ETH ZÜRICH

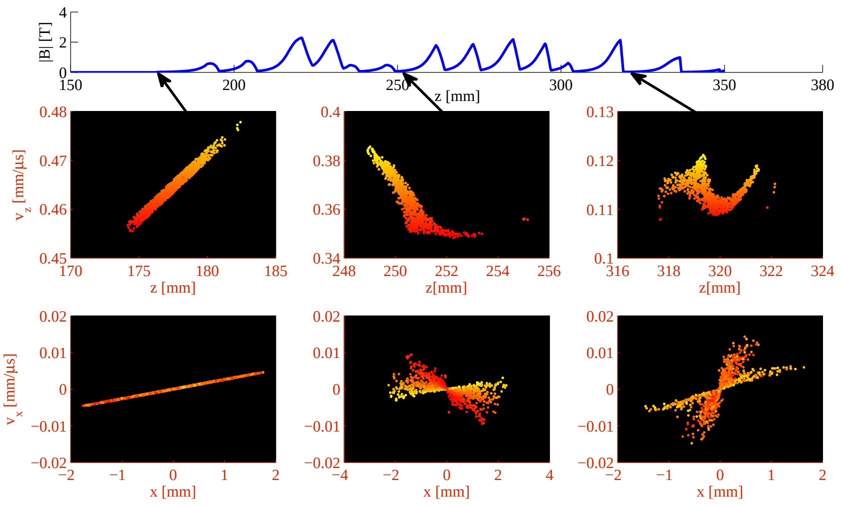
BACHELOR THESIS

Optimizing Zeeman Deceleration of Atomic Hydrogen by Evolutionary Algorithms

Author:
Yves SALATHÉ

Supervisor:
Prof. Dr. Frédéric MERKT

December 17, 2009



Abstract

The theory and computational model to simulate three-dimensional particle trajectories in a multistage Zeeman decelerator is described. Using the Covariance Matrix Adaptation Evolution Strategy (CMA-ES), pulse sequences have been generated that optimize the density and energy of cold hydrogen atoms in a magnetic trap located at the end of the decelerator. An objective function for the optimization of the deceleration and trapping of low-field-seeking particles has been devised and tested. A suggestion for an objective function that can be used in a similar optimization of the deceleration of particles in high-field-seeking states has been made. Throughout the optimization procedure, the simulations serve as a method to evaluate the selected objective function, leading to a simulated enhancement of the number of trapped atoms by more than a factor of 6.

Contents

Chapter 1

Introduction

Research into the development of general techniques to generate cold (1mK–1K) and ultra-cold (< 1mK) samples of atoms and molecules is a particularly active area in the field of atomic and molecular physics at this time. The possibility to laser cool and trap alkali metal atoms [1] has led to many advances, e.g. the observation of Bose-Einstein Condensation (BEC) [2], and a Mott insulating phase in dilute gases [3]. Laser cooling is however limited to a small number of species in the periodic table, including the alkali metals and metastable rare gas atoms. With the desire to study the rich physics associated with the generation of cold molecular samples, new efforts were taken to design and experimentally realize a range of cooling techniques that fill the gaps that are left open by laser cooling. One of these techniques is multistage Stark deceleration, which is a method to decelerate molecules which exhibit a permanent electric dipole moment in their ground state by the electric field gradients of an array of electrodes [4]. Another method is to employ the Stark effect associated with atoms or molecules that have been excited to high Rydberg states [5] where the changed distribution of the Rydberg electron gives rise to a permanent electric dipole moment. Multistage Zeeman deceleration [6, 7, 8, 9], the magnetic analogue of multistage Stark deceleration, can be utilized to decelerate atoms and molecules which possess a magnetic moment arising from an unpaired electron in the ground or metastable states. In the same manner as in multistage Stark deceleration, a pulsed gas beam is exposed to the time-dependent magnetic field of an array of solenoids. After Zeeman deceleration, the particles can be confined at a well-defined position by a magnetic trap field [10, 8]. Once trapped, the radicals can be used to investigate e.g. radical-radical reactions at low temperatures [11] or to measure fundamental constants such as the mass of the neutrino [9]. If the particles are confined at a high local phase-space density, they can in certain cases be cooled down further by evaporative cooling (e.g. see [12], chapter 3.1.2).

For many of these applications, it is important that the number of particles per 3-dimensional position and velocity interval (phase-space density) is as high as possible. This can be achieved by optimizing the phase-space acceptance of the decelerator to match the properties of the gas beam employed, and by then matching the trap characteristics with those of the decelerated cloud of atoms or molecules as it exits the decelerator. In the present work, the Covariance Matrix Adaptation Evolution Strategy (CMA-ES) [13, 14] is applied to optimize the time sequence by which the solenoids are pulsed during the deceleration and trapping of ground state hydrogen. The results of these optimizations have been directly compared to recent experimental measurements and demonstrated a significant enhancement in the local phase-space density of the trapped sample that could be reached.

Chapter 2

The Zeeman effect of the hydrogen atom

2.1 Motivation

In order to understand how Zeeman deceleration works, it is necessary to study the quantum mechanical description of atoms and molecules. The hydrogen atom is the simplest atomic system and serves best as an illustration of the Zeeman effect. The formalism used to calculate the Zeeman effect in H can easily be generalized Furthermore there are ways to generalize the theory of the hydrogen atom to larger atoms or even molecules. In the following, the quantum mechanical treatment of the hydrogen atom is presented with the goal to describe the Zeeman effect which is utilized in Zeeman deceleration.

2.2 Unperturbed Hamiltonian

The hydrogen atom is a two-body system consisting of an electron and a proton. For this treatment, a coordinate system relative to the centre-of-mass of these two particles will be chosen. The electron and the proton can be treated as points with masses m_e and m_p respectively and opposite charges of equal amount e. Through the correspondence principle, the Hamiltonian for this system is given by

$$\hat{H}_0 = -\frac{\hbar^2}{2m}\triangle - \frac{e^2}{4\pi\varepsilon_0 r}, \tag{2.1}$$

where the first term is the kinetic energy of a fictive particle of reduced mass $m = \frac{m_e m_p}{m_e + m_p}$ whereas the second term is the Coulomb potential which describes the attractive force between the opposite charges.

However, due to relativistic effects this Hamiltonian is not complete. Additional terms arise which will be discussed later in this section and will be included as perturbations to the Hamiltonian presented in Equation (2.1). The unperturbed energy levels of the hydrogen atom can be obtained by solving the time-independent Schrödinger equation

$$\hat{H}\Psi = E\Psi. \tag{2.2}$$

It is convenient to choose a spherical coordinate system in order to solve this partial differential equation. The Laplacian in spherical coordinates is given by

$$\triangle = \frac{\partial^2}{\partial r^2} + \frac{1}{r^2}\frac{\partial^2}{\partial \theta^2} + \frac{1}{r^2 \sin^2\theta}\frac{\partial^2}{\partial \phi^2} + \frac{2}{r}\frac{\partial}{\partial r} + \frac{\cos\theta}{r^2 \sin\theta}\frac{\partial}{\partial \theta}. \tag{2.3}$$

Hence, the momentum operator can be split into a radial part and an angular part

$$\hat{p}^2 = \hbar^2 \triangle = \hat{p}_r^2 + \frac{\hat{l}^2}{r^2}, \tag{2.4}$$

where $\hbar = \frac{h}{2\pi}$ with $h \approx 6.626068 \times 10^{-34}$Js [15] being the Planck constant. The radial momentum $\hat{p}_r$ is given by

$$\hat{p}_r = \frac{\hbar}{i}\left(\frac{\partial}{\partial r} + \frac{1}{r}\right) \tag{2.5}$$

and the orbital angular momentum $\hat{\vec{l}}$

$$\hat{\vec{l}} = \frac{\hbar}{i}\left(\vec{r} \times \vec{\nabla}\right). \tag{2.6}$$

In the following ansatz, the solution of (2.2) is separated into an angle-dependent part $Y(\theta,\phi)$ and a radius-dependent part χ

$$\Psi(r,\theta,\phi) = Y(\theta,\phi)\chi(r), \tag{2.7}$$

where the angle-dependent part $Y(\theta,\phi)$ is chosen to be an eigenfunction of the square of the orbital angular momentum operator $\hat{\vec{l}}^2$. Since $\hat{\vec{l}}^2$ commutes with each component of $\hat{\vec{l}}$, the angle-dependent part $Y(\theta,\phi)$ should also be an eigenfunction of $\hat{l}_z$. Functions which fulfil this criterion are the spherical harmonics $Y_l^{m_l}(\theta,\phi)$. The corresponding eigenvalue equations are given by

$$\hat{\vec{l}}^2 Y_l^{m_l}(\theta,\phi) = l(l+1)\hbar^2 Y_l^{m_l}(\theta,\phi) \tag{2.8}$$

and

$$\hat{l}_z Y_l^{m_l}(\theta,\phi) = m_l \hbar Y_l^{m_l}(\theta,\phi), \tag{2.9}$$

where $l = 0,1,2,...,\infty$ and $m_l = -l,-l+1,...,+l$.

By substituting the ansatz (2.7) into the time-independent Schrödinger equation (2.2) and using the relations (2.4) and (2.8), the radial equation

$$\left[\frac{\hat{p}_r^2}{2m} + \frac{l(l+1)\hbar^2}{2mr^2} - \frac{e^2}{4\pi\varepsilon_0 r} - E\right]\chi_l(r) = 0 \tag{2.10}$$

is obtained with $\hat{p}_r^2 = -\hbar^2 \frac{1}{r}\frac{d^2}{dr^2} r$.

A complete treatment of this ordinary differential equation is given in the book “Quantum Mechanics” by A. Messiah [16] in chapter 11, paragraph 4 on the pages 415 and 416. For the solutions to be finite at the origin as well as in the limit $r \longrightarrow \infty$, the following condition on the energy must be fulfilled

$$E_n = -\frac{1}{(4\pi\varepsilon_0)^2}\frac{e^4 m}{2\hbar^2 n^2} \tag{2.11}$$

with $n = 1,2,...,\infty$.

Equation (2.11) describes the discrete energy levels of the hydrogen atom without their fine and hyperfine structure as depicted in Figure 2.1. n is also called the “principal quantum number”. As n goes to infinity, the spacing between two successive

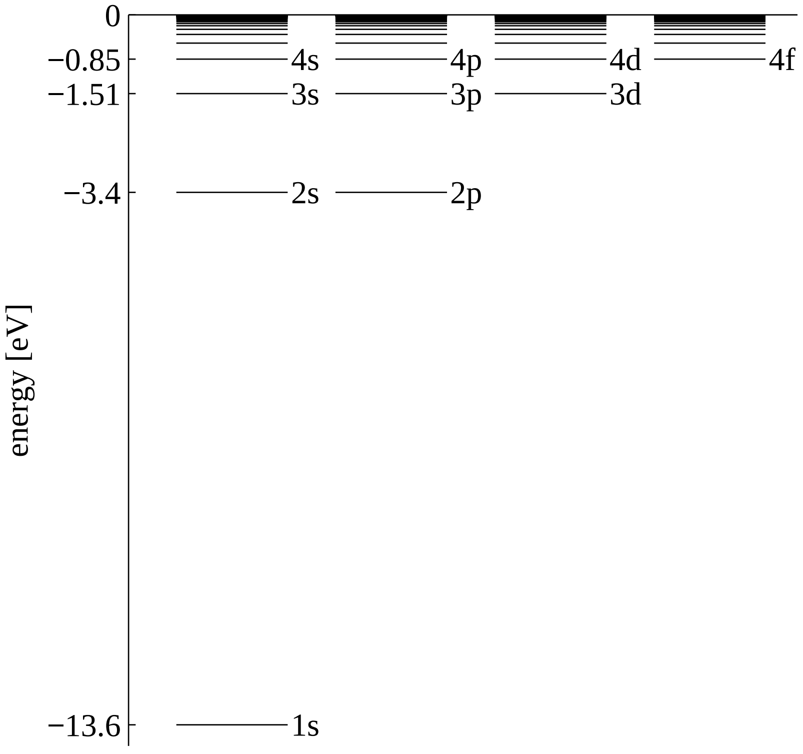


Figure 2.1: Energy levels of the hydrogen atom without fine- and hyperfine structure; The values of the azimuthal quantum number l are denoted by letters where $s \rightarrow l = 0$, $p \rightarrow l = 1$, $d \rightarrow l = 2$, $f \rightarrow l = 3$.

levels becomes smaller and approaches a continuous spectrum in the limit. Particularly important for this discussion is to mention the degeneracy of these energies. For a given energy E_n there exist n^2 different solutions Ψ_{n,l,m_l} to Equation (2.2), each of which corresponding to a given orbital angular momentum with $l = 0, 1, 2, ..., n-1$ and $m_l = -l, -l+1, ..., +l$. But as will be seen later, this is only an *accidental* degeneracy due to the incompleteness of the Hamiltonian (2.1).

2.3 Electron and proton spin

Historic experiments (e.g. the Stern-Gerlach experiment [17]) have shown the necessity to introduce a new property of the electron called its "spin". Moreover the existence of the spin of the electron is predicted by the relativistic theory of Dirac (see for instance [18]).

In the non-relativistic quantum theory the electron spin can be introduced as an intrinsic angular momentum $\hbar\hat{\vec{s}}$ where $\hat{\vec{s}}$ is the electron spin operator. In analogy to the orbital angular momentum l, which was defined in the previous section (Equations (2.8) and (2.9)), the eigenfunctions of the spin operator are defined by the eigenvalue equations

$$\hat{\vec{s}}^2\Psi = s(s+1)\Psi \tag{2.12}$$

and

$$\hat{s}_z\Psi = m_s\Psi, \tag{2.13}$$

with $s = \frac{1}{2}$ and $m_s = \pm\frac{1}{2}$. Note that in this definition $\hbar$ is not contained in the spin operator $\hat{s}$ because the Planck constant is part of the definition of the Bohr magneton μ_B which occurs in the following relation that associates the electron spin with a magnetic moment

$$\hat{\vec{\mu}}_s = g_e \underbrace{\frac{e\hbar}{2m_e}}_{\mu_B} \hat{\vec{s}}, \tag{2.14}$$

where $g_e \approx -2.002319$ [15] is the electron spin g-factor. It has a negative sign due to the negative charge of the electron.

In the following sections the basis $\{|s, m_s\rangle\}$ of eigenfunctions of the spin operator will be used with the shorthand notation

$$|\frac{1}{2}, +\frac{1}{2}\rangle =: |+\rangle \tag{2.15}$$

and

$$|\frac{1}{2}, -\frac{1}{2}\rangle =: |-\rangle. \tag{2.16}$$

In this basis the components of $\hat{\vec{s}}$ are proportional to the Pauli-matrices

$$\hat{s}_x = \frac{1}{2}\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}, \quad \hat{s}_y = \frac{1}{2}\begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix} \quad \text{and } \hat{s}_z = \frac{1}{2}\begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}. \tag{2.17}$$

Not only the electron has a spin $\frac{1}{2}$ but also the proton. Let $\hat{\vec{I}}$ denote the spin operator of the proton. It has the same properties (2.12),(2.13) and (2.17) as the electron spin s. The corresponding magnetic moment is given by

$$\hat{\vec{\mu}}_I = g_p \underbrace{\frac{e\hbar}{2m_p}}_{\mu_N} \hat{\vec{I}}, \tag{2.18}$$

where g_p is the proton g-factor and μ_N is the nuclear magneton.

2.4 Hyperfine structure of the ground state

Since both the electron and the proton have a spin $\frac{1}{2}$ and both of these are associated to a magnetic moment there is an interaction between these two angular momentums. The corresponding splitting of the energy levels is called hyperfine structure. In the ground state ($n = 1$, $l = 0$, $m_l = 0$), this interaction is described by the Hamiltonian [19, 20]

$$\hat{H}_{hf} = A(\hat{\vec{s}} \cdot \hat{\vec{I}}) = A(\hat{s}_x \times \hat{I}_x + \hat{s}_y \times \hat{I}_y + \hat{s}_z \times \hat{I}_z), \tag{2.19}$$

where A is the hyperfine coupling constant.

In the basis of the combined system $\{|s,m_s\rangle \times |I,m_I\rangle\}$, this Hamiltonian has a simple representation which can be obtained by applying the operator $\hat{H}_{hf}$ given by Equation (2.19) onto the basis states:

$$\begin{aligned}
\hat{H}_{hf}|{+}{+}\rangle &= \frac{1}{4}A(|{-}{-}\rangle - |{-}{-}\rangle + |{+}{+}\rangle) = \frac{1}{4}A|{+}{+}\rangle \\
\hat{H}_{hf}|{+}{-}\rangle &= \frac{1}{4}A(|{-}{+}\rangle + |{-}{+}\rangle - |{+}{-}\rangle) = \frac{1}{4}A(2|{-}{+}\rangle - |{+}{-}\rangle) \\
\hat{H}_{hf}|{-}{+}\rangle &= \frac{1}{4}A(|{+}{-}\rangle + |{+}{-}\rangle - |{-}{+}\rangle) = \frac{1}{4}A(2|{+}{-}\rangle - |{-}{+}\rangle) \\
\hat{H}_{hf}|{-}{-}\rangle &= \frac{1}{4}A(|{+}{+}\rangle - |{+}{+}\rangle + |{-}{-}\rangle) = \frac{1}{4}A|{-}{-}\rangle.
\end{aligned}$$

This shows that the corresponding matrix is

$$\hat{H}_{hf} = \frac{1}{4}\begin{pmatrix} A & 0 & 0 & 0 \\ 0 & -A & 2A & 0 \\ 0 & 2A & -A & 0 \\ 0 & 0 & 0 & A \end{pmatrix}. \tag{2.20}$$

In the next section, this Hamiltonian is combined with interaction terms coming from an external magnetic field in order to quantify the splitting as a function of the absolute value of the external magnetic field $\vec{B}$.

2.5 Zeeman effect of the hyperfine structure

If the hydrogen atom is exposed to a homogeneous external magnetic field $\vec{B}$, the Hamiltonian becomes

$$\hat{H} = \hat{H}_0 + \hat{H}_{hf} \underbrace{-(g_e\mu_B\hat{\vec{s}} + g_p\mu_N\hat{\vec{I}}) \cdot \vec{B}}_{\hat{H}'}. \tag{2.21}$$

It is convenient to choose the direction of the B-field as the z-axis of the coordinate system so that $\hat{H}'$ becomes

$$\hat{H}' = -B(g_e\mu_B\hat{s}_z + g_p\mu_N\hat{I}_z). \tag{2.22}$$

This Hamiltonian has the following effect on the basis states $\{|s,m_s\rangle \times |I,m_I\rangle\}$

$$\begin{aligned}
\hat{H}'|{+}{+}\rangle &= -\frac{1}{2}B(g_e\mu_B + g_p\mu_p)|{+}{+}\rangle \\
\hat{H}'|{+}{-}\rangle &= -\frac{1}{2}B(g_e\mu_B - g_p\mu_p)|{+}{-}\rangle \\
\hat{H}'|{-}{+}\rangle &= -\frac{1}{2}B(-g_e\mu_B + g_p\mu_p)|{-}{+}\rangle \\
\hat{H}'|{-}{-}\rangle &= \frac{1}{2}B(g_e\mu_B + g_p\mu_p)|{-}{-}\rangle.
\end{aligned}$$

Note that the matrix representation of $\hat{H}'$ is diagonal.

Including the hyperfine structure and neglecting couplings between states of different n values, the perturbation $\hat{H}'' := \hat{H}_{hf} + \hat{H}'$ to the original Hamiltonian $\hat{H}_0$ is given by

$$\hat{H}'' = \frac{1}{4}\begin{pmatrix} A - 2B\mu_+ & 0 & 0 & 0 \\ 0 & -A - 2B\mu_- & 2A & 0 \\ 0 & 2A & -A + 2B\mu_- & 0 \\ 0 & 0 & 0 & A + 2B\mu_+ \end{pmatrix}, \tag{2.23}$$

where $\mu_+ := g_e\mu_B + g_p\mu_p$ and $\mu_- := g_e\mu_B - g_p\mu_p$. Thus, by first order perturbation theory the difference to the energy of the ground state given by the lowest eigenvalue of the unperturbed Hamiltonian $\hat{H}_0$ can be approximated by ΔE, being the solution to the eigenvalue problem

$$\hat{H}''\Psi = \Delta E \Psi. \tag{2.24}$$

The first two solutions can be obtained directly by choosing $\Psi_1 = |{+}{+}\rangle$ so that

$$\Delta E_1 = \frac{1}{4}(A - 2B(g_e\mu_B + g_p\mu_p)) \tag{2.25}$$

and by choosing $\Psi_2 = |{-}{-}\rangle$ so that

$$\Delta E_2 = \frac{1}{4}(A + 2B(g_e\mu_B + g_p\mu_p)). \tag{2.26}$$

In order to obtain the remaining two eigenvalues, the eigenvalue equation of the submatrix must be determined

$$\begin{pmatrix} -A - 2B\mu_- & 2A \\ 2A & -A + 2B\mu_- \end{pmatrix}\begin{pmatrix} \alpha \\ \beta \end{pmatrix} = 4\Delta E \begin{pmatrix} \alpha \\ \beta \end{pmatrix}. \tag{2.27}$$

The corresponding characteristic equation is

$$\begin{aligned} 0 &= (-A - 2B\mu_- - 4\Delta E)(-A + 2B\mu_- - 4\Delta E) - 4A^2 \\ &= (A + 4\Delta E)^2 - 4(\mu_-)^2B^2 - 4A^2, \end{aligned}$$

which is equivalent to

$$A + 4\Delta E = \pm 2\sqrt{(\mu_-)^2B^2 + A^2}.$$

Hence, the remaining two eigenvalues are

$$\Delta E_3 = \frac{1}{4}(+2\sqrt{(g_e\mu_B - g_p\mu_p)^2B^2 + A^2} - A) \tag{2.28}$$

and

$$\Delta E_4 = \frac{1}{4}(-2\sqrt{(g_e\mu_B - g_p\mu_p)^2B^2 + A^2} - A). \tag{2.29}$$

These eigenvalues both belong to states of the form

$$\psi = c_1|{+}{-}\rangle + c_2|{-}{+}\rangle, \tag{2.30}$$

for which the quantum number that belongs to the projection of the total angular momentum $\hat{F} = \hat{s} + \hat{I}$ onto the z-axis denoted by M_F is zero (see Figure 2.2). The states $|{+}{+}\rangle$ and $|{-}{-}\rangle$, which belong to the energy eigenvalues E_1 and E_2, both belong to $M_F = \pm 1$.

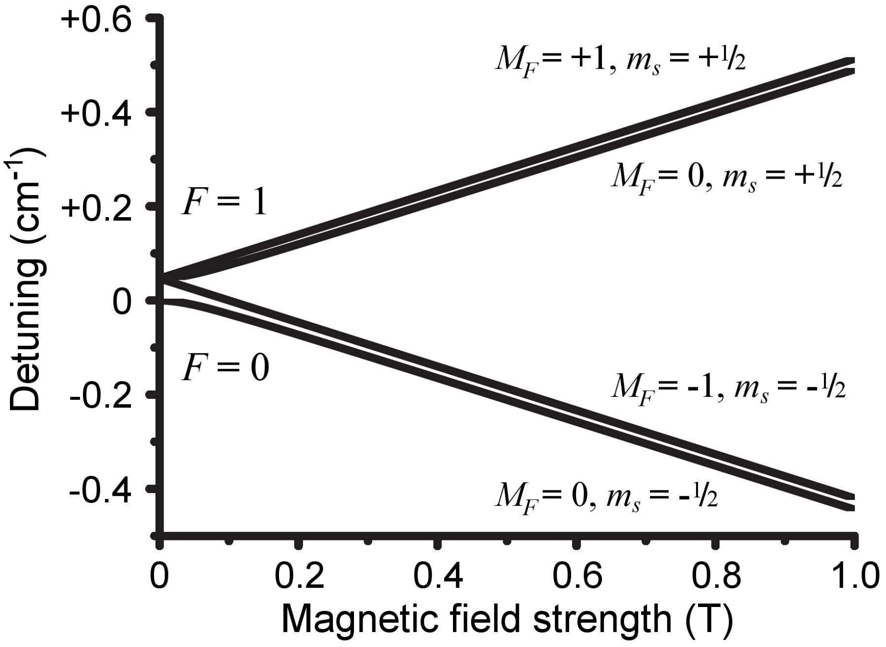


Figure 2.2: The Zeeman-effect of the hydrogen atom in its ground state (taken from [6]). F denotes the quantum number of the total angular momentum $\hat{F} = \hat{s} + \hat{I}$, where $\hat{s}$ denotes the electron spin an $\hat{I}$ denotes the proton spin.

Chapter 3

Zeeman deceleration

3.1 Overview

Zeeman deceleration can be applied to control the translational motion of beams of neutral atoms and molecules that have an unpaired electron and thus a magnetic dipole moment. Atoms or molecules that are in a state where the internal energy increases when the strength of the external magnetic field increases, will be decelerated since this energy will be taken from the kinetic energy of the particles. By switching off the external magnetic field, the internal energy is removed from the particles. Figure 3.1 shows a schematic view of the type of Zeeman decelerator considered in this work.

The electric analogue of multistage Zeeman deceleration is multistage Stark deceleration. Hence, similar procedures can be applied for the deceleration of suitable species using either the Zeeman or the Stark effect. However, multistage Stark deceleration is complementary to Zeeman-deceleration. With Zeeman deceleration it is possible to decelerate paramagnetic atoms and molecules that do not necessarily have an electric dipole moment in their ground state while Stark deceleration can be used to decelerate polar molecules that do not necessarily have a magnetic moment.

3.2 The principle of multistage Zeeman deceleration

As an atom or molecule which exhibits a Zeeman effect moves into a quasi-static magnetic field whose strength has a positive or negative gradient, its internal energy either increases or decreases. This is compensated by a loss or gain of kinetic energy. The corresponding force is

$$F(\vec{x}) = -\vec{\nabla}E(|\vec{B}(\vec{x})|) = -\frac{dE}{d|\vec{B}|}(\vec{x})\ \vec{\nabla}|\vec{B}|(\vec{x}), \tag{3.1}$$

where E is the internal energy of the particle which is a function of $|\vec{B}|$ as a result of the Zeeman-effect. This equation reveals that the internal energy E can be seen as a potential energy. If an atom is in a state where the internal energy is increased when the magnitude of the external B-field increases as the upper two states in Figure 2.5, it will be decelerated (accelerated) if it moves into a region of space where the B-field increases (decreases). The particle is referred to as *low-field-seeking* (*high-field-seeking*) particle. This work primarily considers low-field-seeking particles since the characteristics of the magnetic fields generated by the coils that are used offer significantly greater phase-space acceptance than low-field-seeking particles (see Section 3.4).

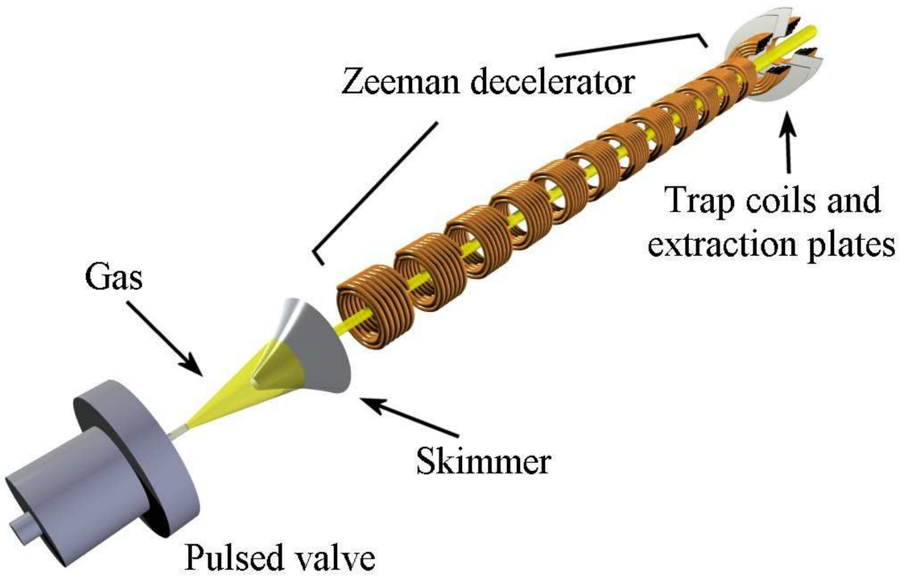


Figure 3.1: Schematic representation of the Zeeman-decelerator and trap which is considered in the present work.

A purely static magnetic field cannot be used to remove energy from a particle, since as soon a particle reaches a saddle-point of the magnitude of the B-field, it will be accelerated again. It can happen that a particle will be reflected by the inhomogeneous field, which means that it is decelerated to zero velocity and then, changing the direction of motion, is accelerated again. The key to the operation of a multistage decelerator as that depicted in Figure 2.5 is to switch off the source of the magnetic field (e.g. the coils) before the time at which the particle is accelerated again. As a reaction to the fall of the current in the coils, the particle will induce a small current into the coils and hence loose some of its total energy to the coils. The most energy can be removed from a single particle by first keeping the magnetic field static and letting the particle move from a region with very low magnitude of the magnetic field to the saddle-point and switching the field off before the particle experiences a negative gradient of the absolute B-field value and accelerates again. By repeating many deceleration steps one after the other, large amounts of kinetic energy can be removed.

3.3 Supersonic expansion

Zeeman deceleration is typically applied to atoms and molecules in a supersonic gas jet that expands into a vacuum. Supersonic means that the laboratory-frame velocity of the particles in the jet, is higher than the local speed of sound. Because of the low particle density in the (imperfect) vacuum, the speed of sound is much lower than in air. The mean velocity of the particles in the jet points into a constant direction which defines the translational or longitudinal axis often denoted by z. To span the planes perpendicular to that axis, two perpendicular axes are defined usually denoted by x for the horizontal axis and y for the vertical axis. Motion in those planes is often referred to as radial motion.

The source of the jet is a nozzle. Within the reservoir of the nozzle, the velocities of the particles are Maxwell-Boltzmann distributed at a temperature T_0. As the particles pass the nozzle this distribution changes to [21] [1]

$$f(v) = N_n v^2 e^{-(v-v_{\text{exp}}(L))^2/\xi(L)^2}, \tag{3.2}$$

where L is the distance from the nozzle, $\xi(L) = (2k_BT(L)/m)^{1/2}$, N_n represents a normalization factor ensuring that $\int_0^{\text{inf}} f(v)dv = 1$, and γ stands for the ratio of heat capacities of the gas species $\gamma = C_p/C_v$ which is for monatomic gases equal to $5/3$. The quantity $v_{\text{exp}}(L)$ can be interpreted as the flow velocity at a distance L from the nozzle and is given by

$$v_{\text{exp}}(L) = M(L)\sqrt{\gamma k_B T(L)/m}. \tag{3.3}$$

The Mach number $M(L)$ represents the ratio of the jet velocity in the laboratory frame to the local speed of sound. The translational temperature $T(L)$ decreases with L because of the pressure difference at the nozzle orifice, which has the effect that particles coming from the nozzle collide with particles that have a translational velocity which is low or does not point into the direction of the expansion. This results in a transfer from random thermal motion to kinetic energy of a directed mass flow. As a result, the Mach number increases. This can be formalized by looking at the energy conservation and using the specific heat at constant pressure of the gas sample C_p

$$C_pT_0 = C_pT(L) + \frac{1}{2}mv_{\text{exp}}^2. \tag{3.4}$$

When the local Mach number becomes significantly different from zero, the velocity distribution is no longer of the Maxwell-Boltzmann kind but approaches a Gaussian, with the most probable speed being equal to the mean speed of the particles in the jet.

Away from the nozzle, the collision rate is substantially reduced as the density of particles decreases, which at some point causes the Mach number to be frozen at a constant terminal Mach number

$$M_T = \lim_{L\to\infty} M(L). \tag{3.5}$$

This terminal Mach number can be given in terms of the temperature of the beam after the expansion by inserting the dependence of v_{exp} on the Mach number (3.3) into the energy conservation (3.4) at an infinite distance from the nozzle

$$C_pT_0 = C_pT(\infty) + \frac{1}{2}M_T^2\gamma k_B T(\infty), \tag{3.6}$$

and solving it for the terminal Mach number

$$M_T = \sqrt{\frac{2C_p(T_0 - T(\infty))}{\gamma k_B T(\infty)}}. \tag{3.7}$$

This shows that also the temperature of the beam has to remain constant after the expansion, if there is no further influence. Hence v_{exp} also remains constant, and the velocity distribution $f(v)$ (3.2) becomes a Gaussian with mean square

$$\mu = v_{\text{exp}}^2 = 2C_p(T_0 - T(\infty))/m \tag{3.8}$$

and standard deviation

$$\sigma = \frac{\xi(\infty)^2}{2} = \frac{k_BT(\infty)}{m}. \tag{3.9}$$

[1]This discussion of the expansion of supersonic beams is mainly inspired by chapter 1.5 of the Ph.D. thesis of T. A. Paul [22].

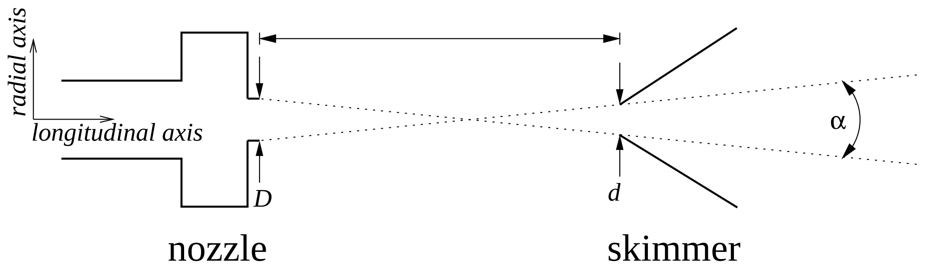


Figure 3.2: An arrangement of a nozzle and a skimmer as used in multistage Zeeman deceleration experiments. The dotted line represents the bounds for the radial position of the particles after they have passed the skimmer.

The simulation described in Chapter 4 starts with the situation after this terminal Mach number has established. Because of the low density of particles in the jet and because of the decreased translational temperature, collisions are so rare that they can be neglected. Each particle moves along a straight line as long as there is no radial acceleration.

At that stage a skimmer is usually applied which imposes an upper limit on the radial velocity. After the skimmer, assuming a collision-free model, the particles are all contained in a cone with an opening angle α which is also called divergence angle (see Figure 3.2). The divergence angle is given by

$$\alpha = 2\arctan\left(\frac{D+d}{2x}\right), \tag{3.10}$$

where D is the diameter of the skimmer opening, d is the diameter of the nozzle orifice and x is the distance from the nozzle orifice to the skimmer opening.

Because there are only few collisions after the Mach number has reached a constant value, the behaviour of the jet can to some extent be described by the rules of classical optics [2]. In this point of view the beam defocuses after having left the nozzle. For many applications, it is necessary to refocus the beams.

3.4 Magnetic field of the solenoids

In cylindrical coordinates (z, r, ϕ), the exact expressions for the longitudinal and radial components of the B-field generated by a single loop of radius R perpendicular to the longitudinal axis and centred at $(z = A, r = 0)$ are given by [10]

$$B_z = \frac{\mu I}{2\pi}\frac{1}{[(R+r)^2+(z-A)^2]^{1/2}}\left[K(k^2)+\frac{R^2-r^2-(z-A)^2}{(R-r)^2+(z-A)^2}E(k^2)\right] \tag{3.11}$$

and

$$B_r = \frac{\mu I}{2\pi r}\frac{z-A}{[(R+r)^2+(z-A)^2]^{1/2}}\left[K(k^2)+\frac{R^2-r^2+(z-A)^2}{(R-r)^2+(z-A)^2}E(k^2)\right], \tag{3.12}$$

where I is the current in the loop, μ is the magnetic permeability of the medium [3], $K(k^2)$ is the complete elliptic integral of the first kind and $E(k^2)$ is the complete elliptic

[2]The main difference to the optics of electromagnetic waves is that atomic and molecular optics can be non-conservative i.e. that the phase-space density can change.

[3]Usually it is assumed that the medium is a perfect vacuum, i.e. $\mu \approx \mu_0$.

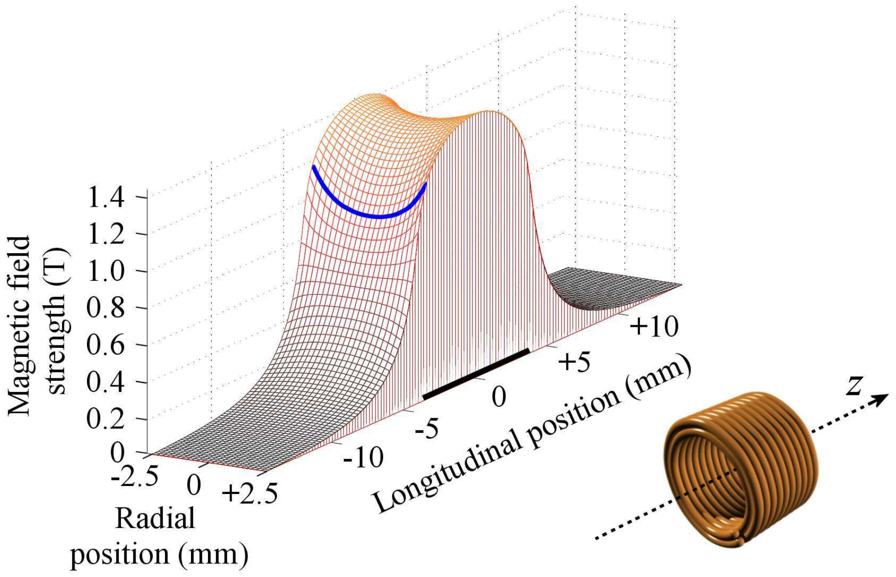


Figure 3.3: The absolute value of the magnetic field inside a single decelerator coil with 42 Windings on two layers. The inner diameter of this particular coil is 5mm and the diameter of the wire amounts to 300μm. The displayed magnetic field strength corresponds to a current of 250A.

integral of the second kind [4]. The argument k^2 to these functions is given by

$$k^2 = \frac{4Rr}{(R+r)^2+(z-A)^2}. \tag{3.13}$$

The angular component B_ϕ vanishes because of the cylindrical symmetry of the solenoid

$$B_\phi = 0. \tag{3.14}$$

In order to compute the components of the B-field from several current loops in a Helmholtz configuration, one simply adds for each loop the components $B_z(z,r)$ and $B_r(z,r)$ independently. Each summand is given by the expressions (3.11) and (3.12) at the corresponding position A along the z-axis, the radius R and current I for each loop.

On the longitudinal axis, the magnitude of the B-field of a solenoid increases towards the centre. In radial directions, the magnitude increases towards the wires. As seen from Figure 3.3, the magnitude of the B-field inside the coils has a convex nature in the radial dimensions. This property is particularly useful for the low-field-seeking particles used in deceleration experiments because they will be accelerated towards the translational axis if they are off the axis. This allows to collimate, guide or focus a beam of such particles.

3.5 Pulse sequence

As explained in Section 3.2, important parameters of a Zeeman decelerator are the points in time at which the coils are switched on and off. The magnetic field pulses

[4]There are polynomial approximations to the complete elliptic integrals such as those given in [23]. Mathematical software packages like Mathematica, Maple and MATLAB offer routines to calculate these functions.

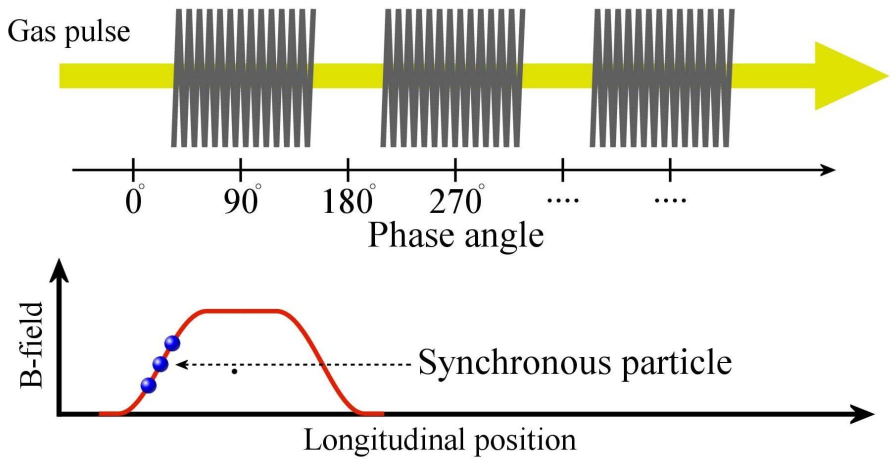


Figure 3.4: Definition of the global phase angle.

can be specified by defining so-called phase angles: If all solenoids in a decelerator are switched on with the same direction of current, the magnitude of the B-field oscillates periodically along the decelerator axis. The phase of this oscillation can be described by the so-called phase angle. This angle is usually defined locally to be zero in the middle of the gap between two solenoids and 90 degrees in the centre of solenoids.

Global phase angles, as shown in Figure 3.4, can be defined by letting the angle to be zero in the middle of the gap between a virtual "zeroth" solenoid and the first solenoid in the Helmholtz configuration. As for the local phase angles, a global phase angle of 90 degrees corresponds to the centre of the first coil. The global phase angle does not start again at zero after each period but increase linearly. Hence, the global phase angle is nothing else than a spatial coordinate normalized to the dimension of the decelerator stages.

The phase angles can be mapped to points in time by considering the motion of a reference particle on the translational axis with zero radial velocity. Switching the coils on and off at a certain phase angle means that the coils are switched on and off when the reference particle is at the position corresponding to this phase angle.

The reference particle is usually chosen to be at the mean position and velocity of a certain bunch of particles that are to be decelerated, and is called the "synchronous particle" (see Figure 3.4). If this particle is decelerated synchronously to the other particles of the bunch, which means that the whole bunch is not separated too much from the synchronous particle as it propagates through the decelerator, then this bunch of particles is called "phase stable". One of the biggest challenges of Zeeman deceleration is to find a pulse sequence which preserves phase stability for a bunch of particles that is as large as possible and, at the same time, removes as much kinetic energy as possible from these particles.

Whenever a solenoid is switched off, the current returns to zero. As a result the particles lose their quantization axis and can undergo a change of their spin state (i.e. a spin-flip). These so-called Majorana transitions result in a loss of particles. Therefore, before a certain solenoid is to be switched off, the succeeding coil should be switched on with the same direction of current to assure that the external B-field at the position of the particles never approaches zero where it can change its direction rapidly.

3.6 Magnetic trap

The type of trap considered in the present work is a two-coil quadrupole trap which is shown schematically in Figure 3.5 (a). The two solenoids A and B are pulsed with equal but opposed currents. Thus, the magnetic fields of the two solenoids cancel in centre between the solenoids. At that position the magnitude of the B-field has a local minimum as can be seen from the field map on Figure 3.5 (b). This local minimum allows to trap particles [5].

For the calculation of the components of the B-field, the same expressions (3.11) and (3.12) as for the solenoids of the decelerator can be used (see Section 3.4). The trap field is obtained by independently adding the longitudinal and radial components from solenoid A with current $-I$ and solenoid B with current $+I$. Close to the centre of the trap, the magnitude of the resulting field seen while moving on a straight line towards the centre decreases almost linearly [6]. In calculating classical particle motion in a trap with the potential depicted in Figure 3.5 (b), the centre of the trap is a singular point since the force (3.1) abruptly changes the direction there. At that point Majorana transitions are possible since the orientation of the atomic magnetic moment relative to the external field direction changes instantaneously and because the external field goes to zero [7] (or to the low magnitude of the Earth's magnetic field). The magnitude of the trap field has saddle points in both longitudinal and radial direction. Usually the radial saddle point corresponds to a lower value than the longitudinal saddle-point.

Loading the trap is carried out by pulsing the solenoids in three separate steps as depicted in Figure 3.6. First, only solenoid A is pulsed with the same direction of the current as the last decelerator coil, so that it acts as an additional coil of the decelerator. The pulsing is done with the usual overlap to the pulse of the last decelerator coil. Second, solenoid B is switched on just before Solenoid A is switched off and again with the same direction of the current. If enough energy has been removed from the decelerated bunch, the particles will be reflected by the field of solenoid B. Finally, in order to prevent the particles from flying back to the decelerator, solenoid A is switched on again with equal but opposed current to solenoid B (third step). At this point, the trap field is established and the particles do not have enough energy to traverse the corresponding potential hills.

The laser beams indicated in Figure 3.5 are used to ionize the trapped atoms or molecules. These ions are then accelerated towards a microchannel plate (MCP) detector by a pulsed electric potential applied to the repeller plates which are located between the two trap solenoids. The ions are detected when they hit the MCP.

[5]It can be proven that it is not possible for an electric or magnetic field to have a local maximum at a point where no charge or current [24] is located. Thus, it is not possible to trap high-field-seeking particles with a static magnetic field.

[6]Note that in some cases it might be more desirable to a have a quadratic (harmonic) behaviour. This can be achieved by a three-coil spherical hexapole trap as described in [10].

[7]There are magnetic trap configurations for which the magnitude of the B-field is not zero at the local minimum. Examples are the Ioffe trap and the baseball trap which are also described in [10]

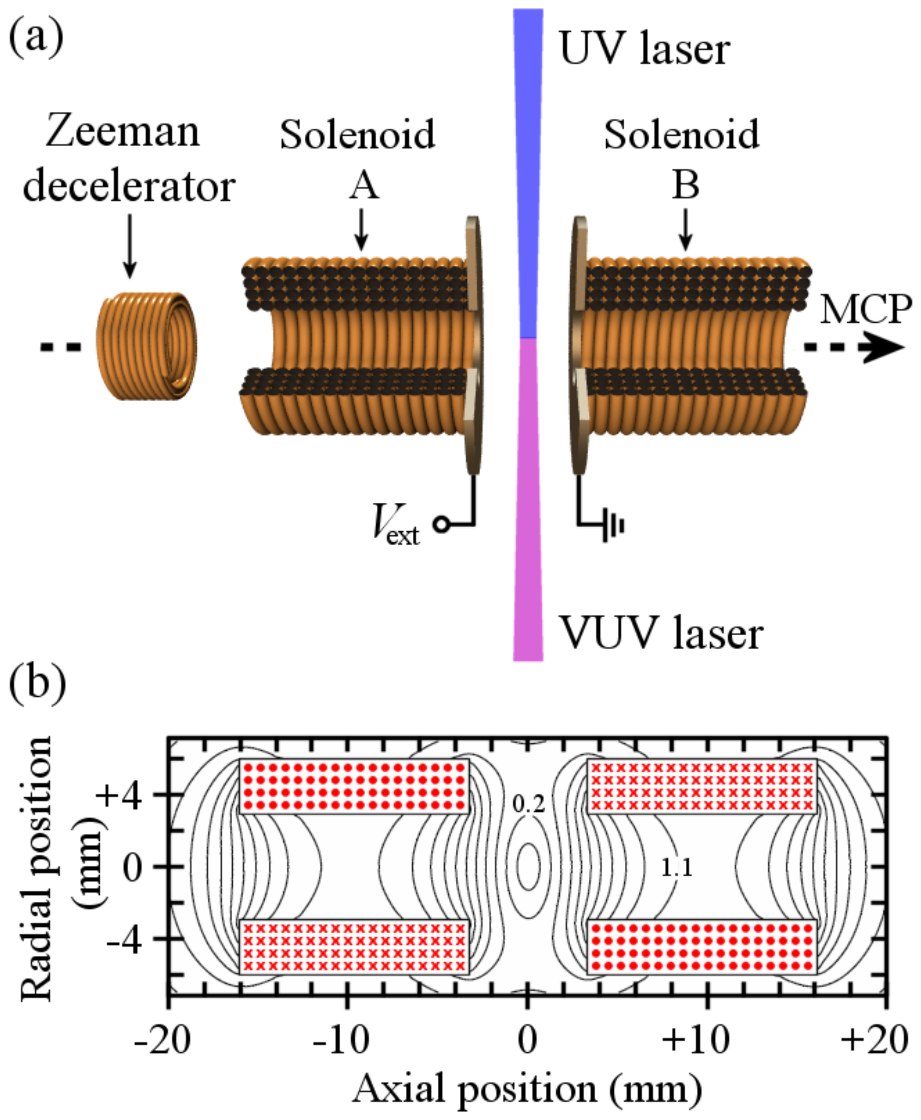


Figure 3.5: (a) Two-coil quadrupole trap at the end of the decelerator. Each coil has 72 windings on 4 layers. The inner diameter is 6mm and the diameter of the wire amounts to 700μm. The gap distance between the two coils is 6.6mm. (b) Contour lines along which the absolute value of the B-field is constant at an operation current of 200A. The labels denote the magnetic field strength in units of Tesla.

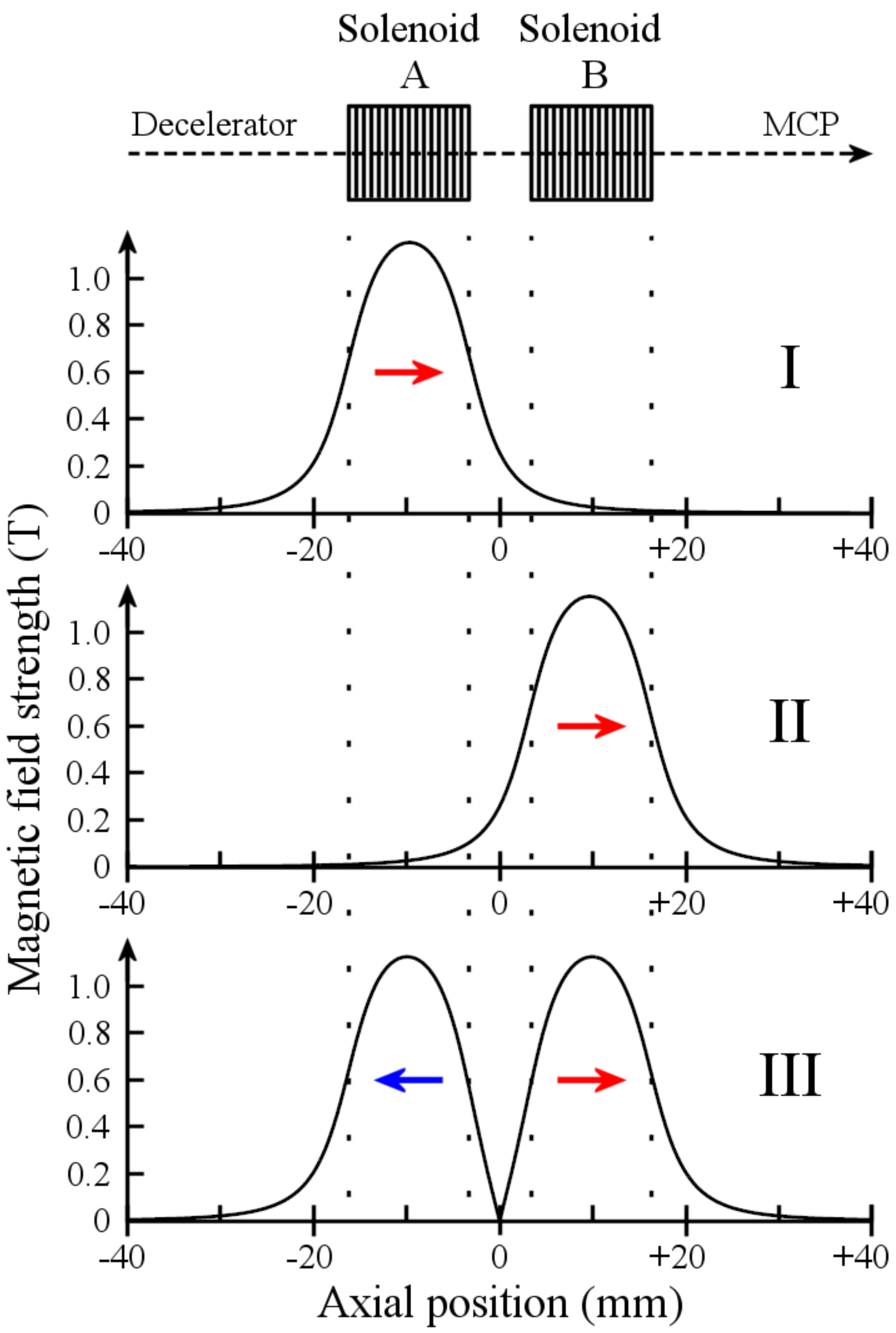


Figure 3.6: The process of loading the trap in three steps. The arrows represent schematically the direction of the current in the solenoids. The magnetic fields correspond to a current of 200A and trap solenoids made of 72 windings on 4 layers where the inner diameter amounts to 6mm.

Chapter 4

Simulation

4.1 Computational model

The goals of the simulation are to determine the operational characteristics of the multistage Zeeman decelerator and to evaluate the objective functions which are described later in Section 5.1. Since the de Broglie wavelength of the hydrogen atoms is very small and because collisions in the gas jet are rare, this can be done by calculating the trajectories according to the classical Newton equation of motion while including the force due to the Zeeman effect (see Section 3.2) and neglecting the collisions. Also the collisions with the background gas in the imperfect vacuum are neglected. This model has been experimentally verified in previous works on Zeeman deceleration and trapping of hydrogen atoms [6, 8] .

As initial condition, a sample of gas particles is taken, for which the particle positions are uniformly distributed within a certain volume and their velocities obey a Gaussian distribution with mean and standard deviation depending on the temperature according to Equations (3.8) and (3.9) in Section 3.3.

4.2 Integration of the equation of motion

To solve Newton's equation of motion, three different schemes of numerical integration have been evaluated: the symplectic Euler method, the Verlet method and the leap frog method. These methods are frequently used in Molecular Dynamics because they are symplectic and time reversible up to rounding errors. Because of that, numerical dissipation is limited, i.e. the energy and phase space density are approximately conserved [25, 26].

The Verlet method can be obtained by adding the two Taylor expansions:

$$\vec{x}(t+\Delta t) = \vec{x}(t) + \Delta t \dot{\vec{x}}(t) + \frac{\Delta t^2}{2}\ddot{\vec{x}}(t) + \frac{\Delta t^3}{6}\dddot{\vec{x}}(t) + \mathcal{O}(\Delta t^4) \tag{4.1}$$

$$\vec{x}(t-\Delta t) = \vec{x}(t) - \Delta t \dot{\vec{x}}(t) + \frac{\Delta t^2}{2}\ddot{\vec{x}}(t) - \frac{\Delta t^3}{6}\dddot{\vec{x}}(t) + \mathcal{O}(\Delta t^4) \tag{4.2}$$

by which one obtains

$$\vec{x}(t+\Delta t) + \vec{x}(t-\Delta t) = 2\vec{x}(t) + \Delta t^2 \ddot{\vec{x}}(t) + \mathcal{O}(\Delta t^4) \tag{4.3}$$

Thus, the time stepping is being done by [27]

$$\vec{x}(t+\Delta t) = 2\vec{x}(t) - \vec{x}(t-\Delta t) + \Delta t^2 \frac{\vec{F}(t,\vec{x}(t))}{m}, \tag{4.4}$$

where $\vec{F}(t,\vec{x}(t))$ is the force on the particle at time t and position $\vec{x}(t)$. Notice that the velocity is not calculated in this method. However, if needed, it can be evaluated using

$$\vec{v}(t) = \frac{\vec{x}(t+\Delta t) - \vec{x}(t-\Delta t)}{2\Delta t}. \tag{4.5}$$

The so called leap frog method, is obtained by subtracting two Taylor expansions of the velocities at half time steps

$$\vec{v}(t+\frac{\Delta t}{2}) = \vec{v}(t) + \frac{\Delta t}{2}\dot{\vec{v}}(t) + \frac{\Delta t^2}{8}\dddot{\vec{x}}(t) + \mathscr{O}(\Delta t^3) \tag{4.6}$$

$$\vec{v}(t-\frac{\Delta t}{2}) = \vec{v}(t) - \frac{\Delta t}{2}\dot{\vec{v}}(t) + \frac{\Delta t^2}{8}\dddot{\vec{x}}(t) + \mathscr{O}(\Delta t^3) \tag{4.7}$$

by which one obtains

$$\vec{v}(t+\frac{\Delta t}{2}) - \vec{v}(t-\frac{\Delta t}{2}) = \Delta t\dot{\vec{v}}(t) + \mathscr{O}(\Delta t^3) \tag{4.8}$$

and thus

$$\vec{v}(t+\frac{\Delta t}{2}) = \vec{v}(t-\frac{\Delta t}{2}) + \Delta t\dot{\vec{v}}(t). \tag{4.9}$$

By doing the same for the positions, but shifting the time by half a step forward, one obtains

$$\vec{x}(t+\Delta t) = \vec{x}(t) + \Delta t\vec{v}(t+\frac{\Delta t}{2}). \tag{4.10}$$

In this scheme one firsts calculates the new velocity at $t+\Delta t/2$ and then inserts it into (4.10). Because the errors in velocity and positions in every single time step are of third order, this scheme is a second order integration method.

When the shift in the time of the velocities by half a step is neglected, one obtains the symplectic Euler method:

$$\vec{v}(t+\Delta t) = \vec{v}(t) + \Delta t\frac{\vec{F}(t,\vec{x}(t))}{m} \tag{4.11}$$

$$\vec{x}(t+\Delta t) = \vec{x}(t) + \Delta t\vec{v}(t+\Delta t) \tag{4.12}$$

This is only a first order integration method. However, it has the advantage over the leap frog method that the velocities and positions are calculated at the same points in time.

4.3 Method of calculating the forces

To make the calculation of the force efficient, the magnetic fields have been precalculated. The transversal and longitudinal components of the B-field of a single solenoid has been calculated on an two-dimensional rectangular grid where one dimension is the longitudinal axis z and the other dimension the transversal axis r using the cylindrical symmetry of the decelerator. One of the two sides of each rectangle of the grid is parallel to the longitudinal axis and has length Δz whereas the other is parallel to the radial axis and has length Δr. The coordinates of the vertices are then given by

$$z_i := z_1 + (i-1)\Delta z, \qquad (i = 1,2,...,m), \tag{4.13}$$

$$r_j := r_1 + (j-1)\Delta r, \qquad (j = 1,2,...,n), \tag{4.14}$$

where (z_1, r_1) is the position of the vertex with the lowest z and r coordinate.

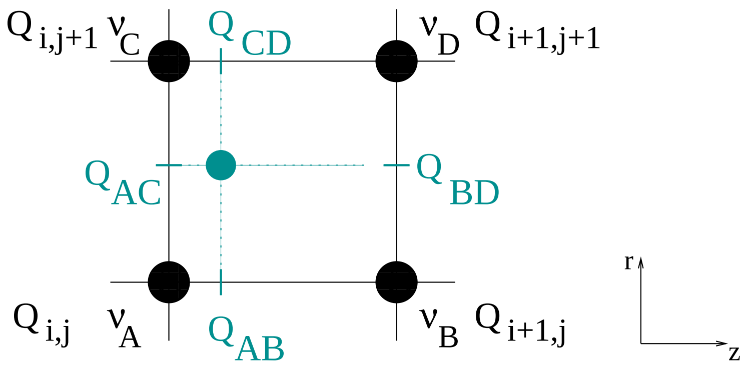


Figure 4.1: Method for approximating the gradient at a certain position by interpolation between the four neighbouring grid points.

The values of the longitudinal and transversal components of the field vectors at the grid points are then stored independently in the $m \times n$ matrices $\mathbf{B_z}$ and $\mathbf{B_r}$ respectively, such that for $i = 1...m$ and $j = 1...n$

$$(\mathbf{B_z})_{i,j} \equiv B_z(z_i, r_j), \tag{4.15}$$

$$(\mathbf{B_r})_{i,j} \equiv B_r(z_i, r_j), \tag{4.16}$$

where $\vec{B}(z,r) = (B_z(z,r), B_r(z,r), 0)$ is the magnetic field in cylindrical coordinates at position (z,r) (see Section 3.4).

The magnetic field at the particle's position (z,r) is then computed by linear interpolation of the values at the four vertices v_A, v_B, v_C, v_D around the position of the particle (see Figure 4.1). To find the entries in the matrices $\mathbf{B_z}$ and $\mathbf{B_r}$ that correspond to these grid points one can compute the indices

$$i = \left\lfloor \frac{z - z_1}{\Delta z} \right\rfloor, \tag{4.17}$$

$$j = \left\lfloor \frac{r - r_1}{\Delta r} \right\rfloor, \tag{4.18}$$

where the brackets $\lfloor \cdot \rfloor$ denote the floor function i.e. the largest integer which is smaller or equal to the real-valued argument. The coordinates of the vertices v_A, v_B, v_C, v_D are then given by $(z_i, r_j), (z_{i+1}, r_j), (z_i, r_{j+1}), (z_{i+1}, r_{j+1})$.

This allows to numerically calculate the gradient of a particular quantity Q stored in the $m \times n$ matrix $\mathbf{Q}$ (e.g. the longitudinal and transversal components of the B-field) by linearly interpolating the values between the four grid points:

$$Q_{AB} \approx \frac{(z - z_i)Q_{i+1,j} + (z_{i+1} - z)Q_{i,j}}{\Delta z}, \tag{4.19}$$

$$Q_{CD} \approx \frac{(z - z_i)Q_{i+1,j+1} + (z_{i+1} - z)Q_{i,j+1}}{\Delta z}, \tag{4.20}$$

$$Q_{AC} \approx \frac{(r - r_j)Q_{i,j+1} + (r_{j+1} - r)Q_{i,j}}{\Delta r}, \tag{4.21}$$

$$Q_{BD} \approx \frac{(r - r_j)Q_{i+1,j+1} + (r_{j+1} - r)Q_{i+1,j}}{\Delta r}. \tag{4.22}$$

Then to a first order approximation the gradient at the position (z,r) is given by:

$$\vec{\nabla}Q(z,r) \approx \begin{pmatrix} \frac{Q_{BD}-Q_{AC}}{\Delta z} \\ \frac{Q_{CD}-Q_{AB}}{\Delta r} \end{pmatrix}. \tag{4.23}$$

In this way one can compute $\vec{\nabla}B_z$ and $\vec{\nabla}B_r$. To compute the gradient of the field strength $|\vec{B}| = \sqrt{B_z^2 + B_r^2}$ from these values one can use the chain rule to obtain

$$\vec{\nabla}|\vec{B}| = \frac{B_z\vec{\nabla}B_z + B_r\vec{\nabla}B_r}{|\vec{B}|}. \tag{4.24}$$

In the simulation, it makes sense to define the grid position given by (z_1, r_1) relative to the centre of a single solenoid. In this way the two matrices that belong to the z and r components of the magnetic field of a certain solenoid have to be stored only once in the beginning of the simulation. For each active solenoid one can then transform the coordinates of a particle to the reference frame of the solenoid and go through the above procedure to compute the magnetic field at its position. However, it is important that the contributions to B_z and B_r as well as $\vec{\nabla}B_z$ and $\vec{\nabla}B_r$ of every active solenoid are summed up *before* computing $|\vec{B}|$ and $\vec{\nabla}|\vec{B}|$ (see also Section 6.2). Before the components of the field are summed up, they can be scaled by a factor which reflects the present current with which the solenoid is operated.

Finally, the force is calculated by evaluating the right hand side of Equation (3.1). The energy values at certain magnetic field strengths are stored in a one-dimensional equidistant grid. One can then use numerical differentiation to compute $dE/d|\vec{B}|$.

4.4 Program structure

Whilst the computational model described above has already been implemented in previous works, this implementation was not efficient enough for the optimization. Therefore it has been decided to implement the model with a new structure. The emphasis for this implementation was on extensibility and generality. Thus an object-oriented approach has been chosen. The programming language of choice was C++ because it implements the concepts of object-oriented programming and, at the same time, inherits the effectiveness of the C programming language [1].

Here the key concepts of the computer program that implements the simulation are presented. The basic structure is given in Figure 4.2. For a detailed documentation of each class the reader is referred to the Appendix C.

The heart of the simulation is the main class “ParticleSim”. Classes, that implement the interface “PacticleSimBuilder” are used for the initialization of the main class. In these classes the input files are read and the necessary steps are taken to generate the objects of the simulation. The method “runSimulation” of the main class contains the main loop. Instead of doing the actual integration of the equation of motion in this main loop, the integration is delegated through the “Integrator” interface. In this way, the method of integration can be easily exchanged. To keep the program efficient, an event system is used: Each integration is performed over the time difference to the next event. Events can be the pulsing of a coil or the recording (output) of partial results such as positions, velocities and energies of the particles at special points in time.

[1] Some care has to be taken not to use virtual functions inside the inner loops where the most processing time is spent. In the present work, compile-time polymorphism through templates has been used instead whenever possible.

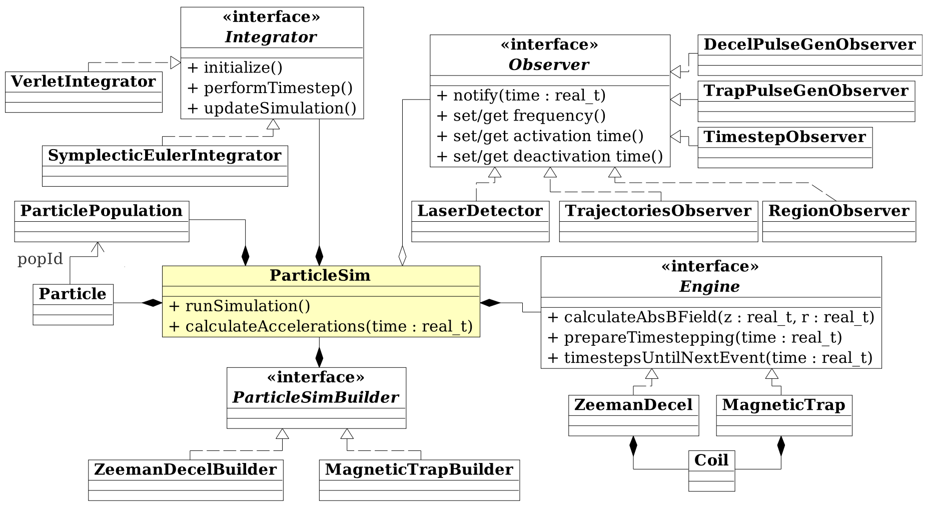


Figure 4.2: UML class diagram showing the major classes and interfaces of the program. Only the most important methods and attributes are displayed in this figure.

The coils themselves are not maintained in the main class. Instead, the magnitude and the magnetic field gradient are determined through the method “calculateAbsBField” of the “Engine” interface. This allows one to use the main class “ParticleSim” in different contexts such as for the decelerator with or without the trap together with the trap or other configurations one would like to consider.

The recording of partial results is sometimes referred to as “measurement” or “observation”. It is done through the “Observer” interface which follows the Observer Pattern, a common design pattern in the field of software design [28, 29]. The observers have a certain frequency with which they are notified. In between these events, the equation of motion can be integrated without producing any output or doing any further calculations such as calculating the time at which the next coil will be pulsed.

Each particle is associated with an identification number of a population to which the particle belongs. These particle populations, which are stored as objects of the class “ParticlePopulation” in an array inside the main class, determine the Zeeman energy and mass of the particles. In this way it is possible to simulate species with different masses and Zeeman sublevels.

Chapter 5

Optimization

5.1 Decision space and objective functions

An optimization problem consists of a set of alternative solutions, the so-called decision space X. The solutions themselves are represented as vectors $\mathbf{x} \in X$ [1]. The objective function $f : X \to \mathbb{R}$ is a quality measure for the elements of the decision space. It maps each element of the decision space to a real number. The goal can be either to minimize or to maximize this function value. In the present work, the convention is to minimize. Through the objective function, a preference structure $(X, \mathit{prefrel})$ can be defined by

$$\mathbf{x}\, \mathit{prefrel}\, \mathbf{y} \Leftrightarrow f(\mathbf{x}) \leq f(\mathbf{y}) \tag{5.1}$$

for any two solution-vectors $\mathbf{x}$ and $\mathbf{y}$ of X. The decision space together with the objective function form the so called problem landscape.

The optimal solutions of a subset $S \subseteq X$ of the decision space are defined as those elements of S that are preferred to every other element of S (according to *prefrel*). The optimal solutions of X itself are called globally optimal. Local optima are optimal with respect to a certain neighbourhood $N(\mathbf{x}) \subseteq X$, where $N(\mathbf{x}) : X \to 2^X$ is a neighbourhood function that tells which elements of X belong to the neighbourhood of a solution $\mathbf{x} \in X$.

In case of Zeeman deceleration the decision space could in principle consist of all adjustable parameters like the pulse sequence, the position and current of each coil and many more. In the present work, only the pulse durations will be taken into consideration.

Two different kinds of objective functions are considered: trapping of low-field-seeking particles and bunching of high field seeking particles. Both kinds consist of multiple objectives. In the following two sections, these objectives will be stated. How to deal with multiple objective functions is described in Section 5.2.

5.1.1 Trapping of particles

In principle the goal is to maximize the local phase-space density in the trap. However, there are two problems with that objective. The first problem is that it is not clear at which point in time the local phase-space density should be taken. The second problem is that at a certain point in time a particle can have zero velocity but a lot of potential energy. Therefore it would be necessary to average over time. But there exist indirect ways to define an objective function that maximizes local phase-space density.

[1]Here, bold letters are used to denote general vectors and matrices while the previously employed arrow-notation $\vec{x}$ is only used to represent 3-dimensional Euclidean vectors.

In the simulation described in Section 4 it is possible to calculate the total energy of a particle by adding kinetic energy and the internal Zeeman energy due to the local external magnetic field. This allows to define an objective function which can be calculated more efficiently than a time-average of local phase-space density. The corresponding objective can be stated as follows:

Find a pulse sequence, which

1. maximizes the number of trapped particles i.e. maximize:
$$\frac{N_\text{trapped}}{N_\text{total}}, \tag{5.2}$$
where N_trapped is the number of trapped particles and N_total is the total number of sampled particles.
2. minimizes the mean total energy of the trapped particle i.e. minimize:
$$\frac{\langle E_\text{total} \rangle}{E_\text{upper}}, \tag{5.3}$$
where E_total is the total energy of each particle, E_upper is the total energy that a particle would need to escape the trap. The average is taken over all trapped particles.

These quantities are taken at the time when the process of loading the trap has been completed, i.e. when the front solenoid A of the trap has been switched on the second time with a current opposed to the rear solenoid B (see Section 3.6).

A particle is trapped, if it is located in the region around the centre of the trap which is bounded by the positions of the saddle points and if its total energy is below the threshold given by the radial saddle-point of the magnitude of the B-field.

An optimization with these two objectives will also optimize the local phase-space density of the trap since the total energy determines how far a particle can move up the potential hills which compose the trap and is at the same time an upper bound on the kinetic energy of the particles.

There is still a remaining problem with these two objectives: the pulse sequences where no particle will end up in the trap cannot be compared. Thus, an optimization algorithm that starts only with incomparable solutions cannot use any information to get into the region of decision space where comparisons can be made. This problem can be addressed by introducing secondary objectives which make those solutions comparable. However, one has to be careful not to guide an optimization into the wrong direction. In practice, the following secondary objectives were successful in guiding the optimization into a region of decision space where particles can be trapped:

Find a pulse sequence, which

1. maximizes the number of particles located inside the trap region
2. minimizes the mean total energy of all particles inside the trap region

at the time when the process of loading the trap has been completed.

Even with this secondary objective function there are incomparable solutions in decision space as it may happen that no particle is inside the trap region. The number of such solutions can be further reduced by introducing a tertiary objective:

> Find a pulse sequence which minimizes the average distance to the centre of the trap of each particle at the time when the process of loading the trap has been completed.

In the simulation it is necessary to remove particles that escape the system boundaries given by the inner surface of the tubes and chambers since those particles are not of interest. Hence, pulsing sequences exist for which all particles are removed before the time the objective function is evaluated and thus these solutions are incomparable.

All these objectives can be merged into a single objective function (see Equation (5.5) as is described in the Section 5.2 devoted to multiobjecitve optimization.

5.1.2 Bunching of high-field-seeking particles

Besides the fact that high-field-seeking particles cannot be trapped (see Section 3.6), these particles are hard to focus because of the radially convex nature of the absolute magnetic field inside the solenoids (see Section 3.4). Nevertheless, the magnetic field in the space between the solenoids is of concave nature in radial direction, i.e. for a point that moves away from the translational axis the magnitude of the magnetic field decreases which makes it possible to focus high-field-seeking particles. It is therefore an interesting question whether a pulse sequence exist that is able to keep a reasonably large number of high-field-seeking particles together and, at the same time, decelerate them to a desired velocity, i.e. increase the local phase-space density of that bunch.

Again similar problems arise as for the trapping of particles described in the previous section, e.g. the question at which time to measure the local phase-space density. An additional question for the bunching of high-field-seeking particles, is *where* to measure the local phase-space density?

The following approach can be followed to overcome these problems:

> For a given desired velocity $\vec{v}_{\text{desired}}$, define the set P of n particles that have the closest velocity to $\vec{v}_{\text{desired}}$ under the condition that there are at least n particles in the simulation. If there are less than n particles in the simulation, then the set P will consist of all remaining particles.
>
> Find a pulse sequence, which
>
> 1. minimizes the mean difference of particle velocities $\vec{v}$ to the desired velocity, e.g. minimize:
>
> $$\langle|\vec{v} - \vec{v}_{\text{desired}}|\rangle, \tag{5.4}$$
>
> where the average is taken over all particles in the set P.
>
> 2. minimize the standard deviation of the positions of the particles in the set P.
> 3. minimize the standard deviation of the velocities of the particles in the set P.
> 4. if $|P| < n$ then maximize $|P|$.
>
> The measurement of these quantities is taken at the point in time after the last pulse of the pulse sequence.

The first objective is an assertion that the particles are decelerated to the desired velocity. The second and third objectives are to optimize the density of this bunch of n particles. The fourth objective is only active if there are less than n remaining particles in the simulation.

5.2 Multiobjective optimization

There are several ways of dealing with multiple objective functions. All approaches have in common that at some point the user is asked to decide which objectives are more important than others. Thus, two different types of multiobjective optimization can be distinguished by whether the user has to make this decision before or after the optimization. The first type of multiobjective optimization is easier to implement than the second, whereas the second type is often more convenient for the user. In the second type of multiobjective optimization the goal is to find (or approximate) the Pareto-optimal set, i.e. the subset of the decision space for which no solution in the decision space exists that is better in all objectives than any of the members of this subset. Thus, one says that the members of the Pareto-optimal set are not dominated. The user can then pick a solution out of this set according to her preferences.

For the trapping of low-field-seeking particles described in Section 5.1.1 it can be argued that the first objective (number of trapped particles) is more important than the second objective (mean energy of the trapped particles) because increasing the number of trapped particles will by itself increase the local phase-space density as a particle has to be slow enough to be trapped. The second objective is only of interest if there are enough particles inside the trap. Therefore the decision can be made before the optimization. The following aggregation based approach has the advantage that it is easy to implement: Take a weighted sum over the objective functions in order to reduce the multiobjective optimization problem to a single objective function (weighted-sum aggregation).

The aggregated objective function which is to be minimized for the trapping of particles reads (cf. Equations (5.2) and (5.3)):

$$f = \begin{cases} \gamma\left(1 - \frac{N_{\text{trapped}}}{N_{\text{total}}}\right) + \frac{\langle E_{\text{total}}\rangle_{\text{trapped}}}{E_{\text{upper}}} & \text{if } N_{\text{trapped}} > 0 \\ \gamma + \delta\left(1 - \frac{N_{\text{region}}}{N_{\text{total}}}\right) + \frac{\langle E_{\text{total}}\rangle_{\text{region}}}{E_{\text{upper}}} & \text{if } N_{\text{trapped}} = 0 \wedge N_{\text{region}} > 0 \\ \gamma + \delta + \varepsilon + \frac{\langle \|\vec{x} - \vec{x}_{\text{trap}}\| \rangle}{r_{\text{region}}} & \text{if } N_{\text{region}} = 0 \wedge N_{\text{final}} > 0 \\ \infty & \text{else} \end{cases}, \tag{5.5}$$

where $\langle\cdot\rangle_{\text{trapped}}$ indicates that the average is taken over all trapped particles whereas $\langle\cdot\rangle_{\text{region}}$ denotes an average over all particles inside the trap region (without the upper bound E_{upper} on the total energy). N_{total} is the total number of sampled particles, N_{trapped} is the number of trapped particles and $N_{\text{region}} \geq N_{\text{trapped}}$ denotes the number of particles that are located in the trap region. $N_{\text{final}} \geq N_{\text{region}}$ denotes the final number of particles that have not yet been removed at the time when the objective function is measured. The weights γ and δ are positive real numbers. In the cases $\gamma = N_{\text{total}}$ and $\delta = N_{\text{total}}$, the first objective (number of particles) will always be weighted more than the second objective (energy of the particles). $\vec{x}$ denotes the position of each particle and $\vec{x}_{\text{trap}}$ denotes the position of the centre of the trap. r_{region} denotes the minimal radius of the trap region which is approximated by an oblate spheroid (rotationally symmetric ellipsoid). The average in the third case is taken over all N_{final} remaining particles. ε should be chosen in such a way that the third case always gives larger objective function values than the second case, e.g. the initial ratio between the average total energy and E_{upper}. Note that in this way, the objective function values are increasing by going from the first case to the second, from the second to the third as well as from the third to the fourth.

The objectives for the bunching of the high-field-seeking particles which are described in Section 5.1.2 can be treated in the same way: It can be argued that it is first necessary to be able to decelerate the particles to the desired velocity. If the desired

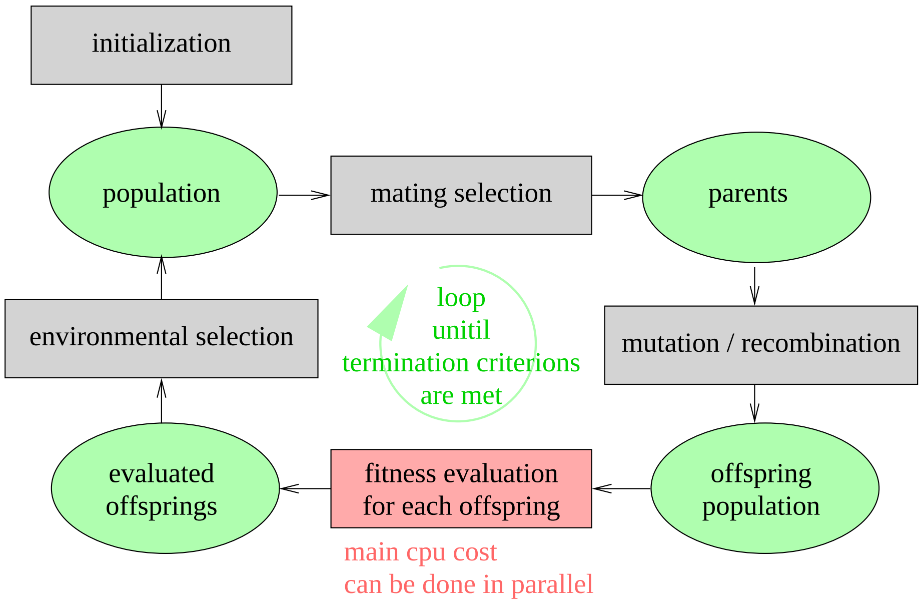


Figure 5.1: Flow chart of an Evolutionary Algorithm. Each loop is called a generation.

velocity is low enough, this goal can only be achieved if the particles do not defocus too much. Thus, the first objective already includes some part of the bunching. This indicates that the first objective should be weighted more than the second and third objectives.

5.3 Choice of the optimization method

Since the objective functions described in Sections 5.1 and 5.2 deal with a finite number of particles, they are not continuous in all points. Indeed, for fixed initial conditions a pulse sequence may be found that traps exactly one particle, but after a slight modification will prevent that particle from being trapped. Discontinuities make it impossible to use gradient-based optimization methods as for instance Quasi-Newton methods [30]. Nevertheless, the objective function can be made continuous, if the expected number of trapped particles for a given distribution of the initial conditions is taken instead. Still, approximating the expected number of trapped particles with the method described in Section 4.1 requires the calculation of a large number of samples.

Even without considering collisions, it is hard to find a closed form of these objective functions since the magnetic field is not static during the switching of the coils and because of the condition that a trajectory which escapes the boundaries of the tube is not accepted. Thus, little is known about the behaviour of these functions. They are assumed to be non-separable because the adjustment of the pulse of one coil changes the distribution of the particles when they come to the next coil. Nevertheless, it may have convex quadratic behaviour in some regions of the decision space.

Furthermore, adjusting the pulse sequence of a decelerator with n solenoids involves an n-dimensional optimization problem if only the switch-off times are adjusted. Because a typical decelerator has $n \geq 12$ solenoids, the optimization problem is high-dimensional.

Evolutionary Algorithms (EA) are successfully applied in highly complex optimization problems in a variety of fields such as circuit design, aircraft design, antenna design and many more. Evolutionary Algorithms are randomized search algorithms that aim at minimizing the number of assumptions about the underlying optimization problem while still being able to exploit the information gained during the optimization process in such a manner that the optimum can be found with the least number of objective function evaluations. Figure 5.1 shows the general scheme of Evolutionary Algorithms. The key concepts of these algorithms are selection and variation (mutation and variation). Selection means that only those solutions are kept in memory and selected for variation that are most promising to yield better solutions in subsequent populations. Variation means not just purely random modification (mutation) of the selected solutions but also the exploitation of information gained from the selected solutions to create the new solutions (recombination).

Evolution Strategies (ES) is a branch of Evolutionary Algorithms which has been designed for those optimization problems that involve real-valued parameters [31]. Derivatives of this branch were successfully applied in the field of online optimization where the objective function value is determined experimentally. By the use of Evolution Strategies, the feedback control optimization of Stark deceleration has been demonstrated recently by Gilijamse et al. [32]. The advantage that Evolution Strategies have over e.g. gradient based methods in online optimization is that they are more robust in case of fluctuations and uncertainties in the objective function evaluation. This is mainly because also solutions that yielded an objective function value which is worse than the best solution found so far can be kept in memory and may be used to generate new solutions. Although the present work does not comprise the coupling to the experiment, this property allows one to sample less particles in the evaluation of the objective function.

The most common variation operation in Evolution Strategies is to modify the selected solutions by a random perturbation, which obeys a multivariate normal distribution $\mathscr{N}(\mathbf{o},\mathbf{C})$ with the mean being the zero-vector $\mathbf{o}$ and covariance matrix $\mathbf{C}$. Traditionally this sampling was done only uncorrelated, i.e. $\mathbf{C}$ being a diagonal matrix. On the diagonal of $\mathbf{C}$ are the variances of the perturbation. The magnitude of the variances determines the typical length of the steps taken from the old population to the new population (step size). The covariance matrix is an example for a so-called strategy parameter. There are several approaches to adjust these additional parameters. One of the early approaches was to associate each solution with slightly modified strategy parameters in order to coevolve these parameters together with the solution vector (self-adaptation).

In recent years, a completely derandomized approach to update the covariance matrix C based on previously sampled solutions, the so-called Covariance Matrix Adaptation (CMA) [13] has been developed by Hansen. This method is capable of adjusting the full covariance matrix, not just its diagonal elements. It has been demonstrated by numerical experiments [33] that the CMA Evolution Strategy (CMA-ES) performs better than other continuous Evolutionary Algorithms on some test problems that are non-separable or multimodal (multiple local optima).

CMA was originally designed to perform well with small population sizes. Nevertheless there is an extension to CMA-ES, called rank-μ update [34], which allows larger population sizes (see next section). This is useful if a large number of processors can be employed in parallel (see Section 5.7).

For these reasons CMA-ES is used for the optimizations in the present work. Comparison to other optimization methods is left as an open topic for further research.

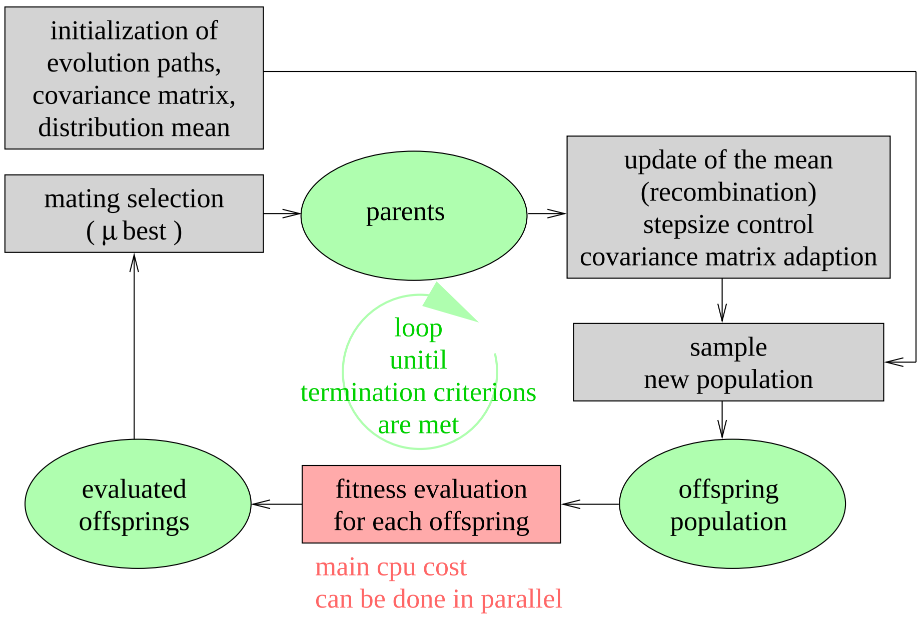

initialization of evolution paths, covariance matrix, distribution mean
mating selection (μ best)
parents
update of the mean (recombination) stepsize control covariance matrix adaption
sample new population
loop unitil termination criterions are met
evaluated offsprings
fitness evaluation for each offspring
offspring population
main cpu cost can be done in parallel

Figure 5.2: The Covariance Matrix Adaptation Evolution Strategy (CMA-ES)

5.4 Optimization algorithm: CMA-ES

This section gives an overview of how CMA-ES works. A detailed description including the recent extensions is given in [14]. CMA-ES is different from traditional Evolutionary Algorithms because it is not necessary to transfer the population of solution vectors itself from one generation to the next. Instead, only the mean vector which is used to sample the next population of offspring [2] and the strategy parameters are maintained as shown in Figure 5.2. CMA-ES is a $(\mu/\mu_W,\lambda)$ Evolution Strategy. This is a common notation to classify Evolution Strategies, where λ is the population size (offspring population), the μ before the slash denotes the number of parents and the μ after the slash has the meaning that all μ parents are recombined. Furthermore the subscript W indicates that it is a weighted recombination (see below) and the comma has the meaning that CMA-ES is non-elitist, i.e. that all parents are replaced by the new offspring population.

The new population is sampled by adding λ different normally distributed steps $\mathbf{y_k} \sim \mathscr{N}(\mathbf{o},\mathbf{C}),(k=1,\ldots,\lambda)$ to the old mean vector $\mathbf{m}$:

$$\mathbf{x}_k = \mathbf{m} + \sigma \mathbf{y}_k, \tag{5.6}$$

where $\mathbf{o}$ is the zero-vector, $\mathbf{C}$ is the covariance matrix and σ is a scalar that scales the covariance matrix (global step size).

The most time consuming part of the algorithm is the evaluation of the objective function (fitness) for each sampled individual $\mathbf{x}_k$. The mean vector is then updated based on the μ best individuals of the new population by taking the weighted average over the steps that were taken from the old mean to the new individuals:

$$\langle \mathbf{y} \rangle_w = \sum_{i=1}^{\mu} \omega_i \mathbf{y}_{i:\lambda}, \tag{5.7}$$

[2]In evolution strategies the offspring population is normally referred to as “the population”.

where $\sum_{i=1}^{\mu} \omega_i = 1, \omega_i > 0$ and the $i:\lambda$ in $\mathbf{y}_{i:\lambda}$ indicates that the steps are sorted according to the fitness of individual i. Since the steps do not include the global step size σ, this weighted average has to be multiplied by σ before they are used to update the mean vector:

$$\mathbf{m} \leftarrow \mathbf{m} + \sigma \langle \mathbf{y} \rangle_w \tag{5.8}$$

Note that this step contains mating selection and weighted recombination (compare Figures 5.1 and 5.2). In order to update the strategy parameters the so-called "variance effective selection mass" $\mu_{eff} = (\sum_{i=1}^{\mu} \omega_i^2)^{-1}$ is used. Note that $\mu_{eff} = \mu$ if the weights are all equal. It is usually better to have linearly decreasing weights: $\omega_i = 2(\mu - i + 1)/(\mu(\mu+1))$ which is the default setting in the implementation of CMA-ES which is used in the present work. Because $\mu_{eff} \approx \lambda/4$ indicates a reasonable setting, the default number of parents is set to be $\mu = \lambda/2$ in this case.

In order to adapt the covariance matrix and the global step size, two so-called cumulated evolution paths $\mathbf{p}_c$ and $\mathbf{p}_\sigma$ are maintained. The cumulated evolution path is a weighted sum of the updates of the mean $\langle \mathbf{y} \rangle_w$ that have been made in the previous generations. By going backwards from the last generation to the first generation, the weights are exponentially fading, so that the most recent generation receives the largest weight. This can be achieved by the following update rules [14]:

$$\mathbf{p}_c \leftarrow (1 - c_c)\mathbf{p}_c + h_\sigma \sqrt{c_c(2 - c_c)\mu_{eff}} \langle \mathbf{y} \rangle_w \tag{5.9}$$

and

$$\mathbf{p}_\sigma \leftarrow (1 - c_\sigma)\mathbf{p}_\sigma + \sqrt{c_\sigma(2 - c_\sigma)\mu_{eff}} \mathbf{C}^{-\frac{1}{2}} \langle \mathbf{y} \rangle_w, \tag{5.10}$$

where c_c and c_σ are decay factors, $\sqrt{c_c(2 - c_c)\mu_{eff}}$ and $\sqrt{c_\sigma(2 - c_\sigma)\mu_{eff}}$ are normalization factors which assure that if the update of the mean is normally distributed with zero mean and covariance matrix $\mathbf{C}$ in each generation, the cumulated evolution paths $\mathbf{p}_c$ and $\mathbf{p}_\sigma$ will have the same distribution (see [14] for a proof). h_σ is a Heaviside function that stalls the update of $\mathbf{p}_c$ if the evolution path $\|\mathbf{p}_\sigma\|$ is too large. The expression for h_σ, as well as some details are also given in [14]. Also notice that for p_σ the vector $\langle \mathbf{y} \rangle_w$ is multiplied from the left by the inverse of the square root of the covariance matrix $\mathbf{C}^{-1/2}$. This isolates the effect of the global step size σ from the effect of $\mathbf{C}$. It is done because $\mathbf{p}_\sigma$ is used for the update of the global step size σ.

By using the cumulated evolution path $\mathbf{p}_c$, the covariance matrix is updated [14]:

$$\begin{aligned} \mathbf{C} \leftarrow & (1 - c_{cov})\mathbf{C} + \frac{c_{cov}}{\mu_{cov}}(\mathbf{p}_c \mathbf{p}_c^T + \delta(h_\sigma)\mathbf{C}) \\ & + c_{cov}\left(1 - \frac{1}{\mu_{cov}}\right) \sum_{i=1}^{\mu} \omega_i \mathbf{y}_{i:\lambda} \mathbf{y}_{i:\lambda}^T. \end{aligned} \tag{5.11}$$

The sum $\sum_{i=1}^{\mu} \omega_i \mathbf{y}_{i:\lambda} \mathbf{y}_{i:\lambda}^T$ is the rank-μ update that has already been mentioned in Section 5.3. With the rank-μ update, information of previously sampled individuals enters the covariance matrix. In this way the information gained from larger populations has more influence on the sampling. The rank-μ update can be disabled by setting $\mu_{cov} = 1$. A typical setting is $\mu_{cov} = \mu_{eff}$.

The global step size σ is updated by comparing the magnitude of the actual cumulated evolution path $\mathbf{p}_\sigma$ to its expected value if the best individuals are uniformly distributed and thus randomly selected. If the actual evolution path is longer than the expected length, the global step size will be increased in order to move faster in that direction. On the other hand if the global step size is smaller than the expected length, the global step size is decreased in order to take a closer look at the proximity of the mean vector. The corresponding update rule is given by [14]:

$$\sigma \leftarrow \sigma \exp\left(\frac{c_\sigma}{d_\sigma}\left(\frac{\|\mathbf{p}_\sigma\|}{\mathbb{E}[\|\mathscr{N}(\mathbf{o},\mathbf{I})\|]} - 1\right)\right), \tag{5.12}$$

where d_σ is a damping factor and $\mathbb{E}[\|\mathscr{N}(\mathbf{o},\mathbf{I})\|]$ denotes the expectation value of the length of a vector that is distributed by an uncorrelated multivariate normal distribution with zero mean, the covariance matrix being equal to the identity matrix $\mathbf{I}$, i.e. all variances being 1 (chi distribution).

Some default values for the parameters of CMA-ES are given in the appendix of [14]. Special care should be taken when choosing the population size λ. By numerical experiments presented in [35] the following resonable default value has been obtained

$$\lambda = 4 + \lfloor 3 \ln n \rfloor, \tag{5.13}$$

where n is the dimension of the decision space (problem dimension). This is at the same time an approximate lower bound for λ. Because of the rank-μ update, increasing the population size usually makes the search more global and can thus reveal better solutions. However, in the worst case the convergence speed decreases linearly with the population size. A common practice is to run the optimization first with a small population size and then restart the optimization with an increased λ.

There are several possible termination criteria. For example, one can impose a lower bound on the objective function value, a lower bound on the variances with which new solutions are sampled or simply on the total number of generations. Additionally there is an upper bound on the condition number of the covariance matrix in order to assure that there are no numerical instabilities. In general one has to be careful when choosing the termination criteria. As an example, the variance can fluctuate during the optimization and thus a lower bound on the variance may prevent the optimization to converge to the optimum. The algorithm can also be supervised by the user through the frequent output that is generated and hence be terminated manually.

Excellent implementations of the CMA-ES algorithm in different programming languages have been given by its inventor N. Hansen [14, 36]. In the present work, the implementation in the C programming language has been used (see Appendix D).

5.5 Decoder function

As described in the previous section, the CMA-ES algorithm produces samples of an n-dimensional vector with a multivariate normal distribution. The objective functions described in Section 5.1 take pulse sequences as their input. Thus a so called decoder function is needed that maps the vectors sampled by the optimization algorithm to pulse sequences (switch-on and switch-off times of the solenoids). The decoder function can also contain some constraints on the possible pulse sequences.

For a decelerator with k solenoids, the first k elements of the sampled vectors are interpreted as the duration of the pulse of the corresponding solenoid (each solenoid is pulsed exactly once). The durations are constrained to be positive (see Section 5.6). The time at which the first solenoid is pulsed is kept constant during the whole optimization. The switch-on time of any other solenoid is set to the time at which its preceding solenoid is switched off minus a constant overlap time (see Section 3.5).

The first coil of the trap is switched on before the last solenoid of the decelerator is switched off with a constant temporal overlap. The duration of the first pulse of the first solenoid of the trap (see Section 3.6) is determined by the $(k+1)$-th element of the sampled vector. The second solenoid is switched on with another constant overlap time before the first pulse of the first solenoids ends. The $(k+2)$-th element of the sampled vector then determines the duration from the point in time when the first coil has become completely inactive (after the ramp of the current) until the second pulse of the first coil (with opposite current) starts.

5.6 Imposing constraints

In many situations it is necessary to exclude some solutions that are not physically meaningful, impossible to implement or not of interest to the user. For example, negative pulse durations have no physical meaning. There is also a lower bound on the pulse duration that can be realized experimentally.

Probably the easiest way to impose constraints on the sampled vectors, is to discard every infeasible vector that has been sampled in a single generation and sample it again until it becomes feasible. The objective function is evaluated after a feasible vector has been sampled by chance. The drawback of this approach is that if the probability of producing a feasible solution is low, the sampling becomes inefficient.

The second method is to assign a penalty to the objective function value which assures that every feasible solution will be preferred to the decision vector that violates the constraint. The disadvantages of this approach are, that it may be possible that the optimization gets lost in a region where only infeasible solutions exist and that computing time may be waisted by sampling and evaluating lots of infeasible solutions.

Another approach is to change the decoder function (see Section 5.5) in such a way, that it only produces feasible solutions. However, it may not always be possible to find a simple mapping to the feasible solutions and the mapping changes the problem landscape and may therefore mislead the search.

As an example, one could assure a certain component of the decision vector is positive by taking the square of the corresponding component of the sampled vector in the decoder function. On the positive quadrant this is an order preserving transformation. The CMA-ES algorithm is invariant to order preserving transformations i.e. these transformations do not change the performance of the algorithm. A difficulty with this approach is that the meaning of the standard deviation is changed: a small deviation from a large mean value leads to a much bigger deviation in the squared value than the same deviation from a small mean value.

Once it is assured that all the sampled solution vectors are positive, one can impose a minimum pulse length by adding the amount of the minimum pulse length to the components of the solution vectors as an additional part of the decoder function.

5.7 Runtime and parallelization

The number of required function evaluations needed to converge to the optimum (if convergence is possible) strongly depends on the dimension of the optimization problem. As stated in [34], for many optimizations this dependence is on the order of n^2 where n is the problem dimension because the covariance matrix has $n(n+1)/2$ degrees of freedom which have to be adjusted to the problem landscape.

The objective function evalutations are the most time consuming step in the optimization algorithm. E.g. for a 12 stage Zeeman decelerator, each simulation takes about 90 seconds. Because the number of required function evaluations is typically on the order of 10^4 to 10^5, it is very beneficial to do as many function evaluations as possible in parallel. This is done by using the OpenMP compiler paradigms (see e.g. [37] for an introduction to OpenMP). The upper limit on the number of independent threads that can be processed in parallel is given by the population size λ. The population size should also be dividable by the number of parallel processors because otherwise some processors have to wait at each generation until the other processes have been finished.

Chapter 6

Optimization results

6.1 Trapping of atomic hydrogen

The first optimization addresses a decelerator of 12 solenoids combined with two additional solenoids for the trapping. Thus, the objective function takes $n = 14$ parameters. The weights in the aggregated objective function have been chosen to be $\gamma = \delta = 500$. Each population consists of 16 individuals. Notice, that this value has been chosen higher than the suggested default value (see Equation (5.13) in Section 5.4) in order to distribute the function evaluations efficiently to 8 independent processors. The program has a mechanism that doubles the population size after a termination criterion has been met and starts the optimization again with the same initial conditions as for the first run except for the different seed of the random number generator.

The objective function has been implemented in such a way, that a solution vector that contains a negative component is penalized by an infinite objective function value (see Section 5.6). A minimum pulse duration of 11μs has been imposed as an additional constraint. As an initial search point, a pulse sequence which has been manually optimized to trap hydrogen atoms in the Zeeman sublevel $F = 1, m_F = 0$ with an initial velocity $v_z = 500$m/s equal to the mean velocity of the beam. The standard deviation has been chosen to be initially equal to 1.0 for all pulse durations. The magnetic fields of the decelerator coils have been calculated for a current of $I_D = 300A$. The ones from the trap correspond to a current of $I_T = 200A$. For the input files which have been used to run the optimizations presented in this chapter, refer to Appendix A. The initial and optimized pulse sequences are given in Appendix B.

Figure 6.1 shows the convergence of the value of the aggregated objective function and the corresponding two single objective functions. While with the initial pulse sequence one is able to trap about 0.2% of the total number of sampled particles, the optimization increases the number of accepted particles to about 1.6%. The average total energy of the hydrogen atoms is also stabilized at about 68% of the upper bound given by the radial saddle point of the magnitude of the trap field which is 0.255T and corresponds to an energy of 0.0171meV. This in turn imposes an upper bound on the kinetic energy and hence on the velocity which is then at most 57.5m/s. Indeed, the average velocity of the simulated hydrogen atoms in the trap is only about 20m/s with the optimized version of the pulse sequence compared to approximately 30m/s that one obtains with the initial pulse sequence.

After the restart and doubling of the population size, the optimization converges to a pulse sequence which leads to more trapped particles at a lower average total energy. This indicates that it is more likely to find a good local optimum (or even the global optimum) with larger population sizes. It can also be seen, that the optimization with

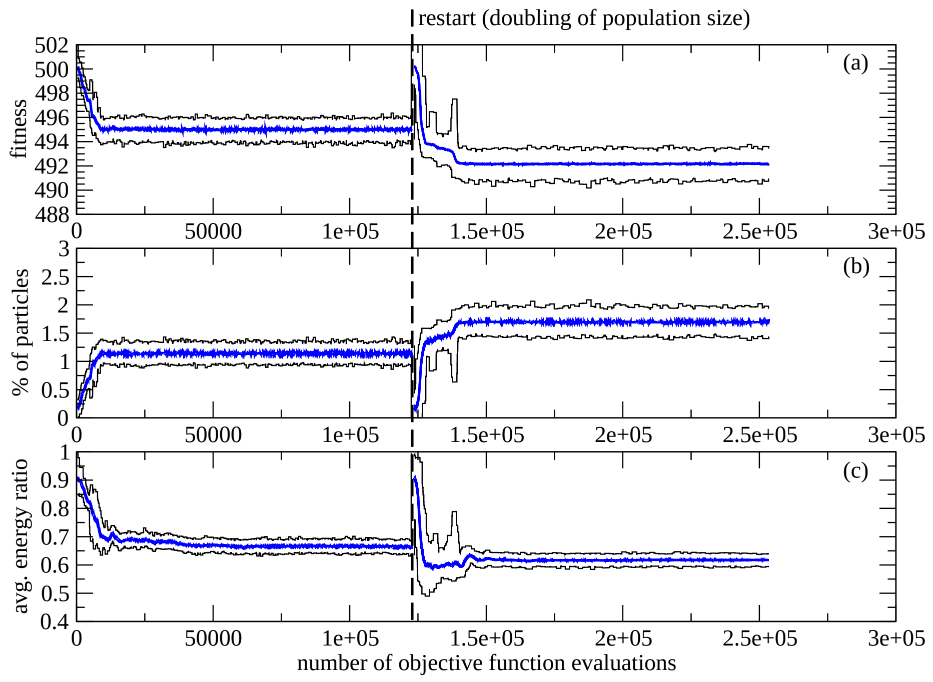


Figure 6.1: Optimization of the trapping of hydrogen atoms after deceleration by the magnetic field of 12 solenoids. The lines in panel (a) show the calculated objective function values, panel (b) shows the percentage of the sampled particles that have been trapped and panel (c) shows an average over the ratio of the total energy of each atom to the minimal energy which is needed to escape the trap. The lines were obtained by taking the median (blue line), minimum and maximum of the corresponding values that belong to the sampled individuals of 50 consecutive generations. The optimization has been restarted with twice the population size (32 compared to 16) after it has reached the stopping criterion (maximal condition number).

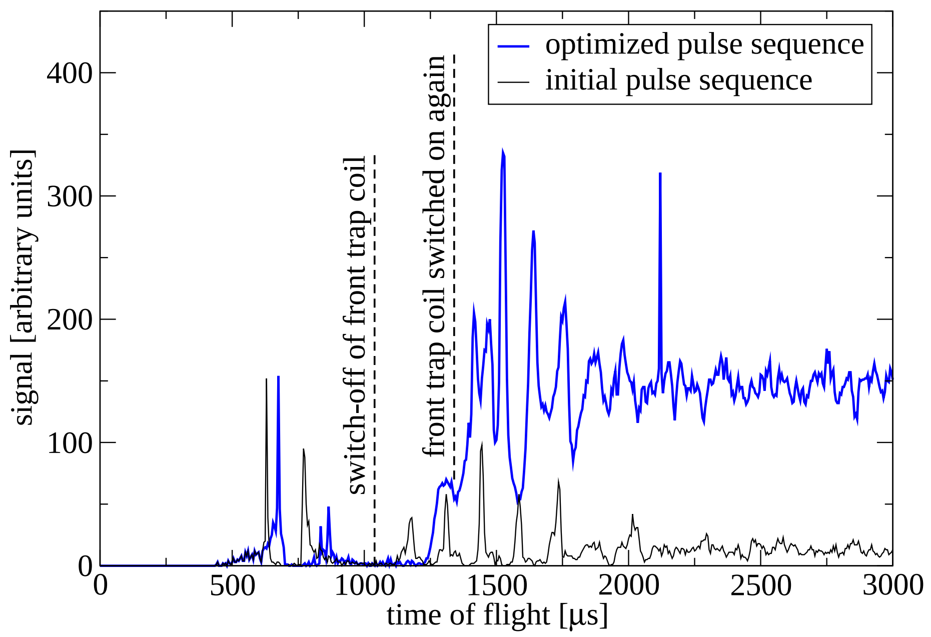


Figure 6.2: Simulated time-of-flight profiles of hydrogen trapping by pulsing 12 deceleration solenoids and 2 trap solenoids with the initial (thin, black line) and the optimized sequence (thick, blue line). The labelled switch-off and switch-on times belong to the optimized pulse sequence.

larger population size needs more function evaluations to stabilize at a local optimum.

The termination criterion, that has been met in the optimization before the restart as well as the succeeding run, was the upper bound on the condition number (9.01×10^{12}) of the covariance matrix used to sample the pulse sequences.

The optimization has a great impact on the expected time-of-flight profile as shown in Figure 6.2. The optimized pulse sequence already influences the first peak which corresponds to particles that are moving initially too fast and which will therefore be guided and even a bit accelerated. The second peak, which belongs to particles that are decelerated but not trapped is substantially reduced with the optimized pulse sequence. Instead a tremendous increase of the signal can be observed after the front trap coil has been switched on again.

Figure 6.3 shows the effect the initial and final pulse sequences have on the magnitude of the B-field seen by a synchronous particle while it is decelerated and trapped. During the optimization, this synchronous particle changes to one that has an initial longitudinal velocity of 476m/s which is slightly below the mean velocity of the beam of 500m/s. Thus less longitudinal velocity is removed from the particles with the optimal pulsing sequence. Instead, the emphasis of the optimized pulse sequence is on the radial focusing. Two different phases of the deceleration can be noticed in the optimized pulse sequence. Initially very low phase angles are obtained which prevent the beam from over-focusing by reducing the radial velocities. At this level mainly the radial phase-space distribution is shaped, as can be seen from Figure 6.4. After this initial level, the phase angles oscillate around 50 degrees and thus decelerate the particles in longitudinal dimension.

In order to study the effect of the weights in the aggregated objective function it has been tried to increase the weight γ in Equation (5.5) on the number of particles from $\gamma = 500$ to $\gamma = 5000$. As a result, the optimization converged to a slightly worse

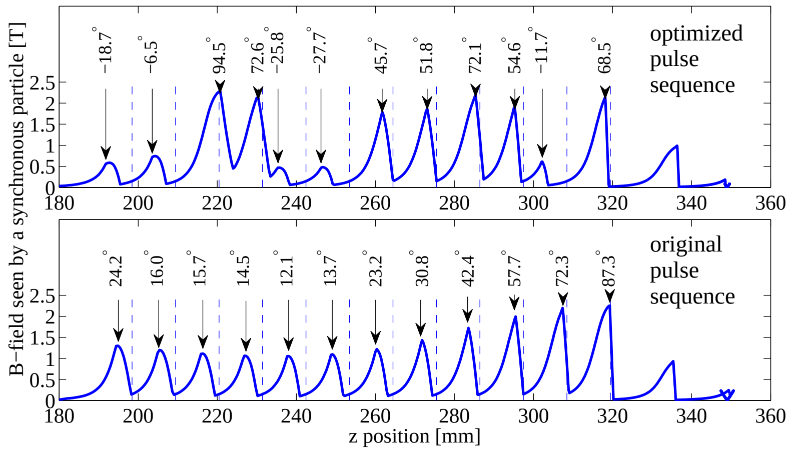


Figure 6.3: The magnitude of the B-field that a synchronous particles sees. In the optimized pulse sequence this particle has an initial velocity of 476m/s which is below the initial velocity of the particle that is synchronous with respect to the initial pulse sequence which is 500m/s. The dashed lines indicate the positions of the coils

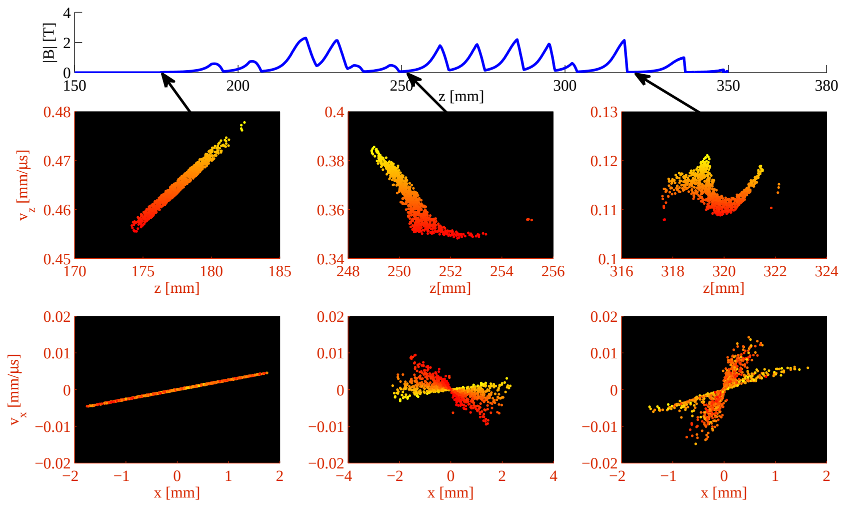


Figure 6.4: Different phase-space distributions of the trapped bunch of particles at different stages of the deceleration. The upper phase-space plots show the transversal distributions and the lower ones show the radial distributions. Each column represents a different point in time. The first column shows the distribution when the beam enters the decelerator, the second column show it after it has been radially focused and the third column shows the distribution at the end of the decelerator before the particles enter the trap. The colours indicate the total velocity of the particles (bright colours belong to high velocities).

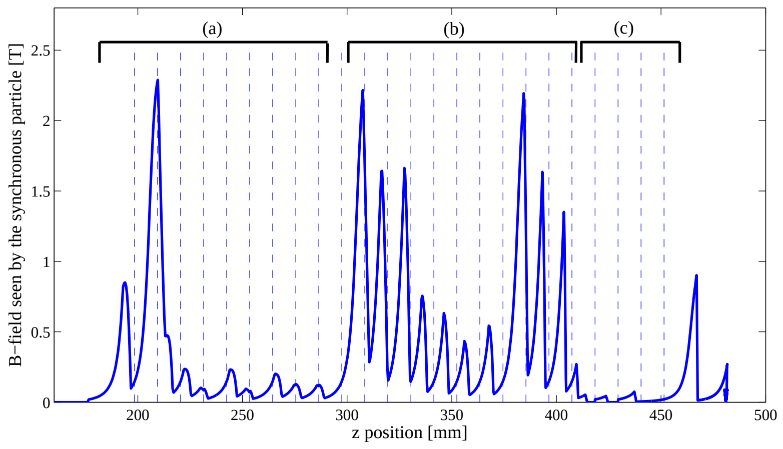


Figure 6.5: B-field that the synchronous particle experiences when 24 solenoids are pulsed with an optimized sequence. In region (a), the beam is radially focused, in region (b) a large fraction of the longitudinal velocity is removed from the particles and region (c) radial focussing before the loading of the trap. The dashed lines indicate the centres of the decelerator coils.

optimum where slightly fewer particles, i.e. about 1.1% were trapped. This can be explained by the fact that the problem landscape is roughened by the discontinuity of the approximated particle density. A rough problem landscape can mislead the optimization whereas the second objective (the energy density) is continuous and may guide the optimization towards better solutions.

To further investigate the properties of good pulse sequences, the optimization of hydrogen trapping has also be done for a decelerator that consists of 24 solenoids. This optimization problem is thus 26 dimensional for which the cosen populations size of $\lambda = 21$ is more than sufficient. These additional degrees of freedom allowed the optimization to find a solution with which one is able to trap about 3.0% of the total number of sampled particles which is a nearly twice as much than when only 12 solenoids are used. Also the mean initial velocity of the trapped particles is 485m/s which is below the mean velocity of the sampled gas beam (500m/s). As can be seen from Figure 6.5, the effect of the optimization on the pulse sequence obeys a similar pattern as for the deceleration by 12 solenoids. Again it has an initial phase (a) which is used for radial focussing and second phase (b) in which longitudinal deceleration takes place. There is an additional region (c), which shows again the pattern of controlling the radial motion of the gas beam. Figure 6.6 shows the convergence behaviour of this optimization.

6.2 Sensitivity to certain approximations

Instead of calculating the three components of the B-field from the solenoids which compose the decelerator and adding them together separately, one could as an approximation, add up only the magnitudes of the magnetic field from single solenoids. Figure 6.7 shows the influence of this approximation on the magnitude of the magnetic field if two successive solenoids are active at the same time. In the approximate treatment the magnitude does not decrease as fast in radial direction and thus a beam of low-field-seeking particles will be slightly less deflected. Nevertheless, a situation as depicted in

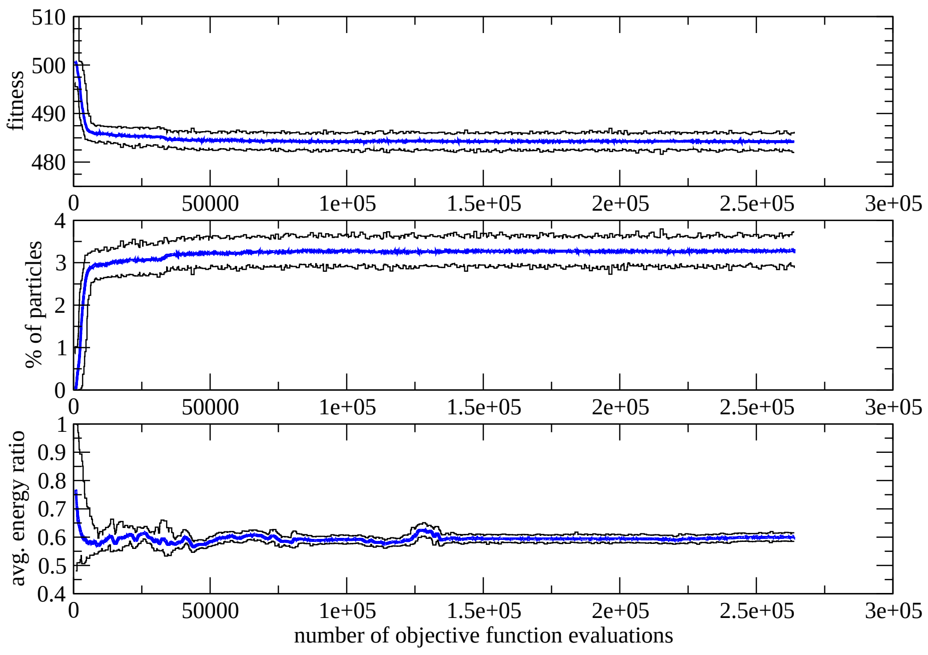


Figure 6.6: Optimization of the trapping of hydrogen atoms after deceleration by the magnetic field of 24 solenoids. The lines were obtained by taking the median (blue line), minimum and maximum of the corresponding values that belong to the sampled individuals of 50 consecutive generations.

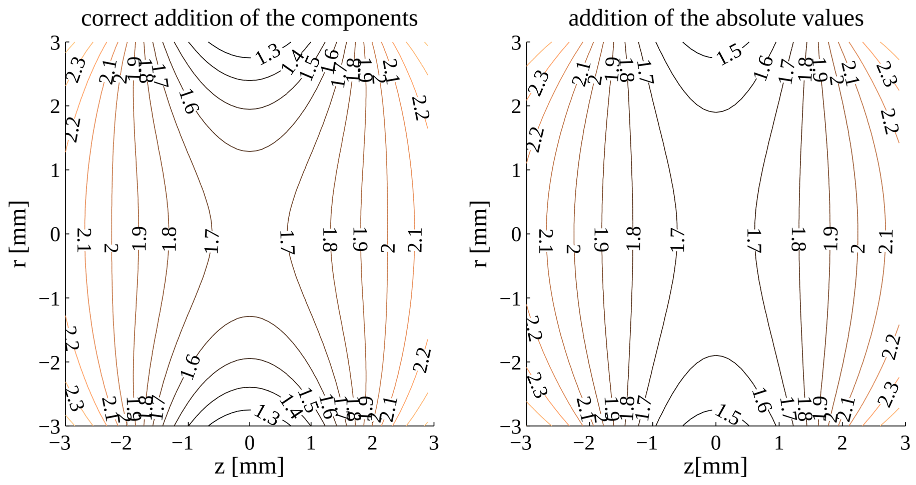


Figure 6.7: Effect of adding the magnitudes instead of the components of the B-field of single coils to obtain the magnetic field of two coils. The graphs show lines of equal magnitude of the B-field originating from two successive decelerator coils, both of which having the same amount and direction of current. The labels of the lines are given in Tesla. The zero-point is the centre between the coils.

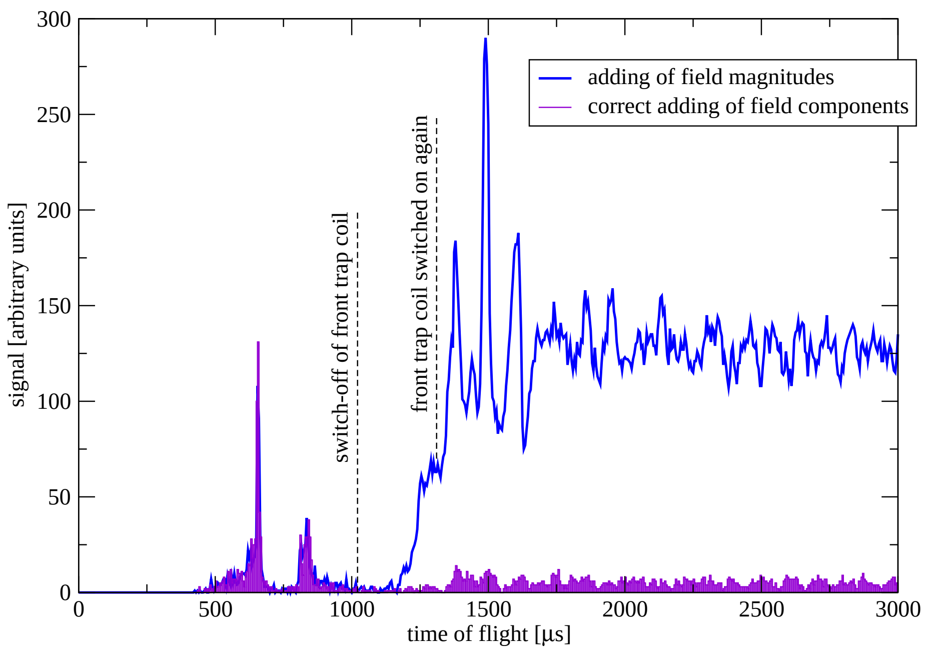


Figure 6.8: Influence of adding the magnitudes of the B-field of the coils instead of the actual components in the decelerator. The pulse sequence to obtain this time-of-flight distribution has been optimized for trapping in the situation were the fields are added wrong. This result shows that there are pulse sequences for which two addition methods lead to tremendously different number of trapped particles.

Figure 6.7 will only occur if the temporal overlap of the pulses is substantial. In this work such high overlaps are never used. For this reason, it was first expected not to influence the quality of the results significantly. However, it has been found out that this is a bad approximation for the trapping (even if the fields are added correctly in the trap) and should therefore no longer be used (see Figure 6.8).

It has further been investigated whether the cylindrical symmetry of the decelerator can be used to reduce the calculation of the trajectories to a two dimensional problem which can speed up the simulation. But with this approach, particles that initially have a position that is not located on the transversal axis and have a velocity that is not within a plane that contains the longitudinal axis cannot be simulated. Hence this is a bad approximation because it will change the density of trapped particles and should therefore also not be used for the optimization of the pulse sequence.

Chapter 7

Conclusions and Outlook

It has been demonstrated that there is a great potential to optimize the pulse sequences of Zeeman decelerators and that the Covariance Matrix Adaptation Evolution Strategy (CMA-ES) is a good tool to achieve this. The results of the optimization show, that the trapping can be improved by radially focusing the beam. Common patterns can be noticed in the optimized pulse sequences by investigating the corresponding phase angles with respect to a synchronous particle. It would be of great interest to see whether these results can be experimentally verified.

Further steps can be taken to improve the optimization:

- Comparison of the performance of CMA-ES on these objective functions with that of different optimization methods.
- It was shown in Section 6.1 that the choice of the weights in the aggregated objective function influences the quality of the revealed optimum. Hence, it would be useful to allow multiobjective optimization to approximate the Pareto optimal front [38].
- Coupling to the experiment. This requires the formulation of an objective function that can be evaluated experimentally. Such an optimization could also be used to investigate the quality of the computational model used in the simulation.

Acknowledgements

Here, I want to thank Professor Merkt for having offered me this interesting and versatile research topic, for his great interest in my proceedings and for the opportunity to work in his group. I also want to give special thanks to Alexander Wiederkehr and Dr. Stephen Hogan for their great assistance and for the insight they gave me into the multistage Zeeman deceleration experiment. Thanks also go to Dr. Urs Hollenstein who provided me with all the computational tools and the permission to load the two Servers “rydberg.ethz.ch” and “herzberg.ethz.ch” up to the limit of their computing power. Furthermore, I want to thank all other members of the Merkt Group for the time we spent together, including the skiing in Arolla. Last but not least I want to thank my life partner Claudia Born for the support she gave me and the patience she had when I was working on the present Bachelor thesis.

Appendix A

Input parameters for the simulation and optimization (version 2.56)

A.1 Description of the input parameters in the different input files

A.1.1 Decelerator ("inputdata_long.txt")

identifier of parameter	**description**
number_of_particles	the total number of particles that will be sampled
mass_factor	the atomic mass number
start_pos_z	the starting position along the z-dimension
start_radial_pos_on_disk	a value of 1 implies that the initial positions are chosen uniformly distributed on a circle with fixed radius of 0.05
starting_velocity_m/s	the average starting velocity along the z-dimension
init_beam_length_mm	the initial extent of the beam along the z-dimension
long_temperature_K	the temperature in the z-dimension
trans_temp_K	the temperature in the transversal dimensions
number_of_stages	the number of stages of the decelerator (not to confuse with the number of coils)
number_of_coils_per_stage	the number of coils per decelerator stage
number_of_coils_in_a_tower	the number of coils in a tower between two succeeding stages
distance_to_middle_first_coil_mm	the distance from origin to the centre of the first decelerator coil
distance_between_stage_coils_mm	the distance between two succeeding coils in a decelerator stage
distance_between_stages_mm	the distance between two succeeding decelerator stages

continued on next page

continued from previous page

identifier of parameter	**description**
distance_towercoils_mm	the distance between two succeeding tower coils
bound_z_mm	the maximum distance a particle can have from the origin along the z-direction
time_offset_in_us	the starting time of the simulation
incoupling_time_us	the in-coupling time which can be used to shift the pulses
time_simulation_stops_us	the point in time at which the simulation stops (This parameter is only effective in the stand-alone decelerator simulation. It is overwritten by the trap simulation and the optimization)
timesteps_ns	the step size with which the numerical integration of Newton's equation is done
ramp_time_coil_ns	the time in which the current of a decelerator coil raises from zero to its maximum and vice-versa
ramp_time_tower_coil_ns	the time in which the current of a tower coil raises from zero to its maximum and vice-versa
current_decelerator_A	the maximum current of the decelerator coils
reference_current_decelerator_A	the current with which the magnetic field stored in the file "decelmapaxrho.txt" has been simulated
current_tower_coils_A	the maximum current of the tower coils
reference_current_tower_coils_A	the current with which the magnetic field stored in the file "towermapaxrho.txt" has been simulated
detector_pos_mm	the distance of the laser detector from the origin along the z-dimension
detector_start_time_us	the time the detector gets active (This should be after the first pulse so that the positions of the particle at the beginning of the first pulse can be calculated in one step)
detector_radius_mm	the radius of the laser beam which is used to detect the particles
detector_inverse_frequency_us	the time interval between two successive measurements by the laser detector
remove_unaccepted_lfs_particles	a value of 1 activates a mechanism that will remove all particles that are at the time before a certain coil is switched on already behind the maximum of the magnetic field the coil would have it would be switched on
check_upper_bound_for_... ...predicted_total_energy	a value of 1 activates a mechanism that removes particles with a velocity that is so low that the particle will never reach large enough phase angles at the switch-off times of all coils that have a higher z-coordinate to get decelerated below the maximum amount of total energy a particle can have and still being trapped

continued on next page

continued from previous page

identifier of parameter	**description**
upper_bound_for_predicted_... ...total_energy_J	the upper bound on the energy a particle can have and still being trapped (this parameter is only effective if the above parameter is equal to 1.

Table A.1: description of the decelerator input file "inputdata_long.txt"

A.1.2 Trap ("inputdata_trap.txt")

identifier of parameter	**description**
timesteps_ns	the step size with which the numerical integration of Newton's equation is done.
ramp_time_trap_coil_ns	the time it takes to raise the current of a trap coil from zero to its maximum
trap_sim_current_A	the current with which the magnetic field in the file "trapmapaxrad.txt" has been simulated
trap_coil1_reference_current_A	the current of the first trap coil that corresponds to the case where the scaling polynomial that has been specified in the pulse file "trap_pulses1.txt" has a value equal to 1
trap_coil1_reference_current_A	the current of the second trap coil that corresponds to the case where the scaling polynomial that has been specified in the pulse file "trap_pulses2.txt" has a value equal to 1
trappos_mm	the position of the centre of the trap
distance_trap_coils_mm	the distance between the two centres of the trap coils along the z-dimension
time_simulation_stops_us	the time at which the simulation stops (this value is overwrites the value in the file "inputdata_long.txt" and is overwritten by the optimization in the objective function evaluation)
max_trap_region_radius_z_mm	the maximum distance that a particle can have from the centre of the cloud along the z-dimension
lower_bound_z_mm	the minimum coordinate a particle can have at any time during the simulation.
upper_bound_z_mm	the maximum distance from the origin along the z-dimension
upper_bound_r_outside_trap_mm	the maximum distance a particle can have in the decelerator from the longitudinal axis
upper_bound_r_in_trap_mm	the maximum distance a particle can have from the longitudinal axis in the trap
scaling_polynomial_degree	the maximum degree of the polynomial that can be used to scale the current in the trap coils and model its temporal evolution. The coefficients of this polynomial in the standard form are given for each pulse in the pulse files "trap_pulse1.txt" and "trap_pulse2.txt"

continued on next page

continued from previous page

identifier of parameter	description
trap_region_radius_z_mm	the maximum distance a trapped particle can have from the centre of the trap along the z-direction
trap_region_radius_r_mm	the maximum distance a trapped particle can have from the centre of the trap along the transversal dimensions
upper_trap_abs_B_T	the maximum magnetic field that leads to a potential energy lower than the maximal total energy a particle can have and still be considered as trapped (value of the saddle point of the potential hill surrounding the centre of the trap)
region_observer_inverse_... ...frequency_us	the time between two successive observations of the particles in the region of the trap ("output_trapregion.txt")

Table A.2: description of the trap input file "inputdata_trap.txt"

A.1.3 Objective function ("inputdata_obj.txt")

identifier of parameter	description
zeeman_energy_file	the file which contains the calculated relationship between the internal energy of a particle and the external magnetic field strength at a discrete number of different possible magnitudes of the magnetic field strength
read_particles_from_file	a value of 1 implies that the initial conditions of the particles are read from a file called "inputdata_particles.txt" and has 6 columns per row for the 6 dimensions of the phase space and one row per particle. The number of rows will overwrite the number of particles set in "inputdata_long.txt"
adjust_tower_pulses	A 1 implies that the times at which the tower coils are switched off are also determined by the solution vectors that will be sampled by the optimization algorithm. Otherwise the tower coils will be left switched on until the last coil of the succeeding decelerator stage will be switched off and the coils of the next tower will be switched on.
first_pulse_start_time_us	the starting time of the pulse of the first coil which is also the first pulse
min_decel_pulse_length_us	the minimum length of a pulse in the decelerator (this value will be added to the pulse durations sampled by the optimization algorithm in order to exclude the possibility that shorter pulses exist)

continued on next page

continued from previous page	
identifier of parameter	**description**
decel_pulse_overlap_us	the time interval with which two successive pulses in the decelerator overlap (switch-off time of the actual coil minus the switch-on time of the next coil)
tower_switch_on_overlap_us	the time interval with which two successive pulses in the towers overlap
decel_to_trap_pulse_overlap_us	the time interval with which the pulse of the last decelerator coil and the first pulse of the first trap coil overlap
trap_pulse_overlap_us	the time interval with which the first pulse of the first trap coil and the pulse of the second trap coil overlap
front_trap_coil_pulse_1_scale	a single factor that is multiplied with the reference current of the first trap coil specified in the file “inputdata_trap.txt” to determine the current of the first pulse of the first trap coil.
front_trap_coil_pulse_2_scale	a single factor that is multiplied with the reference current of the first trap coil specified in the file “inputdata_trap.txt” to determine the current of the second pulse of the first trap coil.
rear_trap_coil_pulse_scale	a single factor that is multiplied with the reference current of the second trap coil specified in the file “inputdata_trap.txt” to determine the current of the pulse of the second trap coil
gamma	weight for the weighted-sum-aggregation (see Section 5.2)
delta	weight for the weighted-sum-aggregation (see Section 5.2)
output_tof	A value of 1 imples that for each function evaluation the output of the laser detector will be written to the file “output_detection.txt” from which a time-of-flight profile can be made.
output_trap_region_observer	A value of 1 implies that for each function evaluation the numbers and energies of particles inside the region of the trap will be written to the file “output_trap_region.txt”.
output_trapped_particles	A value of 1 implies that for each function evaluation the end conditions and initial conditions of those particles that have been trapped are written to the file “output_trapped_parts.txt”.

Table A.3: description of the objective function input file “inputdata_obj.txt”

A.2 Input for H-trapping after deceleration by 12 coils

Here, the input to the simulation and optimization that has been used to obtain the results in Section 6.1 with 12 decelerator coils is presented. See Section A.1 for a description of the different input parameters and files.

A.2.1 Decelerator ("inputdata_long.txt")

identifier of parameter	**assigned value**
number_of_particles	25000
mass_factor	1
start_pos_z	0.0
start_radial_pos_on_disk	1
starting_velocity_m/s	500
init_beam_length_mm	1.0
long_temperature_K	1.45
trans_temp_K	0.003
number_of_stages	1
number_of_coils_per_stage	12
number_of_coils_in_a_tower	2
distance_to_middle_first_coil_mm	198.45
distance_between_stage_coils_mm	11
distance_between_stages_mm	55
distance_towercoils_mm	15
bound_z_mm	379
time_offset_in_us	0
incoupling_time_us	10
time_simulation_stops_us	771
timesteps_ns	10
ramp_time_coil_ns	8000
ramp_time_tower_coil_ns	8000
current_decelerator_A	300
reference_current_decelerator_A	300
current_tower_coils_A	250
reference_current_tower_coils_A	300
detector_pos_mm	349
detector_start_time_us	355
detector_radius_mm	0.35
detector_inverse_frequency_us	5
remove_unaccepted_lfs_particles	1
check_upper_bound_for_predicted_total_energy	1
upper_bound_for_predicted_total_energy_J	4.2726e-23

Table A.4: input to the decelerator simulation "inputdata_long.txt"

A.2.2 Trap ("inputdata_trap.txt")

identifier of parameter	**assigned value**
timesteps_ns	10
ramp_time_trap_coil_ns	10000

continued on next page

continued from previous page

identifier of parameter	assigned value
trap_sim_current_A	250
trap_coil1_reference_current_A	200
trap_coil2_reference_current_A	200
trappos_mm	349
distance_trap_coils_mm	20
time_simulation_stops_us	3000
max_trap_region_radius_z_mm	3
lower_bound_z_mm	0
upper_bound_z_mm	379
upper_bound_r_outside_trap_mm	2.75
upper_bound_r_in_trap_mm	4
scaling_polynomial_degree	0
trap_region_radius_z_mm	2
trap_region_radius_r_mm	4
upper_trap_abs_B_T	.255
region_observer_inverse_frequency_us	2

Table A.5: input to the trap simulation “inputdata_trap.txt”

A.2.3 Objective function (“inputdata_obj.txt”)

identifier of parameter	assigned value
zeeman_energy_file	zeeHF1Mf0.txt
read_particles_from_file	0
adjust_tower_pulses	0
first_pulse_start_time_us	345.323
min_decel_pulse_length_us	11
decel_pulse_overlap_us	3
tower_switch_on_overlap_us	19.756
decel_to_trap_pulse_overlap_us	11
trap_pulse_overlap_us	10
front_trap_coil_pulse_1_scale	.875
front_trap_coil_pulse_2_scale	-1
rear_trap_coil_pulse_scale	1
gamma	500
delta	1
output_tof	0
output_trap_region_observer	0
output_trapped_particles	0

Table A.6: input to the objective function evaluation “inputdata_obj.txt”

A.2.4 Initial conditions of the optimization (“initials.par”)

The pulse sequence that corresponds to the initial search point which is set in the file “initials.par” is given in section B.1.2.

```
function number 0              # 0 -> trap , 1 -> hfs
restarts 1  2                  # nbRestart incPopSizeFactor , read by
    example2.c
#
# Input parameter file for cmaes_t.
# Comments start with '#' or '%' until end of line.
# Actual recent parameter setting is written to file actparcmaes.par.
#

## ---- OBLIGATORY SETTINGS
# these settings, if read, overwrite the input values to cmaes_init

 N 14         # Problem dimension, overwrites parameter in cmaes_init
 initialX 14:   # Initial search point. Syntax: 1==read 1 number, see end of file.
    22.5785
    14.4429
    15.5390
    16.8130
    18.3201
    20.1386
    22.3945
    25.2898
    29.1994
    34.8911
    44.3476
    65.7761
    167.9480
    229.0000

 typicalX 1:   # Typical search point, overwritten by initialX
    0.0         #   these are the read value(s)
 initialStandardDeviations 1:   #  1==read only one number for all coordinates
   1.0e0                        #  numbers should not differ by orders of magnitude

## ---- OPTIONAL SETTINGS

# stop-Parameters can be changed online via signals.par

stopMaxFunEvals   1e299     # max number of f-evaluations, 900*(N+3)*(N+3) is default
# fac*maxFunEvals   1e0     # multiplies stopMaxFunEvals by read number, default is 1
stopMaxIter       1e299     # max number of iterations (generations), inf is default

# stopFitness 1e-9   # stop if function value is smaller than stopFitness
                     # commented == never stop on function value (default)
stopTolFun 0         # stop if function value differences are
                     # smaller than stopTolFun, default=1e-12
stopTolFunHist 0     # stop if function value differences of best values are
                     # smaller than stopTolFunHist, default=0
stopTolX 0           # stop if step sizes/steps in x-space are
                     # smaller than TolX, default=0
stopTolUpXFactor 1e3 # stop if std dev increases more than by TolUpXFactor, default 1e3

seed 2309           # 0 == by time, also regard maxTimeFractionForEigendecomposition
                  #   below, as for values smaller than one the outcome might not be
                  #   exactly reproducible even with the same seed

# diffMinChange 1 # Minimal coordinate wise standard deviation. Syntax see below.
#         1e-299    # Interferes with stopTolX termination criterion! 
     default=0


## ---- internal CPU-time related settings

maxTimeFractionForEigendecompostion 1  # maximal CPU-time fraction for 
     eigensystem
                      # decomposition. Large values (up to one) are 
                          better
                      # w.r.t. the number of function evaluations to 
                          reach a
                      # certain function value. Only >=1 yields exactly
                      # reproducible results.
                      # Default is 0.2==20% which should be faster than 
                          larger values
                      # w.r.t. the CPU-time to reach a certain function 
                          value.

# updatecov       1   # default is updating the eigensystem after
                      # every 1/ccov/N/10-th generation.
# fac*updatecov   3   # multiplier for updatecov

#resume allcmaes.dat   # reads restart distribution from given file


## ---- Strategy internal parameter ----
##       default values are set in readpara_SupplementDefaults()
## --- Selection related parameters
lambda 16         # number of offspring == samplesize
# mu
# weights       log # possible values: log (==default),
                    #   lin (==linearely decreasing), or equal

## --- Adaptation (distribution estimation) related parameters
# fac*damp      1  # increase or decrease damping for step size control
     .
# ccumcov       1  # default is 4/(N+4), 1 means no cumulation for p_c.
# mucov         1  # 1 means only rank 1 update of C
# fac*ccov      1  # multiplier for default learning rate for cov. 
     matrix


## ---- Syntax for input vectors xstart, sigma, mincoorstddev
#
#    N 7               # dimension must be defined before
#    xstart 3 :        # read 3 numbers from next lines, the colon is 
     superfluous
#       0.2 0.3
#       0.4 0.5 0.6
#       0.7
#
# reads the first three numbers starting from the
# line following the keyword "xstart" and recycles
# these numbers (or cut them) to length N, resulting in
#    xstart=[0.2 0.3 0.4 0.2 0.3 0.4 0.2]
# No comments are allowed between the numbers. I.e.
#
#    xstart 3      22 anything here is ok, 22 is disregarded  # still 
     ok
#       0.2 0.3      # this comment fails, if >2 numbers to be read
#       0.4 0.5 0.6
#
# would fail due to the comment between second and third number to
# be read.
```

A.2.5 Signal parameters for the optimization (“signals.par”)

```
#
# Comment characters are '%' and '#' to end of line.
# Uncomment one or more rows (and/or add an uncommented line)
# and save this file to induce an action.
#
# Function cmaes_ReadSignals reads and interprets the syntax
#   as given in this file
#

## --- modify termination condition, see also initials.par for more docu
#stop now                 # manual stop as soon as signals.par is parsed
# stopMaxFunEvals  4.5e6  # stop after given number of function evaluations
# stopMaxIter 3e3         # stop after given number of iterations (generations)
# stopFitness 1e-2      # stop if function value is smaller than stopFitness
# stopTolFun 0      # stop if function value differences are small
# stopTolFunHist 0 # stop if f-value differences between best values are small
# stopTolX 1e-1         # stop if step-sizes/steps in x-space are small
# stopTolUpXFactor 1e3 # stop if std dev increases more than by TolUpXFactor

# checkeigen  1         # number > 0 switches checking on, Check_Eigen() is O(n^3)!
# maxTimeFractionForEigendecompostion 0.2

#write resume allcmaes.dat  # write data for restart

## --- print data to stdout
##   syntax (in one row):
##   print <keyword for cmaes_WriteToFilePtr()>[+<another keyword>]
##         [<seconds>]
##   After the first iteration ONLY the action with the smallest
##     seconds value will take place.
##   For more info see write data below.

#  print gen+fitness    5
## "few" prints Fevals Fval Sigma Max&MinCoorDev AxisRatio MinOfD
  print fewinfo    200 # print every 200 seconds
  print few+clock    2 # clock: used processor time since start
#  print few        2
#  print fitness+xmean 3
#  print gen+few  20
#  print gen+few+few(diag(D)) 0
#  print few(diag(D))
#  print all 100

## --- write data to file (example2.c can also write into data file)
##  syntax (in one row):
##  write <keyword for cmaes_WriteToFilePtr()>[+<another keyword>]
##        [<filename>] [<seconds>]
##   After the first iteration ONLY the action with the smallest
##     seconds value will take place.

##  Default filename is tmpcmaes.dat. Default is seconds=1,
##  and during the first second or so it is 0 with a smooth
##  writing gap transition until up to one second. For
##  seconds=0 writing takes place every generation. For seconds < 0
##  writing is blocked after the first key where seconds < 0 was
##  written. Blocking lasts until all values for seconds are >=0
##   again. For keywords compare cmaes_interface.h and function
##   cmaes_WriteToFilePtr in cmaes.c.

##        KEYWORD(s)                                    FILE
     SECONDS to wait until next writing
# write with default format for plotting
write iter+eval+sigma+axisratio+fbestever+fitness+fmedian+fworst+mindii
     +idxmaxSD+maxSD+idxminSD+minSD    outcmaesfit.dat
write iter+eval+sigma+axisratio+stddevratio+diag(D)        outcmaesaxlen
     .dat
write iter+eval+sigma+idxmaxSD+idxminSD+stddev
     outcmaesstddev.dat
write iter+eval+sigma+0+0+xmean                            outcmaesxmean
     .dat
write iter+eval+sigma+0+fitness+xbest
     outcmaesxrecentbest.dat

##        KEYWORD(s)          FILE        SECONDS to wait until next
     writing

# write few+few(diag(D)) rescmaes.dat     0     # writes every
     generation
# write few+diag(D)      tmp.dat          0     # writes every
     generation
# write few+few(diag(D)) rescmaes.dat           # writes once per second
# write few+few(diag(D)) rescmaes.dat    −1     # writes once, blocks
     further writing
# write gen+xbest          xcmaes.dat     2
# write B                allcmaes.dat   100     # writes every 100
     seconds
# write all              allcmaes.dat   100
# write gen+arfitness    tmpcmaes.dat     0
```

A.3 H-trapping after deceleration by 24 coils

In the case of the optimization of hydrogen trapping after deceleration with 24 coils, the same input files have been used as for the optimization with 12 decelerator coils (see Section A.2) except for a single change of the parameter that determines the number of coils per stage in the file “inputdata_long.txt” and a different initial search point as well as initial standard deviation has been used.

A.3.1 Initial conditions of the optimization (“initials.par”)

The initial pulse sequence which belongs to the durations given in the file “initials.par” is given in section B.2.2.

```
function number 0              # 0 -> trap , 1 -> hfs
restarts 1  2                  # nbRestart incPopSizeFactor , read by
    example2.c
#
# Input parameter file for cmaes_t.
# Comments start with '#' or '%' until end of line.
# Actual recent parameter setting is written to file actparcmaes.par.
#

## —— OBLIGATORY SETTINGS
# these settings, if read, overwrite the input values to cmaes_init

 N 26         # Problem dimension, overwrites parameter in cmaes_init
 initialX 26:   # Initial search point. Syntax: 1==read 1 number, see
      end of file.
  5.29688587756995
  3.82230297072328
  3.90256326021757
  3.98622628559895
  4.07553677446297
  4.16773319683494
  4.26614580154031
  4.36921045499070
  4.47995535692043
  4.59673797382448
  4.72228758124703
  4.85798312059645
  5.00399840127872
  5.16236379965612
  5.33760245803301
  5.53082272360994
  5.74804314527996
  5.99416382825829
  6.27853486093690
  6.61437827766148
  7.02281994643178
  7.54254598925322
  8.24742384020610
  9.32094415818484
  15.4980643952721
  21.9146070008111

 typicalX 1:    # Typical search point, overwritten by initialX
    0.0         #    these are the read value(s)
 initialStandardDeviations  1:    #  1==read only one number for all
      coordinates
   0.3e0                          #  numbers should not differ by orders
         of magnitude

## —— OPTIONAL SETTINGS
```

```

# stop-Parameters can be changed online via signals.par

stopMaxFunEvals   1e299     # max number of f-evaluations, 900*(N+3)*(N+3) is default
# fac*maxFunEvals   1e0     # multiplies stopMaxFunEvals by read number, default is 1
stopMaxIter       1e299     # max number of iterations (generations), inf is default

# stopFitness 1e-9   # stop if function value is smaller than stopFitness
                     # commented == never stop on function value (default)
stopTolFun 0         # stop if function value differences are
                     # smaller than stopTolFun, default=1e-12
stopTolFunHist 0     # stop if function value differences of best values are
                     # smaller than stopTolFunHist, default=0
stopTolX 1e-9        # stop if step sizes/steps in x-space are
                     # smaller than TolX, default=0
stopTolUpXFactor 1e3 # stop if std dev increases more than by TolUpXFactor, default 1e3

seed 210          # 0 == by time, also regard maxTimeFractionForEigendecomposition
                  #   below, as for values smaller than one the outcome might not be
                  #   exactly reproducible even with the same seed

# diffMinChange 1 # Minimal coordinate wise standard deviation. Syntax see below.
#        1e-299   # Interferes with stopTolX termination criterion! default=0


## --- internal CPU-time related settings

maxTimeFractionForEigendecompostion 1  # maximal CPU-time fraction for eigensystem
                     # decomposition. Large values (up to one) are better
                     # w.r.t. the number of function evaluations to reach a
                     # certain function value. Only >=1 yields exactly
                     # reproducible results.
                     # Default is 0.2==20% which should be faster than larger values
                     # w.r.t. the CPU-time to reach a certain function value.

# updatecov       1  # default is updating the eigensystem after
                     # every 1/ccov/N/10-th generation.
# fac*updatecov   3  # multiplier for updatecov

#resume allcmaes.dat   # reads restart distribution from given file


## --- Strategy internal parameter ---
##      default values are set in readpara_SupplementDefaults()
## -- Selection related parameters
lambda 21         # number of offspring == samplesize
# mu
# weights       log # possible values: log (==default),
                    #   lin (==linearely decreasing), or equal
```

```

## -- Adaptation (distribution estimation) related parameters
# fac*damp       1  # increase or decrease damping for step size control.
# ccumcov        1  # default is 4/(N+4), 1 means no cumulation for p_c.
# mucov          1  # 1 means only rank 1 update of C
# fac*ccov       1  # multiplier for default learning rate for cov. matrix


## --- Syntax for input vectors xstart, sigma, mincoorstddev
#
#    N 7               # dimension must be defined before
#    xstart 3 :        # read 3 numbers from next lines, the colon is superfluous
#       0.2 0.3
#       0.4 0.5 0.6
#       0.7
#
# reads the first three numbers starting from the
# line following the keyword "xstart" and recycles
# these numbers (or cut them) to length N, resulting in
#    xstart=[0.2 0.3 0.4 0.2 0.3 0.4 0.2]
# No comments are allowed between the numbers. I.e.
#
#    xstart 3      22 anything here is ok, 22 is disregarded  # still ok
#       0.2 0.3     # this comment fails, if >2 numbers to be read
#       0.4 0.5 0.6
#
# would fail due to the comment between second and third number to
# be read.
```

Appendix B

Pulse sequences

All pulses stated in this section correspond to an in-coupling time (offset) of $10\mu s$.

B.1 H-trapping after deceleration by 12 coils

B.1.1 Optimized pulse sequence

decelerator (“0”)

switch-on time in μs	**switch-off time in μs**
345.3230	402.9799
399.9799	429.0207
426.0207	469.1713
466.1713	492.6993
489.6993	505.6694
502.6694	535.2028
532.2028	578.1331
575.1331	612.8122
609.8122	655.9754
652.9754	696.6058
693.6058	730.5282
727.5282	825.1936

Table B.1: optimized pulse sequence of the decelerator from the file “0”

first trap coil (“trap_pulses1.txt”)

switch-on time in μs	**switch-off time in μs**	**current factor**
814.1936	1027.8840	0.875
1340.0835	10000	-1

Table B.2: optimized pulse sequence of the first trap coil from the file “trap_pulses1.txt”

second trap coil (“trap_pulses2.txt”)

switch-on time in μs	**switch-off time in μs**	**current factor**
1017.8840	10000	1

Table B.3: optimized pulse sequence of the second trap coil from the file "trap_pulses2.txt"

B.1.2 Initial pulse sequence

decelerator ("0")

switch-on time in μs	**switch-off time in μs**
345.3230	378.9015
375.9015	401.3444
398.3444	424.8834
421.8834	449.6964
446.6964	476.0165
473.0165	504.1551
501.1551	534.5496
531.5496	567.8394
564.8394	605.0388
602.0388	647.9299
644.9299	700.2775
697.2775	774.0536

Table B.4: initial pulse sequence of the decelerator from the file "0"

first trap coil ("trap_pulses1.txt")

switch-on time in μs	**switch-off time in μs**	**current factor**
763.0536	931.0016	0.875
1170.0016	10000	-1

Table B.5: initial pulse sequence of the first trap coil from the file "trap_pulses1.txt"

second trap coil ("trap_pulses2.txt")

switch-on time in μs	**switch-off time in μs**	**current factor**
921.0016	10000	1

Table B.6: initial pulse sequence of the second trap coil from the file "trap_pulses2.txt"

B.2 Input for H-trapping after deceleration by 24 coils

B.2.1 Optimized pulse sequence

decelerator (“0”)

switch-on time in μs	switch-off time in μs
345.3230	387.9409
384.9409	424.1470
421.1470	432.1978
429.1978	452.2151
449.2151	470.5931
467.5931	502.9512
499.9512	520.9175
517.9175	553.2996
550.2996	575.3429
572.3429	601.2583
598.2583	654.0715
651.0715	677.3906
674.3906	708.2098
705.2098	733.3093
730.3093	766.8119
763.8119	800.0967
797.0967	841.1879
838.1879	905.6646
902.6646	947.2420
944.2420	1009.4574
1006.4574	1058.8117
1055.8117	1096.9925
1093.9925	1186.1209
1183.1209	1309.0050

Table B.7: optimized pulse sequence of the decelerator from the file “0”

first trap coil (“trap_pulses1.txt”)

switch-on time in μs	switch-off time in μs	current factor
1298.0050	1606.6256	0.875
2000.4368	10000	-1

Table B.8: optimized pulse sequence of the first trap coil from the file “trap_pulses1.txt”

second trap coil (“trap_pulses2.txt”)

switch-on time in μs	switch-off time in μs	current factor
1596.6256	10000	1

Table B.9: optimized pulse sequence of the second trap coil from the file “trap_pulses2.txt”

B.2.2 Initial pulse sequence

decelerator ("0")

switch-on time in μs	switch-off time in μs
345.3230	389.1925
386.1925	425.7947
422.7947	434.0860
431.0860	454.4054
451.4054	471.3275
468.3275	503.8294
500.8294	519.9676
516.9676	557.9850
554.9850	581.0332
578.0332	608.4290
605.4290	659.3301
656.3301	680.9730
677.9730	709.1366
706.1366	735.5728
732.5728	769.7857
766.7857	801.3082
798.3082	843.4819
840.4819	905.8524
902.8524	949.4009
946.4009	1007.7085
1004.7085	1060.8616
1057.8616	1100.7087
1097.7087	1184.5200
1181.5200	1304.3253

Table B.10: initial pulse sequence of the decelerator from the file "0"

first trap coil ("trap_pulses1.txt")

switch-on time in μs	switch-off time in μs	current factor
1301.3253	1601.8240	0.875
1995.5059	10000	-1

Table B.11: initial pulse sequence of the first trap coil from the file "trap_pulses1.txt"

second trap coil ("trap_pulses2.txt")

switch-on time in μs	switch-off time in μs	current factor
1598.8240	10000	1

Table B.12: initial pulse sequence of the second trap coil from the file "trap_pulses2.txt"

Appendix C

Source code of the simulation and optimization (version 2.56)

From the next page, the implementation of the simulation and optimization of Zeeman deceleration in the C++ programming language is given. The Makefile stated in the end of this chapter can be used as an input to the program 'GNU Make'[39] that provides a convenient way to compile the source code. This code depends on the implementation of the CMA-ES optimization algorithm by N. Hansen which is stated in Appendix D.

```
// longdecel.cxx: v2.4, 2009−05−08, Yves Salathe
#include "headers/zeemandecel_builder.h"
#include "headers/magnetictrap_builder.h"
#include "headers/leapfrog_integrator.h"
//#include "headers/symplectic_euler_integrator.h"
//#include "headers/verlet_integrator.h"
#include "headers/matrix.h"
#include <time.h>
#include <cstdlib>

using namespace std;
using namespace simulation;

//////////////////////////
// function definitions //
//////////////////////////
void printUsage(const char *arg0);

/////////////////////
///**MAIN PROGRAM**//
/////////////////////
int main(int argc, char *argv[])
{
    ////////////////////////////////////////////////////////////////////
    //measure runtime                                                 //
    ////////////////////////////////////////////////////////////////////
    time_t runtime_start = time(0);

    ////////////////////////////////////////////////////////////////////
    //parse command line arguments                                    //
    ////////////////////////////////////////////////////////////////////
    std::string particlesFileName;
    bool outputTrajectories = false;
    bool generatePulses = false;
    bool storeEndConditions = false;
    double trajectoryResolution = 0;
    size_t seed = 4589; // seed of the random number generator
    // Note: the seed can be set as a command line algorithm
    int c;
    string pop_state("F1Mf0");
    while ((c = getopt(argc, argv, "t:p:s:z:ge")) != −1)
    {
        switch (c)
        {
        case ’t’:
            outputTrajectories = true;
            trajectoryResolution = std::atof(optarg);
            break;
        case ’p’:
            particlesFileName = optarg;
            break;
        case ’s’:
            seed = std::atoi(optarg);
            break;
        case ’z’:
            pop_state = optarg;
            break;
        case ’g’:
            generatePulses = true;
            break;
        case ’e’:
            storeEndConditions = true;
            break;
        default:
            printUsage(argv[0]);
            return 0;
            break;
        }
    }

    ///////////////////////////////////////////////
    // create the random number generator       //
    ///////////////////////////////////////////////
    boost::mt19937 rng;
    rng.seed((boost::mt19937::result_type)seed);
```

```

        ////////////////////////////////////////////////////////////////////
        //create the generic simulation object                            //
        ////////////////////////////////////////////////////////////////////
        ZeemanDecelBuilder* decelBuilder;
    try {
            decelBuilder = new ZeemanDecelBuilder();

        ////////////////////////////////////////////////////////////////
        //read in b−field of the decelerator and tower coils         //
        ////////////////////////////////////////////////////////////////
        decelBuilder−>readBFieldsFromFiles();

        ///////////////////////////////////////////////
        // create particle population(s)            //
        ///////////////////////////////////////////////
        math::Matrix<ParticlePopulation::real_t> zee;
        bool lfs = false;
        if (pop_state.compare("F0Mf0") == 0) {
            math::Matrix<ZeemanDecel::real_t> zf0M0("zeeHF0Mf0.txt",2);
            zee = zf0M0;
            lfs = false;
        } else if (pop_state.compare("F1Mfm1") == 0) {
            math::Matrix<ZeemanDecel::real_t> zf1Mm1("zeeHF1Mfm1.txt",2);
            zee = zf1Mm1;
            lfs = false;
        } else if (pop_state.compare("F1Mf0") == 0) {
            math::Matrix<ZeemanDecel::real_t> zf1M0("zeeHF1Mf0.txt",2);
            zee = zf1M0;
            lfs = true;
        } else if (pop_state.compare("F1Mfp1") == 0) {
            math::Matrix<ZeemanDecel::real_t> zf1Mp1("zeeHF1mfp1.txt",2);
            zee = zf1Mp1;
            lfs = true;
        } else {
            throw ExWrongInput();
        }
        ParticlePopulation pop(zee);
        // set the atomic mass factor of the members of the population
        pop.setMassFactor(decelBuilder−>getDefaultMassFactor());
        // set the flag that indicates whether the particles are
        // low−field−seeking or not
        pop.setLFS(lfs);
        // add the population to the simulation
        size_t pop_id = decelBuilder−>simulation−>addPopulation(pop);

        if (particlesFileName.empty())
        {
            // (the first argument sets the number of particles)
            // generate random particles in state F=1, M=0
            // TODO: provide a way to say how many particles from
            //        which population should be generated
            decelBuilder−>simulation−>generateRandomParticles(
                    decelBuilder−>getNumberOfParticlesPerPopulation(),
                    pop_id,rng);
        } else
        {
            // read particles from file (assume state F=1, M=0)
            // TODO: provide a way to determine the population
            math::Matrix<ZeemanDecel::real_t>
            particles(particlesFileName.c_str(),6);
            decelBuilder−>simulation−>generateParticlesFromMatrix(
                    particles, pop_id);
        }


        ////////////////////////////////////////////////////////////////////
        // pulsing of the decelerator                                     //
        ////////////////////////////////////////////////////////////////////
        size_t m = decelBuilder−>simulation−>engine.getNumbStages();
        DecelPulseGenObserver<ParticleSim<ZeemanDecel> >* decelPG = 0;
        if (generatePulses == false) {
            /////////////////////////////////////////
            //read in pulsing of decelerator stages//
            /////////////////////////////////////////
        decelBuilder->simulation->engine.readStagePulsesFiles();
        if (m>1)
        {
            //////////////////////////////////////
            //read in pulsing of the tower coils//
            //////////////////////////////////////
            math::Matrix<ZeemanDecel::real_t> towerpulses("towers.txt",2);
            // The second argument to setTowerPulses describes an offset
            // that is added to the times of every tower pulse.
            decelBuilder->simulation->engine.setTowerPulses(towerpulses);
        }
        //////////////////////////////////////////////////
        // if one wants to simulate the trap            //
        // then stop the decelerator simulation after   //
        // the end of the second last pulse             //
        //////////////////////////////////////////////////
#ifdef __TRAP__
        size_t n = decelBuilder->simulation->engine.getNumbStageCoils();
        decelBuilder->simulation->setEndTime(decelBuilder->simulation
                ->engine.getStagePulses(m-1)(n-2,1)
                + decelBuilder->simulation->engine.getStageRampOff());
#endif

    } else {
        /////////////////////////////////////////
        // initialize pulse generator          //
        /////////////////////////////////////////
        decelPG = decelBuilder->addPulseGenObserver();
        decelPG->initializePulses();
        //////////////////////////////////////////////////
        // if one wants to simulate the trap            //
        // then stop the decelerator simulation after   //
        // the end of the second last pulse             //
        //////////////////////////////////////////////////
#ifdef __TRAP__
        size_t n = decelBuilder->simulation->engine.getNumbStageCoils();
        decelPG->stopSimulationAfterPulse(m-1,n-2);
#endif
    }

    ////////////////////////////////////////
    //create and add some observers       //
    ////////////////////////////////////////
    decelBuilder->addLaserObserver();
    if (outputTrajectories) {
        TrajectoriesObserver<ParticleSim<ZeemanDecel> >* traj
                = decelBuilder->addTrajectoriesObserver();
        traj->setInverseFrequency(trajectoryResolution);
    }

    //////////////////////////////////////////////////
    //create integrator                             //
    //////////////////////////////////////////////////
    //VerletIntegrator<ParticleSim<ZeemanDecel> >* integrator =
    //        new VerletIntegrator<ParticleSim<ZeemanDecel> >();
    LeapfrogIntegrator<ParticleSim<ZeemanDecel> >* integrator =
            new LeapfrogIntegrator<ParticleSim<ZeemanDecel> >();

    ////////////////////////////////////////////////////////////
    //simulate the Zeeman-Decelerator                         //
    ////////////////////////////////////////////////////////////
    decelBuilder->simulation->runSimulation(*integrator);

#ifdef __TRAP__
    ////////////////////////////////////////////////////////////
    // create a simulation object based on the                //
    // decelerator simulation but with the trap as the engine //
    ////////////////////////////////////////////////////////////
    MagneticTrapBuilder* trapBuilder;
    trapBuilder = new MagneticTrapBuilder(*decelBuilder->simulation);

    //////////////////////////////////////////////////////////////
    // read in B-fields for the trap simulation                 //
    //////////////////////////////////////////////////////////////
    trapBuilder->readDecelBFieldFromFile();
```

```
    trapBuilder->readTrapBFieldFromFile();

    //////////////////////////////////////////////////////////////
    // pulsing of the trap coils                                //
    //////////////////////////////////////////////////////////////
    TrapPulseGenObserver<ParticleSim<MagneticTrap> >* trapPG;
    if (!generatePulses) {
        trapBuilder->readFirstCoilPulsingFromFile();
        trapBuilder->readSecondCoilPulsingFromFile();
    } else {
        /////////////////////////////////////////
        // initialize pulse generator          //
        /////////////////////////////////////////
        trapPG = trapBuilder->addPulseGenObserver(*decelPG);
    }
    //////////////////////////////////////////////////////////////
    // add trap region observer                                 //
    //////////////////////////////////////////////////////////////
    RegionObserver<ParticleSim<MagneticTrap> >* trapRegionObserver
            = trapBuilder->addTrapRegionObserver();

    //////////////////////////////////////////////////
    //create integrator                             //
    //////////////////////////////////////////////////
    //VerletIntegrator<ParticleSim<MagneticTrap> >* integrator =
    //        new VerletIntegrator<ParticleSim<MagneticTrap> >();
    LeapfrogIntegrator<ParticleSim<MagneticTrap> >* integrator2 =
            new LeapfrogIntegrator<ParticleSim<MagneticTrap> >();

    //////////////////////////////////////////////////
    //start the trap-simulation                     //
    //////////////////////////////////////////////////
    trapBuilder->simulation->runSimulation(*integrator2);

    //////////////////////////////////////////////////////
    // output information about the trapped particles //
    //////////////////////////////////////////////////////
    std::ofstream ofstr_trapped ("output_trapped_parts.txt");
    trapRegionObserver->outputTrappedParticles(ofstr_trapped);
    ofstr_trapped.close();

    ////////////////////////////////////////////////////////////
    // if desired, write generated pulses to their files    //
    ////////////////////////////////////////////////////////////
    if(generatePulses) {
        // write trap pulses down to the files
        trapBuilder->simulation->engine.coil1.getOrigPulsesMatrix()
                .writeToFile("trap_pulses1.txt");
        trapBuilder->simulation->engine.coil2.getOrigPulsesMatrix()
                .writeToFile("trap_pulses2.txt");
        // update the decelerator simulation
        size_t n = decelBuilder->simulation->engine.getNumbStageCoils();
        decelBuilder->simulation->engine.setActualStagePulseEndTime(
                m-1, n-1, trapBuilder->simulation->engine.lastDecelCoil
                        .getPulse(0).endTime);
    }
#endif

    ////////////////////////////////////////////////////////////
    // if desired, write generated pulses to their files    //
    ////////////////////////////////////////////////////////////
    if(generatePulses) {
        decelBuilder->simulation->engine.writeOrigStagePulsesFiles();
        if (decelBuilder->simulation->engine.getNumbStages()>1) {
            decelBuilder->simulation->engine.getOrigTowerPulses()
                    .writeToFile("towers.txt");
        }
    }

    ////////////////////////////////////////////////////////////
    // if desired, store end conditions of the particles    //
    ////////////////////////////////////////////////////////////
    if (storeEndConditions) {
        ofstream ofstr("output_endconditions.txt");
#ifdef __TRAP__
```

```
        typedef ParticleSim<MagneticTrap>::particle_list_t
                particle_list_t;
        const particle_list_t& particles
                = trapBuilder->simulation->getParticles();
#else
        typedef ParticleSim<ZeemanDecel>::particle_list_t
                particle_list_t;
        const particle_list_t& particles
                = decelBuilder->simulation->getParticles();
#endif
        for(particle_list_t::const_iterator p = particles.begin();
                p != particles.end(); ++p) {
            ofstr   << p->pos.getX() << " "
                    << p->pos.getY() << " "
                    << p->pos.getZ() << " "
                    << p->vel.getX() << " "
                    << p->vel.getY() << " "
                    << p->vel.getZ() << endl;
        }
        ofstr.close();
    }

#ifdef __TRAP__
    ///////////////////////////
    // delete the builder    //
    ///////////////////////////
    delete trapBuilder;

    ///////////////////////////
    // delete the integrator //
    ///////////////////////////
    delete integrator2;
#endif

    ///////////////////////////
    // delete the builder    //
    ///////////////////////////
    delete decelBuilder;

    ///////////////////////////
    // delete the integrator //
    ///////////////////////////
    delete integrator;

    ///////////////////////////////////////////
    //output the runtime measurement         //
    ///////////////////////////////////////////
    clock_t cputime = clock();
    time_t runtime_end = time(0);
    std::ofstream ofstr ("runtime.txt");
    ofstr << "Start Date: " << asctime(localtime(&runtime_start));
    ofstr << "End Date: " << asctime(localtime(&runtime_end));
    ofstr << "Elapsed real time: " << (uintmax_t)(runtime_end-runtime_start) << " sec\n"
;
    ofstr << "Elapsed cpu time: " << cputime/CLOCKS_PER_SEC << " sec\n";
    ofstr.close();

} catch(InputData::ExInputNotSpecified &ex) {
        std::cerr << "In file: " << ex.filename << std::endl
                << "Error: " << ex.id << " has to be specified\n";
        return 1;
}

    return 0;
}

void printUsage(const char *arg0)
{
    std::cerr << "usage: " << arg0
            << " [-t <resolution>] [-p <filename>] [-s <seed>]"
            << " [-z state] [-g] [-e]\n";
}
```

```cpp
#include <stdio.h>
#include <string.h> /* strncmp */
#include <math.h>
#include <stdlib.h>
#include <limits>
#include "cma/cmaes_interface.h"
#include "headers/input.h"
#include "headers/objective_trap.h"
#include "headers/objective_hfs.h"

double * optimize(double(*pFun)(double const *, unsigned int, unsigned int),
                  int number_of_restarts,
                  double increment_factor_for_population_size,
                  char *input_parameter_filename);

extern void   random_init( random_t *, long unsigned seed /*=0=clock*/);
extern void   random_exit( random_t *);
extern double random_Gauss( random_t *); /* (0,1)−normally distributed */
extern double random_Uniform( random_t *); /* uniform in [0,1] */

using namespace simulation;

/*___________________________________________________________________________
//___________________________________________________________________________
//
// reads from file "initials.par" here and in cmaes_init()
//___________________________________________________________________________
*/
int main(int argn, char **args)
{
  typedef double (*pfun_t)(double const *, unsigned int, unsigned int);
  typedef void (*pfun_init_t)();
  // array (range) of pointer to objective function
  pfun_t rgpFun[99];
  // array (range) of pointer to a function
  // that initializes the objective function
  pfun_init_t rgpFun_init[99];
  char *filename = "initials.par"; /* input parameter file */
  FILE *fp = NULL;
  int nb = 0, nbrestarts = 0;
  double incpopsize = 2;
  int maxnb, ret=1;
  char c;
  double *x;

  /* Put together objective functions */
  rgpFun[0] = ObjectiveFunctionTrap::eval;
  rgpFun_init[0] = ObjectiveFunctionTrap::initialize;
  rgpFun[1] = ObjectiveFunctionHFS::eval;
  rgpFun_init[1] = ObjectiveFunctionHFS::initialize;
  maxnb = 1;

  /* Read objective function number and number of restarts from file */
  fp = fopen(filename, "r");
  if (fp) {
    fscanf(fp, " function number %d ", &nb);
    /* go to next line, a bit sloppy */
    for (c = ' ', ret = 1; c != '\n' && c != '\0' && c != EOF && ret && ret != EO
F;
         ret=fscanf(fp, "%c", &c))
      ;
    fscanf(fp, " restarts %d %lf", &nbrestarts, &incpopsize);
    fclose(fp);
    if (nb < 0 || nb > maxnb)
      nb = 0;
    if (nbrestarts < 0)
      nbrestarts = 0;
  } else
    printf("main(): could not open %s to read function number", filename);

  /* initialize function */
    try {
        rgpFun_init[nb]();
    } catch (InputData::ExInputNotSpecified &ex) {
        std::cerr << "In file: " << ex.filename << std::endl
```

```cpp
                  << "Error: " << ex.id << " has to be specified\n";
        return 1;
    }
/* Optimize function */

  printf("number of restarts: %d, increasing population size by: %lf\n\n",
        nbrestarts, incpopsize);
  x = optimize(rgpFun[nb], nbrestarts, incpopsize, filename);

  free(x);

  return 0;

} /* main() */

/*___________________________________________________________________________
//___________________________________________________________________________
//
// Somewhat extended interface for optimizing pFun with cmaes_t
// implementing a restart procedure with increasing population size
//___________________________________________________________________________
*/

double * optimize(double(*pFun)(double const *, unsigned int, unsigned int),
        int nrestarts, double incpopsize, char * filename)
{
  cmaes_t evo;       /* the optimizer */
  double *const*pop; /* sampled population */
  double *fitvals;   /* objective function values of sampled population */
  double fbestever=0, *xbestever=NULL; /* store best solution */
  double fmean;
  int i, irun,
    lambda = 0,      /* offspring population size, 0 invokes default */
    countevals = 0;  /* used to set for restarts */
  char const * stop; /* stop message */
  unsigned int seed = 0;   /* seed of the random number generator
                     * 0 means either by time or by the value
                     * given in initials.par */

  for (irun = 0; irun < nrestarts+1; ++irun) /* restarts */
    {
      /* Parameters can be set in three ways. Here as input parameter
       * to cmaes_init, as value read from initials.par in readpara_init
       * during initialization, and as value read from signals.par by
       * calling cmaes_ReadSignals explicitely.
       */
      fitvals = cmaes_init(&evo, 0, NULL, NULL, 0, lambda,seed,filename); /* all
ocs fitvals */
      printf("%s\n", cmaes_SayHello(&evo));
      evo.countevals = countevals; /* a hack, effects the output and termination
 */
      cmaes_ReadSignals(&evo, "signals.par"); /* write initial values, headers in c
ase */

      while(!(stop=cmaes_TestForTermination(&evo)))
        {
          /* Generate population of new candidate solutions */
          pop = cmaes_SamplePopulation(&evo); /* do not change content of pop */

          /* Here optionally handle constraints etc. on pop. You may
           * call cmaes_ReSampleSingle(&evo, i) to resample the i−th
           * vector pop[i], see below.  Do not change pop in any other
           * way. You may also copy and modify (repair) pop[i] only
           * for the evaluation of the fitness function and consider
           * adding a penalty depending on the size of the
           * modification.
           */
          unsigned int popsize = cmaes_Get(&evo, "popsize");
          countevals = cmaes_Get(&evo, "eval");
          /* compute ids and seeds for the function evaluations */
          unsigned int id[popsize];
          unsigned int fe_seed[popsize];
          for (i = 0; i < popsize; ++i) {
              id[i] = countevals+i+1;
              fe_seed[i] = (unsigned int)(random_Uniform(&evo.rand)
                           *std::numeric_limits<unsigned int>::max());
          }
          /* Compute fitness value for each candidate solution */
          #pragma omp parallel for
          for (i = 0; i < popsize; ++i) {
            /* You may resample the solution i until it lies within the
               feasible domain here, e.g. until it satisfies given
               box constraints (variable boundaries). The function
               is_feasible() needs to be user-defined.
               Assumptions: the feasible domain is convex, the optimum
               is not on (or very close to) the domain boundary,
               initialX is feasible and initialStandardDeviations are
               sufficiently small to prevent quasi-infinite looping.
            */
            /* while (!is_feasible(pop[i]))
                 cmaes_ReSampleSingle(&evo, i);
            */
            fitvals[i] = (*pFun)(pop[i],id[i],fe_seed[i]);
          }

          /* update search distribution */
          cmaes_UpdateDistribution(&evo, fitvals);

          /* read control signals for output and termination */
          cmaes_ReadSignals(&evo, "signals.par"); /* from file signals.par */

          fflush(stdout);
        } /* while !cmaes_TestForTermination(&evo) */

      lambda = incpopsize * cmaes_Get(&evo, "lambda");   /* needed for the restar
t */
      seed = (unsigned int)(random_Uniform(&evo.rand)
            *std::numeric_limits<unsigned int>::max()); /* ditto */
      countevals = cmaes_Get(&evo, "eval");     /* ditto */

      /* print some "final" output */
      printf("%.0f generations, %.0f fevals (%.1f sec): f(x)=%g\n",
             cmaes_Get(&evo, "gen"), cmaes_Get(&evo, "eval"),
             evo.eigenTimings.totaltime,
             cmaes_Get(&evo, "funval"));
      printf("  (axis-ratio=%.2e, max/min-stddev=%.2e/%.2e)\n",
             cmaes_Get(&evo, "maxaxislen") / cmaes_Get(&evo, "minaxislen"),
             cmaes_Get(&evo, "maxstddev"), cmaes_Get(&evo, "minstddev")
             );
      printf("Stop (run %d):\n%s\n",  irun+1, cmaes_TestForTermination(&evo));

      /* write some data */
      cmaes_WriteToFile(&evo, "all", "allcmaes.dat");

      /* keep best ever solution */
      if (irun == 0 || cmaes_Get(&evo, "fbestever") < fbestever) {
        fbestever = cmaes_Get(&evo, "fbestever");
        xbestever = cmaes_GetInto(&evo, "xbestever", xbestever); /* alloc mem if n
eeded */
      }
      /* best estimator for the optimum is xmean, therefore check */
      unsigned int fe_seed = (unsigned int)(random_Uniform(&evo.rand)
                       *std::numeric_limits<unsigned int>::max());
      fmean = (*pFun)(cmaes_GetPtr(&evo, "xmean"),
                countevals+1, fe_seed);
      if (fmean < fbestever) {
        fbestever = fmean;
        xbestever = cmaes_GetInto(&evo, "xmean", xbestever);
      }

      cmaes_exit(&evo); /* does not effect the content of stop string and xbeste
ver */

      /* abandon restarts if target fitness value was achieved or MaxFunEvals re
ached */
      if (stop) /* as it can be NULL */ {
        if (strncmp(stop, "Fitness", 7) == 0 || strncmp(stop, "MaxFunEvals", 11) ==
 0)
          break;
      }
```

```
      if (strncmp(stop, "Manual", 6) == 0) {
        printf("Press RETURN to start next run\n"); fflush(stdout);
        getchar();
      }
    } /* for restarts */

  /* print the best solution vector */
  size_t DIM = cmaes_Get(&evo,"dimension");
  printf("\nthe best solution vector ever found is: \n");
  for(size_t i = 0; i<DIM; ++i) {
      printf("%lf ",xbestever[i]);
  }

  return xbestever; /* was dynamically allocated, should be freed in the end */
}

/*
  05/10/05: revised buggy comment on handling constraints by resampling
*/
```

```cpp
// objective_trap.cxx: v2.4, 2009-05-08, Yves Salathe
#include "headers/objective_trap.h"
#include "headers/zeemandecel_builder.h"
#include "headers/magnetictrap_builder.h"
#include "headers/leapfrog_integrator.h"
//#include "headers/symplectic_euler_integrator.h"
//#include "headers/verlet_integrator.h"
#include "headers/matrix.h"
#include "headers/exceptions.h"
#include "headers/assert.h"
#include "headers/nullstream.h"
#include <time.h>
#include <cstdlib>
#include <sstream>
#include <string>
#include <limits>

using namespace std;
using namespace simulation;

InputData ObjectiveFunctionTrap::indata_decel;
InputData ObjectiveFunctionTrap::indata_trap;
math::Matrix<ObjectiveFunctionTrap::real_t> ObjectiveFunctionTrap::zee;
math::Grid2d<ObjectiveFunctionTrap::real_t>
        ObjectiveFunctionTrap::stageBz;
math::Grid2d<ObjectiveFunctionTrap::real_t>
        ObjectiveFunctionTrap::stageBr;
math::Grid2d<ObjectiveFunctionTrap::real_t>
        ObjectiveFunctionTrap::towersBz;
math::Grid2d<ObjectiveFunctionTrap::real_t>
        ObjectiveFunctionTrap::towersBr;
math::Matrix<ObjectiveFunctionTrap::real_t>
        ObjectiveFunctionTrap::BFieldDecelAxRad;
math::Matrix<ObjectiveFunctionTrap::real_t>
        ObjectiveFunctionTrap::BFieldTowersAxRad;
math::Matrix<ObjectiveFunctionTrap::real_t>
        ObjectiveFunctionTrap::BFieldTrapAxRad;
math::Matrix<ObjectiveFunctionTrap::real_t>
        ObjectiveFunctionTrap::particles;
std::vector< math::Matrix<ObjectiveFunctionTrap::real_t> >
        ObjectiveFunctionTrap::origStagePulses;
math::Matrix<ObjectiveFunctionTrap::real_t>
        ObjectiveFunctionTrap::origTowerPulses;
int ObjectiveFunctionTrap::readParticlesFromFile;
int ObjectiveFunctionTrap::adjustTowerPulses;
ObjectiveFunctionTrap::real_t
        ObjectiveFunctionTrap::firstPulseStart;
ObjectiveFunctionTrap::real_t
        ObjectiveFunctionTrap::decelPulseOverlap;
ObjectiveFunctionTrap::real_t
        ObjectiveFunctionTrap::decelToTrapPulseOverlap;
ObjectiveFunctionTrap::real_t
        ObjectiveFunctionTrap::trapPulseOverlap;
ObjectiveFunctionTrap::real_t
        ObjectiveFunctionTrap::minDecelPulseLength;
ObjectiveFunctionTrap::real_t
        ObjectiveFunctionTrap::towerSwitchOnOverlap;
ObjectiveFunctionTrap::real_t
        ObjectiveFunctionTrap::frontTrapCoilFirstPulseScale;
ObjectiveFunctionTrap::real_t
        ObjectiveFunctionTrap::frontTrapCoilSecondPulseScale;
ObjectiveFunctionTrap::real_t
        ObjectiveFunctionTrap::rearTrapCoilPulseScale;
ObjectiveFunctionTrap::real_t
        ObjectiveFunctionTrap::gamma;
ObjectiveFunctionTrap::real_t
        ObjectiveFunctionTrap::delta;
int ObjectiveFunctionTrap::outputTOF;
int ObjectiveFunctionTrap::outputTrapRegion;
int ObjectiveFunctionTrap::outputTrappedParts;

////////////////////////////////////////////////
// initialization of the objective function   //
// before the optimization                    //
////////////////////////////////////////////////
```

```cpp
void ObjectiveFunctionTrap::initialize()
{
    ////////////////////////////////////////////////////////////////////
    // read parameter file for objective function evaluation        //
    ////////////////////////////////////////////////////////////////////
    const char inputFileName[] = "inputdata_obj.txt";
    InputData param(inputFileName);
    string zeeFileName = param.getString("zeeman_energy_file");
    readParticlesFromFile = param.getInt("read_particles_from_file");
    Assert<ExWrongInput>(readParticlesFromFile==0
            || readParticlesFromFile==1);
    adjustTowerPulses = param.getInt("adjust_tower_pulses");
    Assert<ExWrongInput>(adjustTowerPulses==0
            || adjustTowerPulses==1);
    firstPulseStart = param.getReal("first_pulse_start_time_us");
    minDecelPulseLength = param.getReal("min_decel_pulse_length_us");
    decelPulseOverlap = param.getReal("decel_pulse_overlap_us");
    towerSwitchOnOverlap = param.getReal("tower_switch_on_overlap_us");
    decelToTrapPulseOverlap =
            param.getReal("decel_to_trap_pulse_overlap_us");
    trapPulseOverlap = param.getReal("trap_pulse_overlap_us");
    frontTrapCoilFirstPulseScale =
            param.getReal("front_trap_coil_pulse_1_scale");
    frontTrapCoilSecondPulseScale =
            param.getReal("front_trap_coil_pulse_2_scale");
    rearTrapCoilPulseScale = param.getReal("rear_trap_coil_pulse_scale");
    gamma = param.getReal("gamma");
    delta = param.getReal("delta");
    outputTOF = param.getInt("output_tof");
    Assert<ExWrongInput>(outputTOF==0 || outputTOF==1);
    outputTrapRegion = param.getInt("output_trap_region_observer");
    Assert<ExWrongInput>(outputTrapRegion==0 || outputTrapRegion==1);
    outputTrappedParts = param.getInt("output_trapped_particles");
    Assert<ExWrongInput>(outputTrappedParts==0 || outputTrappedParts==1);

    /////////////////////////////////////////////////////////
    // read the input data of the decelerator and the trap //
    /////////////////////////////////////////////////////////
    indata_decel.readFromFile("inputdata_long.txt");
    indata_trap.readFromFile("inputdata_trap.txt");

    ////////////////////////////////////////////
    // read in Zeeman-effect                 //
    ////////////////////////////////////////////
    zee.readFromFile(zeeFileName.c_str(),2);

    ////////////////////////////////////////////
    // read in the magnetic fields           //
    ////////////////////////////////////////////
    //BFieldStages.readGridFromFile("bmap.txt");
    //BFieldTowers.readGridFromFile("tmap.txt");

    // -- decelerator
    BFieldDecelAxRad.readFromFile("decelmapaxrho.txt",4);
    Assert<ExWrongInput>(BFieldDecelAxRad.rows() > 0 &&
            BFieldDecelAxRad.cols() == 4);
    // extract axial part of the field
    stageBz.readGridFromMatrix(BFieldDecelAxRad,2);
    // extract radial part of the field
    stageBr.readGridFromMatrix(BFieldDecelAxRad,3);

    // -- towers (if any)
    if (indata_decel.getInt("number_of_stages") > 1) {
        BFieldTowersAxRad.readFromFile("towermapaxrho.txt",4);
        Assert<ExWrongInput>(BFieldTowersAxRad.rows() > 0 &&
                BFieldTowersAxRad.cols() == 4);
        // extract axial part of the field
        towersBz.readGridFromMatrix(BFieldTowersAxRad,2);
        // extract radial part of the field
        towersBr.readGridFromMatrix(BFieldTowersAxRad,3);
    }

    // -- trap
    BFieldTrapAxRad.readFromFile("trapmapaxrad.txt",4);

    /////////////////////////////////////////////
    // read in particles matrix if desired     //
    /////////////////////////////////////////////
    if (readParticlesFromFile == 1) {
        particles.readFromFile("inputdata_particles.txt",6);
    }

    ////////////////////////////////////////////////////////
    // read in the files that contain the original pulses //
    ////////////////////////////////////////////////////////
    size_t stageId = 0;
    ifstream ifstr;
    ifstr.open("0");
    while (ifstr.good()) {
        math::Matrix<real_t> pulses(ifstr,2);
        origStagePulses.push_back(pulses);
        ifstr.close();
        ++stageId; // go to the next stage
        std::ostringstream filename;
        filename << stageId;
        ifstr.open(filename.str().c_str());
    }
    ifstr.close();
    ifstr.open("towers.txt");
    if (ifstr.good()) {
        origTowerPulses.readFromFile(ifstr,2);
    }
    ifstr.close();
}


///////////////////////////////////////////////
///**objective function for the optimization**//
///////////////////////////////////////////////
double ObjectiveFunctionTrap::eval( double const *x, size_t id, size_t seed )
{
    /////////////////////////////////////////////////////////////////
    //measure runtime                                            //
    /////////////////////////////////////////////////////////////////
    time_t runtime_start = time(0);

    ////////////////////////////////////////////////////////////
    // initialize (seed) the local random number generator     //
    ////////////////////////////////////////////////////////////
    boost::mt19937 rng;
    rng.seed(seed);

    /////////////////////////////////////////////////////////////////
    // get the dimension of the input array                        //
    /////////////////////////////////////////////////////////////////
    size_t DIM = (size_t)(x[-1]);
    Assert<ExWrongInput>(DIM>=2);

    ////////////////////////////////////////////////////////////////
    // check wheter all time intervals are positive               //
    ////////////////////////////////////////////////////////////////
    for (size_t i = 0; i<DIM; ++i) {
        if (x[i] < 0) {
            return std::numeric_limits<double>::infinity();
        }
    }

    //////////////////////////////////////////////////////////////////
    //create the generic simulation object                          //
    //////////////////////////////////////////////////////////////////
    ZeemanDecelBuilder* decelBuilder;
    decelBuilder = new ZeemanDecelBuilder(indata_decel);
    size_t nStages = decelBuilder->simulation->engine.getNumbStages();

    //////////////////////////////////////////////////////////////
    //read in b-field of the decelerator and tower coils         //
    //////////////////////////////////////////////////////////////
    decelBuilder->simulation->engine.setStageCoilB(stageBz,stageBr);
    if(nStages > 1) {
        Assert<ExWrongInput>(BFieldTowersAxRad.rows() > 0 &&
```

```
                BFieldTowersAxRad.cols() == 4);
        decelBuilder->simulation->engine.setTowerCoilB(towersBz,towersBr);
    }

    //////////////////////////////////////////////////
    // create particle population(s)                //
    //////////////////////////////////////////////////

    ParticlePopulation pop(zee);
    // set the atomic mass factor of the members of the population
    pop.setMassFactor(decelBuilder->getDefaultMassFactor());
    // set the flag that indicates that the particles are
    // low-field-seeking
    pop.setLFS(true);
    // add the population to the simulation
    size_t pop_id = decelBuilder->simulation->addPopulation(pop);

    if (readParticlesFromFile == 0) {
        // (the first argument sets the number of particles)
        // generate random particles in state F=1, M=0
        // TODO: provide a way to say how many particles from
        //        which population should be generated
        decelBuilder->simulation->generateRandomParticles(
                decelBuilder->getNumberOfParticlesPerPopulation(),
                pop_id,rng);
    } else {
        // set the particles from a previously read file
        decelBuilder->simulation->generateParticlesFromMatrix(
                particles, pop_id);
    }

    //////////////////////////////////////////////////////////////////////
    // adjust the pulse-lengths of the decelerator-coils               //
    //////////////////////////////////////////////////////////////////////

    // Note: the following matrix that contains the lastPulses is also
    // used to adjust the trap-pulses
    size_t nStageCoils =
            decelBuilder->simulation->engine.getNumbStageCoils();
    size_t nTowerCoils =
            decelBuilder->simulation->engine.getTowerCoils();
    // determine at which stage and coil to start adjusting the pulses
    size_t firstStage = 0;
    size_t firstCoil = 0;
    size_t nOrigStages = 0;
    if (adjustTowerPulses==1) {
        // TODO: make it flexible in this case too
        Assert<ExWrongInput>(DIM == nStages*nStageCoils
                        + (nStages-1)*nTowerCoils + 2);
    } else {
        size_t rem = (DIM-2)%nStageCoils;
        if (rem == 0) {
            firstStage = nStages - (DIM-2)/nStageCoils;
            firstCoil = 0;
            Assert<ExWrongInput>(origStagePulses.size() >= firstStage);
        } else {
            firstStage = nStages -
                    (size_t)ceil((real_t)(DIM-2)/nStageCoils);
            firstCoil = nStageCoils-rem-1;
            Assert<ExWrongInput>(origStagePulses.size() > firstStage);
            Assert<ExWrongInput>(origStagePulses[firstStage].rows()
                    >= firstCoil);
        }
        for (size_t i = 0; i < firstStage; ++i) {
            Assert<ExWrongInput>(origStagePulses[i].rows()
                    >= nStageCoils);
        }
        Assert<ExWrongInput>(origTowerPulses.rows()
                >= firstStage*nTowerCoils);
    }

    size_t nTowerPulses = (nStages-1)*nTowerCoils;
    math::Matrix<ZeemanDecel::real_t> towerPulses(nTowerPulses,2);
    size_t xInd = 0;
    size_t towerInd = 0;
    // set the begin−time of the first pulse
    // Note: this may be overwritten by the original pulses if the
    // dimension of the input vector is smaller than the actual number
    // of coils
    ZeemanDecel::real_t nextStagePulseStart = firstPulseStart;
    for(size_t i = 0; i < nStages; ++i) {
        math::Matrix<ZeemanDecel::real_t> stagePulses(nStageCoils,2);
        // set the stage pulses
        for(size_t j = 0; j < nStageCoils; ++j) {
            if (i > firstStage || (i == firstStage && j >= firstCoil)) {
                // set the begin−time of this pulse
                stagePulses(j,0) = nextStagePulseStart;
                // set the end−time of this pulse
                stagePulses(j,1) = stagePulses(j,0) + minDecelPulseLength + x[xI
nd];
                // set the begin−time of the next pulse if any
                ++xInd;
            } else {
                // set to original pulses
                stagePulses(j,0) = origStagePulses[i](j,0);
                stagePulses(j,1) = origStagePulses[i](j,1);
            }
            nextStagePulseStart = stagePulses(j,1)−decelPulseOverlap;
        }
        decelBuilder−>simulation−>engine.setStagePulses(i,stagePulses);
        // adjust the tower pulses
        if (i < nStages−1) {
            ZeemanDecel::real_t nextTowerPulseStart
                    = stagePulses(nStageCoils−1,1)−towerSwitchOnOverlap;
            for(size_t j = 0; j < nTowerCoils; ++j) {
                if (i >= firstStage) {
                    // set the begin−time of this pulse
                    towerPulses(towerInd,0) = nextTowerPulseStart;
                    // set the end−time of this pulse
                    if(adjustTowerPulses==1) {
                        towerPulses(towerInd,1) =
                                towerPulses(towerInd,0)+x[xInd];
                        // set the begin−time of the next pulse if any
                        nextTowerPulseStart = towerPulses(towerInd,1)
                                −towerSwitchOnOverlap;
                        ++xInd;
                    }
                    // else the end−times will be set afterwards (see
                    // below)
                } else {
                    // set to original pulses
                    towerPulses(towerInd,0)
                            = origTowerPulses(towerInd,0);
                    towerPulses(towerInd,1)
                            = origTowerPulses(towerInd,1);
                }
                ++towerInd;
            }
            if(adjustTowerPulses==1) {
                nextStagePulseStart = towerPulses(towerInd−1,1);
            }
        }
    }

    //////////////////////////////////////////////////////////////
    // set the new end−time of the decelerator−simulation
    // to the time when the second−last coil has been completley
    // turned off
    //////////////////////////////////////////////////////////////
    math::Matrix<ZeemanDecel::real_t> lastStagePulses =
            decelBuilder−>simulation−>engine.getStagePulses(nStages−1);
    real_t decelEndTime = lastStagePulses(nStageCoils−2,1)
            +decelBuilder−>simulation−>engine.getStageRampOff()
            +decelBuilder−>simulation−>engine.getIncouplingTime();
    decelBuilder−>simulation−>setEndTime(decelEndTime);

    //////////////////////////////////////////////////////////////
    // set the tower pulses
    //////////////////////////////////////////////////////////////
    if (adjustTowerPulses == 0) {
```

```
        // If the switch off times of the tower coils are not part of
        // the optimization, they have to be set to something which does
        // not influence the optimization. The best would be to leave
        // them on until the end of the simulation but this would
        // decrease the performance if there is more than one tower.
        real_t towerRampOff = decelBuilder−>simulation−>
                engine.getTowerRampOff();
        for (size_t i = firstStage; i < nStages−1; ++i) {
            for (size_t j = 0; j < nTowerCoils; ++j) {
                if (i < nStages−2) {
                    // Set the end−time of this tower−pulse to the time
                    // before the next tower is switched on so that this
                    // tower coil is switched off when the first coil of
                    // the next tower gets switched on.
                    towerPulses(i*nTowerCoils+j,1) =
                           towerPulses((i+1)*nTowerCoils,0)
                           −towerRampOff;
                } else {
                    // set the end−time of this tower−pulse to the time
                    // when the decelerator is switched off
                    towerPulses(i*nTowerCoils+j,1) = decelEndTime;
                }
            }
        }
    }
    // else the end−times of the tower pulses have been previously
    // defined (see above)

    if (nStages > 1) {
        decelBuilder−>simulation−>engine.setTowerPulses(towerPulses);
    }

    ////////////////////////////////////////
    //create and add some observers       //
    ////////////////////////////////////////
    std::ofstream outputDetector;
    if (outputTOF == 1) {
        std::ostringstream filenameDetector;
        filenameDetector << "fe_" << id << "_detection.txt";
        outputDetector.open(filenameDetector.str().c_str());
        decelBuilder−>addLaserObserver(outputDetector);
    }

    //std::ofstream outputTraj;
    //std::ostringstream filenameTraj;
    //filenameTraj << "fe_" << id << "_traj.txt";
    //outputTraj.open(filenameTraj.str().c_str());
    //decelBuilder−>addTrajectoriesObserver(outputTraj);

    //////////////////////////////////////////////////
    //create integrator                             //
    //////////////////////////////////////////////////
    //VerletIntegrator<ParticleSim<ZeemanDecel> >* integrator =
    //        new VerletIntegrator<ParticleSim<ZeemanDecel> >();
    LeapfrogIntegrator<ParticleSim<ZeemanDecel> >* integrator =
            new LeapfrogIntegrator<ParticleSim<ZeemanDecel> >();

    //////////////////////////////////////////////////
    // store the initial number of particles        //
    //////////////////////////////////////////////////
    size_t initialNParticles = decelBuilder−>simulation−>getNParticles();

    ////////////////////////////////////////////////////////////
    //simulate the Zeeman−Decelerator                         //
    ////////////////////////////////////////////////////////////
    decelBuilder−>simulation−>runSimulation(*integrator);

    ///////////////////////////////////////////////////
    // create a simulation−object based on the         //
    // decelerator−simulation but with the trap−engine //
    ///////////////////////////////////////////////////
    MagneticTrapBuilder* trapBuilder;
    try {
        trapBuilder = new MagneticTrapBuilder(*decelBuilder−>simulation,
                indata_trap);
    } catch(ExWrongInput) {
        return -1;
    }
    trapBuilder->setDecelBFieldFromMatrix(BFieldDecelAxRad);
    trapBuilder->setTrapBFieldFromMatrix(BFieldTrapAxRad);
    ////////////////////////////////////////////////////////
    // set pulsing according to input vector             //
    ////////////////////////////////////////////////////////
    math::Matrix<MagneticTrap::real_t> trapPulses1(2,3);
    math::Matrix<MagneticTrap::real_t> trapPulses2(1,3);
    // 1st pulse of 1st coil
    trapPulses1(0,0) = lastStagePulses(nStageCoils-1,1)
            -decelToTrapPulseOverlap;
    trapPulses1(0,1) = trapPulses1(0,0)+x[DIM-2];
    trapPulses1(0,2) = frontTrapCoilFirstPulseScale;
    // 2nd pulse of 1st coil
    trapPulses1(1,0) = trapPulses1(0,1)
            +trapBuilder->simulation->engine.coil1.getRampOff()
            +x[DIM-1];
    // deterimine the end-time of the trap-simulation
    MagneticTrap::real_t endTimeTrap = trapPulses1(1,0)
            +trapBuilder->simulation->engine.coil1.getRampOn()
            +trapBuilder->simulation->engine.coil1.getIncouplingTime();
    trapPulses1(1,1) = endTimeTrap+1.0;
    trapPulses1(1,2) = frontTrapCoilSecondPulseScale;
    // pulse of 2nd coil
    trapPulses2(0,0) = trapPulses1(0,1)-trapPulseOverlap;
    trapPulses2(0,1) = endTimeTrap+1.0;
    trapPulses2(0,2) = rearTrapCoilPulseScale;

    trapBuilder->simulation->engine.coil1.setPulses(trapPulses1);
    trapBuilder->simulation->engine.coil2.setPulses(trapPulses2);

    ////////////////////////////////////////////////////////////////
    // set the new end-time of the trap-simulation                //
    // to the time when the first coil has been completley turned //
    // turned on for the second time                              //
    ////////////////////////////////////////////////////////////////
    trapBuilder->simulation->setEndTime(endTimeTrap);

    ////////////////////////////////////////////////////////////////
    // add the trap region observer                               //
    ////////////////////////////////////////////////////////////////
    std::ostream* outputStreamTrapRegion;
    if (outputTrapRegion == 1) {
        std::ostringstream filenameTrapRegion;
        filenameTrapRegion << "fe_" << id << "_trapregion.txt";
        outputStreamTrapRegion = new std::ofstream(
                filenameTrapRegion.str().c_str());
    } else {
        outputStreamTrapRegion = new nullstream();
    }
    RegionObserver<ParticleSim<MagneticTrap> >* trapRegionObserver
            = trapBuilder->addTrapRegionObserver(*outputStreamTrapRegion);
    if (outputTrapRegion == 0) {
        trapRegionObserver->setInverseFrequency(trapBuilder->simulation->getEndT
ime()
                -trapBuilder->simulation->getTime());
    }
    ParticleSim<ZeemanDecel>::real_t t
            = trapBuilder->simulation->getTime();
    ParticleSim<ZeemanDecel>::real_t dt
            = trapBuilder->simulation->getTimestep();
    trapBuilder->simulation->engine.prepareTimestepping(t,dt);
    double initialEnergyRatio =
            trapRegionObserver->calculateMeanEnergyRatioForAll();

    //////////////////////////////////////////////////
    //create integrator                             //
    //////////////////////////////////////////////////
    //VerletIntegrator<ParticleSim<MagneticTrap> >* integrator =
    //          new VerletIntegrator<ParticleSim<MagneticTrap> >();
    LeapfrogIntegrator<ParticleSim<MagneticTrap> >* integrator2 =
            new LeapfrogIntegrator<ParticleSim<MagneticTrap> >();
```

```
    //////////////////////////////////////////////////
    //start the trap-simulation                     //
    //////////////////////////////////////////////////
    trapBuilder->simulation->runSimulation(*integrator2);

    //////////////////////////////////////////////////
    // compute objective function values            //
    //////////////////////////////////////////////////

    trapRegionObserver->notify(
            trapBuilder->simulation->getTime(),
            trapBuilder->simulation->getTimestep());
    double objective;
    if(trapRegionObserver->getNParticles() > 0) {
        // Note: this value is always smaller than gamma+1 and greater
        // than zero
        objective = gamma+trapRegionObserver->getMeanEnergyRatio()
                - gamma*((double)trapRegionObserver->
                    getNParticles()
                    /initialNParticles);
    } else if (trapRegionObserver->
            calculateNumberOfParticlesWithoutEnergyBound()>0)
    {
        objective = gamma+delta+trapRegionObserver->
                calculateMinEnergyRatioWithoutEnergyBound()
                - delta*((double)trapRegionObserver->
                    calculateNumberOfParticlesWithoutEnergyBound()
                    /initialNParticles);
    } else if (trapBuilder->simulation->getNParticles() > 0)
    {
        objective = gamma+delta+initialEnergyRatio+trapRegionObserver->
                calculateMeanZDistanceRatioForAll();
    } else
    {
        objective = std::numeric_limits<double>::infinity();
    }
    std::ostringstream filenameOutput;
    filenameOutput << "fe_" << id << "_output.txt";
    std::ofstream ofstr(filenameOutput.str().c_str());
    ofstr   << id << " "
            << seed << " "
            << objective << " "
            << trapRegionObserver->getTimeOfLastMeasurement() << " "
            << trapRegionObserver->getNParticles() << " "
            << trapRegionObserver->getMeanEnergyRatio();
    // output the input vector x
    for (size_t i = 0; i < DIM; ++i) {
        ofstr << " " << x[i];
    }
    for (size_t i = 0; i < nStages; ++i) {
        math::Matrix<ZeemanDecel::real_t> stagePulses =
                decelBuilder->simulation->engine.getStagePulses(i);
        for (size_t j = 0; j < nStageCoils; ++j) {
            ofstr << " " << stagePulses(j,0);
            ofstr << " " << stagePulses(j,1);
        }
    }
    ofstr << std::endl;
    ofstr.close();


    ////////////////////////////////////////////////////
    // output information about the trapped particles //
    ////////////////////////////////////////////////////
    if (outputTrappedParts == 1) {
        std::ostringstream filenameTrappedParts;
        filenameTrappedParts << "fe_" << id << "_output_trapped_parts.txt";
        std::ofstream ofstr_trapped (filenameTrappedParts.str().c_str());
        trapRegionObserver->outputTrappedParticles(ofstr_trapped);
        ofstr_trapped.close();
    }

    ////////////////////////////
    // delete the integrators //
    ////////////////////////////
    delete integrator;
    delete integrator2;

    /////////////////////////////
    // delete the builders     //
    /////////////////////////////
    delete trapBuilder;
    delete decelBuilder;

    outputDetector.close();
    //outputTraj.close();
    delete outputStreamTrapRegion;

    ///////////////////////////////////////////
    //output the runtime measurement         //
    ///////////////////////////////////////////
    clock_t cputime = clock();
    time_t runtime_end = time(0);
    std::ostringstream filenameRuntime;
    filenameRuntime << "fe_" << id << "_runtime.txt";
    std::ofstream ofstrRuntime (filenameRuntime.str().c_str());
    ofstrRuntime << "Start Date: " << asctime(localtime(&runtime_start));
    ofstrRuntime << "End Date: " << asctime(localtime(&runtime_end));
    ofstrRuntime << "Elapsed real time: "
            << (uintmax_t)(runtime_end-runtime_start) << " sec\n";
    ofstrRuntime << "Elapsed cpu time: " << cputime/CLOCKS_PER_SEC
            << " sec\n";
    ofstrRuntime.close();

    return objective;
}
```

```cpp
// objective_hfs.cxx: v2.4, 2009-05-08, Yves Salathe
#include "headers/objective_hfs.h"
#include "headers/zeemandecel_builder.h"
#include "headers/symplectic_euler_integrator.h"
//#include "headers/verlet_integrator.h"
#include "headers/matrix.h"
#include "headers/exceptions.h"
#include "headers/assert.h"
#include <time.h>
#include <cstdlib>
#include <sstream>
#include <string>
#include <limits>

using namespace std;
using namespace simulation;

//boost::mt19937 ObjectiveFunctionHFS::rng;
//ObjectiveFunctionHFS::size_t ObjectiveFunctionHFS::nfe = 0;
math::Matrix<string> ObjectiveFunctionHFS::indata_decel;
math::Matrix<ObjectiveFunctionHFS::real_t> ObjectiveFunctionHFS::zee;
math::Grid2d<ObjectiveFunctionHFS::real_t>
    ObjectiveFunctionHFS::stageBz;
math::Grid2d<ObjectiveFunctionHFS::real_t>
    ObjectiveFunctionHFS::stageBr;
ObjectiveFunctionHFS::real_t ObjectiveFunctionHFS::gamma;
ObjectiveFunctionHFS::real_t ObjectiveFunctionHFS::delta;
ObjectiveFunctionHFS::real_t ObjectiveFunctionHFS::epsilon;
ObjectiveFunctionHFS::real_t ObjectiveFunctionHFS::lowerInitialVelZ;
ObjectiveFunctionHFS::real_t ObjectiveFunctionHFS::lowerPosZ;
ObjectiveFunctionHFS::real_t ObjectiveFunctionHFS::upperVelZ;
ObjectiveFunctionHFS::real_t ObjectiveFunctionHFS::desiredVel;
ObjectiveFunctionHFS::size_t ObjectiveFunctionHFS::nClosest;


/////////////////////////////////////////////////////////////
///**initialize objective function before the optimization**//
/////////////////////////////////////////////////////////////
void ObjectiveFunctionHFS::initialize()
{
    /////////////////////////////////////////////////////////////////
    // read parameter file for objective function evaluation        //
    /////////////////////////////////////////////////////////////////
    const char inputFileName[] = "inputdata_obj.txt";
    math::Matrix<string> param(inputFileName,2);
    // TODO: provide a more flexible input format
    if (param.rows() != 9)
    {
        std::cerr << "file: " << inputFileName << " has a wrong number "
                  << "of lines\n\n";
        throw ExWrongInput();
    }
    string zeeFileName = param(0,1);
    gamma = atof(param(1,1).c_str());
    delta = atof(param(2,1).c_str());
    epsilon = atof(param(3,1).c_str());
    lowerInitialVelZ = atof(param(4,1).c_str());
    lowerPosZ = atof(param(5,1).c_str());
    upperVelZ = atof(param(6,1).c_str());
    desiredVel = atof(param(7,1).c_str());
    nClosest = atoi(param(8,1).c_str());

    ///////////////////////////////////////////////////////
    // read the input data of the decelerator              //
    ///////////////////////////////////////////////////////
    indata_decel.readFromFile("inputdata_long.txt",2);

    ////////////////////////////////////////
    // read in Zeeman-effect                //
    ////////////////////////////////////////
    zee.readFromFile(zeeFileName.c_str(),2);

    ////////////////////////////////////////
    // read in the magnetic fields          //
    ////////////////////////////////////////
```

```cpp
    math::Matrix<ZeemanDecel::real_t> bfield("decelmapaxrho.txt",4);
    // extract axial part of the field
    stageBz.readGridFromMatrix(bfield,2);
    // extract radial part of the field
    stageBr.readGridFromMatrix(bfield,3);
}

////////////////////////////////////////////////
///**objective function for the optimization**//
////////////////////////////////////////////////
double ObjectiveFunctionHFS::eval( double const *x, size_t id, size_t seed )
{
    ///////////////////////////////////////////////////////////////
    //measure runtime                                            //
    ///////////////////////////////////////////////////////////////
    time_t runtime_start = time(0);

    //////////////////////////////////////////////////////////
    // initialize (seed) the local random number generator   //
    //////////////////////////////////////////////////////////
    boost::mt19937 rng;
    rng.seed(seed);

    ///////////////////////////////////////////////////////////////
    // get the dimension of the input array                       //
    ///////////////////////////////////////////////////////////////
    size_t DIM = (size_t)(x[-1]);
    Assert<ExWrongInput>(DIM>=2);

    ////////////////////////////////////////////////////////////////
    //create the generic simulation object                         //
    ////////////////////////////////////////////////////////////////
    ZeemanDecelBuilder* decelBuilder;
    try {
        decelBuilder = new ZeemanDecelBuilder(indata_decel);
    } catch(ExWrongInput) {
        return -1;
    }

    /////////////////////////////////////////////////////////////
    //read in b-field of the decelerator and tower coils         //
    /////////////////////////////////////////////////////////////
    decelBuilder->simulation->engine.setStageCoilB(stageBz,stageBr);

    ///////////////////////////////////////////
    // create particle population(s)           //
    ///////////////////////////////////////////
    ParticlePopulation pop(zee);
    // set the atomic mass factor of the members of the population
    pop.setMassFactor(decelBuilder->getDefaultMassFactor());
    size_t pop_id = decelBuilder->simulation->addPopulation(pop);

    // (the first argument sets the number of particles)
    // generate random particles in state F=1, M=0
    // TODO: provide a way to say how many particles from
    //   which population should be generated
    decelBuilder->simulation->generateRandomParticles(
            decelBuilder->getNumberOfParticlesPerPopulation(),
            pop_id,rng,lowerInitialVelZ);

    /////////////////////////////////////////////////////////////////
    // adjust the pulse-lengths of the last decelerator-tage          //
    // TODO: generalize this to more than just the last stage         //
    /////////////////////////////////////////////////////////////////
    size_t nStages = decelBuilder->simulation->engine.getNumbStages();
    size_t nStageCoils =
            decelBuilder->simulation->engine.getNumbStageCoils();
    Assert<ExWrongInput>(DIM == 2*nStageCoils);
    math::Matrix<ZeemanDecel::real_t> lastStagePulses(nStageCoils,2);
    ZeemanDecel::real_t maxEndTime = 0;
    for(int i = 0; i < nStageCoils; ++i) {
        // set the end-time of this pulse
        lastStagePulses(i,0) = x[i]*x[i];
        lastStagePulses(i,1) = lastStagePulses(i,0)
                +decelBuilder->simulation->engine.getStageRampOn()
                +x[i+nStageCoils]*x[i+nStageCoils];
        if(lastStagePulses(i,1) > maxEndTime) {
            maxEndTime = lastStagePulses(i,1);
        }
    }
    decelBuilder->simulation->engine.setStagePulses(nStages-1,
                lastStagePulses);

    ///////////////////////////////////////////////////////////////
    // set the new end-time of the decelerator-simulation
    // to the time when the last coil has been completley
    // turned off
    ///////////////////////////////////////////////////////////////
    decelBuilder->simulation->setEndTime(
                maxEndTime
                +decelBuilder->simulation->engine.getStageRampOff()
                +decelBuilder->simulation->engine.getIncouplingTime()
            );

    ////////////////////////////////////////
    //create and add some observers       //
    ////////////////////////////////////////
//    std::ostringstream filenameDetector;
//    filenameDetector << "fe_" << id << "_detection.txt";
//    std::ofstream outputDetector(filenameDetector.str().c_str());
//    decelBuilder->addLaserObserver(outputDetector);
//    if (outputTrajectories) {
//        decelBuilder->addTrajectoriesObserver();
//    }

    //////////////////////////////////////////////////
    //create integrator                             //
    //////////////////////////////////////////////////
    //VerletIntegrator<ParticleSim<ZeemanDecel> >* integrator =
    //        new VerletIntegrator<ParticleSim<ZeemanDecel> >();
    SymplecticEulerIntegrator<ParticleSim<ZeemanDecel> >* integrator =
            new SymplecticEulerIntegrator<ParticleSim<ZeemanDecel> >();

    //////////////////////////////////////////////////
    // store the initial number of particles        //
    //////////////////////////////////////////////////
    size_t initialNParticles = decelBuilder->simulation->getNParticles();

    ////////////////////////////////////////////////////////////
    //simulate the Zeeman-Decelerator                         //
    ////////////////////////////////////////////////////////////
    decelBuilder->simulation->runSimulation(*integrator);

    //////////////////////////////////////////////////
    // compute objective function values            //
    //////////////////////////////////////////////////
    const ParticleSim<ZeemanDecel>::particle_list_t& particles
            = decelBuilder->simulation->getParticles();
    Particle::vector_t meanPos;
    Particle::vector_t meanPos2;
    Particle::vector_t meanVel;
    Particle::vector_t meanVel2;
    Particle::vector_t stdPos;
    Particle::vector_t stdVel;
    double objective = std::numeric_limits<double>::infinity();
    double meanDist = std::numeric_limits<double>::infinity();
    if(particles.size() > 0) {
        ParticleSim<ZeemanDecel>::particle_list_t closestParticles;
        // determine the nClosest closest particles to the desired velocity
        for(ParticleSim<ZeemanDecel>::particle_list_t::const_iterator p =
                particles.begin(); p != particles.end(); ++p)
        {
            if(p->pos.getZ() > lowerPosZ && p->vel.getZ() < upperVelZ) {
                // insert the particle into the list of closest particles and
                // keep the list sorted by the distance to the desired velocity
                ParticleSim<ZeemanDecel>::particle_list_t::iterator pc =
                        closestParticles.begin();
                while (pc != closestParticles.end())
                {
                    if(fabs(p->vel.norm2()-desiredVel)
```

```
                            < fabs(pc->vel.norm2()-desiredVel)) {
                        break;
                    }
                    ++pc;
                }
                pc = closestParticles.insert(pc,*p);
                // drop the farthest particle if necessary
                if (closestParticles.size() > nClosest) {
                    closestParticles.pop_back();
                }
            }
        }
        std::ostringstream filenameBunch;
        filenameBunch << "fe_" << id << "_bunch.txt";
        std::ofstream ofstr_bunch(filenameBunch.str().c_str());
        if (closestParticles.size() > 0) {
            meanDist = 0;
        }
        for(ParticleSim<ZeemanDecel>::particle_list_t::const_iterator p =
                closestParticles.begin(); p != closestParticles.end();
                ++p) {
            meanPos += p->pos;
            meanPos2 += p->pos.elemMult(p->pos);
            meanVel += p->vel;
            meanVel2 += p->vel.elemMult(p->vel);
            meanDist += fabs(p->vel.norm2()-desiredVel);
            p->outputInformation(ofstr_bunch);
            ofstr_bunch << std::endl;
        }
        ofstr_bunch.close();
        meanPos = meanPos/closestParticles.size();
        meanPos2 = meanPos2/closestParticles.size();
        meanVel = meanVel/closestParticles.size();
        meanVel2 = meanVel2/closestParticles.size();
        meanDist = meanDist/closestParticles.size();
        stdPos = meanPos.elemMult(meanPos)
                - meanPos2;
        stdVel = meanVel.elemMult(meanVel)
                - meanVel2;
        if (closestParticles.size() == nClosest) {
            objective = gamma*meanDist + delta*stdPos.norm2()
                    + epsilon*stdVel.norm2();
        } else if (closestParticles.size() > 0) {
            double penalty = (nClosest-closestParticles.size())
                    + gamma*(upperVelZ-desiredVel);
            objective = gamma*meanDist + delta*stdPos.norm2()
                    + epsilon*stdVel.norm2() + penalty;
        }
    }
    std::ostringstream filenameOutput;
    filenameOutput << "fe_" << id << "_output.txt";
    std::ofstream ofstr(filenameOutput.str().c_str());
    ofstr   << id << " "
            << seed << " "
            << objective << " "
            << decelBuilder->simulation->getTime() << " "
            << meanDist << " "
            << stdPos.norm2() << " "
            << stdVel.norm2() << " "
            << meanPos.getX() << " "
            << meanPos.getY() << " "
            << meanPos.getZ() << " "
            << meanVel.getX() << " "
            << meanVel.getY() << " "
            << meanVel.getZ() << " ";
    // output the input vector x
    for (size_t i = 0; i < DIM; ++i) {
        ofstr << " " << x[i];
    }
    ofstr << std::endl;
    ofstr.close();

    ////////////////////////////
    // delete the integrators //
    ////////////////////////////
```

```
        delete integrator;

        //////////////////////////////
        // delete the builders     //
        //////////////////////////////
        delete decelBuilder;

    //    outputDetector.close();

        ////////////////////////////////////////////
        //output the runtime measurement          //
        ////////////////////////////////////////////
        clock_t cputime = clock();
        time_t runtime_end = time(0);
        std::ostringstream filenameRuntime;
        filenameRuntime << "fe_" << id << "_runtime.txt";
        std::ofstream ofstrRuntime (filenameRuntime.str().c_str());
        ofstrRuntime << "Start Date: " << asctime(localtime(&runtime_start));
        ofstrRuntime << "End Date: " << asctime(localtime(&runtime_end));
        ofstrRuntime << "Elapsed real time: "
                << (uintmax_t)(runtime_end-runtime_start) << " sec\n";
        ofstrRuntime << "Elapsed cpu time: " << cputime/CLOCKS_PER_SEC
                << " sec\n";
        ofstrRuntime.close();

        return objective;
    }
```

77

```
// assert.h: 2009−03−16, Yves Salathe
#ifndef __ASSERT_H__
#define __ASSERT_H__

/// checks assertion A and throws an instance of exception X if A is not
/// fulfilled
template<class X, class A> inline void Assert(A assertion)
{
    if (!assertion) throw X();
}


/// checks assertion A and throws the instance except if A is not fulfilled
/// Note: this function can be usesd to throw exceptions with parameters
template<class E, class A> inline void Assert(A assertion, E except)
{
    if (!assertion) throw except;
}
#endif
```

```
// basic_math.h: Yves Salathe, 2009-02-23
#ifndef __BASIC_MATH_H__
#define __BASIC_MATH_H__

namespace math
{

/// return the minimum of x and y
template <class T>
inline T min(T x,T y)
{
    return x < y ? x : y;
}

/// return the maximum of x and y
template <class T>
inline T max(T x,T y)
{
    return x < y ? y : x;
}

}
#endif
```

```
// coil.h: 2009-04-09, Yves Salathe
#ifndef __COIL_H__
#define __COIL_H__

#include "grid.h"
#include "matrix.h"
#include "basic_math.h"
#include "polynomial.h"
#include "assert.h"
#include <valarray>
#include <vector>
#include <limits>

namespace simulation
{

///////////////////////////////
///
/// \class Coil
///
/// \brief this class represents coils in the trap simulation
///
///////////////////////////////
class Coil
{
public:
    ////////////////////////////////////////////////////////
    // public types and datastructures                    //
    ////////////////////////////////////////////////////////
    typedef double real_t;
    typedef unsigned int size_t;

    ///////////////////////////////
    ///
    /// \class Pulse
    ///
    /// \brief simple data structure that represents the pulsing
    /// of the trap coils
    ///
    ///////////////////////////////
    class Pulse
    {
    public:
        ///default constructor
        Pulse() { }

        ///constructor
        Pulse(real_t _startTime, real_t _endTime,
              const std::valarray<real_t>& _coeff)
                : startTime(_startTime), endTime(_endTime),
                coeff(_coeff) { }

        /// time at which the pulse ends
        real_t startTime;
        /// time at which the pulse ends
        real_t endTime;
        /// coefficients of a polynomial that describes
        /// the decay of the magnetic field during this
        /// pulse
        std::valarray<real_t> coeff;

        /// add a constant offset to the times of the pulses
        void addIncouplingTime(real_t dt)
        {
            startTime += dt;
            endTime += dt;
        }
    };

    ////////////////////////////////////////////////////////
    // constructors                                       //
    ////////////////////////////////////////////////////////

    /// default constructor (creates an inactive coil)
    Coil()
```

```
    {
        currentPulse = pulses.end();
    }

    ////////////////////////////////////////////////////////
    // setters                                            //
    ////////////////////////////////////////////////////////

    /// sets the values of the B-field from the trap coils
    ///
    /// \param bz the z-component of the B-field
    /// \param br the r-component of the B-field
    void setBField(const math::Grid2d<real_t> bz,
                   const math::Grid2d<real_t> br)
    {
        origBz = bz;
        origBr = br;
        scaleBField();
    }

    /// set the current at which the given field has been calculated
    void setSimCurrent(real_t I)
    {
        Assert<ExWrongInput>(fabs(I)
                             > std::numeric_limits<real_t>::epsilon());
        simCurrent = I;
        scaleBField();
    }

    /// scale the B field of the trap coils according to the current
    /// this corresponds then to the current where the scaling
    /// polynomial and the ramp factor are both 1.0
    void setUnscaledCurrent(real_t I)
    {
        unscaledCurrent = I;
        scaleBField();
    }

    /// sets the pulsing of the first trap coil from a matrix
    ///
    /// the structure of the matrix has to be as follows:
    /// 1. column: start time
    /// 2. column: end time
    /// from the 3. column:
    ///         the coefficients of the polynomial that describes the
    ///         decay of the magnetic field due to unloading of the
    ///         capacitors
    void setPulses(const math::Matrix<real_t>& p)
    {
        Assert<ExWrongInput>(p.cols() > 2);
        for (size_t i = 0; i < p.rows(); ++i)
        {
            std::valarray<real_t> c(p.cols()-2);
            for (size_t j = 0; j < p.cols()-2; ++j)
            {
                c[j] = p(i,j+2);
            }
            origPulses.push_back(Pulse(p(i,0),p(i,1), c));
        }
        shiftPulses();
    }

    /// set the actual pulse (inclusive incoupling time, as opposed to the
    /// original pulse)
    void setActualPulse(size_t i, const Pulse& p)
    {
        origPulses.at(i) = p;
        origPulses.at(i).addIncouplingTime(-incouplingTime);
        shiftPulses();
    }

    /// adds a pulse p to the vector of pulses of this coil
    inline void addPulse(const Pulse& p)
    {
        origPulses.push_back(p);
```

```cpp
        shiftPulses();
    }

    /// adds a pulse p to the vector of pulses of this coil
    inline void addPulseWithoutIncouplingTime(const Pulse& p)
    {
        origPulses.push_back(p);
        origPulses.back().addIncouplingTime(-incouplingTime);
        shiftPulses();
    }

    /// set the current pulse to the one at time t and the
    /// next pulse to the next pulse with startTime greater
    /// than t
    ///
    /// dt is the length of a timestep
    /// this routine assumes that the pulses for this coil are
    /// sorted and excluding each other
    void setCurrentAndNextPulse(real_t t, real_t dt)
    {
        for (currentPulse = pulses.begin();
                currentPulse != pulses.end();
                ++currentPulse)
        {
            if (currentPulse->startTime < t &&
                    t < currentPulse->endTime+rampOff-dt)
            {
                initialScalingFactor = math::evalPoly(
                                           currentPulse->startTime
                                           +rampOn,
                                           currentPulse->coeff);
                endScalingFactor = math::evalPoly(
                                       currentPulse->endTime,
                                       currentPulse->coeff);
                break;
            }
        }
        for (nextPulse = pulses.begin();
                nextPulse != pulses.end();
                ++nextPulse)
        {
            if (nextPulse->startTime > t)
            {
                break;
            }
        }
    }

    /// set the switch on ramp time
    inline void setRampOn(real_t dt)
    {
        rampOn = dt;
    }

    /// set the switch off ramp time
    inline void setRampOff(real_t dt)
    {
        rampOff = dt;
    }

    /// set the position of the coil along the z-axis
    inline void setPosZ(real_t p)
    {
        posZ = p;
    }

    /// set the offset which is added to the timing of the pulses
    inline void setIncouplingTime(real_t dt)
    {
        incouplingTime = dt;
        shiftPulses();
    }

    ////////////////////////////////////////////////////////
    // getters                                            //
```

```cpp
    ////////////////////////////////////////////////////////

    /// get the starting time of the next pulse of this coil
    ///
    /// returns inf if no event occurs in the future
    inline real_t getStartOfNextPulse() const
    {
        real_t t = std::numeric_limits<real_t>::infinity();
        if (hasNextPulse())
        {
            t = nextPulse->startTime;
        }
        return t;
    }

    /// get the time when the coil becomes inactive
    ///
    /// returns inf if not active
    inline real_t getEndOfActive() const
    {
        real_t t = std::numeric_limits<real_t>::infinity();
        if (isActive())
        {
            t = currentPulse->endTime+rampOff;
        }
        return t;
    }

    /// add the value and gradient of both components of the B field
    /// at the point (z,r) from this coil to the given references
    /// bz and br
    inline void addBField(real_t z, real_t r,
                          math::ValGrad<real_t>& bz,
                          math::ValGrad<real_t>& br) const
    {
        real_t relPosZ = z-posZ;
        Bz.linpValueAndGradient(relPosZ,r,scalingFactor,bz);
        Br.linpValueAndGradient(relPosZ,r,scalingFactor,br);
    }

    /// retruns true if the coil is active (e.g. is pulsed)
    inline bool isActive() const
    {
        return (currentPulse != pulses.end());
    }

    /// retruns true if the next pulse is set
    inline bool hasNextPulse() const
    {
        return (nextPulse != pulses.end());
    }

    /// calculate the ramp factor at time t
    inline real_t calculateRampFactor(real_t t) const
    {
        return math::min(1.0,
                         (t-currentPulse->startTime)
                         /rampOn)
               *math::min(1.0,
                          (currentPulse->endTime+rampOff-t)
                          /rampOff
                         );
    }

    /// get the switch on ramp time
    inline real_t getRampOn() const
    {
        return rampOn;
    }

    /// get the switch off ramp time
    inline real_t getRampOff() const
    {
        return rampOff;
    }
```

```
        /// get the position of the coil along the z-axis
        inline real_t getPosZ() const
        {
            return posZ;
        }

        /// get the offset which is added to the timing of the pulses
        inline real_t getIncouplingTime() const
        {
            return incouplingTime;
        }

        /// get the pulsing of the first trap coil as a matrix
        ///
        /// the structure of the matrix has to be as follows:
        /// 1. column: start time
        /// 2. column: end time
        /// from the 3. column:
        ///         the coefficients of the polynomial that describes the
        ///         decay of the magnetic field due to unloading of the
        ///         capacitors
        math::Matrix<real_t> getOrigPulsesMatrix() const
        {
            size_t maxDeg = 0;
            for (std::vector<Pulse>::const_iterator i = origPulses.begin();
                    i != origPulses.end(); ++i) {
                if (i->coeff.size() > maxDeg) maxDeg = i->coeff.size();
            }
            math::Matrix<real_t> p(origPulses.size(),maxDeg+2);
            for (size_t i = 0; i < p.rows(); ++i)
            {
                p(i,0) = origPulses[i].startTime;
                p(i,1) = origPulses[i].endTime;
                for (size_t j = 2; j < p.cols(); ++j)
                {
                    if (j-2 < origPulses[i].coeff.size()) {
                        p(i,j) = origPulses[i].coeff[j-2];
                    }
                }
            }
            return p;
        }

        /// get a constant reference to the original pulses of the coil
        /// (pulses without incoupling time)
        inline const std::vector<Pulse>& getOrigPulses() const
        {
            return origPulses;
        }

        /// get a constant reference to the adjusted pulses of the coil
        /// (pulses with incoupling time)
        inline const std::vector<Pulse>& getPulses() const
        {
            return pulses;
        }

        /// get a constant reference to the adjusted pulse i of the coil
        /// (pulses with incoupling time)
        inline const Pulse& getPulse(size_t i) const
        {
            return pulses.at(i);
        }

        /// get the current at which the given field has been calculated
        inline real_t getSimCurrent() const
        {
            return simCurrent;
        }

        /// get the current which scales the B field of the trap coils
        /// this corresponds then to the current where the scaling
        /// polynomial and the ramp factor are both 1.0
        inline real_t getUnscaledCurrent() const
```

```
        {
            return unscaledCurrent;
        }

        ////////////////////////////////////////////////////
        // other methods                                  //
        ////////////////////////////////////////////////////

        /// calculate the factor with which the field of the
        /// coil is scaled due to the unloading of the
        /// capacitors at time t and stores this for later usage
        inline void calculateScalingFactor(real_t t)
        {
            if (t < currentPulse->startTime+rampOn)
            {
                scalingFactor = calculateRampFactor(t)
                                *initialScalingFactor;
            }
            else if (t > currentPulse->endTime)
            {
                scalingFactor = calculateRampFactor(t)
                                *endScalingFactor;
            }
            else
            {
                scalingFactor = calculateRampFactor(t)
                                *math::evalPoly(t,
                                                currentPulse->coeff);
            }
        }

        ///////////////////////////////
        //exceptions
        ///////////////////////////////

        ///////////////////////////////
        /// \class ExWrongInput
        ///
        /// \brief Exception which is thrown if a wrong input
        /// parameter is set
        ///////////////////////////////
        class ExWrongInput { };

    private:
        /////////////////////////////////////////////////////
        // private member variables (properties of the class) //
        /////////////////////////////////////////////////////

        /// axial component of the B-field (unscaled)
        math::Grid2d<real_t> origBz;
        /// radial component of the B-field (unscaled)
        math::Grid2d<real_t> origBr;
        /// axial component of the B-field
        /// (scaled according to the current)
        math::Grid2d<real_t> Bz;
        /// radial component of the B-field
        /// (scaled according to the current)
        math::Grid2d<real_t> Br;
        /// the current at which the magnetic field has been
        /// simulated
        real_t simCurrent;
        /// the unscaled current
        real_t unscaledCurrent;
        /// original pulsing of the coil (without incoupling time)
        std::vector<Pulse> origPulses;
        /// pulsing of the coil with incoupling time
        std::vector<Pulse> pulses;
        /// iterator of the current pulse
        std::vector<Pulse>::iterator currentPulse;
        /// iterator of the next pulse
        std::vector<Pulse>::iterator nextPulse;
        /// the ramping time B-field of the decelerator coils in us
        real_t rampOn;
        /// the ramping time B-field of the decelerator coils in us
        real_t rampOff;
    /// the axial position of the coil
    real_t posZ;
    /// scaling factor by which the field is scaled
    real_t scalingFactor;
    /// the scaling factor during the switch on ramp
    real_t initialScalingFactor;
    /// the scaling factor during the switch off ramp
    real_t endScalingFactor;
    /// the time offset which is added to the pulsing of the coil)
    real_t incouplingTime;

    /// scale the B−field according to the values for
    /// unscaledCurrent and simCurrent
    void scaleBField()
    {
        Bz = origBz;
        Br = origBr;
        Bz.scaleValues(unscaledCurrent/simCurrent);
        Br.scaleValues(unscaledCurrent/simCurrent);
    }

    /// shift pulses according to the incoupling time that has been
    /// set
    void shiftPulses()
    {
        pulses = origPulses;
        for (std::vector<Pulse>::iterator i = pulses.begin();
                i != pulses.end(); ++i)
        {
            i−>addIncouplingTime(incouplingTime);
        }
    }

};

}
#endif
```

```
// decelpulsegen_observer.h: v2.4, 2009-05-06, Yves Salathe
#ifndef __DECELPULSEGEN_OBSERVER_H__
#define __DECELPULSEGEN_OBSERVER_H__

#include "observer.h"
#include "particle.h"
#include "matrix.h"
#include <cmath>

namespace simulation
{

///////////////////////////////
/// \class DecelPulseGenObserver
///
/// \brief this class describes an observer which can be
///        used to detect particles inside a certain volume
///////////////////////////////
template <class observable_t>
class DecelPulseGenObserver : public Observer<observable_t>
{
public:
    typedef typename Observer<observable_t>::real_t real_t;
    typedef typename Observer<observable_t>::size_t size_t;
    typedef typename observable_t::coord_t coord_t;
    typedef typename observable_t::particle_list_t particle_list_t;

    using Observer<observable_t>::observable;
    using Observer<observable_t>::activateTime;
    using Observer<observable_t>::deactivateTime;
    using Observer<observable_t>::inverseFrequency;
    using Observer<observable_t>::getOutputStream;

    /// default constructor
    DecelPulseGenObserver(observable_t& _observable,
            std::ostream& ostr = std::cout)
            : Observer<observable_t>(_observable, ostr),
            currentStage(0),
            currentCoil(0),
            stopSimulation(false)
    {
        setSwitchOnTimeFirstCoil();
        setStagePulseOverlap();
        setTowerPulseOverlap();
        setInverseFrequency(observable.getTimestep());
        setLeaveTowerCoilsOn();
    }

    /// normal copy constructor
    DecelPulseGenObserver(DecelPulseGenObserver<observable_t>& o)
            : Observer<observable_t>(o),
            currentStage(o.currentStage),
            currentCoil(o.currentCoil),
            stopSimulation(o.stopSimulation)
    {
        setSwitchOnTimeFirstCoil(o.switchOnTimeFirstCoil);
        setStagePulseOverlap(o.stagePulseOverlap);
        setTowerPulseOverlap(o.towerPulseOverlap);
    }

    /// copy constructor with the abilty to specify a new observable
    template<class observer_t>
    DecelPulseGenObserver(observer_t& o, observable_t& _observable)
            : Observer<observable_t>(o, _observable),
            currentStage(o.currentStage),
            currentCoil(o.currentCoil),
            stopSimulation(o.stopSimulation)
    {
        setSwitchOnTimeFirstCoil(o.switchOnTimeFirstCoil);
        setStagePulseOverlap(o.stagePulseOverlap);
        setTowerPulseOverlap(o.towerPulseOverlap);
    }

    /// virtual destructor
    /// this makes sure that the destructors of the derived classes
```

```
    /// are called when their objects get "delete"ed through a
    /// refernece of this type
    virtual ~DecelPulseGenObserver()
    {
    }

    void initializePulses()
    {
        ///////////////////////////////////////////////
        // initialize pulses for pulse-generation    //
        ///////////////////////////////////////////////
        real_t endTime = observable.getEndTime();
        size_t m = observable.engine.getNumbStages();
        size_t n = observable.engine.getNumbStageCoils();
        size_t n_tow = 0;
        if (m > 1) {
            n_tow = observable.engine.getTowerCoils();
        }
        for(size_t i = 0; i < m; ++i) {
            for(size_t j = 0; j < n+n_tow; ++j) {
                if (j < n) {
                    if (i == 0 && j == 0) {
                        observable.engine.setActualStagePulseStartTime
                                (i,j,switchOnTimeFirstCoil);
                    } else {
                        observable.engine.setActualStagePulseStartTime
                                (i,j,endTime);
                    }
                    observable.engine.setActualStagePulseEndTime
                            (i,j,endTime);
                } else if (i < m-1) {
                    observable.engine.setActualTowerPulseStartTime
                            (i,j-n,endTime);
                    observable.engine.setActualTowerPulseEndTime
                            (i,j-n,endTime);
                }
            }
        }
    }

    /// this virtual method is (hopefully) called by observables
    ///
    /// \param time the time of the notification
    /// \param timestep the "resolution" of the time
    virtual void notify(real_t time, real_t timestep)
    {
        if(currentStage < phasedeg.rows()
                    && currentCoil < phasedeg.cols()) {
            size_t n = observable.engine.getNumbStageCoils();
            size_t nextCoil = currentCoil;
            if(currentCoil < n) {
                // current pulse is a stage-coil-pulse
                real_t z = observable.getParticles().front().pos.getZ();
                real_t z_coil = observable.engine.getStageCoilPos(
                        currentStage,currentCoil);
                real_t d = observable.engine.getCoilDist();
                if(((z-z_coil)/d+.5)*180
                        > phasedeg(currentStage,currentCoil)) {
                    observable.engine.setActualStagePulseEndTime(
                            currentStage,currentCoil,time);
                    ++nextCoil;
                }
            } else if (currentStage
                    < observable.engine.getNumbStages()-1) {
                // current pulse is a tower-coil-pulse
                real_t z = observable.getParticles().front().pos.getZ();
                real_t z_coil = observable.engine.getTowerCoilPos(
                        currentStage,currentCoil-n);
                real_t d = observable.engine.getTowerDist();
                if(((z-z_coil)/d+.5)*180
                        > phasedeg(currentStage,currentCoil)) {
                    observable.engine.setActualTowerPulseEndTime(
                            currentStage,currentCoil-n,time);
                    ++nextCoil;
                }
            }
            if(nextCoil != currentCoil) {
                getOutputStream() << "end of pulse ("
                                  << currentStage
                                  << ","
                                  << currentCoil
                                  << ") at time "
                                  << time << std::endl;
                if(stopSimulation && currentStage == stopAfterStage
                        && currentCoil == stopAfterCoil) {
                    if (currentCoil < n) {
                        observable.setEndTime(time +
                                observable.engine.getStageRampOff());
                    } else {
                        observable.setEndTime(time +
                                observable.engine.getTowerRampOff());
                    }
                }
                if(nextCoil < phasedeg.cols()) {
                    // go to the next coil
                    currentCoil = nextCoil;
                } else if(observable.engine.getNumbStages() > 1) {
                    // go to the next stage
                    if (leaveTowerCoilsOn) {
                        size_t nTowerCoils =
                                observable.engine.getTowerCoils();
                        for(size_t i=0; i < nTowerCoils; ++i) {
                            observable.engine.
                                    setActualTowerPulseStartTime(
                                            currentStage,i,
                                            time-towerPulseOverlap);
                        }
                    }
                    ++currentStage;
                    currentCoil = 0;
                }
                if (currentStage < phasedeg.rows()) {
                    if (currentCoil < n) {
                        // next pulse is a stage-coil-pulse
                        observable.engine.setActualStagePulseStartTime(
                                currentStage,currentCoil,
                                time-stagePulseOverlap);
                    } else if(observable.engine.getNumbStages() > 1) {
                        // next pulse is a tower-pulse
                        observable.engine.setActualTowerPulseStartTime(
                                currentStage,currentCoil-n,
                                time-towerPulseOverlap);
                    }
                }
            }
        }
    }

    /// sets the phase angles in degrees according to a given matrix
    /// (m-by-n matrix) where each row corresponds to a certain stage
    /// and each column corresponds to a certain coil and the last columns
    /// correspond to the tower coils
    void setPhaseAngleDegreesFromMatrix
            (const math::Matrix<real_t>& phaseAngleDegrees)
    {
        Assert<ExWrongInput>(phaseAngleDegrees.rows()
                <= observable.engine.getNumbStages());
        if (observable.engine.getNumbStages() > 1 && !leaveTowerCoilsOn) {
            Assert<ExWrongInput>(phaseAngleDegrees.cols()
                    <= observable.engine.getNumbStageCoils()
                    + observable.engine.getTowerCoils());
        } else {
            Assert<ExWrongInput>(phaseAngleDegrees.cols()
                    <= observable.engine.getNumbStageCoils());
        }
        phasedeg = phaseAngleDegrees;
    }

    /// get the phase angles as a matrix
    const math::Matrix<real_t>& getPhaseAngleDegreesMatrix() const
```

```
    {
        return phasedeg;
    }

    /// set the time when to switch on the first coil in us
    void setSwitchOnTimeFirstCoil(real_t t = 0)
    {
        switchOnTimeFirstCoil = t;
    }

    /// set the overlap between two successive stage-coil-pulses in us
    /// this is also used when going from a tower pulse to a stage pulse
    void setStagePulseOverlap(real_t deltat = 3)
    {
        stagePulseOverlap = deltat;
    }

    /// set the overlap between two successive tower-coil-pulses in us
    /// this is also used when going from a stage pulse to a tower pulse
    void setTowerPulseOverlap(real_t deltat = 3)
    {
        towerPulseOverlap = deltat;
    }

    /// sets wheter to leave the tower coils switched on forever
    /// if this option is set, the tower coils will be left on forever
    /// the switch on time of the first coil of the next stage will be
    /// to the switch on time of the tower coil
    void setLeaveTowerCoilsOn(bool v = false)
    {
        leaveTowerCoilsOn = v;
    }

    /// stop the simulation after the pulse of coil j in stage i has
    /// been finished (including the ramp-time)
    ///
    /// this method can be used to do the trap-simulation afterwards
    void stopSimulationAfterPulse(size_t i, size_t j)
    {
        Assert<ExWrongInput>(i < observable.engine.getNumbStages());
        if(observable.engine.getNumbStages() > 1) {
            Assert<ExWrongInput>(j <
                    observable.engine.getNumbStageCoils()
                    + observable.engine.getTowerCoils());
        } else {
            Assert<ExWrongInput>(j <
                    observable.engine.getNumbStageCoils());
        }
        stopAfterStage = i;
        stopAfterCoil = j;
        stopSimulation = true;
    }

private:
    math::Matrix<real_t> phasedeg;
    real_t switchOnTimeFirstCoil;
    real_t stagePulseOverlap;
    real_t towerPulseOverlap;
    size_t currentStage;
    size_t currentCoil;
    bool stopSimulation;
    size_t stopAfterStage;
    size_t stopAfterCoil;
    bool leaveTowerCoilsOn;
};

}
#endif
```

```cpp
// engine.h: v2.4, 2009-05-06, Yves Salathe
#ifndef __ENGINE_H__
#define __ENGINE_H__

#include "valgrad.h"
#include "particle.h"
#include "particle_population.h"

namespace simulation
{

//////////////////////////////////////////////////////////////////
/// \class Engine
///
/// \brief      This class represents an interface for engines like
///             ZeemanDecel or MagneticTrap
//////////////////////////////////////////////////////////////////
class Engine
{
public:
    ///////////////////////////////
    // types and datastructures
    ///////////////////////////////
    typedef unsigned int size_t;
    typedef double real_t;

    ////////////////////////////////////////////////
    // virtual destructor
    ////////////////////////////////////////////////
    virtual ~Engine() { }

    ////////////////////////////////////////////////
    // common methods
    ////////////////////////////////////////////////

    /// calculate the value and the gradient of the absolute values
    /// of the B-field at the given axial and radial coordinates
    ///
    /// Note: this function has to be redefined by implementations of
    /// this interface. It is not declared as virtual for performance
    /// reasons. Instead static (compile-time) polymorphism can be
    /// obtained by templates (see e.g. ParticleSim).
    inline math::ValGrad<real_t> calculateAbsBField (real_t z,
            real_t r) const;

    /// prepare timestepping (this function is called by
    /// ParticleSim<MagneticTrap,integrator_t> at each
    /// event before the actual timestepping is done)
    ///
    /// Note: this function has to be redefined by implementations of
    /// this interface. It is not declared as virtual for performance
    /// reasons. Instead static (compile-time) polymorphism can be
    /// obtained by templates (see e.g. ParticleSim).
    inline void prepareTimestepping(real_t time, real_t timestep);

    /// prepare the calculation of the magnetic field
    /// (e.g. calculate the scaling factors)
    ///
    /// Note: this function has to be redefined by implementations of
    /// this interface. It is not declared as virtual for performance
    /// reasons. Instead static (compile-time) polymorphism can be
    /// obtained by templates (see e.g. ParticleSim).
    inline void setTime(real_t time, real_t timestep);

    /// calculate the number of timesteps until the next event occurs
    /// returns ceil((endtime-time)/timestep) if no event will happen
    ///
    /// Note: this function has to be redefined by implementations of
    /// this interface. It is not declared as virtual for performance
    /// reasons. Instead static (compile-time) polymorphism can be
    /// obtained by templates (see e.g. ParticleSim).
    inline size_t timestepsUntilNextEvent(real_t time, real_t timestep,
                                          real_t endtime) const;

    /// determines wheter the magnetic field will be zero everywhere at
```

```cpp
    /// the time defined by the method setTime
    ///
    /// Note: this function has to be redefined by implementations of
    /// this interface. It is not declared as virtual for performance
    /// reasons. Instead static (compile-time) polymorphism can be
    /// obtained by templates (see e.g. ParticleSim).
    inline bool isActive() const;

    /// check whether the position (z,r) is inside the boundaries
    ///
    /// Note: this function has to be redefined by implementations of
    /// this interface. It is not declared as virtual for performance
    /// reasons. Instead static (compile-time) polymorphism can be
    /// obtained by templates (see e.g. ParticleSim).
    inline bool isPointInsideBoundaries(real_t z, real_t r) const;

    /// check whether a low-field-seeking particle still can be
    /// accepted (e.g. decelerated)
    ///
    /// Note: this function has to be redefined by implementations of
    /// this interface. It is not declared as virtual for performance
    /// reasons. Instead static (compile-time) polymorphism can be
    /// obtained by templates (see e.g. ParticleSim).
    inline bool mayLFSParticleBeAccepted(real_t time,
            const Particle& p, const ParticlePopulation& pop) const;

protected:
    /// calculate the value and the gradient of the absolute values
    /// of the field from given values for the value and gradient of
    /// each component
    inline math::ValGrad<real_t> calculateAbsBFieldFromComponents(
        const math::ValGrad<real_t>& Bz,
        const math::ValGrad<real_t>& Br) const
    {
        real_t absB = sqrt(Bz.val*Bz.val+Br.val*Br.val);
        if (absB >= std::numeric_limits<real_t>::epsilon())
        {
            return math::ValGrad<real_t>(absB,
                                        (Bz.val*Bz.grad_dim1+Br.val*Br.grad_dim
1)/absB,
                                        (Bz.val*Bz.grad_dim2+Br.val*Br.grad_dim
2)/absB);
        }
        else
        {
            return math::ValGrad<real_t>(0,0,0);
        }
    }

};

}
#endif
```

```
// exceptions.h: v2.4, 2009−09−10, Yves Salathe
#ifndef __EXCEPTIONS__
#define __EXCEPTIONS__

namespace simulation {
    ///////////////////////////////
    //exceptions
    ///////////////////////////////

    ///////////////////////////////
    /// \class ExWrongInput
    ///
    /// \brief Exception which is thrown if a wrong input
    /// parameter is set
    ///////////////////////////////
    class ExWrongInput { };

    ///////////////////////////////
    /// \class ExWrongInput
    ///
    /// \brief Exception which is thrown if the size of an array is not
    /// what it should be
    ///////////////////////////////
    class ExSizeMismatch { };


    ///////////////////////////////
    /// \class ExFileOpenError
    ///
    /// \brief This exception is thrown when a file could not be opened
    ///////////////////////////////
    class ExFileOpenError { };
}

#endif
```

```cpp
// grid.h: 2009-02-18, Yves Salathe
#ifndef __GRID_H__
#define __GRID_H__

#include "matrix.h"
#include "valgrad.h"
#include <cmath>
#include <valarray>
#include <limits>

namespace math
{

////////////////////////////////////////////////////////
/// \class Grid2d
///
/// \brief this class describes a 2d scalar grid
////////////////////////////////////////////////////////
template <class value_t>
class Grid2d
{
public:
    typedef unsigned int size_t;
    typedef int int_t;

    /// default constructor creates an empty grid
    Grid2d()
    {
    }

    /// this constructor reads the field from a file
    ///
    /// structure of the file:
    /// column1: dimension 1 of coordinate
    /// column2: dimension 2 of coordinate
    /// column3: value of the field
    /// the coordinate values are assumed to be sorted primary after the
    /// first dimension and secondary after the second dimension and for
    /// each dimension they have to be equidistant
    Grid2d(const char* filename)
    {
        readGridFromFile(filename);
    }

    /// this constructor reads the field from a matrix
    ///
    /// \param A the matrix which specifies the grid
    /// structure of the matrix:
    /// column1: dimension 1 of coordinate
    /// column2: dimension 2 of coordinate
    /// column valCol: value of the field
    /// the coordinate values are assumed to be sorted primary after the
    /// first dimension and secondary after the second dimension and for
    /// each dimension they have to be equidistant
    ///
    /// \param valCol specifies the column which contains the values of
    /// the field (indices begin with 0)
    Grid2d(const Matrix<value_t>& A, size_t valCol=2)
    {
        readGridFromMatrix(A,valCol);
    }

    /// copy constructor
    Grid2d(const Grid2d& g)
    {
        rep = g.rep;
        min_dim1 = g.min_dim1;            // minimal value of dimension 1
        max_dim1 = g.max_dim1;            // maximal value of dimension 1
        min_dim2 = g.min_dim2;            // minimal value of dimension 2
        max_dim2 = g.max_dim2;            // maximal value of dimension 2
        mw_dim1 = g.mw_dim1;              // mesh width along dimension 1
        mw_dim2 = g.mw_dim2;              // mesh width along dimension 2
    }

    /// assignment operator
```

```cpp
    Grid2d& operator=(const Grid2d& g)
    {
        rep = g.rep;
        min_dim1 = g.min_dim1;            // minimal value of dimension 1
        max_dim1 = g.max_dim1;            // maximal value of dimension 1
        min_dim2 = g.min_dim2;            // minimal value of dimension 2
        max_dim2 = g.max_dim2;            // maximal value of dimension 2
        mw_dim1 = g.mw_dim1;              // mesh width along dimension 1
        mw_dim2 = g.mw_dim2;              // mesh width along dimension 2
        return *this;
    }

    /// overwrites the field with values of the file
    ///
    /// structure of the file:
    /// column1: dimension 1 of coordinate
    /// column2: dimension 2 of coordinate
    /// column3: value of the field
    /// the coordinate values are assumed to be sorted primary after the
    /// first dimension and secondary after the second dimension and for
    /// each dimension they have to be equidistant
    void readGridFromFile(const char* filename)
    {
        // read in the file
        Matrix<value_t> file(filename,3);
        readGridFromMatrix(file);
    }

    /// overwrites the field with values of the matrix
    ///
    /// \param A the matrix which specifies the grid
    /// structure of the matrix:
    /// column1: dimension 1 of coordinate
    /// column2: dimension 2 of coordinate
    /// column valCol: value of the field
    /// the coordinate values are assumed to be sorted primary after the
    /// first dimension and secondary after the second dimension and for
    /// each dimension they have to be equidistant
    ///
    /// \param valCol specifies the column which contains the values of
    /// the field (indices begin with 0)
    void readGridFromMatrix(const Matrix<value_t> A, size_t valCol=2)
    {
        // determine the number of values along dimension 2
        // this will be the number of columns of the field
        size_t k = 0;
        while (k+1 < A.rows()
                && A(k+1,0) >= A(k,0)-std::numeric_limits<value_t>::epsilon()
                && A(k+1,0) <= A(k,0)-std::numeric_limits<value_t>::epsilon()
              )
        {
            ++k;
        }
        size_t n = k+1;
        // determine the number of values along dimension 1
        // this will be the number of rows of the field
        size_t m = A.rows()/n;
        // resize the field
        rep.resize(m,n);
        // initialize lower and upper bounds of coordinates
        // according to the first line of the A
        min_dim1 = A(0,0);
        max_dim1 = A(0,0);
        min_dim2 = A(0,1);
        max_dim2 = A(0,1);
        // copy values from A into the field and determine
        // lower and upper bounds of the coordinates
        k = 0;
        for (size_t i = 0; i < rep.rows(); ++i)
        {
            for (size_t j = 0; j < rep.cols(); ++j)
            {
                if (A(k,0) < min_dim1)
                {
                    min_dim1 = A(k,0);
                }
                else if (A(k,0) > max_dim1)
                {
                    max_dim1 = A(k,0);
                }
                if (A(k,1) < min_dim2)
                {
                    min_dim2 = A(k,1);
                }
                else if (A(k,1) > max_dim2)
                {
                    max_dim2 = A(k,1);
                }
                rep(i,j) = A(k,valCol);
                ++k;
            }
        }
        // calculate mesh width
        mw_dim1 = (max_dim1−min_dim1)/(rep.rows()−1);
        mw_dim2 = (max_dim2−min_dim2)/(rep.cols()−1);
    }

    /// get the value and the gradient at position (dim1,dim2)
    /// using linear interpolation
    ///
    /// \param factor is a scaling factor to scale the field with
    /// this procedure will add the value and gradient to the
    /// referenced data structure of type ValGrad
    inline void linpValueAndGradient(value_t dim1, value_t dim2,
                                     value_t factor, ValGrad<value_t>& v) const
    {
        value_t r = (dim1−min_dim1)/mw_dim1;
        value_t c = (dim2−min_dim2)/mw_dim2;
        // aussure that the point is not on the border or
        // outside of the grid (otherwise just do nothing)
        if (r >= 0 && c >= 0 && r < (rep.rows()−1) && c < (rep.cols()−1))
        {
            size_t row_below = (size_t)floor(r);
            size_t col_below = (size_t)floor(c);
            size_t row_above = row_below+1;
            size_t col_above = col_below+1;
            value_t pos1[] = {getPosDim1(row_below),getPosDim2(col_below)};
            value_t pos2[] = {getPosDim1(row_above),getPosDim2(col_below)};
            value_t pos3[] = {getPosDim1(row_below),getPosDim2(col_above)};
            value_t pos4[] = {getPosDim1(row_above),getPosDim2(col_above)};
            value_t val1 = factor*rep(row_below,col_below);
            value_t val2 = factor*rep(row_above,col_below);
            value_t val3 = factor*rep(row_below,col_above);
            value_t val4 = factor*rep(row_above,col_above);
            value_t val12 = ((dim1−pos1[0])*val2 + (pos2[0]−dim1)*val1)
                            /mw_dim1;
            value_t val34 = ((dim1−pos3[0])*val4 + (pos4[0]−dim1)*val3)
                            /mw_dim1;
            value_t val13 = ((dim2−pos1[1])*val3 + (pos3[1]−dim2)*val1)
                            /mw_dim2;
            value_t val24 = ((dim2−pos2[1])*val4 + (pos4[1]−dim2)*val2)
                            /mw_dim2;
            v.val += ((dim1−pos1[0])*val24 + (pos2[0]−dim1)*val13)
                         /mw_dim1;
            v.grad_dim1 += (val24−val13)/mw_dim1;
            v.grad_dim2 += (val34−val12)/mw_dim2;
        }
    }

    /// return the minimal coordinate value along dimension 1
    inline value_t getMinDim1() const
    {
        return min_dim1;
    }

    /// return the minimal coordinate value along dimension 2
    inline value_t getMinDim2() const
    {
        return min_dim2;
    }
```

```

    /// return the maximal coordinate value along dimension 1
    inline value_t getMaxDim1() const
    {
        return max_dim1;
    }

    /// return the maximal coordinate value along dimension 2
    inline value_t getMaxDim2() const
    {
        return max_dim2;
    }

    /// return the distance between two grid points along
    /// dimension 1
    inline value_t dim1Dist() const
    {
        return mw_dim1;
    }

    /// return the distance between two grid points along
    /// dimension 2
    inline value_t dim2Dist() const
    {
        return mw_dim2;
    }

    /// get the position of the point at given indices
    /// along the first dimension
    inline value_t getPosDim1(size_t rowIndex) const
    {
        return min_dim1 + rowIndex*mw_dim1;
    }

    /// get the position of the point at given indices
    /// along the second dimension
    inline value_t getPosDim2(size_t colIndex) const
    {
        return min_dim2 + colIndex*mw_dim2;
    }

    /// get the center of the grid in the first dimension
    inline value_t getCenterDim1() const
    {
        return (max_dim1+min_dim1)/2;
    }

    /// get the center of the grid in the second dimension
    inline value_t getCenterDim2() const
    {
        return (max_dim2+min_dim2)/2;
    }

    /// scale all values of the grid by a factor f
    void scaleValues(value_t f)
    {
        rep.scale(f);
    }

    /// truncate values above a certain threshold
    void truncateValuesAbove(value_t v)
    {
        rep.truncateValuesAbove(v);
    }

    /// get number of rows (1st dimension) in the matrix representation
    size_t getRepRows() const
    {
        return rep.rows();
    }

    /// get number of columns (2nd dimension) in the matrix representation
    size_t getRepCols() const
    {
        return rep.cols();
    }

    /// get an element of the representation by row and column indices
    value_t getRepElem(size_t i, size_t j)
    {
        return rep.elem(i,j);
    }

protected:
    Matrix<value_t> rep;                  ///< matrix representation of the grid
    value_t min_dim1;                     ///< minimal value of dimension 1
    value_t max_dim1;                     ///< maximal value of dimension 1
    value_t min_dim2;                     ///< minimal value of dimension 2
    value_t max_dim2;                     ///< maximal value of dimension 2
    value_t mw_dim1;                      ///< mesh width along dimension 1
    value_t mw_dim2;                      ///< mesh width along dimension 2

};

}
#endif
```

```
// input.h: 2009-11-03, Yves Salathe
#ifndef __INPUT_H__
#define __INPUT_H__

#include "io.h"
#include "assert.h"
#include "matrix.h"
#include <string>
#include <sstream>

namespace simulation
{

class InputData {
public:
    typedef double real_t;
    typedef unsigned int size_t;

    InputData()
    {
    }

    InputData(const char _inputFileName[])
            : inputFileName(_inputFileName), indata(_inputFileName,2)
    {
    }

    InputData(const math::Matrix<std::string>& _indata,
            const char _inputFileName[] = "not specified")
            : inputFileName(_inputFileName), indata(_indata)
    {
    }

    InputData(const InputData& _indata)
            : inputFileName(_indata.inputFileName),
            indata(_indata.indata)
    {
    }

    void readFromFile(const char _inputFileName[])
    {
        indata.readFromFile(_inputFileName,2);
    }

    real_t getReal(const char* id) const
    {
        size_t i = indata.searchInColumn(0,id);
        if (i >= indata.rows()) {
            throw ExInputNotSpecified(id,inputFileName.c_str());
        }
        return std::atof(indata(i,1).c_str());
    }

    size_t getInt(const char* id) const
    {
        size_t i = indata.searchInColumn(0,id);
        if (i >= indata.rows()) {
            throw ExInputNotSpecified(id,inputFileName.c_str());
        }
        return std::atoi(indata(i,1).c_str());
    }

    std::string getString(const char* id) const
    {
        size_t i = indata.searchInColumn(0,id);
        if (i >= indata.rows()) {
            throw ExInputNotSpecified(id,inputFileName.c_str());
        }
        return indata(i,1);
    }

    class ExInputNotSpecified {
    public:
        ExInputNotSpecified(const char* _id, const char* _filename)
                : id(_id), filename(_filename) {}
```

```
            std::string id;
            std::string filename;
    };

private:
    std::string inputFileName;
    math::Matrix<std::string> indata;
};

}
#endif
```

```
// integrator.h: v2.4, 2009-05-06, Yves Salathe
#ifndef __INTEGRATOR_H__
#define __INTEGRATOR_H__

namespace simulation
{

///////////////////////////////////////////////////////////////////
/// \class Integrator
///
/// \brief      This class represents an interface for integrators like
///             VerletIntegrator or SymplecticEulerIntegrator
///////////////////////////////////////////////////////////////////

template<class simulation_t>
class Integrator
{
public:
    ///////////////////////////////
    // types and datastructures
    ///////////////////////////////

    //use the same types as the simulation
    typedef typename simulation_t::real_t real_t;
    typedef typename simulation_t::size_t size_t;
    typedef typename simulation_t::coord_t coord_t;
    typedef typename simulation_t::particle_list_t particle_list_t;
    typedef typename simulation_t::population_list_t population_list_t;

    ///////////////////////////////////////////////
    // virtual destructor
    ///////////////////////////////////////////////
    virtual ~Integrator() { }

    ///////////////////////////////////////////////
    // common methods
    ///////////////////////////////////////////////

    /// initialize the integrator with the current state of the
    /// simulation
    ///
    /// \param simulation       a reference to the simulation
    /// \param time             the start time of the step
    /// \param timestep         the size of the step (time difference)
    ///
    /// Note: this function has to be redefined by implementations of
    /// this interface. It is not declared as virtual for performance
    /// reasons. Instead static (compile-time) polymorphism can be
    /// obtained by templates (see e.g. ParticleSim).
    void initialize(simulation_t& simulation, real_t time,
            real_t timestep);

    /// timestepping: perform a timestep of dt
    /// (integrate equation of motion for all particles)
    ///
    /// \param simulation       a reference to the simulation
    /// \param time             the start time of the step
    /// \param timestep         the size of the step (time difference)
    /// \return the new reference time
    ///
    /// Note: this function has to be redefined by implementations of
    /// this interface. It is not declared as virtual for performance
    /// reasons. Instead static (compile-time) polymorphism can be
    /// obtained by templates (see e.g. ParticleSim).
    inline real_t performTimestep(simulation_t& simulation, real_t t,
            real_t dt);

    /// store the computed particle information back to the simulation
    ///
    /// \param simulation       a reference to the simulation
    ///
    /// Note: this function has to be redefined by implementations of
    /// this interface. It is not declared as virtual for performance
    /// reasons. Instead static (compile-time) polymorphism can be
    /// obtained by templates (see e.g. ParticleSim).
```

```
    void updateSimulation(simulation_t& simulation);
};

}
#endif
```

```
// io.h: 2009−03−16, Yves Salathe
#ifndef __IO_H__
#define __IO_H__

#include<fstream>
#include<string>
#include<sstream>

namespace io
{

//////////////////////////////////
// declaration of functions
//////////////////////////////////
template <class value_t>
unsigned int readline(std::ifstream& in,
                      value_t values[],
                      unsigned int maxValues);

template <class value_t>
unsigned int numblines(std::ifstream& infile);

/////////////////////////////////
// implementation of functions
/////////////////////////////////

/// reads the next line from a file
/// each value of type value_t gets stored in the array values[] until
/// maxValues is reached.
template <class value_t>
unsigned int readline(std::ifstream& in,
                      value_t values[],
                      unsigned int maxValues)
{
    // Check stream
    if (!in.good()) return 0;

    unsigned int valIndex = 0;                // index in values array
    std::string line;                         // one line from textfile in
    getline(in, line);                        // read line from in
    // extract values of type value_t
    std::istringstream istr(line);
    while (istr && valIndex < maxValues)
    {
        istr >> values[valIndex];
        if (!istr.fail())
        {
            ++valIndex;
        }
    }

    return valIndex;
}

/// counts the number of lines containing information of type value_t
/// in a file stream
template <class value_t>
unsigned int numblines(std::ifstream& infile)
{
    unsigned int maxval = 1;
    unsigned int numblines = 0;
    value_t values[maxval];
    // memorize postion in the stream
    std::streampos pos = infile.tellg();
    // go to the beginning of the stream
    infile.seekg(0,std::ios::beg);
    while (infile.good())
    {
        unsigned int valNum = readline(infile, values, maxval);
        if (valNum>0)
            numblines = numblines + 1;
    }
    // go back to original postion in the stream
    infile.clear();
    infile.seekg(pos);
```

```
    return numblines;
}

}
#endif
```

```cpp
// laser_detector.h: v2.4, 2009-05-06, Yves Salathe
#ifndef __LASER_DETECTOR_H__
#define __LASER_DETECTOR_H__

#include "observer.h"
#include "particle.h"
#include <cmath>

namespace simulation
{

///////////////////////////////
/// \class LaserDetector
///
/// \brief this class describes an observer which can be
///        used to detect particles inside a certain volume
///////////////////////////////
template<class observable_t>
class LaserDetector : public Observer<observable_t>
{
public:
    typedef typename Observer<observable_t>::real_t real_t;
    typedef typename Observer<observable_t>::size_t size_t;
    typedef typename observable_t::particle_list_t
            particle_list_t;

    using Observer<observable_t>::observable;
    using Observer<observable_t>::activateTime;
    using Observer<observable_t>::deactivateTime;
    using Observer<observable_t>::inverseFrequency;
    using Observer<observable_t>::getOutputStream;

    /// default constructor
    LaserDetector(observable_t& _observable,
            std::ostream& ostr = std::cout)
            : Observer<observable_t>(_observable, ostr),
            detectorPosZ(0), detectorRadius(1)
    {
    }

    /// normal copy constructor
    LaserDetector(LaserDetector<observable_t>& o)
            : Observer<observable_t>(o)
    {
        setDetectorPosZ(o.getDetectorPosZ());
        setDetectorRadius(o.getDetectorRadius());
    }

    /// copy constructor with the abilty to specify a new observable
    template<class observer_t>
    LaserDetector(observer_t& o,
            observable_t& _observable)
            : Observer<observable_t>(o, _observable)
    {
        setDetectorPosZ(o.getDetectorPosZ());
        setDetectorRadius(o.getDetectorRadius());
    }

    /// virtual destructor
    /// this makes sure that the destructors of the derived classes
    /// are called when their objects get "delete"ed through a
    /// refernece of this type
    virtual ~LaserDetector()
    {
    }

    /// this virtual method is (hopefully) called by observables
    ///
    /// \param time the time of the notification
    /// \param timestep the "resolution" of the time
    virtual void notify(real_t time, real_t timestep)
    {
        for (typename particle_list_t::const_iterator
                p = observable.getParticles().begin();
                p != observable.getParticles().end(); ++p)
```

```cpp
        {
            detect(*p,time);
        }
    }

    /// get the position of the laser along the z-axis
    inline real_t getDetectorPosZ() const
    {
        return detectorPosZ;
    }

    /// get the radius of the laser beam
    inline real_t getDetectorRadius() const
    {
        return detectorRadius;
    }

    /// set the position of the laser along the z-axis
    inline void setDetectorPosZ(real_t pos)
    {
        detectorPosZ = pos;
    }

    /// set the radius of the laser beam
    inline void setDetectorRadius(real_t r)
    {
        detectorRadius = r;
    }

private:
    /// position of the laser along the z-axis
    real_t detectorPosZ;
    /// radius of the laser beam
    real_t detectorRadius;

    /// detection of a particle
    inline void detect(const Particle& p,real_t time)
    {
        if (fabs(p.pos.getZ()-detectorPosZ) < detectorRadius
                && fabs(p.pos.getY()) < detectorRadius)
        {
            getOutputStream() <<0<<" "<<0<<" "<<0<<" "<<time<<" ";
            p.outputInformation(getOutputStream());
            getOutputStream() << std::endl;
        }
    }
};

}
#endif
```

```
// leapfrog_integrator.h: v2.42, 2009-05-07, Yves Salathe
#ifndef __LEAPFROG_INTEGRATOR_H__
#define __LEAPFROG_INTEGRATOR_H__

#include "integrator.h"
#include "particle.h"
#include "matrix.h"
#include "vector3d.h"
#include "assert.h"
#include "exceptions.h"

namespace simulation
{

////////////////////////////////////
/// \class LeapfrogIntegrator
///
/// \brief This class describes a method to integrate newton's equations
///        of motion. It can be used as an integrator in the class
///        of type simulation_t (usually ParticleSim)
////////////////////////////////////
template<class simulation_t>
class LeapfrogIntegrator : public Integrator<simulation_t>
{
public:
    ///////////////////////////////////
    // type definitions
    ///////////////////////////////////

    //use the same types as the simulation
    typedef typename simulation_t::real_t real_t;
    typedef typename simulation_t::size_t size_t;
    typedef typename simulation_t::coord_t coord_t;
    typedef typename simulation_t::particle_list_t particle_list_t;
    typedef typename simulation_t::population_list_t population_list_t;

    /// constructor
    LeapfrogIntegrator() : nParticles(0)
    {
    }

    /// destructor
    ~LeapfrogIntegrator()
    {
        if(nParticles > 0) {
            delete[] id;
            delete[] popId;
            delete[] pos;
            delete[] vel;
            delete[] accel;
        }
    }

    /// initialize the integrator with the current state of the
    /// simulation
    ///
    /// \param simulation       a reference to the simulation
    /// \param time             the start time of the step
    /// \param timestep         the size of the step (time difference)
    void initialize(simulation_t& simulation, real_t time,
            real_t timestep)
    {
        // free previously allocated memory
        if(nParticles > 0) {
            delete[] id;
            delete[] popId;
            delete[] pos;
            delete[] vel;
            delete[] accel;
        }
        // get a reference to the list of particles
        const particle_list_t& particles = simulation.getParticles();
        nParticles = particles.size();
        // allocate memory
        id = new size_t[nParticles];
```

```
        popId = new size_t[nParticles];
        size_t n3 = 3*nParticles;
        pos = new real_t[n3];
        vel = new real_t[n3];
        accel = new real_t[n3];
        // initialize masses and positions
        typename particle_list_t::const_iterator p = particles.begin();
        for(size_t i = 0; i<nParticles; ++i) {
            size_t i3 = 3*i;
            id[i] = p->id;
            popId[i] = p->popId;
            p->pos.writeToArray(&pos[i3]);
            p->vel.writeToArray(&vel[i3]);
            ++p;
        }
    }

    /// timestepping: perform a timestep of dt
    /// (integrate equation of motion for all particles)
    ///
    /// \param simulation       a reference to the simulation
    /// \param time             the start time of the step
    /// \param timestep         the size of the step (time difference)
    /// \return the new reference time
    inline real_t performTimestep(simulation_t& simulation, real_t t,
            real_t dt)
    {
        size_t n3 = 3*nParticles;
        // update positions by a half timestep
        for (size_t i = 0; i<n3; ++i)
        {
            pos[i] += .5*dt*vel[i];
        }
        // calculate the acceleration of each particle at that time
        simulation.calculateAccelerations(t+.5*dt,dt,nParticles, popId,
                pos, accel);
        // update velocities and positions according to the newly
        // calculated accelerations
        for (size_t i = 0; i<n3; ++i) {
            vel[i] += dt*accel[i];
            pos[i] += .5*dt*vel[i];
        }
        return t+dt;
    }

    /// store the computed particle information back to the simulation
    void updateSimulation(simulation_t& simulation)
    {
        // get a reference to the list of particles
        particle_list_t& particles = simulation.getParticles();
        typename particle_list_t::iterator p = particles.begin();
        for(size_t i = 0; i<nParticles; ++i) {
            while(p->id != id[i] && p != particles.end()) {
                ++p;
            }
            if (p != particles.end()) {
                size_t i3 = 3*i;
                p->pos.setFromArray(&pos[i3]);
                p->vel.setFromArray(&vel[i3]);
            } else {
                throw ExSizeMismatch();
            }
        }
    }

private:

    /// the number of particles
    size_t nParticles;
    /// the ids of the particles
    size_t* id;
    /// an array containing the population id of each particle
    size_t* popId;
    /// an array containing the masses of each particle
    real_t* mass;
    /// an array of size 3*numbParticles containing the positions at
    /// time t of the particles in cartesian coordinates
    real_t* pos;
    /// an array of size 3*numbParticles containing the velocities
    /// at time t of the particles in cartesian coordinates
    real_t* vel;
    /// an array of size 3*numbParticles containing the accelerations of the
    /// particles in cartesian coordinates
    real_t* accel;

    /// initialize acceleration for all particles
    inline void initializeAccelerations(real_t v = 0) const
    {
        size_t n3 = 3*nParticles;

        for (size_t i = 0; i<n3; ++i) {
            accel[i] = 0;
        }
    }

};

}
#endif
```

```cpp
// magnetictrap_builder.h: v2.4, 2009−05−21, Yves Salathe
#ifndef __MAGNETICTRAP_BUILDER__
#define __MAGNETICTRAP_BUILDER__

#include "particlesim_builder.h"
#include "magnetictrap.h"
#include "zeemandecel.h"
#include "region_observer.h"
#include "trappulsegen_observer.h"
#include "matrix.h"
#include "grid.h"
#include "input.h"

namespace simulation {
class MagneticTrapBuilder : public ParticleSimBuilder<MagneticTrap> {
public:

    static const size_t nLinesInputData = 18;

    MagneticTrapBuilder(
            const char _inputFileName[] = "inputdata_trap.txt")
                :
                ParticleSimBuilder<MagneticTrap>(),
                indata(_inputFileName)
    {
        setParameters();
    }

    MagneticTrapBuilder(const math::Matrix<std::string>& _indata)
                :
                ParticleSimBuilder<MagneticTrap>(),
                indata(_indata)
    {
        setParameters();
    }

    MagneticTrapBuilder(const ParticleSim<ZeemanDecel>& decelSim,
            const char _inputFileName[] = "inputdata_trap.txt")
                :
                ParticleSimBuilder<MagneticTrap>(decelSim),
                indata(_inputFileName)
    {
        inheritStateFromDeceleratorSimulation(decelSim);
        setParameters();
    }

    MagneticTrapBuilder(const ParticleSim<ZeemanDecel>& decelSim,
            const math::Matrix<std::string>& _indata)
                :
                ParticleSimBuilder<MagneticTrap>(decelSim),
                indata(_indata)
    {
        inheritStateFromDeceleratorSimulation(decelSim);
        setParameters();
    }

    MagneticTrapBuilder(const ParticleSim<ZeemanDecel>& decelSim,
            const InputData& _indata)
                :
                ParticleSimBuilder<MagneticTrap>(decelSim),
                indata(_indata)
    {
        inheritStateFromDeceleratorSimulation(decelSim);
        setParameters();
    }

    virtual ~MagneticTrapBuilder()
    {
    }

    void readDecelBFieldFromFile(
            const char filename[] = "decelmapaxrho.txt")
    {
        ////////////////////////////////////////////////////////////
        //read in b−field of the decelerator coils                //
```

```cpp
        ////////////////////////////////////////////////////////////

        // read the axial and radial field values into a matrix
        math::Matrix<MagneticTrap::real_t> decelB("decelmapaxrho.txt",4);
        setDecelBFieldFromMatrix(decelB);
    }

    void setDecelBFieldFromMatrix(const math::Matrix<double>& decelB)
    {
        ////////////////////////////////////////////////////////////
        //read in b−field of the decelerator coils                //
        ////////////////////////////////////////////////////////////

        // extract axial part of the field
        math::Grid2d<MagneticTrap::real_t> decelBz(decelB,2);
        // extract radial part of the field
        math::Grid2d<MagneticTrap::real_t> decelBr(decelB,3);
        simulation−>engine.lastDecelCoil.setBField(decelBz,decelBr);
    }

    void readTrapBFieldFromFile(
            const char filename[] = "trapmapaxrad.txt")
    {
        ////////////////////////////////////////////////////////////
        //read in b−field of the trap coils                       //
        ////////////////////////////////////////////////////////////

        // read the axial and radial field values into a matrix
        math::Matrix<MagneticTrap::real_t> trapB(filename,4);
        setTrapBFieldFromMatrix(trapB);
    }

    void setTrapBFieldFromMatrix(const math::Matrix<double>& trapB)
    {
        ////////////////////////////////////////////////////////////
        //read in b−field of the trap coils                       //
        ////////////////////////////////////////////////////////////

        // extract axial part of the field
        math::Grid2d<MagneticTrap::real_t> trapBz(trapB,2);
        // extract radial part of the field
        math::Grid2d<MagneticTrap::real_t> trapBr(trapB,3);
        simulation−>engine.coil1.setBField(trapBz,trapBr);
        simulation−>engine.coil2.setBField(trapBz,trapBr);
    }

    void readFirstCoilPulsingFromFile(
            const char filename[] = "trap_pulses1.txt")
    {
        math::Matrix<MagneticTrap::real_t> trap_pulses1(filename,
                3+indata.getInt("scaling_polynomial_degree"));
        simulation−>engine.coil1.setPulses(trap_pulses1);
    }

    void readSecondCoilPulsingFromFile(
            const char filename[] = "trap_pulses2.txt")
    {
        math::Matrix<MagneticTrap::real_t> trap_pulses2(filename,
                3+indata.getInt("scaling_polynomial_degree"));
        simulation−>engine.coil2.setPulses(trap_pulses2);
    }

    ///////////////////////////////////////////////////
    ///create and add trap−region−observer           //
    ///////////////////////////////////////////////////
    RegionObserver<ParticleSim<MagneticTrap> >* addTrapRegionObserver(
            const char output_filename[] = "output_trapregion.txt")
    {
        RegionObserver<ParticleSim<MagneticTrap> >* trapRegionObserver
                = addObserver<RegionObserver<ParticleSim<
                        MagneticTrap> > >(output_filename);
        setTrapRegionObserverParameters(*trapRegionObserver);
        return trapRegionObserver;
    }

    RegionObserver<ParticleSim<MagneticTrap> >* addTrapRegionObserver(
            std::ostream& trapRegionOutput)
    {
        RegionObserver<ParticleSim<MagneticTrap> >* trapRegionObserver
                = addObserver<RegionObserver<ParticleSim<
                        MagneticTrap> > >(trapRegionOutput);
        setTrapRegionObserverParameters(*trapRegionObserver);
        return trapRegionObserver;
    }

    TrapPulseGenObserver<ParticleSim<MagneticTrap> >*
            addPulseGenObserver(
                    const DecelPulseGenObserver<ParticleSim<ZeemanDecel> >&
                        decelPG,
                    const char output_filename[]
                        = "output_pg_trap.txt",
                    const char input_filename[]
                        = "inputdata_pg_trap.txt")
    {
        TrapPulseGenObserver<ParticleSim<MagneticTrap> >* trapPG
                = addPulseGenObserver(output_filename,input_filename);
        const math::Matrix<real_t>& phasedeg
                = decelPG.getPhaseAngleDegreesMatrix();
        trapPG->setLastDecelCoilEndPulsePhaseDeg(
                phasedeg(phasedeg.rows()-1,phasedeg.cols()-1));
        trapPG->setDecelCoilDist(distBetweenStageCoils);
        return trapPG;
    }

    TrapPulseGenObserver<ParticleSim<MagneticTrap> >*
            addPulseGenObserver(const char output_filename[]
                        = "output_pg_trap.txt",
                    const char input_filename[]
                        = "inputdata_pg_trap.txt")
    {
        TrapPulseGenObserver<ParticleSim<MagneticTrap> >* pulseGen
                = addObserver<TrapPulseGenObserver<ParticleSim<
                        MagneticTrap> > >(output_filename);
        math::Matrix<std::string> inputPG(input_filename,2);
        Assert<ExWrongInput>(inputPG.rows() == 6);
        // TODO: provide a more flexible input format
        pulseGen->setPulseOverlap(std::atof(inputPG(0,1).c_str()));
        pulseGen->setFrontTrapCoilEndFirstPulsePhaseDeg(
                std::atof(inputPG(1,1).c_str()));
        pulseGen->setFrontTrapCoilStartSecondPulsePhaseDeg(
                std::atof(inputPG(2,1).c_str()));
        std::valarray<real_t> cf1(std::atof(inputPG(3,1).c_str()),1);
        std::valarray<real_t> cf2(std::atof(inputPG(4,1).c_str()),1);
        std::valarray<real_t> cr(std::atof(inputPG(5,1).c_str()),1);
        pulseGen->setFrontTrapCoilFirstPulseCoeff(cf1);
        pulseGen->setFrontTrapCoilSecondPulseCoeff(cf2);
        pulseGen->setRearTrapCoilPulseCoeff(cr);
        return pulseGen;
    }


private:

    InputData indata;

    real_t distBetweenStageCoils;

    void setParameters()
    {
        ////////////////////////////////////////////////////////////
        // setting simulation parameters according to inputfile //
        ////////////////////////////////////////////////////////////

        // numerical timesteps in us (should be 1-10ns)
        simulation->setTimestep(indata.getReal("timesteps_ns")/1000);
        // time when simulation stops
        simulation->setEndTime(indata.getReal("time_simulation_stops_us"));

        ////////////////////////////////////////////////////////////
```

```
        // setting trap parameters according to inputfile       //
        //////////////////////////////////////////////////////////

        // ramping time B-field of the trap coils in us
        simulation->engine.coil1.setRampOn(indata.getReal("ramp_time_trap_coil_ns")/1
000);
        simulation->engine.coil2.setRampOn(indata.getReal("ramp_time_trap_coil_ns")/1
000);
        // ramping time B-field of the trap coils in us
        simulation->engine.coil1.setRampOff(indata.getReal("ramp_time_trap_coil_ns")/
1000);
        simulation->engine.coil2.setRampOff(indata.getReal("ramp_time_trap_coil_ns")/
1000);
        // current at which the coil has been simulated
        simulation->engine.coil1.setSimCurrent(indata.getReal("trap_sim_current_A"))
;
        simulation->engine.coil2.setSimCurrent(indata.getReal("trap_sim_current_A"))
;
        // unscaled current in the first trap coil
        simulation->engine.coil1.setUnscaledCurrent(indata.getReal("trap_coil1_refere
nce_current_A"));
        // unscaled current in the second trap coil
        simulation->engine.coil2.setUnscaledCurrent(indata.getReal("trap_coil2_refere
nce_current_A"));
        // set the center of the trap and the distance between the trap coils
        simulation->engine.setCenterAndDist(indata.getReal("trappos_mm"),
                              indata.getReal("distance_trap_coils_mm"));
        /// set the maximum radius along the z-axis where Particles can be
        /// trapped
        simulation->engine.setTrapRegionRadiusZ(indata.getReal("max_trap_region_radiu
s_z_mm"));
        // lower bound of z-position for the coordinates of particles
        simulation->engine.setLowerBoundZ(indata.getReal("lower_bound_z_mm"));
        // upper bound of z-position for the coordinates of particles
        simulation->engine.setUpperBoundZ(indata.getReal("upper_bound_z_mm"));
        // upper bound for the distance from the particles to the axis
        simulation->engine.setDecelBoundR(indata.getReal("upper_bound_r_outside_trap_m
m"));
        // upper bound for the distance from the particles to the axis
        simulation->engine.setTrapBoundR(indata.getReal("upper_bound_r_in_trap_mm"))
;
    }

    void setTrapRegionObserverParameters(
            RegionObserver<ParticleSim<MagneticTrap> >&
                    trapRegionObserver)
    {
        trapRegionObserver.setCenterPosZ(simulation->engine.getCenterPosZ());
        trapRegionObserver.setRadiusZ(indata.getReal("trap_region_radius_z_mm"));
        trapRegionObserver.setRadiusX(indata.getReal("trap_region_radius_r_mm"));
        trapRegionObserver.setRadiusY(indata.getReal("trap_region_radius_r_mm"));
        trapRegionObserver.setUpperTotalEnergyAccordingToAbsB(
                indata.getReal("upper_trap_abs_B_T"));
        trapRegionObserver.setActivateTime(simulation->getTime());
        trapRegionObserver.setInverseFrequency(indata.getReal("region_observer_inverse
_frequency_us"));
    }

    void inheritStateFromDeceleratorSimulation(
            const ParticleSim<ZeemanDecel>& decelSim)
    {
        simulation->engine.lastDecelCoil.setPosZ(decelSim.engine.getStageCoilPos
(
                                       decelSim.engine.getNumbStages()-1,
                                       decelSim.engine.getNumbStageCoils()-1));
        simulation->engine.lastDecelCoil.setSimCurrent(decelSim.engine.getStageS
imCurrent());
        simulation->engine.lastDecelCoil.setUnscaledCurrent(decelSim.engine.getS
tageCurrent());
        simulation->engine.lastDecelCoil.setRampOn(decelSim.engine.getStageRampO
n());
        simulation->engine.lastDecelCoil.setRampOff(decelSim.engine.getStageRamp
Off());
        simulation->engine.setDecelBoundR(decelSim.engine.getBoundR());
        const math::Matrix<Coil::real_t>& lastStagePulses =
            decelSim.engine.getOrigStagePulses(decelSim.engine.getNumbStages()−1
);
        size_t indLastPulse = decelSim.engine.getNumbStageCoils()−1;
        std::valarray<Coil::real_t> coeff(1);
        coeff[0] = 1.0;
        simulation−>engine.lastDecelCoil.addPulse(Coil::Pulse(
                                        lastStagePulses(indLastPulse,0),
                                        lastStagePulses(indLastPulse,1),coeff));
        simulation−>engine.setIncouplingTime(decelSim.engine.getIncouplingTime()
);
        distBetweenStageCoils = decelSim.engine.getCoilDist();

        //////////////////////////////////////////////////////
        // copy some observers from the simulation of the   //
        // Zeeman−decelerator to the simulation of the trap //
        //////////////////////////////////////////////////////

        // copy laser detectors
        observer_ptr_list_t new_lasers = simulation−>copyObserversFromList
                <ParticleSim<ZeemanDecel>,
                LaserDetector<ParticleSim<ZeemanDecel> >,
                LaserDetector<ParticleSim<MagneticTrap> > >
                (decelSim.getObservers());
        observers.insert(observers.begin(),
                new_lasers.begin(),new_lasers.end());

        // copy trajectories observers
        observer_ptr_list_t new_trajs = simulation−>copyObserversFromList
                <ParticleSim<ZeemanDecel>,
                TrajectoriesObserver<ParticleSim<ZeemanDecel> >,
                TrajectoriesObserver<ParticleSim<MagneticTrap> > >
                (decelSim.getObservers());
        observers.insert(observers.begin(),
                new_trajs.begin(),new_trajs.end());
    }
};


}

#endif
```

```cpp
// magnetictrap.h: v2.4, 2009−05−06, Yves Salathe
#ifndef __MAGNETICTRAP_H__
#define __MAGNETICTRAP_H__

#include "engine.h"
#include "coil.h"
#include "assert.h"
#include "exceptions.h"
#include "basic_math.h"
#include "vector3d.h"
#include "particle.h"
#include <cmath>
#include <limits>

namespace simulation
{

///////////////////////////////
/// \class MagneticTrap
///
/// \brief This is the class which implements the magnetic trap
///        as an engine which can be used in the class
///        ParticleSim<MagneticTrap,integrator_t>
///////////////////////////////
class MagneticTrap : public Engine
{
public:

    //////////////////////////////////////////////////
    // public variables (properties of the class)   //
    //////////////////////////////////////////////////

    /// the trap coils
    Coil coil1;
    Coil coil2;
    /// the last decelerator coil
    Coil lastDecelCoil;

    //////////////////////////////////////////////////
    // constructors                                 //
    //////////////////////////////////////////////////
    MagneticTrap()
    {
    }

    //////////////////////////////////////////////////////////
    //setters                                              //
    //////////////////////////////////////////////////////////

    /// sets the position of the center of the trap and the
    /// distance between the two trap coils along the z−axis
    inline void setCenterAndDist(real_t p, real_t d)
    {
        coil1.setPosZ(p−d/2);
        coil2.setPosZ(p+d/2);
    }

    /// the maximum radius along the z−axis where Particles can be
    /// trapped
    inline void setTrapRegionRadiusZ(real_t l)
    {
        trapRegionRadiusZ = l;
    }

    /// sets the lower bound for the z−coordinates of the particles
    inline void setLowerBoundZ(real_t d)
    {
        lowerBoundZ = d;
    }

    /// sets the upper bound for the z−coordinates of the particles
    inline void setUpperBoundZ(real_t d)
    {
        upperBoundZ = d;
    }
```

```cpp

    /// sets the upper bound for the distance to the axis for the
    /// particles in the trap
    inline void setTrapBoundR(real_t d)
    {
        trapBoundR = d;
    }

    /// sets the upper bound for the distance to the axis for the
    /// particles in the decelerator
    inline void setDecelBoundR(real_t d)
    {
        decelBoundR = d;
    }

    /// sets the incoupling time (time offset for the pulses) for
    /// coils in the trap simulation
    inline void setIncouplingTime(real_t dt)
    {
        coil1.setIncouplingTime(dt);
        coil2.setIncouplingTime(dt);
        lastDecelCoil.setIncouplingTime(dt);
    }

    ///////////////////////////////
    //getters
    ///////////////////////////////

    /// get the position of the center of the trap along the z−axis
    inline real_t getCenterPosZ() const
    {
        return (coil1.getPosZ()+coil2.getPosZ())/2;
    }

    /// get the distance between the two trap coils along the z−axis
    inline real_t getDistBetweenCoilsZ() const
    {
        return coil2.getPosZ()−coil1.getPosZ();
    }

    /// set the maximum radius along the z−axis where Particles can be
    /// trapped
    inline real_t getTrapRegionRadiusZ() const
    {
        return trapRegionRadiusZ;
    }

    /// get the lower bound for the z−coordinates of the particles
    inline real_t getLowerBoundZ() const
    {
        return lowerBoundZ;
    }

    /// get the upper bound for the z−coordinates of the particles
    inline real_t getUpperBoundZ() const
    {
        return upperBoundZ;
    }

    /// get the upper bound for the distance to the axis for the
    /// particles in the trap
    inline real_t getTrapBoundR() const
    {
        return trapBoundR;
    }

    /// get the upper bound for the distance to the axis for the
    /// particles in the decelerator
    inline real_t getDecelBoundR() const
    {
        return decelBoundR;
    }

    ////////////////////////////////////////////////////////
    // methods used by ParticleSim<MagneticTrap,integrator_t>
    /////////////////////////////////////////////////////////////

    /// calculate the value and the gradient of the absolute values
    /// of the B−field at the given axial and radial coordinates
    inline math::ValGrad<real_t> calculateAbsBField (real_t z,
            real_t r) const
    {
        // sum up the fields of each active coil component−wise
        math::ValGrad<real_t> Bz(0.0,0.0,0.0);
        math::ValGrad<real_t> Br(0.0,0.0,0.0);
        for (size_t i = 0; i<numbActiveCoils; ++i)
        {
            activeCoils[i]−>addBField(z,r,Bz,Br);
        }
        // out of the two components calculate the absolute
        // values and the gradient of the absolute values of the
        // field
        return calculateAbsBFieldFromComponents(Bz,Br);
    }

    /// calculate the number of timesteps until the next event occurs
    /// returns ceil((endtime−time)/timestep) if no event will happen
    inline size_t timestepsUntilNextEvent(real_t time, real_t timestep,
                                          real_t endtime) const
    {
        int min_timesteps = (int)ceil((endtime−time)/timestep);
        if (coil1.isActive())
        {
            int ts = (int)floor((coil1.getEndOfActive()−time)/timestep);
            if(ts > 0 && ts < min_timesteps) min_timesteps = ts;
        }
        else if (coil1.hasNextPulse())
        {
            int ts = (int)floor((coil1.getStartOfNextPulse()−time)/timestep)+1;
            if(ts > 0 && ts < min_timesteps) min_timesteps = ts;
        }
        if (coil2.isActive())
        {
            int ts = (int)floor((coil2.getEndOfActive()−time)/timestep);
            if(ts > 0 && ts < min_timesteps) min_timesteps = ts;
        }
        else if (coil2.hasNextPulse())
        {
            int ts = (int)floor((coil2.getStartOfNextPulse()−time)/timestep)+1;
            if(ts > 0 && ts < min_timesteps) min_timesteps = ts;
        }
        if (lastDecelCoil.isActive())
        {
            int ts = (int)floor((lastDecelCoil.getEndOfActive()−time)/timestep);
            if(ts > 0 && ts < min_timesteps) min_timesteps = ts;
        }
        else if (lastDecelCoil.hasNextPulse())
        {
            int ts = (int)floor((lastDecelCoil.getStartOfNextPulse()−time)/times
tep)+1;
            if(ts > 0 && ts < min_timesteps) min_timesteps = ts;
        }
        return min_timesteps;
    }

    /// prepare timestepping (this function is called by
    /// ParticleSim<MagneticTrap,integrator_t> at each
    /// event before the actual timestepping is done)
    ///
    /// this inserts the active coils into the list of active coils
    inline void prepareTimestepping(real_t time, real_t timestep)
    {
        insertActiveCoils(time,timestep);
#ifdef __DEBUG__
        debugOutputAtEvent(time);
#endif
    }

    /// prepare the magnetic field (e.g. calculate the scaling factors)
    inline void setTime(real_t time, real_t timestep)
```

```
    {
        calculateScalingFactors(time);
    }

    /// check wheter the position (z,r) is inside the boundaries
    inline bool isPointInsideBoundaries(real_t z, real_t r) const
    {
        bool result = false;
        if (fabs(z−getCenterPosZ()) < trapRegionRadiusZ)
        {
            if (r < trapBoundR)
            {
                result = true;
            }
        } else if (r < decelBoundR
                   && z > lowerBoundZ
                   && z < upperBoundZ)
        {
            result = true;
        }
        return result;
    }

    /// check whether a low−field−seeking particle still can be trapped
    inline bool mayLFSParticleBeAccepted(real_t time, const Particle& p,
            const ParticlePopulation& pop)
    {
        bool result = true;
        if (p.pos.getZ() > coil2.getPosZ()) {
            result = false;
        } else if (!coil1.hasNextPulse()) {
            if (p.pos.getZ() < coil1.getPosZ()) {
                result = false;
            }
        }
        return result;
    }

    /// returns true if at least one coil is active
    inline bool isActive() const
    {
        return numbActiveCoils > 0;
    }

private:

    ///////////////////////////////////////////////////////
    //definition of variables (properties of the class)
    ///////////////////////////////////////////////////////

    /// the maximum radius along the z−axis where Particles can be
    /// trapped
    real_t trapRegionRadiusZ;
    /// the upper bound for the z−coordinates of the particles
    real_t lowerBoundZ;
    /// the upper bound for the z−coordinates of the particles
    real_t upperBoundZ;
    /// the upper bound for the distance to the axis for the particles
    /// in the trap
    real_t trapBoundR;
    /// the upper bound for the distance to the axis for the particles
    /// in the decelerator
    real_t decelBoundR;
    /// an array of references to the active coils
    Coil *activeCoils[3];
    /// the number of active coils
    size_t numbActiveCoils;


    ///////////////////////////////
    //private methods
    ///////////////////////////////

    /// insert pointers to the active coils into the list of active coils
    inline void insertActiveCoils(real_t time, real_t timestep)
    {
        numbActiveCoils = 0;
        coil1.setCurrentAndNextPulse(time,timestep);
        if (coil1.isActive())
        {
            activeCoils[numbActiveCoils] = &coil1;
            numbActiveCoils++;
        }
        coil2.setCurrentAndNextPulse(time,timestep);
        if (coil2.isActive())
        {
            activeCoils[numbActiveCoils] = &coil2;
            numbActiveCoils++;
        }
        lastDecelCoil.setCurrentAndNextPulse(time,timestep);
        if (lastDecelCoil.isActive())
        {
            activeCoils[numbActiveCoils] = &lastDecelCoil;
            numbActiveCoils++;
        }
    }

    /// calculate the scaling factor for all active coils
    inline void calculateScalingFactors(real_t time)
    {
        for (size_t i = 0; i<numbActiveCoils; ++i)
        {
            activeCoils[i]->calculateScalingFactor(time);
        }
    }

    /// calculate the value and the gradient of the absolute values
    /// of the field from given values for the value and gradient of
    /// each component
    inline math::ValGrad<real_t> calculateAbsBFieldFromComponents(
        const math::ValGrad<real_t>& Bz,
        const math::ValGrad<real_t>& Br) const
    {
        real_t absB = sqrt(Bz.val*Bz.val+Br.val*Br.val);
        if (absB >= std::numeric_limits<real_t>::epsilon())
        {
            return math::ValGrad<real_t>(absB,
                                         (Bz.val*Bz.grad_dim1+Br.val*Br.grad_dim
1)/absB,
                                         (Bz.val*Bz.grad_dim2+Br.val*Br.grad_dim
2)/absB);
        }
        else
        {
            return math::ValGrad<real_t>(0,0,0);
        }
    }

    /// if wanted, output some debugging information at each pulse
    inline void debugOutputAtEvent(real_t time) const
    {
        std::cerr << "-- trap event at t = " << time << " us --\n";
        if (coil1.isActive())
        {
            std::cerr << "trap coil 1 is active\n"
                      << "time when it gets inactive: "
                      << coil1.getEndOfActive() << std::endl
                      << std::endl;
        }
        else
        {
            std::cerr << "trap coil 1 is inactive\n";
            if (coil1.hasNextPulse())
            {
                std::cerr << "time when it gets active : "
                          << coil1.getStartOfNextPulse() << std::endl
                          << std::endl;
            }
        }
        if (coil2.isActive())
```

```
        {
            std::cerr << "trap coil 2 is active\n"
                      << "time when it gets inactive: "
                      << coil2.getEndOfActive() << std::endl
                      << std::endl;
        }
        else
        {
            std::cerr << "trap coil 2 is inactive\n";
            if (coil1.hasNextPulse())
            {
                std::cerr << "time when it gets active : "
                          << coil2.getStartOfNextPulse() << std::endl
                          << std::endl;
            }
        }
        if (lastDecelCoil.isActive())
        {
            std::cerr << "the last decelerator coil is active\n"
                      << "time when it gets inactive: "
                      << lastDecelCoil.getEndOfActive() << std::endl
                      << std::endl;
        }
        else
        {
            std::cerr << "the last decelerator coil is inactive\n";
            if (lastDecelCoil.hasNextPulse())
            {
                std::cerr << "time when it gets active : "
                          << lastDecelCoil.getStartOfNextPulse() << std::endl
                          << std::endl;
            }
        }
    }
};

}
#endif
```

```
// matrix.h: 2009-02-18, Yves Salathe
#ifndef __MATRIX_H__
#define __MATRIX_H__

#include "io.h"
#include "assert.h"
#include <string>
#include <sstream>
#include <valarray>

namespace math
{

///////////////////////////////////////////////
///
/// \class Matrix
///
/// a simple matrix representation
///
///////////////////////////////////////////////
template <class value_t>
class Matrix
{
public:
    typedef unsigned int size_t;

    /// default constructor creates a matrix with zero size
    Matrix() : val(0), nrows(0), ncols(0)
    {
    }

    /// this constructor creates an empty mxn matrix
    Matrix(size_t m, size_t n)
    {
        resize(m,n);
    }

    /// this constructor creates a matrix from a file stream
    /// ncol: number of columns
    Matrix(std::ifstream& infile, size_t ncol)
    {
        nrows = 0;
        ncols = 0;
        readFromFile(infile, ncol);
    }

    /// this constructor creates a matrix from a file with filename
    /// ncol: number of columns
    Matrix(const char* filename, size_t ncol)
    {
        nrows = 0;
        ncols = 0;
        readFromFile(filename, ncol);
    }

    /// copy constructor
    Matrix(const Matrix& A)
    {
        resize(A.rows(),A.cols());
        for (size_t i = 0; i < nrows; ++i)
        {
            for (size_t j = 0; j < ncols; ++j)
            {
                elem(i,j) = A(i,j);
            }
        }
    }

    /// destructor
    ~Matrix()
    {
    }

    /// assignment operator
    Matrix& operator=(const Matrix& A)
```

```
    {
        resize(A.rows(),A.cols());
        for (size_t i = 0; i < nrows; ++i)
        {
            for (size_t j = 0; j < ncols; ++j)
            {
                elem(i,j) = A(i,j);
            }
        }
        return *this;
    }

    /// resize the matrix
    /// Note: this will delete all values!
    void resize(size_t m, size_t n)
    {
        // delete old values
        nrows = m;
        ncols = n;
        val.resize(nrows*ncols);
    }

    /// overwrite the matrix with the contents of a file stream
    /// Lines correspond to rows of the matrix and the values in each
    /// row are separated by whitespaces.
    /// ncol: specifies the number of columns
    void readFromFile(std::ifstream& infile, size_t ncol)
    {
        // open the file
        Assert<ExFileOpenError>(infile.is_open());
        // determine number of rows and columns
        size_t numblinesb = io::numblines<value_t>(infile);
        numblinesb = io::numblines<value_t>(infile);
        // resize the matrix
        resize(numblinesb, ncol);
        // read in all rows
        value_t values[ncols];
        size_t i = 0;
        while (infile.good())
        {
            size_t valNum = io::readline(infile, values, ncols);
            if (valNum>0)
            {
                if (valNum == ncols)
                {
                    for (size_t j = 0; j < ncols; j++)
                    {
                        if (j < valNum)
                        {
                            elem(i,j) = values[j];
                        }
                    }
                    ++i;
                }
                else
                {
                    throw ExWrongInput();
                }
            }
        }
    }

    /// overwrite the matrix with the contents of a file with filename
    /// Lines correspond to rows of the matrix and the values in each
    /// row are separated by whitespaces.
    /// ncol: specifies the number of columns
    void readFromFile(const char *filename, size_t ncol)
    {
        std::ifstream infile(filename);
        readFromFile(infile,ncol);
        infile.close();
    }

    /// write the matrix to a file
    void writeToFile(const char *filename) const
```

```
        {
            std::ofstream outfile(filename);
            writeToOutputStream(outfile);
            outfile.close();
        }

        /// write the matrix to an output stream
        void writeToOutputStream(std::ostream& ostr) const
        {
            for(size_t i = 0; i<nrows; ++i) {
                for(size_t j = 0; j<ncols; ++j) {
                    ostr << elem(i,j);
                    if (j < ncols-1) ostr << "\t";
                }
                ostr << std::endl;
            }
        }

        /// get value at (rowInd, colInd) constant
        inline const value_t& elem(size_t rowInd, size_t colInd) const
        {
            return val[rowInd*ncols+colInd];
        }

        /// get value at (rowInd, colInd) for modification
        inline value_t& elem(size_t rowInd, size_t colInd)
        {
            return val[rowInd*ncols+colInd];
        }

        /// get value at (rowInd, colInd) constant
        inline const value_t& operator()(size_t rowInd, size_t colInd) const
        {
            return elem(rowInd,colInd);
        }

        /// get value at (rowInd, colInd) for modification
        inline value_t& operator()(size_t rowInd, size_t colInd)
        {
            return elem(rowInd,colInd);
        }

        /// add a scalar s to every element of the matrix
        inline void addScalar(value_t s)
        {
            for (size_t i = 0; i < val.size(); ++i)
            {
                val[i] += s;
            }
        }

        /// get the number of rows
        inline size_t rows() const
        {
            return nrows;
        }

        /// get the number of columns
        inline size_t cols() const
        {
            return ncols;
        }

        /// scale all elements by a factor of f
        inline void scale(value_t f)
        {
            for (size_t i = 0; i < val.size(); ++i)
            {
                val[i] *= f;
            }
        }

        /// scale all elements in column col by a factor of f
        inline void scaleColumn(size_t col, value_t f)
        {
```

```
            for (size_t i = 0; i<nrows; ++i)
            {
                elem(i,col) *= f;
            }
        }

        /// truncate values above a certain threshold
        inline void truncateValuesAbove(value_t v)
        {
            for (size_t i = 0; i < val.size(); ++i)
            {
                if(val[i] > v) {
                    val[i] = v;
                }
            }
        }

        /// search a given value v in column j and return the row index that
        /// corresponds to the first occurrence of v or the number of rows
        /// nothing has been found
        inline size_t searchInColumn(size_t j, value_t v) const
        {
            size_t result = nrows;
            for (size_t i = 0; i<nrows; ++i) {
                if(elem(i,j) == v) {
                    result = i;
                    break;
                }
            }
            return result;
        }


        ////////////////////////////////
        // Exceptions
        ////////////////////////////////

        /// \class ExSizeMismatch
        /// \brief This exception is thrown when the matrix has the wrong size
        class ExSizeMismatch { };
        /// \class ExFileOpenError
        /// \brief This exception is thrown when a file could not be opened
        class ExFileOpenError { };
        /// \class ExWrongInput
        /// \brief This exception is thrown when a wrong input is given
        class ExWrongInput { };

    private:
        std::valarray<value_t> val;          ///< internal representation of the ma
    trix
        size_t nrows;                        ///< number of rows
        size_t ncols;                        ///< number of columns

    };

    }
    #endif
```

```
#ifndef NULLSTREAM_H
#define NULLSTREAM_H

#include <ostream>

struct nullstream : std::ostream
{
    nullstream(): std::ios(0), std::ostream(0) {}
};

#endif
```

```
// objective_hfs.h: v2.4, 2009−05−21, Yves Salathe
#ifndef __OBJECTIVE_HFS_H__
#define __OBJECTIVE_HFS_H__

#include <boost/random/mersenne_twister.hpp>
#include "matrix.h"
#include "grid.h"

namespace simulation {

class ObjectiveFunctionHFS {
public:
    typedef double real_t;
    typedef unsigned int size_t;

    static void initialize();
    static double eval( real_t const *x, size_t id, size_t seed );
private:
//    static size_t nfe; // number of function evaluations
//    static boost::mt19937 rng; // random number generator
    static math::Matrix<std::string> indata_decel;
    static math::Matrix<real_t> zee;
    static math::Grid2d<real_t> stageBz;
    static math::Grid2d<real_t> stageBr;
    static real_t gamma;
    static real_t delta;
    static real_t epsilon;
    static real_t lowerInitialVelZ;
    static real_t lowerPosZ;
    static real_t upperVelZ;
    static real_t desiredVel;
    static size_t nClosest;
};

}
#endif
```

```
// objective_trap.h: v2.4, 2009-05-21, Yves Salathe
#ifndef __OBJECTIVE_TRAP_H__
#define __OBJECTIVE_TRAP_H__

#include <boost/random/mersenne_twister.hpp>
#include "matrix.h"
#include "grid.h"
#include "input.h"
#include <vector>

namespace simulation {

class ObjectiveFunctionTrap {
public:
    typedef double real_t;
    typedef unsigned int size_t;

    static void initialize();
    static double eval( real_t const *x, size_t id, size_t seed );
private:
//    static size_t nfe; // number of function evaluations
//    static boost::mt19937 rng; // random number generator
    static InputData indata_decel;
    static InputData indata_trap;
    static math::Matrix<real_t> zee;
    static math::Grid2d<real_t> stageBz;
    static math::Grid2d<real_t> stageBr;
    static math::Grid2d<real_t> towersBz;
    static math::Grid2d<real_t> towersBr;
    static math::Matrix<real_t> BFieldDecelAxRad;
    static math::Matrix<real_t> BFieldTowersAxRad;
    static math::Matrix<real_t> BFieldTrapAxRad;
    static math::Matrix<real_t> particles;
    static std::vector< math::Matrix<real_t> > origStagePulses;
    static math::Matrix<real_t> origTowerPulses;
    static int readParticlesFromFile;
    static int adjustTowerPulses;
    static real_t firstPulseStart;
    static real_t decelPulseOverlap;
    static real_t decelToTrapPulseOverlap;
    static real_t trapPulseOverlap;
    static real_t minDecelPulseLength;
    static real_t towerSwitchOnOverlap;
    static real_t frontTrapCoilFirstPulseScale;
    static real_t frontTrapCoilSecondPulseScale;
    static real_t rearTrapCoilPulseScale;
    static real_t gamma;
    static real_t delta;
    static int outputTOF;
    static int outputTrapRegion;
    static int outputTrappedParts;
};

}
#endif
```

```cpp
// observer.h: v2.4, 2009-05-06, Yves Salathe
#ifndef __OBSERVER_H__
#define __OBSERVER_H__

#include <iostream>
#include <limits>

namespace simulation
{

///////////////////////////////
/// \class Observer
///
/// \brief this class implements the observer pattern and can be
///        used to obtain data from the simulation
///////////////////////////////
template<class observable_t>
class Observer
{
public:
    typedef unsigned int size_t;
    typedef double real_t;

    /// default constructor
    Observer(observable_t& _observable, std::ostream& ostr = std::cout)
            : observable(_observable),
              activateTime(0),
              deactivateTime(std::numeric_limits<real_t>::infinity()),
              inverseFrequency(1), outputStream(&ostr)
    {
    }

    /// normal copy constructor
    Observer(Observer<observable_t>& o)
            : observable(o.observable)
    {
              setActivateTime(o.getActivateTime());
              setDeactivateTime(o.getDeactivateTime());
              setInverseFrequency(o.getInverseFrequency());
              setOutputStream(o.getOutputStream());
    }

    /// copy constructor with the abilty to specify a new observable
    template<class observer_t>
    Observer(observer_t& o, observable_t& _observable)
            : observable(_observable)
    {
              setActivateTime(o.getActivateTime());
              setDeactivateTime(o.getDeactivateTime());
              setInverseFrequency(o.getInverseFrequency());
              setOutputStream(o.getOutputStream());
    }

    /// virtual destructor
    /// this makes sure that the destructors of the derived classes
    /// are called when their objects get "delete"ed through a
    /// refernece of this type
    virtual ~Observer()
    {
    }

    /// this pure virtual method is (hopefully) called by observables
    ///
    /// \param observable a reference to the observable
    /// \param time the time of the notification
    /// \param timestep the "resolution" of the time
    virtual void notify(real_t time, real_t timestep) = 0;

    ////////////////////////
    // getters
    ////////////////////////


    /// get the time when the observer should become active
    inline real_t getActivateTime() const
```

```cpp
    {
        return activateTime;
    }

    /// get the time when the observer should become inactive
    inline real_t getDeactivateTime() const
    {
        return deactivateTime;
    }

    /// get the inverseFrequency at which the observer should get called
    inline real_t getInverseFrequency() const
    {
        return inverseFrequency;
    }

    /// get the output stream associated to this observer
    inline std::ostream& getOutputStream()
    {
        return *outputStream;
    }

    ////////////////////////
    // setters
    ////////////////////////

    /// set the time when the observer should become active
    inline void setActivateTime(real_t t)
    {
        activateTime = t;
    }

    /// set the time when the observer should become inactive
    inline void setDeactivateTime(real_t t)
    {
        deactivateTime = t;
    }

    /// set the inverseFrequency at which the observer should get called
    inline void setInverseFrequency(real_t f)
    {
        inverseFrequency = f;
    }

    /// set the output stream associated to this observer
    inline void setOutputStream(std::ostream& ostr)
    {
        outputStream = &ostr;
    }

protected:
    /// a reference to the observed observable
    observable_t& observable;
    /// the time when the observer should become active
    real_t activateTime;
    /// the time when the observer should become inactive
    real_t deactivateTime;
    /// the inverseFrequency at which the observer should get called
    real_t inverseFrequency;

private:
    /// an output stream to which the observer should write
    /// (the constructor sets this to cout by default)
    std::ostream* outputStream;
};

}
#endif
```

```cpp
// particle.h: v2.4, 2009-05-06, Yves Salathe
#ifndef __PARTICLE_H__
#define __PARTICLE_H__

#include <cmath>
#include <iostream>
#include "vector3d.h"
#include "particle_population.h"

namespace simulation
{

///////////////////////////////
/// \class Particle
///
/// \brief simple data structure that represents particles
///////////////////////////////
class Particle
{
public:
    ///////////////////////////
    // types
    ///////////////////////////
    typedef unsigned int size_t;
    typedef double real_t;
    typedef math::CartesianCoord<real_t> coord_t;
    typedef math::Vector3d<real_t, coord_t> vector_t;

    /// constructor
    Particle(size_t i,
             vector_t _pos,
             vector_t _vel,
             size_t populationId)
            : id(i), pos(_pos), vel(_vel),
            initPos(_pos), initVel(_vel),
            popId(populationId),
            oldPosInitialized(false) { }

    /// constructor with specification of old position
    Particle(size_t i,
             vector_t _pos,
             vector_t _oldPos,
             vector_t _vel,
             size_t populationId)
            : id(i), pos(_pos), oldPos(_oldPos), vel(_vel),
            initPos(_pos), initVel(_vel),
            popId(populationId),
            oldPosInitialized(true) { }

    /// copy constructor
    Particle(const Particle& p)
            : id(p.id),
            pos(p.pos),
            oldPos(p.oldPos),
            vel(p.vel),
            initPos(p.initPos),
            initVel(p.initVel),
            popId(p.popId),
            oldPosInitialized(p.oldPosInitialized) { }

    /// assignment operator
    Particle& operator=(const Particle& p)
    {
        id = p.id;
        pos = p.pos;
        oldPos = p.oldPos;
        vel = p.vel;
        initPos = p.initPos;
        initVel = p.initVel;
        popId = p.popId;
        oldPosInitialized = p.oldPosInitialized;
        return *this;
    }

    /// identification number
```

```cpp
    size_t id;
    /// position coordinates
    vector_t pos;
    /// position coordinates at time t-dt
    vector_t oldPos;
    /// components of velocity
    vector_t vel;
    /// components of the acceleration
    vector_t accel;
    /// initial position
    vector_t initPos;
    /// initial velocity
    vector_t initVel;
    /// id of the population this particle belongs to
    size_t popId;
    /// determines wheter the old positions of the particle have been
    /// initialized
    bool oldPosInitialized;

    /// returns the distance from the axis
    inline real_t posR() const
    {
        return sqrt(pos.getX()*pos.getX()
                    +pos.getY()*pos.getY());
    }

    /// returns the absolute radial velocity of the particle
    inline real_t absVelR() const
    {
        return sqrt(vel.getX()*vel.getX()
                    +vel.getY()*vel.getY());
    }

    /// output information about the particle
    inline void outputInformation(std::ostream& ostr) const
    {
        ostr    <<pos.getX()<<" "
                <<pos.getY()<<" "
                <<pos.getZ()<<" "
                <<vel.getX()<<" "
                <<vel.getY()<<" "
                <<vel.getZ()<<" "
                <<posR()<<" "
                <<initPos.getX()<<" "
                <<initPos.getY()<<" "
                <<initPos.getZ()<<" "
                <<initVel.getX()<<" "
                <<initVel.getY()<<" "
                <<initVel.getZ()<<" "
                <<id<<" "
                <<popId;
    }

};

}
#endif
```

```
// particle_population.h: v2.4, 2009-05-06, Yves Salathe
#ifndef __PARTICLE_POPULATION_H__
#define __PARTICLE_POPULATION_H__

#include <cmath>
#include <iostream>
#include "vector3d.h"
#include "valgrad.h"
#include "matrix.h"
#include "assert.h"
#include "exceptions.h"

namespace simulation
{

///////////////////////////////
/// \class ParticlePopulation
///
/// \brief simple data structure that represents a population of
/// particles
///////////////////////////////
class ParticlePopulation
{
public:
    ///////////////////////////
    // types
    ///////////////////////////
    typedef unsigned int size_t;
    typedef double real_t;

    ///////////////////////////
    // public methods
    ///////////////////////////

    /// default constructor
    ParticlePopulation() { }

    /// constructor
    ParticlePopulation(const math::Matrix<real_t>& _zeemanEn,
                       real_t _massFactor = 1)
    {
        setZeemanEffect(_zeemanEn);
        setMassFactor(_massFactor);
        setLFS();
    }

    /// calculate the energy of the particle according to the
    /// absolute value of the B-field (absB) at it’s position
    inline real_t calculateZeemanEnergy(real_t absB) const
    {
        real_t a = absB/zdist;
        size_t zindex = (size_t)floor(a);
#ifdef __SAVE__
        if (absB < 0 || absB >=
                zeemanEn(zeemanEn.rows()-1,0))
        {
            a = zeemanEn(zeemanEn.rows()-1,0)/zdist;
            zindex = zeemanEn.rows()-2;
            std::cerr
                << "Warning: invalid B-field seen:\n"
                << "B = " << absB
                << std::endl;
        }
#endif
        // linear interpolation:
        real_t b = a-zindex;
        return b*zeemanEn(zindex+1,1) + (1.0-b)*zeemanEn(zindex,1);
    }

    /// calculate the first derivative of the energy of the
    /// particle with respect to the absolute value of the B-field
    /// (absB) at it’s position
    inline real_t calculateZeemanEnergyDeriv(real_t absB) const
    {
        size_t zindex = (size_t)floor(absB/zdist);
```

```
#ifdef __SAVE__
        if (absB < 0 || absB >=
                zeemanEn(zeemanEn.rows()-1,0))
        {
            zindex = zeemanEn.rows()-2;
            std::cerr
                << "Warning: invalid B-field seen:\n"
                << "B = " << absB
                << std::endl;
        }
#endif
        return (zeemanEn(zindex+1,1)-zeemanEn(zindex,1))/zdist;
    }

    /// sets the factor to scale mass of the particle
    inline void setMassFactor(real_t m)
    {
        massFactor = m;
        mass = massFactor*1.00782503*1.66053886e-27;
    }

    /// sets the tag that indicates whether the particles are
    /// low-field-seeking or not
    inline void setLFS(bool v = false)
    {
        lfs = v;
    }

    /// get the factor to scale mass of the particle
    inline real_t getMassFactor() const
    {
        return massFactor;
    }

    /// get the mass of the particle
    inline real_t getMass() const
    {
        return mass;
    }

    /// set the 1d-grid which stores the B-field dependence of energy
    ///
    /// \param z the matrix that stores the Zeeman-effect has to have
    /// equidistant values for the absolute values of the B-field
    /// in the first column with the according energy in the second
    /// column
    inline void setZeemanEffect(const math::Matrix<real_t>& z)
    {
        Assert<ExWrongInput>(z.rows()>=2);
        Assert<ExWrongInput>(z.cols()==2);
        zeemanEn = z;
        zdist = z(1,0)-z(0,0);
    }

    /// get the 1d-grid which stores the B-field dependence of energy
    inline const math::Matrix<real_t>& getZeemanEffect() const
    {
        return zeemanEn;
    }

    /// get the tag that indicates whether the particles are
    /// low-field-seeking or not
    inline bool isLFS() const
    {
        return lfs;
    }

    /// calculate the kinetic energy of a particle of this population
    /// at velocity v
    inline real_t calculateKineticEnergy(const real_t v) const
    {
        // Note: there is a factor of 1e6 because we calculate
        // velocities in units of mm/us instead of m/s
        return .5*mass*v*v*1e6;
    }
```

```
private:
    /// 1d−grid which stores the B−field dependence of energy
    math::Matrix<real_t> zeemanEn;
    /// mesh−width of the 1d−grid "zeemanEffect"
    real_t zdist;
    /// scales mass of the particles
    real_t massFactor;
    /// mass of the atoms in kg
    real_t mass;
    /// a tag that indicates whether the particles are low−field−seeking
    /// or not
    bool lfs;
};

}
#endif
```

```
// particlesim_builder.h: v2.4, 2009−05−21, Yves Salathe
#ifndef __PARTICLESIM_BUILDER__
#define __PARTICLESIM_BUILDER__

#include "particlesim.h"
#include "observer.h"
#include "matrix.h"
#include "grid.h"
#include "exceptions.h"
#include <fstream>

namespace simulation {

template<class engine_t>
class ParticleSimBuilder {
public:

    typedef typename ParticleSim<engine_t>::real_t real_t;
    typedef typename ParticleSim<engine_t>::size_t size_t;
    typedef typename std::list<std::ofstream* > ofstream_ptr_list_t;
    typedef typename ParticleSim<engine_t>::observer_ptr_list_t observer_ptr_lis
t_t;

    ParticleSim<engine_t>* simulation;

    ParticleSimBuilder()
    {
        simulation = new ParticleSim<engine_t>();
    }

    ParticleSimBuilder(const ParticleSim<engine_t>& decelSim)
    {
        simulation = new ParticleSim<engine_t>(decelSim);
    }

    virtual ~ParticleSimBuilder()
    {
        // delete the simulation
        delete simulation;
        // delete all observers
        for (typename observer_ptr_list_t::iterator i
                = observers.begin();
                i != observers.end(); ++i)
        {
            delete *i;
        }
        // close and delete all output streams
        for (typename ofstream_ptr_list_t::iterator i
                = outputFileStreams.begin();
                i != outputFileStreams.end(); ++i)
        {
            (*i)−>close();
            delete *i;
        }
    }

    template<class observer_t>
    observer_t* addObserver(const char output_filename[])
    {
        std::ofstream* observerOutput = new std::ofstream(output_filename);
        outputFileStreams.push_back(observerOutput);
        observer_t* observer = addObserver<observer_t>(*observerOutput);
        return observer;
    }

    template<class observer_t>
    observer_t* addObserver(std::ostream& outputStream)
    {
        observer_t* observer = new observer_t(*simulation, outputStream);
        observers.push_back(observer);
        simulation−>addObserver(observer);
        return observer;
    }

protected:
```

```

    ofstream_ptr_list_t outputFileStreams;
    observer_ptr_list_t observers;


};

}

#endif
```

```cpp
// particlesim.h: v2.4, 2009-05-06, Yves Salathe
#ifndef __PARTICLESIM_H__
#define __PARTICLESIM_H__

#include "particle.h"
#include "particle_population.h"
#include "observer.h"
#include "matrix.h"
#include "grid.h"
#include "vector3d.h"
#include "basic_math.h"
#include "exceptions.h"
#include <iostream>
#include <list>
#include <vector>
#include <algorithm>
#include <cmath>
#include <limits>
//#include <nag.h>
//#include <nagg05.h>
#include <boost/random/variate_generator.hpp>
#include <boost/random/mersenne_twister.hpp>
#include <boost/random/uniform_real.hpp>
#include <boost/random/normal_distribution.hpp>

namespace simulation
{

///////////////////////////////
/// \class ParticleSim
///
/// \brief This is a class which provides the methods for the
/// simulation of the motion of particles.
///
/// The class also provides a way of obtaining data through
/// observers
///
///////////////////////////////
template<class engine_t>
class ParticleSim
{
public:
    ///////////////////////////////
    // types and datastructures
    ///////////////////////////////

    typedef unsigned int size_t;
    typedef double real_t;
    // use the same coordinate type as the particles
    typedef Particle::coord_t coord_t;
    typedef std::list<Particle> particle_list_t;
    typedef std::vector<ParticlePopulation> population_list_t;
    typedef std::list<Observer<ParticleSim<engine_t> >* >
            observer_ptr_list_t;

    ///////////////////////////////
    // public fields
    ///////////////////////////////
    engine_t engine;

    ///////////////////////////////
    //constructors
    ///////////////////////////////

    /// constructor
    ///
    /// \param e The engine defines the context of the simulation
    ///         (see public variable engine).
    ParticleSim() : pi(atan2(0,-1))
    {
        // set default values
        setTimestep();
        setTime();
        setEndTime();
        setStartPosX();
```

```cpp
        setStartPosY();
        setStartPosZ();
        setInitBeamRadius();
        setInitBeamLength();
        setVx();
        setVy();
        setVz();
        setTz();
        setTxy();
        setInitialRadialPositionOnDisk();
        setRemoveUnacceptedLFS();
    }

    /// copy constructor which copies the parameters from a
    /// particle simulation with a different type of engine
    ///
    /// \param p        The reference to an object of
    ///         ParticleSim<engine2_t> from which
    ///         the parameters should be copied
    template<class engine2_t>
    ParticleSim(const ParticleSim<engine2_t>& p)
            : pi(atan2(0,-1))
    {
        setTimestep(p.getTimestep());
        setTime(p.getTime());
        setEndTime(p.getEndTime());
        setStartPosX(p.getStartPosX());
        setStartPosY(p.getStartPosY());
        setStartPosZ(p.getStartPosZ());
        setInitBeamRadius(p.getInitBeamRadius());
        setInitBeamLength(p.getInitBeamLength());
        setVx(p.getVx());
        setVy(p.getVy());
        setVz(p.getVz());
        setTz(p.getTz());
        setTxy(p.getTxy());
        setInitialRadialPositionOnDisk(p.isInitialRadialPositionOnDisk());
        setRemoveUnacceptedLFS(p.getRemoveUnacceptedLFS());
        setPopulations(p.getPopulations());
        setParticles(p.getParticles());
    }

    ~ParticleSim() {
    }

    ///////////////////////////////
    // the main simulation
    ///////////////////////////////

    /// the actual simulation is implemented here
    ///

    ///
    /// \param integrator       The integrator defines how to integrate the
    ///                 equations of motion.
    template<class integrator_t>
    void runSimulation(integrator_t integrator)
    {

        // tell the engine to prepare the timestepping
        // (insert active coils etc.)
        engine.prepareTimestepping(time,timestep);
        // see wheter there is some magnetic field
        if (!engine.isActive() && !particles.empty()) {
            // determine the number of timesteps until the next event
            size_t n = timestepsUntilNextEvent();
            // integrate over a big timestep as long as nothing happens
            // (particles have no acceleration)
            moveWithoutAccel(n*timestep);
        }
        // start the main loop
        while (time <= endtime && !particles.empty())
        {
            // tell the engine to prepare the timestepping
            // (insert active coils etc.)
            engine.prepareTimestepping(time,timestep);
            // check if particles have left the boundaries
            removeParticlesOutsideBoundaries();
            // perform measurements
            measurements();
            // determine the number of timesteps until the next
            // event
            size_t n = timestepsUntilNextEvent();
            // initialize the integrator
            integrator.initialize(*this,time,timestep);
            // loop over the timesteps until the next event
            for (size_t i = 0; i<n; ++i)
            {
                // integrate equation of motion for all
                // particles
                time = integrator.performTimestep(*this, time, timestep);
            }
            // tell the integrator that he should update us
            integrator.updateSimulation(*this);
        }
        // check if particles have left the boundaries
        removeParticlesOutsideBoundaries();
        // also perform some measurements at the end of the simulation
        measurements();
    }

    ///////////////////////////////
    //setters
    ///////////////////////////////

    /// sets the numerical timesteps in us (should be 1−10ns)
    inline void setTimestep(real_t dt = .001)
    {
        timestep = dt;
    }

    /// sets the current time of the simulations in us
    inline void setTime(real_t t = 0)
    {
        time = t;
    }

    /// sets time to stop the simulation
    inline void setEndTime(real_t t = 2000)
    {
        endtime = t;
    }

    /// makes a copy of the referenced list of particles
    void setParticles(const particle_list_t& p)
    {
        particles = p;
    }

    /// sets starting position in x
    inline void setStartPosX(real_t v = 0)
    {
        startposx = v;
    }

    /// sets starting position in y
    inline void setStartPosY(real_t v = 0)
    {
        startposy = v;
    }

    /// sets starting position in z (possibility to set offset)
    inline void setStartPosZ(real_t v = 0)
    {
        startposz = v;
    }

    /// sets initial radius of the beam in the x−y−surface
    inline void setInitBeamRadius(real_t v = 0.05)
    {
```

```
        initbeamradius = v;
    }

    /// sets initial length of the beam along z−direction
    inline void setInitBeamLength(real_t v = 1.0)
    {
        initbeamlength = v;
    }

    /// sets starting velocity in x (sets maximum of gaussian velocity
    /// distribution in x)
    inline void setVx(real_t v = 0)
    {
        vx = v;
    }

    /// sets starting velociy in y  (sets maximum of gaussian velocity
    /// distribution in y)
    inline void setVy(real_t v = 0)
    {
        vy = v;
    }

    /// sets the velocity of the sychron. particle in mm/us
    inline void setVz(real_t v = .35)
    {
        vz = v;
    }

    /// sets the relative (to synchronous particle)
    /// temperature in z direction
    inline void setTz(real_t T = 1)
    {
        Tz = T;
    }

    /// sets the relative temperature in x,y direction
    inline void setTxy(real_t T = 0.005)
    {
        Txy = T;
    }

    /// dermine wheter to choose the radial initial position on a disk
    /// (true) or otherwise on a square (false)
    inline void setInitialRadialPositionOnDisk(bool v = true)
    {
        initialRadialPositionOnDisk = v;
    }

    /// determine whether the low−field−seeking particles that are
    /// not accepted will be removed
    /// Note: this functionality requires that the tag that indicates
    /// whether the particles are low−field−seeking or not is set
    /// correctly in the particle populations
    inline void setRemoveUnacceptedLFS(bool v = false)
    {
        removeUnacceptedLFS = v;
    }

    ///////////////////////////////
    //getters
    ///////////////////////////////

    /// get the numerical timesteps in us (should be 1−10ns)
    inline real_t getTimestep() const
    {
        return timestep;
    }

    /// get current time of the simulations in us
    inline real_t getTime() const
    {
        return time;
    }
```

```
        /// get time to stop the simulation
        inline real_t getEndTime() const
        {
            return endtime;
        }

        /// get the number of particles
        inline size_t getNParticles() const
        {
            return particles.size();
        }

        /// get a constant reference to the list of particles
        inline const particle_list_t& getParticles() const
        {
            return particles;
        }

        /// get a modifyable reference to the list of particles
        inline particle_list_t& getParticles()
        {
            return particles;
        }

        /// starting position in x
        inline real_t getStartPosX() const
        {
            return startposx;
        }

        /// starting position in y
        inline real_t getStartPosY() const
        {
            return startposy;
        }

        /// starting position in z (possibility to set offset)
        inline real_t getStartPosZ() const
        {
            return startposz;
        }

        /// initial radius of the beam in the x−y−surface
        inline real_t getInitBeamRadius() const
        {
            return initbeamradius;
        }

        /// initial length of the beam along z−direction
        inline real_t getInitBeamLength() const
        {
            return initbeamlength;
        }

        /// starting velocity in x (sets maximum of gaussian velocity
        /// distribution in x)
        inline real_t getVx() const
        {
            return vx;
        }

        /// starting velociy in y  (sets maximum of gaussian velocity
        /// distribution in y)
        inline real_t getVy() const
        {
            return vy;
        }

        /// get the velocity of the sychron. particle in mm/us
        inline real_t getVz() const
        {
            return vz;
        }

        /// get the relative (to synchronous particel resp. vz)
```

```
        /// temperature in z direction
        inline real_t getTz() const
        {
            return Tz;
        }

        /// get the relative temperature in x,y direction
        inline real_t getTxy() const
        {
            return Txy;
        }

        /// determine whether to choose the radial initial position on a disk
        /// (true) or otherwise on a square (false)
        inline bool isInitialRadialPositionOnDisk() const
        {
            return initialRadialPositionOnDisk;
        }

        /// determine whether the low−field−seeking particles that are
        /// not accepted will be removed
        inline bool getRemoveUnacceptedLFS() const
        {
            return removeUnacceptedLFS;
        }

        ///////////////////////////////
        //other public methods
        ///////////////////////////////

        /// add a new population
        ///
        /// \return the id of the population
        inline size_t addPopulation(const ParticlePopulation& p)
        {
            populations.push_back(p);
            return populations.size()−1;
        }

        /// make a copy of a vector of populations
        inline void setPopulations(const population_list_t&
                _populations)
        {
            populations = _populations;
        }

        /// get a vector of the populations
        inline const population_list_t& getPopulations() const
        {
            return populations;
        }

        /// adds a particle with a certain position and velocity
        /// (in cartesian coordinates) to the simulation
        inline void addParticle(size_t id,
                                math::Vector3d<real_t,coord_t> pos,
                                math::Vector3d<real_t,coord_t> vel,
                                size_t populationId)
        {
            particles.push_back(Particle(id,pos,vel,populationId));
        }

        /// This method generates particles from a matrix.
        ///
        /// \param p                The matrix has one line per particle where:
        ///                         1. column: x−coordinate of initial position
        ///                         2. column: y−coordinate of initial position
        ///                         3. column: z−coordinate of initial position
        ///                         4. column: x−coordinate of initial velocity
        ///                         5. column: y−coordinate of initial velocity
        ///                         6. column: z−coordinate of initial velocity
        ///                         The other columns are ignored.
        ///
        /// \param populationId     the population to which the particles
        ///                         belong
        void generateParticlesFromMatrix(const math::Matrix<real_t>& p,
                                         size_t populationId)
        {
            for (size_t i = 0; i < p.rows(); ++i)
            {
                math::Vector3d<real_t,coord_t> pos(p(i,0),p(i,1),p(i,2));
                math::Vector3d<real_t,coord_t> vel(p(i,3),p(i,4),p(i,5));
                addParticle(i,pos,vel,populationId);
            }
        }


        /// This method generates numbofpart particles with random
        /// positions and velocities.
        ///
        /// The positions are uniformly distributed in a given cylindrical
        /// volume.
        /// The velocities are gaussian distributed where the variance
        /// depends on the temperature.
        ///
        /// \param numbofpart       the number of particles
        /// \param population       the population to which the particles
        ///                         belong
        /// This method generates numbofpart particles with random
        /// positions and velocities.
        void generateRandomParticles(size_t numbofpart,
                                     size_t populationId)
        {
            boost::mt19937 rng;
            generateRandomParticles(numbofpart,populationId,rng);
        }


        /// This method generates numbofpart particles with random
        /// positions and velocities. It has the ability to specify the random
        /// number generator.
        ///
        /// The positions are uniformly distributed in a given cylindrical
        /// volume.
        /// The velocities are gaussian distributed where the variance
        /// depends on the temperature.
        ///
        /// \param numbofpart       the number of particles
        /// \param population       the population to which the particles
        ///                         belong
        /// \param rng              a reference to the random number
        ///                         generator (see boost library)
        template <class rng_t>
        void generateRandomParticles(size_t numbofpart,
                                     size_t populationId,
                                     rng_t& rng,
                                     real_t lowerBoundVelZ = 0)
        {
            boost::uniform_real<real_t> uniform01(0,1);
            boost::normal_distribution<real_t> distVelX(vx*1000,
                    sigvxy(populations[populationId].getMass()));
            boost::normal_distribution<real_t> distVelY(vy*1000,
                    sigvxy(populations[populationId].getMass()));
            boost::normal_distribution<real_t> distVelZ(vz*1000,
                    sigvz(populations[populationId].getMass()));
            boost::variate_generator<boost::mt19937&,
            boost::uniform_real<real_t> >
            randU(rng, uniform01);
            boost::variate_generator<boost::mt19937&,
            boost::normal_distribution<real_t> >
            randVelX(rng,distVelX),
            randVelY(rng,distVelY),
            randVelZ(rng,distVelZ);
            size_t last_id = 0;
            if (!particles.empty()) {
                last_id = particles.back().id;
            }
            for (size_t i = 0; i < numbofpart; ++i)
            {
                math::Vector3d<real_t,coord_t> pos;
```

```
                if (initialRadialPositionOnDisk)
                {
                    real_t pos_r = sqrt(randU())*initbeamradius;
                    real_t pos_phi = randU()*2*pi;
                    pos.setX(pos_r*cos(pos_phi)+startposx);
                    pos.setY(pos_r*sin(pos_phi)+startposy);
                }
                else
                {
                    pos.setX(randU()*initbeamradius);
                    pos.setY(randU()*initbeamradius);
                }
                pos.setZ(randU()*initbeamlength−initbeamlength/2+startposz);
                real_t velZ;
                while ((velZ = randVelZ()*0.001) < lowerBoundVelZ) { };
                math::Vector3d<real_t, coord_t> vel(randVelX()*0.001,
                    randVelY()*0.001,velZ);
                addParticle(i,pos,vel,populationId);
            }
        }

        /// register a new observer
        ///
        /// the observer will be called by the simulation according to
        /// activateTime, deactivateTime and Frequency set by the
        /// observer
        void addObserver(Observer<ParticleSim<engine_t> >* o)
        {
            observers.push_back(o);
        }

        /// unregister a new observer
        ///
        /// the observer will be called by the simulation according to
        /// activateTime, deactivateTime and Frequency set by the
        /// observer
        void removeObserver(Observer<ParticleSim<engine_t> >* o)
        {
            typename observer_ptr_list_t
            ::iterator i =
                std::find(observers.begin(),
                          observers.end(), o);
            observers.erase(i);
        }

        /// get a list of the observers
        const observer_ptr_list_t& getObservers() const
        {
            return observers;
        }


        /// calculate the mean absolute velocity of the particles
        real_t calculateMeanAbsVelocity() const
        {
            real_t v = 0.0;
            for (particle_list_t::const_iterator
                    p = particles.begin();
                    p != particles.end(); ++p)
            {
                v += (*p).vel.norm2();
            }
            return v/particles.size();
        }

        /// calculate the mean absolute radial velocity of the particles
        real_t calculateMeanAbsRadialVelocity() const
        {
            real_t v = 0.0;
            for (particle_list_t::const_iterator
                    p = particles.begin();
                    p != particles.end(); ++p)
            {
                v += (*p).absVelR();
            }
        return v/particles.size();
    }

    /// calculate the mean velocity in z−direction of the particles
    real_t calculateMeanVelocityZ() const
    {
        real_t v = 0.0;
        for (particle_list_t::const_iterator
                p = particles.begin();
                p != particles.end(); ++p)
        {
            v += (*p).vel.getZ();
        }
        return v/particles.size();
    }

    /// calculate the maximum absolute velocity of the particles
    real_t calculateMaxAbsVelocity() const
    {
        real_t v = 0.0;
        for (particle_list_t::const_iterator
                p = particles.begin();
                p != particles.end(); ++p)
        {
            real_t av = (*p).vel.norm2();
            if (av > v) v = av;
        }
        return v;
    }

    /// calculate the maximum absolute radial velocity in z−direction of the par
ticles
    real_t calculateMaxAbsRadialVelocity() const
    {
        real_t v = 0.0;
        for (particle_list_t::const_iterator
                p = particles.begin();
                p != particles.end(); ++p)
        {
            real_t av = (*p).absVelR();
            if (av > v) v = av;
        }
        return v;
    }

    /// calculate the maximum absolute velocity in z−direction of the particles
    real_t calculateMaxAbsVelocityZ() const
    {
        real_t v = 0.0;
        for (particle_list_t::const_iterator
                p = particles.begin();
                p != particles.end(); ++p)
        {
            real_t av = fabs((*p).vel.getZ());
            if (av > v) v = av;
        }
        return v;
    }

    /// calculate the minimum absolute velocity of the particles
    real_t calculateMinAbsVelocity() const
    {
        real_t v = std::numeric_limits<real_t>::infinity();
        for (particle_list_t::const_iterator
                p = particles.begin();
                p != particles.end(); ++p)
        {
            real_t av = (*p).vel.norm2();
            if (av < v) v = av;
        }
        return v;
    }

    /// calculate the minimum absolute radial velocity in z−direction
    /// of the particles
```

```
    real_t calculateMinAbsRadialVelocity() const
    {
        real_t v = std::numeric_limits<real_t>::infinity();
        for (particle_list_t::const_iterator
                p = particles.begin();
                p != particles.end(); ++p)
        {
            real_t av = (*p).absVelR();
            if (av < v) v = av;
        }
        return v;
    }

    /// calculate the minimum absolute velocity in z−direction of the particles
    real_t calculateMinAbsVelocityZ() const
    {
        real_t v = std::numeric_limits<real_t>::infinity();
        for (particle_list_t::const_iterator
                p = particles.begin();
                p != particles.end(); ++p)
        {
            real_t av = fabs((*p).vel.getZ());
            if (av < v) v = av;
        }
        return v;
    }

    /// output information about all particles
    void outputInformationAboutAllParticles(std::ostream& ostr)
    {
        for (particle_list_t::const_iterator
                p = particles.begin();
                p != particles.end(); ++p)
        {
            ostr <<0<<"\t"<<0<<"\t"<<0<<"\t"<<time<<"\t";
            p−>outputInformation(ostr);
            ostr <<std::endl;
        }
    }

    /// calculate the acceleration of n particles at a given time and
    /// with a given timestep
    ///
    /// \param t        time
    /// \param dt       time−step
    /// \param n        number of particles
    /// \param popId    array with n elements containing the population
    ///                 id of each particle
    /// \param pos      array with 3*n positions where each triplet belongs
    ///                 to one particle
    /// \param accel    array where to write the accelerations of each
    ///                 particle (same format as in pos)
    inline void calculateAccelerations(real_t t, real_t dt,
            size_t n, size_t popId[], real_t pos[], real_t accel[])
    {
        // tell the engine to prepare the magnetic
        // field for this timestep (e.g. scale
        // values etc.)
        engine.setTime(t,dt);
        for (size_t i = 0; i<n; ++i)
        {
            size_t i3 = 3*i;
            // calculate radial position of the particle
            real_t pos_r = sqrt(pos[i3]*pos[i3]+pos[i3+1]*pos[i3+1]);
            // calculate the field that the particle sees
            math::ValGrad<real_t> Bv = engine.calculateAbsBField(
                        pos[i3+2],pos_r);
            real_t dE_dB = populations[popId[i]]
                    .calculateZeemanEnergyDeriv(Bv.val);
            // calculate acceleration
            // Note: there is a factor of 1e−6 because we calculate
            // in units of us and mm instead of s and m respectively
            // longitudial acceleration:
            accel[i3+2] = −Bv.grad_dim1*dE_dB
                    /populations[popId[i]].getMass()*1e−6;
            // update radial velocity and position
            // Note that for the case where the radial position of
            // the particle is zero, we assume that the radial
            // acceleration is also zero.
            if (pos_r > std::numeric_limits<real_t>::epsilon())
            {
                // radial acceleration:
                real_t accelr = -Bv.grad_dim2*dE_dB
                        /populations[popId[i]].getMass()*1e-6;
                accel[i3] = accelr*pos[i3]/pos_r;
                accel[i3+1] = accelr*pos[i3+1]/pos_r;
            } else {
                accel[i3] = 0;
                accel[i3+1] = 0;
            }
        }
    }

    /// calculate the kinetic energy of the particle
    inline real_t calculateKineticEnergy(const Particle& p) const
    {
        real_t v = p.vel.norm2();
        // Note: there is a factor of 1e6 because we calculate
        // velocities in units of mm/us instead of m/s
        return .5*populations[p.popId].getMass()*v*v*1e6;
    }



    /// copy specific observers of class src_observer_t associated
    /// with a different simulation of type simulation_t from a list
    /// of observers to this simulation with new type target_observer_t
    ///
    /// members of the list which are not of type src_observer_t will
    /// not be copied
    template<class simulation_t, class src_observer_t,
            class target_observer_t>
    observer_ptr_list_t copyObserversFromList(
            const std::list<Observer<simulation_t>* > obs)
    {
        observer_ptr_list_t copiedObservers;
        for(typename std::list<Observer<simulation_t>* >::const_iterator o
                = obs.begin(); o != obs.end(); ++o) {
            if(src_observer_t* co
                    = dynamic_cast<src_observer_t* >(*o)) {
                Observer< ParticleSim<engine_t> > *newobs
                        = new target_observer_t(*co,*this);
                copiedObservers.push_back(newobs);
                addObserver(newobs);
            }
        }
        return copiedObservers;
    }

protected:
    ///////////////////////////////
    // definition of constants
    ///////////////////////////////
    ///<Boltzmannkonst in SI-units
    static const real_t kk = 1.3806504e-23;
    const real_t pi;                      ///<the famous ratio (calculated in th
e constructor)

    ///////////////////////////////
    //definition of variables (properties of the class)
    ///////////////////////////////

    /// list of particles
    particle_list_t particles;
    /// numerical timesteps in us (should be 1-10ns)
    real_t timestep;
    /// the actual time of the simulations in us
    real_t time;
    /// sets time to stop the simulation
    real_t endtime;
```

```
    /// starting position in x-direction
    real_t startposx;
    /// starting position in y-direction
    real_t startposy;
    /// starting position in z-direction (possibility to set offset)
    real_t startposz;
    /// initial radius of the beam in the x-y-surface
    real_t initbeamradius;
    /// initial length of the beam along z-direction
    real_t initbeamlength;
    /// starting velocity in x (sets maximum of gaussian velocity
    /// distribution in x)
    real_t vx;
    /// starting velociy in y  (sets maximum of gaussian velocity
    /// distribution in y)
    real_t vy;
    /// velocity of the sychron. particle in mm/us
    real_t vz;
    /// relative (to synchronous particel resp. vz)
    /// temperature in z direction
    real_t Tz;
    /// relative temperature in x,y direction
    real_t Txy;
    /// wheter to choose the radial initial position on a disk
    /// (true) or otherwise on a square (false)
    bool initialRadialPositionOnDisk;
    /// list of observers
    observer_ptr_list_t observers;
    /// list of particle-populations
    population_list_t populations;
    /// whether to remove the particles that are not accepted
    bool removeUnacceptedLFS;

    ///////////////////////////////
    //calculated properties
    ///////////////////////////////

    /// width of velocity distribution in z
    inline real_t sigvz(real_t mass) const
    {
        return sqrt(kk*Tz/mass);
    }

    /// width of velocity distribution in r
    inline real_t sigvxy(real_t mass) const
    {
        return sqrt(kk*Txy/mass);
    }

    ///////////////////////////////
    //protected methods
    ///////////////////////////////

    /// calculate the number of timesteps until the next event occurs
    inline size_t timestepsUntilNextEvent() const
    {
        return math::min(timestepsUntilNextMeasurement(),
                engine.timestepsUntilNextEvent(time,timestep,endtime));
    }

    /// calculate the number of timesteps until the next measurement
    /// returns ceil((endtime-time)/timestep) if no measurement will happen
    inline size_t timestepsUntilNextMeasurement() const
    {
        int min_timesteps = (int)ceil((endtime-time)/timestep);
        for (typename observer_ptr_list_t::const_iterator o
                = observers.begin();
                o != observers.end(); ++o)
        {
            if (time >= (**o).getActivateTime() && time <=
                    (**o).getDeactivateTime())
            {
                int ts = (int)ceil(((**o).getInverseFrequency()
                        -fmod(time-(**o).getActivateTime(),
                            (**o).getInverseFrequency()))/timestep);
                if (ts > 0 && ts < min_timesteps) min_timesteps = ts;
            }
        }
        return min_timesteps;
    }

    /// integerate equation of motion with no acceleration over a time
    /// difference of dt
    inline void moveWithoutAccel(real_t delta_t)
    {
        for (particle_list_t::iterator p
                = particles.begin();
                p != particles.end(); ++p) {
            p->pos = p->pos+p->vel*delta_t;
        }
        time = time + delta_t;
    }

    /// check wheter the particles are still within the boundaries
    /// defined by the engine and removes them if not
    inline void removeParticlesOutsideBoundaries()
    {
        for (particle_list_t::iterator
                p = particles.begin();
                p != particles.end(); ++p)
        {
            if (!engine.isPointInsideBoundaries(p->pos.getZ(),
                                               p->posR()))
            {
#ifdef __DEBUG__
                debugOutputAtParticleRemoval(*p);
#endif
                p = particles.erase(p);
            }
            else if (removeUnacceptedLFS) {
                // Note: this functionality requires that the tag
                // that indicates whether the particles are
                // low-field-seeking or not is set correctly
                // in the particle population
                const ParticlePopulation& pop = populations[p->popId];
                if (pop.isLFS()) {
                    if (!engine.mayLFSParticleBeAccepted(time,*p,pop)) {
#ifdef __DEBUG__
                        debugOutputAtParticleRemoval(*p);
#endif
                        p = particles.erase(p);
                    }
                }
            }
        }
    }

    /// perform some measurements (detection of the particles using a
    /// laser, particle trajectories, engine specific measurements)
    ///
    /// each of these measurements can be activated or deactivated using
    /// compiler options
    inline void measurements()
    {
        /// tell the engine the time
        engine.setTime(time,timestep);
        for (typename observer_ptr_list_t::iterator o
                = observers.begin();
                o != observers.end(); ++o)
        {
            if (time > (**o).getActivateTime()
                    && time < (**o).getDeactivateTime()
                    && fmod(time-(**o).getActivateTime(),
                            (**o).getInverseFrequency())<timestep)
            {
                (**o).notify(time,timestep);
            }
        }
    }
```

```
        /// if wanted, output some debugging information when a particle
        /// reaches the system boundaries and therefore gets removed
        void inline debugOutputAtParticleRemoval(const Particle& p) const
        {
            std::cerr << "-- particle " << p.id
                      << " will be removed at t = "
                      << time << " us" << ":\n";
            p.outputInformation(std::cerr);
            std::cerr << std::endl;
        }

};

}
#endif
```

```
// polynomial.h: 2009-04-07, Yves Salathe
#ifndef __POLYNOMIAL_H__
#define __POLYNOMIAL_H__

#include <valarray>

namespace math
{

double evalPoly(double t, const std::valarray<double>& coeff);

/// evaluate a polynomial p(t) in standard form with given
/// coefficients
double evalPoly(double t, const std::valarray<double>& coeff)
{
    int n = coeff.size();                    // order of the polynomial
    double res = coeff[n-1];
    for (int i = n-2; i>=0; --i)
    {
        res = t*res + coeff[i];
    }
    return res;
}

}
#endif
```

```
// region_observer.h: v2.4, 2009-05-06, Yves Salathe
#ifndef __REGION_OBSERVER_H__
#define __REGION_OBSERVER_H__

#include "observer.h"
#include "particle.h"
#include "valgrad.h"
#include "vector3d.h"
#include <limits>
#include <cmath>

namespace simulation
{

///////////////////////////////
/// \class RegionObserver
///
/// \brief      The number of particles inside an ellipsoid and the mean
///             total energy of these particles are the output of this
///             observer.
///
///             Additionally an upper bound for the total energy for a
///             particle to be considered being inside the Region can be
///             specified according to an absolute value of the magnetic
///             field which corresponds to that energy according to the
///             Zeeman-effect of the particle using the method
///             setUpperTotalEnergyAccordingToAbsB()
///
///////////////////////////////
template <class observable_t>
class RegionObserver : public Observer<observable_t>
{
public:
    typedef typename Observer<observable_t>::real_t real_t;
    typedef typename Observer<observable_t>::size_t size_t;
    typedef typename observable_t::coord_t coord_t;
    typedef typename observable_t::particle_list_t particle_list_t;

    using Observer<observable_t>::observable;
    using Observer<observable_t>::activateTime;
    using Observer<observable_t>::deactivateTime;
    using Observer<observable_t>::inverseFrequency;
    using Observer<observable_t>::getOutputStream;

    /// default constructor
    RegionObserver(observable_t& _observable,
            std::ostream& ostr = std::cout)
            : Observer<observable_t>(_observable, ostr)
    {
    }

    /// normal copy constructor
    RegionObserver(RegionObserver<observable_t>& o)
            : Observer<observable_t>(o)
    {
        setRadiusX2(getRadiusX2());
        setRadiusY2(getRadiusY2());
        setRadiusX2(getRadiusZ2());
    }

    /// copy constructor with the abilty to specify a new observable
    template<class observer_t>
    RegionObserver(observer_t& o, observable_t& _observable)
            : Observer<observable_t>(o, _observable)
    {
    }

    /// virtual destructor
    /// this makes sure that the destructors of the derived classes
    /// are called when their objects get "delete"ed through a
    /// refernece of this type
    virtual ~RegionObserver()
    {
    }

```

```
    /// this virtual method is (hopefully) called by observables
    ///
    /// \param time the time of the notification
    /// \param timestep the "resolution" of the time
    virtual void notify(real_t time, real_t timestep)
    {
        nParticles = 0;
        real_t sumTotalEnergy = 0;
        real_t sumEnergyRatio = 0;
        const particle_list_t& particles
                = observable.getParticles();
        for (typename particle_list_t::const_iterator
                p = particles.begin();
                p != particles.end(); ++p)
        {
            real_t dz = p->pos.getZ()-centerPosZ;
            if (p->pos.getX()*p->pos.getX()/radiusX2
                        + p->pos.getY()*p->pos.getY()/radiusY2
                        + dz*dz/radiusZ2 < 1)
            {
                real_t kineticEnergy = observable.
                        calculateKineticEnergy(*p);
                math::ValGrad<real_t> Bv = observable.engine.
                        calculateAbsBField(p->pos.getZ(),p->posR());
                real_t potentialEnergy =
                        observable.getPopulations()[p->popId]
                        .calculateZeemanEnergy(Bv.val);
                real_t totalEnergy = potentialEnergy+kineticEnergy;
                real_t upperTotalEnergy = observable
                        .getPopulations()[p->popId]
                        .calculateZeemanEnergy(upperB);
                if (totalEnergy<upperTotalEnergy) {
                    ++nParticles;
                    sumTotalEnergy += totalEnergy;
                    sumEnergyRatio += totalEnergy/upperTotalEnergy;
//                    getOutputStream() << time << " ";
//                    p->outputInformation(getOutputStream());
//                    getOutputStream() << " " << kineticEnergy
//                                      << " " << potentialEnergy
//                                      << " " << totalEnergy
//                                      << " " << upperTotalEnergy
//                                      << std::endl;
                }
            }
        }
        meanTotalEnergy = sumTotalEnergy/nParticles;
        meanEnergyRatio = sumEnergyRatio/nParticles;
        timeOfLastMeasurement = time;
        getOutputStream() << time << " " << nParticles << " "
                << meanTotalEnergy << " " << meanEnergyRatio
                << std::endl;
    }

    void outputTrappedParticles(std::ostream& ostr)
    {
        const particle_list_t& particles
                = observable.getParticles();
        for (typename particle_list_t::const_iterator
                p = particles.begin();
                p != particles.end(); ++p)
        {
            real_t dz = p->pos.getZ()-centerPosZ;
            if (p->pos.getX()*p->pos.getX()/radiusX2
                        + p->pos.getY()*p->pos.getY()/radiusY2
                        + dz*dz/radiusZ2 < 1)
            {
                real_t kineticEnergy = observable.
                        calculateKineticEnergy(*p);
                math::ValGrad<real_t> Bv = observable.engine.
                        calculateAbsBField(p->pos.getZ(),p->posR());
                real_t potentialEnergy =
                        observable.getPopulations()[p->popId]
                        .calculateZeemanEnergy(Bv.val);
                real_t totalEnergy = potentialEnergy+kineticEnergy;
                real_t upperTotalEnergy = observable
```

```
                        .getPopulations()[p->popId]
                        .calculateZeemanEnergy(upperB);
                if (totalEnergy<upperTotalEnergy) {
                    p->outputInformation(ostr);
                    ostr << " " << kineticEnergy
                                  << " " << potentialEnergy
                                  << " " << totalEnergy
                                  << " " << upperTotalEnergy
                                  << std::endl;
                }
            }
        }
    }

    /// calculate the mean ratio of the total energy to the particle’s
    /// upper limit given by the upper limit of the magnetic field for
    /// each particle located in the region (without upper bound for the
    /// energy)
    real_t calculateMeanEnergyRatioWithoutEnergyBound()
    {
        real_t sumRatio = 0.0;
        size_t n = 0;
        const particle_list_t& particles
                = observable.getParticles();
        for (typename particle_list_t::const_iterator
                p = particles.begin();
                p != particles.end(); ++p)
        {
            real_t dz = p->pos.getZ()-centerPosZ;
            if (p->pos.getX()*p->pos.getX()/radiusX2
                        + p->pos.getY()*p->pos.getY()/radiusY2
                        + dz*dz/radiusZ2 < 1)
            {
                real_t kineticEnergy = observable.
                        calculateKineticEnergy(*p);
                math::ValGrad<real_t> Bv = observable.engine.
                        calculateAbsBField(p->pos.getZ(),p->posR());
                real_t potentialEnergy =
                        observable.getPopulations()[p->popId]
                        .calculateZeemanEnergy(Bv.val);
                real_t totalEnergy = potentialEnergy+kineticEnergy;
                real_t upperTotalEnergy = observable
                        .getPopulations()[p->popId]
                        .calculateZeemanEnergy(upperB);
                sumRatio += totalEnergy/upperTotalEnergy;
                ++n;
            }
        }
        return sumRatio/n;
    }

    /// calculate the minimum ratio of the total energy to the particle’s
    /// upper limit given by the upper limit of the magnetic field for
    /// each particle located in the region (without upper bound for the
    /// energy)
    real_t calculateMinEnergyRatioWithoutEnergyBound()
    {
        real_t minRatio = std::numeric_limits<real_t>::infinity();
        const particle_list_t& particles
                = observable.getParticles();
        for (typename particle_list_t::const_iterator
                p = particles.begin();
                p != particles.end(); ++p)
        {
            real_t dz = p->pos.getZ()-centerPosZ;
            if (p->pos.getX()*p->pos.getX()/radiusX2
                        + p->pos.getY()*p->pos.getY()/radiusY2
                        + dz*dz/radiusZ2 < 1)
            {
                real_t kineticEnergy = observable.
                        calculateKineticEnergy(*p);
                math::ValGrad<real_t> Bv = observable.engine.
                        calculateAbsBField(p->pos.getZ(),p->posR());
                real_t potentialEnergy =
                        observable.getPopulations()[p->popId]
```

```
                        .calculateZeemanEnergy(Bv.val);
                real_t totalEnergy = potentialEnergy+kineticEnergy;
                real_t upperTotalEnergy = observable
                        .getPopulations()[p->popId]
                        .calculateZeemanEnergy(upperB);
                real_t ratio = totalEnergy/upperTotalEnergy;
                if(ratio < minRatio) minRatio = ratio;
            }
        }
        return minRatio;
    }

    /// calculate the number of particles located inside the region
    /// (without upper bound for the energy)
    size_t calculateNumberOfParticlesWithoutEnergyBound()
    {
        size_t n = 0;
        const particle_list_t& particles
                = observable.getParticles();
        for (typename particle_list_t::const_iterator
                p = particles.begin();
                p != particles.end(); ++p)
        {
            real_t dz = p->pos.getZ()-centerPosZ;
            if (p->pos.getX()*p->pos.getX()/radiusX2
                        + p->pos.getY()*p->pos.getY()/radiusY2
                        + dz*dz/radiusZ2 < 1)
            {
                ++n;
            }
        }
        return n;
    }

    /// calculate the mean ratio of the total energy to the particle’s
    /// upper limit given by the upper limit of the magnetic field for
    /// each particle in the simulation (not just those in the region)
    real_t calculateMeanEnergyRatioForAll()
    {
        real_t sumRatio = 0.0;
        const particle_list_t& particles
                = observable.getParticles();
        for (typename particle_list_t::const_iterator
                p = particles.begin();
                p != particles.end(); ++p)
        {
            real_t kineticEnergy = observable.
                    calculateKineticEnergy(*p);
            math::ValGrad<real_t> Bv = observable.engine.
                    calculateAbsBField(p->pos.getZ(),p->posR());
            real_t potentialEnergy =
                    observable.getPopulations()[p->popId]
                    .calculateZeemanEnergy(Bv.val);
            real_t totalEnergy = potentialEnergy+kineticEnergy;
            real_t upperTotalEnergy = observable
                    .getPopulations()[p->popId]
                    .calculateZeemanEnergy(upperB);
            sumRatio += totalEnergy/upperTotalEnergy;
        }
        return sumRatio/particles.size();
    }

    /// calculate the mean ratio between the distance of each particle to
    /// the center of this region in z-direction and the radius of this
    /// region in z-direction (for all particles in the simulation)
    real_t calculateMeanZDistanceRatioForAll()
    {
        real_t sumdz = 0.0;
        const particle_list_t& particles
                = observable.getParticles();
        for (typename particle_list_t::const_iterator
                p = particles.begin();
                p != particles.end(); ++p)
        {
            sumdz += fabs(p->pos.getZ()-centerPosZ);
        }
        real_t radiusZ = sqrt(radiusZ2);
        return sumdz/radiusZ/particles.size();
    }

    /// set the position of the center of the region
    void setCenterPosZ(real_t p) { centerPosZ = p; }

    /// sets the radius along the x−axis of the region (ellipsoid)
    inline void setRadiusX(real_t v)
    {
        setRadiusX2(v*v);
    }

    /// sets the radius along the y−axis of the region (ellipsoid)
    inline void setRadiusY(real_t v)
    {
        setRadiusY2(v*v);
    }

    /// sets the radius along the z−axis of the region (ellipsoid)
    inline void setRadiusZ(real_t v)
    {
        setRadiusZ2(v*v);
    }

    /// sets the radius along the x−axis of the region (ellipsoid)
    inline void setRadiusX2(real_t v)
    {
        radiusX2 = v;
    }

    /// sets the radius along the y−axis of the region (ellipsoid)
    inline void setRadiusY2(real_t v)
    {
        radiusY2 = v;
    }

    /// sets the radius along the z−axis of the region (ellipsoid)
    inline void setRadiusZ2(real_t v)
    {
        radiusZ2 = v;
    }

    /// Additionally an upper bound for the total energy for a
    /// particle to be considered being inside the Region can be
    /// specified according to an absolute value of the magnetic
    /// field which corresponds to that energy according to the
    /// Zeeman−effect of the particle using the method
    /// setUpperTotalEnergyAccordingToAbsB()
    inline void setUpperTotalEnergyAccordingToAbsB(real_t absB)
    {
        upperB = absB;
    }

    /// get the position of the center of the region
    inline real_t getCenterPosZ() const { return centerPosZ; }

    /// get the radius along the x−axis of the region (ellipsoid)
    inline real_t getRadiusX2() const
    {
        return radiusX2;
    }

    /// get the radius along the y−axis of the region (ellipsoid)
    inline real_t getRadiusY2() const
    {
        return radiusY2;
    }

    /// gets the radius along the z−axis of the region (ellipsoid)
    inline real_t getRadiusZ2() const
    {
        return radiusZ2;
    }
```

```

    /// Additionally an upper bound for the total energy for a
    /// particle to be considered being inside the Region can be
    /// specified according to an absolute value of the magnetic
    /// field which corresponds to that energy according to the
    /// Zeeman−effect of the particle using the method
    /// setUpperTotalEnergyAccordingToAbsB()
    inline real_t getUpperTotalEnergyAccordingToAbsB() const
    {
        return upperB;
    }

    /// get the  number of detected particles
    inline size_t getNParticles() const
    { return nParticles; }

    /// get the mean of total energy
    inline real_t getMeanTotalEnergy() const
    { return meanTotalEnergy; }

    /// get the mean of the ratio between the energy of the
    /// particles inside the region and their upper bound
    /// specified by setUpperTotalEnergyAccordingToAbsB()
    inline real_t getMeanEnergyRatio() const
    { return meanEnergyRatio; }

    /// get the time of the last measurement
    inline real_t getTimeOfLastMeasurement() const
    { return timeOfLastMeasurement; }

private:
    /// the position of the center of the region along the Z−axis
    real_t centerPosZ;
    /// radius along the x−axis of the region (ellipsoid) squared
    real_t radiusX2;
    /// radius along the y−axis of the region (ellipsoid) squared
    real_t radiusY2;
    /// radius along the z−axis of the region (ellipsoid) squared
    real_t radiusZ2;
    /// Additionally an upper bound for the total energy for a
    /// particle to be considered being inside the Region can be
    /// specified according to an absolute value of the magnetic
    /// field which corresponds to that energy according to the
    /// Zeeman−effect of the particle using the method
    /// setUpperTotalEnergyAccordingToAbsB()
    real_t upperB;
    /// number of detected particles
    size_t nParticles;
    /// mean of total energy
    real_t meanTotalEnergy;
    /// mean of the ratio between the energy of the
    /// particles inside the region and their upper bound
    /// specified by setUpperTotalEnergyAccordingToAbsB()
    real_t meanEnergyRatio;
    /// the time of the last measurement
    real_t timeOfLastMeasurement;
};

}
#endif
```

```cpp
// symplectic_euler_integrator.h: v2.42, 2009−05−07, Yves Salathe
#ifndef __SYMPLECTIC_EULER_INTEGRATOR_H__
#define __SYMPLECTIC_EULER_INTEGRATOR_H__

#include "integrator.h"
#include "particle.h"
#include "matrix.h"
#include "vector3d.h"
#include "assert.h"
#include "exceptions.h"

namespace simulation
{

////////////////////////////////////
/// \class SymplecticEulerIntegrator
///
/// \brief This class describes a method to integrate newton’s equations
///        of motion. It can be used as an integrator in the class
///        of type simulation_t (usually ParticleSim)
////////////////////////////////////
template<class simulation_t>
class SymplecticEulerIntegrator : public Integrator<simulation_t>
{
public:
    ////////////////////////////////////
    // type definitions
    ////////////////////////////////////

    //use the same types as the simulation
    typedef typename simulation_t::real_t real_t;
    typedef typename simulation_t::size_t size_t;
    typedef typename simulation_t::coord_t coord_t;
    typedef typename simulation_t::particle_list_t particle_list_t;
    typedef typename simulation_t::population_list_t population_list_t;

    /// constructor
    SymplecticEulerIntegrator() : nParticles(0)
    {
    }

    /// destructor
    ~SymplecticEulerIntegrator()
    {
        if(nParticles > 0) {
            delete[] id;
            delete[] popId;
            delete[] pos;
            delete[] vel;
            delete[] accel;
        }
    }

    /// initialize the integrator with the current state of the
    /// simulation
    ///
    /// \param simulation       a reference to the simulation
    /// \param time             the start time of the step
    /// \param timestep         the size of the step (time difference)
    void initialize(simulation_t& simulation, real_t time,
            real_t timestep)
    {
        // free previously allocated memory
        if(nParticles > 0) {
            delete[] id;
            delete[] popId;
            delete[] pos;
            delete[] vel;
            delete[] accel;
        }
        // get a reference to the list of particles
        const particle_list_t& particles = simulation.getParticles();
        nParticles = particles.size();
        // allocate memory
        id = new size_t[nParticles];
```

```cpp
        popId = new size_t[nParticles];
        size_t n3 = 3*nParticles;
        pos = new real_t[n3];
        vel = new real_t[n3];
        accel = new real_t[n3];
        // initialize masses and positions
        typename particle_list_t::const_iterator p = particles.begin();
        for(size_t i = 0; i<nParticles; ++i) {
            size_t i3 = 3*i;
            id[i] = p−>id;
            popId[i] = p−>popId;
            p−>pos.writeToArray(&pos[i3]);
            p−>vel.writeToArray(&vel[i3]);
            ++p;
        }
    }

    /// timestepping: perform a timestep of dt
    /// (integrate equation of motion for all particles)
    ///
    /// \param simulation       a reference to the simulation
    /// \param time             the start time of the step
    /// \param timestep         the size of the step (time difference)
    /// \return the new reference time
    inline real_t performTimestep(simulation_t& simulation, real_t t,
            real_t dt)
    {
        // calculate the acceleration of each particle at that time
        simulation.calculateAccelerations(t,dt,nParticles, popId, pos, accel);
        // go through the list of particles and integrate the
        // equations of motion for each of them separately
        size_t n3 = 3*nParticles;
        for (size_t i = 0; i<n3; ++i)
        {
            vel[i] += dt*accel[i];
            pos[i] += dt*vel[i];
        }
        return t+dt;
    }

    /// store the computed particle information back to the simulation
    void updateSimulation(simulation_t& simulation)
    {
        // get a reference to the list of particles
        particle_list_t& particles = simulation.getParticles();
        typename particle_list_t::iterator p = particles.begin();
        for(size_t i = 0; i<nParticles; ++i) {
            while(p−>id != id[i] && p != particles.end()) {
                ++p;
            }
            if (p != particles.end()) {
                size_t i3 = 3*i;
                p−>pos.setFromArray(&pos[i3]);
                p−>vel.setFromArray(&vel[i3]);
            } else {
                throw ExSizeMismatch();
            }
        }
    }

private:

    /// the number of particles
    size_t nParticles;
    /// the ids of the particles
    size_t* id;
    /// an array containing the population id of each particle
    size_t* popId;
    /// an array containing the masses of each particle
    real_t* mass;
    /// an array of size 3*numbParticles containing the positions at
    /// time t of the particles in cartesian coordinates
    real_t* pos;
    /// an array of size 3*numbParticles containing the velocities
    /// at time t of the particles in cartesian coordinates
    real_t* vel;
    /// an array of size 3*numbParticles containing the accelerations of the
    /// particles in cartesian coordinates
    real_t* accel;

    /// initialize acceleration for all particles
    inline void initializeAccelerations(real_t v = 0) const
    {
        size_t n3 = 3*nParticles;

        for (size_t i = 0; i<n3; ++i) {
            accel[i] = 0;
        }
    }

};

}
#endif
```

```
// timestep_observer.h: v2.4, 2009-05-06, Yves Salathe
#ifndef __TIMESTEP_OBSERVER_H__
#define __TIMESTEP_OBSERVER_H__

#include "observer.h"
#include "particle.h"
#include "valgrad.h"
#include "vector3d.h"
#include <cmath>

namespace simulation
{

///////////////////////////////
/// \class TimestepObserver
///
/// \brief this class describes an observer which can be
///        used to detect particles inside a certain volume
///////////////////////////////
template <class observable_t>
class TimestepObserver : public Observer<observable_t>
{
public:
    typedef typename Observer<observable_t>::real_t real_t;
    typedef typename Observer<observable_t>::size_t size_t;
    typedef typename observable_t::coord_t coord_t;
    typedef typename observable_t::particle_list_t particle_list_t;

    using Observer<observable_t>::observable;
    using Observer<observable_t>::activateTime;
    using Observer<observable_t>::deactivateTime;
    using Observer<observable_t>::inverseFrequency;
    using Observer<observable_t>::getOutputStream;

    /// default constructor
    TimestepObserver(observable_t& _observable,
            std::ostream& ostr = std::cout)
            : Observer<observable_t>(_observable, ostr)
    {
    }

    /// normal copy constructor
    TimestepObserver(TimestepObserver<observable_t>& o)
            : Observer<observable_t>(o)
    {
    }

    /// copy constructor with the abilty to specify a new observable
    template<class observer_t>
    TimestepObserver(observer_t& o, observable_t& _observable)
            : Observer<observable_t>(o, _observable)
    {
    }

    /// virtual destructor
    /// this makes sure that the destructors of the derived classes
    /// are called when their objects get "delete"ed through a
    /// refernece of this type
    virtual ~TimestepObserver()
    {
    }

    /// this virtual method is (hopefully) called by observables
    ///
    /// \param time the time of the notification
    /// \param timestep the "resolution" of the time
    virtual void notify(real_t time, real_t timestep)
    {
        const particle_list_t& particles
                = observable.getParticles();
        for (particle_list_t::const_iterator
                p = particles.begin();
                p != particles.end(); ++p)
        {
            real_t max_v = observable.calculateMaxAbsVelocity();
```

```
            real_t new_dt = spacialResolution/max_v;
        }
    }

    /// set the desired spacial resolution in mm
    void setSpacialResolution(real_t ds)
    {
        spacialResolution = ds;
    }

    /// get the desired spacial resolution in mm
    real_t getSpacialResolution() const
    {
        return spacialResolution;
    }

private:
    real_t spacialResolution;

};

}
#endif
```

```
// trajectories_observer.h: v2.4, 2009−05−06, Yves Salathe
#ifndef __TRAJECTORIES_OBSERVER_H__
#define __TRAJECTORIES_OBSERVER_H__

#include "observer.h"
#include "particle.h"
#include "valgrad.h"
#include "vector3d.h"
#include <cmath>

namespace simulation
{

///////////////////////////////
/// \class TrajectoriesObserver
///
/// \brief this class describes an observer which can be
///        used to detect particles inside a certain volume
///////////////////////////////
template <class observable_t>
class TrajectoriesObserver : public Observer<observable_t>
{
public:
    typedef typename Observer<observable_t>::real_t real_t;
    typedef typename Observer<observable_t>::size_t size_t;
    typedef typename observable_t::coord_t coord_t;
    typedef typename observable_t::particle_list_t particle_list_t;

    using Observer<observable_t>::observable;
    using Observer<observable_t>::activateTime;
    using Observer<observable_t>::deactivateTime;
    using Observer<observable_t>::inverseFrequency;
    using Observer<observable_t>::getOutputStream;

    /// default constructor
    TrajectoriesObserver(observable_t& _observable,
            std::ostream& ostr = std::cout)
            : Observer<observable_t>(_observable, ostr)
    {
    }

    /// normal copy constructor
    TrajectoriesObserver(TrajectoriesObserver<observable_t>& o)
            : Observer<observable_t>(o)
    {
    }

    /// copy constructor with the abilty to specify a new observable
    template<class observer_t>
    TrajectoriesObserver(observer_t& o, observable_t& _observable)
            : Observer<observable_t>(o, _observable)
    {
    }

    /// virtual destructor
    /// this makes sure that the destructors of the derived classes
    /// are called when their objects get "delete"ed through a
    /// refernece of this type
    virtual ~TrajectoriesObserver()
    {
    }

    /// this virtual method is (hopefully) called by observables
    ///
    /// \param time the time of the notification
    /// \param timestep the "resolution" of the time
    virtual void notify(real_t time, real_t timestep)
    {
        const particle_list_t& particles
        = observable.getParticles();
        for (typename particle_list_t::const_iterator
                p = particles.begin();
                p != particles.end(); ++p)
        {
            outputTrajectory(*p,time);
```

```
        }
    }

private:
    /// output of the trajectory of a particle
    inline void outputTrajectory(const Particle& p, real_t time)
    {
        getOutputStream() << time << " ";
        p.outputInformation(getOutputStream());
        math::ValGrad<real_t> Bv =
            observable.engine.calculateAbsBField(p.pos.getZ(),p.posR());
        getOutputStream() << " "
            << " " << observable.calculateKineticEnergy(p)
            << " "
            << observable.getPopulations()[p.popId].calculateZeemanEnergy(Bv.val
)
            << " "
            << observable.getPopulations()[p.popId].calculateZeemanEnergyDeriv(B
v.val)
            << " "
            << Bv.val
            << " "
            << Bv.grad_dim1
            << " "
            << Bv.grad_dim2
            << std::endl;
    }
};

}
#endif
```

```
// trappulsegen_observer.h: v2.4, 2009-05-06, Yves Salathe
#ifndef __TRAPPULSEGEN_OBSERVER_H__
#define __TRAPPULSEGEN_OBSERVER_H__

#include "observer.h"
#include "particle.h"
#include "matrix.h"
#include "coil.h"
#include <cmath>

namespace simulation
{

///////////////////////////////
/// \class TrapPulseGenObserver
///
/// \brief this class describes an observer which can be
///        used to detect particles inside a certain volume
///////////////////////////////
template <class observable_t>
class TrapPulseGenObserver : public Observer<observable_t>
{
public:
    typedef typename Observer<observable_t>::real_t real_t;
    typedef typename Observer<observable_t>::size_t size_t;
    typedef typename observable_t::coord_t coord_t;
    typedef typename observable_t::particle_list_t particle_list_t;

    using Observer<observable_t>::observable;
    using Observer<observable_t>::activateTime;
    using Observer<observable_t>::deactivateTime;
    using Observer<observable_t>::inverseFrequency;
    using Observer<observable_t>::getOutputStream;

    /// default constructor
    TrapPulseGenObserver(observable_t& _observable,
            std::ostream& ostr = std::cout)
            : Observer<observable_t>(_observable, ostr),
            frontPulseNo(0)
    {
        setPulseOverlap();
        setInverseFrequency(observable.getTimestep());
    }

    /// normal copy constructor
    TrapPulseGenObserver(TrapPulseGenObserver<observable_t>& o)
            : Observer<observable_t>(o),
            frontPulseNo(o.frontPulseNo)
    {
        setPulseOverlap(o.pulseOverlap);
    }

    /// copy constructor with the abilty to specify a new observable
    template<class observer_t>
    TrapPulseGenObserver(observer_t& o, observable_t& _observable)
            : Observer<observable_t>(o, _observable),
            frontPulseNo(o.frontPulseNo)
    {
        setPulseOverlap(o.pulseOverlap);
    }

    /// virtual destructor
    /// this makes sure that the destructors of the derived classes
    /// are called when their objects get "delete"ed through a
    /// refernece of this type
    virtual ~TrapPulseGenObserver()
    {
    }

    /// this virtual method is (hopefully) called by observables
    ///
    /// \param time the time of the notification
    /// \param timestep the "resolution" of the time
    virtual void notify(real_t time, real_t timestep)
    {
```

```
        real_t z = observable.getParticles().front().pos.getZ();
        bool pulsesAdjusted = false;
        if (time >= observable.engine.lastDecelCoil.getPulse(0)
                        .startTime
                && time < observable.engine.lastDecelCoil.getPulse(0)
                        .endTime)
        {
            real_t z_coil = observable.engine.lastDecelCoil.getPosZ();
            real_t d = decelCoilDist;
            if(((z-z_coil)/d+.5)*180 > phasedeg_lastdecel_off) {
                // deactivate last decelerator coil
                Coil::Pulse decelPulse = observable.engine.
                        lastDecelCoil.getPulse(0);
                decelPulse.endTime = time;
                observable.engine.lastDecelCoil.setActualPulse(0,
                        decelPulse);
                // activate front trap coil
                Coil::Pulse frontPulse1(time-pulseOverlap,
                        observable.getEndTime(),
                        coeff_front_1);
                observable.engine.coil1.addPulseWithoutIncouplingTime
                        (frontPulse1);
                getOutputStream()
                        << "switch off last decelerator coil "
                        << "at time " << time << std::endl;
                pulsesAdjusted = true;
            }
        }
        if (observable.engine.coil1.getPulses().size() >= 1) {
            switch(frontPulseNo) {
                case 0:
                    if (time >= observable.engine.coil1.getPulse(0)
                                    .startTime
                            && time < observable.engine.coil1.getPulse(0)
                                    .endTime)
                    {
                        real_t z_coil = observable.engine.coil1.getPosZ();
                        real_t d = observable.engine.getDistBetweenCoilsZ();
                        if(((z-z_coil)/d+.5)*180 > phasedeg_front_1off) {
                            // deactivate front trap coil
                            Coil::Pulse frontPulse1 = observable.engine
                                    .coil1.getPulse(0);
                            frontPulse1.endTime = time;
                            observable.engine.coil1.setActualPulse(0,
                                    frontPulse1);
                            // activate rear trap coil
                            Coil::Pulse rearPulse(time-pulseOverlap,
                                    observable.getEndTime(),
                                    coeff_rear);
                            observable.engine.coil2
                                    .addPulseWithoutIncouplingTime
                                            (rearPulse);
                            ++frontPulseNo;
                            getOutputStream()
                                    << "switch off front trap coil "
                                    << "at time " << time << std::endl;
                            pulsesAdjusted = true;
                        }
                    }
                    break;
                case 1:
                    if (observable.engine.coil1.getPulses().size() == 1) {
                        real_t velZ = observable.getParticles().front()
                                .vel.getZ();
                        real_t z_coil = observable.engine.coil1.getPosZ();
                        real_t d = observable.engine.getDistBetweenCoilsZ();
                        if (velZ < 0 && ((z-z_coil)/d+.5)*180
                                < phasedeg_front_2on) {
                            // activate front trap coil
                            Coil::Pulse frontPulse2(time-timestep,observable
                                    .getEndTime(),coeff_front_2);
                            observable.engine.coil1
                                    .addPulseWithoutIncouplingTime
                                            (frontPulse2);
                            getOutputStream()
```

```
                                          << "second pulse of front trap coil "
                                          << "at time " << time << std::endl;
                                  pulsesAdjusted = true;
                              }
                          }
                          break;
                      default:
                          // this should never happen
                          break;
                  }
              }
              if(pulsesAdjusted) {
                  observable.engine.prepareTimestepping(time,timestep);
              }
          }

          /// set the phaseangle at which to switch off the last decelerator
          /// coil in degrees
          void setLastDecelCoilEndPulsePhaseDeg(real_t phi)
          {
              phasedeg_lastdecel_off = phi;
          }

          /// set the phaseangle at which to switch off the front trap
          /// coil in degrees
          void setFrontTrapCoilEndFirstPulsePhaseDeg(real_t phi)
          {
              phasedeg_front_1off = phi;
          }

          /// set the phaseangle at which to switch on the front trap
          /// coil in degrees
          void setFrontTrapCoilStartSecondPulsePhaseDeg(real_t phi)
          {
              phasedeg_front_2on = phi;
          }

          /// set the overlap between two successive stage−coil−pulses in us
          /// this is also used when going from a tower pulse to a stage pulse
          void setPulseOverlap(real_t deltat = 3)
          {
              pulseOverlap = deltat;
          }

          /// set the scaling of the first pulse of the front trap coil
          void setFrontTrapCoilFirstPulseCoeff(const std::valarray<real_t>& v)
          {
              coeff_front_1.resize(v.size());
              coeff_front_1 = v;
          }

          /// set the scaling of the second pulse of the front trap coil
          void setFrontTrapCoilSecondPulseCoeff(const std::valarray<real_t>& v)
          {
              coeff_front_2.resize(v.size());
              coeff_front_2 = v;
          }

          /// set the scaling of the second pulse of the front trap coil
          void setRearTrapCoilPulseCoeff(const std::valarray<real_t>& v)
          {
              coeff_rear.resize(v.size());
              coeff_rear = v;
          }


          /// set the distance between the decelerator coils
          void setDecelCoilDist(real_t d)
          {
              decelCoilDist = d;
          }

      private:
          real_t pulseOverlap;
          real_t phasedeg_lastdecel_off;
```

```
          real_t phasedeg_front_1off;
          real_t phasedeg_front_2on;
          std::valarray<real_t> coeff_front_1;
          std::valarray<real_t> coeff_front_2;
          std::valarray<real_t> coeff_rear;
          real_t decelCoilDist;
          size_t frontPulseNo;
      };

      }
      #endif
```

```
// valgrad.h: 2009−02−18, Yves Salathe
#ifndef __VALGRAD_H__
#define __VALGRAD_H__

namespace math
{

///////////////////////////////////////////////////////////////////////
/// \class ValGrad
///
/// \brief this simple data structure describes a value and the
/// corresponding gradient
///////////////////////////////////////////////////////////////////////
template <class value_t>
class ValGrad
{
public:

    /// default constructor
    inline ValGrad(value_t v=0.0, value_t g1=0.0, value_t g2=0.0)
            : val(v),grad_dim1(g1),grad_dim2(g2) { }

    /// copy constructor
    inline ValGrad(const ValGrad& v)
            : val(v.val),
            grad_dim1(v.grad_dim1),
            grad_dim2(v.grad_dim2) { }

    /// assignment operator
    inline ValGrad& operator=(ValGrad& v)
    {
        val = v.val;
        grad_dim1 = v.grad_dim1;
        grad_dim2 = v.grad_dim2;
        return *this;
    }

    /// summation operator
    inline ValGrad& operator+=(ValGrad& v)
    {
        val += v.val;
        grad_dim1 += v.grad_dim1;
        grad_dim2 += v.grad_dim2;
        return *this;
    }

    value_t val;
    value_t grad_dim1;
    value_t grad_dim2;
};

}
#endif
```

```
// vector.h: v2.4, 2009-05-06, Yves Salathe
#ifndef __VECTOR3D_H__
#define __VECTOR3D_H__

#include <cmath>

namespace math
{

/// \class Vector3d
///
/// \brief this class uses engine delegation as described by
/// Todd Veldhuizen in "Techniques for Scientific C++"
template<class T, class E> class Vector3d
{

public:
    /// default constructor
    inline Vector3d() { }

    /// constructor
    inline Vector3d(T x, T y, T z)
    {
        setX(x);
        setY(y);
        setZ(z);
    }

    /// copy constructor
    template<class T2, class E2>
    inline Vector3d(Vector3d<T2,E2>& v)
            : engine(v)
    {
    }

    /// set the vector from an array of 3 elements
    inline void setFromArray(const T a[])
    {
        engine.setX(a[0]);
        engine.setY(a[1]);
        engine.setZ(a[2]);
    }

    /// write the vector to an array of 3 elements
    inline void writeToArray(T a[]) const
    {
        a[0] = engine.getX();
        a[1] = engine.getY();
        a[2] = engine.getZ();
    }

    /// get first cartesian coordinate
    inline T getX() const
    {
        return engine.getX();
    }

    /// get second cartesian coordinate
    inline T getY() const
    {
        return engine.getY();
    }

    /// get second cartesian coordinate
    inline T getZ() const
    {
        return engine.getZ();
    }

    /// set first cartesian coordinate
    inline void setX(T x)
    {
        return engine.setX(x);
    }

```

```
    /// set second cartesian coordinate
    inline void setY(T y)
    {
        return engine.setY(y);
    }

    /// set second cartesian coordinate
    inline void setZ(T z)
    {
        return engine.setZ(z);
    }

    /// returns the 2-norm of the vector
    inline T norm2() const
    {
        return engine.norm2();
    }

    /// returns the square of the 2-norm of the vector
    inline T norm2_2() const
    {
        return engine.norm2_2();
    }

    /// sets the length of the vector in the 2-norm
    inline void set_norm2(T l)
    {
        engine.set_norm2(l);
    }

    /// returns the dot product of vector v with this vector
    template <class T2, class E2>
    inline T dot(const Vector3d<T2,E2>& v) const
    {
        return engine.dot(v);
    }

    /// calculates the corss-product of vector v2 with this vector
    template <class T2, class E2>
    inline Vector3d<T,E> cross(const Vector3d<T2,E2>& v2) const
    {
        Vector3d<T,E> v3;
        engine.cross(v2, v3);
        return v3;
    }

    /// element-wise multiplication
    template <class T2, class E2>
    inline Vector3d<T,E> elemMult(const Vector3d<T2,E2>& v2) const
    {
        Vector3d<T,E> v3;
        v3.setX(getX()*v2.getX());
        v3.setY(getY()*v2.getY());
        v3.setZ(getZ()*v2.getZ());
        return v3;
    }

    /// assignment
    template <class T2, class E2>
    inline Vector3d<T,E>& operator=(const Vector3d<T2,E2>& v)
    {
        engine.assign(v);
        return *this;
    }

    /// compound association with addition
    template <class T2, class E2>
    inline Vector3d<T,E>& operator+=(const Vector3d<T2,E2>& v)
    {
        setX(getX() + v.getX());
        setY(getY() + v.getY());
        setZ(getZ() + v.getZ());
        return *this;
    }

    /// compound association with subtraction
    template <class T2, class E2>
    inline Vector3d<T,E>& operator-=(const Vector3d<T2,E2>& v)
    {
        setX(getX() - v.getX());
        setY(getY() - v.getY());
        setZ(getZ() - v.getZ());
        return *this;
    }

    /// compound association with multiplication by a scalar
    inline Vector3d<T,E>& operator*=(T s)
    {
        engine.mult(s,*this);
        return *this;
    }

    /// compound association with division by a scalar
    inline Vector3d<T,E>& operator/=(T s)
    {
        engine.div(s,*this);
        return *this;
    }

    /// vector addition
    template <class T2, class E2>
    inline Vector3d<T,E> operator+(const Vector3d<T2,E2>& rv) const
    {
        return Vector3d<T,E>(*this)+=rv;
    }

    /// vector subtraction
    template <class T2, class E2>
    inline Vector3d<T,E> operator-(const Vector3d<T2,E2>& rv) const
    {
        return Vector3d<T,E>(*this)-=rv;
    }

    /// multiplication by scalar s
    inline Vector3d<T,E> operator*(T s) const
    {
        return Vector3d<T,E>(*this)*=s;
    }

    /// division by scalar s
    inline Vector3d<T,E> operator/(T s) const
    {
        return Vector3d<T,E>(*this)/=s;
    }

    /// normalize the vector
    inline void normalize()
    {
        engine.normalize();
    }

private:
    E engine;
};

template <class T>
class CartesianCoord
{
    /// This engine for Vector3d uses cartesian coordinates to store the
    /// vector
public:

    /// default constructor
    inline CartesianCoord() { }

    /// copy constructor
    template <class T2, class E2>
    inline CartesianCoord(const Vector3d<T2,E2>& v)
            : x(v.getX()),y(v.getY()),z(v.getZ())
    {
```

```
    }

    /// assign a different vector to this vector
    template <class T2, class E2>
    inline void assign(const Vector3d<T2,E2>& v)
    {
        x = v.getX();
        y = v.getY();
        z = v.getZ();
    }

    /// get first cartesian coordinate
    inline T getX() const
    {
        return x;
    }

    /// get second cartesian coordinate
    inline T getY() const
    {
        return y;
    }

    /// get second cartesian coordinate
    inline T getZ() const
    {
        return z;
    }

    /// set first cartesian coordinate
    inline void setX(T v)
    {
        x = v;
    }

    /// set second cartesian coordinate
    inline void setY(T v)
    {
        y = v;
    }

    /// set second cartesian coordinate
    inline void setZ(T v)
    {
        z = v;
    }

    /// returns the 2-norm of the vector
    inline T norm2() const
    {
        return std::sqrt(norm2_2());
    }

    /// returns the square of the 2-norm of the vector
    inline T norm2_2() const
    {
        return x*x+y*y+z*z;
    }

    /// sets the length of the vector
    inline void set_norm2(T l)
    {
        T n = norm2();
        T f = l/n;
        x = f*x;
        y = f*y;
        z = f*z;
    }

    /// returns the dot product of vector v with this vector
    template <class T2, class E2>
    inline T dot(const Vector3d<T2,E2>& v) const
    {
        return x*v.getX()+y*v.getY()+z*v.getZ();
    }
    /// calculates the corss−product of vector v with this vector
    /// and stores it in vector v3
    template <class T2, class E2, class T3, class E3>
    inline void cross(const Vector3d<T2,E2>& v2,
                      Vector3d<T3,E3>& v3) const
    {
        v3.setX(y*v2.getZ() − z*v2.getY());
        v3.setY(z*v2.getX() − x*v2.getZ());
        v3.setZ(x*v2.getY() − y*v2.getX());
    }

    /// multiplication by scalar s
    template <class T2, class E2>
    inline void mult(T s, Vector3d<T2,E2>& v) const
    {
        v.setX(x*s);
        v.setY(y*s);
        v.setZ(z*s);
    }

    /// division by scalar s
    template <class T2, class E2>
    inline void div(T s, Vector3d<T2,E2>& v) const
    {
        v.setX(x/s);
        v.setY(y/s);
        v.setZ(z/s);
    }

    /// normalize vector w.r.t. 2−norm
    inline void normalize()
    {
        T n = norm2();
        x = x/n;
        y = y/n;
        z = z/n;
    }

private:
    T x;
    T y;
    T z;
};

}
#endif
```

```cpp
// verlet_integrator.h: v2.42, 2009-05-07, Yves Salathe
#ifndef __VERLET_INTEGRATOR_H__
#define __VERLET_INTEGRATOR_H__

#include "integrator.h"
#include "particle.h"
#include "matrix.h"
#include "vector3d.h"
#include "assert.h"
#include "exceptions.h"

namespace simulation
{

////////////////////////////////////
/// \class VerletIntegrator
///
/// \brief This class describes a method to integrate newton's equations
///        of motion. It can be used as an integrator in the class
///        of type simulation_t (usually ParticleSim)
////////////////////////////////////
template<class simulation_t>
class VerletIntegrator : public Integrator<simulation_t>
{
public:
    ////////////////////////////////////
    // type definitions
    ////////////////////////////////////

    //use the same types as the simulation
    typedef typename simulation_t::real_t real_t;
    typedef typename simulation_t::size_t size_t;
    typedef typename simulation_t::coord_t coord_t;
    typedef typename simulation_t::particle_list_t particle_list_t;
    typedef typename simulation_t::population_list_t population_list_t;

    /// constructor
    VerletIntegrator() : nParticles(0)
    {
    }

    /// destructor
    ~VerletIntegrator()
    {
        if(nParticles > 0) {
            delete[] id;
            delete[] popId;
            delete[] oldPos;
            delete[] pos;
            delete[] accel;
        }
    }

    void initialize(simulation_t& simulation,
            real_t time, real_t timestep)
    {
        // free previously allocated memory
        if(nParticles > 0) {
            delete[] id;
            delete[] popId;
            delete[] oldPos;
            delete[] pos;
            delete[] accel;
        }
        // get a reference to the list of particles
        const particle_list_t& particles = simulation.getParticles();
        nParticles = particles.size();
        // allocate memory
        id = new size_t[nParticles];
        popId = new size_t[nParticles];
        size_t n3 = 3*nParticles;
        oldPos = new real_t[n3];
        pos = new real_t[n3];
        accel = new real_t[n3];
        // initialize masses and positions
```

```cpp
        typename particle_list_t::const_iterator p = particles.begin();
        for(size_t i = 0; i<nParticles; ++i) {
            size_t i3 = 3*i;
            id[i] = p->id;
            popId[i] = p->popId;
            p->pos.writeToArray(&pos[i3]);
            // initialize old positions
            if (p->oldPosInitialized) {
                p->oldPos.writeToArray(&oldPos[i3]);
            } else {
                initializeOldPosition(*p,i3,timestep);
            }
            ++p;
        }
    }

    /// timestepping: perform a timestep of dt
    /// (integrate equation of motion for all particles)
    ///
    /// \param simulation       a reference to the simulation
    /// \param time             the start time of the step
    /// \param timestep         the size of the step (time difference)
    /// \return the new reference time
    inline real_t performTimestep(simulation_t& simulation,
                                  real_t t, real_t dt)
    {
        // calculate the acceleration of each particle at that time
        simulation.calculateAccelerations(t,dt,nParticles, popId, pos, accel);
        // go through the list of particles and integrate the
        // equations of motion for each of them separately
        size_t n3 = 3*nParticles;
        for (size_t i = 0; i<n3; ++i)
        {
            real_t currentPos = pos[i];
            pos[i] = 2*pos[i]-oldPos[i]+dt*dt*accel[i];
            oldPos[i] = currentPos;
        }
        last_dt = dt;
        return t+dt;
    }

    /// store the computed particle information back to the simulation
    void updateSimulation(simulation_t& simulation)
    {
        // get a reference to the list of particles
        particle_list_t& particles = simulation.getParticles();
        typename particle_list_t::iterator p = particles.begin();
        for(size_t i = 0; i<nParticles; ++i) {
            while(p->id != id[i] && p != particles.end()) {
                ++p;
            }
            if (p != particles.end()) {
                size_t i3 = 3*i;
                p->pos.setFromArray(&pos[i3]);
                p->oldPos.setFromArray(&oldPos[i3]);
                p->oldPosInitialized = true;
                p->vel = (p->pos - p->oldPos)/last_dt;
            } else {
                throw ExSizeMismatch();
            }
        }
    }

private:

    /// the number of particles
    size_t nParticles;
    /// the ids of the particles
    size_t* id;
    /// an array containing the population id of each particle
    size_t* popId;
    /// an array containing the masses of each particle
    real_t* mass;
    /// an array of size 3*numbParticles containing the positions
    /// at time t-dt of the particles in cartesian coordinates
```

```
        real_t* oldPos;
        /// an array of size 3*numbParticles containing the positions at
        /// time t of the particles in cartesian coordinates
        real_t* pos;
        /// an array of size 3*numbParticles containing the accelerations of the
        /// particles in cartesian coordinates
        real_t* accel;
        /// the last time step size used
        real_t last_dt;

        /// initialize old position of particle i3/3
        inline void initializeOldPosition(Particle p, size_t i3, real_t dt) {
            oldPos[i3] = pos[i3] − dt*p.vel.getX();
            oldPos[i3+1] = pos[i3+1] − dt*p.vel.getY();
            oldPos[i3+2] = pos[i3+2] − dt*p.vel.getZ();
        }

        /// initialize acceleration for all particles
        inline void initializeAccelerations(real_t v = 0) const
        {
            size_t n3 = 3*nParticles;

            for (size_t i = 0; i<n3; ++i) {
                accel[i] = 0;
            }
        }

    };

    }
    #endif
```

```
// walltime.h: 2009-02-26, Yves Salathe
#ifndef __WALLTIME_H__
#define __WALLTIME_H__

#include <sys/time.h>

double walltime(double *t0)
{
    // get walltime in seconds
    double mic, time;
    double mega=0.000001;
    struct timeval tp;
    struct timezone tzp;
    static long base_sec = 0;
    static long base_usec = 0;

    (void) gettimeofday(&tp, &tzp);
    if (base_sec == 0)
    {
        base_sec  = tp.tv_sec;
        base_usec = tp.tv_usec;
    }
    time = (double)(tp.tv_sec - base_sec);
    mic = (double)(tp.tv_usec - base_usec);
    time = (time + mic * mega) - *t0;
    return(time);
}
#endif
```

```
   // zeemandecel_builder.h: v2.4, 2009-05-21, Yves Salathe
#ifndef __ZEEMANDECEL_BUILDER__
#define __ZEEMANDECEL_BUILDER__

#include "particlesim_builder.h"
#include "zeemandecel.h"
#include "laser_detector.h"
#include "trajectories_observer.h"
#include "decelpulsegen_observer.h"
#include "matrix.h"
#include "grid.h"
#include "assert.h"
#include "exceptions.h"
#include "input.h"
#include <list>
#include <fstream>

namespace simulation {

class ZeemanDecelBuilder : public ParticleSimBuilder<ZeemanDecel> {
public:

    ZeemanDecelBuilder(
            const char _inputFileName[] = "inputdata_long.txt")
                :
                ParticleSimBuilder<ZeemanDecel>(),
                indata(_inputFileName)
    {
        setParameters();
    }

    ZeemanDecelBuilder(const math::Matrix<std::string>& _indata)
                :
                ParticleSimBuilder<ZeemanDecel>(),
                indata(_indata)
    {
        setParameters();
    }

    ZeemanDecelBuilder(const InputData& _indata)
                :
                ParticleSimBuilder<ZeemanDecel>(),
                indata(_indata)
    {
        setParameters();
    }

    virtual ~ZeemanDecelBuilder()
    {
    }

    void readBFieldsFromFiles(
            const char stages_field_filename[] = "decelmapaxrho.txt",
            const char towers_field_filename[] = "towermapaxrho.txt")
    {
        ///////////////////////////////////////////////////////////////
        //read in b-field of the decelerator coils                  //
        ///////////////////////////////////////////////////////////////

        math::Matrix<ZeemanDecel::real_t> bfield(stages_field_filename,4);
        // extract axial part of the field
        math::Grid2d<ZeemanDecel::real_t> stageBz(bfield,2);
        // extract radial part of the field
        math::Grid2d<ZeemanDecel::real_t> stageBr(bfield,3);
        simulation->engine.setStageCoilB(stageBz,stageBr);
        if (simulation->engine.getNumbStages() > 1)
        {
            //////////////////////////////////////////////////////
            // if there is more than one stage                  //
            // read in b-field of the tower coils               //
            //////////////////////////////////////////////////////
            math::Matrix<ZeemanDecel::real_t>
                    tfield(towers_field_filename,4);
            // extract axial part of the field
            math::Grid2d<ZeemanDecel::real_t> towerBz(tfield,2);
```

```
            // extract radial part of the field
            math::Grid2d<ZeemanDecel::real_t> towerBr(tfield,3);
            simulation->engine.setTowerCoilB(towerBz,towerBr);
        }
    }

    real_t getDefaultMassFactor() const
    {
        return indata.getReal("mass_factor");
    }

    size_t getNumberOfParticlesPerPopulation() const
    {
        return indata.getInt("number_of_particles");
    }

    LaserDetector<ParticleSim<ZeemanDecel> >*
            addLaserObserver(const char output_filename[]
            = "output_detection.txt")
    {
        LaserDetector<ParticleSim<ZeemanDecel> >* laser
                = addObserver<LaserDetector<ParticleSim<ZeemanDecel> > >
                        (output_filename);
        setLaserParameters(*laser);
        return laser;
    }

    LaserDetector<ParticleSim<ZeemanDecel> >*
            addLaserObserver(std::ostream& laserOutput)
    {
        LaserDetector<ParticleSim<ZeemanDecel> >* laser
                = addObserver<LaserDetector<ParticleSim<ZeemanDecel> > >
                        (laserOutput);
        setLaserParameters(*laser);
        return laser;
    }

    TrajectoriesObserver<ParticleSim<ZeemanDecel> >*
            addTrajectoriesObserver(const char output_filename[]
            = "output_trajectories.txt")
    {
        return addObserver<TrajectoriesObserver<ParticleSim<
                ZeemanDecel> > >(output_filename);
    }

    TrajectoriesObserver<ParticleSim<ZeemanDecel> >*
            addTrajectoriesObserver(std::ostream& trajOutput)
    {
        return addObserver<TrajectoriesObserver<ParticleSim<
                ZeemanDecel> > >(trajOutput);
    }

    DecelPulseGenObserver<ParticleSim<ZeemanDecel> >*
            addPulseGenObserver(const char output_filename[]
                        = "output_pg_decel.txt",
                    const char input_filename[]
                        = "inputdata_pg_decel.txt",
                    const char phaseangles_filename[]
                        = "inputdata_pg_decel_phasedeg.txt")
    {
        DecelPulseGenObserver<ParticleSim<ZeemanDecel> >* pulseGen
                = addObserver<DecelPulseGenObserver<ParticleSim<
                        ZeemanDecel> > >(output_filename);
        math::Matrix<std::string> inputPG(input_filename,2);
        Assert<ExWrongInput>(inputPG.rows() == 4);
        // TODO: provide a more flexible input format
        pulseGen->setSwitchOnTimeFirstCoil(std::atof(inputPG(0,1).c_str()));
        pulseGen->setStagePulseOverlap(std::atof(inputPG(1,1).c_str()));
        pulseGen->setTowerPulseOverlap(std::atof(inputPG(2,1).c_str()));
        size_t leaveTowerCoilsOn = std::atoi(inputPG(3,1).c_str());
        Assert<ExWrongInput>(leaveTowerCoilsOn == 0
                || leaveTowerCoilsOn == 1);
        pulseGen->setLeaveTowerCoilsOn(leaveTowerCoilsOn == 1);
        ZeemanDecel::size_t m = simulation->engine.getNumbStages();
        ZeemanDecel::size_t n = simulation->engine.getNumbStageCoils();
```

```
            ZeemanDecel::size_t n_tow = simulation->engine.getTowerCoils();
            if (m > 1 && !leaveTowerCoilsOn) {
                math::Matrix<real_t> phasedeg(phaseangles_filename,n+n_tow);
                pulseGen->setPhaseAngleDegreesFromMatrix(phasedeg);
            } else {
                math::Matrix<real_t> phasedeg(phaseangles_filename,n);
                pulseGen->setPhaseAngleDegreesFromMatrix(phasedeg);
            }
            return pulseGen;
        }

    private:

        InputData indata;

        void setParameters()
        {
            // initial beam length in mm
            simulation->setInitBeamLength(indata.getReal("init_beam_length_mm"));
            // velocity of the sychron. particle in mm/us
            simulation->setVz(indata.getReal("starting_velocity_m/s")/1000);
            // relative (to synchronous particel resp. vz)
            // temperature in z direction
            simulation->setTz(indata.getReal("long_temperature_K"));
            // relative temperature in x,y direction
            simulation->setTxy(indata.getReal("trans_temp_K"));
            // numerical timesteps in us (should be 1-10ns)
            simulation->setTimestep(indata.getReal("timesteps_ns")/1000);
            // sets starting time of the simulations in us
            simulation->setTime(indata.getReal("time_offset_in_us"));
            simulation->setEndTime(
                    indata.getReal("time_simulation_stops_us"));
            // sets the initial z-position of the particles
            simulation->setStartPosZ(indata.getReal("start_pos_z"));
            simulation->setInitialRadialPositionOnDisk(
                    indata.getInt("start_radial_pos_on_disk"));
            // sets the number of coils per stage
            simulation->engine.setNumbStageCoils(
                    indata.getInt("number_of_coils_per_stage"));
            // distance to the middle of the first coil in mm
            simulation->engine.setDistToMidFirstCoil(
                    indata.getReal("distance_to_middle_first_coil_mm"));
            // ramping time B-field of the decelerator coils in us
            simulation->engine.setStageRampOn(
                    indata.getReal("ramp_time_coil_ns")/1000);
            // ramping time B-field of the decelerator coils in us
            simulation->engine.setStageRampOff(
                    indata.getReal("ramp_time_coil_ns")/1000);
            // current that is used in the decelerator coils in Ampere
            simulation->engine.setStageCurrent(
                    indata.getReal("current_decelerator_A"));
            // reference current that has been used to simulate the decelerator
            // coils in Ampere
            simulation->engine.setStageSimCurrent(
                    indata.getReal("reference_current_decelerator_A"));
            // sets the number of decelerator stages
            simulation->engine.setNumbStages(
                    indata.getInt("number_of_stages"));
            // sets distance between two decelerator stages
            // (from the middle of the last coil to the middle of the next)
            simulation->engine.setDistBetweenStages(
                    indata.getReal("distance_between_stages_mm"));
            // number of coils in the tower
            simulation->engine.setTowerCoils(
                    indata.getInt("number_of_coils_in_a_tower"));
            // distance between tower coils
            simulation->engine.setTowerDist(
                    indata.getReal("distance_towercoils_mm"));
            // current in the towercoils
            simulation->engine.setTowerCurrent(
                    indata.getReal("current_tower_coils_A"));
            // reference current that has been used to simulate the tower
            // coils in Ampere
            simulation->engine.setTowerSimCurrent(
                    indata.getReal("reference_current_tower_coils_A"));
```

```
            // switching time of the tower coils
            simulation->engine.setTowerRampOn(
                    indata.getReal("ramp_time_tower_coil_ns")/1000);
            // switching time of the tower coils
            simulation->engine.setTowerRampOff(
                    indata.getReal("ramp_time_tower_coil_ns")/1000);
            // z-position where particles are taken out of the simulation
            simulation->engine.setBoundZ(indata.getReal("bound_z_mm"));
            // sets distance from the last coil of a stage to the first
            // coil inside the succeeding tower
            simulation->engine.setDistStageToTower(
                    (simulation->engine.getDistBetweenStages()
                    -(simulation->engine.getTowerCoils()-1)
                    *simulation->engine.getTowerDist())/2);
            simulation->engine.setIncouplingTime(
                    indata.getReal("incoupling_time_us"));
            simulation->engine.setCoilDist(
                    indata.getReal("distance_between_stage_coils_mm"));
            simulation->setRemoveUnacceptedLFS(
                    indata.getInt("remove_unaccepted_lfs_particles"));
            if(indata.getInt("check_upper_bound_for_predicted_total_energy")
                    == 1)
            {
                simulation->engine.setUpperTargetTotalEnergy(
                        indata.getReal
                            ("upper_bound_for_predicted_total_energy_J"));
            }
        }

        void setLaserParameters(LaserDetector<ParticleSim<ZeemanDecel> >& laser)
        {
            laser.setDetectorPosZ(
                    indata.getReal("detector_pos_mm"));
            laser.setActivateTime(
                    indata.getReal("detector_start_time_us"));
            laser.setDetectorRadius(
                    indata.getReal("detector_radius_mm"));
            laser.setInverseFrequency(
                    indata.getReal("detector_inverse_frequency_us"));
        }

    };


    }

    #endif
```

```
// zeemandecel.h: v2.4, 2009−05−06, Yves Salathe
#ifndef __ZEEMANDECEL_H__
#define __ZEEMANDECEL_H__

#include "engine.h"
#include "matrix.h"
#include "vector3d.h"
#include "grid.h"
#include "assert.h"
#include "exceptions.h"
#include "basic_math.h"
#include "particle.h"
#include <vector>
#include <iostream>
#include <cmath>
#include <limits>

namespace simulation
{

///////////////////////////////
/// \class ZeemanDecel
///
/// \brief This is the class which implements the Zeeman−decelerator
///        as an engine which can be used in the class
///        ParticleSim<ZeemanDecel,integrator_t>.
///////////////////////////////
class ZeemanDecel : public Engine
{
public:

    ///////////////////////////////
    //constructors
    ///////////////////////////////

    /// constructor
    ZeemanDecel() : activeStageCoils(0),
            activeTowerCoils(0), numbstages(1), numbstagecoils(12),
            towercoils(0), maxStageAbsB(0)
    {
        // set default values
        setStageSimCurrent();
        setTowerSimCurrent();
        setStageCurrent();
        setTowerCurrent();
        setStageRampOn();
        setStageRampOff();
        setTowerRampOn();
        setTowerRampOff();
        setNumbStageCoils();
        setTowerCoils();
        setNumbStages();
        setDistToMidFirstCoil();
        setCoilDist();
        setTowerDist();
        setDistStageToTower();
        setBoundR();
        setBoundZ();
        setDistBetweenStages();
        setIncouplingTime();
        setUpperTargetTotalEnergy();
    }

    ///////////////////////////////
    //destructor
    ///////////////////////////////

    ~ZeemanDecel()
    {
        if (activeStageCoils != 0)
        {
            delete[] activeStageCoils;
        }
        if (activeTowerCoils != 0)
        {
```

```
            delete[] activeTowerCoils;
        }
        if (maxStageAbsB != 0)
        {
            delete[] maxStageAbsB;
        }
    }

    ///////////////////////////////
    //setters
    ///////////////////////////////

    /// sets the absolute values of the B−field from the decelerator coils
    inline void setStageCoilB(const math::Grid2d<real_t> bz,
            const math::Grid2d<real_t> br)
    {
        origStageCoilBz = bz;
        origStageCoilBr = br;
        scaleStageCoilB();
    }

    /// sets the absolute values of the B−field from the tower coils
    inline void setTowerCoilB(const math::Grid2d<real_t> bz,
            const math::Grid2d<real_t> br)
    {
        origTowerCoilBz = bz;
        origTowerCoilBr = br;
        scaleTowerCoilB();
    }

    /// sets the current at which the magnetic field of the decelerator
    /// coils has been simulated
    inline void setStageSimCurrent(real_t I = 300)
    {
        Assert<ExWrongInput>(fabs(I)
                        > std::numeric_limits<real_t>::epsilon());
        stageSimCurrent = I;
        scaleStageCoilB();
    }

    /// sets the current at which the magnetic field of the tower
    /// coils has been simulated
    inline void setTowerSimCurrent(real_t I = 300)
    {
        Assert<ExWrongInput>(fabs(I)
                        > std::numeric_limits<real_t>::epsilon());
        towerSimCurrent = I;
        scaleTowerCoilB();
    }

    /// sets the current that is used in the decelerator coils in Ampere
    /// and scales the B field of the stage coils acording to the current
    inline void setStageCurrent(real_t I = 300)
    {
        current = I;
        scaleStageCoilB();
    }

    /// sets the current in the tower coils and scales the B−field
    /// according to this current
    inline void setTowerCurrent(real_t I = 250)
    {
        towercurrent = I;
        scaleTowerCoilB();
    }

    /// sets the pulsing of the coils in a specific stage
    /// stageInd: index of stage, counting begins with 0
    void setStagePulses(size_t stageInd, const math::Matrix<real_t>& pulses)
    {
        Assert<ExWrongInput>(stageInd < numbstages);
        Assert<ExWrongInput>(pulses.rows() == numbstagecoils);
        Assert<ExWrongInput>(pulses.cols() == 2);
        origStagePulses[stageInd] = pulses;
        stagePulses[stageInd] = origStagePulses[stageInd];
```

```
            stagePulses[stageInd].addScalar(incouplingTime);
        }

        /// sets the pulsing of the coils in the towers
        void setTowerPulses(const math::Matrix<real_t>& pulses)
        {
            Assert<ExWrongInput>(pulses.rows() ==
                                 (numbstages-1)*towercoils);
            Assert<ExWrongInput>(pulses.cols() == 2);
            origTowerPulses = pulses;
            towerPulses = origTowerPulses;
            towerPulses.addScalar(incouplingTime);
        }

        /// sets the start-time of the pulse of the j-th decelerator coil
        /// in the i-th stage (counting begins with 0)
        /// including the incoupling time as opposed to the orginal
        /// pulse to which the incoupling time will be added
        void setActualStagePulseStartTime(size_t i, size_t j, real_t t)
        {
            Assert<ExWrongInput>(i < numbstages);
            Assert<ExWrongInput>(j < numbstagecoils);
            stagePulses[i](j,0) = t;
            origStagePulses[i](j,0) = t-incouplingTime;
        }

        /// sets the end-time of the pulse of the j-th decelerator coil
        /// in the i-th stage (counting begins with 0)
        /// including the incoupling time as opposed to the orginal
        /// pulse to which the incoupling time will be added
        void setActualStagePulseEndTime(size_t i, size_t j, real_t t)
        {
            Assert<ExWrongInput>(i < numbstages);
            Assert<ExWrongInput>(j < numbstagecoils);
            stagePulses[i](j,1) = t;
            origStagePulses[i](j,1) = t-incouplingTime;
        }

        /// sets the start-time of the pulse of the j-th tower coil
        /// after the i-th stage (counting begins with 0)
        /// including the incoupling time as opposed to the orginal
        /// pulse to which the incoupling time will be added
        void setActualTowerPulseStartTime(size_t i, size_t j, real_t t)
        {
            Assert<ExWrongInput>(i < numbstages-1);
            Assert<ExWrongInput>(j < towercoils);
            towerPulses(i*towercoils+j,0) = t;
            origTowerPulses(i*towercoils+j,0) = t-incouplingTime;
        }

        /// sets the end-time of the pulse of the j-th tower coil
        /// after the i-th stage (counting begins with 0)
        /// including the incoupling time as opposed to the orginal
        /// pulse to which the incoupling time will be added
        void setActualTowerPulseEndTime(size_t i, size_t j, real_t t)
        {
            Assert<ExWrongInput>(i < numbstages-1);
            Assert<ExWrongInput>(j < towercoils);
            towerPulses(i*towercoils+j,1) = t;
            origTowerPulses(i*towercoils+j,1) = t-incouplingTime;
        }

        /// sets the ramping time B-field of the decelerator coils in us
        inline void setStageRampOn(real_t dt = 8)
        {
            rampOn = dt;
        }

        /// sets the ramping time B-field of the decelerator coils in us
        inline void setStageRampOff(real_t dt = 8)
        {
            rampOff = dt;
        }

        /// sets the switching on time of the tower coils in us
```

```
        inline void setTowerRampOn(real_t dt = 10)
        {
            towerrampOn = dt;
        }

        /// sets the switching off time of the tower coils in us
        inline void setTowerRampOff(real_t dt = 10)
        {
            towerrampOff = dt;
        }

        /// sets number of coils in one decelerator stage
        inline void setNumbStageCoils(size_t n = 12)
        {
            numbstagecoils = n;
            initializeStagePulses();
        }

        /// sets the number of tower coils
        inline void setTowerCoils(size_t n = 2)
        {
            towercoils = n;
            initializeTowerPulses();
        }

        /// sets the number of decelerator stages
        inline void setNumbStages(size_t n = 2)
        {
            numbstages = n;
            initializeStagePulses();
            initializeTowerPulses();
        }

        /// sets the distance to the middle of the first coil in mm
        inline void setDistToMidFirstCoil(real_t d = 198.45)
        {
            disttomidfirstcoil = d;
        }

        /// sets the distance from the middle of one coil to the middle
        /// of the next in mm
        inline void setCoilDist(real_t d = 11)
        {
            coildist = d;
        }

        /// sets the distance between tower coils
        inline void setTowerDist(real_t d = 15)
        {
            towerdist = d;
        }

        /// sets the distance from last decelerator coil to first tower coil
        inline void setDistStageToTower(real_t d = 20)
        {
            diststagetotower = d;
        }

        /// sets an upper bound for the distance to the axis for the particles
        inline void setBoundR(real_t v = 2.5)
        {
            boundr = v;
        }

        /// sets the upper bound for the z-coordinates of the particles
        inline void setBoundZ(real_t d = 551.45)
        {
            boundz = d;
        }

        /// sets the distance between two decelerator stages
        /// (from the middle of the last coil to the middle of the next)
        inline void setDistBetweenStages(real_t d = 55)
        {
            distDecelStages = d;
        }

        /// set the offset which is added to the timing of the pulses
        inline void setIncouplingTime(real_t dt = 0)
        {
            incouplingTime = dt;
            for (size_t i = 0; i<numbstages; ++i)
            {
                stagePulses[i] = origStagePulses[i];
                stagePulses[i].addScalar(incouplingTime);
            }
            towerPulses = origTowerPulses;
            towerPulses.addScalar(incouplingTime);
        }

        /// set the upper bound for the kinetic energy to which the particles
        /// should be decelerated
        inline void setUpperTargetTotalEnergy(real_t kinE =
                std::numeric_limits<real_t>::infinity())
        {
            upperTargetE = kinE;
        }

        ///////////////////////////////
        //getters
        ///////////////////////////////

        /// get the values of the B-field from the decelerator coils
        inline const math::Grid2d<real_t>& getStageCoilBz() const
        {
            return origStageCoilBz;
        }

        inline const math::Grid2d<real_t>& getStageCoilBr() const
        {
            return origStageCoilBr;
        }

        /// get the values of the B-field from the tower coils
        inline const math::Grid2d<real_t>& getTowerCoilBz() const
        {
            return origTowerCoilBz;
        }

        inline const math::Grid2d<real_t>& getTowerCoilBr() const
        {
            return origTowerCoilBr;
        }

        /// get the current that is used in the decelerator coils in Ampere
        inline real_t getStageSimCurrent() const
        {
            return stageSimCurrent;
        }

        /// get the current in the towercoils
        inline real_t getTowerSimCurrent() const
        {
            return towerSimCurrent;
        }

        /// get the current that is used in the decelerator coils in Ampere
        inline real_t getStageCurrent() const
        {
            return current;
        }

        /// get the current in the towercoils
        inline real_t getTowerCurrent() const
        {
            return towercurrent;
        }

        /// get the ramping time B-field of the decelerator coils in us
        /// to switch on
```

```
        inline real_t getStageRampOn() const
        {
            return rampOn;
        }

        /// get the ramping time B-field of the decelerator coils in us
        /// to switch off
        inline real_t getStageRampOff() const
        {
            return rampOff;
        }

        /// get the switching on time of the tower coils
        inline real_t getTowerRampOn() const
        {
            return towerrampOn;
        }

        /// get the switching off time of the tower coils
        inline real_t getTowerRampOff() const
        {
            return towerrampOff;
        }

        /// get the upper bound for the distance to the axis for the particles
        inline real_t getBoundR() const
        {
            return boundr;
        }

        /// get the upper bound for the z-coordinates of the particles
        inline real_t getBoundZ() const
        {
            return boundz;
        }

        /// get the number of decelerator stages
        inline size_t getNumbStages() const
        {
            return numbstages;
        }

        /// number of coils in one decelerator stage
        inline size_t getNumbStageCoils() const
        {
            return numbstagecoils;
        }

        /// get the number of tower coils
        inline size_t getTowerCoils() const
        {
            return towercoils;
        }

        /// get the distance to the middle of the first coil in mm
        inline real_t getDistToMidFirstCoil() const
        {
            return disttomidfirstcoil;
        }

        /// get the distance from the middle of one coil to the middle
        /// of the next in mm
        inline real_t getCoilDist() const
        {
            return coildist;
        }

        /// get the distance between tower coils
        inline real_t getTowerDist() const
        {
            return towerdist;
        }

        /// get the distance from last decelerator coil to first tower coil
        inline real_t getDistStageToTower() const
        {
            return diststagetotower;
        }

        /// get the distance between two decelerator stages
        /// (from the middle of the last coil to the middle of the next)
        inline real_t getDistBetweenStages() const
        {
            return distDecelStages;
        }

        /// get a constant reference to the matrix where the original pulses of
        /// the coils of stage i are stored (pulses without incoupling
        /// time)
        inline const math::Matrix<real_t>& getOrigStagePulses(size_t i) const
        {
            return origStagePulses.at(i);
        }

        /// get a constant reference to the matrix where the pulses of the
        /// coils of stage i are stored (pulses with incoupling time)
        inline const math::Matrix<real_t>& getStagePulses(size_t i) const
        {
            return stagePulses.at(i);
        }

        /// get a constant reference to the matrix where the original pulses of
        /// the coils of the towers are stored (pulses without incoupling
        /// time)
        inline const math::Matrix<real_t>& getOrigTowerPulses() const
        {
            return origTowerPulses;
        }

        /// get a constant reference to the matrix where the pulses of the
        /// coils of the towers are stored (pulses with incoupling time)
        inline const math::Matrix<real_t>& getTowerPulses() const
        {
            return towerPulses;
        }

        /// get the position of the coil c in stage s
        /// Note: numbering here begins with 0
        inline real_t getStageCoilPos(size_t s, size_t c) const
        {
            return disttomidfirstcoil
                   +s*((numbstagecoils−1)*coildist+distDecelStages)+c*coildist;
        }

        /// get the position of the tower coil c after stage s
        /// Note: numbering here begins with 0
        inline real_t getTowerCoilPos(size_t s, size_t c) const
        {
            // determine number of stages before tower coil
            return getStageCoilPos(s,numbstagecoils−1)
                   +diststagetotower+c*towerdist;
        }

        /// get the offset which is added to the timing of the pulses
        inline real_t getIncouplingTime() const
        {
            return incouplingTime;
        }

        /// get the upper bound for the kinetic energy to which the particles
        /// should be decelerated
        inline real_t getUpperTargetTotalEnergy() const
        {
            return upperTargetE;
        }

        ///////////////////////////////
        //other public methods
        ///////////////////////////////

```

```
        /// reads stage pulses from files labeled with the stage number
        void readStagePulsesFiles()
        {
            for (size_t i = 0; i < numbstages; ++i)
            {
                std::ostringstream filename;
                filename << i;
                origStagePulses[i].readFromFile(filename.str().c_str(),2);
                stagePulses[i] = origStagePulses[i];
                stagePulses[i].addScalar(incouplingTime);
            }
        }

        /// writes stage pulses to files labeled with the stage number
        void writeOrigStagePulsesFiles()
        {
            for (size_t i = 0; i < numbstages; ++i)
            {
                std::ostringstream filename;
                filename << i;
                origStagePulses[i].writeToFile(filename.str().c_str());
            }
        }

        /////////////////////////////////////////////////////////
        // methods used by ParticleSim<ZeemanDecel,integrator_t>
        /////////////////////////////////////////////////////////

        /// calculate the value and the gradient of the absolute values
        /// of the field at the given axial and radial coordinates
        inline math::ValGrad<real_t> calculateAbsBField
                (real_t posZ, real_t posR) const
        {
            // initialize a vector of 3 components
            // 1. component: value of the value of the B field component
            // 2. component: absolute value of the gradient of the
            //               absolute value of the B field
            // 3. component: r−component of the gradient of the
            //               absolute value of the B field
            math::ValGrad<real_t> Bz(0.0,0.0,0.0);
            math::ValGrad<real_t> Br(0.0,0.0,0.0);
            // loop over all active coils
            for (size_t i = 0; i < nActiveStageCoils; ++i)
            {
                // component−wise summation
                stageCoilBz.linpValueAndGradient(
                    posZ−activeStageCoils[i].posZ,
                    posR,
                    activeStageCoils[i].rampfactor,
                    Bz);
                stageCoilBr.linpValueAndGradient(
                    posZ−activeStageCoils[i].posZ,
                    posR,
                    activeStageCoils[i].rampfactor,
                    Br);
            }
            for (size_t i = 0; i < nActiveTowerCoils; ++i)
            {
                // component−wise summation
                towerCoilBz.linpValueAndGradient(
                    posZ−activeTowerCoils[i].posZ,
                    posR,
                    activeTowerCoils[i].rampfactor,
                    Bz);
                towerCoilBr.linpValueAndGradient(
                    posZ−activeTowerCoils[i].posZ,
                    posR,
                    activeTowerCoils[i].rampfactor,
                    Br);
            }
            return calculateAbsBFieldFromComponents(Bz,Br);
        }

        /// determine the number of timesteps until the next event
        /// event = switch on or switch off of a coil)
```

```
    size_t timestepsUntilNextEvent(real_t time, real_t timestep,
                                   real_t endtime) const
    {
        // minimal positive time difference
        // for the start and end time of a coil
        // initialize both with the maximal possible time
        int min_timesteps = (int)ceil((endtime-time)/timestep);
        // determine minimum end time of active stage coils
        for (size_t i = 0; i < nActiveStageCoils; ++i)
        {
            // round down end times
            int ts = (int)floor((activeStageCoils[i].endTime+rampOff-time)/times
tep);
            if (ts > 0 && ts < min_timesteps)
            {
                min_timesteps = ts;       // new minimum
            }
        }
        // determine minimum end time of active tower coils
        for (size_t i = 0; i < nActiveTowerCoils; ++i)
        {
            // round down end times
            int ts = (int)floor((activeTowerCoils[i].endTime+towerrampOff-time)/
timestep);
            if (ts > 0 && ts < min_timesteps)
            {
                min_timesteps = ts;       // new minimum
            }
        }
        // determine minimum start time of inactive stage coils
        for (size_t i = 0; i < numbstages; ++i)
        {
            for (size_t j = 0; j < numbstagecoils; ++j)
            {
                // round down start times
                int ts = (int)floor((stagePulses[i](j,0)-time)/timestep)+1;
                if (ts > 0 && ts < min_timesteps)
                {
                    min_timesteps = ts;   // new minimum
                }
            }
        }
        // determine minimum start time of inactive tower coils
        for (size_t i = 0; i < (numbstages-1)*towercoils; ++i)
        {
            // round down start times
            int ts = (int)floor((towerPulses(i,0)-time)/timestep)+1;
            if (ts > 0 && ts < min_timesteps)
            {
                min_timesteps = ts;       // new minimum
            }
        }

        return min_timesteps >= 0 ? (size_t)min_timesteps : 0;
    }

    /// prepare timestepping (this function is called by
    /// ParticleSim<ZeemanDecel,integrator_t> at each event before
    /// the actual timestepping is done)
    ///
    /// this inserts the active coils into the lists of active coils
    inline void prepareTimestepping(real_t time, real_t timestep)
    {
        insertActiveStageCoils(time,timestep);
        insertActiveTowerCoils(time,timestep);
#ifdef __DEBUG__
        debugOutputAtEvent(time);
#endif
    }

    /// prepare the magnetic field (e.g. calculate the scaling factors)
    inline void setTime(real_t time, real_t timestep)
    {
        // calculate the rampfactor for each coil
        calculateRamp(time);
```

```
    }

    /// check whether the position (z,r) is inside the boundaries
    inline bool isPointInsideBoundaries(real_t z, real_t r) const
    {
        bool result = false;
        if (r < boundr && z < boundz)
        {
            result = true;
        }
        return result;
    }

    /// check whether a low-field-seeking particle still can be accepted
    /// Assuming that the coils will be pulsed one after another,
    /// this can be done by imposing the constraint that the particles
    /// longitudinal position is not larger than the centre of the coil
    /// that will follow after the foremost currently pulsed coil (if any).
    inline bool mayLFSParticleBeAccepted(real_t time, const Particle& p,
            const ParticlePopulation& pop) const
    {
        bool result = true;
        if (p.vel.getZ()<0) {
            result = false;
        } else if (nActiveStageCoils > 0 && p.pos.getZ() >
                (activeStageCoils[nActiveStageCoils-1].posZ+coildist)) {
            result = false;
        }
        if (upperTargetE < std::numeric_limits<real_t>::infinity()) {
            if (!mayLFSParticleBeDeceleratedTo(time, p, pop,
                    upperTargetE)) {
                result = false;
            }
        }
        return result;
    }

    /// check whether a low-field-seeking particle can be decelerated to
    /// a kinetic energy which is at most a given value
    inline bool mayLFSParticleBeDeceleratedTo(real_t time,
            const Particle& p, const ParticlePopulation& pop,
            real_t upperE) const
    {
        real_t totalE = pop.calculateKineticEnergy(p.vel.norm2())
                + pop.calculateZeemanEnergy(
                        calculateAbsBField(p.pos.getZ(),p.posR()).val);
        // calculate the velocity that corresponds to the case where all
        // total energy is kinetic energy
        real_t maxVel = sqrt(2*totalE/pop.getMass());
        if (totalE <= upperE) {
            return true;
        }
        real_t partPosZ = p.pos.getZ();
        for (size_t i = 0; i<numbstages; ++i) {
            for (size_t j = 0; j<numbstagecoils; ++j) {
                real_t Dt = stagePulses[i](j,1) + rampOff - time;
                if (Dt >= 0) {
                    real_t coilPosZ = getStageCoilPos(i,j);
                    if (partPosZ < coilPosZ) {
                        // calculate the position along the z-axis after
                        // a time interval Dt assuming that all the
                        // particles total energy is kinetic energy of
                        // motion into longitudinal direction
                        real_t newPartPosZ = partPosZ + Dt*maxVel;
                        real_t maxB = 0;
                        if (newPartPosZ > coilPosZ) {
                            // set the maximal experienced B-Field to the
                            // value at the centre of the coil
                            maxB = maxStageAbsB[nMaxStageAbsB-1];
                        } else if (newPartPosZ-coilPosZ
                                > stageCoilBz.getMinDim1()) {
                            // set the maximal experienced B-Field to the
                            // value at the new (pretended) position of
                            // the particle
                            size_t ind = (size_t)ceil((newPartPosZ-coilPosZ
                                        −stageCoilBz.getMinDim1())
                                        /stageCoilBz.dim1Dist())−1;
                                maxB = maxStageAbsB[ind];
                            }
                            // subtract the potential energy the
                            // particle would have with the maximal
                            // experienced B−field
                            totalE = totalE − pop.calculateZeemanEnergy(maxB);
                            if (totalE<=upperE) {
                                return true;
                            }
                        }
                    }
                }
            }
            return (totalE<=upperE);
        }

        /// returns true if at least one coil is active
        inline bool isActive() const
        {
            return (nActiveStageCoils > 0 || nActiveTowerCoils > 0);
        }

    private:

        ////////////////////////////////////////////////////////
        // private types and datastructures
        ////////////////////////////////////////////////////////

        //////////////////////////////
        ///
        /// \class Coil
        ///
        /// \brief simple data structure that represents active coils
        /// in the decelerator
        ///
        //////////////////////////////
        class Coil
        {
        public:

            /// default constructor
            Coil() { }

            /// constructor
            Coil(real_t z, real_t st, real_t et)
                    : posZ(z), startTime(st), endTime(et) { }

            /// position of the grid point with the lowest z−coordinate)
            real_t posZ;
            /// starting time (before ramp)
            real_t startTime;
            /// end time (after ramp)
            real_t endTime;
            /// factor to which the field is scaled during switch off and
            /// switch on
            real_t rampfactor;
        };

        ///////////////////////////////////////////////////
        //definition of variables (properties of the class)
        ///////////////////////////////////////////////////

        /// B−field of the decelerator coils unscaled
        math::Grid2d<real_t> origStageCoilBz;
        math::Grid2d<real_t> origStageCoilBr;
        /// B−field of the tower coils unscaled
        math::Grid2d<real_t> origTowerCoilBr;
        math::Grid2d<real_t> origTowerCoilBz;
        /// B−field of the decelerator coils
        /// scaled according to the ratio current/stageSimCurrent
        math::Grid2d<real_t> stageCoilBz;
        math::Grid2d<real_t> stageCoilBr;
        /// B−field of the decelerator coils
```

```
        /// scaled according to the ratio towerCurrent/towerSimCurrent
        math::Grid2d<real_t> towerCoilBz;
        math::Grid2d<real_t> towerCoilBr;
        /// the current at which the magnetic field of the decelerator
        /// coils has been simulated
        real_t stageSimCurrent;
        /// the current at which the magnetic field of the tower
        /// coils has been simulated
        real_t towerSimCurrent;
        /// original pulsing of the coils in each stage
        /// (without incoupling time)
        std::vector< math::Matrix<real_t> > origStagePulses;
        /// pulsing of the coils in each stage
        std::vector< math::Matrix<real_t> > stagePulses;
        /// original pulsing of the tower coils
        /// (without incoupling time)
        math::Matrix<real_t> origTowerPulses;
        /// pulsing of the tower coils
        math::Matrix<real_t> towerPulses;
        /// array of active stage coils
        Coil* activeStageCoils;
        /// number of active stage coils
        size_t nActiveStageCoils;
        /// array of active tower coils
        Coil* activeTowerCoils;
        /// number of active tower coils
        size_t nActiveTowerCoils;
        /// current that is used in the decelerator coils in Ampere
        real_t current;
        /// current in the towercoils
        real_t towercurrent;
        /// ramping time B−field of the decelerator coils in us to switch on
        real_t rampOn;
        /// ramping time B−field of the decelerator coils in us to switch off
        real_t rampOff;
        /// switching on time of the tower coils
        real_t towerrampOn;
        /// switching off time of the tower coils
        real_t towerrampOff;
        /// the upper bound for the distance to the axis for the particles
        real_t boundr;
        /// the upper bound for the z−coordinates of the particles
        real_t boundz;
        /// the number of decelerator stages
        size_t numbstages;
        /// number of coils in one decelerator stage
        size_t numbstagecoils;
        /// number of coils in the tower
        size_t towercoils;
        /// distance from the middle of one coil to the middle of the next in mm
        real_t coildist;
        /// distance between tower coils
        real_t towerdist;
        /// distance to the middle of the first coil in mm
        real_t disttomidfirstcoil;
        /// distance between two decelerator stages (from the middle of
        /// the last coil to the middle of the next)
        real_t distDecelStages;
        /// distance from last decelerator coil to first towercoil
        real_t diststagetotower;
        /// the time offset which is added to the pulsing of the coils)
        real_t incouplingTime;
        /// an array that contains the maximal absolute B−field values
        /// along the z−axis \of the stage coils
        real_t* maxStageAbsB;
        /// the number of elements of the array maxStageAbsB
        size_t nMaxStageAbsB;
        /// upper bound for the kinetic energy to which the particles
        /// should be decelerated
        real_t upperTargetE;

        ///////////////////////////////
        //private methods
        ///////////////////////////////
```

```
    /// calculate the rampfactor for each coil at that time
    inline void calculateRamp(real_t time)
    {
        for (size_t i = 0; i < nActiveStageCoils; ++i)
        {
            // rampfactor of stage coil
            activeStageCoils[i].rampfactor = math::min(1.0,(time-activeStageCoil
s[i].startTime)/rampOn)*math::min(1.0,(activeStageCoils[i].endTime+rampOff-time)
/rampOff);
        }
        for (size_t i = 0; i < nActiveTowerCoils; ++i)
        {
            // rampfactor of tower coil
            // Note: max is needed to assure that it is
            // non negative
            activeTowerCoils[i].rampfactor = math::min(1.0,(time-activeTowerCoil
s[i].startTime)/towerrampOn)*math::min(1.0,(activeTowerCoils[i].endTime+towerram
pOff-time)/towerrampOff);
        }
    }

    /// inserts stage coils that are active at that time into the list of
    /// active coils
    void insertActiveStageCoils(real_t time, real_t dt)
    {
        if (activeStageCoils == 0)
        {
            // if necessary, reserve some memory
            activeStageCoils =
                new Coil[numbstages*numbstagecoils];
        }
        size_t k = 0;
        for (size_t i = 0; i<numbstages; ++i)
        {
            for (size_t j = 0; j<numbstagecoils; ++j)
            {
                // insert coil if it has to run during the next timestep
                if (stagePulses[i](j,0) < time &&
                        time < stagePulses[i](j,1)+rampOff-dt)
                {
                    activeStageCoils[k].posZ =
                        getStageCoilPos(i,j)
                        -stageCoilBz.getCenterDim1();
                    activeStageCoils[k].startTime =
                        stagePulses[i](j,0);
                    activeStageCoils[k].endTime =
                        stagePulses[i](j,1);
#ifdef __DEBUG__
                    debugOutputAtStageCoil(k);
#endif
                    ++k;
                }
            }
        }
        nActiveStageCoils = k;
    }

    /// inserts tower coils that are active at that time into the list of
    /// active coils
    void insertActiveTowerCoils(real_t time, real_t dt)
    {
        if (activeTowerCoils == 0)
        {
            // if necessary, reserve some memory
            activeTowerCoils =
                new Coil[(numbstages-1)*towercoils];
        }
        size_t k = 0;
        for (size_t i = 0; i<numbstages-1; ++i)
        {
            for (size_t j = 0; j<towercoils; ++j)
            {
                // insert coil if it has to run during the next timestep
                if (towerPulses(i*towercoils+j,0) < time &&
                        time < towerPulses(i*towercoils+j,1)+towerrampOff-dt)
```

```
                {
                    activeTowerCoils[k].posZ =
                        getTowerCoilPos(i,j)
                        -towerCoilBz.getCenterDim1();
                    activeTowerCoils[k].startTime =
                        towerPulses(i*towercoils+j,0);
                    activeTowerCoils[k].endTime =
                        towerPulses(i*towercoils+j,1);
#ifdef __DEBUG__
                    debugOutputAtTowerCoil(k);
#endif
                    ++k;
                }
            }
        }
        nActiveTowerCoils = k;
    }

    /// scale the B field of the decelerator coils according to the current
    inline void scaleStageCoilB()
    {
        stageCoilBz = origStageCoilBz;
        stageCoilBr = origStageCoilBr;
        stageCoilBz.scaleValues(current/stageSimCurrent);
        stageCoilBr.scaleValues(current/stageSimCurrent);
        storeMaxStageCoilB();
    }

    /// store the maximum absolute B-field as a function of z
    inline void storeMaxStageCoilB()
    {
        // delete old array if necessary
        if (maxStageAbsB != 0)
        {
            delete[] maxStageAbsB;
        }
        nMaxStageAbsB = (size_t)ceil((stageCoilBz.getCenterDim1()
                -stageCoilBz.getMinDim1())
                /stageCoilBz.dim1Dist());
        // allocate memory for the array
        maxStageAbsB = new real_t[nMaxStageAbsB];
        for(size_t i = 0; i < nMaxStageAbsB; ++i) {
            size_t ncols = stageCoilBz.getRepCols();
            real_t maxB = 0;
            for(size_t j = 0; j < ncols; ++j) {
                real_t Bz = stageCoilBz.getRepElem(i,j);
                real_t Br = stageCoilBr.getRepElem(i,j);
                real_t B = sqrt(Bz*Bz + Br*Br);
                if (B > maxB) {
                    maxB = B;
                }
            }
            maxStageAbsB[i] = maxB;
        }
    }

    /// scale the B field of the tower coils according to the current
    inline void scaleTowerCoilB()
    {
        towerCoilBz = origTowerCoilBz;
        towerCoilBr = origTowerCoilBr;
        towerCoilBz.scaleValues(towercurrent/towerSimCurrent);
        towerCoilBr.scaleValues(towercurrent/towerSimCurrent);
    }

    /// if wanted, output some debugging information at each event
    void inline debugOutputAtEvent(real_t time) const
    {
        std::cerr << "-- Pulse at t = " << time << " us --\n"
                  << "# active stage coils: "
                  << nActiveStageCoils
                  << std::endl
                  << "# active tower coils: "
                  << nActiveTowerCoils
                  << std::endl;
    }

    /// if wanted, output some debugging information at each pulse
    void inline debugOutputAtStageCoil(size_t k) const
    {
        std::cerr << "−− active stage coil "
                  << k << " −−\n"
                  << "posZ = " << activeStageCoils[k].posZ
                  << std::endl
                  << "startTime = " << activeStageCoils[k].startTime
                  << std::endl
                  << "endTime = " << activeStageCoils[k].endTime
                  << std::endl
                  << std::endl;
    }

    /// if wanted, output some debugging information at tower coils
    void inline debugOutputAtTowerCoil(size_t k) const
    {
        std::cerr << "−− active tower coil "
                  << k << " −−\n"
                  << "posZ = " << activeTowerCoils[k].posZ
                  << std::endl
                  << "startTime = " << activeTowerCoils[k].startTime
                  << std::endl
                  << "endTime = " << activeTowerCoils[k].endTime
                  << std::endl
                  << std::endl;
    }

    // initialize stage pulses
    void initializeStagePulses()
    {
        origStagePulses.resize(0);
        origStagePulses.resize(numbstages,math::Matrix<real_t>(numbstagecoils,2)
);
        stagePulses.resize(0);
        stagePulses.resize(numbstages,math::Matrix<real_t>(numbstagecoils,2));
    }

    // initialize tower pulses
    void initializeTowerPulses()
    {
        origTowerPulses.resize(towercoils*(numbstages−1),2);
        towerPulses.resize(towercoils*(numbstages−1),2);
    }
};

}
#endif
```

```
#Makefile: 2009-04-08, Yves Salathe

#####################################################################
# global settings                                                   #
#####################################################################

# common libraries
LIBS=#-lpthread -lnagc_nag
# general compiler flags
CXXFLAGS=$(LIBS) -Wall # -D__SAVE__# -D__TRAJECTORIES__ -D__NO_DETECT__  #-D__2D
_SIMULATION__
# compiler flags for optimization
OPTFLAGS=-O3
# compiler flags for optimization for icpc
#OPTFLAGS_ICPC=-ipo -O3 -no-prec-div -static -xHost
OPTFLAGS_ICPC=-fast
# compiler flags for debugging
DFLAGS=-ggdb -D__DEBUG__ #-D__REFERENCE_PARTICLE__
# compiler flags for debugging for icpc
#DFLAGS_ICPC=-ggdb -debug inline-debug-info #-D__DEBUG__
# compiler flags for icpc
CXXFLAGS_ICPC=-wd981 $(DFLAGS_ICPC) #-Weffc++
# compiler flags for profiling
PFLAGS=-pg
# compilers
CXX=g++
CXX_ICPC=icpc


####################################################################
# header files that are common to all simulations which use       #
# ParticleSim                                                      #
####################################################################
PARTICLESIM_HEADERS=headers/particlesim.h \
                headers/engine.h \
                headers/particle.h \
                headers/particle_population.h \
                headers/assert.h \
                headers/basic_math.h \
                headers/matrix.h \
                headers/grid.h \
                headers/valgrad.h \
                headers/vector3d.h \
                headers/io.h \
                headers/observer.h \
                headers/laser_detector.h \
                headers/trajectories_observer.h \
                headers/decelpulsegen_observer.h \
                headers/integrator.h \
                headers/input.h \
                headers/symplectic_euler_integrator.h


####################################################################
# all executables                                                  #
####################################################################

EXECS=longdecel longdecel_debug longdecel_icpc \
        longdecel_trap longdecel_trap_debug longdecel_trap_icpc \
        longdecel_cma_icpc

.PHONY: all
all: $(EXECS)


####################################################################
# decelerator simulation                                           #
####################################################################
LONGDECEL_HEADERS=$(PARTICLESIM_HEADERS) \
                headers/zeemandecel_builder.h headers/zeemandecel.h

longdecel: longdecel.cxx $(LONGDECEL_HEADERS)
        $(CXX) $(CXXFLAGS) $(OPTFLAGS) longdecel.cxx -o longdecel

longdecel_debug: longdecel.cxx $(LONGDECEL_HEADERS)
```

```
        $(CXX) $(CXXFLAGS) $(DFLAGS) $(PFLAGS) longdecel.cxx -o longdecel_debug

longdecel_icpc: longdecel.cxx $(LONGDECEL_HEADERS)
        $(CXX_ICPC) $(CXXFLAGS_ICPC) $(CXXFLAGS) $(OPTFLAGS_ICPC) longdecel.cxx
-o longdecel_icpc &> icpc_output.txt

#longdecel_icpc_debug: longdecel.cxx $(LONGDECEL_HEADERS)
#       $(CXX_ICPC) $(CXXFLAGS_ICPC) $(CXXFLAGS) $(OPTFLAGS_ICPC) $(DFLAGS_ICPC)
 longdecel.cxx -o longdecel_icpc_debug &> icpc_debug_output.txt


#longdecel_traj_icpc: longdecel.cxx $(LONGDECEL_HEADERS)
#       $(CXX_ICPC) $(CXXFLAGS_ICPC) $(CXXFLAGS) $(OPTFLAGS_ICPC) \
#                       -D __TRAJECTORIES__ -D __NO_DETECT__ longdecel.cxx \
#                       -o longdecel_traj_icpc &> icpc_traj_output.txt

#####################################################################
# decelerator combined with trap simulation                         #
#####################################################################
LONGDECEL_TRAP_HEADERS=$(LONGDECEL_HEADERS) \
        headers/magnetictrap_builder.h headers/magnetictrap.h \
        headers/coil.h headers/polynomial.h headers/region_observer.h \
        headers/trappulsegen_observer.h

longdecel_trap: longdecel.cxx $(LONGDECEL_TRAP_HEADERS)
        $(CXX) $(CXXFLAGS) $(OPTFLAGS) -D __TRAP__ longdecel.cxx -o longdecel_tr
ap

longdecel_trap_debug: longdecel.cxx $(LONGDECEL_TRAP_HEADERS)
        $(CXX) $(CXXFLAGS) $(DFLAGS) $(PFLAGS) -D __TRAP__ longdecel.cxx -o long
decel_trap_debug

longdecel_trap_icpc: longdecel.cxx $(LONGDECEL_TRAP_HEADERS)
        $(CXX_ICPC) $(CXXFLAGS_ICPC) $(CXXFLAGS) $(OPTFLAGS_ICPC) -D __TRAP__ lo
ngdecel.cxx -o longdecel_trap_icpc &> icpc_trap_output.txt

#longdecel_trap_icpc_debug: longdecel.cxx $(LONGDECEL_TRAP_HEADERS)
#       $(CXX_ICPC) $(CXXFLAGS_ICPC) $(CXXFLAGS) $(OPTFLAGS_ICPC) $(DFLAGS_ICPC)
 -D __TRAP__ longdecel.cxx -o longdecel_trap_icpc_debug &> icpc_trap_debug_outpu
t.txt

#longdecel_trap_traj_icpc: longdecel.cxx $(LONGDECEL_TRAP_HEADERS)
#       $(CXX_ICPC) $(CXXFLAGS_ICPC) $(CXXFLAGS) $(OPTFLAGS_ICPC) \
#               -D __TRAP__ -D __TRAJECTORIES__ -D __NO_DETECT__ \
#               longdecel.cxx -o longdecel_trap_traj_icpc \
#               &> icpc_trap_traj_output.txt

#####################################################################
# optimization using CMA                                            #
#####################################################################
LONGDECEL_CMA_OBJS = longdecel_cma.o cmaes.o objective_trap.o \
                objective_hfs.o
LONGDECEL_CMA_HEADERS =cma/cmaes_interface.h \
                headers/objective_trap.h headers/objective_hfs.h

longdecel_cma_icpc: $(LONGDECEL_CMA_OBJS)
        $(CXX_ICPC) $(CXXFLAGS_ICPC) $(CXXFLAGS) $(OPTFLAGS_ICPC) \
        -lm -openmp $(LONGDECEL_CMA_OBJS) -o longdecel_cma_icpc

longdecel_cma.o: longdecel_cma.cxx $(LONGDECEL_CMA_HEADERS)
        $(CXX_ICPC) $(CXXFLAGS_ICPC) $(CXXFLAGS) $(OPTFLAGS_ICPC) \
                -ansi -openmp -c longdecel_cma.cxx \
                &> icpc_longdecel_cma_output.txt

cmaes.o: cma/cmaes.c cma/cmaes.h cma/cmaes_interface.h
        $(CXX_ICPC) $(CXXFLAGS_ICPC) $(CXXFLAGS) $(OPTFLAGS_ICPC) \
        -ansi -c cma/cmaes.c &> cma/icpc_cmaes_output.txt

objective_trap.o: objective_trap.cxx headers/objective_trap.h \
                $(LONGDECEL_TRAP_HEADERS)
        $(CXX_ICPC) $(CXXFLAGS_ICPC) $(CXXFLAGS) $(OPTFLAGS_ICPC) \
        -openmp -c objective_trap.cxx &> icpc_objective_trap_output.txt

objective_hfs.o: objective_hfs.cxx headers/objective_hfs.h
        $(CXX_ICPC) $(CXXFLAGS_ICPC) $(CXXFLAGS) $(OPTFLAGS_ICPC) \
        -openmp -c objective_hfs.cxx &> icpc_objective_hfs_output.txt

#################################################
# documentation                                 #
#################################################

DOCUMENTATION=doc
DOXYGEN=doxygen
DOXYGEN_CONF=doxygen.conf
# compilers
.PHONY: doc
doc: $(DOXYGEN_CONF)
        $(DOXYGEN) $(DOXYGEN_CONF)


#################################################
# general utilities                             #
#################################################

.PHONY: clean
clean:
        rm -f $(LONGDECEL_OBJS) $(LONGDECEL_DEBUG_OBJS) \
        $(LONGDECEL_ICPC_OBJS) $(EXECS) $(LONGDECEL_ICPC_DEBUG_OBJS) \
        $(LONGDECEL_CMA_OBJS)
```

Appendix D

Source code of the CMA-ES implementation

On the following pages, the implementation of the CMA-ES optimization algorithm by N. Hansen [36], which has been used in this work, is presented.

```c
/* --------------------------------------------------------- */
/* --- File: cmaes_interface.h - Author: Nikolaus Hansen --- */
/* ---------------------- last modified:  IV 2007        --- */
/* --------------------------------- by: Nikolaus Hansen --- */
/* --------------------------------------------------------- */
/*
     CMA-ES for non-linear function minimization.

     Copyright (C) 1996, 2003, 2007 Nikolaus Hansen.
     e-mail: hansen AT bionik.tu-berlin.de
             hansen AT lri.fr

     License: see file cmaes.c
*/
#include "cmaes.h"

/* --------------------------------------------------------- */
/* ------------------ Interface ---------------------------- */
/* --------------------------------------------------------- */

/* --- initialization, constructors, destructors --- */
double * cmaes_init(cmaes_t *, int dimension , double *xstart,
                double *stddev, long seed, int lambda,
                unsigned int seed, const char *input_parameter_filename);
void cmaes_resume_distribution(cmaes_t *evo_ptr, char *filename);
void cmaes_exit(cmaes_t *);

/* --- core functions --- */
double * const * cmaes_SamplePopulation(cmaes_t *);
double *         cmaes_UpdateDistribution(cmaes_t *,
                                          const double *rgFitnessValues);
const char *     cmaes_TestForTermination(cmaes_t *);

/* --- additional functions --- */
double * const * cmaes_ReSampleSingle( cmaes_t *t, int index);
double const *   cmaes_ReSampleSingle_old(cmaes_t *, double *rgx);
double *         cmaes_SampleSingleInto( cmaes_t *t, double *rgx);
void             cmaes_UpdateEigensystem(cmaes_t *, int flgforce);

/* --- getter functions --- */
double         cmaes_Get(cmaes_t *, char const *keyword);
const double * cmaes_GetPtr(cmaes_t *, char const *keyword); /* e.g. "xbestever"
 */
double *       cmaes_GetNew( cmaes_t *t, char const *keyword);
double *       cmaes_GetInto( cmaes_t *t, char const *keyword, double *mem);

/* --- online control and output --- */
void           cmaes_ReadSignals(cmaes_t *, char const *filename);
void           cmaes_WriteToFile(cmaes_t *, const char *szKeyWord,
                                 const char *output_filename);
char *         cmaes_SayHello(cmaes_t *);
/* --- misc --- */
double *       cmaes_NewDouble(int n);
void           cmaes_FATAL(char const *s1, char const *s2, char const *s3,
                           char const *s4);
```

```c
/* --------------------------------------------------------- */
/* --- File: cmaes.h ----------- Author: Nikolaus Hansen --- */
/* ---------------------- last modified: VIII 2007       --- */
/* --------------------------------- by: Nikolaus Hansen --- */
/* --------------------------------------------------------- */
/*
     CMA-ES for non-linear function minimization.

     Copyright (C) 1996, 2003-2007  Nikolaus Hansen.
     e-mail: hansen@bionik.tu-berlin.de

     License: see file cmaes.c

*/
#ifndef NH_cmaes_h /* only include ones */
#define NH_cmaes_h

#include <time.h>

typedef struct
/* random_t
 * sets up a pseudo random number generator instance
 */
{
  /* Variables for Uniform() */
  long int startseed;
  long int aktseed;
  long int aktrand;
  long int *rgrand;

  /* Variables for Gauss() */
  short flgstored;
  double hold;
} random_t;

typedef struct
/* timings_t
 * time measurement, used to time eigendecomposition
 */
{
  /* for outside use */
  double totaltime;
  double tictoctime;
  double lasttictoctime;

  /* local fields */
  clock_t lastclock;
  time_t lasttime;
  clock_t ticclock;
  time_t tictime;
  short istic;
  short isstarted;

  double lastdiff;
  double tictoczwischensumme;
} timings_t;

typedef struct
/* readpara_t
 * collects all parameters, in particular those that are read from
 * a file before to start. This should split in future?
 */
{
  /* input parameter */
  int N; /* problem dimension, must stay constant */
  unsigned int seed;
  double * xstart;
  double * typicalX;
  int typicalXcase;
  double * rgInitialStds;
  double * rgDiffMinChange;

  /* termination parameters */
  double stopMaxFunEvals;
  double facmaxeval;
```

```c
  double stopMaxIter;
  struct { int flg; double val; } stStopFitness;
  double stopTolFun;
  double stopTolFunHist;
  double stopTolX;
  double stopTolUpXFactor;

  /* internal evolution strategy parameters */
  int lambda;          /* -> mu, <- N */
  int mu;              /* -> weights, (lambda) */
  double mucov, mueff; /* <- weights */
  double *weights;     /* <- mu, -> mueff, mucov, ccov */
  double damps;        /* <- cs, maxeval, lambda */
  double cs;           /* -> damps, <- N */
  double ccumcov;      /* <- N */
  double ccov;         /* <- mucov, <- N */
  struct { int flgalways; double modulo; double maxtime; } updateCmode;
  double facupdateCmode;

  /* supplementary variables */

  char *weigkey;
  char resumefile[99];
  char **rgsformat;
  void **rgpadr;
  char **rgskeyar;
  double ***rgp2adr;
  int n1para, n1outpara;
  int n2para;
} readpara_t;

typedef struct
/* cmaes_t
 * CMA-ES "object"
 */
{
  char *version;
  readpara_t sp;
  random_t rand; /* random number generator */

  double sigma;  /* step size */

  double *rgxmean;  /* mean x vector, "parent" */
  double *rgxbestever;
  double **rgrgx;   /* range of x-vectors, lambda offspring */
  int *index;       /* sorting index of sample pop. */
  double *arFuncValueHist;

  short flgIniphase; /* not really in use anymore */
  short flgStop;

  double chiN;
  double **C;  /* lower triangular matrix: i>=j for C[i][j] */
  double **B;  /* matrix with normalize eigenvectors in columns */
  double *rgD; /* axis lengths */

  double *rgpc;
  double *rgps;
  double *rgxold;
  double *rgout;
  double *rgBDz;   /* for B*D*z */
  double *rgdTmp;  /* temporary (random) vector used in different places */
  double *rgFuncValue;
  double *publicFitness; /* returned by cmaes_init() */

  double gen; /* Generation number */
  double countevals;
  double state; /* 1 == sampled, 2 == not in use anymore, 3 == updated */

  double maxdiagC; /* repeatedly used for output */
  double mindiagC;
  double maxEW;
  double minEW;

  char sOutString[330]; /* 4x80 */
```

```
    short flgEigensysIsUptodate;
    short flgCheckEigen; /* control via signals.par */
    double genOfEigensysUpdate;
    timings_t eigenTimings;

    double dMaxSignifKond;
    double dLastMinEWgroesserNull;

    short flgresumedone;

    time_t printtime;
    time_t writetime; /* ideally should keep track for each output file */
    time_t firstwritetime;
    time_t firstprinttime;

  } cmaes_t;


#endif
```

```
/* −−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−− */
/* −−− File: cmaes.c  −−−−−−−− Author: Nikolaus Hansen   −−− */
/* −−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−− */
/*
     CMA−ES for non−linear function minimization.

     Copyright 1996, 2003, 2007 Nikolaus Hansen
     e−mail: hansen .AT. bionik.tu−berlin.de
             hansen .AT. lri.fr

    This program is free software; you can redistribute it and/or modify
    it under the terms of the GNU General Public License, version 3,
    as published by the Free Software Foundation.

    This program is distributed in the hope that it will be useful,
    but WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
    GNU General Public License for more details.

    You should have received a copy of the GNU General Public License
    along with this program.  If not, see <http://www.gnu.org/licenses/>.

*/
/* −−− Changes : −−−
  03/03/21: argument const double *rgFunVal of
            cmaes_ReestimateDistribution() was treated incorrectly.
  03/03/29: restart via cmaes_resume_distribution() implemented.
  03/03/30: Always max std dev / largest axis is printed first.
  03/08/30: Damping is adjusted for large mueff.
  03/10/30: Damping is adjusted for large mueff always.
  04/04/22: Cumulation time and damping for step size adjusted.
            No iniphase but conditional update of pc.
  05/03/15: in ccov−setting mucov replaced by mueff.
  05/10/05: revise comment on resampling in example.c
  05/10/13: output of "coorstddev" changed from sigma * C[i][i]
            to correct sigma * sqrt(C[i][i]).
  05/11/09: Numerical problems are not anymore handled by increasing
            sigma, but lead to satisfy a stopping criterion in
            cmaes_Test().
  05/11/09: Update of eigensystem and test for numerical problems
            moved right before sampling.
  06/02/24: Non−ansi array definitions replaced (thanks to Marc
            Toussaint).
  06/02/25: Overflow in time measurement for runs longer than
            2100 seconds. This could lead to stalling the
            covariance matrix update for long periods.
            Time measurement completely rewritten.
  06/02/26: Included population size lambda as parameter to
            cmaes_init (thanks to MT).
  06/02/26: Allow no initial reading/writing of parameters via
            "non" and "writeonly" keywords for input parameter
            filename in cmaes_init.
  06/02/27: Optimized code regarding time spent in updating the
            covariance matrix in function Adapt_C2().
  07/08/03: clean up and implementation of an exhaustive test
            of the eigendecomposition (via #ifdef for now)
  07/08/04: writing of output improved
  07/08/xx: termination criteria revised and more added,
            damp replaced by damps=damp*cs, documentation improved.
            Interface significantly changed, evaluateSample function
            and therefore the function pointer argument removed.
            Renaming of functions in accordance with Java code.
            Clean up of parameter names, mainly in accordance with
            Matlab conventions. Most termination criteria can be
            changed online now. Many more small changes, but not in
            the core procedure.
  07/10/29: ReSampleSingle() got a better interface. ReSampleSingle()
            is now ReSampleSingle_old only for backward
            compatibility. Also fixed incorrect documentation. The new
            function SampleSingleInto() has an interface similar to
            the old ReSampleSingle(), but is not really necessary.
  07/11/20: bug: stopMaxIter did not translate into the correct default
            value but into −1 as default. This lead to a too large
            damps and the termination test became true from the first
            iteration. (Thanks to Michael Calonder)
```

```
  07/11/20: new default stopTolFunHist = 1e−13;  (instead of zero)


  Wish List

    o as writing time is measure for all files at once, the display
      cannot be independently written to a file via signals.par, while
      this would be desirable.

    o clean up sorting of eigenvalues and vectors which is done repeatedly.

    o either use cmaes_Get() in cmaes_WriteToFilePtr(): revise the
      cmaes_write that all keywords available with get and getptr are
      recognized. Also revise the keywords, keeping backward
      compatibility. (not only) for this it would be useful to find a
      way how cmaes_Get() signals an unrecognized keyword. For GetPtr
      it can return NULL.

    o or break cmaes_Get() into single getter functions, being a nicer
      interface, and compile instead of runtime error, and faster. For
      file signals.par it does not help.

    o writing data depending on timing in a smarter way, e.g. using 10%
      of all time. First find out whether clock() is useful for measuring
      disc writing time and then timings_t class can be utilized.
      For very large dimension the default of 1 seconds waiting might
      be too small.

    o allow modification of best solution depending on delivered f(xmean)

    o re−write input and output procedures
*/

#include <math.h>   /* sqrt() */
#include <stddef.h> /* size_t */
#include <stdlib.h> /* NULL, free */
#include <string.h> /* strlen() */
#include <stdio.h>  /* sprintf(), NULL? */
#include "cmaes_interface.h" /* <time.h> via cmaes.h */

/* −−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−− */
/* −−−−−−−−−−−−−−−−−−−− Declarations −−−−−−−−−−−−−−−−−−−−−−−−−− */
/* −−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−− */

/* −−−−−−−−−−−−−−−−−−− External Visibly −−−−−−−−−−−−−−−−−−−− */

/* see cmaes_interface.h for those, not listed here */

long   random_init(random_t *, long unsigned seed /* 0==clock */);
void   random_exit(random_t *);
double random_Gauss(random_t *); /* (0,1)−normally distributed */
double random_Uniform(random_t *);
long   random_Start(random_t *, long unsigned seed /* 0==1 */);

void   timings_start(timings_t *timing); /* fields totaltime and tictoctime */
double timings_update(timings_t *timing);
void   timings_tic(timings_t *timing);
double timings_toc(timings_t *timing);

void readpara_init (readpara_t *, int dim, int seed,  const double * xstart,
                    const double * sigma, int lambda, unsigned int seed,
                    const char * filename);
void readpara_exit(readpara_t *);
void readpara_ReadFromFile(readpara_t *, const char *szFileName);
void readpara_SupplementDefaults(readpara_t *);
void readpara_SetWeights(readpara_t *, const char * mode);
void readpara_WriteToFile(readpara_t *, const char *filenamedest,
                          const char *parafilesource);

double const * cmaes_SetMean(cmaes_t *, const double *xmean);
void cmaes_WriteToFile(cmaes_t *, const char *key, const char *name);
void cmaes_WriteToFileAW(cmaes_t *t, const char *key, const char *name,
                         char * append);
void cmaes_WriteToFilePtr(cmaes_t *, const char *key, FILE *fp);
void cmaes_ReadFromFilePtr(cmaes_t *, FILE *fp);
```

```c
void cmaes_FATAL(char const *s1, char const *s2,
                 char const *s3, char const *s4);


/* ------------------- Locally visibly ----------------------- */

static char * getTimeStr(void);
static void TestMinStdDevs( cmaes_t *);
/* static void WriteMaxErrorInfo( cmaes_t *); */

static void Eigen( int N,  double **C, double *diag, double **Q,
                   double *rgtmp);
static int  Check_Eigen( int N,  double **C, double *diag, double **Q);
static void QLalgo2 (int n, double *d, double *e, double **V);
static void Householder2(int n, double **V, double *d, double *e);
static void Adapt_C2(cmaes_t *t, int hsig);

static void FATAL(char const *sz1, char const *s2,
                  char const *s3, char const *s4);
static void ERRORMESSAGE(char const *sz1, char const *s2,
                         char const *s3, char const *s4);
static void   Sorted_index( const double *rgFunVal, int *index, int n);
static int    SignOfDiff( const void *d1, const void * d2);
static double rgdouMax( const double *rgd, int len);
static double rgdouMin( const double *rgd, int len);
static double douMax( double d1, double d2);
static double douMin( double d1, double d2);
static int    intMin( int i, int j);
static int    MaxIdx( const double *rgd, int len);
static int    MinIdx( const double *rgd, int len);
static double myhypot(double a, double b);
static double * new_double( int n);
static void * new_void( int n, size_t size);

/* --------------------------------------------------------- */
/* ---------------- Functions: cmaes_t --------------------- */
/* --------------------------------------------------------- */

static char *
getTimeStr(void) {
  time_t tm = time(NULL);
  static char s[33];

  /* get time */
  strncpy(s, ctime(&tm), 24); /* TODO: hopefully we read something useful */
  s[24] = '\0'; /* cut the \n */
  return s;
}

char *
cmaes_SayHello(cmaes_t *t)
{
  /* write initial message */
  sprintf(t->sOutString,
          "(%d,%d)-CMA-ES(mu_eff=%.1f), Ver=\"%s\", dimension=%d, randomSeed=%d (%s)",
          t->sp.mu, t->sp.lambda, t->sp.mueff, t->version, t->sp.N,
          t->sp.seed, getTimeStr());

  return t->sOutString;
}

double *
cmaes_init(cmaes_t *t, /* "this" */
                int dimension,
                double *inxstart,
                double *inrgstddev, /* initial stds */
                long int inseed,
                int lambda,
                unsigned int seed,
                const char *input_parameter_filename)
{
  int i, j, N;
  double dtest, trace;

  t->version = "3.02.03.beta";
```

```c
  readpara_init (&t->sp, dimension, inseed, inxstart, inrgstddev,
                 lambda, seed, input_parameter_filename);
  t->sp.seed = random_init( &t->rand, (unsigned) t->sp.seed);

  N = t->sp.N; /* for convenience */

  /* initialization  */
  for (i = 0, trace = 0.; i < N; ++i)
    trace += t->sp.rgInitialStds[i]*t->sp.rgInitialStds[i];
  t->sigma = sqrt(trace/N); /* t->sp.mueff/(0.2*t->sp.mueff+sqrt(N)) * sqrt(trac
e/N); */

  t->chiN = sqrt((double) N) * (1. - 1./(4.*N) + 1./(21.*N*N));
  t->flgEigensysIsUptodate = 1;
  t->flgCheckEigen = 0;
  t->genOfEigensysUpdate = 0;
  timings_start(&t->eigenTimings);
  t->flgIniphase = 0; /* do not use iniphase, hsig does the job now */
  t->flgresumedone = 0;
  t->flgStop = 0;

  for (dtest = 1.; dtest && dtest < 1.1 * dtest; dtest *= 2.)
    if (dtest == dtest + 1.)
      break;
  t->dMaxSignifKond = dtest / 1000.; /* not sure whether this is really save, 10
0 does not work well enough */

  t->gen = 0;
  t->countevals = 0;
  t->state = 0;
  t->dLastMinEWgroesserNull = 1.0;
  t->printtime = t->writetime = t->firstwritetime = t->firstprinttime = 0;

  t->rgpc = new_double(N);
  t->rgps = new_double(N);
  t->rgdTmp = new_double(N+1);
  t->rgBDz = new_double(N);
  t->rgxmean = new_double(N+2); t->rgxmean[0] = N; ++t->rgxmean;
  t->rgxold = new_double(N+2); t->rgxold[0] = N; ++t->rgxold;
  t->rgxbestever = new_double(N+3); t->rgxbestever[0] = N; ++t->rgxbestever;
  t->rgout = new_double(N+2); t->rgout[0] = N; ++t->rgout;
  t->rgD = new_double(N);
  t->C = (double**)new_void(N, sizeof(double*));
  t->B = (double**)new_void(N, sizeof(double*));
  t->publicFitness = new_double(t->sp.lambda);
  t->rgFuncValue = new_double(t->sp.lambda+1);
  t->rgFuncValue[0]=t->sp.lambda; ++t->rgFuncValue;
  t->arFuncValueHist = new_double(10+(int)ceil(3.*10.*N/t->sp.lambda)+1);
  t->arFuncValueHist[0] = (double)(10+(int)ceil(3.*10.*N/t->sp.lambda));
  t->arFuncValueHist++;

  for (i = 0; i < N; ++i) {
      t->C[i] = new_double(i+1);
      t->B[i] = new_double(N);
    }
  t->index = (int *) new_void(t->sp.lambda, sizeof(int));
  for (i = 0; i < t->sp.lambda; ++i)
    t->index[i] = i; /* should not be necessary */
  t->rgrgx = (double **)new_void(t->sp.lambda, sizeof(double*));
  for (i = 0; i < t->sp.lambda; ++i) {
    t->rgrgx[i] = new_double(N+2);
    t->rgrgx[i][0] = N;
    t->rgrgx[i]++;
  }

  /* Initialize newed space  */

  for (i = 0; i < N; ++i)
    for (j = 0; j < i; ++j)
       t->C[i][j] = t->B[i][j] = t->B[j][i] = 0.;

  for (i = 0; i < N; ++i)
    {
      t->B[i][i] = 1.;
      t->C[i][i] = t->rgD[i] = t->sp.rgInitialStds[i] * sqrt(N / trace);
      t->C[i][i] *= t->C[i][i];
      t->rgpc[i] = t->rgps[i] = 0.;
    }

  t->minEW = rgdouMin(t->rgD, N); t->minEW = t->minEW * t->minEW;
  t->maxEW = rgdouMax(t->rgD, N); t->maxEW = t->maxEW * t->maxEW;

  t->maxdiagC=t->C[0][0]; for(i=1;i<N;++i) if(t->maxdiagC<t->C[i][i]) t->maxdiag
C=t->C[i][i];
  t->mindiagC=t->C[0][0]; for(i=1;i<N;++i) if(t->mindiagC>t->C[i][i]) t->mindiag
C=t->C[i][i];

  /* set xmean */
  for (i = 0; i < N; ++i)
    t->rgxmean[i] = t->rgxold[i] = t->sp.xstart[i];
  /* use in case xstart as typicalX */
  if (t->sp.typicalXcase)
    for (i = 0; i < N; ++i)
      t->rgxmean[i] += t->sigma * t->rgD[i] * random_Gauss(&t->rand);

  if (strcmp(t->sp.resumefile, "_no_")  != 0)
    cmaes_resume_distribution(t, t->sp.resumefile);

  return (t->publicFitness);

} /* cmaes_init() */

/* --------------------------------------------------------- */
/* --------------------------------------------------------- */

void
cmaes_resume_distribution(cmaes_t *t, char *filename)
{
  int i, j, res, n;
  double d;
  FILE *fp = fopen( filename, "r");
  if(fp == NULL) {
    ERRORMESSAGE("cmaes_resume_distribution(): could not open '",
                 filename, "'",0);
    return;
  }
  /* count number of "resume" entries */
  i = 0; res = 0;
  while (1) {
    if ((res = fscanf(fp, " resume %lg", &d)) == EOF)
      break;
    else if (res==0)
      fscanf(fp, " %*s");
    else if(res > 0)
      i += 1;
  }

  /* go to last "resume" entry */
  n = i; i = 0; res = 0; rewind(fp);
  while (i<n) {
    if ((res = fscanf(fp, " resume %lg", &d)) == EOF)
      FATAL("cmaes_resume_distribution(): Unexpected error, bug",0,0,0);
    else if (res==0)
      fscanf(fp, " %*s");
    else if(res > 0)
      ++i;
  }
  if (d != t->sp.N)
    FATAL("cmaes_resume_distribution(): Dimension numbers do not match",0,0,0);

  /* find next "xmean" entry */
  while (1) {
    if ((res = fscanf(fp, " xmean %lg", &d)) == EOF)
      FATAL("cmaes_resume_distribution(): 'xmean' not found",0,0,0);
    else if (res==0)
      fscanf(fp, " %*s");
    else if(res > 0)
      break;
  }
```

```c
  /* read xmean */
  t->rgxmean[0] = d; res = 1;
  for(i = 1; i < t->sp.N; ++i)
    res += fscanf(fp, " %lg", &t->rgxmean[i]);
  if (res != t->sp.N)
    FATAL("cmaes_resume_distribution(): xmean: dimensions differ",0,0,0);

  /* find next "path for sigma" entry */
  while (1) {
    if ((res = fscanf(fp, " path for sigma %lg", &d)) == EOF)
      FATAL("cmaes_resume_distribution(): 'path for sigma' not found",0,0,0);
    else if (res==0)
      fscanf(fp, " %*s");
    else if(res > 0)
      break;
  }

  /* read ps */
  t->rgps[0] = d; res = 1;
  for(i = 1; i < t->sp.N; ++i)
    res += fscanf(fp, " %lg", &t->rgps[i]);
  if (res != t->sp.N)
    FATAL("cmaes_resume_distribution(): ps: dimensions differ",0,0,0);

  /* find next "path for C" entry */
  while (1) {
    if ((res = fscanf(fp, " path for C %lg", &d)) == EOF)
      FATAL("cmaes_resume_distribution(): 'path for C' not found",0,0,0);
    else if (res==0)
      fscanf(fp, " %*s");
    else if(res > 0)
      break;
  }
  /* read pc */
  t->rgpc[0] = d; res = 1;
  for(i = 1; i < t->sp.N; ++i)
    res += fscanf(fp, " %lg", &t->rgpc[i]);
  if (res != t->sp.N)
    FATAL("cmaes_resume_distribution(): pc: dimensions differ",0,0,0);

  /* find next "sigma" entry */
  while (1) {
    if ((res = fscanf(fp, " sigma %lg", &d)) == EOF)
      FATAL("cmaes_resume_distribution(): 'sigma' not found",0,0,0);
    else if (res==0)
      fscanf(fp, " %*s");
    else if(res > 0)
      break;
  }
  t->sigma = d;

  /* find next entry "covariance matrix" */
  while (1) {
    if ((res = fscanf(fp, " covariance matrix %lg", &d)) == EOF)
      FATAL("cmaes_resume_distribution(): 'covariance matrix' not found",0,0,0);
    else if (res==0)
      fscanf(fp, " %*s");
    else if(res > 0)
      break;
  }
  /* read C */
  t->C[0][0] = d; res = 1;
  for (i = 1; i < t->sp.N; ++i)
    for (j = 0; j <= i; ++j)
      res += fscanf(fp, " %lg", &t->C[i][j]);
  if (res != (t->sp.N*t->sp.N+t->sp.N)/2)
    FATAL("cmaes_resume_distribution(): C: dimensions differ",0,0,0);

  t->flgIniphase = 0;
  t->flgEigensysIsUptodate = 0;
  t->flgresumedone = 1;
  cmaes_UpdateEigensystem(t, 1);

} /* cmaes_resume_distribution() */
```

```
/* --------------------------------------------------------- */
/* --------------------------------------------------------- */

void
cmaes_exit(cmaes_t *t)
{
  int i, N = t->sp.N;
  t->state = -1; /* not really useful at the moment */
  free( t->rgpc);
  free( t->rgps);
  free( t->rgdTmp);
  free( t->rgBDz);
  free( --t->rgxmean);
  free( --t->rgxold);
  free( --t->rgxbestever);
  free( --t->rgout);
  free( t->rgD);
  for (i = 0; i < N; ++i) {
    free( t->C[i]);
    free( t->B[i]);
  }
  for (i = 0; i < t->sp.lambda; ++i)
    free( --t->rgrgx[i]);
  free( t->rgrgx);
  free( t->C);
  free( t->B);
  free( t->index);
  free( t->publicFitness);
  free( --t->rgFuncValue);
  free( --t->arFuncValueHist);
  random_exit (&t->rand);
  readpara_exit (&t->sp);
} /* cmaes_exit() */


/* --------------------------------------------------------- */
/* --------------------------------------------------------- */
double const *
cmaes_SetMean(cmaes_t *t, const double *xmean)
/*
 * Distribution mean could be changed before SamplePopulation().
 * This might lead to unexpected behaviour if done repeatedly.
 */
{
  int i, N=t->sp.N;

  if (t->state >= 1 && t->state < 3)
    FATAL("cmaes_SetMean: mean cannot be set inbetween the calls of ",
          "SamplePopulation and UpdateDistribution",0,0);

  if (xmean != NULL && xmean != t->rgxmean)
    for(i = 0; i < N; ++i)
      t->rgxmean[i] = xmean[i];
  else
    xmean = t->rgxmean;

  return xmean;
}

/* --------------------------------------------------------- */
/* --------------------------------------------------------- */
double * const *
cmaes_SamplePopulation(cmaes_t *t)
{
  int iNk, i, j, N=t->sp.N;
  double sum;
  double const *xmean = t->rgxmean;

  /* cmaes_SetMean(t, xmean); * xmean could be changed at this point */

  /* calculate eigensystem  */
  cmaes_UpdateEigensystem(t, 0);

  /* treat minimal standard deviations and numeric problems */
  TestMinStdDevs(t);
```

```

  for (iNk = 0; iNk < t->sp.lambda; ++iNk)
    { /* generate scaled random vector (D * z)    */
      for (i = 0; i < N; ++i)
        t->rgdTmp[i] = t->rgD[i] * random_Gauss(&t->rand);
      /* add mutation (sigma * B * (D*z)) */
      for (i = 0; i < N; ++i) {
        for (j = 0, sum = 0.; j < N; ++j)
          sum += t->B[i][j] * t->rgdTmp[j];
        t->rgrgx[iNk][i] = xmean[i] + t->sigma * sum;
      }
    }
  if(t->state == 3 || t->gen == 0)
    ++t->gen;
  t->state = 1;

  return(t->rgrgx);
} /* SamplePopulation() */

/* --------------------------------------------------------- */
/* --------------------------------------------------------- */
double const *
cmaes_ReSampleSingle_old( cmaes_t *t, double *rgx)
{
  int i, j, N=t->sp.N;
  double sum;

  if (rgx == NULL)
    FATAL("cmaes_ReSampleSingle(): Missing input double *x",0,0,0);

  for (i = 0; i < N; ++i)
    t->rgdTmp[i] = t->rgD[i] * random_Gauss(&t->rand);
  /* add mutation (sigma * B * (D*z)) */
  for (i = 0; i < N; ++i) {
    for (j = 0, sum = 0.; j < N; ++j)
      sum += t->B[i][j] * t->rgdTmp[j];
    rgx[i] = t->rgxmean[i] + t->sigma * sum;
  }
  return rgx;
}

/* --------------------------------------------------------- */
/* --------------------------------------------------------- */
double * const *
cmaes_ReSampleSingle( cmaes_t *t, int index)
{
  int i, j, N=t->sp.N;
  double *rgx;
  double sum;
  static char s[99];

  if (index < 0 || index >= t->sp.lambda) {
    sprintf(s, "index==%d must be between 0 and %d", index, t->sp.lambda);
    FATAL("cmaes_ReSampleSingle(): Population member ",s,0,0);
  }
  rgx = t->rgrgx[index];

  for (i = 0; i < N; ++i)
    t->rgdTmp[i] = t->rgD[i] * random_Gauss(&t->rand);
  /* add mutation (sigma * B * (D*z)) */
  for (i = 0; i < N; ++i) {
    for (j = 0, sum = 0.; j < N; ++j)
      sum += t->B[i][j] * t->rgdTmp[j];
    rgx[i] = t->rgxmean[i] + t->sigma * sum;
  }
  return(t->rgrgx);
}

/* --------------------------------------------------------- */
/* --------------------------------------------------------- */
double *
cmaes_SampleSingleInto( cmaes_t *t, double *rgx)
{
  int i, j, N=t->sp.N;
  double sum;
```

```c
  if (rgx == NULL)
    rgx = new_double(N);

  for (i = 0; i < N; ++i)
    t->rgdTmp[i] = t->rgD[i] * random_Gauss(&t->rand);
  /* add mutation (sigma * B * (D*z)) */
  for (i = 0; i < N; ++i) {
    for (j = 0, sum = 0.; j < N; ++j)
      sum += t->B[i][j] * t->rgdTmp[j];
    rgx[i] = t->rgxmean[i] + t->sigma * sum;
  }
  return rgx;
}

/* --------------------------------------------------------- */
/* --------------------------------------------------------- */
double *
cmaes_UpdateDistribution( cmaes_t *t, const double *rgFunVal)
{
  int i, j, iNk, hsig, N=t->sp.N;
  double sum;
  double psxps;

  if(t->state == 3)
    FATAL("cmaes_UpdateDistribution(): You need to call \n",
          "SamplePopulation() before update can take place.",0,0);
  if(rgFunVal == NULL)
    FATAL("cmaes_UpdateDistribution(): ",
          "Fitness function value array input is missing.",0,0);

  if(t->state == 1)  /* function values are delivered here */
    t->countevals += t->sp.lambda;
  else
    ERRORMESSAGE("cmaes_UpdateDistribution(): unexpected state",0,0,0);

  /* assign function values */
  for (i=0; i < t->sp.lambda; ++i)
    t->rgrgx[i][N] = t->rgFuncValue[i] = rgFunVal[i];


  /* Generate index */
  Sorted_index(rgFunVal, t->index, t->sp.lambda);

  /* Test if function values are identical, escape flat fitness */
  if (t->rgFuncValue[t->index[0]] ==
      t->rgFuncValue[t->index[(int)t->sp.lambda/2]]) {
    t->sigma *= exp(0.2+t->sp.cs/t->sp.damps);
    ERRORMESSAGE("Warning: sigma increased due to equal function values\n",
                 "  Reconsider the formulation of the objective function",0,0);
  }

  /* update function value history */
  for(i = (int)*(t->arFuncValueHist-1)-1; i > 0; --i) /* for(i = t->arFuncValueH
ist[-1]-1; i > 0; --i) */
    t->arFuncValueHist[i] = t->arFuncValueHist[i-1];
  t->arFuncValueHist[0] = rgFunVal[t->index[0]];

  /* update xbestever */
  if (t->rgxbestever[N] > t->rgrgx[t->index[0]][N] || t->gen == 1)
    for (i = 0; i <= N; ++i) {
      t->rgxbestever[i] = t->rgrgx[t->index[0]][i];
      t->rgxbestever[N+1] = t->countevals;
    }

  /* calculate xmean and rgBDz~N(0,C) */
  for (i = 0; i < N; ++i) {
    t->rgxold[i] = t->rgxmean[i];
    t->rgxmean[i] = 0.;
    for (iNk = 0; iNk < t->sp.mu; ++iNk)
      t->rgxmean[i] += t->sp.weights[iNk] * t->rgrgx[t->index[iNk]][i];
    t->rgBDz[i] = sqrt(t->sp.mueff)*(t->rgxmean[i] - t->rgxold[i])/t->sigma;
  }

  /* calculate z := D^(-1) * B^(-1) * rgBDz into rgdTmp */
```

```c
  for (i = 0; i < N; ++i) {
    for (j = 0, sum = 0.; j < N; ++j)
      sum += t->B[j][i] * t->rgBDz[j];
    t->rgdTmp[i] = sum / t->rgD[i];
  }

  /* TODO?: check length of t->rgdTmp and set an upper limit, e.g. 6 stds */
  /* in case of manipulation of arx,
     this can prevent an increase of sigma by several orders of magnitude
     within one step; a five-fold increase in one step can still happen.
  */
  /*
    for (j = 0, sum = 0.; j < N; ++j)
      sum += t->rgdTmp[j] * t->rgdTmp[j];
    if (sqrt(sum) > chiN + 6. * sqrt(0.5)) {
      rgdTmp length should be set to upper bound and hsig should become zero
    }
  */

  /* cumulation for sigma (ps) using B*z */
  for (i = 0; i < N; ++i) {
    for (j = 0, sum = 0.; j < N; ++j)
      sum += t->B[i][j] * t->rgdTmp[j];
    t->rgps[i] = (1. - t->sp.cs) * t->rgps[i] +
      sqrt(t->sp.cs * (2. - t->sp.cs)) * sum;
  }

  /* calculate norm(ps)^2 */
  for (i = 0, psxps = 0.; i < N; ++i)
    psxps += t->rgps[i] * t->rgps[i];

  /* cumulation for covariance matrix (pc) using B*D*z~N(0,C) */
  hsig = sqrt(psxps) / sqrt(1. - pow(1.-t->sp.cs, 2*t->gen)) / t->chiN
    < 1.4 + 2./(N+1);
  for (i = 0; i < N; ++i) {
    t->rgpc[i] = (1. - t->sp.ccumcov) * t->rgpc[i] +
      hsig * sqrt(t->sp.ccumcov * (2. - t->sp.ccumcov)) * t->rgBDz[i];
  }

  /* stop initial phase */
  if (t->flgIniphase &&
      t->gen > douMin(1/t->sp.cs, 1+N/t->sp.mucov))
    {
      if (psxps / t->sp.damps / (1.-pow((1. - t->sp.cs), t->gen))
          < N * 1.05)
        t->flgIniphase = 0;
    }

#if 0
  /* remove momentum in ps, if ps is large and fitness is getting worse */
  /* This is obsolete due to hsig and harmful in a dynamic environment */
  if(psxps/N > 1.5 + 10.*sqrt(2./N)
     && t->arFuncValueHist[0] > t->arFuncValueHist[1]
     && t->arFuncValueHist[0] > t->arFuncValueHist[2]) {
    double tfac = sqrt((1 + douMax(0, log(psxps/N))) * N / psxps);
    for (i=0; i<N; ++i)
      t->rgps[i] *= tfac;
    psxps *= tfac*tfac;
  }
#endif

  /* update of C  */

  Adapt_C2(t, hsig);

  /* Adapt_C(t); not used anymore */

#if 0
  if (t->sp.ccov != 0. && t->flgIniphase == 0) {
    int k;

    t->flgEigensysIsUptodate = 0;

    /* update covariance matrix */
    for (i = 0; i < N; ++i)
```

```
      for (j = 0; j <=i; ++j) {
        t->C[i][j] = (1 - t->sp.ccov) * t->C[i][j]
          + t->sp.ccov * (1./t->sp.mucov)
            * (t->rgpc[i] * t->rgpc[j]
               + (1-hsig)*t->sp.ccumcov*(2.-t->sp.ccumcov) * t->C[i][j]);
        for (k = 0; k < t->sp.mu; ++k) /* additional rank mu update */
          t->C[i][j] += t->sp.ccov * (1-1./t->sp.mucov) * t->sp.weights[k]
            * (t->rgrgx[t->index[k]][i] - t->rgxold[i])
            * (t->rgrgx[t->index[k]][j] - t->rgxold[j])
            / t->sigma / t->sigma;
      }
  }
#endif


  /* update of sigma */
  t->sigma *= exp(((sqrt(psxps)/t->chiN)-1.)*t->sp.cs/t->sp.damps);

  t->state = 3;

  return (t->rgxmean);

} /* cmaes_UpdateDistribution() */


/* --------------------------------------------------------- */
/* --------------------------------------------------------- */
static void
Adapt_C2(cmaes_t *t, int hsig)
{
  int i, j, k, N=t->sp.N;
  if (t->sp.ccov != 0. && t->flgIniphase == 0) {

    /* definitions for speeding up inner-most loop */
    double ccovmu = t->sp.ccov * (1-1./t->sp.mucov);
    double sigmasquare = t->sigma * t->sigma;

    t->flgEigensysIsUptodate = 0;

    /* update covariance matrix */
    for (i = 0; i < N; ++i)
      for (j = 0; j <=i; ++j) {
        t->C[i][j] = (1 - t->sp.ccov) * t->C[i][j]
          + t->sp.ccov * (1./t->sp.mucov)
            * (t->rgpc[i] * t->rgpc[j]
               + (1-hsig)*t->sp.ccumcov*(2.-t->sp.ccumcov) * t->C[i][j]);
        for (k = 0; k < t->sp.mu; ++k) { /* additional rank mu update */
          t->C[i][j] += ccovmu * t->sp.weights[k]
            * (t->rgrgx[t->index[k]][i] - t->rgxold[i])
            * (t->rgrgx[t->index[k]][j] - t->rgxold[j])
            / sigmasquare;
        }
      }
    /* update maximal and minimal diagonal value */
    t->maxdiagC = t->mindiagC = t->C[0][0];
    for (i = 1; i < N; ++i) {
      if (t->maxdiagC < t->C[i][i])
        t->maxdiagC = t->C[i][i];
      else if (t->mindiagC > t->C[i][i])
        t->mindiagC = t->C[i][i];
    }
  } /* if ccov... */
}


/* --------------------------------------------------------- */
/* --------------------------------------------------------- */
static void
TestMinStdDevs(cmaes_t *t)
  /* increases sigma */
{
  int i, N = t->sp.N;
  if (t->sp.rgDiffMinChange == NULL)
    return;

```

```
  for (i = 0; i < N; ++i)
    while (t->sigma * sqrt(t->C[i][i]) < t->sp.rgDiffMinChange[i])
      t->sigma *= exp(0.05+t->sp.cs/t->sp.damps);

} /* cmaes_TestMinStdDevs() */


/* --------------------------------------------------------- */
/* --------------------------------------------------------- */
void cmaes_WriteToFile(cmaes_t *t, const char *key, const char *name)
{
  cmaes_WriteToFileAW(t, key, name, "a"); /* default is append */
}

/* --------------------------------------------------------- */
/* --------------------------------------------------------- */
void cmaes_WriteToFileAW(cmaes_t *t, const char *key, const char *name,
                         char *appendwrite)
{
  char *s = "tmpcmaes.dat";
  FILE *fp;

  if (name == NULL)
    name = s;

  fp = fopen( name, appendwrite);

  if(fp == NULL) {
    ERRORMESSAGE("cmaes_WriteToFile(): could not open '", name,
                 "' with flag ", appendwrite);
    return;
  }

  if (appendwrite[0] == 'w') {
    /* write a header line, very rudimentary */
    fprintf(fp, "%% # %s (randomSeed=%d, %s)\n", key, t->sp.seed, getTimeStr());
  } else
    if (t->gen > 0 || strncmp(name, "outcmaesfit", 11) != 0)
      cmaes_WriteToFilePtr(t, key, fp); /* do not write fitness for gen==0 */

  fclose(fp);

} /* WriteToFile */

/* --------------------------------------------------------- */
void cmaes_WriteToFilePtr(cmaes_t *t, const char *key, FILE *fp)

/* this hack reads key words from input key for data to be written to
 * a file, see file signals.par as input file. The length of the keys
 * is mostly fixed, see key += number in the code! If the key phrase
 * does not match the expectation the output might be strange.  for
 * cmaes_t *t == NULL it solely prints key as a header line. Input key
 * must be zero terminated.
 */
{
  int i, k, N=(t ? t->sp.N : 0);
  char const *keyend, *keystart;
  char *s = "few";
  if (key == NULL)
    key = s;
  keystart = key; /* for debugging purpose */
  keyend = key + strlen(key);

  while (key < keyend)
    {
      if (strncmp(key, "axisratio", 9) == 0)
        {
          fprintf(fp, "%.2e", sqrt(t->maxEW/t->minEW));
          while (*key != '+' && *key != '\0' && key < keyend)
           ++key;
          fprintf(fp, "%c", (*key=='+') ? '\t':'\n');
        }
      if (strncmp(key, "idxminSD", 8) == 0)
        {
          int mini=0; for(i=N-1;i>0;--i) if(t->mindiagC==t->C[i][i]) mini=i;
```

```
            fprintf(fp, "%d", mini+1);
            while (*key != '+' && *key != '\0' && key < keyend)
             ++key;
            fprintf(fp, "%c", (*key=='+') ? '\t':'\n');
          }
        if (strncmp(key, "idxmaxSD", 8) == 0)
          {
            int maxi=0; for(i=N-1;i>0;--i) if(t->maxdiagC==t->C[i][i]) maxi=i;
            fprintf(fp, "%d", maxi+1);
            while (*key != '+' && *key != '\0' && key < keyend)
             ++key;
            fprintf(fp, "%c", (*key=='+') ? '\t':'\n');
          }
        /* new coordinate system == all eigenvectors */
        if (strncmp(key, "B", 1) == 0)
          {
            /* int j, index[N]; */
            int j, *index=(int*)(new_void(N,sizeof(int))); /* MT */
            Sorted_index(t->rgD, index, N); /* should not be necessary, see end of
   QLalgo2 */
            /* One eigenvector per row, sorted: largest eigenvalue first */
            for (i = 0; i < N; ++i)
              for (j = 0; j < N; ++j)
                fprintf(fp, "%g%c", t->B[j][index[N-1-i]], (j==N-1)?'\n':'\t');
            ++key;
            free(index); /* MT */
          }
        /* covariance matrix */
        if (strncmp(key, "C", 1) == 0)
          {
            int j;
            for (i = 0; i < N; ++i)
              for (j = 0; j <= i; ++j)
                fprintf(fp, "%g%c", t->C[i][j], (j==i)?'\n':'\t');
            ++key;
          }
        /* (processor) time (used) since begin of execution */
        if (strncmp(key, "clock", 4) == 0)
          {
            timings_update(&t->eigenTimings);
            fprintf(fp, "%.1f %.1f",  t->eigenTimings.totaltime,
                    t->eigenTimings.tictoctime);
            while (*key != '+' && *key != '\0' && key < keyend)
              ++key;
            fprintf(fp, "%c", (*key=='+') ? '\t':'\n');
          }
        /* ratio between largest and smallest standard deviation */
        if (strncmp(key, "stddevratio", 11) == 0) /* std dev in coordinate axes */
          {
            fprintf(fp, "%g", sqrt(t->maxdiagC/t->mindiagC));
            while (*key != '+' && *key != '\0' && key < keyend)
             ++key;
            fprintf(fp, "%c", (*key=='+') ? '\t':'\n');
          }
        /* standard deviations in coordinate directions (sigma*sqrt(C[i,i])) */
        if (strncmp(key, "coorstddev", 10) == 0
            || strncmp(key, "stddev", 6) == 0) /* std dev in coordinate axes */
          {
            for (i = 0; i < N; ++i)
              fprintf(fp, "%s%g", (i==0) ? "":"\t", t->sigma*sqrt(t->C[i][i]));
            while (*key != '+' && *key != '\0' && key < keyend)
             ++key;
            fprintf(fp, "%c", (*key=='+') ? '\t':'\n');
          }
        /* diagonal of D == roots of eigenvalues, sorted */
        if (strncmp(key, "diag(D)", 7) == 0)
          {
            for (i = 0; i < N; ++i)
              t->rgdTmp[i] = t->rgD[i];
            qsort(t->rgdTmp, (unsigned) N, sizeof(double), &SignOfDiff); /* superf
  luous */
            for (i = 0; i < N; ++i)
              fprintf(fp, "%s%g", (i==0) ? "":"\t", t->rgdTmp[i]);
            while (*key != '+' && *key != '\0' && key < keyend)
              ++key;
```

```
            fprintf(fp, "%c", (*key=='+') ? '\t':'\n');
          }
        if (strncmp(key, "dim", 3) == 0)
          {
            fprintf(fp, "%d", N);
            while (*key != '+' && *key != '\0' && key < keyend)
              ++key;
            fprintf(fp, "%c", (*key=='+') ? '\t':'\n');
          }
        if (strncmp(key, "eval", 4) == 0)
          {
            fprintf(fp, "%.0f", t->countevals);
            while (*key != '+' && *key != '\0' && key < keyend)
             ++key;
            fprintf(fp, "%c", (*key=='+') ? '\t':'\n');
          }
        if (strncmp(key, "few(diag(D))", 12) == 0)/* between four and six axes */
          {
            int add = (int)(0.5 + (N + 1.) / 5.);
            for (i = 0; i < N; ++i)
              t->rgdTmp[i] = t->rgD[i];
            qsort(t->rgdTmp, (unsigned) N, sizeof(double), &SignOfDiff);
            for (i = 0; i < N-1; i+=add)        /* print always largest */
              fprintf(fp, "%s%g", (i==0) ? "":"\t", t->rgdTmp[N-1-i]);
            fprintf(fp, "\t%g\n", t->rgdTmp[0]);        /* and smallest */
            break; /* number of printed values is not determined */
          }
        if (strncmp(key, "fewinfo", 7) == 0) {
          fprintf(fp," Fevals  Function Value        Sigma   ");
          fprintf(fp, "MaxCoorDev MinCoorDev AxisRatio   MinDii     Time in eig\n");
          while (*key != '+' && *key != '\0' && key < keyend)
            ++key;
        }
        if (strncmp(key, "few", 3) == 0) {
          fprintf(fp, " %5.0f ", t->countevals);
          fprintf(fp, "%.15e", t->rgFuncValue[t->index[0]]);
          fprintf(fp, " %.2e  %.2e %.2e", t->sigma, t->sigma*sqrt(t->maxdiagC),
                  t->sigma*sqrt(t->mindiagC));
          fprintf(fp, " %.2e  %.2e", sqrt(t->maxEW/t->minEW), sqrt(t->minEW));
          while (*key != '+' && *key != '\0' && key < keyend)
            ++key;
          fprintf(fp, "%c", (*key=='+') ? '\t':'\n');
        }
        if (strncmp(key, "funval", 6) == 0 || strncmp(key, "fitness", 6) == 0)
          {
            fprintf(fp, "%.15e", t->rgFuncValue[t->index[0]]);
            while (*key != '+' && *key != '\0' && key < keyend)
              ++key;
            fprintf(fp, "%c", (*key=='+') ? '\t':'\n');
          }
        if (strncmp(key, "fbestever", 9) == 0)
          {
            fprintf(fp, "%.15e", t->rgxbestever[N]); /* f-value */
            while (*key != '+' && *key != '\0' && key < keyend)
              ++key;
            fprintf(fp, "%c", (*key=='+') ? '\t':'\n');
          }
        if (strncmp(key, "fmedian", 7) == 0)
          {
            fprintf(fp, "%.15e", t->rgFuncValue[t->index[(int)(t->sp.lambda/2)]]);
            while (*key != '+' && *key != '\0' && key < keyend)
              ++key;
            fprintf(fp, "%c", (*key=='+') ? '\t':'\n');
          }
        if (strncmp(key, "fworst", 6) == 0)
          {
            fprintf(fp, "%.15e", t->rgFuncValue[t->index[t->sp.lambda-1]]);
            while (*key != '+' && *key != '\0' && key < keyend)
              ++key;
            fprintf(fp, "%c", (*key=='+') ? '\t':'\n');
          }
        if (strncmp(key, "arfunval", 8) == 0 || strncmp(key, "arfitness", 8) == 0)
          {
            for (i = 0; i < N; ++i)
              fprintf(fp, "%s%.10e", (i==0) ? "" : "\t",
                      t->rgFuncValue[t->index[i]]);
            while (*key != ’+’ && *key != ’\0’ && key < keyend)
              ++key;
            fprintf(fp, "%c", (*key==’+’) ? ’\t’:’\n’);
          }
        if (strncmp(key, "gen", 3) == 0)
          {
            fprintf(fp, "%.0f", t->gen);
            while (*key != ’+’ && *key != ’\0’ && key < keyend)
              ++key;
            fprintf(fp, "%c", (*key==’+’) ? ’\t’:’\n’);
          }
        if (strncmp(key, "iter", 4) == 0)
          {
            fprintf(fp, "%.0f", t->gen);
            while (*key != ’+’ && *key != ’\0’ && key < keyend)
             ++key;
            fprintf(fp, "%c", (*key==’+’) ? ’\t’:’\n’);
          }
        if (strncmp(key, "sigma", 5) == 0)
          {
            fprintf(fp, "%.4e", t->sigma);
            while (*key != ’+’ && *key != ’\0’ && key < keyend)
             ++key;
            fprintf(fp, "%c", (*key==’+’) ? ’\t’:’\n’);
          }
        if (strncmp(key, "minSD", 5) == 0) /* minimal standard deviation */
          {
            fprintf(fp, "%.4e", sqrt(t->mindiagC));
            while (*key != ’+’ && *key != ’\0’ && key < keyend)
              ++key;
            fprintf(fp, "%c", (*key==’+’) ? ’\t’:’\n’);
          }
        if (strncmp(key, "maxSD", 5) == 0)
          {
            fprintf(fp, "%.4e", sqrt(t->maxdiagC));
            while (*key != ’+’ && *key != ’\0’ && key < keyend)
              ++key;
            fprintf(fp, "%c", (*key==’+’) ? ’\t’:’\n’);
          }
        if (strncmp(key, "mindii", 6) == 0)
          {
            fprintf(fp, "%.4e", sqrt(t->minEW));
            while (*key != ’+’ && *key != ’\0’ && key < keyend)
             ++key;
            fprintf(fp, "%c", (*key==’+’) ? ’\t’:’\n’);
          }
        if (strncmp(key, "0", 1) == 0)
          {
            fprintf(fp, "0");
            ++key;
            fprintf(fp, "%c", (*key==’+’) ? ’\t’:’\n’);
          }
        if (strncmp(key, "lambda", 6) == 0)
          {
            fprintf(fp, "%d", t->sp.lambda);
            while (*key != ’+’ && *key != ’\0’ && key < keyend)
              ++key;
            fprintf(fp, "%c", (*key==’+’) ? ’\t’:’\n’);
          }
        if (strncmp(key, "N", 1) == 0)
          {
            fprintf(fp, "%d", N);
            ++key;
            fprintf(fp, "%c", (*key==’+’) ? ’\t’:’\n’);
          }
        if (strncmp(key, "resume", 6) == 0)
          {
            fprintf(fp, "\n# resume %d\n", N);
            fprintf(fp, "xmean\n");
            cmaes_WriteToFilePtr(t, "xmean", fp);
            fprintf(fp, "path for sigma\n");
            for(i=0; i<N; ++i)
              fprintf(fp, "%g%s", t->rgps[i], (i==N-1) ? "\n":"\t");
            fprintf(fp, "path for C\n");
```

```
            for(i=0; i<N; ++i)
              fprintf(fp, "%g%s", t->rgpc[i], (i==N-1) ? "\n":"\t");
            fprintf(fp, "sigma %g\n", t->sigma);
            /* note than B and D might not be up-to-date */
            fprintf(fp, "covariance matrix\n");
            cmaes_WriteToFilePtr(t, "C", fp);
            while (*key != ’+’ && *key != ’\0’ && key < keyend)
              ++key;
          }
        if (strncmp(key, "xbest", 5) == 0) { /* best x in recent generation */
          for(i=0; i<N; ++i)
            fprintf(fp, "%s%g", (i==0) ? "":"\t", t->rgrgx[t->index[0]][i]);
          while (*key != ’+’ && *key != ’\0’ && key < keyend)
            ++key;
          fprintf(fp, "%c", (*key==’+’) ? ’\t’:’\n’);
        }
        if (strncmp(key, "xmean", 5) == 0) {
          for(i=0; i<N; ++i)
            fprintf(fp, "%s%g", (i==0) ? "":"\t", t->rgxmean[i]);
          while (*key != ’+’ && *key != ’\0’ && key < keyend)
            ++key;
          fprintf(fp, "%c", (*key==’+’) ? ’\t’:’\n’);
        }
        if (strncmp(key, "all", 3) == 0)
          {
            time_t ti = time(NULL);
            fprintf(fp, "\n# --------- %s\n", asctime(localtime(&ti)));
            fprintf(fp, " N %d\n", N);
            fprintf(fp, " seed %d\n", t->sp.seed);
            fprintf(fp, "function evaluations %.0f\n", t->countevals);
            fprintf(fp, "elapsed (CPU) time [s] %.2f\n", t->eigenTimings.totaltime);
            fprintf(fp, "function value f(x)=%g\n", t->rgrgx[t->index[0]][N]);
            fprintf(fp, "maximal standard deviation %g\n", t->sigma*sqrt(t->maxdiagC));
            fprintf(fp, "minimal standard deviation %g\n", t->sigma*sqrt(t->mindiagC));
            fprintf(fp, "sigma %g\n", t->sigma);
            fprintf(fp, "axisratio %g\n", rgdouMax(t->rgD, N)/rgdouMin(t->rgD, N));
            fprintf(fp, "xbestever found after %.0f evaluations, function value %g\n",
                    t->rgxbestever[N+1], t->rgxbestever[N]);
            for(i=0; i<N; ++i)
              fprintf(fp, " %12g%c", t->rgxbestever[i],
                      (i%5==4||i==N-1)?’\n’:’ ’);
            fprintf(fp, "xbest (of last generation, function value %g)\n",
                    t->rgrgx[t->index[0]][N]);
            for(i=0; i<N; ++i)
              fprintf(fp, " %12g%c", t->rgrgx[t->index[0]][i],
                      (i%5==4||i==N-1)?’\n’:’ ’);
            fprintf(fp, "xmean \n");
            for(i=0; i<N; ++i)
              fprintf(fp, " %12g%c", t->rgxmean[i],
                      (i%5==4||i==N-1)?’\n’:’ ’);
            fprintf(fp, "Standard deviation of coordinate axes (sigma*sqrt(diag(C)))\n");
            for(i=0; i<N; ++i)
              fprintf(fp, " %12g%c", t->sigma*sqrt(t->C[i][i]),
                      (i%5==4||i==N-1)?’\n’:’ ’);
            fprintf(fp, "Main axis lengths of mutation ellipsoid (sigma*diag(D))\n");
            for (i = 0; i < N; ++i)
              t->rgdTmp[i] = t->rgD[i];
            qsort(t->rgdTmp, (unsigned) N, sizeof(double), &SignOfDiff);
            for(i=0; i<N; ++i)
              fprintf(fp, " %12g%c", t->sigma*t->rgdTmp[N-1-i],
                      (i%5==4||i==N-1)?’\n’:’ ’);
            fprintf(fp, "Longest axis (b_i where d_ii=max(diag(D))\n");
            k = MaxIdx(t->rgD, N);
            for(i=0; i<N; ++i)
              fprintf(fp, " %12g%c", t->B[i][k], (i%5==4||i==N-1)?’\n’:’ ’);
            fprintf(fp, "Shortest axis (b_i where d_ii=max(diag(D))\n");
            k = MinIdx(t->rgD, N);
            for(i=0; i<N; ++i)
              fprintf(fp, " %12g%c", t->B[i][k], (i%5==4||i==N-1)?’\n’:’ ’);
            while (*key != ’+’ && *key != ’\0’ && key < keyend)
              ++key;
          } /* "all" */

  #if 0 /* could become generic part */
        s0 = key;
```

```c
      d = cmaes_Get(t, key); /* TODO find way to detect whether key was found */
      if (key == s0) /* this does not work, is always true */
        {
          /* write out stuff, problem: only generic format is available */
          /* move in key until "+" or end */
        }
#endif

      if (*key == ’\0’)
        break;
      else if (*key != ’+’) { /* last key was not recognized */
        ERRORMESSAGE("cmaes_t:WriteToFilePtr(): unrecognized key ’", key, "’", 0);
        while (*key != ’+’ && *key != ’\0’ && key < keyend)
          ++key;
      }
      while (*key == ’+’)
        ++key;
    } /* while key < keyend */

  if (key > keyend)
    FATAL("cmaes_t:WriteToFilePtr(): BUG regarding key sequence",0,0,0);

} /* WriteToFilePtr */

/* --------------------------------------------------------- */
double
cmaes_Get( cmaes_t *t, char const *s)
{
  int N=t->sp.N;

  if (strncmp(s, "axisratio", 5) == 0) { /* between lengths of longest and shortest
 principal axis of the distribution ellipsoid */
    return (rgdouMax(t->rgD, N)/rgdouMin(t->rgD, N));
  }
  else if (strncmp(s, "eval", 4) == 0) { /* number of function evaluations */
    return (t->countevals);
  }
  else if (strncmp(s, "fctvalue", 6) == 0
           || strncmp(s, "funcvalue", 6) == 0
           || strncmp(s, "funvalue", 6) == 0
           || strncmp(s, "fitness", 3) == 0) { /* recent best function value */
    return(t->rgFuncValue[t->index[0]]);
  }
  else if (strncmp(s, "fbestever", 7) == 0) { /* ever best function value */
    return(t->rgxbestever[N]);
  }
  else if (strncmp(s, "generation", 3) == 0
           || strncmp(s, "iteration", 4) == 0) {
    return(t->gen);
  }
  else if (strncmp(s, "maxeval", 4) == 0
           || strncmp(s, "MaxFunEvals", 8) == 0
           || strncmp(s, "stopMaxFunEvals", 12) == 0) { /* maximal number of funct
ion evaluations */
    return(t->sp.stopMaxFunEvals);
  }
  else if (strncmp(s, "maxgen", 4) == 0
           || strncmp(s, "MaxIter", 7) == 0
           || strncmp(s, "stopMaxIter", 11) == 0) { /* maximal number of generatio
ns */
    return(ceil(t->sp.stopMaxIter));
  }
  else if (strncmp(s, "maxaxislength", 5) == 0) { /* sigma * max(diag(D)) */
    return(t->sigma * sqrt(t->maxEW));
  }
  else if (strncmp(s, "minaxislength", 5) == 0) { /* sigma * min(diag(D)) */
    return(t->sigma * sqrt(t->minEW));
  }
  else if (strncmp(s, "maxstddev", 4) == 0) { /* sigma * sqrt(max(diag(C))) */
    return(t->sigma * sqrt(t->maxdiagC));
  }
  else if (strncmp(s, "minstddev", 4) == 0) { /* sigma * sqrt(min(diag(C))) */
    return(t->sigma * sqrt(t->mindiagC));
  }
  else if (strncmp(s, "N", 1) == 0 || strcmp(s, "n") == 0 ||
```

```c
           strncmp(s, "dimension", 3) == 0) {
    return (N);
  }
  else if (strncmp(s, "lambda", 3) == 0
           || strncmp(s, "samplesize", 8) == 0
           || strncmp(s, "popsize", 7) == 0) { /* sample size, offspring populati
on size */
    return(t->sp.lambda);
  }
  else if (strncmp(s, "sigma", 3) == 0) {
    return(t->sigma);
  }
  FATAL( "cmaes_Get(cmaes_t, char const * s): No match found for s=’", s, "’",0);
  return(0);
} /* cmaes_Get() */

/* --------------------------------------------------------- */
double *
cmaes_GetInto( cmaes_t *t, char const *s, double *res)
{
  int i, N = t->sp.N;
  double const * res0 = cmaes_GetPtr(t, s);
  if (res == NULL)
    res = new_double(N);
  for (i = 0; i < N; ++i)
    res[i] = res0[i];
  return res;
}

/* --------------------------------------------------------- */
double *
cmaes_GetNew( cmaes_t *t, char const *s)
{
  return (cmaes_GetInto(t, s, NULL));
}

/* --------------------------------------------------------- */
const double *
cmaes_GetPtr( cmaes_t *t, char const *s)
{
  int i, N=t->sp.N;

  /* diagonal of covariance matrix */
  if (strncmp(s, "diag(C)", 7) == 0) {
    for (i = 0; i < N; ++i)
      t->rgout[i] = t->C[i][i];
    return(t->rgout);
  }
  /* diagonal of axis lengths matrix */
  else if (strncmp(s, "diag(D)", 7) == 0) {
    return(t->rgD);
  }
  /* vector of standard deviations sigma*sqrt(diag(C)) */
  else if (strncmp(s, "stddev", 3) == 0) {
    for (i = 0; i < N; ++i)
      t->rgout[i] = t->sigma * sqrt(t->C[i][i]);
    return(t->rgout);
  }
  /* bestever solution seen so far */
  else if (strncmp(s, "xbestever", 7) == 0)
    return(t->rgxbestever);
  /* recent best solution of the recent population */
  else if (strncmp(s, "xbest", 5) == 0)
    return(t->rgrgx[t->index[0]]);
  /* mean of the recent distribution */
  else if (strncmp(s, "xmean", 1) == 0)
    return(t->rgxmean);

  return(NULL);
}

/* --------------------------------------------------------- */
/* tests stopping criteria
 *   returns a string of satisfied stopping criterion for each line
 *   otherwise NULL
```

```c
*/
const char *
cmaes_TestForTermination( cmaes_t *t)
{
  double range, fac;
  int iAchse, iKoo;
  static char sTestOutString[3024];
  char * cp = sTestOutString;
  int i, cTemp, N=t->sp.N;
  cp[0] = '\0';

      /* function value reached */
      if ((t->gen > 1 || t->state > 1) && t->sp.stStopFitness.flg &&
          t->rgFuncValue[t->index[0]] <= t->sp.stStopFitness.val)
        cp += sprintf(cp, "Fitness: function value %7.2e <= stopFitness (%7.2e)\n",
                      t->rgFuncValue[t->index[0]], t->sp.stStopFitness.val);

      /* TolFun */
      range = douMax(rgdouMax(t->arFuncValueHist, (int)douMin(t->gen,*(t->arFunc
ValueHist-1))),
                     rgdouMax(t->rgFuncValue, t->sp.lambda)) -
        douMin(rgdouMin(t->arFuncValueHist, (int)douMin(t->gen, *(t->arFuncValue
Hist-1))),
               rgdouMin(t->rgFuncValue, t->sp.lambda));

      if (t->gen > 0 && range <= t->sp.stopTolFun) {
        cp += sprintf(cp,
                      "TolFun: function value differences %7.2e < stopTolFun=%7.2e\n",
                      range, t->sp.stopTolFun);
      }

      /* TolFunHist */
      if (t->gen > *(t->arFuncValueHist-1)) {
        range = rgdouMax(t->arFuncValueHist, (int)*(t->arFuncValueHist-1))
          - rgdouMin(t->arFuncValueHist, (int)*(t->arFuncValueHist-1));
        if (range <= t->sp.stopTolFunHist)
          cp += sprintf(cp,
                        "TolFunHist: history of function value changes %7.2e stopTolFunHist=%7.2e",
                        range, t->sp.stopTolFunHist);
      }

      /* TolX */
      for(i=0, cTemp=0; i<N; ++i) {
        cTemp += (t->sigma * sqrt(t->C[i][i]) < t->sp.stopTolX) ? 1 : 0;
        cTemp += (t->sigma * t->rgpc[i] < t->sp.stopTolX) ? 1 : 0;
      }
      if (cTemp == 2*N) {
        cp += sprintf(cp,
                      "TolX: object variable changes below %7.2e \n",
                      t->sp.stopTolX);
      }

      /* TolUpX */
      for(i=0; i<N; ++i) {
        if (t->sigma * sqrt(t->C[i][i]) > t->sp.stopTolUpXFactor * t->sp.rgIniti
alStds[i])
          break;
      }
      if (i < N) {
        cp += sprintf(cp,
                      "TolUpX: standard deviation increased by more than %7.2e, larger initial standard de
viation recommended \n",
                      t->sp.stopTolUpXFactor);
      }

      /* Condition of C greater than dMaxSignifKond */
      if (t->maxEW >= t->minEW * t->dMaxSignifKond) {
        cp += sprintf(cp,
                      "ConditionNumber: maximal condition number %7.2e reached. maxEW=%7.2e,minE
W=%7.2e,maxdiagC=%7.2e,mindiagC=%7.2e\n",
                      t->dMaxSignifKond, t->maxEW, t->minEW, t->maxdiagC, t->min
diagC);
      } /* if */

      /* Principal axis i has no effect on xmean, ie.
```

```c
         x == x + 0.1 * sigma * rgD[i] * B[i] */
      for (iAchse = 0; iAchse < N; ++iAchse)
        {
          fac = 0.1 * t->sigma * t->rgD[iAchse];
          for (iKoo = 0; iKoo < N; ++iKoo){
            if (t->rgxmean[iKoo] != t->rgxmean[iKoo] + fac * t->B[iKoo][iAchse])
              break;
          }
          if (iKoo == N)
            {
              /* t->sigma *= exp(0.2+t->sp.cs/t->sp.damps); */
              cp += sprintf(cp,
                            "NoEffectAxis: standard deviation 0.1*%7.2e in principal axis %d without ef
fect\n",
                            fac/0.1, iAchse);
              break;
            } /* if (iKoo == N) */
        } /* for iAchse      */

      /* Component of xmean is not changed anymore */
      for (iKoo = 0; iKoo < N; ++iKoo)
        {
          if (t->rgxmean[iKoo] == t->rgxmean[iKoo] +
              0.2*t->sigma*sqrt(t->C[iKoo][iKoo]))
            {
              /* t->C[iKoo][iKoo] *= (1 + t->sp.ccov); */
              /* flg = 1; */
              cp += sprintf(cp,
                            "NoEffectCoordinate: standard deviation 0.2*%7.2e in coordinate %d witho
ut effect\n",
                            t->sigma*sqrt(t->C[iKoo][iKoo]), iKoo);
              break;
            }

        } /* for iKoo */
      /* if (flg) t->sigma *= exp(0.05+t->sp.cs/t->sp.damps); */

      if(t->countevals >= t->sp.stopMaxFunEvals)
        cp += sprintf(cp, "MaxFunEvals: conducted function evaluations %.0f >= %g\n",
                      t->countevals, t->sp.stopMaxFunEvals);
      if(t->gen >= t->sp.stopMaxIter)
        cp += sprintf(cp, "MaxIter: number of iterations %.0f >= %g\n",
                      t->gen, t->sp.stopMaxIter);
      if(t->flgStop)
        cp += sprintf(cp, "Manual: stop signal read\n");

#if 0
  else if (0) {
    for(i=0, cTemp=0; i<N; ++i) {
      cTemp += (sigma * sqrt(C[i][i]) < stopdx) ? 1 : 0;
      cTemp += (sigma * rgpc[i] < stopdx) ? 1 : 0;
    }
    if (cTemp == 2*N)
      flgStop = 1;
  }
#endif

  if (cp - sTestOutString>320)
    ERRORMESSAGE("Bug in cmaes_t:Test(): sTestOutString too short",0,0,0);

  if (cp != sTestOutString) {
    return sTestOutString;
  }

  return(NULL);

} /* cmaes_Test() */

/* --------------------------------------------------------- */
void cmaes_ReadSignals(cmaes_t *t, char const *filename)
{
  const char *s = "signals.par";
  FILE *fp;
  if (filename == NULL)
    filename = s;
  fp = fopen( filename, "r");
  if(fp == NULL) {
    return;
  }
  cmaes_ReadFromFilePtr( t, fp);
  fclose(fp);
}
/* --------------------------------------------------------- */
void cmaes_ReadFromFilePtr( cmaes_t *t, FILE *fp)
/* reading commands e.g. from signals.par file
*/
{
  char *keys[15];
  char s[199], sin1[99], sin2[129], sin3[99], sin4[99];
  int ikey, ckeys, nb;
  double d;
  static int flglockprint = 0;
  static int flglockwrite = 0;
  static long countiterlastwritten;
  static long maxdiffitertowrite; /* to prevent long gaps at the beginning */
  int flgprinted = 0;
  int flgwritten = 0;
  double deltaprinttime = time(NULL)-t->printtime; /* using clock instead might
not be a good */
  double deltawritetime = time(NULL)-t->writetime; /* idea as disc time is not C
PU time? */
  double deltaprinttimefirst = t->firstprinttime ? time(NULL)-t->firstprinttime
: 0; /* time is in seconds!? */
  double deltawritetimefirst = t->firstwritetime ? time(NULL)-t->firstwritetime
: 0;
  if (countiterlastwritten > t->gen) { /* probably restarted */
    maxdiffitertowrite = 0;
    countiterlastwritten = 0;
  }

  keys[0] = " stop%98s %98s";        /* s=="now" or eg "MaxIter+" %lg"-number */
                                        /* works with and without space */
  keys[1] = " print %98s %98s";       /* s==keyword for WriteFile */
  keys[2] = " write %98s %128s %98s"; /* s1==keyword, s2==filename */
  keys[3] = " check%98s %98s";
  keys[4] = " maxTimeFractionForEigendecompostion %98s";
  ckeys = 5;
  strcpy(sin2, "tmpcmaes.dat");

  if (cmaes_TestForTermination(t))
    {
      deltaprinttime = time(NULL); /* forces printing */
      deltawritetime = time(NULL);
    }
  while(fgets(s, sizeof(s), fp) != NULL)
    {
      if (s[0] == '#' || s[0] == '%') /* skip comments  */
        continue;
      sin1[0] = sin2[0] = sin3[0] = sin4[0] = '\0';
      for (ikey=0; ikey < ckeys; ++ikey)
        {
          if((nb=sscanf(s, keys[ikey], sin1, sin2, sin3, sin4)) >= 1)
            {
              switch(ikey) {
              case 0 : /* "stop", reads "stop now" or eg. stopMaxIter */
                if (strncmp(sin1, "now", 3) == 0)
                  t->flgStop = 1;
                else if (strncmp(sin1, "MaxFunEvals", 11) == 0) {
                  if (sscanf(sin2, " %lg", &d) == 1)
                    t->sp.stopMaxFunEvals = d;
                }
                else if (strncmp(sin1, "MaxIter", 4) == 0) {
                  if (sscanf(sin2, " %lg", &d) == 1)
                    t->sp.stopMaxIter = d;
                }
                else if (strncmp(sin1, "Fitness", 7) == 0) {
                  if (sscanf(sin2, " %lg", &d) == 1)
                    {
                      t->sp.stStopFitness.flg = 1;
                      t->sp.stStopFitness.val = d;
```

```c
                    }
                }
                else if (strncmp(sin1, "TolFunHist", 10) == 0) {
                  if (sscanf(sin2, " %lg", &d) == 1)
                    t->sp.stopTolFunHist = d;
                }
                else if (strncmp(sin1, "TolFun", 6) == 0) {
                  if (sscanf(sin2, " %lg", &d) == 1)
                    t->sp.stopTolFun = d;
                }
                else if (strncmp(sin1, "TolX", 4) == 0) {
                  if (sscanf(sin2, " %lg", &d) == 1)
                    t->sp.stopTolX = d;
                }
                else if (strncmp(sin1, "TolUpXFactor", 4) == 0) {
                  if (sscanf(sin2, " %lg", &d) == 1)
                    t->sp.stopTolUpXFactor = d;
                }
                break;
              case 1 : /* "print" */
                d = 1; /* default */
                if (sscanf(sin2, "%lg", &d) < 1 && deltaprinttimefirst < 1)
                  d = 0; /* default at first time */
                if (deltaprinttime >= d && !flglockprint) {
                  cmaes_WriteToFilePtr(t, sin1, stdout);
                  flgprinted = 1;
                }
                if(d < 0)
                  flglockprint += 2;
                break;
              case 2 : /* "write" */
                /* write header, before first generation */
                if (t->countevals < t->sp.lambda && t->flgresumedone == 0)
                  cmaes_WriteToFileAW(t, sin1, sin2, "w"); /* overwrite */
                d = 0.9; /* default is one with smooth increment of gaps */
                if (sscanf(sin3, "%lg", &d) < 1 && deltawritetimefirst < 2)
                  d = 0; /* default is zero for the first second */
                if(d < 0)
                  flglockwrite += 2;
                if (!flglockwrite) {
                  if (deltawritetime >= d) {
                    cmaes_WriteToFile(t, sin1, sin2);
                    flgwritten = 1;
                  } else if (d < 1
                             && t->gen-countiterlastwritten > maxdiffitertowrite
) {
                    cmaes_WriteToFile(t, sin1, sin2);
                    flgwritten = 1;
                  }
                }
                break;
              case 3 : /* check, checkeigen 1 or check eigen 1 */
                if (strncmp(sin1, "eigen", 5) == 0) {
                  if (sscanf(sin2, " %lg", &d) == 1) {
                    if (d > 0)
                      t->flgCheckEigen = 1;
                    else
                      t->flgCheckEigen = 0;
                  }
                  else
                    t->flgCheckEigen = 0;
                }
                break;
              case 4 : /* maxTimeFractionForEigendecompostion */
                if (sscanf(sin1, " %lg", &d) == 1)
                  t->sp.updateCmode.maxtime = d;
                break;
              default :
                break;
              }
              break; /* for ikey */
            } /* if line contains keyword */
        } /* for each keyword */
    } /* while not EOF of signals.par */
  if (t->writetime == 0)
```

```c
    t->firstwritetime = time(NULL);
  if (t->printtime == 0)
    t->firstprinttime = time(NULL);

  if (flgprinted)
    t->printtime = time(NULL);
  if (flgwritten) {
    t->writetime = time(NULL);
    if (t->gen-countiterlastwritten > maxdiffitertowrite)
      ++maxdiffitertowrite; /* smooth prolongation of writing gaps/intervals */
    countiterlastwritten = (long int) t->gen;
  }
  --flglockprint;
  --flglockwrite;
  flglockprint = (flglockprint > 0) ? 1 : 0;
  flglockwrite = (flglockwrite > 0) ? 1 : 0;
} /*  cmaes_ReadFromFilePtr */

/* ========================================================= */
static int
Check_Eigen( int N,  double **C, double *diag, double **Q)
/*
   exhaustive test of the output of the eigendecomposition
   needs O(n^3) operations

   writes to error file
   returns number of detected inaccuracies
*/
{
    /* compute Q diag Q^T and Q Q^T to check */
  int i, j, k, res = 0;
  double cc, dd;
  static char s[324];

  for (i=0; i < N; ++i)
    for (j=0; j < N; ++j) {
      for (cc=0.,dd=0., k=0; k < N; ++k) {
        cc += diag[k] * Q[i][k] * Q[j][k];
        dd += Q[i][k] * Q[j][k];
      }
      /* check here, is the normalization the right one? */
      if (fabs(cc - C[i>j?i:j][i>j?j:i])/sqrt(C[i][i]*C[j][j]) > 1e-10
          && fabs(cc - C[i>j?i:j][i>j?j:i]) > 3e-14) {
        sprintf(s, "%d %d: %.17e %.17e, %e",
                i, j, cc, C[i>j?i:j][i>j?j:i], cc-C[i>j?i:j][i>j?j:i]);
        ERRORMESSAGE("cmaes_t:Eigen(): imprecise result detected ",
                     s, 0, 0);
        ++res;
      }
      if (fabs(dd - (i==j)) > 1e-10) {
        sprintf(s, "%d %d %.17e ", i, j, dd);
        ERRORMESSAGE("cmaes_t:Eigen(): imprecise result detected (Q not orthog.)",
                     s, 0, 0);
        ++res;
      }
    }
  return res;
}

/* --------------------------------------------------------- */
/* --------------------------------------------------------- */
void
cmaes_UpdateEigensystem(cmaes_t *t, int flgforce)
{
  int i, N = t->sp.N;

  timings_update(&t->eigenTimings);

  if(flgforce == 0) {
    if (t->flgEigensysIsUptodate == 1)
      return;

    /* return on modulo generation number */
    if (t->sp.updateCmode.flgalways == 0 /* not implemented, always ==0 */
        && t->gen < t->genOfEigensysUpdate + t->sp.updateCmode.modulo
```

```c
        )
      return;

    /* return on time percentage */
    if (t->sp.updateCmode.maxtime < 1.00
        && t->eigenTimings.tictoctime > t->sp.updateCmode.maxtime * t->eigenTimi
ngs.totaltime
        && t->eigenTimings.tictoctime > 0.0002)
      return;
  }
  timings_tic(&t->eigenTimings);

  Eigen( N, t->C, t->rgD, t->B, t->rgdTmp);

  timings_toc(&t->eigenTimings);

  /* find largest and smallest eigenvalue, they are supposed to be sorted anyway
 */
  t->minEW = rgdouMin(t->rgD, N);
  t->maxEW = rgdouMax(t->rgD, N);

  if (t->flgCheckEigen)
    /* needs O(n^3)! writes, in case, error message in error file */
    i = Check_Eigen( N, t->C, t->rgD, t->B);

#if 0
  /* Limit Condition of C to dMaxSignifKond+1 */
  if (t->maxEW > t->minEW * t->dMaxSignifKond) {
    ERRORMESSAGE("Warning: Condition number of covariance matrix at upper limit.",
                 " Consider a rescaling or redesign of the objective function. " ,"","");
    printf("\nWarning: Condition number of covariance matrix at upper limit\n");
    tmp = t->maxEW/t->dMaxSignifKond - t->minEW;
    tmp = t->maxEW/t->dMaxSignifKond;
    t->minEW += tmp;
    for (i=0;i<N;++i) {
      t->C[i][i] += tmp;
      t->rgD[i] += tmp;
    }
  } /* if */
  t->dLastMinEWgroesserNull = minEW;
#endif

  for (i = 0; i < N; ++i)
    t->rgD[i] = sqrt(t->rgD[i]);

  t->flgEigensysIsUptodate = 1;
  t->genOfEigensysUpdate = t->gen;

  return;

} /* cmaes_UpdateEigensystem() */


/* ========================================================= */
static void
Eigen( int N,  double **C, double *diag, double **Q, double *rgtmp)
/*
   Calculating eigenvalues and vectors.
   Input:
     N: dimension.
     C: symmetric (1:N)xN-matrix, solely used to copy data to Q
     niter: number of maximal iterations for QL-Algorithm.
     rgtmp: N+1-dimensional vector for temporal use.
   Output:
     diag: N eigenvalues.
     Q: Columns are normalized eigenvectors.
 */
{
  int i, j;

  if (rgtmp == NULL) /* was OK in former versions */
    FATAL("cmaes_t:Eigen(): input parameter double *rgtmp must be non-NULL", 0,0,0);

  /* copy C to Q */
  if (C != Q) {
    for (i=0; i < N; ++i)
      for (j = 0; j <= i; ++j)
        Q[i][j] = Q[j][i] = C[i][j];
  }

#if 0
    Householder( N, Q, diag, rgtmp);
    QLalgo( N, diag, Q, 30*N, rgtmp+1);
#else
    Householder2( N, Q, diag, rgtmp);
    QLalgo2( N, diag, rgtmp, Q);
#endif

}


/* ========================================================= */
static void
QLalgo2 (int n, double *d, double *e, double **V) {
  /*
    -> n     : Dimension.
    -> d     : Diagonale of tridiagonal matrix.
    -> e[1..n-1] : off-diagonal, output from Householder
    -> V     : matrix output von Householder
    <- d     : eigenvalues
    <- e     : garbage?
    <- V     : basis of eigenvectors, according to d

    Symmetric tridiagonal QL algorithm, iterative
    Computes the eigensystem from a tridiagonal matrix in roughtly 3N^3 operatio
ns

    code adapated from Java JAMA package, function tql2.
  */

  int i, k, l, m;
  double f = 0.0;
  double tst1 = 0.0;
  double eps = 2.22e-16; /* Math.pow(2.0,-52.0);  == 2.22e-16 */

      /* shift input e */
      for (i = 1; i < n; i++) {
         e[i-1] = e[i];
      }
      e[n-1] = 0.0; /* never changed again */

      for (l = 0; l < n; l++) {

        /* Find small subdiagonal element */

         if (tst1 < fabs(d[l]) + fabs(e[l]))
           tst1 = fabs(d[l]) + fabs(e[l]);
         m = l;
         while (m < n) {
           if (fabs(e[m]) <= eps*tst1) {
             /* if (fabs(e[m]) + fabs(d[m]+d[m+1]) == fabs(d[m]+d[m+1])) { */
               break;
            }
            m++;
         }

         /* If m == l, d[l] is an eigenvalue, */
         /* otherwise, iterate. */

         if (m > l) {
            int iter = 0;
            do { /* while (fabs(e[l]) > eps*tst1); */
               double dl1, h;
               double g = d[l];
               double p = (d[l+1] - g) / (2.0 * e[l]);
               double r = myhypot(p, 1.);

               iter = iter + 1;  /* Could check iteration count here */

               /* Compute implicit shift */
```

```c
               if (p < 0) {
                  r = -r;
               }
               d[l] = e[l] / (p + r);
               d[l+1] = e[l] * (p + r);
               dl1 = d[l+1];
               h = g - d[l];
               for (i = l+2; i < n; i++) {
                  d[i] -= h;
               }
               f = f + h;

               /* Implicit QL transformation. */

               p = d[m];
             {
               double c = 1.0;
               double c2 = c;
               double c3 = c;
               double el1 = e[l+1];
               double s = 0.0;
               double s2 = 0.0;
               for (i = m-1; i >= l; i--) {
                  c3 = c2;
                  c2 = c;
                  s2 = s;
                  g = c * e[i];
                  h = c * p;
                  r = myhypot(p, e[i]);
                  e[i+1] = s * r;
                  s = e[i] / r;
                  c = p / r;
                  p = c * d[i] - s * g;
                  d[i+1] = h + s * (c * g + s * d[i]);

                  /* Accumulate transformation. */

                  for (k = 0; k < n; k++) {
                     h = V[k][i+1];
                     V[k][i+1] = s * V[k][i] + c * h;
                     V[k][i] = c * V[k][i] - s * h;
                  }
               }
               p = -s * s2 * c3 * el1 * e[l] / dl1;
               e[l] = s * p;
               d[l] = c * p;
             }

               /* Check for convergence. */

            } while (fabs(e[l]) > eps*tst1);
         }
         d[l] = d[l] + f;
         e[l] = 0.0;
      }

      /* Sort eigenvalues and corresponding vectors. */
#if 1
      /* TODO: really needed here? So far not, but practical and only O(n^2) */
      {
      int j;
      double p;
      for (i = 0; i < n-1; i++) {
         k = i;
         p = d[i];
         for (j = i+1; j < n; j++) {
            if (d[j] < p) {
               k = j;
               p = d[j];
            }
         }
         if (k != i) {
            d[k] = d[i];
            d[i] = p;
            for (j = 0; j < n; j++) {
               p = V[j][i];
               V[j][i] = V[j][k];
               V[j][k] = p;
            }
         }
      }
      }
#endif
} /* QLalgo2 */

/* ========================================================= */
static void
Householder2(int n, double **V, double *d, double *e) {
  /*
     Householder transformation of a symmetric matrix V into tridiagonal form.
   -> n             : dimension
   -> V             : symmetric nxn-matrix
   <- V             : orthogonal transformation matrix:
                      tridiag matrix == V * V_in * V^t
   <- d             : diagonal
   <- e[0..n-1]     : off diagonal (elements 1..n-1)

   code slightly adapted from the Java JAMA package, function private tred2()

  */

  int i,j,k;

      for (j = 0; j < n; j++) {
         d[j] = V[n-1][j];
      }

      /* Householder reduction to tridiagonal form */

      for (i = n-1; i > 0; i--) {

        /* Scale to avoid under/overflow */

         double scale = 0.0;
         double h = 0.0;
         for (k = 0; k < i; k++) {
            scale = scale + fabs(d[k]);
         }
         if (scale == 0.0) {
            e[i] = d[i-1];
            for (j = 0; j < i; j++) {
               d[j] = V[i-1][j];
               V[i][j] = 0.0;
               V[j][i] = 0.0;
            }
         } else {

           /* Generate Householder vector */

            double f, g, hh;

            for (k = 0; k < i; k++) {
               d[k] /= scale;
               h += d[k] * d[k];
            }
            f = d[i-1];
            g = sqrt(h);
            if (f > 0) {
               g = -g;
            }
            e[i] = scale * g;
            h = h - f * g;
            d[i-1] = f - g;
            for (j = 0; j < i; j++) {
               e[j] = 0.0;
            }

            /* Apply similarity transformation to remaining columns */
```

```c
            for (j = 0; j < i; j++) {
               f = d[j];
               V[j][i] = f;
               g = e[j] + V[j][j] * f;
               for (k = j+1; k <= i-1; k++) {
                  g += V[k][j] * d[k];
                  e[k] += V[k][j] * f;
               }
               e[j] = g;
            }
            f = 0.0;
            for (j = 0; j < i; j++) {
               e[j] /= h;
               f += e[j] * d[j];
            }
            hh = f / (h + h);
            for (j = 0; j < i; j++) {
               e[j] -= hh * d[j];
            }
            for (j = 0; j < i; j++) {
               f = d[j];
               g = e[j];
               for (k = j; k <= i-1; k++) {
                  V[k][j] -= (f * e[k] + g * d[k]);
               }
               d[j] = V[i-1][j];
               V[i][j] = 0.0;
            }
         }
         d[i] = h;
      }

      /* Accumulate transformations */

      for (i = 0; i < n-1; i++) {
         double h;
         V[n-1][i] = V[i][i];
         V[i][i] = 1.0;
         h = d[i+1];
         if (h != 0.0) {
            for (k = 0; k <= i; k++) {
               d[k] = V[k][i+1] / h;
            }
            for (j = 0; j <= i; j++) {
               double g = 0.0;
               for (k = 0; k <= i; k++) {
                  g += V[k][i+1] * V[k][j];
               }
               for (k = 0; k <= i; k++) {
                  V[k][j] -= g * d[k];
               }
            }
         }
         for (k = 0; k <= i; k++) {
            V[k][i+1] = 0.0;
         }
      }
      for (j = 0; j < n; j++) {
         d[j] = V[n-1][j];
         V[n-1][j] = 0.0;
      }
      V[n-1][n-1] = 1.0;
      e[0] = 0.0;

} /* Housholder() */


#if 0
/* ========================================================= */
static void
WriteMaxErrorInfo(cmaes_t *t)
{
  int i,j, N=t->sp.N;
  char *s = (char *)new_void(200+30*(N+2), sizeof(char)); s[0] = '\0';
  sprintf( s+strlen(s),"\nKomplett-Info\n");
  sprintf( s+strlen(s)," Gen       %20.12g\n", t->gen);
  sprintf( s+strlen(s)," Dimension %d\n", N);
  sprintf( s+strlen(s)," sigma     %e\n", t->sigma);
  sprintf( s+strlen(s)," lastminEW %e\n",
           t->dLastMinEWgroesserNull);
  sprintf( s+strlen(s)," maxKond   %e\n\n", t->dMaxSignifKond);
  sprintf( s+strlen(s),"     x-Vektor         rgD     Basis...\n");
  ERRORMESSAGE( s,0,0,0);
  s[0] = '\0';
  for (i = 0; i < N; ++i)
    {
      sprintf( s+strlen(s), " %20.12e", t->rgxmean[i]);
      sprintf( s+strlen(s), " %10.4e", t->rgD[i]);
      for (j = 0; j < N; ++j)
        sprintf( s+strlen(s), " %10.2e", t->B[i][j]);
      ERRORMESSAGE( s,0,0,0);
      s[0] = '\0';
    }
  ERRORMESSAGE( "\n",0,0,0);
  free( s);
} /* WriteMaxErrorInfo() */
#endif

/* --------------------------------------------------------- */
/* --------------- Functions: timings_t -------------------- */
/* --------------------------------------------------------- */
/* timings_t measures overall time and times between calls
 * of tic and toc. For small time spans (up to 1000 seconds)
 * CPU time via clock() is used. For large time spans the
 * fall-back to elapsed time from time() is used.
 * timings_update() must be called often enough to prevent
 * the fallback. */
/* --------------------------------------------------------- */
void
timings_start(timings_t *t) {
  t->totaltime = 0;
  t->tictoctime = 0;
  t->lasttictoctime = 0;
  t->istic = 0;
  t->lastclock = clock();
  t->lasttime = time(NULL);
  t->lastdiff = 0;
  t->tictoczwischensumme = 0;
  t->isstarted = 1;
}

double
timings_update(timings_t *t) {
/* returns time between last call of timings_*() and now,
 *    should better return totaltime or tictoctime?
 */
  double diffc, difft;
  clock_t lc = t->lastclock; /* measure CPU in 1e-6s */
  time_t lt = t->lasttime;   /* measure time in s */

  if (t->isstarted != 1)
    FATAL("timings_started() must be called before using timings... functions",0,0,0);

  t->lastclock = clock(); /* measures at most 2147 seconds, where 1s = 1e6 CLOCK
S_PER_SEC */
  t->lasttime = time(NULL);

  diffc = (double)(t->lastclock - lc) / CLOCKS_PER_SEC; /* is presumably in [-21
??, 21??] */
  difft = difftime(t->lasttime, lt);                    /* is presumably an inte
ger */

  t->lastdiff = difft; /* on the "save" side */

  /* use diffc clock measurement if appropriate */
  if (diffc > 0 && difft < 1000)
    t->lastdiff = diffc;

```

```
  if (t->lastdiff < 0)
    FATAL("BUG in time measurement", 0, 0, 0);

  t->totaltime += t->lastdiff;
  if (t->istic) {
    t->tictoczwischensumme += t->lastdiff;
    t->tictoctime += t->lastdiff;
  }

  return t->lastdiff;
}

void
timings_tic(timings_t *t) {
  if (t->istic) { /* message not necessary ? */
    ERRORMESSAGE("Warning: timings_tic called twice without toc",0,0,0);
    return;
  }
  timings_update(t);
  t->istic = 1;
}

double
timings_toc(timings_t *t) {
  if (!t->istic) {
    ERRORMESSAGE("Warning: timings_toc called without tic",0,0,0);
    return -1;
  }
  timings_update(t);
  t->lasttictoctime = t->tictoczwischensumme;
  t->tictoczwischensumme = 0;
  t->istic = 0;
  return t->lasttictoctime;
}

/* --------------------------------------------------------- */
/* ---------------- Functions: random_t -------------------- */
/* --------------------------------------------------------- */
/* --------------------------------------------------------- */
/* X_1 exakt :          0.79788456)  */
/* chi_eins simuliert : 0.798xx   (seed -3) */
/*                     +-0.001 */
/* --------------------------------------------------------- */
/*
   Gauss() liefert normalverteilte Zufallszahlen
   bei vorgegebenem seed.
*/
/* --------------------------------------------------------- */
/* --------------------------------------------------------- */

long
random_init( random_t *t, long unsigned inseed)
{
  clock_t cloc = clock();

  t->flgstored = 0;
  t->rgrand = (long *) new_void(32, sizeof(long));
  if (inseed < 1) {
    while ((long) (cloc - clock()) == 0)
      ; /* TODO: remove this for time critical applications? */
    inseed = (long)abs(100*time(NULL)+clock());
  }
  return random_Start(t, inseed);
}

void
random_exit(random_t *t)
{
  free( t->rgrand);
}

/* --------------------------------------------------------- */
long random_Start( random_t *t, long unsigned inseed)
{
  long tmp;
```

```
    int i;

    t->flgstored = 0;
    t->startseed = inseed;
    if (inseed < 1)
      inseed = 1;
    t->aktseed = inseed;
    for (i = 39; i >= 0; --i)
    {
      tmp = t->aktseed/127773;
      t->aktseed = 16807 * (t->aktseed - tmp * 127773)
        - 2836 * tmp;
      if (t->aktseed < 0) t->aktseed += 2147483647;
      if (i < 32)
        t->rgrand[i] = t->aktseed;
    }
    t->aktrand = t->rgrand[0];
    return inseed;
  }

  /* --------------------------------------------------------- */
  double random_Gauss(random_t *t)
  {
    double x1, x2, rquad, fac;

    if (t->flgstored)
    {
      t->flgstored = 0;
      return t->hold;
    }
    do
    {
      x1 = 2.0 * random_Uniform(t) - 1.0;
      x2 = 2.0 * random_Uniform(t) - 1.0;
      rquad = x1*x1 + x2*x2;
    } while(rquad >= 1 || rquad <= 0);
    fac = sqrt(-2.0*log(rquad)/rquad);
    t->flgstored = 1;
    t->hold = fac * x1;
    return fac * x2;
  }

  /* --------------------------------------------------------- */
  double random_Uniform( random_t *t)
  {
    long tmp;

    tmp = t->aktseed/127773;
    t->aktseed = 16807 * (t->aktseed - tmp * 127773)
      - 2836 * tmp;
    if (t->aktseed < 0)
      t->aktseed += 2147483647;
    tmp = t->aktrand / 67108865;
    t->aktrand = t->rgrand[tmp];
    t->rgrand[tmp] = t->aktseed;
    return (double)(t->aktrand)/(2.147483647e9);
  }

  static char *
  szCat(const char *sz1, const char*sz2,
        const char *sz3, const char *sz4);

  /* --------------------------------------------------------- */
  /* -------------- Functions: readpara_t -------------------- */
  /* --------------------------------------------------------- */
  void
  readpara_init (readpara_t *t,
                 int dim,
                 int inseed,
                 const double * inxstart,
                 const double * inrgsigma,
                 int lambda,
                 unsigned int seed,
                 const char * filename)
  {
```

```
    int i, N;
    t->rgsformat = (char **) new_void(45, sizeof(char *));
    t->rgpadr = (void **) new_void(45, sizeof(void *));
    t->rgskeyar = (char **) new_void(11, sizeof(char *));
    t->rgp2adr = (double ***) new_void(11, sizeof(double **));
    t->weigkey = (char *)new_void(7, sizeof(char));

    /* All scalars:  */
    i = 0;
    t->rgsformat[i] = " N %d";        t->rgpadr[i++] = (void *) &t->N;
    t->rgsformat[i] = " seed %d";     t->rgpadr[i++] = (void *) &t->seed;
    t->rgsformat[i] = " stopMaxFunEvals %lg"; t->rgpadr[i++] = (void *) &t->stopMaxFun
Evals;
    t->rgsformat[i] = " stopMaxIter %lg"; t->rgpadr[i++] = (void *) &t->stopMaxIter;
    t->rgsformat[i] = " stopFitness %lg"; t->rgpadr[i++]=(void *) &t->stStopFitness.va
l;
    t->rgsformat[i] = " stopTolFun %lg"; t->rgpadr[i++]=(void *) &t->stopTolFun;
    t->rgsformat[i] = " stopTolFunHist %lg"; t->rgpadr[i++]=(void *) &t->stopTolFunHist
;
    t->rgsformat[i] = " stopTolX %lg"; t->rgpadr[i++]=(void *) &t->stopTolX;
    t->rgsformat[i] = " stopTolUpXFactor %lg"; t->rgpadr[i++]=(void *) &t->stopTolUpXF
actor;
    t->rgsformat[i] = " lambda %d";       t->rgpadr[i++] = (void *) &t->lambda;
    t->rgsformat[i] = " mu %d";            t->rgpadr[i++] = (void *) &t->mu;
    t->rgsformat[i] = " weights %5s";     t->rgpadr[i++] = (void *) t->weigkey;
    t->rgsformat[i] = " fac*cs %lg";t->rgpadr[i++] = (void *) &t->cs;
    t->rgsformat[i] = " fac*damps %lg";    t->rgpadr[i++] = (void *) &t->damps;
    t->rgsformat[i] = " ccumcov %lg";     t->rgpadr[i++] = (void *) &t->ccumcov;
    t->rgsformat[i] = " mucov %lg";      t->rgpadr[i++] = (void *) &t->mucov;
    t->rgsformat[i] = " fac*ccov %lg";  t->rgpadr[i++]=(void *) &t->ccov;
    t->rgsformat[i] = " updatecov %lg"; t->rgpadr[i++]=(void *) &t->updateCmode.modul
o;
    t->rgsformat[i] = " maxTimeFractionForEigendecompostion %lg"; t->rgpadr[i++]=(void *)
&t->updateCmode.maxtime;
    t->rgsformat[i] = " resume %59s";     t->rgpadr[i++] = (void *) t->resumefile;
    t->rgsformat[i] = " fac*maxFunEvals %lg";   t->rgpadr[i++] = (void *) &t->facmaxev
al;
    t->rgsformat[i] = " fac*updatecov %lg"; t->rgpadr[i++]=(void *) &t->facupdateCmode
;
    t->n1para = i;
    t->n1outpara = i-2; /* disregard last parameters in WriteToFile() */

    /* arrays */
    i = 0;
    t->rgskeyar[i]  = " typicalX %d";   t->rgp2adr[i++] = &t->typicalX;
    t->rgskeyar[i]  = " initialX %d";    t->rgp2adr[i++] = &t->xstart;
    t->rgskeyar[i]  = " initialStandardDeviations %d"; t->rgp2adr[i++] = &t->rgInitialStds
;
    t->rgskeyar[i]  = " diffMinChange %d"; t->rgp2adr[i++] = &t->rgDiffMinChange;
    t->n2para = i;

    t->N = dim;
    t->seed = inseed;
    t->xstart = NULL;
    t->typicalX = NULL;
    t->typicalXcase = 0;
    t->rgInitialStds = NULL;
    t->rgDiffMinChange = NULL;
    t->stopMaxFunEvals = -1;
    t->stopMaxIter = -1;
    t->facmaxeval = 1;
    t->stStopFitness.flg = -1;
    t->stopTolFun = 1e-12;
    t->stopTolFunHist = 1e-13;
    t->stopTolX = 0; /* 1e-11*insigma would also be reasonable */
    t->stopTolUpXFactor = 1e3;

    t->mu = -1;
    t->mucov = -1;
    t->weights = NULL;
    strcpy(t->weigkey, "log");

    t->cs = -1;
    t->ccumcov = -1;
    t->damps = -1;
```

```c
  t->ccov = -1;

  t->updateCmode.modulo = -1;
  t->updateCmode.maxtime = -1;
  t->updateCmode.flgalways = 0;
  t->facupdateCmode = 1;
  strcpy(t->resumefile, "_no_");

  if (strcmp(filename, "non") != 0 && strcmp(filename, "writeonly") != 0)
    readpara_ReadFromFile(t, filename);

  if (lambda > 0)
    t->lambda = lambda;

  if (seed > 0)
    t->seed = seed;

  if (t->N <= 0)
    t->N = dim;

  N = t->N;
  if (N == 0)
    FATAL("readpara_readpara_t(): problem dimension N undefined.\n",
          "  (no default value available).",0,0);
  if (t->xstart == NULL && inxstart == NULL && t->typicalX == NULL) {
    ERRORMESSAGE("Warning: initialX undefined. typicalX = 0.5...0.5 used.","","","");
    printf("\nWarning: initialX undefined. typicalX = 0.5...0.5 used.\n");
  }
  if (t->rgInitialStds == NULL && inrgsigma == NULL) {
    /* FATAL("initialStandardDeviations undefined","","",""); */
    ERRORMESSAGE("Warning: initialStandardDeviations undefined. 0.3...0.3 used.","","","");
    printf("\nWarning: initialStandardDeviations. 0.3...0.3 used.\n");
  }

  if (t->xstart == NULL) {
    t->xstart = new_double(N);

    /* put inxstart into xstart */
    if (inxstart != NULL) {
      for (i=0; i<N; ++i)
        t->xstart[i] = inxstart[i];
    }
    /* otherwise use typicalX or default */
    else {
      t->typicalXcase = 1;
      for (i=0; i<N; ++i)
        t->xstart[i] = (t->typicalX == NULL) ? 0.5 : t->typicalX[i];
    }
  } /* xstart == NULL */

  if (t->rgInitialStds == NULL) {
    t->rgInitialStds = new_double(N);
    for (i=0; i<N; ++i)
      t->rgInitialStds[i] = (inrgsigma == NULL) ? 0.3 : inrgsigma[i];
  }

  readpara_SupplementDefaults(t);
  if (strcmp(filename, "non") != 0)
    readpara_WriteToFile(t, "actparcmaes.par", filename);
} /* readpara_init */

/* --------------------------------------------------------- */
/* --------------------------------------------------------- */
void readpara_exit(readpara_t *t)
{
  if (t->xstart != NULL) /* not really necessary */
    free( t->xstart);
  if (t->typicalX != NULL)
    free( t->typicalX);
  if (t->rgInitialStds != NULL)
    free( t->rgInitialStds);
  if (t->rgDiffMinChange != NULL)
    free( t->rgDiffMinChange);
  if (t->weights != NULL)
    free( t->weights);
```

```c
  free(t->rgsformat);
  free(t->rgpadr);
  free(t->rgskeyar);
  free(t->rgp2adr);
  free(t->weigkey);
}

/* --------------------------------------------------------- */
/* --------------------------------------------------------- */
void
readpara_ReadFromFile(readpara_t *t, const char * filename)
{
  char s[1000], *ss = "initials.par";
  int ipara, i;
  int size;
  FILE *fp;
  if (filename == NULL)
    filename = ss;
  fp = fopen( filename, "r");
  if(fp == NULL) {
    ERRORMESSAGE("cmaes_ReadFromFile(): could not open '", filename, "'",0);
    return;
  }
  for (ipara=0; ipara < t->n1para; ++ipara)
    {
      rewind(fp);
      while(fgets(s, sizeof(s), fp) != NULL)
        { /* skip comments  */
          if (s[0] == '#' || s[0] == '%')
            continue;
          if(sscanf(s, t->rgsformat[ipara], t->rgpadr[ipara]) == 1) {
            if (strncmp(t->rgsformat[ipara], " stopFitness ", 13) == 0)
              t->stStopFitness.flg = 1;
            break;
          }
        }
    } /* for */
  if (t->N <= 0)
    FATAL("readpara_ReadFromFile(): No valid dimension N",0,0,0);
  for (ipara=0; ipara < t->n2para; ++ipara)
    {
      rewind(fp);
      while(fgets(s, sizeof(s), fp) != NULL) /* read one line */
        { /* skip comments  */
          if (s[0] == '#' || s[0] == '%')
            continue;
          if(sscanf(s, t->rgskeyar[ipara], &size) == 1) { /* size==number of val
ues to be read */
            if (size > 0) {
              *t->rgp2adr[ipara] = new_double(t->N);
              for (i=0;i<size&&i<t->N;++i) /* start reading next line */
                if (fscanf(fp, " %lf", &(*t->rgp2adr[ipara])[i]) != 1)
                  break;
              if (i<size && i < t->N) {
                ERRORMESSAGE("readpara_ReadFromFile ", filename, ": ",0);
                FATAL( "'", t->rgskeyar[ipara],
                       "' not enough values found.\n",
                       "  Remove all comments between numbers.");
              }
              for (; i < t->N; ++i) /* recycle */
                (*t->rgp2adr[ipara])[i] = (*t->rgp2adr[ipara])[i%size];
            }
          }
        }
    } /* for */
  fclose(fp);
  return;
} /* readpara_ReadFromFile() */

/* --------------------------------------------------------- */
/* --------------------------------------------------------- */
void
readpara_WriteToFile(readpara_t *t, const char *filenamedest,
                     const char *filenamesource)
  {
    int ipara, i;
    size_t len;
    time_t ti = time(NULL);
    FILE *fp = fopen( filenamedest, "a");
    if(fp == NULL) {
      ERRORMESSAGE("cmaes_WriteToFile(): could not open '",
                   filenamedest, "'",0);
      return;
    }
    fprintf(fp, "\n# Read from %s at %s\n", filenamesource,
            asctime(localtime(&ti))); /* == ctime() */
    for (ipara=0; ipara < 1; ++ipara) {
      fprintf(fp, t->rgsformat[ipara], *(int *)t->rgpadr[ipara]);
      fprintf(fp, "\n");
    }
    for (ipara=0; ipara < t->n2para; ++ipara) {
      if(*t->rgp2adr[ipara] == NULL)
        continue;
      fprintf(fp, t->rgskeyar[ipara], t->N);
      fprintf(fp, "\n");
      for (i=0; i<t->N; ++i)
        fprintf(fp, "%7.3g%c", (*t->rgp2adr[ipara])[i], (i%5==4)?'\n':' ');
      fprintf(fp, "\n");
    }
    for (ipara=1; ipara < t->n1outpara; ++ipara) {
      if (strncmp(t->rgsformat[ipara], " stopFitness ", 13) == 0)
        if(t->stStopFitness.flg == 0) {
          fprintf(fp, " stopFitness\n");
          continue;
        }
      len = strlen(t->rgsformat[ipara]);
      if (t->rgsformat[ipara][len-1] == 'd') /* read integer */
        fprintf(fp, t->rgsformat[ipara], *(int *)t->rgpadr[ipara]);
      else if (t->rgsformat[ipara][len-1] == 's') /* read string */
        fprintf(fp, t->rgsformat[ipara], (char *)t->rgpadr[ipara]);
      else {
        if (strncmp(" fac*", t->rgsformat[ipara], 5) == 0) {
          fprintf(fp, " ");
          fprintf(fp, t->rgsformat[ipara]+5, *(double *)t->rgpadr[ipara]);
        } else
          fprintf(fp, t->rgsformat[ipara], *(double *)t->rgpadr[ipara]);
      }
      fprintf(fp, "\n");
    } /* for */
    fprintf(fp, "\n");
    fclose(fp);
  } /* readpara_WriteToFile() */

  /* --------------------------------------------------------- */
  /* --------------------------------------------------------- */
  void
  readpara_SupplementDefaults(readpara_t *t)
  {
    double t1, t2;
    int N = t->N;
    clock_t cloc = clock();

    if (t->seed < 1) {
      while ((int) (cloc - clock()) == 0)
        ; /* TODO: remove this for time critical applications!? */
      t->seed = (unsigned int)abs(100*time(NULL)+clock());
    }

    if (t->stStopFitness.flg == -1)
      t->stStopFitness.flg = 0;

    if (t->lambda < 2)
      t->lambda = 4+(int)(3*log((double)N));
    if (t->mu == -1) {
      t->mu = t->lambda/2;
      readpara_SetWeights(t, t->weigkey);
    }
    if (t->weights == NULL)
      readpara_SetWeights(t, t->weigkey);
```

```c

    if (t->cs > 0) /* factor was read */
      t->cs *= (t->mueff + 2.) / (N + t->mueff + 3.);
    if (t->cs <= 0 || t->cs >= 1)
      t->cs = (t->mueff + 2.) / (N + t->mueff + 3.);

    if (t->ccumcov <= 0 || t->ccumcov > 1)
      t->ccumcov = 4. / (N + 4);

    if (t->mucov < 1) {
      t->mucov = t->mueff;
    }
    t1 = 2. / ((N+1.4142)*(N+1.4142));
    t2 = (2.*t->mueff-1.) / ((N+2.)*(N+2.)+t->mueff);
    t2 = (t2 > 1) ? 1 : t2;
    t2 = (1./t->mucov) * t1 + (1.-1./t->mucov) * t2;
    if (t->ccov >= 0) /* ccov holds the read factor */
      t->ccov *= t2;
    if (t->ccov < 0 || t->ccov > 1) /* set default in case */
      t->ccov = t2;

    if (t->stopMaxFunEvals == -1)  /* may depend on ccov in near future */
      t->stopMaxFunEvals = t->facmaxeval*900*(N+3)*(N+3);
    else
      t->stopMaxFunEvals *= t->facmaxeval;

    if (t->stopMaxIter == -1)
      t->stopMaxIter = ceil((double)(t->stopMaxFunEvals / t->lambda));

    if (t->damps < 0)
      t->damps = 1; /* otherwise a factor was read */
    t->damps = t->damps
      * (1 + 2*douMax(0., sqrt((t->mueff-1.)/(N+1.)) - 1))     /* basic factor */
      * douMax(0.3, 1. -                                       /* modify for short
 runs */
                (double)N / (1e-6+douMin(t->stopMaxIter, t->stopMaxFunEvals/t-
>lambda)))
      + t->cs;                                                 /* minor increment
*/

    if (t->updateCmode.modulo < 1)
      t->updateCmode.modulo = 1./t->ccov/(double)(N)/10.;
    t->updateCmode.modulo *= t->facupdateCmode;
    if (t->updateCmode.maxtime < 0)
      t->updateCmode.maxtime = 0.20; /* maximal 20% of CPU-time */

  } /* readpara_SupplementDefaults() */


  /* --------------------------------------------------------- */
  /* --------------------------------------------------------- */
  void
  readpara_SetWeights(readpara_t *t, const char * mode)
  {
    double s1, s2;
    int i;

    if(t->weights != NULL)
      free( t->weights);
    t->weights = new_double(t->mu);
    if (strcmp(mode, "lin") == 0)
      for (i=0; i<t->mu; ++i)
        t->weights[i] = t->mu - i;
    else if (strncmp(mode, "equal", 3) == 0)
      for (i=0; i<t->mu; ++i)
        t->weights[i] = 1;
    else if (strcmp(mode, "log") == 0)
      for (i=0; i<t->mu; ++i)
        t->weights[i] = log(t->mu+1.)-log(i+1.);
    else
      for (i=0; i<t->mu; ++i)
        t->weights[i] = log(t->mu+1.)-log(i+1.);

    /* normalize weights vector and set mueff */
    for (i=0, s1=0, s2=0; i<t->mu; ++i) {
```

```
    s1 += t->weights[i];
    s2 += t->weights[i]*t->weights[i];
  }
  t->mueff = s1*s1/s2;
  for (i=0; i<t->mu; ++i)
    t->weights[i] /= s1;

  if(t->mu < 1 || t->mu > t->lambda ||
     (t->mu==t->lambda && t->weights[0]==t->weights[t->mu-1]))
    FATAL("readpara_SetWeights(): invalid setting of mu or lambda",0,0,0);

} /* readpara_SetWeights() */

/* --------------------------------------------------------- */
/* --------------------------------------------------------- */
static int
intMin( int i, int j)
{
  return i < j ? i : j;
}
static double
douMax( double i, double j)
{
  return i > j ? i : j;
}
static double
douMin( double i, double j)
{
  return i < j ? i : j;
}
static double
rgdouMax( const double *rgd, int len)
{
  int i;
  double max = rgd[0];
  for (i = 1; i < len; ++i)
    max = (max < rgd[i]) ? rgd[i] : max;
  return max;
}

static double
rgdouMin( const double *rgd, int len)
{
  int i;
  double min = rgd[0];
  for (i = 1; i < len; ++i)
    min = (min > rgd[i]) ? rgd[i] : min;
  return min;
}

static int
MaxIdx( const double *rgd, int len)
{
  int i, res;
  for(i=1, res=0; i<len; ++i)
    if(rgd[i] > rgd[res])
      res = i;
  return res;
}
static int
MinIdx( const double *rgd, int len)
{
  int i, res;
  for(i=1, res=0; i<len; ++i)
    if(rgd[i] < rgd[res])
      res = i;
  return res;
}

static double
myhypot(double a, double b)
/* sqrt(a^2 + b^2) numerically stable. */
{
  double r = 0;
  if (fabs(a) > fabs(b)) {
```

```
    r = b/a;
    r = fabs(a)*sqrt(1+r*r);
  } else if (b != 0) {
    r = a/b;
    r = fabs(b)*sqrt(1+r*r);
  }
  return r;
}

static int SignOfDiff(const void *d1, const void * d2)
{
  return *((double *) d1) > *((double *) d2) ? 1 : -1;
}

#if 1
/* dirty index sort */
static void Sorted_index(const double *rgFunVal, int *index, int n)
{
  int i, j;
  for (i=1, index[0]=0; i<n; ++i) {
    for (j=i; j>0; --j) {
      if (rgFunVal[index[j-1]] < rgFunVal[i])
        break;
      index[j] = index[j-1]; /* shift up */
    }
    index[j] = i; /* insert i */
  }
}
#endif

static void * new_void(int n, size_t size)
{
  static char s[70];
  void *p = calloc((unsigned) n, size);
  if (p == NULL) {
    sprintf(s, "new_void(): calloc(%ld,%ld) failed",(long)n,(long)size);
    FATAL(s,0,0,0);
  }
  return p;
}

double *
cmaes_NewDouble(int n)
{
  return new_double(n);
}

static double * new_double(int n)
{
  static char s[170];
  double *p = (double *) calloc((unsigned) n, sizeof(double));
  if (p == NULL) {
    sprintf(s, "new_double(): calloc(%ld,%ld) failed",
            (long)n,(long)sizeof(double));
    FATAL(s,0,0,0);
  }
  return p;
}

/* --------------------------------------------------------- */
/* --------------------------------------------------------- */

/* ========================================================= */
void
cmaes_FATAL(char const *s1, char const *s2, char const *s3,
            char const *s4)
{
  time_t t = time(NULL);
  ERRORMESSAGE( s1, s2, s3, s4);
  ERRORMESSAGE("*** Exiting cmaes_t ***",0,0,0);
  printf("\n -- %s %s\n", asctime(localtime(&t)),
           s2 ? szCat(s1, s2, s3, s4) : s1);
  printf(" *** CMA-ES ABORTED, see errcmaes.err *** \n");
  fflush(stdout);
  exit(1);
```

```
}

/* ========================================================= */
static void
FATAL(char const *s1, char const *s2, char const *s3,
      char const *s4)
{
  cmaes_FATAL(s1, s2, s3, s4);
}

/* ========================================================= */
void ERRORMESSAGE( char const *s1, char const *s2,
                   char const *s3, char const *s4)
{
#if 1
  /*  static char szBuf[700];  desirable but needs additional input argument
      sprintf(szBuf, "%f:%f", gen, gen*lambda);
  */
  time_t t = time(NULL);
  FILE *fp = fopen( "errcmaes.err", "a");
  if (!fp)
    {
      printf("\nFATAL ERROR: %s\n", s2 ? szCat(s1, s2, s3, s4) : s1);
      printf("cmaes_t could not open file 'errcmaes.err'.");
      printf("\n *** CMA-ES ABORTED *** ");
      fflush(stdout);
      exit(1);
    }
  fprintf( fp, "\n -- %s %s\n", asctime(localtime(&t)),
           s2 ? szCat(s1, s2, s3, s4) : s1);
  fclose (fp);
#endif
}

/* ========================================================= */
char *szCat(const char *sz1, const char*sz2,
            const char *sz3, const char *sz4)
{
  static char szBuf[700];

  if (!sz1)
    FATAL("szCat() : Invalid Arguments",0,0,0);

  strncpy ((char *)szBuf, sz1, (unsigned)intMin( (int)strlen(sz1), 698));
  szBuf[intMin( (int)strlen(sz1), 698)] = '\0';
  if (sz2)
    strncat ((char *)szBuf, sz2,
             (unsigned)intMin((int)strlen(sz2)+1, 698 - (int)strlen((char const
*)szBuf)));
  if (sz3)
    strncat((char *)szBuf, sz3,
            (unsigned)intMin((int)strlen(sz3)+1, 698 - (int)strlen((char const *
)szBuf)));
  if (sz4)
    strncat((char *)szBuf, sz4,
            (unsigned)intMin((int)strlen(sz4)+1, 698 - (int)strlen((char const *
)szBuf)));
  return (char *) szBuf;
}
```

Bibliography

[1] C. N. Cohen-Tannoudji, “Nobel lecture: Manipulating atoms with photons,” *Rev. Mod. Phys.*, vol. 70, pp. 707–719, Jul 1998.

[2] E. A. Cornell and C. E. Wieman, “Nobel lecture: Bose-einstein condensation in a dilute gas, the first 70 years and some recent experiments,” *Rev. Mod. Phys.*, vol. 74, pp. 875–893, Aug 2002.

[3] M. Greiner, O. Mandel, T. Esslinger, T. W. Hänsch, and I. Bloch, “Quantum phase transition from a superfluid to a Mott insulator in a gas of ultracold atoms,” *Nature*, vol. 415, pp. 39–44, January 2002.

[4] H. L. Bethlem, G. Berden, and G. Meijer, “Decelerating neutral dipolar molecules,” *Phys. Rev. Lett.*, vol. 83, pp. 1558–1561, Aug 1999.

[5] E. Vliegen and F. Merkt, “On the electrostatic deceleration of argon atoms in high Rydberg states by time-dependent inhomogeneous electric fields,” *Journal of Physics B: Atomic, Molecular and Optical Physics*, vol. 38, no. 11, pp. 1623–1636, 2005.

[6] N. Vanhaecke, U. Meier, M. Andrist, B. H. Meier, and F. Merkt, “Multistage Zeeman deceleration of hydrogen atoms,” *Physical Review A (Atomic, Molecular, and Optical Physics)*, vol. 75, no. 3, p. 031402, 2007.

[7] S. D. Hogan, D. Sprecher, M. Andrist, N. Vanhaecke, and F. Merkt, “Zeeman deceleration of H and D,” *Physical Review A (Atomic, Molecular, and Optical Physics)*, vol. 76, no. 2, p. 023412, 2007.

[8] S. D. Hogan, A. W. Wiederkehr, H. Schmutz, and F. Merkt, “Magnetic trapping of hydrogen after multistage Zeeman deceleration,” *Physical Review Letters*, vol. 101, no. 14, p. 143001, 2008.

[9] E. Narevicius, A. Libson, C. G. Parthey, I. Chavez, J. Narevicius, U. Even, and M. G. Raizen, “Stopping supersonic beams with a series of pulsed electromagnetic coils: An atomic coilgun,” *Physical Review Letters*, vol. 100, no. 9, p. 093003, 2008.

[10] T. Bergeman, G. Erez, and H. J. Metcalf, “Magnetostatic trapping fields for neutral atoms,” *Phys. Rev. A*, vol. 35, no. 4, pp. 1535–1546, 1987.

[11] I. R. Sims and I. W. M. Smith, “Gas-phase reactions and energy transfer at very low temperatures,” *Annual Review of Physical Chemistry*, vol. 46, no. 1, pp. 109–138, 1995.

[12] H. L. Bethlem and G. Meijer, “Production and application of translationally cold molecules,” *International Reviews in Physical Chemistry*, vol. 22, no. 1, pp. 73–128, 2003.

[13] N. Hansen and A. Ostermeier, “Completely derandomized self-adaptation in evolution strategies,” *Evol. Comput.*, vol. 9, no. 2, pp. 159–195, 2001.

[14] N. Hansen, “The CMA Evolution Strategy: a comparing review,” in *Towards a new evolutionary computation. Advances on estimation of distribution algorithms* (J. Lozano, P. Larranaga, I. Inza, and E. Bengoetxea, eds.), pp. 75–102, Springer, 2006.

[15] P. J. Mohr, B. N. Taylor, and D. B. Newell, “CODATA recommended values of the fundamental physical constants: 2006,” *Reviews of Modern Physics*, vol. 80, no. 2, p. 633, 2008.

[16] A. Messiah, *Quantum Mechanics, Two Volumes Bound as One*, ch. 11, pp. 415,416. Dover Publications, Inc., Mineola, New York, 1999.

[17] W. Gerlach and O. Stern, “Der experimentelle Nachweis der Richtungsquantelung im Magnetfeld,” *Zeitschrift für Physik A*, vol. 9, no. 1, pp. 349–352, 1922.

[18] A. Messiah, *Quantum Mechanics, Two Volumes Bound as One*, ch. 20, p. 875ff. Dover Publications, Inc., Mineola, New York, 1999.

[19] R. P. Feynman, R. B. Leighton, and M. L. Sands, *The Feynman Lectures on physics, Vol. 3, Quantum Mechanics*, ch. 12. Addison-Wesley, Reading, Massachusetts, 1965.

[20] G. K. Woodgate, *Elementary Atomic Structure*, ch. 9, p. 180. Clarendon Press, Oxford, 1980.

[21] H. Haberland, U. Buck, and M. Tolle, “Velocity distribution of supersonic nozzle beams,” *Review of Scientific Instruments*, vol. 56, no. 9, pp. 1712–1716, 1985.

[22] T. A. Paul, *Development and spectroscopic applications of a solid-state vacuum ultraviolet laser system in atomic and molecular physics*. PhD thesis, ETH Zurich, 2008.

[23] M. Abramowitz and I. A. Stegun, *Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables*, ch. 17.8, pp. 600–607. New York: Dover, ninth dover printing, tenth gpo printing ed., 1964.

[24] W. H. Wing, “On neutral particle trapping in quasistatic electromagnetic fields,” *Progress in Quantum Electronics*, vol. 8, no. 3-4, pp. 181 – 199, 1984.

[25] J. A. G. Roberts and G. R. W. Quispel, “Chaos and time-reversal symmetry. order and chaos in reversible dynamical systems,” *Physics Reports*, vol. 216, no. 2-3, pp. 63 – 177, 1992.

[26] R. D. Skeel, G. Zhang, and T. Schlick, “A family of symplectic integrators: stability, accuracy, and molecular dynamics applications,” *SIAM J. Sci. Comput*, vol. 18, pp. 203–222, 1997.

[27] L. Verlet, “Computer "experiments" on classical fluids. i. thermodynamical properties of Lennard-Jones molecules,” *Phys. Rev.*, vol. 159, p. 98, Jul 1967.

[28] E. Gamma, R. Helm, R. E. Johnson, and J. M. Vlissides, “Design patterns: Abstraction and reuse of object-oriented design,” in *ECOOP ’93: Proceedings of the 7th European Conference on Object-Oriented Programming*, (London, UK), pp. 406–431, Springer-Verlag, 1993.

[29] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Professional, 1995.

[30] J. Nocedal and S. J. Wright, *Numerical Optimization*. Springer, August 1999.

[31] H.-G. Beyer and H.-P. Schwefel, "Evolution strategies – a comprehensive introduction," *Natural Computing: an international journal*, vol. 1, no. 1, pp. 3–52, 2002.

[32] J. J. Gilijamse, J. Küpper, S. Hoekstra, N. Vanhaecke, S. Y. T. van de Meerakker, and G. Meijer, "Optimizing the Stark-decelerator beamline for the trapping of cold molecules using evolutionary strategies," *Physical Review A (Atomic, Molecular, and Optical Physics)*, vol. 73, no. 6, p. 063410, 2006.

[33] S. Kern, S. D. Müller, N. Hansen, D. Büche, J. Ocenasek, and P. Koumoutsakos, "Learning probability distributions in continuous evolutionary algorithms – a comparative review," *Natural Computing: an international journal*, vol. 3, no. 1, pp. 77–112, 2004.

[34] N. Hansen, S. Muller, and P. Koumoutsakos, "Reducing the time complexity of the derandomized Evolution Strategy with covariance matrix adaptation (CMA-ES)," *Evolutionary Computation*, vol. 11, no. 1, pp. 1–18, 2003.

[35] N. Hansen, "Evaluating the CMA Evolution Strategy on multimodal test functions," in *Parallel Problem Solving from Nature - PPSN VIII*, pp. 282–291, Springer, 2004.

[36] N. Hansen, "CMA Evolution Strategy source code." `http://www.lri.fr/~hansen/cmaes_inmatlab.html`. Accessed 15 December, 2009.

[37] P. Arbenz and W. Petersen, *Introduction to Parallel Computing: A Practical Guide with Examples in C*. Oxford University Press, 2004.

[38] C. Igel, N. Hansen, and S. Roth, "Covariance matrix adaptation for multi-objective optimization," *Evol. Comput.*, vol. 15, no. 1, pp. 1–28, 2007.

[39] "GNU Make - GNU Project - Free Software Foundation (FSF)." `http://www.gnu.org/software/make/`. Accessed 15 December, 2009.