Geosci. Model Dev., 8, 1005–1032, 2015
www.geosci-model-dev.net/8/1005/2015/
doi:10.5194/gmd-8-1005-2015

Geoscientific Model Development
Open Access

libmpdata++ 1.0: a library of parallel MPDATA solvers for systems of generalised transport equations

A. Jaruga[1], **S. Arabas**[1], **D. Jarecka**[1,2], **H. Pawlowska**[1], **P. K. Smolarkiewicz**[3], **and M. Waruszewski**[1]

[1]Institute of Geophysics, Faculty of Physics, University of Warsaw, Warsaw, Poland
[2]National Center for Atmospheric Research, Boulder, CO, USA
[3]European Centre for Medium-Range Weather Forecasts, Reading, UK

Correspondence to: A. Jaruga (ajaruga@igf.fuw.edu.pl) and H. Pawlowska (hanna.pawlowska@igf.fuw.edu.pl)

Received: 18 August 2014 – Published in Geosci. Model Dev. Discuss.: 26 November 2014
Revised: 2 March 2015 – Accepted: 3 March 2015 – Published: 8 April 2015

Abstract. This paper accompanies the first release of libmpdata++, a C++ library implementing the multi-dimensional positive-definite advection transport algorithm (MPDATA) on regular structured grid. The library offers basic numerical solvers for systems of generalised transport equations. The solvers are forward-in-time, conservative and non-linearly stable. The libmpdata++ library covers the basic second-order-accurate formulation of MPDATA, its third-order variant, the infinite-gauge option for variable-sign fields and a flux-corrected transport extension to guarantee non-oscillatory solutions. The library is equipped with a non-symmetric variational elliptic solver for implicit evaluation of pressure gradient terms. All solvers offer parallelisation through domain decomposition using shared-memory parallelisation.

The paper describes the library programming interface, and serves as a user guide. Supported options are illustrated with benchmarks discussed in the MPDATA literature. Benchmark descriptions include code snippets as well as quantitative representations of simulation results. Examples of applications include homogeneous transport in one, two and three dimensions in Cartesian and spherical domains; a shallow-water system compared with analytical solution (originally derived for a 2-D case); and a buoyant convection problem in an incompressible Boussinesq fluid with interfacial instability. All the examples are implemented out of the library tree. Regardless of the differences in the problem dimensionality, right-hand-side terms, boundary conditions and parallelisation approach, all the examples use the same unmodified library, which is a key goal of libmpdata++ design. The design, based on the principle of separation of concerns, prioritises the user and developer productivity. The libmpdata++ library is implemented in C++, making use of the Blitz++ multi-dimensional array containers, and is released as free/libre and open-source software.

1 Introduction

The MPDATA advection scheme introduced in Smolarkiewicz (1983) has grown into a family of numerical algorithms for geosciences and beyond (see for example Grabowski and Smolarkiewicz, 2002; Cotter et al., 2002; Smolarkiewicz and Szmelter, 2009; Ortiz and Smolarkiewicz, 2009; Hyman et al., 2012; Charbonneau and Smolarkiewicz, 2013). MPDATA stands for multi-dimensional positive-definite advection transport algorithm[1]. It is a finite-difference/finite-volume algorithm for solving the generalised transport equation

$$\partial_t(G\psi) + \nabla \cdot (G\boldsymbol{u}\psi) = GR. \tag{1}$$

Equation (1) describes the advection of a scalar field ψ in a flow with velocity $\boldsymbol{u}$. The field R on the right-hand side (rhs) is a total of source/sink terms. The scalar field G can represent the fluid density, the Jacobian of coordinate transformation or their product and satisfies the equation

$$\partial_t(G) + \nabla \cdot (G\boldsymbol{u}) = 0. \tag{2}$$

[1]In fact, MPDATA is sign-preserving, rather than merely positive-definite, but for historical reasons the name remains unchanged.

In the homogeneous case ($R \equiv 0$), MPDATA is at least second-order-accurate in space and time, conservative and non-linearly stable.

The history of MPDATA spans 3 decades: Smolarkiewicz (1984)–Kühnlein et al. (2012), Smolarkiewicz et al. (2014) and is widely documented in the literature – see Smolarkiewicz and Margolin (1998), Smolarkiewicz (2006) and Prusa et al. (2008) for reviews. Notwithstanding, from the authors' experience the software engineering aspects still overshadow the benefits of MPDATA. To facilitate the use of MPDATA schemes, hereby we present a new implementation of the MPDATA family of algorithms for regular structured grids – libmpdata++.

In the development of libmpdata++ we strive to comply with the best sought-after practices among the scientific community (Wilson et al., 2014), in particular with the paradigm of maximising code reuse. This paradigm is embodied in the "open source computational libraries – the main foundation upon which academic and also a significant part of industrial computational research rests" (Bangerth and Heister, 2013).

The libmpdata++ has been developed in C++[2], making extensive use of object-oriented programming (OOP) and template programming. The primary goals when designing libmpdata++ were to maintain strict separation of concerns and to reproduce within the code the mathematical "blackboard abstractions" used for documenting numerical algorithms. The adopted design contributes to the readability, maintainability and conciseness of the code. The current development of libmpdata++ is an extension of the research on OOP implementation of the basic MPDATA scheme presented in Arabas et al. (2014).

The goal of this article is twofold: first, to document the library interface by providing usage examples and, second, to validate the correctness of the implementation by verifying the results against published benchmarks.

The structure of the paper is as follows. Section 2 outlines the library design. The four sections that follow correspond to four types of equation systems solved by the implemented algorithms, namely homogeneous advective transport, inhomogeneous transport, transport with prognosed velocity, and systems featuring elliptic pressure equation. Each of these sections outlines the implemented algorithms, describes the library interface and provides usage examples. Each example is accompanied with a definition of the solved problem, description of the program code and discussion of the results. An index of libmpdata++ options documented in the article is provided in Appendix A.

The paper structure reflects the solver inheritance hierarchy in libmpdata++. All features discussed in preceding sections apply to the one that follows. The set of discussed problems was selected to match the tutorial structure of the paper. The presentation begins with simple examples focusing on the basic library interface. Subsequent examples use increasingly more complicated cases with the most complex reflecting potential for applications to cloud dynamics (Grabowski and Smolarkiewicz, 2002).

The library and programs used to generate all results presented in the paper are released as free and open-source software – see the section on code availability at the end of the paper.

2 Library design

2.1 Dependencies and supported platforms

The libmpdata++ package is a header-only C++ library. It is built upon the Blitz++[3] array containers. We refer the reader to the Blitz++ documentation (Veldhuizen, 2006) for description of the Blitz++ interface, to which the user is exposed while working with libmpdata++. The libmpdata++ core also depends on several components of the Boost[4] library collection, however these are used internally only. Output handlers included in the library depend additionally on gnuplot-iostream[5] and HDF5[6], but their use is optional. Example programs discussed within this article require gnuplot[7], ParaView[8], Python, and the following Python packages: h5py[9], matplotlib[10] and scipy[11].

The library code requires a C++11-compliant compiler. In the current development workflow, we employ continuous integration on Linux with GNU g++[12] and LLVM clang++[13] compilers and on Apple OSX with the Apple clang++[14] compiler. Consequently, these are considered the supported platforms.

2.2 Components

Components of the library are grouped as follows.

– Solvers:
  – **mpdata** intended for solving homogeneous transport problems (Sect. 3)
  – **mpdata_rhs** extending the above with rhs term handling (Sect. 4)
  – **mpdata_rhs_vip** adding prognosed-velocity support (Sect. 5)

[2]In the C++11 revision of the language.

[3]see http://sf.net/projects/blitz/
[4]see http://boost.org/
[5]see http://gitorious.org/gnuplot-iostream/
[6]see http://hdfgroup.org/HDF5/
[7]see http://gnuplot.info/
[8]see http://paraview.org/
[9]see http://h5py.org/
[10]see http://matplotlib.org/
[11]see http://scipy.org/
[12]see http://gcc.gnu.org/
[13]see http://llvm.org/
[14]see http://apple.com/xcode

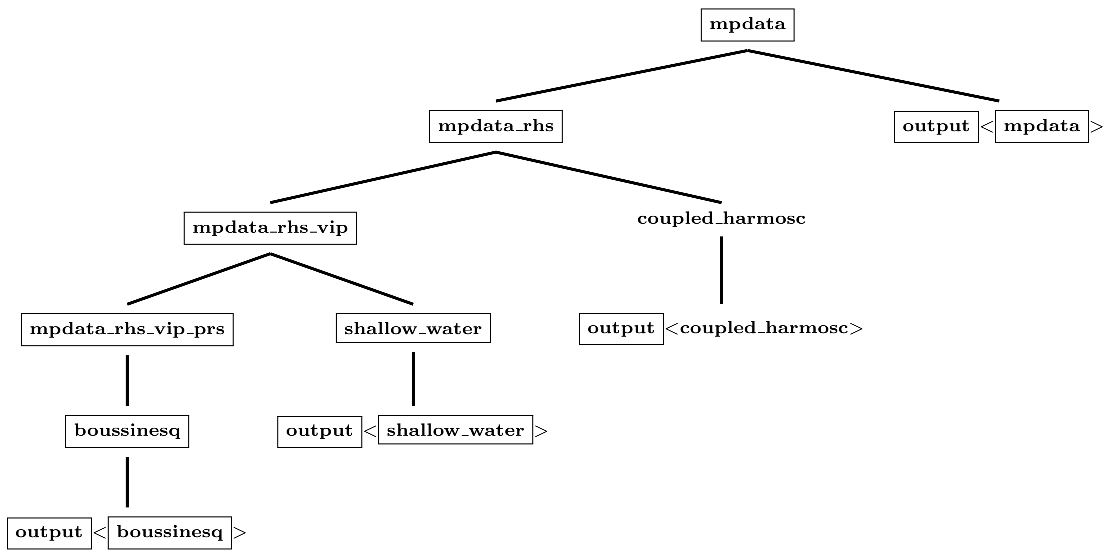

Figure 1. Inheritance diagram of classes mentioned in the paper. Classes defined within libmpdata++ have their names surrounded with black frames. The **coupled_harmosc** class is an example of a user-defined class defined out of the library tree. The solid black lines show the inheritance relations. The **output** label depicts any of the output handlers available in libmpdata++.

  - **mpdata_rhs_vip_prs** further extending the above with elliptic pressure equation solvers (Sect. 6).

- Output handlers:
  - **gnuplot** offering direct communication with the gnuplot program with no intermediate output files
  - **hdf5** offering basic HDF5 output compatible with netCDF[15] readers
  - **hdf5_xdmf** implementing the eXtensible Data Model and Format[16] standard supported for instance by the ParaView visualisation tool.

- Boundary conditions:
  - **cyclic** implementing periodic boundaries
  - **open** giving zero-divergence condition on domain edges
  - **polar** applicable with spherical coordinates.

- Concurrency handlers:
  - **serial** for single-thread operation
  - **cxx11_thread** for multi-threading using C++11 Thread support library
  - **boost_thread** for multi-threading using Boost.Thread
  - **openmp** for multi-threading using OpenMP
  - **threads** that defaults to **openmp** if supported by the compiler and falls back to **boost_thread** otherwise.

Performing integration with libmpdata++ requires choosing one of the solvers, one output handler, one boundary condition per each domain edge and one concurrency handler.

The inheritance diagram in Fig. 1 shows relationships between libmpdata++ solvers defined within the library. The diagram includes as well an example user-defined class **coupled_harmosc** defined out of the library tree. The **mpdata** solver is displayed at the top, as it is the base class for all other classes.

2.3 Computational domain and grid

The arrangement of the computational domain used in libmpdata++ is shown in Fig. 2. The initial condition for the dependent variable ψ is assumed to be known in $nx \times ny$ data points. The outermost data points are located at the boundaries of the domain.

The dual, staggered Arakawa-C grid (Arakawa and Lamb, 1977) used in libmpdata++ is shown in Fig. 3. In this spatial discretisation approach, the cell-mean values of the scalar fields ψ, and G reside in the centres of computational cells, – corresponding to the data points of the primary grid in Fig. 2 – whereas the components of the velocity field $\boldsymbol{u}$ are specified at the cell edges of the dual grid in Fig. 3.

2.4 Error and progress reporting

There are several error-handling mechanisms used within libmpdata++.

[15]see http://www.unidata.ucar.edu/software/netcdf/
[16]see http://xdmf.org/

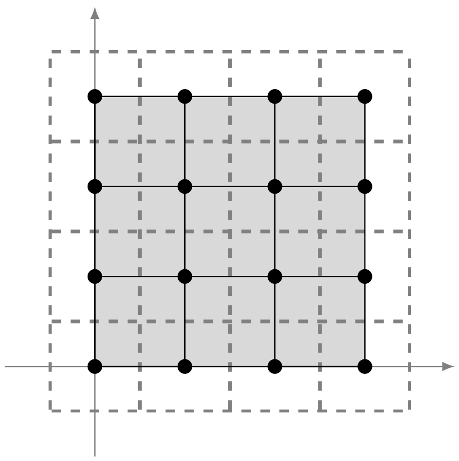

Figure 2. Schematic of a 2-D computational domain. Bullets mark the data points for the dependent variable ψ in Eq. (1), solid lines depict edges of primary grid and dashed lines mark edges of dual grid in Fig. 3.

First, there are sanity checks within the code implemented using **static_assert()** calls. These are reported during compilation, for instance when invalid values of compile-time parameters are supplied.

Second, there are available numerous run-time sanity checks, implemented using **assert()** calls. These are often time-consuming and are not intended to be executed in production runs. To disable them, one needs to compile the program using libmpdata++ with the **-DNDEBUG** compiler flag. Examples of such checks include detection of **NaN** values within the model state variables, which may be useful to trace origins of numerical instability problems.

Third, the user may chose to activate the Blitz++ debug mode that enables run-time array range checks. Activating Blitz++ debug mode requires compiling the program using libmpdata++ with the **-DBZ_DEBUG** flag and linking with **libblitz**.

Finally, libmpdata++ reports run-time errors by throwing **std::runtime_error** exceptions.

Simulation progress is communicated to the user by continuously updating the process threads' names with the percentage of work completed (can be observed e.g. by invoking **top -H**).

3 Advective transport

The focus of this section is on the advection algorithm used within libmpdata++. Section 3.1 provides a short introduction to the implemented MPDATA scheme. Section 3.2 describes the library interface needed for the homogeneous transport cases. The following Sects. 3.3–3.8 show examples of usage of libmpdata++ along with the references to other MPDATA benchmarks.

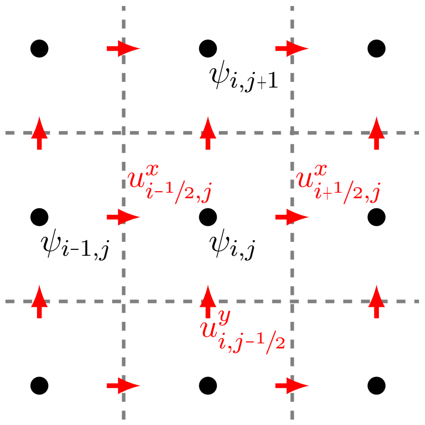

Figure 3. A schematic of a 2-D Arakawa-C grid. Bullets denote the cell centres and dashed lines denote the cell walls corresponding to the dual grid in Fig. 2.

3.1 Implemented algorithms

This subsection is intended to provide the reader with an outline of selected MPDATA features that correspond to the options presently available in libmpdata++. For the full derivation of the scheme and its options see the reviews in Smolarkiewicz and Margolin (1998) and Smolarkiewicz (2006), whereas for an extended discussion of stability, positivity and convexity see Smolarkiewicz and Szmelter (2005).

In the present implementation, it is assumed that G is constant in time. Consequently, the governing homogeneous transport Eq. (1) can be written as

$$\partial_t \psi + \frac{1}{G} \nabla \cdot (G \boldsymbol{u} \psi) = 0. \tag{3}$$

This particular form is solved by the **mpdata** solver of libmpdata++.

The following paragraphs will focus on the algorithms used for handling Eq. (3). The rules for applying source and sink terms are presented in Sect. 4.

3.1.1 Basic MPDATA

MPDATA is an, at least, second-order-accurate iterative scheme in which all iterations take the form of a first-order-accurate donor-cell pass (alias upwind, upstream; cf. Press et al., 2007, Sect. 20.1.3). For the one-dimensional[17] case, after the discretisation in space (subscripts i) and time (su-

[17]One-dimensional case was chosen for simplicity, multi-dimensional MPDATA formulæ can be found in Smolarkiewicz and Margolin (1998, Sect. 2.2).

perscripts n), the donor-cell pass applied to Eq. (3) yields

$$\psi_i^{n+1} = \psi_i^n - \frac{1}{G_i}\Big[F\big(\psi_i^n, \psi_{i+1}^n, G_{i+1/2}, u_{i+1/2}^{n+1/2}\big) - F\big(\psi_{i-1}^n, \psi_i^n, G_{i-1/2}, u_{i-1/2}^{n+1/2}\big)\Big]. \quad (4)$$

The flux function F is defined as

$$F(\psi_L, \psi_R, G, u) \equiv \big([u]^+\psi_L + [u]^-\psi_R\big)\, G\frac{\Delta t}{\Delta x}, \quad (5)$$

where $[u]^+ \equiv \max(u,0)$ and $[u]^- \equiv \min(u,0)$.

In the case of a time-varying velocity field, the velocity components are evaluated at an intermediate time level denoted by the $n+1/2$ superscript in Eq. (4). Association of the velocity components with dual-cell edges is denoted by fractional indices $i+1/2$ and $i-1/2$; see Fig. 3.

Hereafter, $Gu\frac{\Delta t}{\Delta x}$ is written compactly as GC, where C denotes the Courant number. GC is referred to as the advector, while the scalar field ψ as the advectee – the nomenclature adopted after Randall (2013).

Evaluation of Eq. (4) concludes the first pass of MPDATA. To compensate for the implicit diffusion of the donor-cell pass, the subsequent passes of MPDATA reuse Eqs. (4) and (5), but with ψ replaced with the result of the preceding pass and $\boldsymbol{u}$ replaced with the "anti-diffusive" pseudo-velocity. The pseudo-velocity is analytically derived by expanding Eq. (4) in the second-order Taylor series about spatial point i and time level n, and representing the leading, dissipative truncation error as an advective flux; see Smolarkiewicz (1984) for a derivation. A single corrective pass ensures second-order accuracy in time and space. Subsequent corrective passes decrease the amplitude of the leading error, within second-order accuracy. The one-dimensional formula for the basic antidiffusive advector is written as

$$GC_{i+1/2}^{k+1} = \left[\left|GC_{i+1/2}^k\right| - \frac{\left(GC_{i+1/2}^k\right)^2}{0.5(G_{i+1}+G_i)}\right]\frac{\psi_{i+1}^k - \psi_i^k}{\psi_{i+1}^k + \psi_i^k}, \quad (6)$$

where k numbers MPDATA passes. For $k=1$, C^k is the flow-velocity-based Courant number, whereas for $k>1$, C^k is the pseudo-velocity-based Courant number. The number of corrective passes can be chosen within libmpdata++.

The library features two implementations of the donor-cell algorithm defined by Eqs. (4) and (5). The default one is a "straightforward" summation. The alternative, more resource-intensive, is the compensated summation algorithm of Kahan (1965) which reduces round-off error arising when summing numbers of different magnitudes.

3.1.2 Third-order-accurate variant

Accounting for third-order terms in the Taylor series expansion while deriving the pseudo-velocity improves the accuracy of MPDATA. When $G \equiv 1$, $u =$ constant and three or more corrective passes are applied, the procedure ensures third-order accuracy in time and space. The formulæ for the third-order scheme, derived analytically in Margolin and Smolarkiewicz (1998), can be found in Smolarkiewicz and Margolin (1998, Eq. 36).

3.1.3 Divergent-flow variant

In case of a divergent flow, the pseudo-velocity formulæ are augmented with an additional term proportional to the flow divergence. This additional term is implemented in libmpdata++ following Smolarkiewicz and Margolin (1998, Sect. 3.2(3)).

3.1.4 Non-oscillatory option

Solutions obtained with the basic MPDATA are sign-preserving, and thus non-oscillatory near zero. Generally, however, they feature dispersive ripples characteristic of higher-order numerical schemes. These can be suppressed by limiting the pseudo-velocities, in the spirit of flux-corrected transport. Application of the limiters reduces somewhat the accuracy of the scheme (Smolarkiewicz and Grabowski, 1990), yet this loss is generally outweighed by ensuring non-oscillatory (or ripple-free) solutions. Noteworthy, because MPDATA is built upon the donor-cell scheme characterised by small phase error, the non-oscillatory corrections have to deal with errors in signal amplitude only. The non-oscillatory option is a default option within the libmpdata++. For the derivation and further discussion of the multi-dimensional non-oscillatory option see Smolarkiewicz and Grabowski (1990).

3.1.5 Variable-sign scalar fields

The basic MPDATA formulation assumes that the advected field ψ is exclusively either non-negative or non-positive. In particular, this assumption is evident in the ψ-fraction factor $\frac{\psi_{i+1}^k - \psi_i^k}{\psi_{i+1}^k + \psi_i^k}$ of Eq. (6), which can become unbounded in case of a variable-sign field. The libmpdata++ library includes implementations of two MPDATA options intended for simulating advection of variable-sign field.

The first method replaces ψ with $|\psi|$ in all ψ-fraction factors that enter the pseudo-velocity expressions. This approach is robust but it reduces the solution quality where ψ crosses through zero; see Sect. 3.2(4) in Smolarkiewicz and Margolin (1998).

The default method, is the "infinite-gauge" variant of the algorithm, a generalised one-step Lax–Wendroff (linear, oscillatory) limit of MPDATA at infinite constant background, discussed in Smolarkiewicz (2006, Sect. 4.2). In practice, the infinite-gauge option of MPDATA is used with the non-oscillatory enhancement.

3.2 Library interface

3.2.1 Compile-time parameters

Compile-time parameters include number of dimensions, number of equations and algorithm options. Most of the compile-time parameters are declared by defining integer constants within the compile-time parameter structure. Listing 1 depicts a minimal definition that inherits from the **ct_params_default_t** structure containing default values for numerous parameters.

```
struct ct_params_t : ct_params_default_t
{
  using real_t = double;
  enum { n_dims = 1 };
  enum { n_eqns = 1 };
};
```

Listing 1. Example definition of compile-time parameters structure.

All solvers expect a structure with compile-time parameters as their first template parameter, as exemplified in Listing 2.

```
using slv_t = solvers::mpdata<ct_params_t>;
```

Listing 2. Example alias declaration combining solver- and compile-time parameters choice.

3.2.2 Choosing library components

The library components listed in Sect. 2.2 are chosen through template parameters. First, the solver is equipped with an output mechanism by passing the solver type as a template parameter to the output type, as exemplified in Listing 3. The output classes inherit from solvers.

```
using slv_out_t = output::gnuplot<slv_t>;
```

Listing 3. Example alias declaration of an output mechanism.

Second, the concurrency handlers expect solver class (equipped with output) as the first template parameter. Subsequent template parameters control boundary condition types on each of the domain edges (see Listing 4).

3.2.3 Run-time parameters

Run-time parameters include the grid size, number of MPDATA passes and output file name. The list of applicable run-time parameters is defined by fields of the **rt_params_t** structure. This structure is defined within each solver and extended when equipping the solver with an output mechanism.

```
using run_t = concurr::openmp<
  slv_out_t,
  bcond::cyclic, bcond::cyclic
>;
```

Listing 4. Example alias declaration of a concurrency handler.

The concurrency handlers expect an instance of the run-time parameters structure as their constructor argument. Example code depicting how to set the run-time parameters and then instantiate a concurrency handler is presented in Listing 5.

```
typename slv_out_t::rt_params_t p;
p.grid_size = { nx };
run_t run(p);
```

Listing 5. Example run-time parameter structure declaration followed by a concurrency handler instantiation.

3.2.4 Public methods

The concurrency handlers act as controlling logic for the other components and, hence, the user is exposed to the public interface of these handlers only.

Listing 6 contains signatures of methods implemented by each of the concurrency handlers.

```
blitz::Array<real_t, n_dims> advectee(int eqn = 0)
```

```
blitz::Array<real_t, n_dims> advector(int dim = 0)
```

```
blitz::Array<real_t, n_dims> g_factor()
```

```
void advance(int)
```

```
bool *panic_ptr()
```

Listing 6. Signatures of all the methods within libmpdata++ application programming interface.

The **advectee()** is an accessor method for the advected scalar fields. It can be used for setting the initial condition as well as for examining the solver state. It expects an index of the requested advectee as the argument (advected scalar fields are numbered from zero). This provides choice between different advected variables. The returned **blitz::Array** is zero-base indexed and has the same size as the computational grid (set with the **grid_size** field of the run-time parameters structure, see Listing 5).

The **advector()** method allows accessing the components of the vector field of Courant numbers multiplied by the G

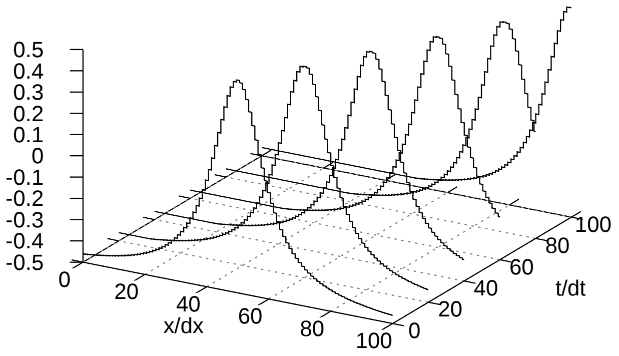

Figure 4. Simulation results generated by the code in Listing 7.

factor (i.e. a Jacobian of coordinate transformation, a fluid density field or their product). The argument selects the vector field components numbered from zero. The size of the returned array depends on the component. It equals the grid size in all but the selected dimension in which it is reduced by 1 (i.e. $nx \times (ny-1)$ for the "y" component and so forth; cf. Fig. 3).

The **g_factor()** is an accessor method for the G field. The returned array has the same size as the one returned by **advectee()**. The default value is set to $G \equiv 1$ (for details, see Sect. 3.8).

The **advance()** method launches the time-stepping logic of the solver advancing the solution by the number of time steps given as argument.

The **panic_ptr()** method returns a pointer to a Boolean variable that if set to true will cause the solver to stop the computations after the currently computed time step. This method may be used, for instance, to implement signal handling within programs using libmpdata++.

All multi-dimensional arrays used in libmpdata++ use the default Blitz++ "row-major" memory layout with the last dimension varying fastest. Domain decomposition for parallel computations is done over the first dimension only.

3.3 Basic example

The source code presented in this subsection is intended to serve as a minimal complete example on how to use libmpdata++. In other examples presented throughout the paper, only the fragments of code that differ significantly from the minimal example will be presented.

The example consists of an elemental transport problem for a one-dimensional, variable-sign field advected with a constant velocity. The simulation results using code in Listing 7 are shown in Fig. 4. Spatial and temporal directions are depicted on the abscissa and ordinate, respectively. Cell-mean values of the transported field are shown on the applicate and are presented in compliance with the assumption of data points representing grid-cell means of the transported field.

```
#include <libmpdata++/solvers/mpdata.hpp>
#include <libmpdata++/concurr/serial.hpp>
#include <libmpdata++/output/gnuplot.hpp>

using namespace libmpdataxx;

int main()
{
  // compile-time parameters
  struct ct_params_t : ct_params_default_t
  {
    using real_t = double;
    enum { n_dims = 1 };
    enum { n_eqns = 1 };
  };

  // solver choice
  using slv_t = solvers::mpdata<ct_params_t>;

  // output choice
  using slv_out_t = output::gnuplot<slv_t>;

  // concurrency choice
  using run_t = concurr::serial<
    slv_out_t, bcond::open, bcond::open
  >;         //left bcond   //right bcond

  // run-time parameters
  typename slv_out_t::rt_params_t p;

  int nx = 101, nt = 100;
  ct_params_t::real_t dx = 0.1;

  p.grid_size = { nx };
  p.outfreq = 20;

  // instantiation
  run_t run(p);

  // initial condition
  blitz::firstIndex i;
  // Witch of Agnesi with a=.5
  run.advectee() = -.5 + 1 / (
    pow(dx*(i - (nx-1)/2.), 2) + 1
  );
  // Courant number
  run.advector() = .5;

  // integration
  run.advance(nt);
}
```

Listing 7. A usage example of libmpdata++. The listing contains the code needed to generate Fig. 4.

The code in Listing 7 begins with three include statements that reflect the choice of the library components: solver, concurrency handler and output mechanism. All compile-time parameters are grouped into a structure passed as a template parameter to the solver. Here, this structure is named **ct_params_t** and inherits from **ct_params_default_t** what results in assigning default values to parameters not defined within the inheriting class. The solvers expect the structure to contain a type **real_t** which controls the floating point format used. The two constants that do not have default values and need to be explicitly defined are **n_dims** and **n_eqns**. They control the dimensionality of the problem and the number of equations to be solved, respectively.

Choice between different solver types, output mechanisms and concurrency handlers is done via type alias declaration. Here, the basic **mpdata** solver is chosen which is then equipped with the **gnuplot** output mechanism. All output classes expect a solver class as their first template parameter, which is used to define the parent class (i.e. output classes inherit from solvers).

Classes representing concurrency handlers expect the output class and the boundary conditions as their template parameters. In the example, a basic serial handler is used and open boundary conditions on both ends of the domain are chosen.

The choice of run-time parameters is done by assigning values to the member fields of the **rt_params_t** structure defined within the solver class and augmented with additional fields by the output class. In this example, the instance of **rt_params_t** structure is named **p**, the grid size is set to 101 points and the output is set to be done every 20 time steps. An instance of the **rt_params_t** structure is expected as the constructor parameter for concurrency handlers.

The grid step **dx** is set to 0.1 and the number of time steps to 100. Initial values of the Courant number and the transported scalar fields are set by assigning them to the arrays returned by the **advector()** and **advectee()** methods. In this example, the Courant number equals 0.5 and the advected shape is described by the Witch of Agnesi formula $y(x) = 8a^3/(x^2+4a^2)$ with the coefficient $a = 0.5$. Initial shape is centred in the middle of the computational domain and is shifted downwards by 0.5. Finally, the actual integration is performed by calling the **advance()** method with the number of time steps as argument.

3.4 Example: advection scheme options

The following example is intended to present MPDATA advection scheme options described in Sect. 3.1. The way of choosing different options is discussed, and the calling sequence of the library interface is shown for the case of advecting multiple scalar fields.

The example consists of transporting two boxcar signals with different MPDATA options. In all tests, the first signal extends from 2 to 4 and the second signal extends from -1 to 1, to observe the solution for fixed-sign and variable-sign signals. Listing 8 shows the compile-time parameters structure fields common to all cases presented within this example. The number of dimensions is set to 1 and the number of equations to solve is set to 2. Consistent with Listing 7 from the basic example, **p** shown in Listing 9 is an instance of **rt_params_t** structure with run-time parameters of the simulation. Setting the **outfreq** field to the number of time steps results in plotting the initial condition and the final state. The **outvars** field contains a map with a structure containing a variable name, here left empty, and unit defined for each of the advected scalar fields. Listing 10 shows how to set initial values to multiple scalar fields using the **advectee()** method with an integer argument specifying the index of the equation in the solved system.

```
enum { n_dims = 1 };
enum { n_eqns = 2 };
```

Listing 8. Compile-time parameters for the example presented in Sect. 3.4.

```
int nx = 601, nt = 1200;
// run-time parameters
p.grid_size = { nx };
p.outfreq = nt;
p.outvars = {
  {0, {.name = "", .unit = "1"}},
  {1, {.name = "", .unit = "1"}}
};
```

Listing 9. Run-time parameters for the example presented in Sect. 3.4.

```
// initial condition
blitz::firstIndex i;
run.advectee(0) = where(
  i <= 75 || i >= 125,   // if
  2,                     // then
  4                      // else
);
run.advectee(1) = where(
  i <= 75 || i >= 125,   // if
  -1,                    // then
  1                      // else
);
run.advector() = -.75;  // Courant
```

Listing 10. Initial condition and velocity field for the example presented in Sect. 3.4.

3.4.1 Variable-sign scalar fields

The libmpdata++ library is equipped with two options for handling variable-sign fields; recall the discussion in Sect. 3.1.5. The option using absolute values is named **abs**, whereas the “infinite-gauge” option is dubbed **iga**. The option flags are defined in the **opts** namespace. The option choice is made by defining the **opts** field of the compile-time parameters structure, in analogy to **n_dims** or **n_eqns**.

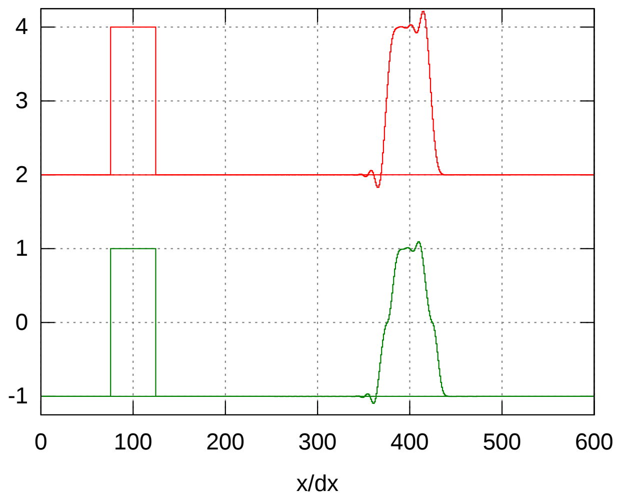

Figure 5. Result of the simulation with the advection scheme option for variable-sign signal set to absolute value; cf. Listing 11.

```
enum { opts = opts::abs };
```

Listing 11. Advection scheme options for Fig. 5, variable-sign option is set to absolute value.

In the first test, the choice of handling variable-sign signal is set to **abs** (Listing 11). Figure 5 shows the result of simulation with parameters set in Listings 8, 9, 10 and 11. The final signal shows dispersive ripples characteristic of higher-order schemes. It is also evident that the ripple magnitude depends on the constant background, a manifestation of the scheme non-linearity. Furthermore, the final variable-sign signal features a bogus saddle point at the zero crossings (cf. Sect. 3.1.5), and this can be eliminated by using the infinite-gauge (alias **iga**) option. Listing 12 shows how to choose the **iga** option. Figure 6 shows the result of the simulation with parameters set in Listings 8, 9, 10 and 12. Although **iga** evinces more pronounced oscillations, their magnitudes do not depend on the constant background. This, together with the robust behaviour of **iga** when crossing zero, substantiates the discussion of Sect. 3.1.5 on **iga** amounting to a linear limit of MPDATA.

3.4.2 Third-order-accurate variant

Choosing third-order variant enhances the accuracy of the scheme when used with more than two passes of MPDATA or with **iga**; recall Sect. 3.1.2. Option **tot** enables the third-order variant of the MPDATA scheme. Figure 7 shows the result of the same test as in Figs. 5 and 6 but with MPDATA options set as in Listing 13. The resulting signal is evidently more accurate and symmetric, but the oscillations are still present.

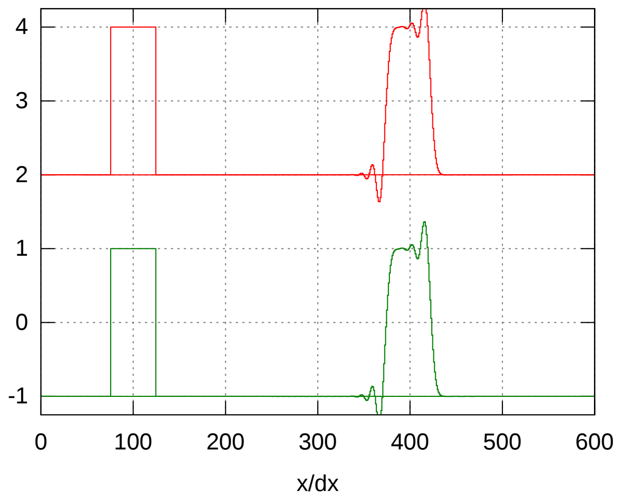

Figure 6. As in Fig. 5 but with variable-sign option set to “infinite-gauge”; cf. Listing 12.

```
enum { opts = opts::iga };
```

Listing 12. Advection scheme options for Fig. 6, variable-sign option is set to “infinite-gauge”.

3.4.3 Non-oscillatory option

To eliminate oscillations apparent in the preceding tests, the non-oscillatory (**fct**) option (Sect. 3.1.4) needs to be chosen. This option can be used together with all other MPDATA options, such as basic scheme, variable-sign signals (**abs** or **iga**) and the third-order-accurate variant (**tot**).

Here, **fct** is selected together with **iga**; cf. Listing 14. This is the default setting, i.e. when inheriting from the default parameters structure, and not overriding the **opts** setting, as illustrated in Listing 7. Figure 8 shows the corresponding results. The solutions for both fixed-sign and variable-sign signals have indistinguishable profiles and all of the dispersive ripples have been suppressed.

To further enhance the accuracy of the solution, **fct** and **iga** can be combined with the **tot** variant; cf. Listing 15. The corresponding result is shown in Fig. 9. Enabling the third-order-accurate variant improves the symmetry of the solution, as compared to the results presented in Fig. 8.

3.5 Example: convergence tests in 1-D

In this subsection the convergence test originated in Smolarkiewicz and Grabowski (1990) is used to quantify the accuracy of various MPDATA options.

The test consists of a series of one-dimensional simulations with Courant numbers

$$C \in (0.05, 0.1, 0.15, 0.2, \ldots, 0.85, 0.9, 0.95)$$

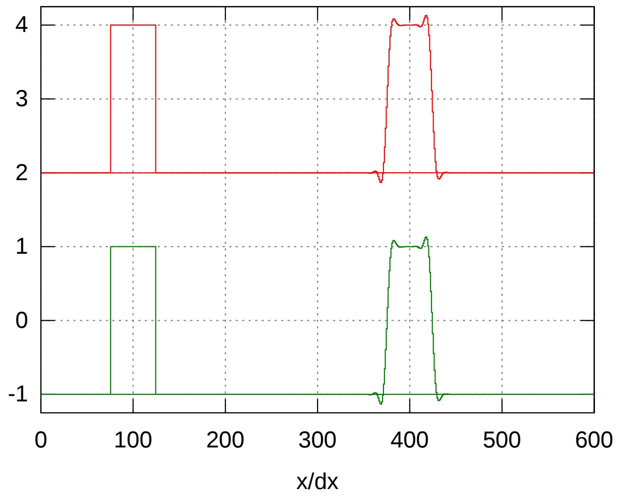

Figure 7. As in Fig. 5 but with variable-sign option set to "infinite-gauge" and third-order-accurate variant; cf. Listing 13.

```
enum { opts = opts::iga | opts::tot };
```

Listing 13. Advection scheme options for Fig. 7, variable-sign option is set to "infinite-gauge" and third-order accuracy variant is chosen.

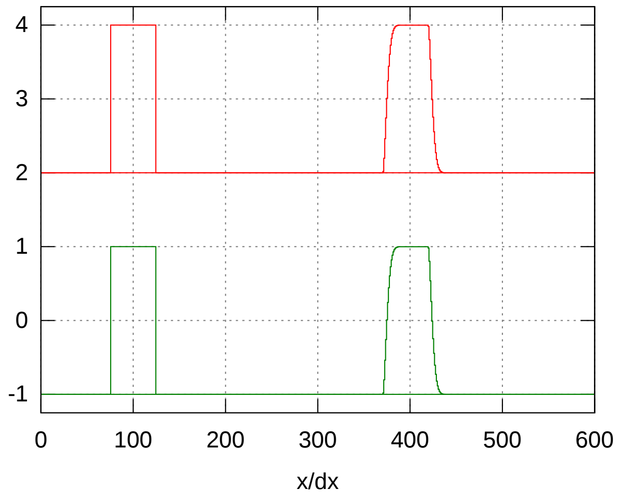

Figure 8. As in Fig. 5 but with options set to infinite-gauge treatment of variable-sign signal and flux corrections; cf. Listing 14.

```
enum { opts = opts::iga | opts::fct };
```

Listing 14. Advection scheme options for Fig. 8, variable-sign option is set to "infinite-gauge" and non-oscillatory option is enabled. This is the default setting in libmpdata++.

and grid increments

$$\Delta x \in \left(\frac{\Delta x_m}{2^0}, \frac{\Delta x_m}{2^1}, \frac{\Delta x_m}{2^2}, \frac{\Delta x_m}{2^3}, \frac{\Delta x_m}{2^4}, \frac{\Delta x_m}{2^5}, \frac{\Delta x_m}{2^6}, \frac{\Delta x_m}{2^7}\right),$$

where $\Delta x_m = 1$ is the maximal increment. The series amounts to 152 simulations for each option. In each simulation, the number of time steps NT and the number of grid cells NX is adjusted so that the total time T and total length of the domain X remain constant. The domain size $X = 44\Delta x_m$ and simulation time $T = 1$ are selected. The advective velocity is set to $u = \Delta x_m / T = 1$.

In each simulation, a Gaussian profile

$$\psi_{\text{ex}}(x)_{t=0} = \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-\frac{(x-x_0)^2}{2\sigma^2}\right) \quad (7)$$

is advected, and the result of the simulation is compared with the exact solution ψ_{ex}. The initial profiles and the exact solutions are calculated by analytically integrating function (7) over the grid-cell extents, to comply with the inherent MPDATA assumption of a data point representing the grid-cell mean of transported field. The dispersion parameter of the initial profile (7) is set to $\sigma = 1.5\Delta x_m$, while the profile is centred in the middle of the domain $x_0 = 0.5X$.

As a measure of accuracy, a truncation-error function is introduced

$$\text{err}(C, \Delta x) \equiv \frac{1}{T}\sqrt{\sum_{i=1}^{NX}[\psi_{\text{ex}}(x_i) - \psi(x_i)]^2 / NX}\,\Bigg|_{t=T}. \quad (8)$$

The results of the convergence test for the generic first-order-accurate donor-cell scheme, the basic MPDATA and its third-order-accurate variant are shown in Fig. 10a–c. Each figure displays, in polar coordinates, the base-two logarithm of the truncation-error function (8) for the entire series of 152 simulations. The radius and angle, respectively,

$$r = \ln_2\left(\frac{\Delta x}{\Delta x_m}\right) + 8, \quad \phi = C\frac{\pi}{2}, \quad (9)$$

indicate changes in grid increment and Courant number. Thus, closer to the origin are simulation results for finer grids, closer to the abscissa are points for small Courant numbers, and closer to the ordinate are points with Courant numbers approaching unity. The contour interval of dashed isolines and of the colour map is set to 1, corresponding to error reduction by the factor of 2. Lines of constant grid-cell size and constant Courant number are overlaid with white contours.

The figures contain information on the convergence rate of MPDATA options. When moving along the lines of constant Courant number towards the origin, thus increasing the spatial and temporal resolution, the number of crossed dashed isolines determines the order of the scheme; cf. Sect. 8.1 in Margolin and Smolarkiewicz (1998). Therefore, the results in Fig. 10a–c attest to the first-, second- and third-order asymptotic convergence rates, respectively. Furthermore, the shape of dashed isolines conveys the dependency of the solution accuracy on the Courant number. In particular, they show that at

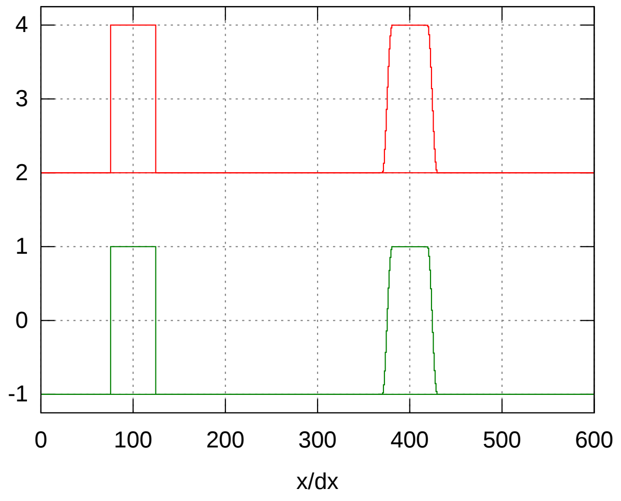

Figure 9. As in Fig. 5 but with options set to infinite-gauge treatment of variable-sign signal, non-oscillatory option and third-order accuracy variant; cf. Listing 15.

```
enum { opts = opts::iga | opts::tot | opts::fct };
```

Listing 15. Advection scheme options for Fig. 9, variable-sign option is set to "infinite-gauge", non-oscillatory option is enabled and third-order accuracy variant is chosen.

fixed spatial resolution the solution accuracy increases with the Courant number. Moreover, as the order of the convergence increases the isolines become more circular indicating more isotropic solution accuracy in the Courant number.

Figure 10b reproduces the solution in Fig. 1 of Smolarkiewicz and Grabowski (1990) and, thus, verifies the libmpdata++ implementation. For further verification Fig. 11a and b shows results of the convergence test for (i) three-pass MPDATA (run-time solver parameter **n_iters** = **3**), and (ii) for two-pass MPDATA with **fct** option. These results reproduce Figs. 2 and 3 from Smolarkiewicz and Grabowski (1990). Noteworthy, an interesting feature of Fig. 11a is the groove of the third-order convergence rate formed around $\phi = 45°$, characteristic of MPDATA with three or more passes (Margolin and Smolarkiewicz, 1998). Next, comparing Fig. 11b with Fig.10b shows that the price to be paid for an oscillation-free result is a reduction in the convergence rate (from 2 to ~ 1.8; Sect. 4 in Smolarkiewicz and Grabowski, 1990).

Figure 11c and d documents original results for the convergence test applied to the "infinite-gauge" limit of MPDATA. In particular, Fig. 11c shows that **iga** is as accurate as three-pass MPDATA, (cf. Sect. 4 in Smolarkiewicz and Clark, 1986), whereas Fig. 11d reveals that the third-order-accurate **iga** is more anisotropic in Courant number than the third-order-accurate standard MPDATA in Fig. 10c.

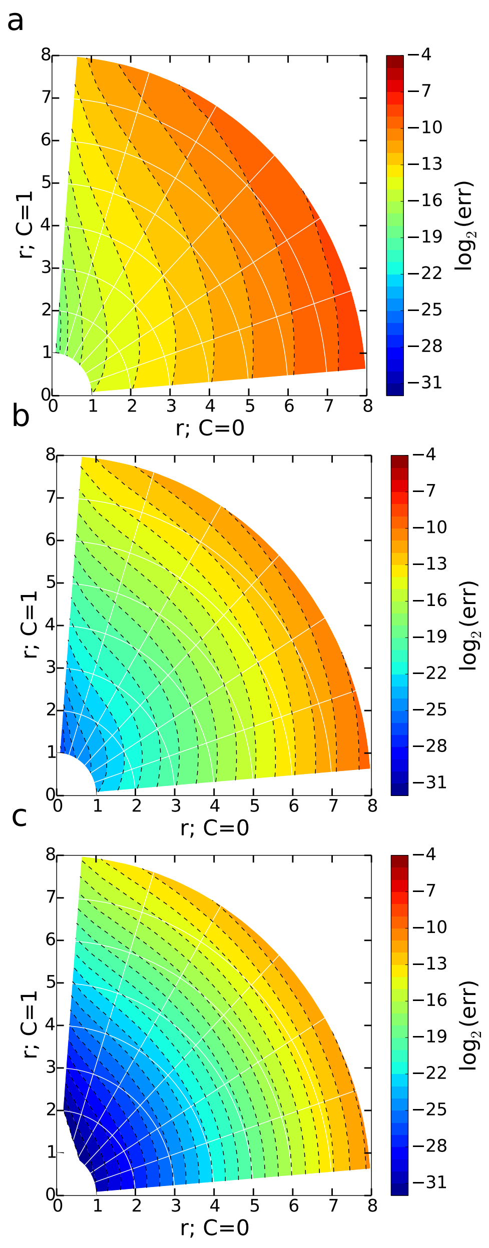

Figure 10. The result of the convergence test **(a)** for the donor-cell scheme, **(b)** for the basic MPDATA and **(c)** for the third-order-accurate variant.

The convergence test results for the default setting of libmpdata++ (**iga** plus **fct**) are not shown, because they resemble results from Fig. 11b with somewhat enhanced accuracy for well-resolved fields (i.e. small grid cells).

3.6 Example: rotating cone in 2-D

This example introduces the libmpdata++ programming interface for two-dimensional simulations with the velocity field varying in space. Test results are compared with published MPDATA benchmarks. The example is based on the

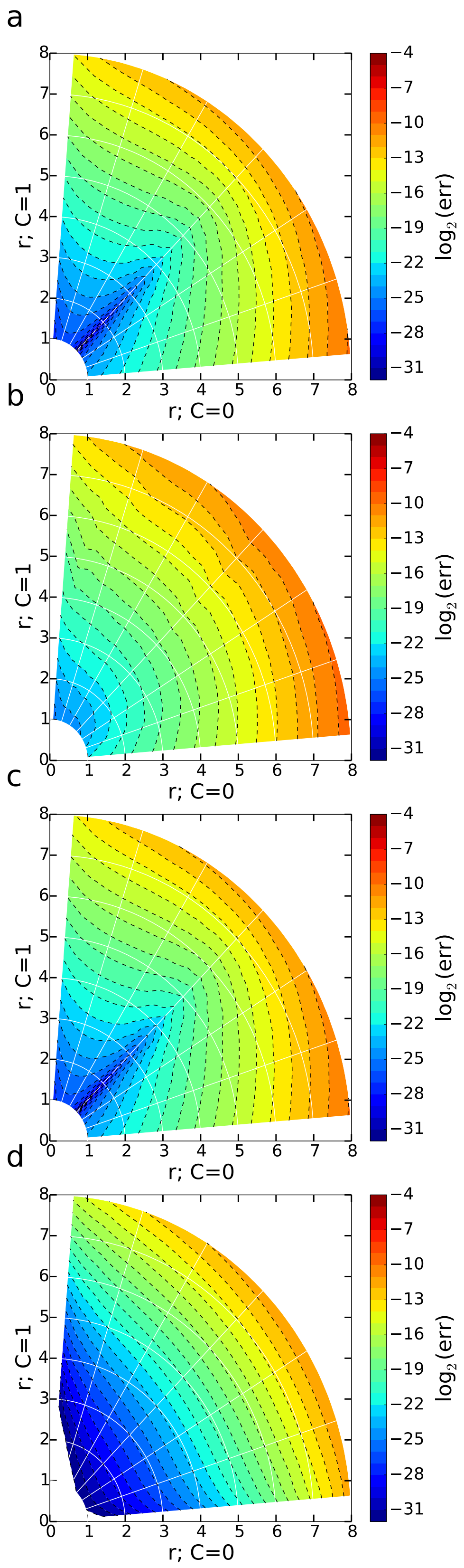

Figure 11. As in Fig. 10 **(a)** for three passes of MPDATA, **(b)** for two passes with non-oscillatory option, **(c)** for infinite-gauge option, and **(d)** for infinite-gauge with third-order-accurate variant.

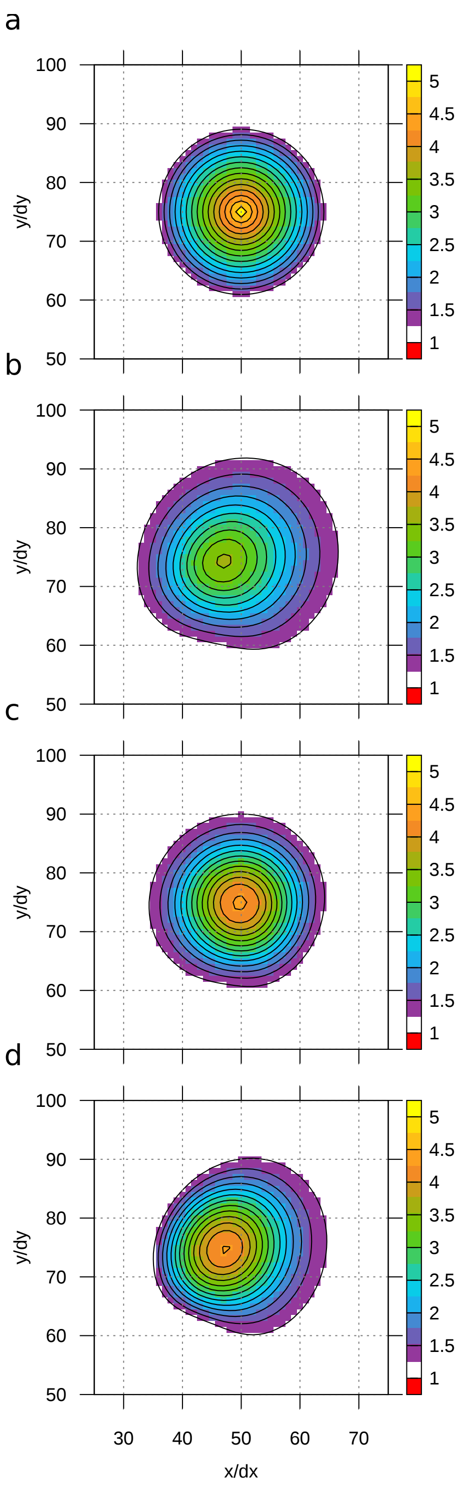

Figure 12. The results of the example presented in Sect. 3.6; only a quarter of the domain, centred over the cone's initial location, is shown. Abscissa and ordinate mark the spatial dimensions. Colours correspond to the amplitude of the advected field. Panel **(a)** shows initial condition of Sect. 3.6, **(b)** results for basic MPDATA with **fct**, **(c)** for MPDATA with three passes with **fct** and **tot** and **(d)** for the default setting of libmpdata++ (**iga** and **fct**).

classical solid-body rotation test (Molenkamp, 1968). The current setup follows Smolarkiewicz and Margolin (1998). The initial condition features a cone centred around the point $(x_0, y_0) = (50\Delta x, 75\Delta y)$. The grid interval is $\Delta x = \Delta y = 1$, and the domain size is $100\Delta x \times 100\Delta y$ – thus containing 101×101 data points (cf. Fig. 2). The height of the cone is set to 4, the radius to $15\Delta x$, and the background level to 1. The flow velocity is specified as $(u, v) = \omega\,(y - y_c, -(x - x_c))$, where angular velocity $\omega = 10^{-1}$ and (x_c, y_c) denotes coordinates of the domain centre. With time interval $\Delta t = 0.1$, one full rotation requires 628 time steps. The total integration time corresponds to six full rotations.

```
enum { n_dims = 2 };
enum { n_eqns = 1 };
```

Listing 16. Compile-time parameter settings for the rotating-cone test.

```
p.n_iters = 3;
```

Listing 17. Run-time parameter responsible for setting the number of MPDATA passes in Fig. 12c.

Implementation of the setup using the libmpdata++ interface begins with definition of the compile-time parameters structure. The test features a single scalar field in a two-dimensional space, what is reflected in the values of **n_dims** and **n_eqns** set in Listing 16. In one of the test runs, the number of MPDATA passes (**n_iters**) is set to 3, instead of the default value of 2. Corresponding field of run-time parameters structure is shown in Listing 17. During instantiation of the concurrency handler, four boundary-condition settings (two per each dimension) are passed as template arguments. In this example, open boundary conditions (**bcond::open**) are set in both dimensions – see Listing 18.

The choice of the **threads** concurrency handler in Listing 18 results in multi-threaded calculations – using OpenMP if the compiler supports it, or using Boost.Thread otherwise. The number of computational subdomains (and hence threads) is controlled by the **OMP_NUM_THREADS** environment variable, regardless if OpenMP or Boost.Thread implementation is used. The default is to use all CPUs/cores available in the system. Notably, replacing **concurr::serial** from the previous examples with **concurr::threads** is the only modification needed to enable domain decomposition via shared-memory parallelism.

The way the initial condition and the velocity field are set is shown in Listing 19. The Courant number components are specified using calls to the **advector()** method with the argument defining the component index.

The initial condition is displayed in Fig. 12a, and the results after total integration time are shown in Fig. 12b–d.

```
// instantiation
concurr::threads<
  slv_out_t,
  bcond::open, bcond::open,
  bcond::open, bcond::open
> run(p);
```

Listing 18. Concurrency handler instantiation for the rotating-cone test.

```
// temporary array of the same ...
decltype(run.advectee())         // type
  tmp(run.advectee().extent()); // and size
// ... as the one returned by advectee()

// helper vars for Blitz++ tensor notation
blitz::firstIndex i;
blitz::secondIndex j;

// cone shape ...
tmp = blitz::pow(i * dx - x0, 2) +
      blitz::pow(j * dy - y0, 2);

// ... cut off at zero
run.advectee() = h0 + where(
  tmp - pow(r, 2) <= 0,                // if
  h - blitz::sqrt(tmp / pow(r/h,2)),   // then
  0.                                   // else
);

// constant-angular-velocity rotational field
run.advector(x) =  omega * (j * dy - yc) * dt/dx;
run.advector(y) = -omega * (i * dx - xc) * dt/dy;
```

Listing 19. Initial condition for the rotating-cone test.

All plots are centred around cone's initial location and show only a quarter of the computational domain. The isolines of the advected cone are plotted with 0.25 intervals. The results in Fig. 12b and c were obtained with the **fct** and the three-pass **tot** + **fct** MPDATA, respectively, whereas Fig. 12d shows test results for the default setting of libmpdata++. These results match those presented in Smolarkiewicz and Margolin (1998, Fig. 1) and Smolarkiewicz and Szmelter (2005, Fig. 4 and Table 1). In particular, the rms errors – defined on the rhs of Eq. (8) – are 0.37×10^{-3}, 0.11×10^{-3} and 0.27×10^{-3} for the **fct**, three-pass **tot fct** and the default libmpdata++ options, respectively.

3.7 Example: revolving sphere in 3-D

This example extends Sect. 3.6 to three spatial dimensions. It exemplifies how to specify a three-dimensional setup using libmpdata++. Furthermore, the option is described for saving the simulation results to HDF5 files with XDMF annotations.

The setup follows Smolarkiewicz and Szmelter (2005): the domain size is $100\times100\times100$, with a uniform grid consisting of 59 grid points in each direction. The time step is 0.036π. The initial condition is a sphere of radius 15 centred around the point $(x_0, y_0, z_0) = (50 - 25/\sqrt{3}, 50 + 25/\sqrt{3}, 50 + 25/\sqrt{3})$ with a constant density of 4. The sphere is rotating with constant angular velocity $\boldsymbol{\Omega} = \omega/\sqrt{3}(1, 1, 1)$ of magnitude $\omega = 0.1$. The components of the advecting velocity field are $(u, v, w) = (-\Omega_z(y-y_c)+\Omega_y(z-z_c), \Omega_z(x-x_c)-\Omega_x(z-z_c), -\Omega_y(x-x_c)+\Omega_x(y-y_c))$, where the coordinates of the rotation centre are $(x_c, y_c, z_c) = (50, 50, 50)$. The test lasts for one revolution which takes 556 time steps.

Specifying the 3-D setup with the libmpdata++ programming interface calls starts by setting the **n_dims** field to 3 (Listing 20). Listing 21 shows the choice of recommended three-dimensional output handler **hdf5_xdmf**. This results in output consisting of HDF5 files with XDMF annotation that can be viewed, for example, with the ParaView visualisation software. This output is saved in a directory specified by the **outdir** field of the run-time parameters; see Listing 22.

```
enum { n_dims = 3 };
```

Listing 20. Compile time parameter setting for the revolving-sphere test.

```
using slv_out_t = output::hdf5_xdmf<slv_t>;
```

Listing 21. Alias declaration of an output mechanism for the revolving-sphere test.

```
p.outdir = dir_name;
```

Listing 22. Run-time parameters field specifying output directory for the revolving-sphere test.

Figure 13a shows the initial condition, Fig. 13b shows the results after one revolution for the default libmpdata++ options. The grey volume is composed of dual-grid cells (Sect. 2.3) encompassing data points with cell-mean values of density greater than or equal to 1.

Obtained results can be compared with those presented in Smolarkiewicz and Szmelter (2005, Figs. 9–13 and Table 4). In particular, for the default libmpdata++ setting, the rms error is 2.8×10^{-3}, and it compares favourably with the L_2 norm in their Table 4.

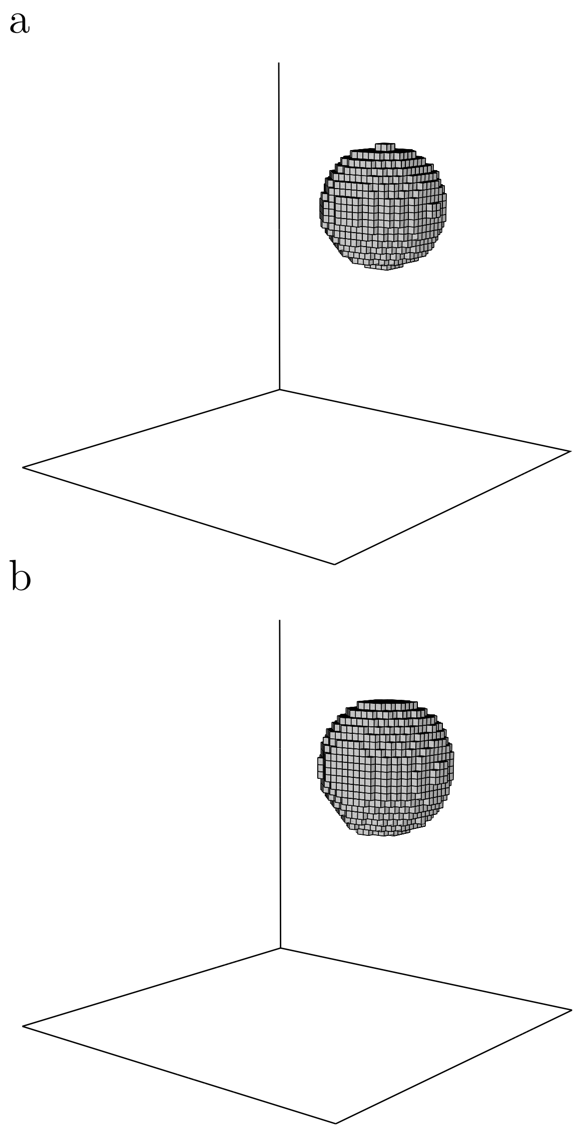

Figure 13. The results of the example presented in Sect. 3.7. The whole computational domain is shown. The grey volume encompasses data points with values of density greater or equal to 1. Panel **(a)** shows initial condition, **(b)** results for the default libmpdata++ options.

3.8 Example: 2-D advection on a sphere

This subsection concludes homogeneous transport examples with a 2-D solid-body rotation test on a spherical surface (Williamson and Rasch, 1989). The purpose of this example is to present methods for setting up the simulations in spherical coordinates.[18]

Following Smolarkiewicz and Rasch (1991) only the case when the initial field rotates over the poles is presented. The initial condition is a cone centred around the point $(3\pi/2, 0)$ with height and radius equal to 1 and $7\pi/64$, respectively.

[18]The same method, used here to specify a Jacobian of coordinate transformation, can be applied to prescribe a variable-in-space fluid density.

The wind field is given by

$$u = -U \sin\phi \cos\lambda, \quad v = U \sin\lambda, \tag{10}$$

where λ and ϕ denote respectively longitude and latitude, and $U = \pi/128$. The computational domain $[0, 2\pi] \times [-\pi/2, \pi/2]$ is resolved with 128×64 grid increments $\Delta\lambda = \Delta\phi$ and is shifted by $0.5\Delta\phi$ so that there are no data points on the poles. The test is run for 5120 time steps corresponding to one revolution around the globe.

The advection equation in spherical coordinates has the form of the generalised transport Eq. (1) with the Jacobian of coordinate transformation

$$G = \cos\phi. \tag{11}$$

In order to solve the generalised transport equation with $G \not\equiv 1$ the **nug** option has to be set; see Listing 23.

```
enum { opts = opts::nug };
```

Listing 23. Compile-time parameter field for the example presented in Sect. 3.8.

Boundary conditions in this example incorporate principles of differential geometry (cf. chapter XIV in Maurin, 1980) in the classical spherical latitude–longitude framework (Szmelter and Smolarkiewicz, 2010). They are cyclic (**bcond::cyclic**) in the zonal direction, whereas in the meridional direction they represent two degenerated charts (of the atlas composed of three) defining differentiation of dependent variables in vicinity of the poles (**bcond::polar**; Listing 24). The setting of G is done using the **g_factor()** accessor method as shown in Listing 25; note the shift in latitude by $\Delta\phi/2$.

```
concurr::threads<
  slv_out_t,
  bcond::cyclic, bcond::cyclic,
  bcond::polar, bcond::polar
> run(p);
```

Listing 24. Concurrency handler for the example presented in Sect. 3.8.

```
run.g_factor() = dlmb * dphi *
  blitz::cos(dphi * (j + 0.5) - pi / 2);
```

Listing 25. The Jacobian setting for the example presented in Sect. 3.8.

The initial condition for the test is plotted in Fig. 14a, whereas the results are displayed in Fig. 14b and c. All figures use orthographic projection, with the perspective centred at the initial condition (the true solution), with the contour interval 0.1. Figure 14b shows the result for the default libmpdata++ options. There is a visible deformation in the direction of motion, consistent with earlier Cartesian rotational tests. The result in Fig. 14c, obtained using three passes of MPDATA with **fct** and **tot**, shows reduced deformation and reproduces Fig. 6 in Smolarkiewicz and Rasch (1991). Error norms were calculated following Smolarkiewicz and Rasch (1991, Eqs. 24a–e) to take into account the effects of

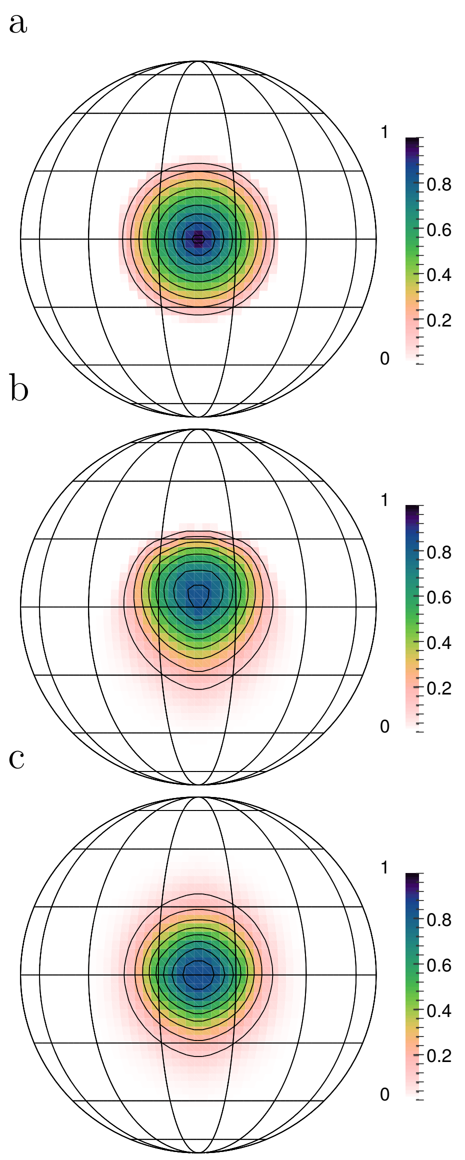

Figure 14. The results of the example presented in Sect. 3.8. The plots are centred over the cone's initial location and show the advected field plotted in spherical coordinates. Colours mark the amplitude of the advected field. Panel **(a)** shows the initial condition, **(b)** results for the default libmpdata++ options and **(c)** results for the three-pass MPDATA with **fct** and **tot**.

coordinate transformation. For instance, the "energy" conservation error (their ERR2) is -0.066 for the default libmpdata++ setting and -0.11 for the three-pass MPDATA with **tot** and **fct**, which agrees with the values presented in Smolarkiewicz and Rasch (1991, Table 1).

4 Inhomogeneous advective transport

4.1 Implemented algorithms

As of the current release, libmpdata++ provides three ways of handling source terms in the inhomogeneous extension of Eq. (3):

$$\partial_t \psi + \frac{1}{G} \nabla \cdot (G\boldsymbol{u}\psi) = R. \tag{12}$$

The available time integration schemes include the two variants of the first-order-accurate Euler-forward scheme (hereafter referred to as **euler_a** and **euler_b**) and the second-order-accurate Crank–Nicolson scheme (**trapez**). The Euler schemes are implemented to account for parameterised forcings (e.g. due to cloud microphysics), whereas the Crank–Nicolson scheme is standard for basic dynamics (e.g. pressure gradient, Coriolis and buoyancy forces). In both Euler schemes, while calculating the solver state at the time level $n+1$, the right-hand-side at the time level n is only needed. In the **euler_a** option (Eq. 13), the source terms are computed and applied standardly after the advection:

$$\psi^{n+1} = \mathrm{ADV}(\psi^n) + \Delta t R^n. \tag{13}$$

In the **euler_b** option (Eq. 14), the source terms are computed and applied (arguably in the Lagrangian spirit; Sect. 3.2 in Smolarkiewicz and Szmelter, 2009) before the advection

$$\psi^{n+1} = \mathrm{ADV}(\psi^n + \Delta t R^n). \tag{14}$$

In the **trapez** option (Eq. 15), half of the source terms are computed and applied as in the **euler_a** and half as in the **euler_b** (arguably in the spirit of the Lagrangian trapezoidal rule; Sect. 2.2 in Smolarkiewicz and Szmelter, 2009):

$$\psi^{n+1} = \mathrm{ADV}(\psi^n + 0.5\Delta t R^n) + 0.5\Delta t R^{n+1}. \tag{15}$$

4.2 Library interface

The logic for handling source terms is implemented in the **mpdata_rhs** solver that inherits from the **mpdata** class (Fig. 1). Consequently, all options discussed in the preceding section apply. The choice of the source-term integration scheme is controlled by the **rhs_scheme** compile-time parameter with the valid values of **euler_a**, **euler_b** or **trapez**.

The user is expected to provide information on the source terms by defining a derived class of **mpdata_rhs** with the **update_rhs()** method overloaded. The **update_rhs()** signature is given in Listing 26, whereas the usage example is given in Sect. 4.3. The method is called by the solver with the following arguments:

– a vector of arrays **rhs** storing the source terms for each equation of the integrated system,

– a floating-point value **dt** with the time-step value,

– an integer number **at** indicating if the source terms are to be computed at time level n (if at $=0$) or $n+1$ (if at $=1$).

```
virtual void update_rhs(
  arrvec_t<typename parent_t::arr_t> &rhs,
  const typename parent_t::real_t &dt,
  const int &at
)
```

Listing 26. Signature of the method used for defining source terms.

Calculation of forcings at the $n+1$ time level is needed if the **rhs_scheme=trapez** option is chosen. The case of **at** equal to zero is used in the Euler schemes and in the very first time step when using the **trapez** option (i.e. once per simulation). When the **trapez** option is used, the **dt** passed to the **update_rhs()** method equals half of the original time step.

The **update_rhs()** method is expected to first call **parent_t::update_rhs()** to zero out the source and sink terms stored in **rhs**. Later, it is expected to calculate the **rhs** terms in a given time step by summing all sources and sinks and "augment assign" them to the **rhs** field (e.g. using the $+=$ operator).

All elements of the **rhs** vector corresponding to subsequent equations in the system are expected to be modified in a single **update_rhs()** call.

4.3 Example: translating oscillator

The purpose of this example is to show how to include rhs terms in libmpdata++ by creating a user-defined class out of the library tree.

A system of two one-dimensional advection equations,

$$\begin{aligned} \partial_t \psi + \partial_x(u_o \psi) &= \omega\phi, \\ \partial_t \phi + \partial_x(u_o \phi) &= -\omega\psi, \end{aligned} \tag{16}$$

represents a harmonic oscillator translating with $u_o =$ constant; see Sect. 4.1 in Smolarkiewicz (2006) for a discussion.[19] Applying the trapezoidal rule to integrate the PDE

[19]The implicit manner of prescribing forcings, similar to the one presented herein, is an archetype for integrating Coriolis force in Prusa et al. (2008).

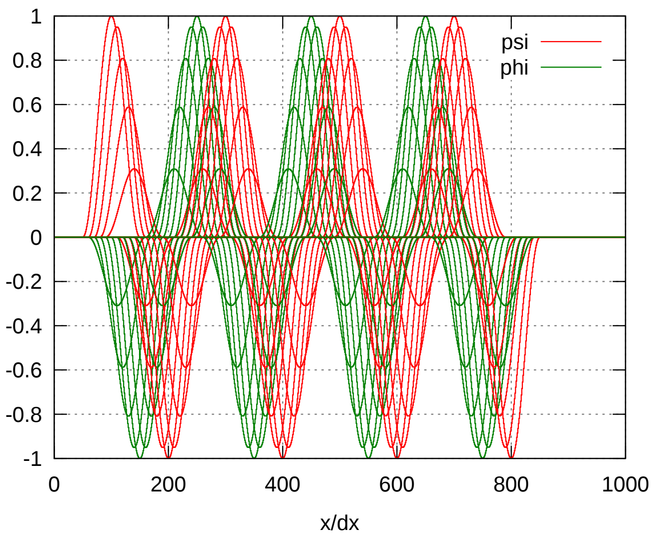

Figure 15. Simulation results of the example presented in Sect. 4.3. Abscissa marks the spatial dimension and ordinate represents the oscillator amplitude. The oscillator state is plotted every 20 time steps.

(partial differential equation) system (16) leads to the following system of coupled implicit algebraic equations:

$$\begin{aligned} \psi_i^{n+1} &= \psi_i^* + 0.5\ \Delta t\ \omega\ \phi_i^{n+1}, \\ \phi_i^{n+1} &= \phi_i^* - 0.5\ \Delta t\ \omega\ \psi_i^{n+1}, \end{aligned} \tag{17}$$

where ψ_i^* and ϕ_i^* stand for

$$\psi_i^* = \mathrm{MPDATA}\left(\psi_i^n + 0.5\ \Delta t\ \omega\ \phi_i^n, C\right), \tag{18}$$

$$\phi_i^* = \mathrm{MPDATA}\left(\phi_i^n - 0.5\ \Delta t\ \omega\ \psi_i^n, C\right). \tag{19}$$

Substituting in Eq. (17) ψ_i^{n+1} with ϕ_i^{n+1} and vice versa and then regrouping leads to

$$\begin{aligned} \psi_i^{n+1} &= \frac{\psi_i^* + 0.5\ \Delta t\ \omega\ \phi_i^*}{1 + (0.5\ \Delta t\ \omega)^2}, \\ \phi_i^{n+1} &= \frac{\phi_i^* - 0.5\ \Delta t\ \omega\ \psi_i^*}{1 + (0.5\ \Delta t\ \omega)^2}. \end{aligned} \tag{20}$$

Implementation of forcing terms prescribed in Eq. (20) is presented in Listing 27. A new solver **coupled_harmosc** is defined by inheriting from the **mpdata_rhs** class. A member field **omega** is defined to store the frequency of oscillations.

The rhs terms are defined for both variables, **ix::psi** and **ix::phi**, within the **update_rhs()** method. The method implements both implicit and explicit formulæ, the two cases are switched by the **at** argument. Defining forcings for both n and $n+1$ cases allows using this class with both **euler** and **trapez** options. The current state of the model is obtained via a call to the **state()** method. Note how the formulæ defined in **update_rhs()** in case (1) loosely resemble the mathematical notation presented in Eq. (20). The 0.5 is absent because the

```
#include <libmpdata++/solvers/mpdata_rhs.hpp>

template <class ct_params_t>
struct coupled_harmosc : public
  libmpdataxx::solvers::mpdata_rhs<ct_params_t>
{ // aliases
  using parent_t =
    libmpdataxx::solvers::mpdata_rhs<ct_params_t>;
  using ix = typename ct_params_t::ix;
  // member fields
  typename ct_params_t::real_t omega;

  // method called by mpdata_rhs
  void update_rhs(
    libmpdataxx::arrvec_t<
      typename parent_t::arr_t
    > &rhs,
    const typename parent_t::real_t &dt,
    const int &at
  ) {
    parent_t::update_rhs(rhs, dt, at);

    // just to shorten code
    const auto &psi = this->state(ix::psi);
    const auto &phi = this->state(ix::phi);
    const auto &i = this->i;

    switch (at)
    { // explicit solution for R^{n}
      // (note: with trapez used only at t=0)
      case (0):
      rhs.at(ix::psi)(i) += omega * phi(i);
      rhs.at(ix::phi)(i) -= omega * psi(i);
      break;

      // implicit solution for R^{n+1}
      case (1):
      rhs.at(ix::psi)(i) += (
        (psi(i) + dt * omega * phi(i))
        / (1 + pow(dt * omega, 2))
        - psi(i)
      ) / dt;
      rhs.at(ix::phi)(i) += (
        (phi(i) - dt * omega * psi(i))
        / (1 + pow(dt * omega, 2))
        - phi(i)
      ) / dt;
      break;
    }
  }
  // run-time parameters
  struct rt_params_t : parent_t::rt_params_t {
    typename ct_params_t::real_t omega = 0;
  };
  // ctor
  coupled_harmosc(
    typename parent_t::ctor_args_t args,
    const rt_params_t &p
  ) : parent_t(args, p), omega(p.omega)
  { assert(omega != 0); }
};
```

Listing 27. Definition of the solver used in the example presented in Sect. 4.3.

Δt passed as argument in **trapez** option is already divided by 2.

Next, the **rt_params_t** structure is augmented (by inheriting from parent's **rt_params_t**) with the **omega**. Lastly, the **coupled_harmosc** constructor is defined. Within it, the choice of the **omega** is handled by copying its value from the **p.omega** to **omega** member field and then checking if the user has altered the default value of 0.

```
struct ct_params_t : ct_params_default_t
{
  using real_t = double;
  enum { n_dims = 1 };
  enum { n_eqns = 2 };
  enum { rhs_scheme =
    solvers::rhs_scheme_t::trapez };
  struct ix { enum {psi, phi}; };
};
```

Listing 28. Compile-time parameter structure for the example presented in Sect. 4.3.

For inhomogeneous transport, the **rhs_scheme** within the **ct_params_t** structure needs to be defined. In this example it is set to **trapez** (Listing 28). MPDATA advection scheme options are set to default by inheriting from the **ct_params_t_default** structure. The structure **ix** allows calling advected variables by their labels, **phi** and **psi**, rather than integer numbers. Lastly, when defining the **rt_params_t** structure a value is assigned to the member field **p.omega**; see Listing 29.

```
// run-time parameters
using boost::math::constants::pi;
p.dt = 1;
p.omega = 2 * pi<real_t>() / p.dt / 400;
```

Listing 29. Run-time parameter structure for the example presented in Sect. 4.3.

In the present example, the initial condition for ψ is defined as $\psi(x) = 0.5[1+\cos(2\pi x/100)]$ for $x \in (50, 150)$ and zero elsewhere. The initial condition for ϕ is set to zero.

The result of 1400 s of simulated time is shown in Fig. 15. Note that the solutions for both ψ and ϕ remain in phase and feature no substantial amplitude error. This contrasts with calculations using Euler-forward schemes (not shown). In particular, at the end of the simulation, the rms error is 1×10^{-7} and 1×10^{-18} for the analogous experiment with $u_o \equiv 0$ (not shown).

5 Transport with prognosed velocity

5.1 Implemented algorithms

Whenever the velocity field changes in time, the second-order accuracy of the solution at $n+1$ requires an estimate of the advector at $n+1/2$. This is provided by linear extrapolations from n and $n-1$ values (Smolarkiewicz and Margolin, 1998). Furthermore, when the velocity is a dependent variable of the model, Eq. (12) embodies equations of motion. Then the velocity (or momentum) components are treated as advected scalars (i.e. advectees) and are predicted at the centres of the dual-grid cells (Fig. 3). The advector field is then interpolated linearly to the centres of the cell walls.

5.2 Library interface

The algorithms for interpolating in space and extrapolating in time the advector field from the model variables are defined in the **mpdata_rhs_vip** class and all user-created solvers with time-varying velocity must inherit from this class.

The transported fields may represent either velocity or momenta. In the latter case the prognosed velocity components are calculated as ratios of two advectee fields (e.g. momentum components and density). The index of the advectee that forms the common denominator for all velocity components should be assigned to **vip_den**. The **vip_i**, **vip_j** and **vip_k** store the index of the advected fields appearing in the numerators for each velocity component. These velocity components are then used to calculate the advector field. In cases when the velocity components are model variables (as in the example of Sect. 6.3), the common denominator is redundant and the value -1 should be assigned to **vip_den**.

For systems where numerators and denominators can uniformly approach zeros, the **vip_eps** value is defined to prevent divisions by zero. Then, if the denominator at a given grid-point is less than the **vip_eps**, the resulting advector is set to zero therein. The default setting (represented with **vip_eps** set to 0) gives no protection from divisions by zeros. Any user-defined **vip_eps** > 0 activates the above algorithm.

The **vip_i**, **vip_j**, **vip_k** and **vip_den** are expected to be members of the compile-time parameters structure **ct_params_t** of the **mpdata_rhs_vip** class. The **vip_eps** value is a run-time parameter.

As of the current release, the prognosed-velocity features of libmpdata++ are implemented for constant $G \equiv 1$ only.

5.3 Example: 1-D shallow-water system

The aim of this example is to show how to define simulations with prognosed velocity field. The necessary compile-time and run-time parameters as well as the user-defined class with source and sink terms are discussed. The obtained results are compared with the analytical solution and a published MPDATA benchmark.

The idealised system of 1-D inviscid shallow-water equations is considered, with both the surface friction and background rotation neglected. The simulated physical scenario is a slab-symmetric parabolic drop spreading under gravity; see Frei (1993) for a general context and Schär and Smolarkiewicz (1996) for the bespoke analytical solutions. The corresponding governing equations take the dimensionless form

$$\begin{aligned}&\partial_t h + \partial_x(uh) = 0,\\&\partial_t(uh) + \partial_x(uuh) = -h\partial_x h,\end{aligned} \tag{21}$$

where h is a normalised depth of the fluid layer and u is a normalised velocity. Following Schär and Smith (1993) the selected velocity scale is $u_o = (gh_o)^{1/2}$, where h_o is the initial height of the drop and g denotes the gravitational acceleration. The characteristic timescale is $t_o = a/u_o$, where a denotes the initial half-width of the drop. At the initial time a drop is confined to $|x| \le 1$ and centred about $x = 0$,

$$h(x,t=0) = \begin{cases}1-x^2, & \text{for } |x| \le 1\\ 0, & \text{for } |x| > 1.\end{cases} \tag{22}$$

The time step is set to 0.01 and the grid spacing is set to 0.05. The crux of the test is a synchronous solution for the depth and momentum near the drop edge that accurately diagnoses the velocity.

The definition of the rhs terms for Sect. 5.3 is presented in Listing 30. Only the method for calculating the forcing terms is shown; for the full out-of-the-library-tree definition of source-terms see Listing 27. As in Listing 27, the definition in Listing 30 attempts to follow the mathematical notation. Because of the use of the **grad** function, the **nabla** namespace is included.

```
void update_rhs(
  libmpdataxx::arrvec_t<
    typename parent_t::arr_t
  > &rhs,
  const typename parent_t::real_t &dt,
  const int &at
) {
  using namespace libmpdataxx::formulae::nabla;

  parent_t::update_rhs(rhs, dt, at);

  rhs.at(ix::qx)(this->i) -=
    this->g
    * this->state(ix::h)(this->i)
    * grad(this->state(ix::h), this->i, this->di);
}
```

Listing 30. Method for calculating source and sink terms in the example presented in Sect. 5.3.

Listing 31 specifies the compile-time parameters structure. Because fluid flow in this example is divergent the **opts::dfl** correction is enabled; cf. Sect. 3.1.3. The **rhs_scheme** is set to **trapez**.[20] Within the **ix** structure, the equation indices are assigned. Furthermore, the recipe for calculating the velocity is defined by assigning the indices to **vip_i** and **vip_den**. Lack of the rhs terms is specified by toggling the nth bit of the **hints_norhs** field, where n is the index of the homogeneous equation. This prevents the unnecessary summation of zeros.

```
template <int opts_arg>
struct ct_params_t : ct_params_default_t
{
  using real_t = ::real_t;
  enum { n_dims = 1 };
  enum { n_eqns = 2 };

  // options
  enum { opts = opts_arg | opts::dfl };
  enum { rhs_scheme = solvers::trapez };

  // indices
  struct ix {
    enum { qx, h };
    enum { vip_i=qx, vip_den=h };
  };

  // hints
  enum { hint_norhs = opts::bit(ix::h) };
};
```

Listing 31. Compile-time parameters for the example presented in Sect. 5.3.

Listing 32 shows the run-time parameters structure. The value of gravitational acceleration **p.g** is set to 1 to follow the dimensionless notation of Eq. (21), and the **vip_eps** is set arbitrarily to 10^{-8}.

```
// run-time parameters
typename solver_t::rt_params_t p;
p.dt = .01;
p.di = .05;
p.grid_size = { int(16 / p.di) };
p.g = 1;
p.vip_eps = 1e-8;
```

Listing 32. Run-time parameters for the example presented in Sect. 5.3.

The results of the test are plotted in Fig. 16. Figure 16a shows the initial condition (black) and the analytical solution for $t = 3$ (blue). Solid lines mark the fluid depth and the dashed line the velocity. The remaining two panels show

[20]Because the equation for h is homogeneous, the momentum forcing at $n+1$ time level can be readily evaluated after advecting h.

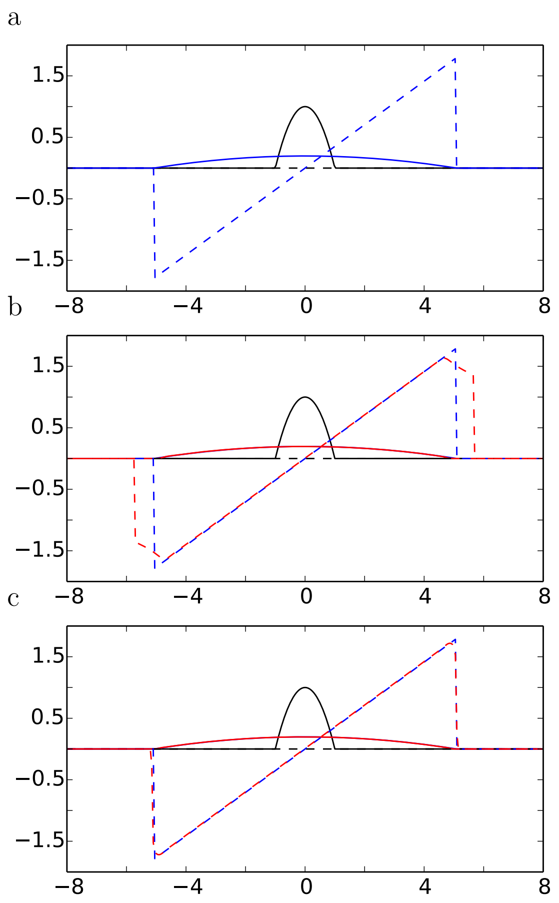

Figure 16. Simulation results of the example presented in Sect. 5.3. Solid lines represent fluid height and dashed lines represent fluid velocity. Initial condition is plotted in black, analytical solution in blue and test results in red. **(a)** shows the initial condition and analytical solution at $t = 3$. **(b)** and **(c)** show numerical results plotted over **(a)** obtained with options **abs** + **fct** and **iga** + **fct**, respectively.

numerical results[21] at $t = 3$ for different MPDATA options (red) plotted over the top panel. Figure 16b shows the solution with options **abs** and **fct**, whereas Fig. 16c shows the solution obtained with options **iga** and **fct**.

All presented results are free of apparent artefacts near the drop edge. The **abs**+**fct** in the central panel compares well with Fig. 7b in Schär and Smolarkiewicz (1996), whereas the **iga**+**fct** solution in the bottom panel closely reproduces the analytical result. The rms error, on the rhs of Eq. (8), at the end of the simulation is 5.77×10^{-4} for **abs**+**fct** and 1.87×10^{-4} for **iga**+**fct** options.

[21]Similar to advector field evaluation discussed in Sec. 5.2 the **vip_eps** value was used as cut-off value to prevent divisions by zero when calculating velocity field.

5.4 Example: 2-D shallow-water system

The 2-D shallow-water test discussed here is an original axis-symmetric extension of the 1-D slab-symmetric test in Sect. 5.3. The corresponding dimensionless equations take the form

$$\begin{aligned}
&\partial_t h + \partial_x(uh) + \partial_y(vh) = 0,\\
&\partial_t(uh) + \partial_x(uuh) + \partial_y(vuh) = -h\partial_x h, \qquad (23)\\
&\partial_t(vh) + \partial_x(uvh) + \partial_y(vvh) = -h\partial_y h.
\end{aligned}$$

As in the 1-D case, h stands for the fluid height and u and v are the velocity components in x and y directions, respectively. Again, the initial condition consists of a parabolic drop centred at the origin and confined to $x^2 + y^2 \le 1$,

$$h(x,y,t=0) = \begin{cases} 1 - x^2 - y^2, & \text{for } \sqrt{x^2+y^2} \le 1 \\ 0, & \text{for } \sqrt{x^2+y^2} > 1. \end{cases} \qquad (24)$$

Following the method presented by Frei (1993) and Schär and Smolarkiewicz (1996), the analytical solution of the system (23) can be obtained as

$$\begin{aligned}
&h(x,y,t) = \frac{1}{\lambda^2} - \frac{x^2+y^2}{\lambda^4},\\
&u(x,t) = x\frac{\dot\lambda}{\lambda}, \qquad (25)\\
&v(y,t) = y\frac{\dot\lambda}{\lambda}.
\end{aligned}$$

Here $\lambda(t)$ is half-width of the drop, evolving according to

$$\lambda(t) = \sqrt{2t^2+1} \qquad (26)$$

and $\dot\lambda \equiv \mathrm{d}\lambda/\mathrm{d}t$ is the velocity of the leading edge.

Figure 17 shows a perspective display of drop height at $t = 3$, whereas Fig. 18 shows the profiles of velocity and height of the drop. Similarly to Fig. 16, the top panel shows the initial condition (black) and analytical solution for $t = 3$ (blue). Central and bottom panels show corresponding numerical results at $t = 3$ (red). Solid lines represent the fluid height and the dashed lines the velocity. The central panel shows the solution with options **abs** and **fct**, whereas the bottom panel shows the solution with options **iga** and **fct**. As in the 1-D case, the velocity field near the drop edge is regular and the **iga**+**fct** result closely follows the analytical solution. The rms error for **abs** and **fct** equals 1.60×10^{-4} and for **abs** and **iga** 0.70×10^{-4}; see Jarecka et al. (2015) for a discussion.

6 Systems with elliptic pressure equation

6.1 Implemented algorithms

The libmpdata++ library includes an implicit representation of pressure gradient terms for incompressible fluid equations.

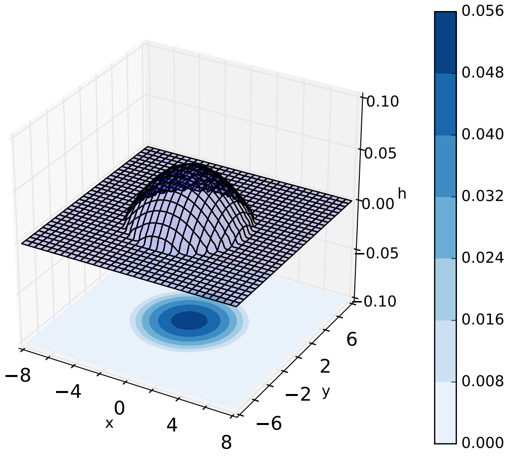

Figure 17. Drop height at $t = 3$ of the example presented in Sect. 5.4.

This necessitates the solution of an elliptic Poisson problem for pressure. The elliptic problem is solved after applying all explicitly known forcings to ensure a non-divergent velocity field at the end of each time step. As of the current release, the library is equipped with the minimal- and conjugate-residual variational iterative solvers. For the derivation of used schemes and further discussion of the elliptic problem see Smolarkiewicz and Margolin (1994), Smolarkiewicz and Szmelter (2011) and references therein.

6.2 Library interface

The methods for solving the elliptic problem are implemented in the **mpdata_rhs_vip_prs** class (Fig. 1). This class inherits from the **mpdata_rhs_vip** class. Therefore, the way to specify other source terms as well as the time-varying velocity field remains unchanged.

The choice of elliptic solver is controlled by setting the compile-time parameter **prs_scheme** to **mr** and **cr** for the minimal-residual and conjugate-residual solver, respectively. The iterations within the elliptic solver stop when the divergence of the velocity field is lower than a threshold tolerance set by a run-time parameter **prs_tol** (cf. Smolarkiewicz et al., 1997).

6.3 Example: Boussinesq convection

The goal of this example is to show the user interface for simulations featuring an elliptic pressure equation. The governing PDE system consists of momentum, potential temperature, and mass-continuity equations for an ideal, 2-D, incompressible Boussinesq fluid

$$\partial_t \boldsymbol{v} + \nabla \cdot (\boldsymbol{v} \otimes \boldsymbol{v}) = -\nabla \pi - \boldsymbol{g}\frac{\theta'}{\theta_o}, \tag{27}$$

$$\partial_t \theta + \nabla \cdot (\boldsymbol{v}\theta) = 0, \tag{28}$$

$$\nabla \cdot \boldsymbol{v} = 0. \tag{29}$$

Here, $\boldsymbol{v} = (u, w)$ denotes the velocity field, π is the pressure perturbation about the hydrostatic reference state normalised by the reference density ρ_o constant in the Boussinesq model and $\boldsymbol{g}$ is the gravitational acceleration. Furthermore, θ' represents the potential temperature perturbation about the reference state $\theta_o =$ constant, and $\otimes$ denotes the tensor product.

Combining the velocity prediction from the momentum equation, according to Eq. (15), with the mass continuity Eq. (29) leads to the elliptic Poisson problem:

$$-\frac{1}{\rho_o}\nabla \cdot (\rho_o(\widehat{\boldsymbol{v}} - 0.5\Delta t \nabla \pi)) = 0, \tag{30}$$

where $\widehat{\boldsymbol{v}}$ is the velocity field after the advection summed with all the explicitly known source terms at time level $n+1$,

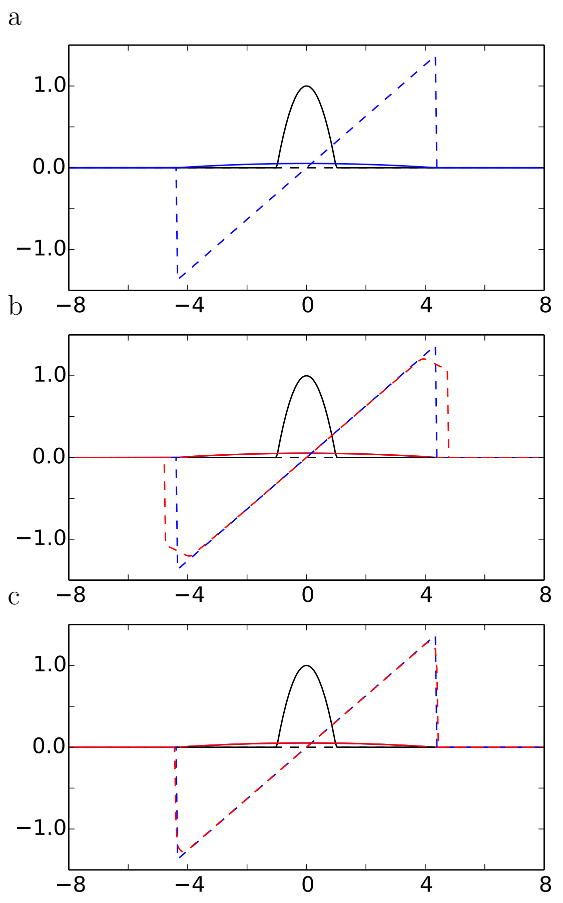

Figure 18. The same as in Fig. 16 but for a cross section of the two-dimensional case.

namely buoyancy in this example.[22] In Eq. (30) the pressure perturbation field π is unknown, and it needs to be adjusted such that the final velocity field $\widehat{\boldsymbol{v}} - 0.5\Delta t \nabla\pi$ satisfies the mass continuity Eq. (29). Denoting $0.5\Delta t\pi$ as ϕ allows symbolising Eq. (30) using standard notation for linear sparse problems (Smolarkiewicz and Margolin, 1994):

$$\mathcal{L}(\phi) - \mathcal{R} = 0. \tag{31}$$

The setup of the test follows Smolarkiewicz and Pudykiewicz (1992). It consists of a circular potential temperature anomaly of radius 250 m, embedded in a neutrally stratified quiescent environment, with $\theta_\mathrm{o} = 300\,\mathrm{K}$, in the domain resolved with 200×200 grid cells of the size $\mathrm{d}x = \mathrm{d}y = 10\,\mathrm{m}$. The initial anomaly $\theta' = 0.5\,\mathrm{K}$ is centred in the horizontal, 260 m above the lower boundary. The value of g is set to $9.81\,\mathrm{m\,s^{-1}}$. The time step is set to $\Delta t = 0.75\,\mathrm{s}$ and the simulation takes 800 time steps.

```
struct ct_params_t : ct_params_default_t
{
  using real_t = double;
  enum { n_dims = 2 };
  enum { n_eqns = 3 };
  enum { rhs_scheme = solvers::trapez };
  enum { prs_scheme = solvers::cr };
  struct ix { enum {
    u, w, tht,
    vip_i=u, vip_j=w, vip_den=-1
  }; };
};
```

Listing 33. Compile-time parameters for the example presented in Sect. 6.3.

Listing 33 shows the compile-time parameters structure. The time integration scheme for the buoyancy forcing is set to **trapez**, as the user has a choice of the algorithm. However, as of the current release, the elliptic problem formulation requires forcings to be independent of velocity if handled using the **trapez** scheme. The implicit pressure gradient terms are always integrated with the trapezoidal rule (15), regardless of the **rhs_scheme** setting. In Listing 33 the elliptic solver option is set to the conjugate-residual scheme **cr**. The **vip_den** is set to -1, because here the velocity components are the model kinematic variables; cf. the discussion in second paragraph of Sect. 5.2.

The convergence threshold of the elliptic solver, $\nabla \cdot (\boldsymbol{v}) \leq \varepsilon$, is set to 10^{-7} via the run-time parameter **prs_tol** (Listing 34).

Listing 35 shows the buoyancy forcing definition.

The evolved θ fields after 200, 600 and 800 time steps are shown in Fig. 19a–c. These results correspond to plots

```
p.prs_tol = 1e-7;
```

Listing 34. Run-time parameter field setting the accuracy of the pressure solver.

```
// explicit forcings
void update_rhs(
  libmpdataxx::arrvec_t<
    typename parent_t::arr_t
  > &rhs,
  const real_t &dt,
  const int &at
) {
  parent_t::update_rhs(rhs, dt, at);

  const auto &Tht = this->state(ix::tht);
  const auto &ijk = this->ijk;

  rhs.at(ix::w)(ijk) +=
    g * (Tht(ijk) - Tht_ref) / Tht_ref;
}
```

Listing 35. Method for calculating source and sink terms in the example presented in Sect. 6.3.

from Fig. 3 in Smolarkiewicz and Pudykiewicz (1992) and illustrate that libmpdata++ captures the interfacial instabilities and sharp gradients, including small turbulent structures in Fig. 19c. Yet, the solutions contain small (imperceptible in the plots) under- and overshoots, developing at the rate of $\Delta\theta/\Delta t \sim \Delta t \theta_\mathrm{o} \nabla \cdot (\boldsymbol{v})$. These oscillations depend on the magnitude of the residual errors, $\nabla \cdot \boldsymbol{v} \neq 0$, controlled by the convergence threshold **prs_tol**. For substantiation, Table 1 displays the magnitude of such spurious extrema $\delta\theta_\mathrm{max}$ – defined as the larger from the maximal magnitudes of normalised under- and overshoots with respect to their initial values – against **prs_tol** at the time of Fig. 19c. Note that $\delta\theta_\mathrm{max}$ is bounded by **prs_tol**($\times\, 800\Delta t$).

The conservation errors for θ' and $(\theta')^2$ are defined as

$$\mathrm{err1} = \frac{\sum\theta' - \sum\theta'_\mathrm{o}}{\sum\theta'_\mathrm{o}} 100\,\%\,, \tag{32}$$

$$\mathrm{err2} = \frac{\sum(\theta')^2 - \sum(\theta'_\mathrm{o})^2}{\sum(\theta'_\mathrm{o})^2} 100\,\%\,, \tag{33}$$

where θ'_o indicates the initial perturbation and $\sum$ stands for summing over the whole computational domain. At the end of the simulation $\mathrm{err1} \approx 1 \times 10^{-11}$ is orders of magnitude smaller than in semi-Lagrangian calculations of Smolarkiewicz and Pudykiewicz (1992), whereas $\mathrm{err2} = -14$ matches their value, reflecting the implicit LES (ILES) property of non-oscillatory numerics; see Smolarkiewicz (2006), Prusa et al. (2008) and references therein.

[22] Because the potential temperature Eq. (28) is homogeneous, the buoyancy at the $n+1$ time level can be readily evaluated after advecting θ.

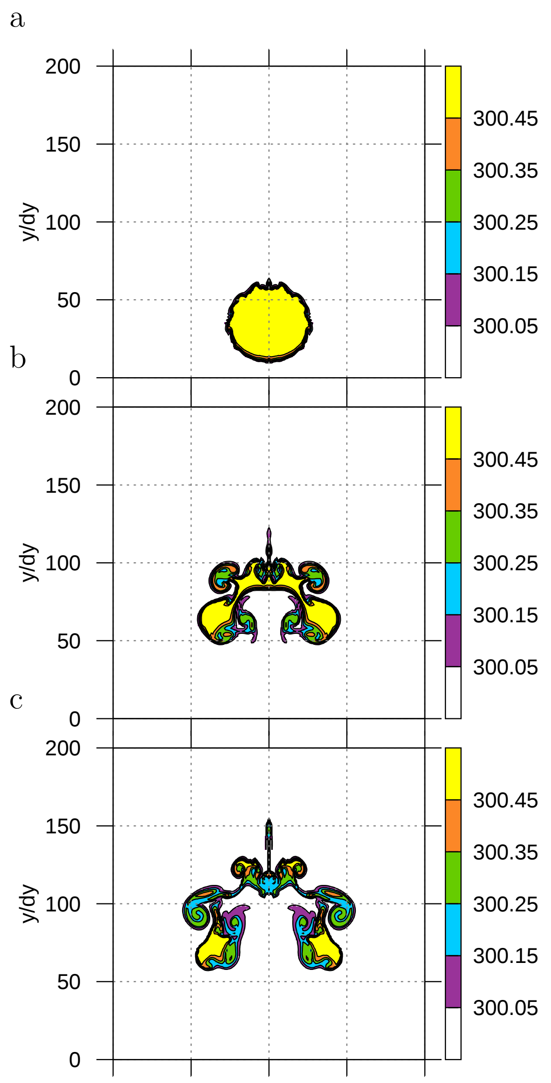

Figure 19. The results of the example presented in Sect. 6.3. Abscissa and ordinate mark the spatial dimensions. Colours correspond to potential temperature values. Panel **(a)** shows results from the 200th, **(b)** from the 600th and **(c)** from the 800th time step.

Table 1. Maximal spurious extrema of the θ field after 800 time steps for various values of the convergence threshold **prs_tol**.

prs_tol	10^{-5}	10^{-7}	10^{-9}
$\delta\theta_{\max}$	3×10^{-4}	8×10^{-6}	1×10^{-7}

7 Remarks

In this paper the first release of libmpdata++ was introduced. Versatility of the user interface as well as the correctness of the implementation were illustrated with a series of examples with increasing degree of physical, mathematical and programming complexity. Starting from elementary advection in the Cartesian domain, through passive advection on the sphere, through slab- and axis-symmetric water drop spreading under gravity, to buoyant convection in an incompressible Boussinesq fluid, the accompanying discussions included code snippets, description of the user interface and comparison with previously published benchmarks.

Our priority in the development of libmpdata++ is the researcher productivity. In case of scientific software such as libmpdata++, the researchers are both users and developers of the library. The adherence to the principle of separation of concerns and employment of programming techniques that promote code conciseness – e.g. the current release consists of less than 10k lines of code – contribute to the developers' productivity. The user productivity is amplified by ensuring that the release of the library is accompanied with example-rich documentation. Both the users and developers benefit from the free/libre open-source software release of the library.

Our experience with the current version of libmpdata++ indicates that the embraced object-oriented techniques and modular design of the library generally do not come as a trade-off for performance. On small grids, however, there is a noticeable overhead compared to the original Fortran implementation. For example, in serial runs, up to 5-times longer execution times were measured for the 3-D revolving-sphere tests discussed in Sect. 3.7 (59^3 grid). The relative performance improves with increasing grid size, reaching execution times on a par with the original Fortran implementation on the $(6 \times 59)^3$ grid. On the other hand, the separation of concerns obtained with the object-oriented design of the library allowed equipping the code with the multi-threading mechanism, without any substantial changes in the numerics code. Noteworthy, for all 2-D and 3-D examples presented in the paper, a minimum of fivefold speed-up is obtained when executing on six threads. The library is in active development and improvements in performance are expected. Furthermore, equipping the library with distributed-memory parallelisation is planned for a subsequent release.

Appendix A: Index of options

Tables A1, A2 and A3 provide a reference of libmpdata++ options documented in the article.

Table A1. Fields of compile-time parameter structures.

Parameter	Possible values	Relevant section	Relevant listing	Short description
available in **mpdata** and inheriting classes				
opts	combinations of **abs, dfl, fct, iga, nkh, nug, tot, eps, npa**	3, 3.4	11, 14, 15	MPDATA algorithm options (see Table A3). Options can be combined with the "|" operator.
real_t	**float**, **double**	3.3	1, 7	Floating point format used.
n_dims	integer constant	3.3	1, 7, 16, 20	Dimensionality of the solved problem.
n_eqns	integer constant	3.3	1, 7, 28	Number of advected variables (number of the solved equations).
available in **mpdata_rhs** and inheriting classes				
rhs_scheme	**euler_a**, **euler_b**, **trapez**	4	28, 31	Source/sink term integration scheme
hint_norhs	integer constant interpreted as a bit field indexed by equation number	5.3	31	Flag for equations with no source/sink terms (to avoid summation of zeros when calculating source terms).
available in **mpdata_rhs_vip** and inheriting classes				
vip_i, **vip_j**, **vip_k**	integer constant	5.2	31, 33	Indices of advected variables representing velocity or momentum components in up to three dimensions.
vip_den	integer constant	5.2	31, 33	Optional index of density-like advected variable by which the above-defined momenta are divided to obtain velocity.
available in **mpdata_rhs_vip_prs** and inheriting classes				
prs_scheme	**solvers::mr**, **solvers::cr**	6.2	33	Elliptic pressure solver algorithm type (minimal-residual or conjugate-residual).

Table A2. Fields of run-time parameter structures.

Parameter	Possible values	Relevant sections	Relevant listings	Short description
available in **mpdata** and inheriting classes				
n_iters	integer constant	3.1.1, 3.5	17	Number of corrective iterations performed within the MPDATA algorithm. One iteration results in a donor-cell scheme. Two (the default) or more iterations result in MPDATA scheme.
grid_size	array of integer constants	3.2.3, 3.3	7	Number of grid points per each dimension.
available in **mpdata_rhs** and inheriting classes				
dt	floating-point constant	4.2	29, 32	Time step.
available in **mpdata_rhs_vip** and inheriting classes				
vip_eps	floating-point constant	5.2	32	Cut-off value for preventing divisions by zero when calculating velocity field from momenta (for simulations in which the advected variables represent momenta and not velocity).
di, dj, dk	floating-point constant		32	Grid spacing.
available in **mpdata_rhs_vip_prs** and inheriting classes				
prs_tol	floating-point constant	6.2	34	Tolerance of the elliptic pressure solver.
common to all output handlers				
outfreq	integer number	3.4	7, 9	Output interval (in number of time steps). The default value is set to 1, resulting in output performed in every time step.
outdir	string		22	Directory where output files are saved.
outvars	map associating equation indices with pairs of strings representing names and units of advected variables	3.4	9	List of variables to include in the output files. Mandatory for simulations with more than one advected field.

Table A3. Options of MPDATA defined through the compile-time parameter **opts** (see Listings 11–15).

Option	Default	Relevant section	Short description
abs		3.1.5, 3.4.1, 3.5, 3.6, 5.3, 5.4	Using absolute values in "pseudo-velocity" formulation. (One of the two possible options for handling variable-sign signals.)
dfl		3.1.3, 5.3, 5.4	Augmenting the "pseudo-velocity" formulæ with a term proportional to flow divergence. (To be used with divergent flows only.)
fct	✓	3.1.4, 3.4.3, 3.5, 3.6	Non-oscillatory option of MPDATA.
iga	✓	3.1.5, 3.4.1, 3.5, 3.6, 5.3, 5.4	Linear limit of MPDATA algorithm at infinite constant background. (One of the two possible options for handling variable-sign signals.)
khn		3.1.1	Employing Kahan summation algorithm in donor-cell calls of MPDATA algorithm.
nug		3.8	Accounting for non-constant density of the fluid and/or coordinate transformation.
tot		3.1.2, 3.4.2, 3.5, 3.6, 3.7	Accounting for third-order terms in "pseudo-velocity" formulæ.
pfc			Protecting from divisions by zero in ψ-fraction factors (as the last term in Eq. (6)) by conditionally assigning zeros to all grid points for which the denominator equals zero. The default is to augment the denominator with a small positive number ϵ ($\sim 10^{-7}$ for single precision and $\sim 10^{-16}$ for double precision). The default behaviour requires the signal to be non-negative unless **iga** or **abs** is selected.
npa			Evaluating $[u]^+$ as $(u+\|u\|)/2$ instead of $\max(u,0)$ (and analogously for $[u]^-$; see Eq. 5)

Code availability

The library is released under the GNU General Public License v3.0. The 1.0 release of the library accompanying this publication is available for download as an electronic supplement to the paper and tagged as "1.0.1" at the project repository; see project website for a list of pointers to relevant resources: http://libmpdataxx.igf.fuw.edu.pl/.

All example programs needed to generate plots and error norms discussed in the paper are shipped with libmpdata++ and are located in the "paper_2015_GMD" folder of the release tarball and the public code repository. To allow automatic regression testing, reference data in the form of both model output (e.g. hdf5 files) and calculated error norms (text files) are stored in "refdata" subfolders. Execution of test programs and verification of the output against reference data is automated using CMake/CTest and is a part of the continuous-integration workflow used in development of the library. It takes ca. 15 min to execute all the discussed example programs on commodity hardware (e.g. a multi-core laptop or a virtual machine in a cloud-computing system).

The Supplement related to this article is available online at doi:10.5194/gmd-8-1005-2015-supplement.

Acknowledgements. Personal reviews from Christian Kühnlein and Willem Deconinck as well as the reviews from Christian Jacobs and Douglas Jacobsen helped to improve the presentation. Development of libmpdata++ was funded by Poland's National Science Centre (Narodowe Centrum Nauki) (decisions no. 2011/01/N/ST10/01483 and 2012/06/M/ST10/00434). P. K. Smolarkiewicz acknowledges support by funding received from the European Research Council under the European Union's Seventh Framework Programme (FP7/2012/ERC grant agreement no. 320375). Part of the work was carried out during visits of A. Jaruga to the National Center for Atmospheric Research (NCAR) in Boulder, Colorado, USA, and to the European Centre for Medium-Range Weather Forecasts (ECMWF), Reading, UK. Part of the work was carried out during visits of D. Jarecka to NCAR funded by Polish Ministry of Science and Higher Education – project no. 1119/MOB/13/2014/0. NCAR is operated by the University Corporation for Atmospheric Research. Figures were generated using gnuplot, ParaView and matplotlib. The authors acknowledge computational time granted by the Center for Cooperative Work on Computational Science, University of Hyogo, Japan. Development of libmpdata++ continuously benefits from the computational services offered by Travis at their continuous-integration platform.

Edited by: H. Weller

References

Arabas, S., Jarecka, D., Jaruga, A., and Fijałkowski, M.: Formula translation in Blitz++, NumPy and modern Fortran: a case study of the language choice tradeoffs, Sci. Prog., 22, 201–222, doi:10.3233/SPR-140379, 2014.

Arakawa, A. and Lamb, V. R.: Computational design of the basic dynamical process of the UCLA general circulation model, in: General Circulation Models of the Atmosphere, vol. 17 of Methods in Computational Physics: Advances in Research and Applications, Elsevier, 173–265, doi:10.1016/B978-0-12-460817-7.50009-4, 1977.

Bangerth, W. and Heister, T.: What makes computational open source software libraries successful?, Comp. Sci. & Discuss., 6, 015010, doi:10.1088/1749-4699/6/1/015010, 2013.

Charbonneau, P. and Smolarkiewicz, P.: Modeling the solar dynamo, Science, 340, 42–43, doi:10.1126/science.1235954, 2013.

Cotter, C. S., Smolarkiewicz, P. K., and Szczyrba, I. N.: A viscoelastic fluid model for brain injuries, Int. J. Numer. Meth. Fl., 40, 303–311, doi:10.1002/fld.287, 2002.

Frei, C.: Dynamics of a two-dimensional ribbon of shallow water on an f-plane, Tellus A, 45, 44–53, doi:10.1034/j.1600-0870.1993.00004.x, 1993.

Grabowski, W. and Smolarkiewicz, P.: A multiscale anelastic model for meteorological research, Mon. Weather Rev., 130, 939–955, doi:10.1175/1520-0493(2002)130<0939:AMAMFM>2.0.CO;2, 2002.

Hyman, J., Smolarkiewicz, P., and Winter, C.: Heterogeneities of flow in stochastically generated porous media, Phys. Rev. E, 86, 056701, doi:10.1103/PhysRevE.86.056701, 2012.

Jarecka, D., Jaruga, A., and Smolarkiewicz, P.: A spreading drop of shallow water, J. Comput. Phys., 289, 53–61, doi:10.1016/j.jcp.2015.02.003, 2015.

Kahan, W.: Pracniques: further remarks on reducing truncation errors, Comm. ACM, 8, p. 40, doi:10.1145/363707.363723, 1965.

Kühnlein, C., Smolarkiewicz, P., and Dörnbrack, A.: Modelling atmospheric flows with adaptive moving meshes, J. Comput. Phys., 231, 2741–2763, doi:10.1016/j.jcp.2011.12.012, 2012.

Margolin, L. and Smolarkiewicz, P.: Antidiffusive velocities for multipass donor cell advection, J. Sci. Comput., 20, 907–929, doi:10.1137/S106482759324700X, 1998.

Maurin, K.: Analysis Part II: Integration, Distributions, Holomorphic Functions, Tensor and Harmonic Analysis, Reidel, 1980.

Molenkamp, C.: Accuracy of finite-difference methods applied to the advection equation., J. Appl. Meteorol., 7, 160–167, doi:10.1175/1520-0450(1968)007<0160:AOFDMA>2.0.CO;2, 1968.

Ortiz, P. and Smolarkiewicz, P. K.: Coupling the dynamics of boundary layers and evolutionary dunes, Phys. Rev., 79, 041307, doi:10.1103/PhysRevE.79.041307, 2009.

Press, W., Teukolsky, S., Vetterling, W., and Flannery, B.: Numerical recipes. The art of scientific computing, 3rd edn., Cambridge University Press, 2007.

Prusa, J., Smolarkiewicz, P., and Wyszogrodzki, A.: EULAG, a computational model for multiscale flows, Comput. Fluids, 37, 1193–1207, doi:10.1016/j.compfluid.2007.12.001, 2008.

Randall, D.: Lectures on numerical modelling of the atmosphere, available at: http://kiwi.atmos.colostate.edu/group/dave/at604pdf/Chapter_11.pdf (last access: October 2014), 2013.

Schär, C. and Smith, R. B.: Shallow-water flow past isolated topography. I – Vorticity production and wake formation, J. Atmos. Sci., 50, 1373–1412, doi:10.1175/1520-0469(1993)050<1373:SWFPIT>2.0.CO;2, 1993.

Schär, C. and Smolarkiewicz, P.: A synchronous and iterative flux-correction formalism for coupled transport equations, J. Comput. Phys., 128, 101–120, doi:10.1006/jcph.1996.0198, 1996.

Smolarkiewicz, P.: A simple positive definite advection scheme with small implicit diffusion, Mon. Weather Rev., 111, 479–486, doi:10.1175/1520-0493(1983)111<0479:ASPDAS>2.0.CO;2, 1983.

Smolarkiewicz, P.: A fully multidimensional positive definite advection transport algorithm with small implicit diffusion, J. Comput. Phys., 54, 325–362, doi:10.1016/0021-9991(84)90121-9, 1984.

Smolarkiewicz, P.: Multidimensional positive definite advection transport algorithm: an overview, Int. J. Numer. Meth. Fl., 50, 1123–1144, doi:10.1002/fld.1071, 2006.

Smolarkiewicz, P. and Clark, T.: The multidimensional positive definite advection transport algorithm – further development and applications, J. Computat. Phys., 67, 396–438, doi:10.1016/0021-9991(86)90270-6, 1986.

Smolarkiewicz, P. and Grabowski, W.: The multidimensional positive definite advection transport algorithm: Nonoscillatory option, J. Comput. Phys., 86, 355–375, doi:10.1016/0021-9991(90)90105-A, 1990.

Smolarkiewicz, P. and Margolin, L.: Variational elliptic solver for atmospheric applications, Tech. Rep. LA-12712-MS, Los Alamos National Lab., doi:10.2172/10130964, 1994.

Smolarkiewicz, P. and Margolin, L.: MPDATA: a finite-difference solver for geophysical flows, J. Comput. Phys., 140, 459–480, doi:10.1006/jcph.1998.5901, 1998.

Smolarkiewicz, P. and Pudykiewicz, J.: A class of semi-Lagrangian approximations for fluids, J. Atmos. Sci., 49, 2082–2096, doi:10.1175/1520-0469(1992)049<2082:ACOSLA>2.0.CO;2, 1992.

Smolarkiewicz, P. and Rasch, P.: Monotone advection on the sphere: an Eulerian versus semi-Lagrangian approach, J. Atmos. Sci., 48, 793–810, doi:10.1175/1520-0469(1991)048<0793:MAOTSA>2.0.CO;2, 1991.

Smolarkiewicz, P. and Szmelter, J.: MPDATA: an edge-based unstructured-grid formulation, J. Comp. Phys., 206, 624–649, doi:10.1016/j.jcp.2004.12.021, 2005.

Smolarkiewicz, P. and Szmelter, J.: Iterated upwind schemes for gas dynamics, J. Comput. Phys., 228, 33–54, doi:10.1016/j.jcp.2008.08.008, 2009.

Smolarkiewicz, P. and Szmelter, J.: A nonhydrostatic unstructured-mesh soundproof model for simulation of internal gravity waves, Acta Geophys., 59, 1109–1134, doi:10.2478/s11600-011-0043-z, 2011.

Smolarkiewicz, P. K., Grubišić, V., and Margolin, L. G.: On forward-in-time differencing for fluids: stopping criteria for iterative solutions of anelastic pressure equations, Mon. Weather Rev., 125, 647–654, doi:10.1175/1520-0493(1997)125<0647:OFITDF>2.0.CO;2, 1997.

Smolarkiewicz, P., Kühnlein, C., and Wedi, N.: A consistent framework for discrete integrations of soundproof and compressible PDEs of atmospheric dynamics, J. Comput. Phys., 263, 185–205, doi:10.1016/j.jcp.2014.01.031, 2014.

Szmelter, J. and Smolarkiewicz, P.: An edge-based unstructured mesh discretisation in geospherical framework, J. Comput. Phys., 229, 4980–4995, doi:10.1016/j.jcp.2010.03.017, 2010.

Veldhuizen, T.: Blitz++ user's guide. A C++ class library for scientific computing, Tech. rep., available at: http://blitz.sf.net/resources/blitz-0.9.pdf (last access: October 2014), 2006.

Williamson, D. and Rasch, P.: Two-dimensional semi-Lagrangian transport with shape-preserving interpolation, Mon. Weather Rev., 117, 102–129, doi:10.1175/1520-0493(1989)117<0102:TDSLTW>2.0.CO;2, 1989.

Wilson, G., Aruliah, D. A., Brown, C. T., Chue Hong, N. P., Davis, M., Guy, R. T., Haddock, S. H. D., Huff, K., Mitchell, I. M., Plumbley, M., Waugh, B., White, E. P., and Wilson, P.: Best practices for scientific computing, PLoS Biol., 12, e1001745, doi:10.1371/journal.pbio.1001745, 2014.