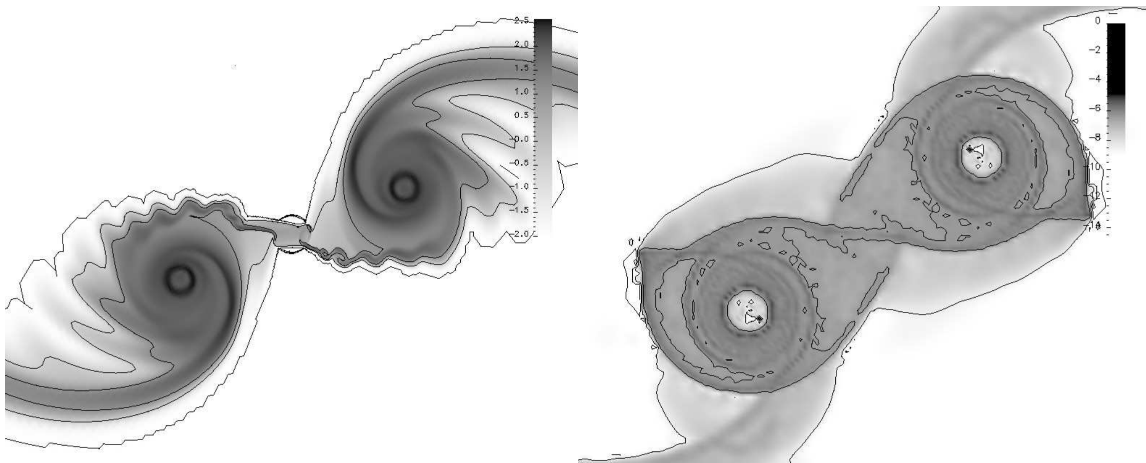


Figure 36: Circumstellar disks logarithmic density structure for Dual-Grid simulation (left) and SPH simulation (right) while approaching periastron. The binary separation is about 0.1 AU.

P	2074 yr
e	0.13
q	0.63
i	42°
a	190 AU
T_*	4000 K
R_*	2.6 $R_\odot$
M_*	0.95 $M_\odot$
d	140 ± 10 pc
M_d	1.1 $M_\odot$
R_d	2100 AU

Table 12: Parameters for UY Aur (Close et al. 1998).

P	490 yr
e	0.25
q	0.77
i	43° ± 5°
a	67 AU
T_*	3800 K
R_*	2.8 $R_\odot$
M_*	0.65 $M_\odot$
d	150 pc
M_d	0.15 $M_\odot$
R_d	600 AU

Table 13: Parameters for GG Tau (Dutrey et al. 1994; Guilloteau et al. 1999).

the origin as with the SPH calculations. The Dual-Grid technique is applicable to simulation of circumbinary systems including the origin and the inner circumstellar regions.

6.5 Systems

We looked at two different types of binary systems, namely the two wide optical systems GG Tau and UY Aur and the two close spectroscopic systems DQ Tau and AK Sco.

GG Tau (Dutrey et al. 1994; Guilloteau et al. 1999) and UY Aur (Close et al. 1998; Duvert et al. 1998) are the only two well known systems where the binary and the disk have been imaged directly. These systems have a wide separation of tens to over 100 AU, a mass ratio different from unity. In tables 13 and 12 the system parameters used for the simulations are shown.

The two close spectroscopic binary systems with circumbinary disks, DQ Tau and AK Sco, are both of T Tauri type. The physical parameters for the systems are taken from Alencar et al. (2003) for AK Sco and Mathieu et al. (1997) for DQ Tau. The relevant information for the simulations has been summarized in tables 14 and 15. Both stars are assumed to have the same effective temperature T_* and the same radius R_*.

In addition to these real systems we looked at a test system for testing numerical stability and accuracy. This system consists of an equal mass, non-eccentric binary with separation of 1 AU.

6.6 Results, General

Grid based calculations have been performed on two different regions of the whole system. First, high resolution (110×204) calculations extending only over the circumbinary disk (0.4–12 AU for AK Sco and 0.3–80 AU for DQ Tau) have been performed to evolve these regions in higher resolution and for a long time. Second, high resolution (180×247 for the $r - \varphi$ grid, 101×101 for the Cartesian grid) calculations extending over the circumstellar disks and up to the very inner region of the circumbinary disk (up to 1.2 AU for AK Sco and

P	$15^d.8043 \pm 0^d.0024$
e	0.556 ± 0.018
i	$23°$
$a \sin i$	$0.024 \pm 0.009\,\mathrm{AU}$
$M_\mathrm{A} \sin^3 i$	$0.033 \pm 0.002\,\mathrm{M}_\odot$
$M_\mathrm{B} \sin^3 i$	$0.033 \pm 0.002\,\mathrm{M}_\odot$
q	1.0 ± 0.03
T_*	$4000\,\mathrm{K}$
R_*	$1.785\,\mathrm{R}_\odot$
d	$140\,\mathrm{pc}$
M_d	$0.002 - 0.02\,\mathrm{M}_\odot$
R_d	$50\,\mathrm{AU}$
A_v	2.0

Table 14: Parameters for DQ Tau (Mathieu et al. 1997).

P	$13^d.609453 \pm 0^d.000026$
e	0.4712 ± 0.0020
i	$63°$
$a \sin i$	$0.14318 \pm 0.00005\,\mathrm{AU}$
$M_\mathrm{A} \sin^3 i$	$1.064 \pm 0.007\,\mathrm{M}_\odot$
$M_\mathrm{B} \sin^3 i$	$1.050 \pm 0.007\,\mathrm{M}_\odot$
$q = M_\mathrm{B}/M_\mathrm{A}$	0.987 ± 0.005
T_*	$6500 \pm 100\,\mathrm{K}$
R_*	$1.59 \pm 0.35\,\mathrm{R}_\odot$
d	$152\,\mathrm{pc}$
M_d	$0.005\,\mathrm{M}_\odot$
R_d	$40\,\mathrm{AU}$
A_v	0.5 ± 0.1

Table 15: Parameters for AK Sco (Alencar et al. 2003).

up to 1.0 AU for DQ Tau) have been run to allow comparisons with the SPH calculations. Finally, low resolution (128×91, 37×37) simulations of the whole system using the Dual-Grid technique have been performed to show that indeed these regions decouple in the timescales we are interested in.

The overall azimuthally averaged circumbinary disk shape does not change significantly on timescales of the orbital period of the binary system. Induced by the gravitational torques of the central binary a nice a spiral structure develops in the inner regions of the disk as can be seen in fig. 37.

Also one can observe periodic formation of spiral arms from the edge of the circumbinary disk going down to the circumstellar environment (fig. 38) connected to the binary orbital period. The quasi-periodic state of the *circumstellar* environment of the eccentric system is characterized by the following phases:

1. Going from periastron to apastron, material is accreted to the circumstellar environment from the circumbinary disk and the remaining material in the gap.
2. Around apastron phase circumstellar envelopes/disks have been formed.
3. Approaching periastron, these circumstellar disks are torn off due to gravitational torques exerted on the disks by the stars, and a part of the mass is accreted onto the stars.

This leads to periodic accretion events which is both confirmed from observations (Basri et al. 1997) showing periodic brightening events and from numerical simulations.

6.7 Results, Equal Mass Binary

Before discussing the results concerning particular astronomical objects, we would like to present the details of a specific test case of an equal mass binary star on a circular orbit.

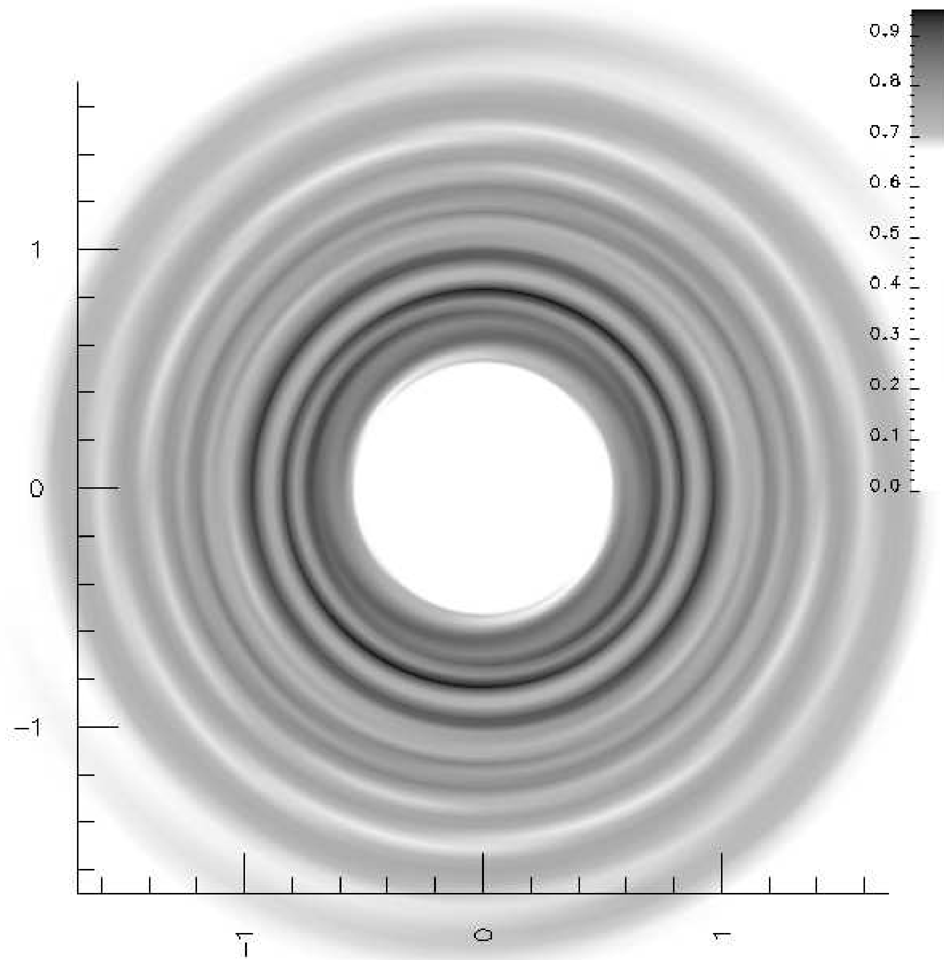


Figure 37: Linear surface density (g/cm^2) of the inner circumbinary disk of the DQ Tau system after 50 orbital periods. Scaling is in AU. The spiral structure in the circumbinary disk induced by the binary is clearly seen.

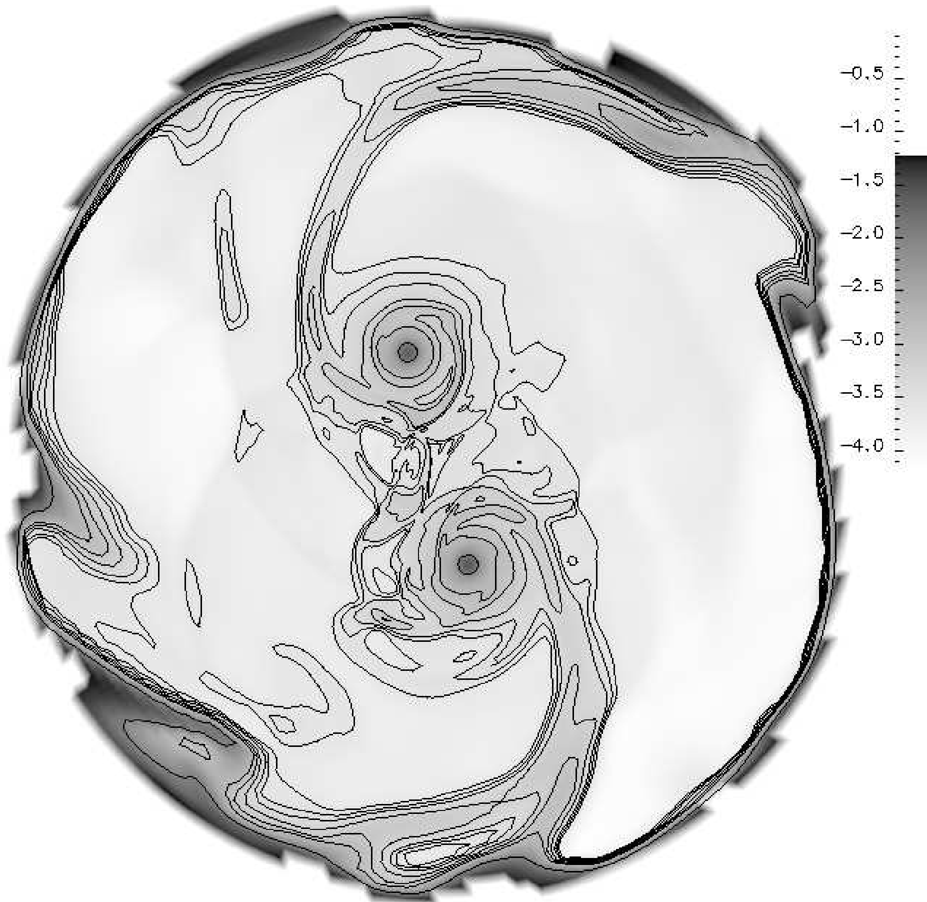


Figure 38: The spiral arms onto the circumstellar environment for the DQ Tau system after 50 orbital periods. The gray scale follows logarithmic surface density annotated with iso-lines. Plotted is the region inside the circumbinary disk gap which is located at 0.4 AU. The binary is in apastron phase.

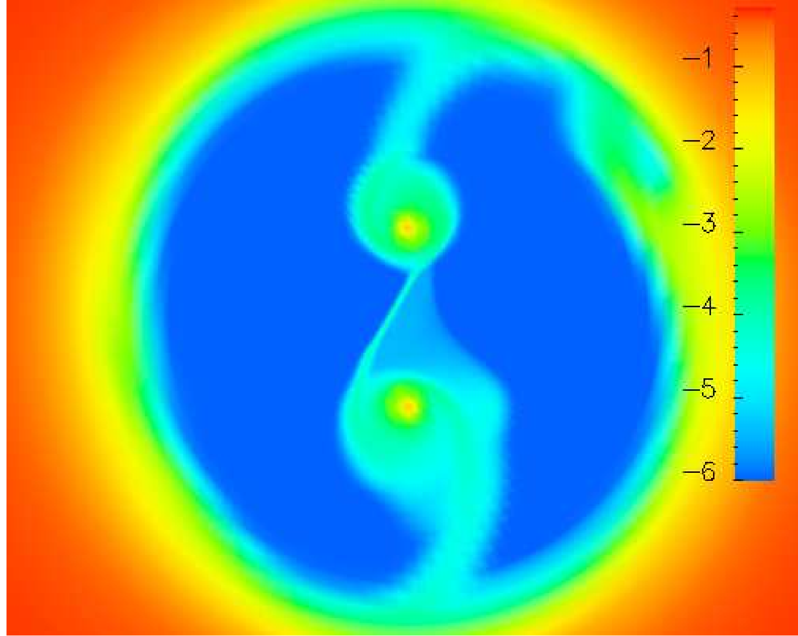


Figure 39: Evolution of an equal mass binary after 90 orbital periods. Color encoding is logarithmic surface density. Both the inner part of the circumbinary disk and the circumstellar disks are visible.

Such a system is well suited to check important properties (symmetries, stability) of the numerical method used. Also general properties of accretion disks in binary systems can be derived from such a test case.

The equal mass binary on a circular orbit serves as a testing platform for numerical stability and accuracy. In a coordinate frame corotating with the binary one expects after some transient time a quasi-stationary situation, which is symmetric with respect to the line connecting the stars. As the viscosity is solved implicitly by solving a large matrix system iteratively, exact symmetry cannot be preserved numerically. Also usual density contrasts are of the order of 10^3 for numerical floor to circumstellar disks and from circumstellar disks to the circumbinary disk. Hence, this test serves as an excellent example for estimating the ability to preserve symmetry over long time scales.

In fig. 39 you can see a circumbinary disk after 90 orbits evolution time in almost perfect quasi-stationary state. The deviation from line-symmetry is small. Only with very high resolution simulations we observe spiral features in the outer regions of the disk, while spiral arms originating from the inner gap of the circumbinary disk flowing onto the circumstellar disks can be observed even for low resolutions. Those spiral arms feeding the circumstellar disks remain stable for the whole period of a non-eccentric binary.

The accretion rates for both accretion mechanisms onto the two stars are nearly identical and do not show any significant variations for different numerical parameters. Thus, our implementation of the accretion mechanism is robust.

Changing model parameters such as Σ_0, gap width, or disk size we obtain characteristic variations of the resulting spectral energy distribution of an evolved disk. Figures 40-42

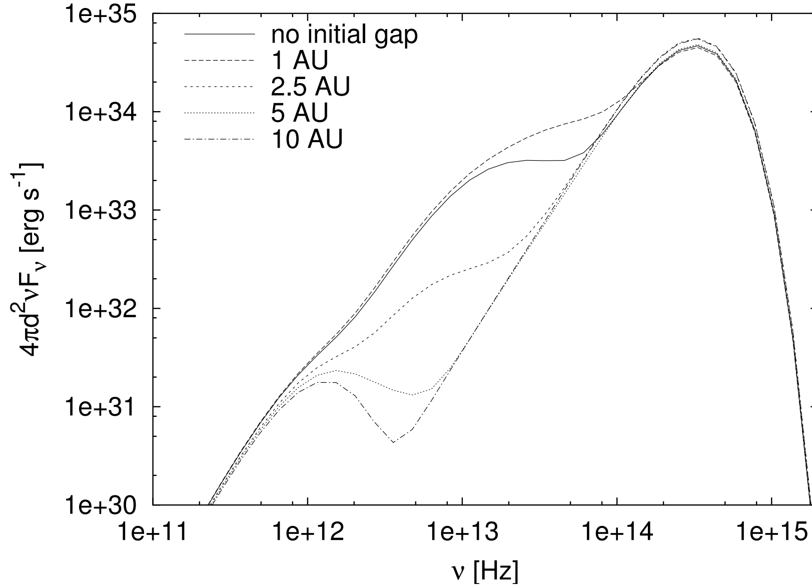


Figure 40: Dependency of the gap width on the SED.

show dependencies of these parameters on the spectral energy distribution that are used to fit parameters of the spectroscopic systems to match observed spectra.

6.8 Results, Wide Systems

There are only two well known systems where the binary and the disk have been imaged directly, GG Tau (Dutrey et al. 1994; Guilloteau et al. 1999) and UY Aur (Close et al. 1998; Duvert et al. 1998). These systems have a much wider separation of tens to over 100 AU and a mass ratio different from unity and show consequently a quite different behavior. In tables 13 and 12 the system parameters used for the simulations are shown.

From our simulations we derive averaged accretion rates for both UY Aur and GG Tau that are comparable with the observed quantities in table 16. In the case of UY Aur we get $0.5 \cdot 10^{-7}\,\mathrm{M_{\odot}yr^{-1}}$ for the primary and $0.25 \cdot 10^{-6}\,\mathrm{M_{\odot}yr^{-1}}$ for the secondary, respectively. Simulations of GG Tau lead to accretion rates of $0.11 \cdot 10^{-8}\,\mathrm{M_{\odot}yr^{-1}}$ for the primary and $0.33 \cdot 10^{-9}\,\mathrm{M_{\odot}yr^{-1}}$ for the secondary. As expected from the difference of the disk masses ($0.15\,\mathrm{M_{\odot}}$ vs. $1.1\,\mathrm{M_{\odot}}$) the accretion rate for GG Tau is one order of magnitude less than the one for UY Aur.

Accretion is reinforced at periastron phases as for the spectroscopic systems, but events are an order of magnitude lower for GG Tau and nearly vanish for UY Aur due to the lower eccentricity of the systems. Also both systems show more phase dependent accretion for the secondary than for the primary.

In figs. 43 and 44 you can see the inner parts of the simulated circumbinary disks. The circumstellar disks remain intact over the whole period. It has not been possible to generate meaningful spectral energy distributions of the disks as they are too cold for these wide systems. Also observational data is lacking for these systems.

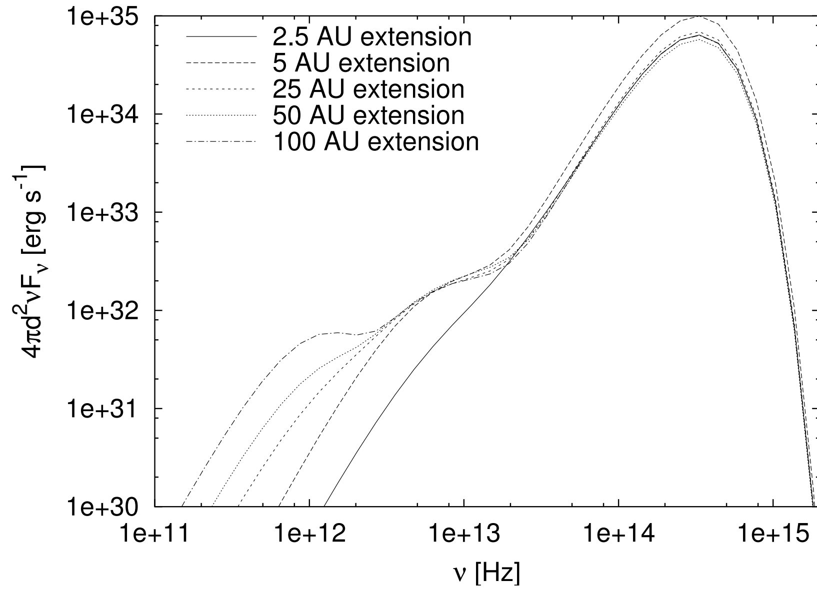


Figure 41: Dependency of the disk size on the SED.

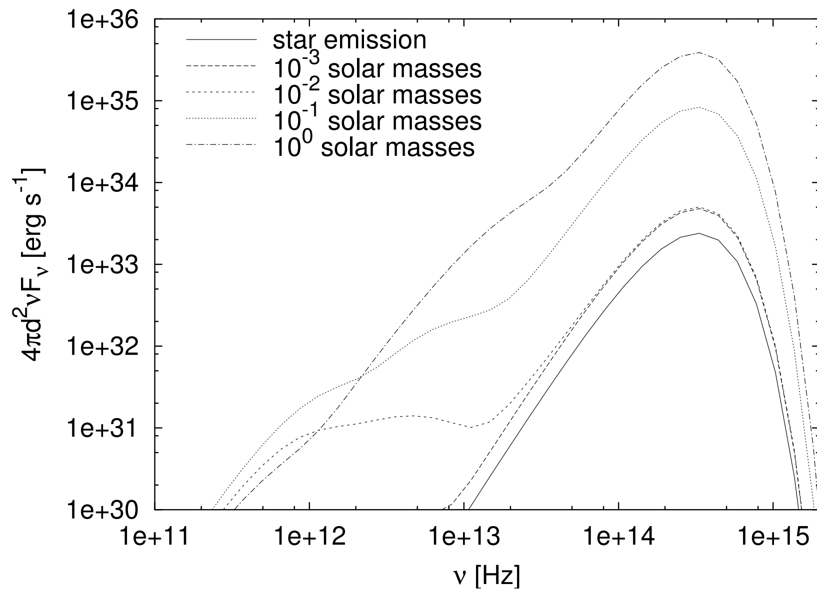


Figure 42: Dependency of Σ_0 on the SED.

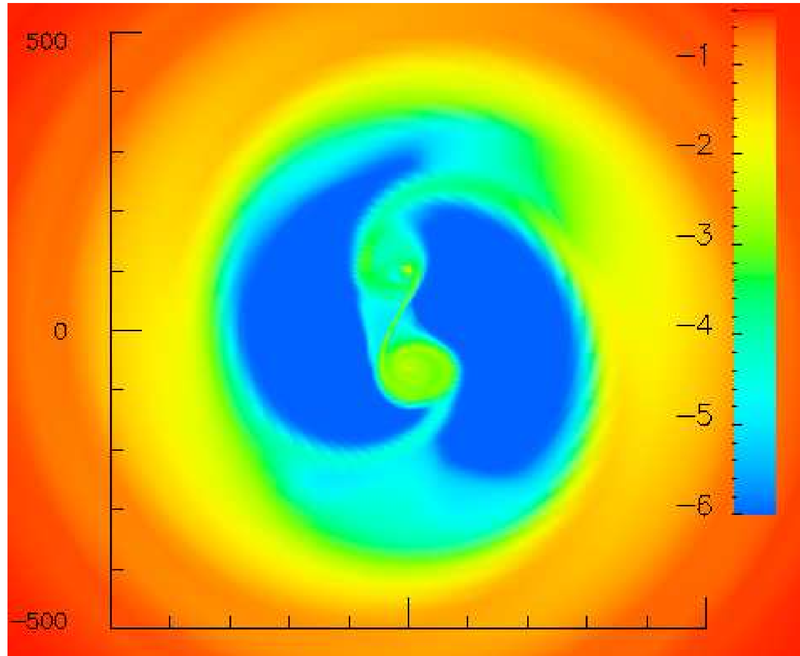


Figure 43: UY Aur circumbinary disk after 20.5 orbital periods. Color coding is $\log(\Sigma)$, the length scales are in AU.

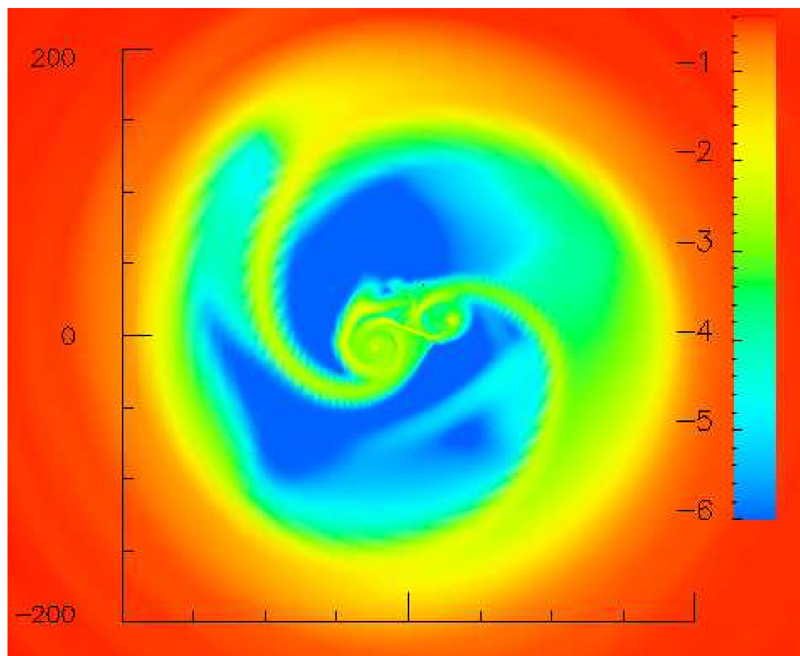


Figure 44: GG Tau circumbinary disk after 56.4 orbital periods. Color coding is $\log(\Sigma)$, the length scales are in AU.

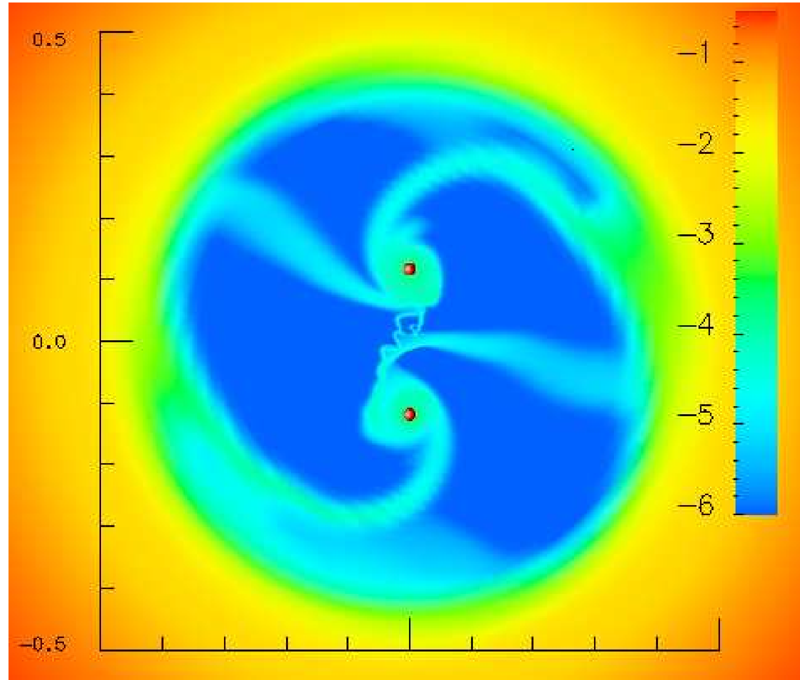


Figure 45: AK Sco circumbinary disk after 82.0 orbital periods in apastron. Color coding is $\log(\Sigma)$, the size of the stars reflects the actual stellar radii, the length scales are in AU

6.9 Results, Spectroscopic Systems

Hereafter we discuss models for the two spectroscopic young binary stars DQ Tau and AK Sco. Images showing the disk configuration in apastron and periastron phase are shown in figs. 45 and 46.

Generally, for eccentric systems we observe spiral features in the outer circumbinary disk only for very high resolution simulations, while again spiral arms feed the circumstellar disks even with low resolutions. Opposed to the non-eccentric systems the spiral arms develop during apastron phase and are torn off together with the outer parts of the circumstellar disks after periastron phase for high eccentric systems. This tearing off is caused by raising centrifugal forces acting on the circumstellar material.

6.9.1 Accretion

For both systems we compared the accretion rates resulting from the simulation with accretion rates derived from of observations as done by Hartigan et al. (1995) and Gullbring et al. (1998) (see table 16). As no data has been available for AK Sco, a comparison with observational data has not been possible, but as both spectroscopic systems have similar system parameters, accretion rates of the same order of magnitude can be expected.

As can be seen from figs. 47 and 48 the accretion process happens synchronized with periastron phase of the binaries. This is in agreement with numerical simulations from Artymowicz & Lubow (1996) and Rozyczka & Laughlin (1997) and with observations which

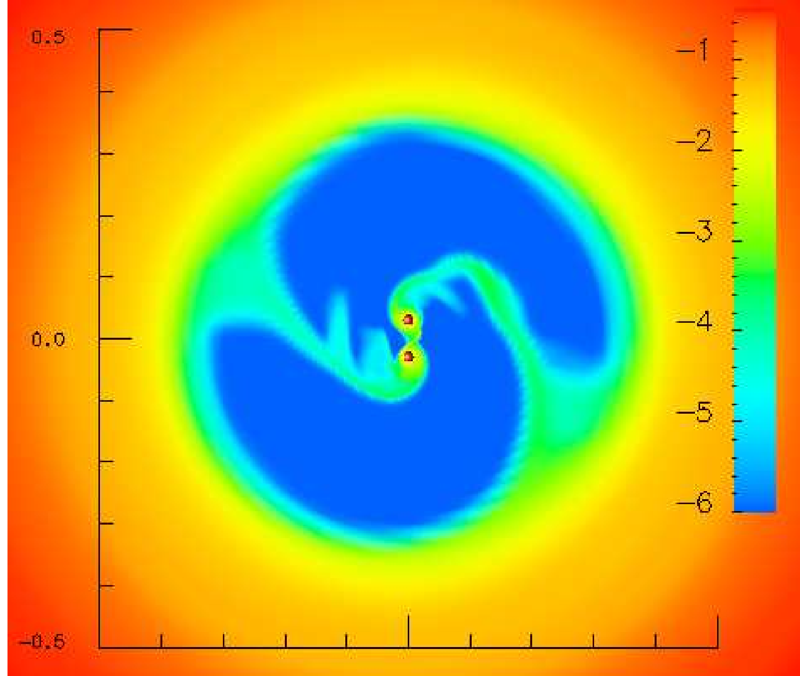


Figure 46: DQ Tau circumbinary disk after 85.5 orbital periods in periastron. Color coding is $\log(\Sigma)$, the size of the stars reflects the actual stellar radii, the length scales are in AU.

	Hartigan	Gullbring	simulation
DQ Tau	$0.50 \cdot 10^{-7}$	$0.60 \cdot 10^{-9}$	$0.62 \cdot 10^{-8}$
AK Sco	n.a.	n.a.	$0.83 \cdot 10^{-8}$
GG Tau	$0.20 \cdot 10^{-6}$	$0.175 \cdot 10^{-7}$	$0.11 \cdot 10^{-8}$
UY Aur	$0.25 \cdot 10^{-6}$	$0.656 \cdot 10^{-7}$	$0.5 \cdot 10^{-7}$

Table 16: Accretion rates in $M_{\odot}yr^{-1}$ derived from observational data from Hartigan et al. (1995) and Gullbring et al. (1998) compared to rates as resulting from our simulations. Our rates are for the primary component of the binaries, for the other data we assume it is an averaged accretion rate for both stars.

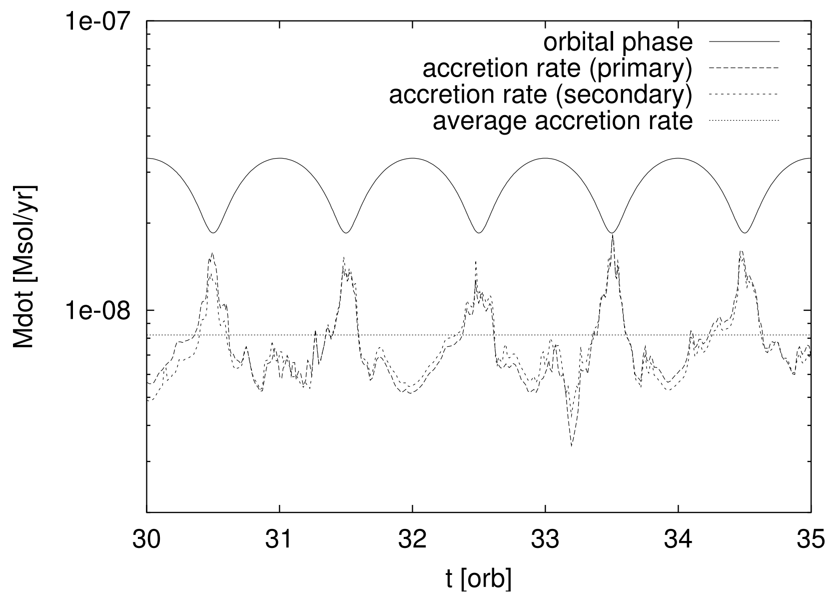


Figure 47: Time-dependent accretion rate of the AK Sco system. The orbital phase curve schematically shows the distance of the two stars.

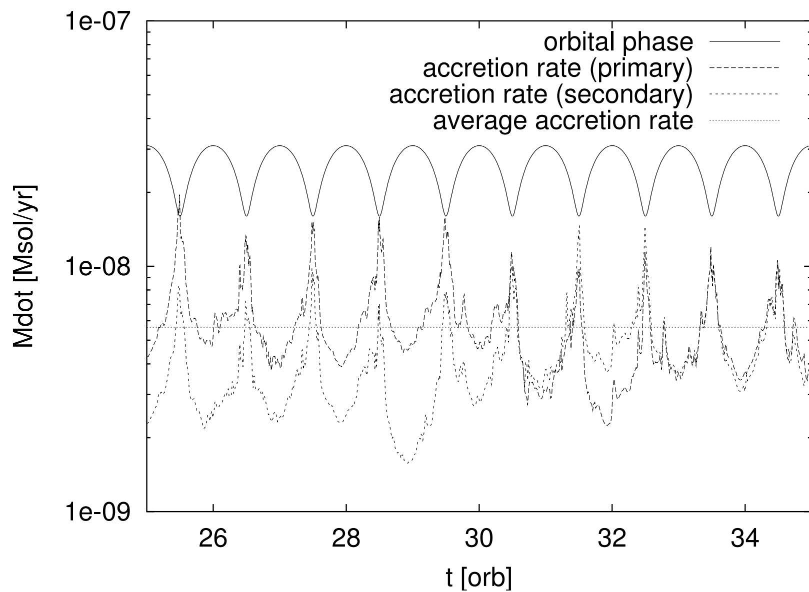


Figure 48: Time-dependent accretion rate of the DQ Tau system. The orbital phase curve schematically shows the distance of the two stars.

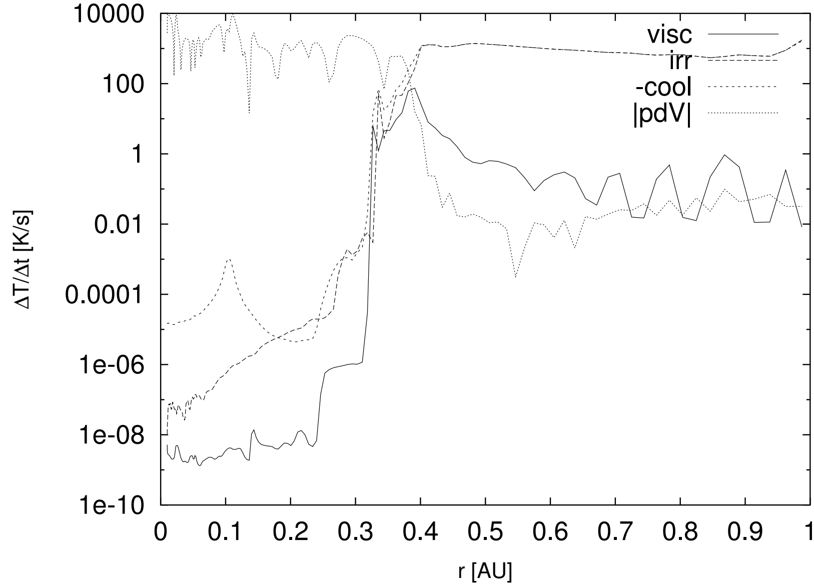


Figure 49: Contribution of the different heating/cooling terms to the rate of temperature change (DQ Tau after 50 orbital periods). Only the inner part of the system is displayed. The irradiation irr and the radiative cooling -cool match exactly beyond 0.4 AU.

show excess luminosity at these phases (Mathieu et al. 1997).

From our simulations we can derive an averaged accretion rate of $0.83 \cdot 10^{-8}\,\mathrm{M_\odot yr^{-1}}$ for the primary, $0.82 \cdot 10^{-8}\,\mathrm{M_\odot yr^{-1}}$ for the secondary of AK Sco and $0.62 \cdot 10^{-8}\,\mathrm{M_\odot yr^{-1}}$ for the primary, $0.51 \cdot 10^{-8}\,\mathrm{M_\odot yr^{-1}}$ for the secondary of DQ Tau. These accretion rates are in good agreement with the observational rates.

6.9.2 Irradiation Effects

For motivating the inclusion of irradiation effects, we first analyze the individual contribution of the different terms in the radiative balance eq. (6.9), and the pressure work $p\nabla \cdot \mathbf{u}$, to the temperature change rate. The azimuthally averaged rate of the temperature change at quasi-stationary equilibrium of the DQ Tau system is plotted in fig. 49 for the inner regions of the disk and fig. 50 for the outer regions. Here, visc denotes the rate from the viscous heating, irr the rate from the irradiation process, cool the emission contribution and pdV the rate from the pressure work. Note, that the inner edge of the disk is approximately at $r = 0.4\,\mathrm{AU}$ for this model.

As one can easily infer from fig. 50 the circumbinary disk is dominated by irradiation effects (which in fact match the cooling rates exactly), and as such qualifies as a passive disk. The effects due to viscous heating and pressure work can be neglected here.

In the inner optically thin part the pressure work is dominating due to the dynamic motion in this region caused by the gravitational torques exerted by the eccentric binary. One can also see that the emitted flux in this region is negligible compared to the flux emitted by the circumbinary disk.

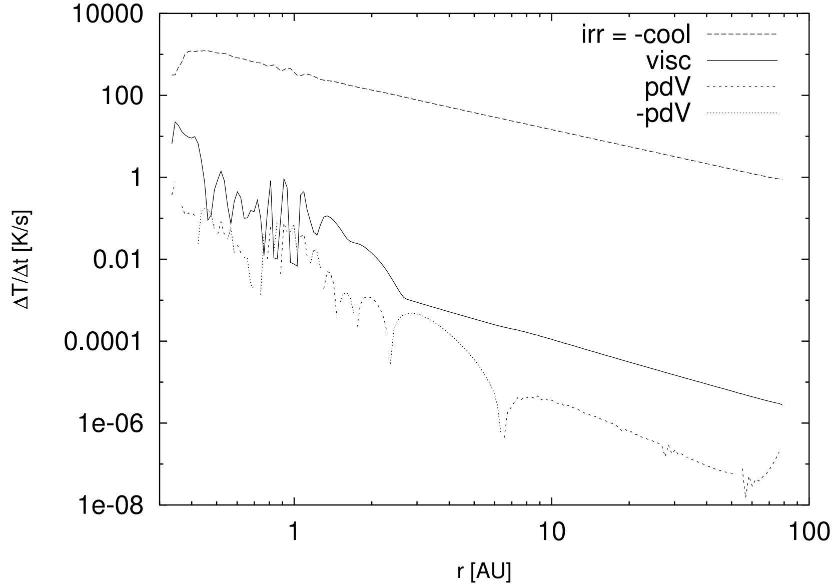


Figure 50: Contribution of the different heating/cooling terms to the rate of temperature change (DQ Tau after 50 orbital periods). Plotted is the outer circumbinary disk part of the system. The irradiation by the central stars and the radiative cooling match exactly (topmost dashed line).

6.9.3 DQ Tau Circumbinary Disk

For both the DQ Tau and the AK Sco system we shall see missing flux at around 5×10^{13}, resp. 10^{14} Hz which can be accounted to the modeling of the T Tauri type stars emission as a simple blackbody.

From full simulations including the gap formation process we infer an initial gap with a radius of 0.4 AU for the DQ Tau system. The stars combined effective blackbody temperature is fitted manually to 3500 K to match the observations. Starting from the parameters presented in table 14 and power-law fits from Mathieu et al. (1997) we adjust the initial density distribution power-law index, the disk mass and its extension to best match the observed SED after going to quasi-stationary state.

The SED of DQ Tau can be fitted best with a thin disk extending to 80 AU which has an unusually flat surface density distribution $\Sigma \propto r^{-0.25}$ and a total disk mass of $7 \times 10^{-4}\,\mathrm{M}_{\odot}$. This is less mass and a larger disk radius as obtained by Mathieu et al. (1997). After fifty orbital periods, the equilibrium surface temperature distribution of the circumbinary disk follows $T(r) = 155\,\mathrm{K}\, r^{-0.42}$ as can be seen in fig. 52.

6.9.4 AK Sco

In fig. 16 of Alencar et al. (2003) the authors propose a circumbinary disk mass of $M_\mathrm{d} = 0.02\,\mathrm{M}_{\odot}$ and a disk extend of $R_\mathrm{d} = 12\,\mathrm{AU}$ for the best fit to the observed SED. The surface density distribution is that of a minimum mass solar nebula as in the used model (Chiang & Goldreich 1997), so $\Sigma(r) \sim r^{-1.5}$. In fig. 53 the emission of a model evolved with these

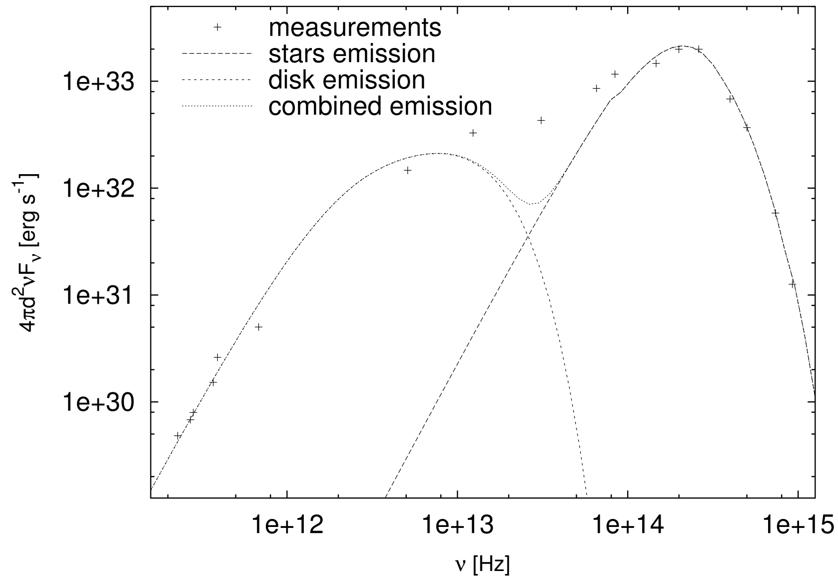


Figure 51: Computed SED of DQ Tau. Crosses show observational data taken from Mathieu et al. (1997).

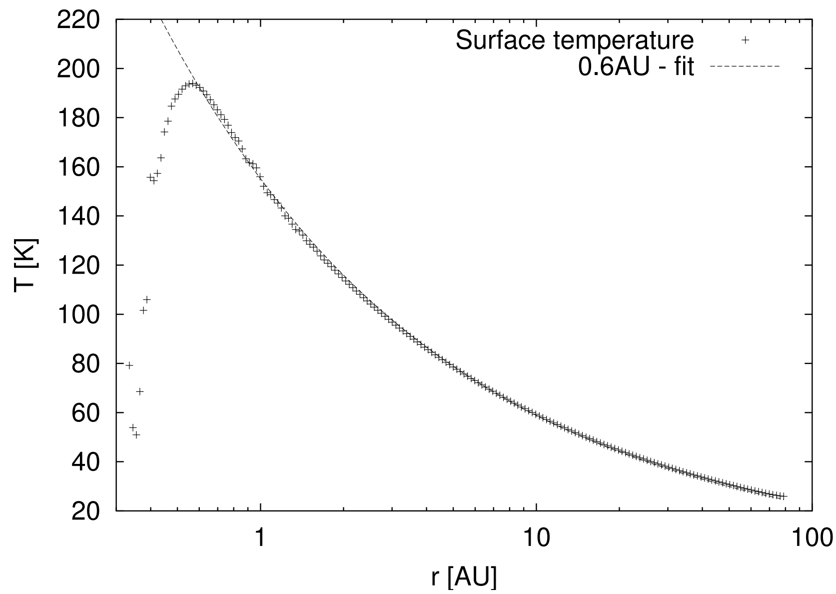


Figure 52: Azimuthally averaged surface temperature of the equilibrium DQ Tau circumbinary disk.

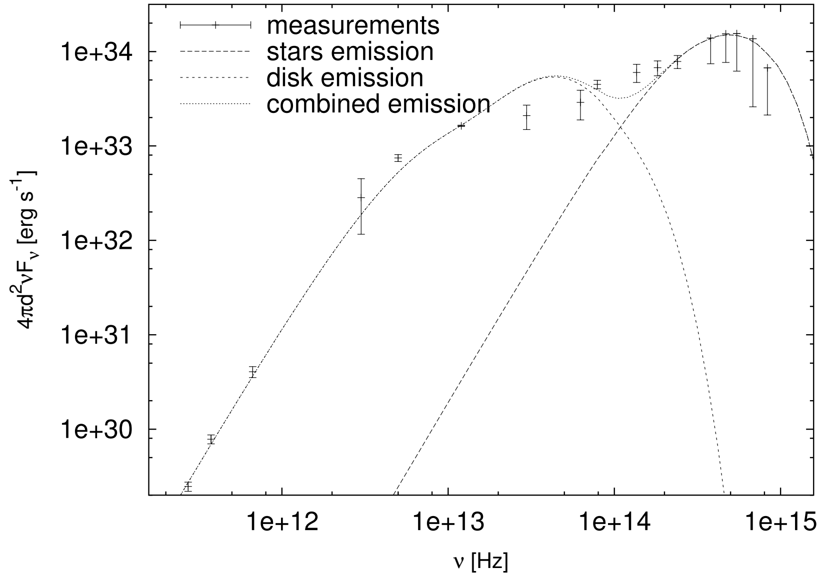


Figure 53: Computed SED of AK Sco with parameters and observational data (crosses with error-bars) taken from Alencar et al. (2003).

parameters imposed initially is displayed.

Clearly visible is the excess flux of the model in the infrared (around 5×10^{13} Hz) which we can reduce by lowering the disk mass and slightly flattening the disk density profile. Using a disk mass of $M_{\mathrm{d}} = 0.01\,\mathrm{M}_{\odot}$ and an initial density profile according to $\Sigma(r) \sim r^{-1.0}$ we obtain a spectrum as seen in fig. 54.

For the AK Sco system the stars combined effective blackbody temperature is best fitted to the observations by assuming $T_{*} = 6500\,\mathrm{K}$. In a quasi-stationary state after about fifty orbital periods of the binary the surface temperature can be fitted to the power-law $T(r) = 264\,\mathrm{K}\ r^{-0.47}$ for $r > 2\,\mathrm{AU}$ and a surprisingly steep linear behavior in the region between one and two AU ($T(r) = 617\,\mathrm{K} - 215\,\mathrm{K}\ r$) as shown in fig. 55. This is due to the different opacities in the dense region of the disk and compared to the case of DQ Tau three orders of magnitude higher maximum surface density which is related to the AK Sco disk being much smaller and more massive.

We note that our models are not able to fit the higher flux observed in the SED of both systems in the region between the disk and stellar contributions. This may be attributed to line emission features (Alencar et al. 2003) which cannot be modeled by our simulations.

6.9.5 Circumstellar Disks

For the region inside the circumbinary disk gap we see a tiny amount of warm gas and streamers feeding circumstellar disk like structures from the inner circumbinary disk edge. Due to the low mass this material does not contribute to the continuum part of the observed spectral energy distributions but rather would show up in lines and the UV part of the spectra.

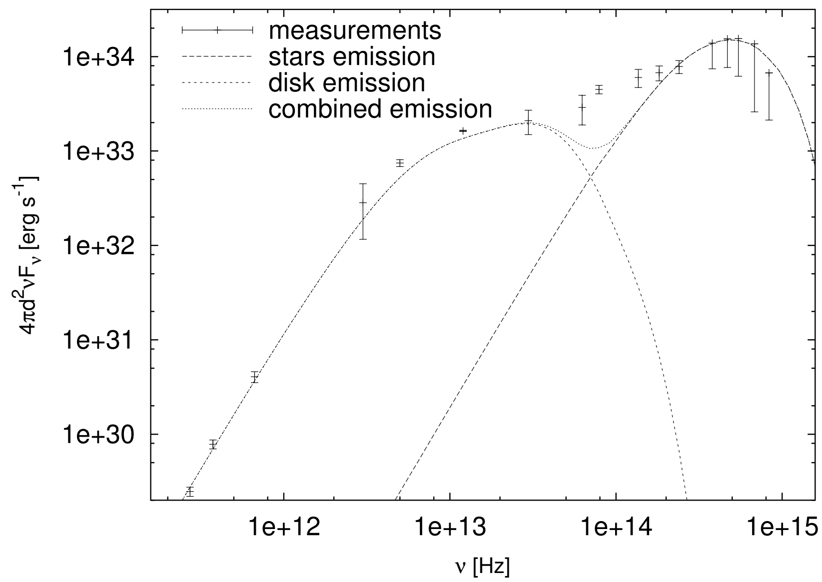


Figure 54: Computed SED of the AK Sco system with best-fit parameters. Crosses with error-bars show observational data taken from Alencar et al. (2003).

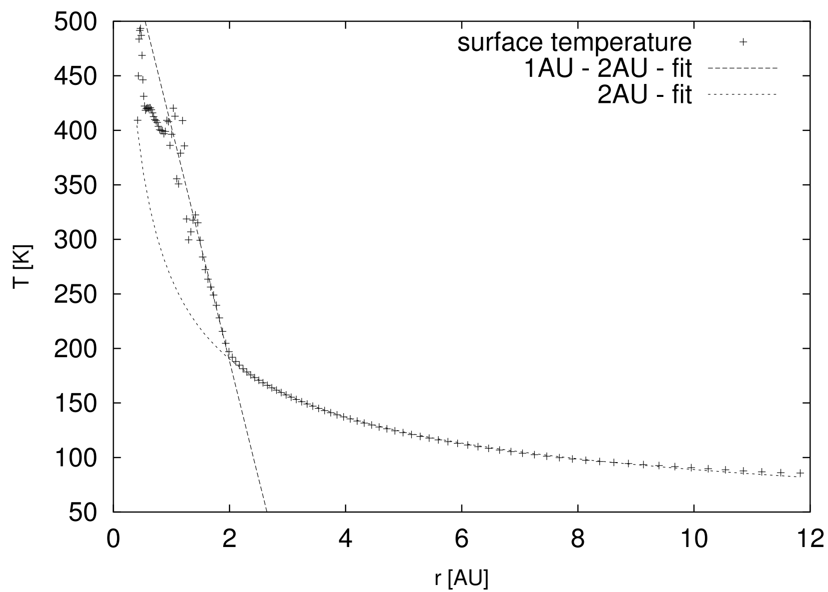


Figure 55: Azimuthally averaged surface temperature of the equilibrium AK Sco circumbinary disk.

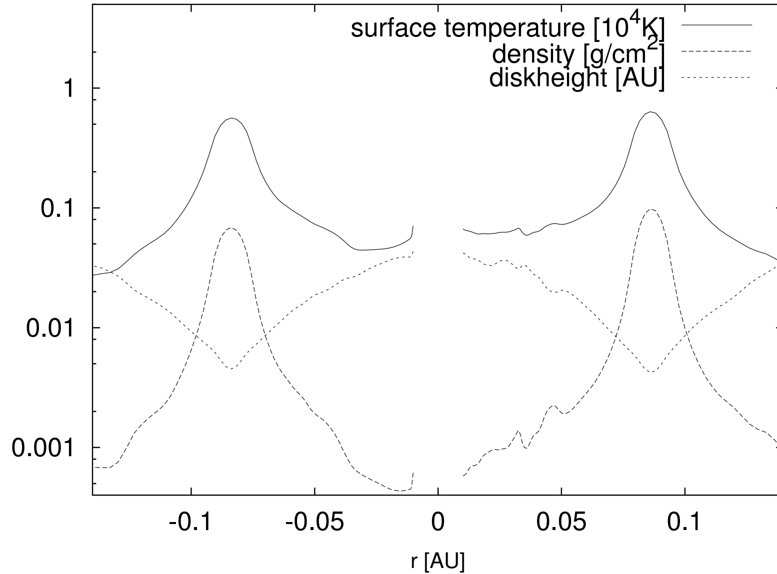


Figure 56: Profiles of the circumstellar material along the connecting line of the binary at apastron in the system of DQ Tau (stellar radii 0.008 AU).

Also, the observed periodic brightening in the light-curve from the systems originate here, which is believed to originate from periodic accretion events. These periodic events of accretion have been confirmed by Artymowicz & Lubow (1996) and Rozyczka & Laughlin (1997) through numerical simulations.

The circumstellar disk like structures around DQ Tau form through accumulation of material in the stellar gravitational potential. Without removing any material through an accretion process, disk profiles according to fig. 56 form. They cannot be identified with a classical accretion disk due to shape and size as one can see from fig. 57 which shows the circumstellar material and velocity distribution. They rather would form sort of an envelope, but this remains to be investigated in three-dimensional calculations.

6.10 Conclusion

With the present calculations we have modeled the accretion flow onto the stars within a circumbinary disk and their circumstellar environment including emitted spectra. We have done this in more detail using more realistic physics, much higher resolution and a much longer time integration than done before. The models include a self-consistent approximative vertical energy transport including irradiation from the stars using detailed opacities and equation of state. The numerical resolution we have reached by now is so fine that the individual stars in close binaries are already resolved by typically about 10×10 grid-cells for high-resolution calculations. Our main achievements may be summarized as follows:

1. Development of a new Dual-Grid technique combining two coordinate systems, which is ideally suited for modeling planar accretion disks including their central objects.

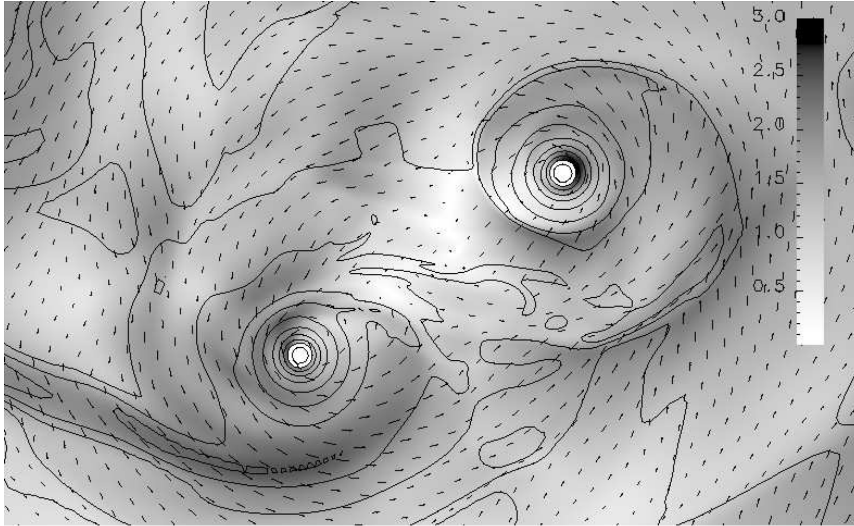


Figure 57: Circumstellar material in the DQ Tau system after 41 orbital periods, corresponding to fig. 56. Gray scale coding is velocity magnitude annotated with vector glyphs, isolines are equally spaced logarithmic surface density. The stellar cores are the white circles.

2. Performing long-time integrations of binary stars and circumbinary disks, including a detailed vertical energy balance and honoring irradiation effects.

3. For close binaries with separations of only a fraction of an AU we find a relatively large accretion rate crossing the gap of the order of $10^{-8}\mathrm{M}_{\odot}\mathrm{yr}^{-1}$.

4. For wide binaries ($a = 10 - 200\,\mathrm{AU}$) the accretion rate is comparable to those of the close binaries only because of the large disk masses of the two systems.

5. For a suitable variation of the initial density distribution $\Sigma \sim r^{-d}$, the circumbinary disk mass M_{d} and the circumbinary disk extent R_{d} the spectral energy distributions of DQ Tau and AK Sco could be fitted well to the observational data if irradiation effects are included.

For the close systems the flaring disk settles into a quasi-stationary state after around 50 orbital periods of the binary with a temperature distribution following approximately $T(r) \propto r^{-0.45}$ for larger radii in both systems. We find that irradiation plays the dominant role for the heating balance in these outer regions. In the inner part of the disk just beyond the inner gap, the additional heating due to the tidal effects is important, particularly for small disks around tight binaries, such as AK Sco.

For the present model parameter we do not find any indication for a previously observed self-shadowing instability of the disk due to the stellar irradiation (Dullemond 2000). A proper numerical treatment of the strongly non-linear cooling is required to prevent artificial numerical instabilities, which is achieved by using a time-relaxation scheme as opposed to directly solving for the equilibrium.

References

Adams, F. C., Shu, F. H., & Lada, C. J. 1988, ApJ, 326, 865

Alencar, S. H. P., Melo, C. H. F., Dullemond, C. P., et al. 2003, A&A, 409, 1037

Allen, G. e. a. 2000, in Proceedings of the First EGrid Meeting, Poznan

Artymowicz, P., Clarke, C. J., Lubow, S. H., & Pringle, J. E. 1991, ApJ, 370, L35

Artymowicz, P. & Lubow, S. H. 1994, ApJ, 421, 651

Artymowicz, P. & Lubow, S. H. 1996, ApJ, 467, L77

Balbus, S. A. & Hawley, J. F. 1991, ApJ, 376, 214

Bardeen, J. M., Thorne, K. S., & Meltzer, D. W. 1966, ApJ, 145, 505

Basri, G., Johns-Krull, C. M., & Mathieu, R. D. 1997, AJ, 114, 781

Bate, M. R. & Bonnell, I. A. 1997, MNRAS, 285, 33

Benz, W. 1990, in Numerical Modelling of Nonlinear Stellar Pulsations Problems and Prospects, 269

Caramana, E. J., Shashkov, M. J., & Whalen, P. P. 1998, Journal of Computational Physics, 144, 70

Cardelli, J. A., Clayton, G. C., & Mathis, J. S. 1989, ApJ, 345, 245

Chiang, E. I. & Goldreich, P. 1997, ApJ, 490, 368

Close, L. M., Dutrey, A., Roddier, F., et al. 1998, ApJ, 499, 883

Courant, R. & Friedrichs, K. O. 1948, Supersonic flow and shock waves (Pure and Applied Mathematics, New York: Interscience, 1948)

D'Angelo, G. 2003, Ph.D. Thesis

D'Angelo, G., Kley, W., & Henning, T. 2003, ApJ, 586, 540

Dullemond, C. P. 2000, A&A, 361, L17

Duquennoy, A. & Mayor, M. 1991, A&A, 248, 485

Dutrey, A., Guilloteau, S., & Simon, M. 1994, A&A, 286, 149

Duvert, G., Dutrey, A., Guilloteau, S., et al. 1998, A&A, 332, 867

Fryxell, B., Olson, K., Ricker, P., et al. 2000, ApJS, 131, 273

Günther, R. 2001, Dynamik und Entwicklung von zirkumbinären Scheiben (Diploma Thesis, University of Tübingen,), 64

Günther, R. & Kley, W. 2002, A&A, 387, 550

Günther, R., Schäfer, C., & Kley, W. 2004, A&A, 423, 559

Guilloteau, S., Dutrey, A., & Simon, M. 1999, A&A, 348, 570

Gullbring, E., Hartmann, L., Briceno, C., & Calvet, N. 1998, ApJ, 492, 323

Hartigan, P., Edwards, S., & Ghandour, L. 1995, ApJ, 452, 736

Hawley, J. F., Wilson, J. R., & Smarr, L. L. 1984, ApJS, 55, 211

Henshaw, W. D. 1996, in Sixth International Conference on Numerical Combustion, New Orleans

Hubeny, I. 1990, ApJ, 351, 632

Hurley, M., Roberts, P. H., & Wright, K. 1966, ApJ, 143, 535

Kippenhahn, R. & Weigert, A. 1991, Stellar Structure and Evolution, 2nd edn. (Springer)

Klahr, H. H. 1998, Ph.D. Thesis

Kley, W. 1988, PhD thesis, Ludwig - Maximilians - Universität München

Kley, W. 1989, A&A, 208, 98

Kley, W. 1998, A&A, 338, L37

Kley, W. 1999, MNRAS, 303, 696

Kley, W. 2000, MNRAS, 313, L47

Ledoux, P. & Walraven, T. 1958, Handbuch der Physik, 51, 353

Lin, D. N. C. & Papaloizou, J. 1985, in Protostars and Planets II, 981

Mathieu, R. D., Ghez, A. M., Jensen, E. L. N., & Simon, M. 2000, Protostars and Planets IV, 703

Mathieu, R. D., Stassun, K., Basri, G., et al. 1997, AJ, 113, 1841

Mihalas, D. & Mihalas, B. W. 1984, Foundations of radiation hydrodynamics (New York, Oxford University Press, 1984, 731 p.)

Monaghan, J. J. 1992, Annual Review Astronomy Astrophysics, 30, 543

MPI Forum. 1995, MPI: A Message-Passing Interface Standard, MPI Forum

MPI Forum. 1997, MPI-2: Extensions to the Message-Passing Interface, MPI Forum

Norman, M. L. 2000, in Astrophyiscal Plasmas: Codes, Models, and Observations, Mexico City

OpenMP Architecture Review Board. 2002, OpenMP C and C++ Application Program Interface, 2nd edn.

Ostriker, J. 1964, ApJ, 140, 1056

Reynders, J. V. W., Hinker, P. J., Cummings, J. C., et al. 1996, in Parallel Programming in C++, ed. G. V. Wilson & P. Lu (MIT Press), 547–588

Ritchmyer, R. D. & Morton, K. W. 1967, Difference methods for initial-value problems (Interscience Tracts in Pure and Applied Mathematics, New York: Interscience, 1967, 2nd ed.)

Rozyczka, M. & Laughlin, G. 1997, in ASP Conf. Ser. 121: IAU Colloq. 163: Accretion Phenomena and Related Outflows, 792

Ruden, S. P. & Pollack, J. B. 1991, ApJ, 375, 740

Ruffert, M. 1992, A&A, 265, 82

Schäfer, C., Speith, R., Hipp, M., & Kley, W. 2004, A&A, 418, 325

Sedov, L. I. 1977, Moscow Izdatel Nauka

Shakura, N. I. & Sunyaev, R. A. 1973, A&A, 24, 337

Sod, G. A. 1978, Journal of Computational Physics, 27, 1

Stone, J. M. & Norman, M. L. 1992, ApJS, 80, 753

Tassoul, J. 1978, Theory of rotating stars (Princeton Series in Astrophysics, Princeton: University Press, 1978)

van Leer, B. 1977, Journal of Computational Physics, 23, 276

Veldhuizen, T. L. 1995, C++ Report, 7, 26

Yorke, H. W., Bodenheimer, P., & Laughlin, G. 1993, ApJ, 411, 274

Ziegler, U. & Yorke, H. W. 1997, Computer Physics Communications, 101, 54

Zinnecker, H. & Mathieu, R. 2001, in IAU Symposium, Vol. 200

A Using Advanced C++ Techniques for CFD

Choosing the programming language for computational fluid dynamics applications is no longer only a question of performance and politics, but also a matter of allowing rapid development and deployment of new numerical techniques and easy adaption for specific problems. While popular and widely available, both the Fortran 77 and the C programming language have their problems in the latter two areas. This leaves us with the in the numerical community well known Fortran 90 language and its successors. However, one disadvantage is the lack of a free compiler.

C++ is not usually considered for numerical applications due to performance and interoperability problems caused by missing standard conformance of early implementations. This situation has vastly improved in the last five years so that both free and commercial compilers with good performance and standards conformance are available today. This allows to use new programming techniques, that one only became aware of recently, in numerical applications. These both speed up the applications and increase the expressiveness of the C++ programming language.

This chapter will focus on using modern C++ programming techniques (usually known as template meta programming) for rapid development of computational fluid dynamics codes. We will base this work on the freely available POOMA library that is introduced in appendix B. Here we thoroughly introduce C++ expression templates and show how they can be used in the process of doing hight-level optimizations on scientific programs. We start by motivating expression templates by listing features from Fortran 90/95 we would like to have available in C++. Looking at naive attempts to implement these with standard object-oriented techniques, we will see the problems in getting optimized code. This leads directly to the introduction of expression templates. Last but not least we will shed some light on the rumors of problems with C++ compilers and the possibilities to solve these.

A.1 Language and Library Features

Before going into details we briefly enumerate the features desirable for CFD development of different programming languages. We will assume the reader is familiar with the Fortran 77 and the C programming languages, where from our point of view C is Fortran 77 with pointers (and dynamic storage) but without multi-dimensional arrays.

There are numerous features Fortran 90/95 add over Fortran 77 that are useful for rapid development of CFD algorithms:

- dynamic storage (pointers, allocate and deallocate)
- derived types including access control
- a module concept with the possibility of function and operator overloading
- array expressions on full arrays and sections of arrays, including reductions and masked operations
- stencil operations with the FORALL construct (Fortran 95 only)

Of these the first three are available natively in the C++ programming language, too. The last two, multi-dimensional arrays and expressions on them can be implemented as

a library. Furthermore, exposing the internals of array expressions allows for automatic parallelization done by the library without user interaction. One implementation of such array functionality is the POOMA library. See sec. B.2 for a side-by-side comparison of POOMA and Fortran 95 arrays.

Extending C++ by multi-dimensional arrays and array expressions is done by extensively using operator overloading and partial template specialization. At the same time this allows the array infrastructure to be used with complex types such as complex numbers, vectors and tensors.

A.2 Object-Oriented Programming and Pitfalls

Object-oriented programming languages like C++ make development easier, but performance tuning harder. The same abstractions that allow programmers to focus on what their program is doing, rather than how it is doing it, also make it harder for compilers to re-order operations, predict how many times a loop will be executed, or re-use an area of memory instead of making an unnecessary copy.

For example, suppose that a class `FloatVector` is being used to store and operate on vectors of floating-point values. As well as constructors, a destructor, and element access methods, this class also has overloaded operators that add, multiply, and assign whole vectors:

```
class FloatVector
{
public:
    FloatVector();                      // default constructor
    FloatVector(int size);              // constructor
    FloatVector(int size, float val);   // constructor with initial value
    FloatVector(const FloatVector&);    // copy constructor
    virtual ~FloatVector();             // destructor

    int size() const;                   // get vector size

    float getAt(int i) const;           // get element i
    void setAt(int i, float val);       // set element i to val

    // arithmetic operators creating new vectors
    FloatVector operator+(const FloatVector&);
    FloatVector operator*(const FloatVector&);

    // element-wise assignment
    FloatVector& operator=(const FloatVector&);

protected:
    int len_m;                          // vector length
    float* val_m;                       // value array
};
```

Look closely at what happens when a seemingly innocuous statement like the following is executed:

```
FloatVector V, W, X, Y;                                          (1)
// initialization
V = W * X + Y;
```

W*X allocates a new FloatVector and fills it with the element wise product of W and X by looping over the raw block of floats encapsulated by those two vectors. The call to the addition operator then creates another temporary FloatVector, and executes another loop to fill it. The call to the assignment operator doesn't create a third temporary, but does execute a third loop. The net result is that the initialization statement (1) does the equivalent of the following code (note that the end of the scope limits the temporaries life-time):

```
{                                                                (2)
  FloatVector temp_1(W.size());
  for (int i=0; i<temp_1.size(); ++i)
    temp_1.setAt(i, W.getAt(i) * X.getAt(i));

  FloatVector temp_2(Y.size());
  for (int i=0; i<temp_2.size(); ++i)
    temp_2.setAt(i, temp_1.getAt(i) + Y.getAt(i));

  for (int i=0; i<V.size(); ++i)
    V.setAt(i, temp_2.getAt(i));
}
```

Clearly, if this program has been written more C-like, the three loops would have been combined, and the two temporary vectors eliminated, to create the more efficient code shown below:

```
for (int i=0; i<V.size(); ++i)                                   (3)
  V.setAt(i, W.getAt(i) * X.getAt(i) + Y.getAt(i));
```

Turning the compact C++ expression (1) into the single optimized loop (3) is beyond the capabilities of existing compilers, even disallowed from a language lawyer perspective. Because operations may involve aliasing—i.e., because an expression like V=W*X+V can assign to a vector while also reading from it—optimizers must stay on the side of caution, and neither eliminate temporaries nor fuse loops. This has led many programmers to believe that C++ is intrinsically less efficient than C or Fortran 77, and that while object-oriented languages are good for building user interfaces, they will never deliver the performance needed for modern scientific applications.

This conclusion is wrong. By making full use of the features of the ISO C++ standard, one can give a modern C++ compiler the information it needs to compile C++ programs that achieve Fortran 77 levels of performance. What's more, one does not sacrifice either readability or usability in order to achieve this: in fact, such programs are more portable, and more readable, than many of their peers.

In order to understand how these techniques work, it is necessary to have at least some understanding of how compilers optimize code, and what C++ templates can do. The sections below discuss each of these topics in turn.

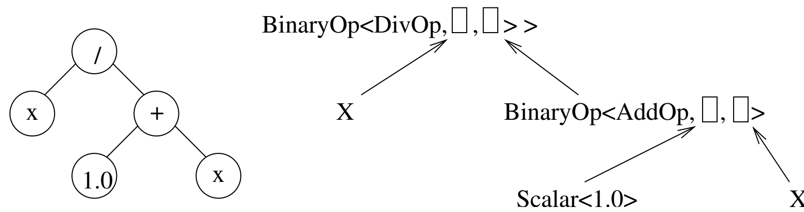


Figure 58: Parse tree of -B+2*C (left), expression template representation (right).

A.3 Expression Templates

Expression templates are designed to avoid the creation of temporary objects for intermediate results of expressions of complex objects (such as the `FloatVector` discussed in the previous chapter). Expression templates have been introduced by Veldhuizen (1995). In the following the working mechanisms of expression templates are introduced.

Parse trees[5] are commonly used by compilers to store the essential features of the source of a program. The leaf nodes of a parse tree consist of atomic symbols in the language, such as variable names or numerical constants. The parse tree's intermediate nodes represent ways of combining those values, such as arithmetic operators and control flow statements. For example, the expression `X/(1.0+X)` could be represented by the parse tree in fig. 58.

Parse trees are often represented textually using prefix notation, in which the non-terminal combiner and its arguments are strung together in a parenthesized list. For example, the expression `X/(1.0+X)` can be represented as `(/ X (+ 1.0 X))`.

What makes all of this relevant to high-performance computing is that the expression `(/ X (+ 1.0 X))` could equally easily be written as `BinaryOp<DivOp, X, BinaryOp<AddOp, Scalar<1.0>, X> >`: it is just a different notation. However, this notation is very similar to the syntax of C++ templates — so similar, in fact, that it can actually be implemented given a careful enough set of template definitions. As discussed earlier, by providing more information to the optimizer as programs are being compiled, template libraries can increase the scope for performance optimization. Any facility for representing expressions as trees must provide:

- a representation for leaf nodes (operands);
- a way to represent operations to be performed at the leaves (i.e. functions on individual operands);
- a representation for non-leaf nodes (operators);
- a way to represent operations to be performed at non-leaf nodes (i.e. combiners);
- a way to pass information (such as the function to be performed at the leaves) downward in the tree; and
- a way to collect and combine information moving up the tree.

[5] Compiler-internal representation of a program.

C++ templates have not been designed with these requirements in mind, but it turns out that they can satisfy them. The central idea is to use the compiler's representation of type information in an instantiated template to store operands and operators. For example, suppose that a set of classes have been defined to represent the basic arithmetic operations:

```
struct AddOp
{
  static double apply(double x, double y)
  {
    return x + y;
  }
};
struct MulOp
{
  static double apply(double x, double y)
  {
    return x * y;
  }
};
```

and so on. Now suppose that a templated class `BinaryOp` has been defined as follows:

```
template<class Operator, class RHS>
struct BinaryOp
{
  // constructor triggers type identification needed for template expansion
  BinaryOp(Operator op, const Vector& leftArg, const RHS& rightArg)
    : left_m(leftArg),
      right_m(rightArg)
  {}

  // calculate value of expression at specified index by recursing
  double apply(int i) const
  {
    return Operator::apply(leftArg.apply(i), rightArg.apply(i));
  }

  const Vector& left_m;
  const RHS& right_m;
};
```

If `b` and `c` have been defined as `Vector`, and if `Vector::apply()` returns the vector element at the specified index, then when the compiler sees the following expression:

```
BinaryOp<MulOp, Vector, Vector>(MulOp(), b, c).apply(3)
```

it translates the expression into `b.apply(3) * c.apply(3)`. The creation of the intermediate instance of `BinaryOp` can be optimized away completely, since all that object does is record a couple of references to arguments.

The question is where do these complications pay off? Consider what happens when the complicated expression above is nested inside an even more complicated expression, which adds an element of another vector `a` to the original expression's result:

```
BinaryOp<AddOp, Vector, BinaryOp<MulOp, Vector, Vector> >
  (a, BinaryOp<MulOp, Vector, Vector>(b, c)).apply(3);
```

This expression calculates `a.apply(3) + (b.apply(3) * c.apply(3))`. If the expression has been wrapped in a `for` loop, and the loop's index has been used in place of the constant `3`, the expression would calculate new values for an entire vector:

```
BinaryOp<AddOp, Vector, BinaryOp<MulOp, Vector, Vector> >                    (4)
  expr(a, BinaryOp<MulOp, Vector, Vector>(b, c));
for (int i=0; i<vectorLength; ++i)
  double tmp = expr.apply(i);
```

The possible nesting of `BinaryOp` inside itself is the reason that the `BinaryOp` template has two type parameters. The first argument to a `BinaryOp` is always a `Vector`, but the second may be either a `Vector` or an expression involving `Vector`s.

The code above is not something any reasonable person would want to write. However, having a compiler create this loop and its contained expression automatically is entirely plausible. The first step is to overload addition and multiplication for vectors, so that `operator+(Vector,Vector)` and `operator*(Vector,Vector)` instantiate `BinaryOp` with `AddOp` (and `MulOp`) as its first type argument, and invokes the `apply()` method of the instantiated object. The second step is to overload the assignment operator so that it generates the loop shown in (4):

```
template<class Op>
Vector& operator=(Vector& target, BinaryOp<Op>& expr)
{
    for (int i=0; i<vectorLength; ++i)
        target.set(i, expr.apply(i));
    return target;
}
```

With these operator definitions, the simple expression

```
Vector x, a, b, c;
x = a + b * c;
```

is automatically translated into the efficient loop (4), rather than into the inefficient loops shown earlier. The expression on the right hand side is turned into an instance of a templated class whose type encodes the operations to be performed, while the implementation of the assignment operator causes that expression to be evaluated exactly once for each legal index. No temporaries are created, and only a single loop is executed.

Libraries based on expression templates push C++ to its limits. Defining the templated classes such a library requires is a painstaking task, as is ensuring that their expansion produces the correct result, but once it has been done, programmers can take full advantage of operator overloading to create compact, readable, maintainable programs without sacrificing performance.

A.4 Data-Flow Analysis and Out-of-order Execution

Passing data-parallel expressions through the expression template machinery front end allows to create iteration objects referring to the expression template expanders. Deferring

the execution of the iterates and queuing them for later execution allows analyzing of inter-iterate data-dependencies. Iterates can then be reordered and executed out-of-order. This allows minimizing effective communication latency by issuing unrelated iterates between starting and finishing communication. It also allows scheduling for optimized cache locality.

The run-time overhead of this data-flow analysis is small compared to the gains of reduced communication latency. This also keeps complexity away from the compiler (which in case of Fortran 90 could also do such kind of analysis statically during compilation) and hidden inside the library. One can then easily tune the scheduling based on the parallel implementation, be it OpenMP, where cache-locality is important, or MPI, where communication latency is dominant.

A.5 C++ Compilers and their Problems

Using C++ for computational fluid dynamics has long time been a problem because of diverging support for parts of the language by the compilers. Also few compilers could handle all the possible uses of the specified template machinery correctly if at all.

This situation has much improved recently, not only by the advent of commercial ISO conforming C++ front-ends on which nearly all hardware vendor compilers are based on nowadays, but also because new programming techniques instrumenting the C++ language to all its extents are now widely deployed in the scientific and commercial community.

With introduction of the ISO/IEC 14882 international standard of the C++ programming language (also known as C++98) and the revised ISO/IEC 9899:1999 standard of the C programming language (also known as C99) there is now fine grained documentation on which implementations and users can rely on.

Both this and recent quantum leaps in compiler optimization technology such as doing whole-program analysis on a tree level makes C++ compete with C and Fortran dialects in both usability and performance.

There are remaining unsolved problems with todays build-environment. One is that most compilers produce very long and cryptic error messages if they encounter an error while expanding templated functions and classes, particularly if those functions and classes are nested. Since POOMA uses templates extensively, it is not uncommon for a single error to result in several pages of complaints from the compiler. Also, programs that use templates extensively are also slower to compile than programs that do not, and the executables produced by the compiler can be surprisingly large. Finally, some debuggers still provide only limited support for inspecting templated functions and classes. All of these problems are actively being addressed by vendors, primarily in response to the growing popularity of the Standard Template Library (STL). Once again, the large and growing user base for C++ means that scientific programmers can take advantage of the fact that even the best tools are constantly being improved.

The performance of the C subset of C++ is usually equal to the performance of C and Fortran77 programs. Depending on the use of C++ abstraction mechanisms, C++ can fall behind on performance dependent on the quality of the compiler implementation. This is also known as the *abstraction penalty*.

Reducing (or better removing) the abstraction penalty is a task for both the compiler and the programmer. First the programmer needs to ensure that in principle abstraction can be removed with language conforming transformations. A simple example is intro-

ducing wrapper classes with single members as used extensively in the standard template library.

```
template <class T>
class Number {
public:
        Number() {}
        Number(const T& v) : val(v) {}
        operator T() const { return val; }
        Number<T>& operator=(const T& v) { val = v; return *this; }
private:
        T val;
};
```

Here using a variable of type `T` and `Number<T>` can and should result in identical code generated by the compiler. That is, the abstraction penalty for this kind of abstraction should be zero.

A commonly cited test-suite for checking the abstraction penalty is the so called Stepanov benchmark[6]. It has been developed by Alex Stepanov, one of the inventors of the STL. Another one is OOPack[7], developed by Arch D. Robinson of Kuck & Associates.

For reducing the abstraction penalty, function inlining, complete loop peeling and scalarization of structures are the key optimizations a compiler has to perform. Nearly all compilers are able to do this for a small hierarchy of abstraction.

For expression templates used in the POOMA library, the abstraction level of the expression inside the evaluation loop can become very large dependent on the complexity of the expression. Unfortunately most compilers will fail on the task of completely removing the abstraction penalty due to missed inlining possibilities. Complete inlining of the whole hierarchy of methods into a call-free loop body is required for the other optimizers being able to do their work. See sec. 3.2 for a way to improve the GNU Compiler Collection (GCC) in this regard.

[6] http://anubis.dkuug.dk/jtc1/sc22/wg21/docs/18015/D_3.cpp

[7] http://www.planet-source-code.com/vb/scripts/ShowCode.asp?txtCodeId=168&lngWId=3

B Introduction to the POOMA Library

POOMA is a C++ class library for high-performance scientific computation. POOMA runs efficiently on single-processor desktop machines, shared-memory multiprocessors, and parallel supercomputers. POOMA provides multi-dimensional arrays supporting parallel decompositions, multi-threading, and out-of-order execution for maximum performance. The syntax presented to the user (see example in fig. 59) is very similar to that of Fortran 90/95. In fact, the combination of C++ and POOMA provides so many of the features of Fortran 90 that one might well ask whether it wouldn't better to just use the latter language.

The simple answer is that the abstraction facilities of C++ are much more powerful than those in Fortran. A more powerful answer is economics. While the various flavors of Fortran are still the lingua franca of scientific computing, Fortran's user base is shrinking, particularly in comparison to C++. Networking, graphics, database access, and operating system interfaces are available to C++ programmers long before they are available for Fortran. Support tools such as debuggers and memory inspectors are primarily targeted at C++ developers, as are hundreds of books, journal articles, and web sites.

Until recently, Fortran had two powerful arguments in its favor: legacy applications and performance. However, the importance of the former is diminishing as the invention of new algorithms force programmers to rewrite old codes, while the invention of techniques such as expression templates has made it possible for C++ programs to match the performance of optimized Fortran 77.

POOMA has been designed and implemented by scientists working at the former Advanced Computing Laboratories of the Los Alamos National Laboratory and made freely available for unrestricted use, modification and redistribution. The library can be obtained from Savannah.nongnu.org[8] as FreePOOMA both as tarball and by anonymous CVS.

With the following discussion we will refer to the improved POOMA library as it is available on the CD packed with the appendix of this work. There is reference documentation available for the POOMA library that can be extracted from the source using the Doxygen tool[9]. This reference documentation is also available as html browsable on the CD.

The following is intended as a brief introduction to the POOMA library, including an overview of the most useful features. It does not supersede using the reference documentation to gather more detailed information about the introduced features. In particular this introduction is not an exhaustive presentation of the POOMA library and its features but a quick-start guide for the impatient.

B.1 Initialization and Overview

The POOMA library has to be initialized at program start and taken down at program end. This is done by calling the libraries `initialize` and `finalize` functions in the main program:

```
int main(int argc, char **argv)
{
```

[8] `http://savannah.nongnu.org/projects/freepooma/`
[9] `http://www.stack.nl/~dimitri/doxygen/index.html`

```
#include "Pooma/Arrays.h"
#include <iostream>

const int N = 20; // The size of each side of the domain.

int main(int argc, char **argv)
{
    Pooma::initialize(argc, argv);

    // The array we'll be solving for
    Array<2> x(N, N);
    x = 0.0;

    // The right hand side of the equation (spike in the center)
    Array<2> b(N, N);
    b = 0.0;
    b(N/2, N/2) = -1.0;

    // Specify the interior of the domain
    Interval<1> I(1, N-2), J(1, N-2);

    // Iterate 200 times
    for (int i=0; i<200; ++i)
    {
        x(I,J) = 0.25*(x(I+1,J) + x(I-1,J) + x(I,J+1) + x(I,J-1) - b(I,J));
    }

    // Print out the result
    std::cout << x << std::endl;

    Pooma::finalize();
    return 0;
}
```

Figure 59: Laplace solver using Jacobi iteration.

```
  Pooma::initialize(argc, argv);
  ...
  Pooma::finalize();
  return 0;
}
```

POOMA takes over the responsibility for initializing the MPI subsystem, if appropriate.

Like for MPI you can query the number of contexts participating in computation (like the MPI world communicator size) using the `Pooma::contexts()` function. An identifier of the own context (like the MPI rank) can be queried using the `Pooma::context()` function. These functions return one and zero for serial operation.

Usually textual user interaction should be restricted to one process. POOMA provides the `Pooma::Inform` class which is iostream compatible and can be used to do serialized I/O.

B.2 Arrays

The POOMA library provides an abstract array object which mimics multi-dimensional array support of Fortran 90. Thus, in essence you declare an array like

```
Array<3> a(16, 32, 16);
```

which declares a three-dimensional array `a` of size 16x32x16. Array indices start at zero unless you specify otherwise. Individual elements of an array can be accessed using the overloaded function call operator or the `read()` method:

```
a(1, 5, 6) = 0.0;
double x = a.read(5, 5, 0);
```

Here only the function call operator returns an lvalue.

B.2.1 Array Expressions

Like with Fortran 90/95, POOMA arrays can be used in expressions, including reductions and selections. For the following examples we assume the two-dimensional arrays `a`, `b`, `c` being available through

```
Array<2> a(16, 32), b(16, 32), c(16, 32);
```

You can assign a constant (not dependent on array index) expression to an array

```
a = sin(2*M_PI);
```

which is equivalent to

```
for (int i=0; i<16; ++i)
  for (int j=0; j<32; ++j)
    a(i, j) = sin(2*M_PI);
```

You can assign an array expression to another array

```
a = a + sin(b);
```

or using the accumulate operator

```
a += sin(b);
```

which is equivalent to

```
for (int i=0; i<16; ++i)
  for (int j=0; j<32; ++j)
    a(i, j) += sin(b(i, j));
```

Arrays and array expressions can be reduced to a single value by one of the reduction functions like in

```
double m1 = min(a);
double m2 = sum(b + c);
```

which performs the equivalent to the Fortran 90 and C variants

```
m1 = MINVAL(a)
m2 = SUM(b + c)
```

```
double m1 = a(0, 0);
for (int i=0; i<16; ++i)
  for (int j=0; j<32; ++j)
    if (a(i, j) < m1)
      m1 = a(i, j);
double m2 = 0.0;
for (int i=0; i<16; ++i)
  for (int j=0; j<32; ++j)
    m2 += b(i, j) + c(i, j);
```

Expressions and assignments can be made conditional by using a mask created by a where expression:

```
a = where(b < 0.0, c);
a = where(b < 0.0, b, c);
```

which operates similarly to the Fortran 90 WHERE:

```
WHERE (b < 0.0) a = c
WHERE (b < 0.0)
  a = b
ELSEWHERE
  a = c
ENDWHERE
```

and is equivalent to the following C code:

```
for (int i=0; i<16; ++i)
  for (int j=0; j<32; ++j)
    if (b(i, j) < 0.0)
      a(i, j) = c(i, j);
for (int i=0; i<16; ++i)
  for (int j=0; j<32; ++j)
    if (b(i, j) < 0.0)
      a(i, j) = b(i, j);
    else
      a(i, j) = c(i, j);
```

B.2.2 Array Sub-Objects

Like in Fortran 90 you can select a part of an array and operate on it like it were a normal array. Types of selections include sub-domain selections, domain slices and element selections.

Selections generally are done using one of the overloaded function call operators of the array. Domains can be specified in various ways with the POOMA library, including using one-dimensional points which can be either plain `int` or the `Loc<1>` type, and one-dimensional (continuous) intervals using the `Interval<1>` type. With this brief introduction, selecting a single element can be done in different ways:

```
Array<2> a(16, 16);
double x = a(1, 1);
x = a(Loc<1>(1), Loc<1>(1));
x = a(Loc<1>(1), 1);
x = a(1, Loc<1>(1));
x = a(Loc<2>(1, 1));
```

hereby introducing higher dimensional location types. The library will automatically combine the domains provided as arguments to the arrays `operator()` into one higher-dimensional selection domain for performing the selection. Selecting a sub-domain out of an array is done in a similar way

```
Array<2> a(16, 16);
a(Interval<1>(2, 4), Interval<1>(16)) = 0.0;
a(Interval<2>(Interval<1>(2, 4), 16)) = 0.0;
```

introducing higher dimensional interval types and assigning zero to all elements with first index two to four. The Fortran 90 variant would look like

```
REAL, DIMENSION(0:15, 0:15) :: A
A(2:4,) = 0.0
```

Providing different strides than one as it is done using `start:end:stride` specification in Fortran 90 can be done using a `Range` domain instead of an `Interval` domain like `Range<1>(start, end, stride)`. Slicing an array is possible, too, for forming expressions involving arrays with different dimensions:

```
Array<2> a(16, 16);
Array<1> b(14);
b = a(2, Range<1>(14, 1, -1));
```

which assigns values of row 1 to 14 of column two of array a in reverse order to the elements of array b like the Fortran 90 variant

```
REAL, DIMENSION(0:15, 0:15) :: A
REAL, DIMENSION(0:13) :: B
B = A(2, 14:1:-1)
```

Array selections can be used in array expressions just like a normal array. Remember though that indices of a selection start at zero, so the following two expressions are identical, just like their Fortran 90 counterpart:

```
Array<1> a(8), b(8);
a(Interval<1>(1, 5)) = b(Interval<1>(3, 7));
for (int i=1; i<=5; ++i)
  a(i) = b(i+2);

REAL, DIMENSION(0:7) :: A, B
A(1:5) = B(3:7)
```

The stencil example presented in sec. B.5.1 can be written using selections like the following:

```
Array<2> a(16, 32), b(16, 32);
Interval<1> I(1,14), J(1,30);
a(I, J) = 0.25*(b(I-1, J) + b(I+1, J) + b(I, J-1) + b(I, J+1));
```

Here arithmetic expressions involving a domain (like `Interval` and `Range`) and a point-domain (like `int` and `Loc`) result in shifting the domain.

B.3 Parallel Setup

Setup of a parallel computation is done by specifying an appropriate data layout connecting the topology of your domain to that of the parallel computer. In the most simple case this is by subdividing the domain into rectangular sub-domains organized on a grid and assigning an equal number of sub-domains to each node/processor of the parallel computer.

The POOMA library provides different layout classes from which the two, `DomainLayout` and `GridLayout` are the most interesting. `DomainLayout` is used for serial computations and is constructed from the computational vertex domain and optionally a `GuardLayers` object. The `GridLayout` does a grid subdivision as specified by one of the constructor variants. The most simple is just specifying the vertex domain, the guard layers and the `DistributedTag` tag object:

```
GridLayout(const Interval<Dim>&, const GuardLayers&, const DistributedTag&);
```

In this case the subdivision is done automatically based on the number of available contexts. The other interesting case is specifying the subdivision manually by providing either an additional `Loc` object specifying the grid-counts for each direction, or by providing a `Grid` object in place of the domain object:

```
GridLayout(const Interval<Dim>&, const Loc<Dim>&, const GuardLayers<Dim>&,
           const DistributedTag&);
GridLayout(const Grid<Dim>&, const GuardLayers<Dim>&, const DistributedTag&);
```

The complete sequence for a setup of a three-dimensional parallel array would then look like

```
Interval<3> domain(16, 32, 16);
GridLayout<3> layout(domain, GuardLayers<3>(1), DistributedTag());
typedef MultiPatch<Remote<Brick>, GridTag> > EngineTag_t;
Array<3, double, EngineTag_t> a(layout);
```

where the array is automatically distributed over all available processors. There is one level of guard cells beyond each internal and external face to allow for fast finite differencing without communication. Note that updating of internal guards is done automatically whenever necessary.

B.4 Fields

POOMA introduces another data object apart from `Array`, namely a `Field`. Think of fields as of arrays with an attached mesh describing coordinates, ways to specify boundary conditions and relations between fields. Also fields may consist of more than one data object to represent different materials and/or centering points on the grid.

As opposed to arrays, fields are templated on the mesh type instead of the dimensionality. Other things, like parallel setup using layouts is the same for arrays and fields. Generally fields can be substituted for arrays wherever the latter are valid.

B.4.1 Meshes

A mesh contains coordinates for each indexed field position and information about the coordinate system. Also some geometry related methods are available to make finite difference code covariant (see reference documentation). An appropriate `Mesh` type can be specified using the `MeshTraits` traits class which is declared as follows:

```
template <int Dim, class T, class MeshTag, class CoordinateSystemTag>
struct MeshTraits;
```

The traits class contains the definition of a mesh type via `MeshTraits::Mesh_t`. Possible types for `MeshTag` include `UniformRectilinearTag` (default) and `RectilinearTag`, available types for `CoordinateSystemTag` are `CartesianTag` (default), `CylindricalTag` and `SphericalTag`. The default coordinate type `T` is `double`. Creation of a simple field can be done like the following:

```
Interval<2> domain(16, 16);
MeshTraits<2>::Mesh_t mesh(domain, Vector<2>(0.0, 0.0), Vector<2>(1.0, 1.0));
DomainLayout<2> layout(domain);
Centering<2> cell = canonicalCentering<2>(CellType, Continuous);
Field<MeshTraits<2>::Mesh_t> f(cell, layout, mesh);
```

which constructs a two-dimensional cell-centered field with a Cartesian grid with origin $(0, 0)$ and uniform vertex spacing $(1, 1)$.

B.4.2 Materials and Centerings

Fields can consist of multiple materials and/or have multiple centering points. Both can be specified at field construction time via one of

```
template <class Layout, class Mesh>
Field(const Centering<Dim>&, const Layout&, const Mesh& mesh);
template <class Layout, class Mesh>
Field(int materials, const Centering<Dim>&, const Layout&, const Mesh& mesh);
```

where for the centering you can supply one of the default centering objects as returned from

```
Centering<Dim> canonicalCentering<Dim>(CenteringType, ContinuityType);
```

The centering type should be one of `CellType`, `VertexType`, `FaceType` and `EdgeType`, the continuity type should be either `Continuous` or `Discontinuous`, where a discontinuous field does not share faces, edges and vertices between cells. Centering points and materials can be accessed using the following methods from the field class:

`subField(int m, int c)`	material m of centering point c
`center(int c)`	centering point c of all materials
`material(int m)`	material m with all centering points

These all return fields you can operate on in the usual ways.

B.4.3 Relations

There is a way in POOMA to specify relations between fields. This means if the content of one field that is related to others change, the other fields will be updated in a way specified by the relation once their contents is needed. This is done by associating a C++ functor with such a relation. Relations are very powerful, as can be seen from the next two examples.

Example one is a simple self-relation maintaining a minimum floor density. First the definition of the functor:

```
struct limit_rh
{
  limit_rh() {}
  limit_rh(double rhomin) : rhomin_m(rhomin) {}
  template <class F>
  limit_rh(const limit_rh& l, const F&) : rhomin_m(l.rhomin_m) {}
  template <class F>
  void operator()(const F& rh) const
  {
    Interval<F::dimensions> I = rh.physicalDomain();
    rh(I) = where(rh(I) < rhomin_m, rhomin_m);
  }
  double rhomin_m;
};
```

This is then applied to the density field `rh` using

```
  Pooma::newRelation(limit_rh(1e-6), rh);
```

and whenever `rh` is altered a limiting pass is scheduled before the next use of `rh`. Example two is maintaining the pressure by relating it to density and temperature:

```
struct ideal_gas
{
  ideal_gas() {}
  template <class F>
  ideal_gas(const ideal_gas&, const F&) {}
  template <class F1, class F2, class F3>
  void operator()(const F1& pg, const F2& T, const F3& rh) const
  {
    Interval<F1::dimensions> I = pg.totalDomain();
    pg(I) = (T*rh)(I);
  }
};
```

Apply this relation functor to `pg` using

```
  Pooma::newRelation(ideal_gas(), pg, T, rh);
```

and get pressure calculated from density and temperature automatically whenever needed.

B.4.4 Boundary Conditions

Boundary conditions are applied to fields using self-referencing relations. There are a set of predefined boundary condition functors that can be used, together with a simplified relation initialization:

ConstantFaceBC	constant value on the boundary and in the external guards
PeriodicFaceBC	periodicity in the applied dimension
PosReflectFaceBC	positive value reflecting (tangential velocity, density)
NegReflectFaceBC	negative value reflecting (normal velocity)
ZeroGradientFaceBC	same as first non-guard value in the external guards

These are applied using the generic

```
template <class Target>
add*(const Target&, int face, ...);
template <class Target>
addAll*(const Target&, ...);
```

where * is to be substituted by the name of the boundary condition from the table above and the dots denote extra arguments specific to the boundary condition. Face is the face the boundary condition is for (0, 1 for x lower, upper, 2, 3 for y lower, upper, etc.), the `addAll*` variants add the boundary condition to all faces of the target. See reference documentation for details on the different predefined boundary conditions.

B.5 Finite Differencing

There are several ways of implementing finite difference equations. One is the use of array expressions with sub-objects as has been shown in sec. B.2.1 and sec. B.2.2. Other ways include using array stencils or the ScalarCode infrastructure, which will be introduced here.

B.5.1 Array Stencils

POOMA provides a way to specify stencil operations which can be applied to an array producing a result array. This is modeled after the Fortran 95 FORALL construct, but needs more elaboration on the operation. Basically you need to provide the library with the stencil extent and an implementation of a function call operator (functor) that performs the required computation. An example looks like

```
struct MyStencil {
  template <class A>
  typename A::Element_t operator()(const A& a, int i, int j) const
  {
    return 0.25*(a(i-1,j) + a(i+1,j) + a(i,j-1) + a(i,j+1));
  }
  int lowerExtent() const { return 1; }
  int upperExtent() const { return 1; }
};
...
  Array<2> a(16, 32), b(16, 32);
  Interval<1> I(1, 14), J(1,30);
  a(I, J) = Stencil<MyStencil>()(b, Interval<2>(I, J));
```

This is equivalent to the following Fortran 95 loop:

```
FORALL(I = 1:14, J = 1:30)
  A(I, J) = 0.25*(B(I-1,J) + B(I+1,J) + B(I,J-1) + B(I,J+1))
END FORALL
```

More complex operations involving multiple arrays can be constructed in a similar way using the ScalarCode functionality of the POOMA library.

B.5.2 ScalarCode

We are presenting a more powerful and at the same time more low-level approach to finite differencing here. The so called ScalarCode infrastructure enables you to specify a C++ functor handling per grid-point operation. Up to now this is the same as for stencils, but with ScalarCode you can use more than one object on the right hand side of the equation, and you can even use more than one left hand side, i.e. do multiple equations in parallel.

For a ScalarCode functor you need to specify the shape and type of arguments. This is done via the scalarCodeInfo method of the functor. A typical functor would look like

```
template <int Dim>
struct MomentumfluxX : public Deltas<Dim> {
  MomentumfluxX(double dt_) : dt(dt_) {}

  inline void scalarCodeInfo(ScalarCodeInfo<Dim, 3> &i) const
  {
    i.extent().lower(0) = 1;
    i.extent().upper(0) = 2;
    i.write(0, true);
    i.write(1, false);
    i.write(2, false);
    i.useGuards(0, false);
    i.useGuards(1, true);
    i.useGuards(2, true);
  }

  double dt;

  using Deltas<Dim>::dX;

  template <class F1, class F2, class F3>
  inline void operator()(const F1 &flx, const F2 &vx, const F3 &flmx,
                         const Loc<Dim>& I) const
  {
    const typename F3::Mesh_t &m = flx.mesh();
    Loc<Dim> IND = Pooma::NoInit();
    double s, ddm1, dd, ddp1, grad1, grad;

    s = (vx(I) + vx(I+dX)) * 0.5*dt;
    IND = s > 0.0 ? I : I+dX;
    ddm1 = vx(IND-dX);
    dd = vx(IND);
    ddp1 = vx(IND+dX);
    grad1 = (dd-ddm1)*(ddp1-dd);
    grad = grad1 > 0.0
      ? 2.0*grad1 / ((ddp1-dd)*m.vertexSpacing(IND)(0)
                     + (dd-ddm1)*m.vertexSpacing(IND+dX)(0))
      : 0.0;
    flx(I) = 0.5*(flmx(I) + flmx(I+dX))
      * (dd + (m.cellPosition(I)(0) - m.vertexPosition(IND)(0) - 0.5*s) * grad);
  }
};
```

which calculates the momentum-flux `flx` in x-direction from the x-velocity `vx` and the mass-flux `flmx` in x-direction. Note how you can easily use local variables and control-flow statements inside the functor. The functor is invoked via

```
(ScalarCode<MomentumfluxX<Dim> >(dt))(flx, v.center(0), flm.center(0));
```

thereby applied to the physical domain of the field `flx`. Another variant can be used to explicitly specify the evaluation domain of a scalar code iterate, namely:

```
(ScalarCode<MomentumfluxX<Dim> >(dt))(flx, grow(flx.physicalDomain(), 1),
                                      v.center(0), flm.center(0));
```

to apply the functor to the first external guard cells, too.

B.6 File I/O

I/O of POOMA arrays and fields is done via the `HDF5File` class that encapsulates a file and handles serial and parallel file I/O via the HDF5 library[10] and MPI-2 parallel I/O[11] if requested. At construction time of the file object you specify whether you want to initiate parallel I/O. From the pristine object you can open a file for reading or create a new one by one of

```
bool open(const char *name);
bool create(const char *name);
```

where `true` is returned for all operations that are successful and `false` if an error occurred. Closing the file is done automatically at `HDF5File` object destruction time.

You can traverse/create the HDF5 hierarchy using the `popLoc`, `pushLoc` and `pushLocRel` methods which maintain a stack of groups and possibly leaf datasets. Usually you will not need these for simple file structures.

You can use HDF5 attributes to store simple data in the file, like identifier – value pairs. This is done using the

```
template <class T>
bool setAttribute(const char *id, const T& value);
template <class T>
bool getAttribute(const char *id, T& value);
```

methods which store or retrieve an object of type `T` at the current location using the specified identifier. After `getAttribute` all contexts will have an updated value.

POOMA fields and arrays are stored using HDF5 datasets, where for fields additional meta-data such as known relations are stored and reconstructed on read. The interface to reading and writing fields and arrays is

```
template <class MeshTag, class T, class EngineTag>
inline bool writeField(const char *id, const Field<MeshTag, T, EngineTag>& f,
                       const GuardLayers<MeshTag::dimensions>& g);
template <class T2, class MeshTag, class T, class EngineTag>
bool writeFieldOther(const char *id, const Field<MeshTag, T, EngineTag>& f,
                     const GuardLayers<MeshTag::dimensions>& g);

template <class MeshTag, class T, class EngineTag, int Dim>
bool readField(const char *id, const Field<MeshTag, T, EngineTag>& f,
```

[10] `http://hdf.ncsa.uiuc.edu/HDF5/`

[11] e.g. MPICH with ROMIO

```
               GuardLayers<Dim>& g);

template <int Dim, class T, class EngineTag>
inline bool writeArray(const char *id, const Array<Dim, T, EngineTag>& a);
template <class T2, int Dim, class T, class EngineTag>
inline bool writeArrayOther(const char *id, const Array<Dim, T, EngineTag>& a);

template <int Dim, class T, class EngineTag>
bool readArray(const char *id, const Array<Dim, T, EngineTag>& f);
```

where the `write*Other` variants allow the on-disk data type `T2` to differ from the in-memory data type `T`. The guard layer arguments of the field routines let you choose/query the amount of external guard layers saved to / read from disk (defaults to zero). You can read/write parts of an array or a field by passing an appropriate view to the read/write routines. But note that you need to always read all data that has been written.

C Leafify Patch

```
Index: gcc/c-common.c
===================================================================
RCS file: /cvs/gcc/gcc/gcc/c-common.c,v
retrieving revision 1.523
diff -u -c -3 -p -r1.523 c-common.c
*** gcc/c-common.c 28 Jun 2004 02:11:55 -0000 1.523
--- gcc/c-common.c 6 Jul 2004 09:16:37 -0000
*************** static tree handle_noreturn_attribute (t
*** 517,522 ****
--- 517,523 ----
  static tree handle_noinline_attribute (tree *, tree, tree, int, bool *);
  static tree handle_always_inline_attribute (tree *, tree, tree, int,
       bool *);
+ static tree handle_leafify_attribute (tree *, tree, tree, int, bool *);
  static tree handle_used_attribute (tree *, tree, tree, int, bool *);
  static tree handle_unused_attribute (tree *, tree, tree, int, bool *);
  static tree handle_const_attribute (tree *, tree, tree, int, bool *);
*************** const struct attribute_spec c_common_att
*** 578,583 ****
--- 579,586 ----
        handle_noinline_attribute },
    { "always_inline",         0, 0, true,  false, false,
        handle_always_inline_attribute },
+   { "leafify",               0, 0, true,  false, false,
+                              handle_leafify_attribute },
    { "used",                  0, 0, true,  false, false,
        handle_used_attribute },
    { "unused",                0, 0, false, false, false,
*************** handle_always_inline_attribute (tree *no
*** 3967,3972 ****
--- 3970,3998 ----
    return NULL_TREE;
  }

+ /* Handle a "leafify" attribute; arguments as in
+    struct attribute_spec.handler.  */
+
+ static tree
+ handle_leafify_attribute (tree *node, tree name,
+                           tree args ATTRIBUTE_UNUSED,
+                           int flags ATTRIBUTE_UNUSED, bool *no_add_attrs)
+ {
+   if (TREE_CODE (*node) == FUNCTION_DECL)
+     {
+       /* Do nothing else, just set the attribute.  We'll get at
+          it later with lookup_attribute.  */
+     }
+   else
+     {
+       warning ("`%s' attribute ignored", IDENTIFIER_POINTER (name));
```

```
+       *no_add_attrs = true;
+     }
+
+   return NULL_TREE;
+ }
+
+
  /* Handle a "used" attribute; arguments as in
     struct attribute_spec.handler.  */

Index: gcc/cgraphunit.c
===================================================================
RCS file: /cvs/gcc/gcc/gcc/cgraphunit.c,v
retrieving revision 1.67
diff -u -c -3 -p -r1.67 cgraphunit.c
*** gcc/cgraphunit.c 24 Jun 2004 22:42:26 -0000 1.67
--- gcc/cgraphunit.c 6 Jul 2004 09:16:37 -0000
*************** cgraph_decide_inlining_of_small_function
*** 1439,1444 ****
--- 1439,1505 ----
    free (heap_node);
  }

+ /* Find callgraph nodes closing a circle in the graph.  The
+    resulting hashtab can be used to avoid walking the circles.
+    Uses the cgraph nodes ->aux field which needs to be zero
+    before and will be zero after operation.  */
+
+ static void
+ cgraph_find_cycles (struct cgraph_node *node, htab_t cycles)
+ {
+   struct cgraph_edge *e;
+
+   if (node->aux)
+     {
+       void **slot;
+       slot = htab_find_slot (cycles, node, INSERT);
+       if (!*slot)
+  {
+    if (cgraph_dump_file)
+      fprintf (cgraph_dump_file, "Cycle contains %s\n", cgraph_node_name (node));
+    *slot = node;
+  }
+       return;
+     }
+
+   node->aux = node;
+   for (e = node->callees; e; e = e->next_callee)
+     {
+        cgraph_find_cycles (e->callee, cycles);
+     }
+   node->aux = 0;
```

```
+ }
+
+ /* Leafify the cgraph node.  We have to be careful in recursing
+    as to not run endlessly in circles of the callgraph.
+    We do so by using a hashtab of cycle entering nodes as generated
+    by cgraph_find_cycles.  */
+
+ static void
+ cgraph_leafify_node (struct cgraph_node *node, htab_t cycles)
+ {
+   struct cgraph_edge *e;
+
+   for (e = node->callees; e; e = e->next_callee)
+     {
+       /* Inline call, if possible, and recurse.  Be sure we are not
+   entering callgraph circles here.  */
+       if (e->inline_failed
+    && e->callee->local.inlinable
+    && !cgraph_recursive_inlining_p (node, e->callee,
+        &e->inline_failed)
+    && !htab_find (cycles, e->callee))
+  {
+    if (cgraph_dump_file)
+          fprintf (cgraph_dump_file, " inlining %s", cgraph_node_name (e->callee));
+           cgraph_mark_inline_edge (e);
+    cgraph_leafify_node (e->callee, cycles);
+  }
+       else if (cgraph_dump_file)
+  fprintf (cgraph_dump_file, " !inlining %s", cgraph_node_name (e->callee));
+     }
+ }
+
  /* Decide on the inlining.  We do so in the topological order to avoid
     expenses on updating data structures.  */

*************** cgraph_decide_inlining (void)
*** 1477,1482 ****
--- 1538,1561 ----

        node = order[i];

+       /* Handle nodes to be leafified, but don't update overall unit size.  */
+       if (lookup_attribute ("leafify", DECL_ATTRIBUTES (node->decl)) != NULL)
+         {
+    int old_overall_insns = overall_insns;
+    htab_t cycles;
+      if (cgraph_dump_file)
+          fprintf (cgraph_dump_file,
+             "Leafifying %s\n", cgraph_node_name (node));
+    cycles = htab_create (7, htab_hash_pointer, htab_eq_pointer, NULL);
+    cgraph_find_cycles (node, cycles);
+    cgraph_leafify_node (node, cycles);
```

```
+    htab_delete (cycles);
+    overall_insns = old_overall_insns;
+    /* We don't need to consider always_inline functions inside the leafified
+       function anymore.  */
+    continue;
+         }
+
        if (!node->local.disregard_inline_limits)
   continue;
        if (cgraph_dump_file)
Index: gcc/doc/extend.texi
===================================================================
RCS file: /cvs/gcc/gcc/gcc/doc/extend.texi,v
retrieving revision 1.198
diff -u -c -3 -p -r1.198 extend.texi
*** gcc/doc/extend.texi 23 Jun 2004 08:41:55 -0000 1.198
--- gcc/doc/extend.texi 6 Jul 2004 09:16:38 -0000
*************** The keyword @code{__attribute__} allows
*** 1894,1900 ****
  attributes when making a declaration.  This keyword is followed by an
  attribute specification inside double parentheses.  The following
  attributes are currently defined for functions on all targets:
! @code{noreturn}, @code{noinline}, @code{always_inline},
  @code{pure}, @code{const}, @code{nothrow},
  @code{format}, @code{format_arg}, @code{no_instrument_function},
  @code{section}, @code{constructor}, @code{destructor}, @code{used},
--- 1894,1900 ----
  attributes when making a declaration.  This keyword is followed by an
  attribute specification inside double parentheses.  The following
  attributes are currently defined for functions on all targets:
! @code{noreturn}, @code{noinline}, @code{always_inline}, @code{leafify},
  @code{pure}, @code{const}, @code{nothrow},
  @code{format}, @code{format_arg}, @code{no_instrument_function},
  @code{section}, @code{constructor}, @code{destructor}, @code{used},
*************** Generally, functions are not inlined unl
*** 1936,1941 ****
--- 1936,1949 ----
  For functions declared inline, this attribute inlines the function even
  if no optimization level was specified.

+ @cindex @code{leafify} function attribute
+ @item leafify
+ Generally, inlining into a function is limited.  For a function marked with
+ this attribute, every call inside this function will be inlined, if possible.
+ Whether the function itself is considered for inlining depends on its size and
+ the current inlining parameters.  The @code{leafify} attribute only works
+ reliably in unit-at-a-time mode.
+
  @item cdecl
  @cindex functions that do pop the argument stack on the 386
  @opindex mrtd
```

D Supplements

The included CD contains

- Source code to the extended POOMA library.
- Browsable html of the POOMA reference documentation.
- Source code to the three-dimensional CFD code.
- OpenDX HDF5 importer and visualization networks.
- Source code to the performance testing CFD code.
- Patches to various GCC versions for the `leafify` function attribute.

The contents of the CD are also available for restricted access and an undefined time from `http://www.tat.physik.uni-tuebingen.de/~rguenth/phd/`. Permanent availability cannot be guaranteed.

Ich danke meinen Mitstreitern aus C10A02 für die immer nette und zeitweise auch produktive Arbeitsumgebung. Wilhelm Kley, Hubert Klahr und Jochen Peitz danke ich für ihre fachliche Unterstützung in vielen aufschlußreichen Gesprächen.