Scientific Programming 22 (2014) 201–222
DOI 10.3233/SPR-140379
IOS Press

Formula translation in Blitz++, NumPy and modern Fortran: A case study of the language choice tradeoffs

Sylwester Arabas [a,*], Dorota Jarecka [a], Anna Jaruga [a] and Maciej Fijałkowski [b]
[a] *Institute of Geophysics, Faculty of Physics, University of Warsaw, Warsaw, Poland*
[b] *PyPy Team*

Abstract. Three object-oriented implementations of a prototype solver of the advection equation are introduced. The presented programs are based on Blitz++ (C++), NumPy (Python) and Fortran's built-in array containers. The solvers constitute implementations of the Multidimensional Positive-Definite Advective Transport Algorithm (MPDATA). The introduced codes serve as examples for how the application of object-oriented programming (OOP) techniques and new language constructs from C++11 and Fortran 2008 allow to reproduce the mathematical notation used in the literature within the program code. A discussion on the tradeoffs of the programming language choice is presented. The main angles of comparison are code brevity and syntax clarity (and hence maintainability and auditability) as well as performance. All performance tests are carried out using free and open-source compilers. In the case of Python, a significant performance gain is observed when switching from the standard interpreter (CPython) to the PyPy implementation of Python. Entire source code of all three implementations is embedded in the text and is licensed under the terms of the GNU GPL license.

Keywords: Object-oriented programming, advection equation, MPDATA, C++, Fortran, Python

1. Introduction

Object-oriented programming (OOP) "*has become recognised as the almost unique successful paradigm for creating complex software*" [25, Section 1.3]. It is intriguing that, while the quoted statement comes from the very book subtitled *The Art of Scientific Computing*, hardly any (if not none) of the currently operational weather and climate prediction systems – flagship examples of complex scientific software – make extensive use of OOP techniques.[1]

Application of OOP techniques in development of numerical modelling software may help to:

(i) maintain modularity and separation of program logic layers (e.g. separation of numerical algorithms, parallelisation mechanisms, data input/output, error handling and the description of physical processes); and

(ii) shorten and simplify the source code and improve its readability by reproducing within the program logic the mathematical notation used in the literature.

The first application is attainable, yet arguably cumbersome, with procedural programming. The latter, virtually impossible to obtain with procedural programming, is the focus of this paper. The importance of reproducing the mathematical notation in the code lays primarily in the fact that code readability and brevity significantly contribute to code maintainability [37].

The key aim of this paper is to show how OOP techniques can be used to faithfully reproduce within the code what can be referred to as *blackboard abstractions* [26]. These may relate to several levels of mathematical abstraction. Object-oriented logic can be used to make the code resemble analytical formulae (e.g. [35]) and/or numerical algorithms, the latter being exemplified in this paper. For this purpose, a sample implementation of a numerical scheme for solving the advection equation is introduced in C++, Python and modern Fortran – OOP languages commonly used in scientific computing (see e.g. [9, Chapter 8]). Pre-

[*] Corresponding author. E-mail: sarabas@igf.fuw.edu.pl.

[1] Fortran has been the language of choice in oceanic [12], weather-prediction [32] and Earth system [16] modelling, and none of its 20th-century editions were object-oriented languages (for discussion, see e.g. [20]).

sented implementations and the results of benchmark tests provide a basis for discussion on the tradeoffs of programming language choice. The discussion concerns in principle the development of finite-difference solvers for partial differential equations, but is likely applicable to some extent to the scientific programming in general.

All three programs include an equally structured implementation of the two-dimensional version of the Multidimensional Positive Definite Advective Transport Algorithm (MPDATA) [27]. MPDATA is an example of a numerical procedure used in weather, climate, ocean and solar simulation systems (e.g. [1,8,10,39], respectively). The basic MPDATA scheme presented herein is complex enough to contain a wide range of mathematical abstractions that can be represented using OOP constructs, yet it is simple enough to allow inclusion of the entire source code within the paper text. All relevant MPDATA formulae are given in the text alongside corresponding code fragments allowing comparison of the relevant syntax with the mathematical notation. These formulae are presented without derivation or detailed discussion (see [28] for a recent review of MPDATA-based techniques including an introductory description of the algorithm and an exhaustive list of references).

The paper is structured as follows. In Section 2 we introduce the "formula translation" part of the three implementations briefly describing the algorithm itself and discussing where and how the OOP techniques were applied in its implementation. The remaining part of the implementations – the solver logic – is presented in Appendix A. Usage example is given in Appendix B. Section 3 covers performance evaluation of the three implementations. Section 4 covers discussion of the tradeoffs of the programming language choice. Section 5 closes the article with a brief summary.

The entire code is licensed under the terms of the GNU General Public License version 3 [29]. All listings include line numbers printed to the left of the source code, with separate numbering for C++ (listings prefixed with C, black frame),

listing C.0 (C++)
```
1 // code licensed under the terms of GNU GPL v3
2 // copyright holder: University of Warsaw
```

Python (listings prefixed with P, blue frame[2]) and

listing P.0 (Python)
```
1 # code licensed under the terms of GNU GPL v3
2 # copyright holder: University of Warsaw
```

Fortran (listings prefixed with F, red frame).

listing F.0 (Fortran)
```
1 ! code licensed under the terms of GNU GPL v3
2 ! copyright holder: University of Warsaw
```

Programming language constructs when inlined in the text are typeset in bold, e.g. **GOTO 2**.

2. Implementation of the formulae

Double-precision floating-point format is used in all three implementations. The codes begin with the following definitions:

listing C.1 (C++)
```
3 using real_t = double;
```

listing P.1 (Python)
```
3 real_t = 'float64'
```

listing F.1 (Fortran)
```
3 module real_m
4   implicit none
5   integer, parameter :: real_t = kind(0.d0)
6 end module
```

which provide a convenient way of switching to different precision.[3]

All codes are structured in a way allowing compilation of the code in exactly the same order as presented in the text within one source file.

The language syntax and OOP nomenclature are used without introduction in the paper. For an overview of OOP in context of C++, Python and Fortran, consult for example [31, Part III], [22, Chapter 5] and [18, Chapter 11], respectively.

2.1. Array containers

MPDATA is, in its most basic form presented herein, a solver for systems of advection equations of the following form:

$$\partial_t \psi = -\nabla \cdot (\vec{v}\psi) \qquad (1)$$

that describe evolution of a scalar field ψ transported by the fluid flow with velocity $\vec{v}$. Solution of Eq. (1) using MPDATA implies discretisation onto a grid of the scalar field ψ and the Courant number vector field $\vec{C}$. An "x" component of the Courant number field is defined as $C_x = v_x \cdot \frac{\Delta t}{\Delta x}$, where Δt is the solver timestep and Δx is the grid spacing.

Presented C++ implementation of MPDATA is built upon the Blitz++ library.[4] Blitz offers object-oriented representation of n-dimensional arrays, and

[2]The colors are visible in the online version of the article; http://dx.doi.org/10.3233/SPR-140379.

[3]Fortran's **selected_real_kind**() intrinsic function may be used instead to improve portability.

[4]Blitz++ is a C++ class library for scientific computing which uses the expression templates technique to achieve high performance, see http://sf.net/projects/blitz/.

array-valued mathematical expressions. In particular, it offers loop-free notation for array arithmetics that does not incur creation of intermediate temporary objects. Blitz++ is a header-only library[5] – to use it, it is enough to include the appropriate header file, and optionally expose the required classes to the present namespace:

listing C.2 (C++)
```cpp
#include <blitz/array.h>
using arr_t = blitz::Array<real_t, 2>;
using rng_t = blitz::Range;
using idx_t = blitz::RectDomain<2>;
```

Here **arr_t**, **rng_t** and **idx_t** serve as alias identifiers and are introduced in order to shorten the code.

The power of Blitz++ comes from the ability to express array expressions as objects. In particular, it is possible to define a function that returns an array expression; i.e. not the resultant array, but an object representing a "recipe" defining the operations to be performed on the arguments. As a consequence, the return types of such functions become unintelligible. Luckily, the **auto** return type declaration from the C++11 standard allows to simplify the code significantly, even more if used through the following preprocessor macro:

listing C.3 (C++)
```cpp
#define return_macro(expr) \
  -> decltype(safeToReturn(expr)) \
{ return safeToReturn(expr); }
```

For example, definition of a function returning its array-valued argument doubled, reads: **auto f(arr_t x) return_macro(2** $*$ **x)**. This is the only preprocessor macro defined herein. The call to **blitz :: safeToReturn**() function is included in order to ensure that all arrays involved in the returned expression continue to exist in the caller scope.

For the Python implementation of MPDATA, the NumPy[6] package is used. In order to make the code compatible with both the standard CPython as well as the alternative PyPy implementation of Python [5], the following sequence of **import** statements is used:

listing P.2 (Python)
```python
try:
  import numpypy
  from _numpypy.pypy import set_invalidation
  set_invalidation(False)
except ImportError:
  pass
import numpy
try:
  numpy.seterr(all='ignore')
except AttributeError:
  pass
```

First, the PyPy's built-in NumPy implementation named **numpypy** is imported if applicable (i.e. if running PyPy), and the lazy evaluation mode is turned on through the **set_invalidation**(**False**) call. PyPy's lazy evaluation obtained with the help of a just-in-time compiler enables to achieve an analogous to Blitz++ temporary-array-free handling of array-valued expressions (see discussion in Section 3). Second, to match the settings of C++ and Fortran compilers used herein, the NumPy package is instructed to ignore any floating-point errors, if such an option is available in the interpreter.[7] The above lines conclude all code modifications that needed to be added in order to run the code with PyPy.

Among the three considered languages only Fortran is equipped with built-in array handling facilities of practical use in high-performance computing. Therefore, there is no need for using an external package as with C++ and Python. Fortran array-handling features are not object-oriented, though (e.g. it is impossible to overload array operators or to provide custom constructor-like initialisation logic).

2.2. *Containers for sequences of arrays*

As discussed above, discretisation in space of the scalar field $\psi(x, y)$ into its $\psi_{[i,j]}$ grid representation requires floating-point array containers. In turn, discretisation in time requires a container class for storing sequences of such arrays, i.e. $\{\psi^{[n]}, \psi^{[n+1]}\}$. Similarly the components of the vector field $\vec{C}$ are in fact a $\{C^{[x]}, C^{[y]}\}$ array sequence.

Using an additional array dimension to represent the sequence elements is not considered for two reasons. First, the $C^{[x]}$ and $C^{[y]}$ arrays constituting the sequence have different sizes (see discussion of the Arakawa-C grid in Section 2.3). Second, the order of dimensions would need to be different for different languages to assure that the contiguous dimension is used for one of the space dimensions and not for time levels.

In the C++ implementation, the Boost[8] **ptr_vector** class is used to represent sequences of Blitz++ arrays and at the same time to handle automatic freeing of dynamically allocated memory. The **ptr_vector** class

[5]Blitz++ requires linking with **libblitz** if debug mode is used.

[6]NumPy is a Python package for scientific computing offering support for multi-dimensional arrays and a library of numerical algorithms, see http://numpy.org/.

[7]**numpy**.**seterr**() is not supported in PyPy as of version 1.9.

[8]Boost is a free and open-source collection of peer-reviewed C++ libraries available at http://boost.org/. Several parts of Boost have been integrated into or inspired new additions to the C++ standard.

is further customised by defining a derived structure with the element-access [ ] operator overloaded with a modulo variant:

listing C.4 (C++)
```cpp
#include <boost/ptr_container/ptr_vector.hpp>
struct arrvec_t : boost::ptr_vector<arr_t>
{
  const arr_t &operator[](const int i) const
  {
    return this->at((i + this->size()) % this->size());
  }
};
```

Consequently the last element of any such sequence may be accessed at index -1, the last but one at -2, and so on.

In the Python implementation, the built-in **tuple** type is used to store sequences of NumPy arrays. Employment of negative indices for handling from-the-end addressing of elements is a built-in feature of all sequence containers in Python.

Fortran does not feature any built-in sequence container capable of storing arrays, hence a custom **arrvec_t** type is introduced:

listing F.2 (Fortran)
```fortran
module arrvec_m
  use real_m
  implicit none

  type :: arr_t
    real(real_t), allocatable :: a(:,:)
  end type

  type :: arrptr_t
    class(arr_t), pointer :: p
  end type

  type :: arrvec_t
    class(arr_t), allocatable :: arrs(:)
    class(arrptr_t), allocatable :: at(:)
    integer :: length
    contains
    procedure :: ctor => arrvec_ctor
    procedure :: init => arrvec_init
  end type

  contains

  subroutine arrvec_ctor(this, n)
    class(arrvec_t) :: this
    integer, intent(in) :: n

    this%length = n
    allocate(this%at( -n : n-1 ))
    allocate(this%arrs( 0 : n-1 ))
  end subroutine

  subroutine arrvec_init(this, n, i, j)
    class(arrvec_t), target :: this
    integer, intent(in) :: n
    integer, intent(in) :: i(2), j(2)

    allocate(this%arrs(n)%a( i(1) : i(2), j(1) : j(2) ))
    this%at(n)%p => this%arrs(n)
    this%at(n - this%length)%p => this%arrs(n)
  end subroutine
end module
```

The **arr_t** type is defined solely for the purpose of overcoming the limitation of lack of an array-of-arrays construct, and its only member field is a two-dimensional array. An array of **arr_t** is used hereinafter as a container for sequences of arrays.

The **arrptr_t** type is defined solely for the purpose of overcoming Fortran's limitation of not supporting allocatables of pointers. The **arrptr_t**'s single member field is a pointer to an instance of **arr_t**. Creating an allocatable of **arrptr_t**, instead of a multi-element pointer of **arr_t**, ensures automatic memory deallocation.

Type **arrptr_t** is used to implement the from-the-end addressing of elements in **arrvec_t**. The array data is stored in the **arrs** member field (of type **arr_t**). The **at** member field (of type **arrptr_t**) stores pointers to the elements of **arrs**. It has double the length of **arrs** and is initialised in a cyclic manner so that the -1 element of **at** points to the last element of **arrs**, and so on. Assuming **psi** is an instance of **arrptr_t**, the $(\mathbf{i}, \mathbf{j})$ element of the **n**-th array in **psi** may be accessed with **psi%at(n)%p%a(i, j)**.

The **ctor(n)** method initialises the container for a given number of elements **n**. The **init(n, i, j)** method initialises the **n**-th element of the container with a newly allocated 2D array spanning indices **i(1):i(2)**, and **j(1):j(2)** in the first, and last dimensions respectively.[9]

2.3. Staggered grid

The so-called Arakawa-C staggered grid [3] depicted in Fig. 1 is a natural choice for MPDATA. As a consequence, the discretised representations of the

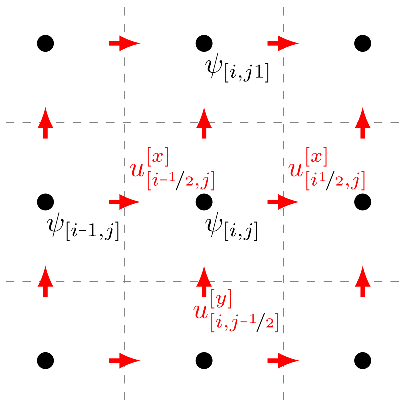

Fig. 1. A schematic of the Arakawa-C grid. (Colors are visible in the online version of the article; http://dx.doi.org/10.3233/SPR-140379.)

[9] In Fortran, when an array is passed as a function argument its base is locally set to unity, regardless of the setting at the caller scope.

ψ scalar field, and each component of the $\vec{C}$ vector field are defined over different grid point locations. In mathematical notation this can be indicated by usage of fractional indices, e.g. $C^{[x]}_{[i-1/2,j]}$, $C^{[x]}_{[i+1/2,j]}$, $C^{[y]}_{[i,j-1/2]}$ and $C^{[y]}_{[i,j+1/2]}$ to depict the grid values of the $\vec{C}$ vector components surrounding $\psi_{[i,j]}$. However, fractional indexing does not have a built-in counterpart in any of the employed programming languages. A desired syntax would translate $i-\frac{1}{2}$ to $i-1$ and $i+\frac{1}{2}$ to i. OOP offers a convenient way to implement such notation by overloading the $+$ and $-$ operators for objects representing array indices.

In the C++ implementation, first a global instance **h** of an empty structure **hlf_t** is defined, and then the plus and minus operators for **hlf_t** and **rng_t** are overloaded:

listing C.5 (C++)
```cpp
struct hlf_t {} h;

inline rng_t operator+(const rng_t &i, const hlf_t &)
{
  return i;
}

inline rng_t operator-(const rng_t &i, const hlf_t &)
{
  return i-1;
}
```

This way, the arrays representing vector field components can be indexed using $(\mathbf{i}+\mathbf{h},\mathbf{j})$, $(\mathbf{i}-\mathbf{h},\mathbf{j})$ etc., where **h** represents the half.

In NumPy, in order to prevent copying of array data during slicing, one needs to operate on the so-called array views. Array views are obtained when indexing the arrays with objects of the Python's built-it **slice** type (or tuples of such objects in case of multi-dimensional arrays). Python forbids overloading of operators of built-in types such as **slices**, and does not define addition/subtraction operators for **slice** and **int** pairs. Consequently, a custom logic has to be defined not only for fractional indexing, but also for shifting the slices by integer intervals ($i \pm 1$). It is implemented here by declaring a **shift** class with the adequate operator overloads:

listing P.3 (Python)
```python
class shift():
  def __init__(self, plus, mnus):
    self.plus = plus
    self.mnus = mnus
  def __radd__(self, arg):
    return type(arg)(
      arg.start + self.plus,
      arg.stop  + self.plus
    )
  def __rsub__(self, arg):
    return type(arg)(
      arg.start - self.mnus,
      arg.stop  - self.mnus
    )
```

and two instances of it to represent unity and half in expressions like $\mathbf{i}+\mathbf{one}$, $\mathbf{i}+\mathbf{hlf}$, where **i** is an instance of **slice**:[10]

listing P.4 (Python)
```python
one = shift(1,1)
hlf = shift(0,1)
```

In the Fortran implementation, fractional array indexing is obtained through definition and instantiation of an object representing the half, and having appropriate operator overloads:

listing F.3 (Fortran)
```fortran
module arakawa_c_m
  implicit none

  type :: half_t
  end type

  type(half_t) :: h

  interface operator (+)
    module procedure ph
  end interface

  interface operator (-)
    module procedure mh
  end interface

  contains

  elemental function ph(i, h) result (return)
    integer, intent(in) :: i
    type(half_t), intent(in) :: h
    integer :: return
    return = i
  end function

  elemental function mh(i, h) result (return)
    integer, intent(in) :: i
    type(half_t), intent(in) :: h
    integer :: return
    return = i - 1
  end function
end module
```

2.4. Array index permutations

Hereinafter, the $\pi^{d}_{a,b}$ symbol is used to denote a cyclic permutation of an order d of a set $\{a,b\}$. It is used to generalise the MPDATA formulae into multiple dimensions using the following notation:

$$\sum_{d=0}^{1} \psi_{[i,j]+\pi^{d}_{1,0}} \equiv \psi_{[i+1,j]} + \psi_{[i,j+1]}. \qquad (2)$$

Blitz++ ships with the **RectDomain** class (aliased here as **idx_t**) for specifying array ranges in multiple dimensions. The π permutation is implemented in C++ as a function **pi**() returning an instance of **idx_t**. In order to ensure compile-time evaluation, the permutation order is passed via the template parameter **d** (note the different order of **i** and **j** arguments in the two

[10]One could argue that not using an own implementation of a slice-representing class in NumPy is a design flaw – being able to modify behaviour of a hypothetical numpy.slice class through inheritance would allow to implement the same behaviour as obtained in listing P.3 without the need to represent the unity as a separate object.

template specialisations):

listing C.6 (C++)
```
template<int d>
inline idx_t pi(const rng_t &i, const rng_t &j);

template<>
inline idx_t pi<0>(const rng_t &i, const rng_t &j)
{
  return idx_t({i,j});
};

template<>
inline idx_t pi<1>(const rng_t &j, const rng_t &i)
{
  return idx_t({i,j});
};
```

NumPy uses tuples of slices for addressing multi-dimensional array with a single object. Therefore, the following definition of function **pi**() suffices to represent π:

listing P.5 (Python)
```
def pi(d, *idx):
  return (idx[d], idx[d-1])
```

Fortran does not feature an analogous mechanism for specifying array ranges in multiple dimensions using a single entity. As a workaround, in the Fortran implementation, **pi**() returns a pointer to the array elements specified by **i** and **j** interpreted as (i, j) or (j, i) depending on the value of the argument **d**. In addition to **pi**(), a helper **span**() function returning the length of one of the vectors passed as argument is defined:

listing F.4 (Fortran)
```
module pi_m
  use real_m
  implicit none
  contains
  function pi(d, arr, i, j) result(return)
    integer, intent(in) :: d
    real(real_t), allocatable, target :: arr(:,:)
    real(real_t), pointer :: return(:,:)
    integer, intent(in) :: i(2), j(2)
    select case (d)
      case (0)
        return => arr( i(1) : i(2), j(1) : j(2) )
      case (1)
        return => arr( j(1) : j(2), i(1) : i(2) )
    end select
  end function

  pure function span(d, i, j) result(return)
    integer, intent(in) :: i(2), j(2)
    integer, intent(in) :: d
    integer :: return
    select case (d)
      case (0)
        return = i(2) - i(1) + 1
      case (1)
        return = j(2) - j(1) + 1
    end select
  end function
end module
```

The **span**() function is used to shorten the declarations of arrays to be returned from functions in the Fortran implementation (see listings F.7 and F.12–F.15).

It is worth noting here that the C++ implementation of **pi**() is branchless thanks to employment of template specialisation. With Fortran one needs to rely on compiler optimisations to eliminate the conditional expression within **pi**() that depends on value of **d** which is always known at compile time.

2.5. *Donor-cell formulae*

MPDATA is an iterative algorithm in which each iteration takes the form of the so-called donor-cell formula (which itself is a first-order advection scheme).

MPDATA and donor-cell are explicit forward-in-time algorithms – they allow to predict $\psi^{[n+1]}$ as a function of $\psi^{[n]}$ where n and $n+1$ denote two adjacent time levels. The donor-cell scheme may be written as [27, Eq. (2)]:

$$\begin{aligned}\psi^{[n+1]}_{[i,j]} &= \psi^{[n]}_{[i,j]} \\ &\quad - \sum_{d=0}^{N-1} \Big(F\big[\psi^{[n]}_{[i,j]}, \psi^{[n]}_{[i,j]+\pi^d_{1,0}}, C^{[d]}_{[i,j]+\pi^d_{1/2,0}}\big] \\ &\quad - F\big[\psi^{[n]}_{[i,j]+\pi^d_{-1,0}}, \psi^{[n]}_{[i,j]}, C^{[d]}_{[i,j]+\pi^d_{-1/2,0}}\big]\Big), \end{aligned} \tag{3}$$

where N is the number of dimensions, and F is the so-called flux function [27, Eq. (3)]:

$$\begin{aligned} F(\psi_L, \psi_R, C) &= \max(C, 0) \cdot \psi_L + \min(C, 0) \cdot \psi_R \\ &= \frac{C + |C|}{2} \cdot \psi_L + \frac{C - |C|}{2} \cdot \psi_R. \end{aligned} \tag{4}$$

In C++, the flux function takes the following form:

listing C.7 (C++)
```
template<class T1, class T2, class T3>
inline auto F(
  const T1 &psi_l, const T2 &psi_r, const T3 &C
) return_macro(
  (
    (C + abs(C)) * psi_l +
    (C - abs(C)) * psi_r
  ) / 2
)
```

Equation (3) is split into the terms under the summation (effectively the 1-dimensional donor-cell formula):

listing C.8 (C++)
```
template<int d>
inline auto donorcell(
  const arr_t &psi, const arr_t &C,
  const rng_t &i, const rng_t &j
) return_macro(
  F(
    psi(pi<d>(i,   j)),
    psi(pi<d>(i+1, j)),
      C(pi<d>(i+h, j))
  ) -
  F(
    psi(pi<d>(i-1, j)),
    psi(pi<d>(i,   j)),
      C(pi<d>(i-h, j))
  )
)
```

and the actual two-dimensional donor-cell formula:

listing C.9 (C++)

```cpp
void donorcell_op(
  const arrvec_t &psi, const int n,
  const arrvec_t &C,
  const rng_t &i, const rng_t &j
) {
  psi[n+1](i,j) = psi[n](i,j) - (
    donorcell<0>(psi[n], C[0], i, j) +
    donorcell<1>(psi[n], C[1], j, i)
  );
}
```

In Python, the same formulae are expressed as follows:

listing P.6 (Python)

```python
def f(psi_l, psi_r, C):
  return (
    (C + abs(C)) * psi_l +
    (C - abs(C)) * psi_r
  ) / 2
```

listing P.7 (Python)

```python
def donorcell(d, psi, C, i, j):
  return (
    f(
      psi[pi(d, i,     j)],
      psi[pi(d, i+one, j)],
        C[pi(d, i+hlf, j)]
    ) -
    f(
      psi[pi(d, i-one, j)],
      psi[pi(d, i,     j)],
        C[pi(d, i-hlf, j)]
    )
  )
```

listing P.8 (Python)

```python
def donorcell_op(psi, n, C, i, j):
  psi[n+1][i,j] = psi[n][i,j] - (
    donorcell(0, psi[n], C[0], i, j) +
    donorcell(1, psi[n], C[1], j, i)
  )
```

The Fortran counterparts are:

listing F.5 (Fortran)

```fortran
module donorcell_m
  use real_m
  use arakawa_c_m
  use pi_m
  use arrvec_m
  implicit none
  contains
```

listing F.6 (Fortran)

```fortran
  elemental function F(psi_l, psi_r, C) result (return)
    real(real_t) :: return
    real(real_t), intent(in) :: psi_l, psi_r, C
    return = (                                          &
      (C + abs(C)) * psi_l +                            &
      (C - abs(C)) * psi_r                              &
    ) / 2
  end function
```

listing F.7 (Fortran)

```fortran
  function donorcell(d, psi, C, i, j) result (return)
    integer :: d
    integer, intent(in) :: i(2), j(2)
    real(real_t) :: return(span(d, i, j), span(d, j, i))
    real(real_t), allocatable, intent(in) :: psi(:,:), C(:,:)
    return = (                                          &
      F(                                                &
        pi(d, psi, i,   j),                             &
        pi(d, psi, i+1, j),                             &
        pi(d,   C, i+h, j)                              &
      ) -                                               &
      F(                                                &
        pi(d, psi, i-1, j),                             &
        pi(d, psi, i,   j),                             &
        pi(d, C,   i-h, j)                              &
      )                                                 &
    )
  end function
```

listing F.8 (Fortran)

```fortran
  subroutine donorcell_op(psi, n, C, i, j)
    class(arrvec_t), allocatable :: psi
    class(arrvec_t), pointer :: C
    integer, intent(in) :: n
    integer, intent(in) :: i(2), j(2)

    real(real_t), pointer :: ptr(:,:)
    ptr => pi(0, psi%at(n+1)%p%a, i, j)
    ptr = pi(0, psi%at(n)%p%a, i, j) - (                      &
      donorcell(0, psi%at(n)%p%a, C%at(0)%p%a, i, j) +  &
      donorcell(1, psi%at(n)%p%a, C%at(1)%p%a, j, i)    &
    )
  end subroutine
```

listing F.9 (Fortran)

```fortran
end module
```

The brevity of the code in the above listings as well as its similarity to the mathematical notation is the main point of this paper. The "formula translation" features include:

- loop-free notation;
- array-valued functions enabling reuse of sub-expressions;
- fractional indexing obtained with the help of operator overloading;
- dimension-independent indexing with the help of permutation functions.

The same features are applied to translation of more complex formulae in the following section.

2.6. MPDATA formulae

MPDATA introduces corrective steps to the algorithm defined by Eqs (3) and (4). Each corrective step has the form of a donor-cell pass, with the Courant number fields corresponding to the MPDATA antidiffusive velocities of the following form (Eqs (13), (14) in [27]):

$$
\begin{aligned}
&C'^{[d]}_{[i,j]+\pi^d_{1/2,0}} \\
&\quad = \left|C^{[d]}_{[i,j]+\pi^d_{1/2,0}}\right| \cdot \left[1 - \left|C^{[d]}_{[i,j]+\pi^d_{1/2,0}}\right|\right] \cdot A^{[d]}_{[i,j]}(\psi) \\
&\qquad - \sum_{q=0,q\neq d}^{N} C^{[d]}_{[i,j]+\pi^d_{1/2,0}} \cdot \overline{C}^{[q]}_{[i,j]+\pi^d_{1/2,0}} \cdot B^{[d]}_{[i,j]}(\psi),
\end{aligned}
\tag{5}
$$

where ψ and C represent values from the previous iteration and where:

$$
\begin{aligned}
&\overline{C}^{[q]}_{[i,j]+\pi^d_{1/2,0}} \\
&\quad = \frac{1}{4} \cdot \left(C^{[q]}_{[i,j]+\pi^d_{1,1/2}} + C^{[q]}_{[i,j]+\pi^d_{0,1/2}}\right. \\
&\qquad \left. + C^{[q]}_{[i,j]+\pi^d_{1,-1/2}} + C^{[q]}_{[i,j]+\pi^d_{0,-1/2}}\right).
\end{aligned}
\tag{6}
$$

For positive-definite ψ, the A and B terms take the following form:[11]

$$A^{[d]}_{[i,j]} = \frac{\psi_{[i,j]+\pi^d_{1,0}} - \psi_{[i,j]}}{\psi_{[i,j]+\pi^d_{1,0}} + \psi_{[i,j]}}, \tag{7}$$

$$\begin{aligned} B^{[d]}_{[i,j]} = \frac{1}{2}\big(&\psi_{[i,j]+\pi^d_{1,1}} + \psi_{[i,j]+\pi^d_{0,1}} - \psi_{[i,j]+\pi^d_{1,-1}} \\ &- \psi_{[i,j]+\pi^d_{0,-1}}\big)/\big(\psi_{[i,j]+\pi^d_{1,1}} + \psi_{[i,j]+\pi^d_{0,1}} \\ &+ \psi_{[i,j]+\pi^d_{1,-1}} + \psi_{[i,j]+\pi^d_{0,-1}}\big). \end{aligned} \tag{8}$$

If the (positive-defined) denominator in Eqs (7) or (8) equals zero for a given i and j, the corresponding $A_{[i,j]}$ and $B_{[i,j]}$ are set to zero. This may be conveniently represented with the **where** construct in all three considered languages:

listing C.10 (C++)

```
template<class nom_t, class den_t>
inline auto mpdata_frac(
  const nom_t &nom, const den_t &den
) return_macro(
  where(den > 0, nom / den, 0)
)
```

listing P.9 (Python)

```
def mpdata_frac(nom, den):
  return numpy.where(den > 0, nom/den, 0)
```

listing F.10 (Fortran)

```
module mpdata_m
  use arrvec_m
  use arakawa_c_m
  use pi_m
  implicit none
  contains
```

listing F.11 (Fortran)

```
  function mpdata_frac(nom, den) result (return)
    real(real_t), intent(in) :: nom(:,:), den(:,:)
    real(real_t) :: return(size(nom, 1), size(nom, 2))
    where (den > 0)
      return = nom / den
    elsewhere
      return = 0
    end where
  end function
```

The A term defined in Eq. (7) takes the following form:

listing C.11 (C++)

```
template<int d>
inline auto mpdata_A(const arr_t &psi,
  const rng_t &i, const rng_t &j
) return_macro(
  mpdata_frac(
    psi(pi<d>(i+1, j)) - psi(pi<d>(i,j)),
    psi(pi<d>(i+1, j)) + psi(pi<d>(i,j))
  )
)
```

listing P.10 (Python)

```
def mpdata_A(d, psi, i, j):
  return mpdata_frac(
    psi[pi(d, i+one, j)] - psi[pi(d, i, j)],
    psi[pi(d, i+one, j)] + psi[pi(d, i, j)]
  )
```

listing F.12 (Fortran)

```
  function mpdata_A(d, psi, i, j) result (return)
    integer :: d
    real(real_t), allocatable, intent(in) :: psi(:,:)
    integer, intent(in) :: i(2), j(2)
    real(real_t) :: return(span(d, i, j), span(d, j, i))
    return = mpdata_frac(                                  &
      pi(d, psi, i+1, j) - pi(d, psi, i, j),               &
      pi(d, psi, i+1, j) + pi(d, psi, i, j)                &
    )
  end function
```

The B term defined in Eq. (8) takes the following form:

listing C.12 (C++)

```
template<int d>
inline auto mpdata_B(const arr_t &psi,
  const rng_t &i, const rng_t &j
) return_macro(
 mpdata_frac(
    psi(pi<d>(i+1, j+1)) + psi(pi<d>(i, j+1)) -
    psi(pi<d>(i+1, j-1)) - psi(pi<d>(i, j-1)),
    psi(pi<d>(i+1, j+1)) + psi(pi<d>(i, j+1)) +
    psi(pi<d>(i+1, j-1)) + psi(pi<d>(i, j-1))
  ) / 2
)
```

listing P.11 (Python)

```
def mpdata_B(d, psi, i, j):
  return mpdata_frac(
    psi[pi(d, i+one, j+one)] + psi[pi(d, i, j+one)] -
    psi[pi(d, i+one, j-one)] - psi[pi(d, i, j-one)],
    psi[pi(d, i+one, j+one)] + psi[pi(d, i, j+one)] +
    psi[pi(d, i+one, j-one)] + psi[pi(d, i, j-one)]
  ) / 2
```

listing F.13 (Fortran)

```
  function mpdata_B(d, psi, i, j) result (return)
    integer :: d
    real(real_t), allocatable, intent(in) :: psi(:,:)
    integer, intent(in) :: i(2), j(2)
    real(real_t) :: return(span(d, i, j), span(d, j, i))
    return = mpdata_frac(                                  &
      pi(d, psi, i+1, j+1) + pi(d, psi, i,   j+1)          &
    - pi(d, psi, i+1, j-1) - pi(d, psi, i,   j-1),         &
      pi(d, psi, i+1, j+1) + pi(d, psi, i,   j+1)          &
    + pi(d, psi, i+1, j-1) + pi(d, psi, i,   j-1)          &
    ) / 2
  end function
```

Equation (6) takes the following form:

listing C.13 (C++)

```
template<int d>
inline auto mpdata_C_bar(
  const arr_t &C,
  const rng_t &i,
  const rng_t &j
) return_macro(
  (
    C(pi<d>(i+1, j+h)) + C(pi<d>(i, j+h)) +
    C(pi<d>(i+1, j-h)) + C(pi<d>(i, j-h))
  ) / 4
)
```

listing P.12 (Python)

```
def mpdata_C_bar(d, C, i, j):
  return (
    C[pi(d, i+one, j+hlf)] + C[pi(d, i,   j+hlf)] +
    C[pi(d, i+one, j-hlf)] + C[pi(d, i,   j-hlf)]
  ) / 4
```

listing F.14 (Fortran)

```
  function mpdata_C_bar(d, C, i, j) result (return)
    integer :: d
    real(real_t), allocatable, intent(in) :: C(:,:)
    integer, intent(in) :: i(2), j(2)
    real(real_t) :: return(span(d, i, j), span(d, j, i))

    return = (                                             &
      pi(d, C, i+1, j+h) + pi(d, C, i,   j+h) +            &
      pi(d, C, i+1, j-h) + pi(d, C, i,   j-h)              &
    ) / 4
  end function
```

[11] Since $\psi \geqslant 0$, $|A| \leqslant 1$ and $|B| \leqslant 1$. See [28, Section 4.2] for description of adaptation of the formulae for advection of fields of variable sign.

Equation (5) takes the following form:

listing C.14 (C++)
```cpp
template<int d>
inline auto mpdata_C_adf(
  const arr_t &psi,
  const rng_t &i, const rng_t &j,
  const arrvec_t &C
) return_macro(
  abs(C[d](pi<d>(i+h, j)))
  * (1 - abs(C[d](pi<d>(i+h, j))))
  * mpdata_A<d>(psi, i, j)
  - C[d](pi<d>(i+h, j))
  * mpdata_C_bar<d>(C[d-1], i, j)
  * mpdata_B<d>(psi, i, j)
)
```

listing P.13 (Python)
```python
def mpdata_C_adf(d, psi, i, j, C):
  return (
    abs(C[d][pi(d, i+hlf, j)])
    * (1 - abs(C[d][pi(d, i+hlf, j)]))
    * mpdata_A(d, psi, i, j)
    - C[d][pi(d, i+hlf, j)]
    * mpdata_C_bar(d, C[d-1], i, j)
    * mpdata_B(d, psi, i, j)
  )
```

listing F.15 (Fortran)
```fortran
  function mpdata_C_adf(d, psi, i, j, C) result (return)
    integer :: d
    integer, intent(in) :: i(2), j(2)
    real(real_t) :: return(span(d, i, j), span(d, j, i))
    real(real_t), allocatable, intent(in) :: psi(:,:)
    class(arrvec_t), pointer :: C
    return =                                          &
      abs(pi(d, C%at(d)%p%a, i+h, j))                 &
      * (1 - abs(pi(d, C%at(d)%p%a, i+h, j)))         &
      * mpdata_A(d, psi, i, j)                        &
      - pi(d, C%at(d)%p%a, i+h, j)                    &
      * mpdata_C_bar(d, C%at(d-1)%p%a, i, j)          &
      * mpdata_B(d, psi, i, j)
  end function
```

listing F.16 (Fortran)
```fortran
end module
```

The above listings conclude the formula-translation part of this paper. Implementation of a prototype MPDATA solver using the above code is presented in Appendix A.

3. Performance evaluation

3.1. Setup

The three introduced implementations of MPDATA were tested with the following setups employing free and open-source tools:

C++:

- GCC g++ 4.8.0[12] and Blitz++ 0.10
- LLVM Clang 3.2 and Blitz 0.10

Python:

- CPython 2.7.3 and NumPy 1.7
- PyPy 1.9.0 with built-in NumPy implementation

Fortran:

- GCC gfortran 4.8.0[12]

[12]GNU Compiler Collection packaged in the Debian's gcc-snapshot_20130222-1.

The performance tests were run on a Debian and an Ubuntu GNU/Linux systems with the above-listed software obtained via binary packages from the distributions' package repositories (most recent package versions at the time of writing). The tests were performed on two 64-bit machines equipped with an AMD Phenom™ II X6 1055T (800 MHz) and an Intel® Core™ i5-2467M (1.6 GHz) processors.

For both C++ and Fortran, the compilers were invoked with the **−Ofast** and the **−march** = **native** options. The CPython interpreter was invoked with the **−OO** option.

In addition to the standard Python implementation CPython, the Python code was tested with PyPy. PyPy is an alternative implementation of Python featuring a just-in-time compiler. PyPy includes an experimental partial re-implementation of NumPy that compiles NumPy expressions into native assembler. Thanks to employment of lazy evaluation of array expressions (cf. Section 2.1) PyPy allows to eliminate the use of temporary matrices for storing intermediate results, and to perform multiple operations on the arrays within a single array index traversal.[13] Consequently, PyPy allows to overcome the same performance-limiting factors as those addressed by Blitz++, although the underlying mechanisms are different. In contrast to other solutions for improving performance of NumPy-based codes such as Cython,[14] numexpr[15] or Numba,[16] PyPy does not require any modifications to the code. Thus, PyPy may serve as a drop-in replacement for CPython, ready to be used with previously-developed codes.

The same set of tests was run with all four setups. Each test set consisted of 16 program runs. The test programs are analogous to the example code presented in Appendix B. The tests were run with different grid sizes ranging from 64×64 to 2048×2048. The Gaussian impulse was advected for $nt = 2^{24}/(nx \cdot ny)$ timesteps, in order to assure comparable timing accuracy for all grid sizes (2^{24} chosen arbitrarily). Three MPDATA iterations were used (i.e. two corrective steps). The tests were run multiple times; program start-up, data loading, and output verification times were subtracted from the reported values (see caption of Fig. 3 for details).

[13]Lazy evaluation available in PyPy 1.9 has been temporarily removed from PyPy during a refactoring of the code. It'll be reinstantiated in the codebase as soon as possible, but past PyPy 2.0 release.

[14]See http://cython.org.

[15]See http://code.google.com/p/numexpr/.

[16]See http://numba.pydata.org/.

3.2. Results

Figure 2 presents a plot of the peak memory use[17] (identical for both considered CPUs) as a function of grid size. The plotted values are normalised by the nominal size of all data arrays used in the program (i.e. two $(nx+2)\times(ny+2)$ arrays representing the two time levels of ψ, a $(nx+1)\times(ny+2)$ array representing the $C^{[x]}$ component of the Courant number field, a $(nx+2)\times(ny+1)$ array representing the $C^{[y]}$ component, and two pairs of arrays of the size of $C^{[x]}$ and $C^{[y]}$ for storing the antidiffusive velocities, all composed of 8-byte double-precision floating point numbers). Plotted statistics reveal a notable memory footprint of the Python interpreter itself for both CPython and PyPy, losing its significance for domains larger than 1024×1024. The roughly asymptotic values reached in all four setups for grid sizes larger that 1024×1024 are indicative of the amount of temporary memory used for array manipulation. PyPy- and Blitz++-based setups consume notably less memory than Fortran and CPython. This confirms the effectiveness of the just-in-time compilation (PyPy) and the expression-template technique (Blitz++) for elimination of temporary storage during array operations.

The CPU time statistics presented in Figs 3 and 4 reveal minor differences between results obtained with the two different processors. Presented results lead to the following observations (where by referring to language names, only the results obtained with the herein considered program codes, and software/hardware configurations are meant):

- Fortran gives shortest execution times for any domain size;
- C++ execution times are less than twice those of Fortran for grids larger than 256×256;
- CPython requires from around 4 to almost 10 times more CPU time than Fortran depending on the grid size;
- PyPy execution times are in most cases closer to C++ than to CPython.

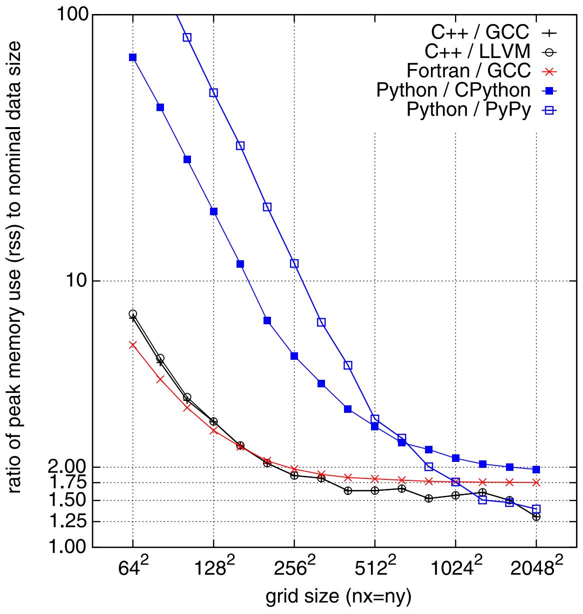

Fig. 2. Memory consumption statistics for the test runs described in Section 3 plotted as a function of grid size. Peak resident set size (rss) values are normalised by the size of data that needs to be allocated in the program to store all declared grid-sized arrays. Asymptotic values reached at the largest grid sizes are indicative of temporary storage requirements. (Colors are visible in the online version of the article; http://dx.doi.org/10.3233/SPR-140379.)

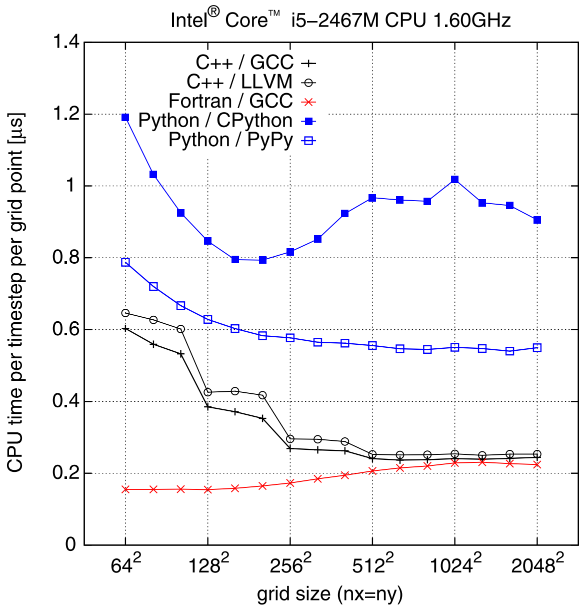

Fig. 3. Execution time statistics for the test runs described in Section 3 plotted as a function of grid size. Values of the total user mode CPU time are normalised by the grid size $(nx\cdot ny)$ and the number of timesteps $nt=2^{24}/(nx\cdot ny)$. The time reported for an $nt=0$ run for a corresponding domain size is subtracted from the values before normalisation. Both the $nt=0$ and $nt=2^{24}/(nx\cdot ny)$ runs are repeated three times and only the shortest time is taken into account. Results obtained with an Intel® Core™ i5 1.6 GHz processor. (Colors are visible in the online version of the article; http://dx.doi.org/10.3233/SPR-140379.)

[17]The resident set size (rss) as reported by the GNU time utility (version packaged in Debian as 1.7-24).

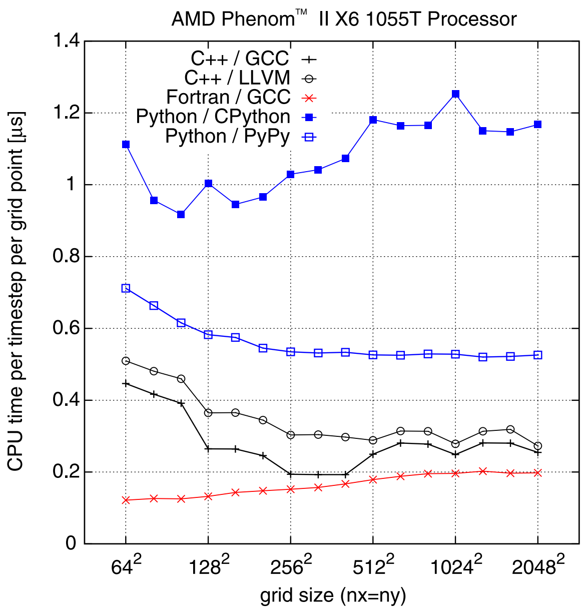

Fig. 4. Same as Fig. 3 for an AMD Phenom™ II 800 MHz processor. (Colors are visible in the online version of the article; http://dx.doi.org/10.3233/SPR-140379.)

The support for OOP features in gfortran, the NumPy support in PyPy, and the relevant optimisation mechanisms in GCC are still in active development and hence the performance with some of the setups may likely change with newer versions of these packages.

It is worth mentioning, that even though the three implementations are equally structured, the three considered languages have some inherent differences influencing the execution times. Notably, while Fortran and Blitz++ offer runtime array-bounds and array-shape checks as options not intended for use in production binaries, NumPy performs them always. Additionally, the C++ and Fortran setups may, in principle, benefit from auto-vectorisation features which do not yet have counterparts in CPython or PyPy. Finally, Fortran uses different ordering for storing array elements in memory, but since all tests were carried out using square grids, this should not have had any impact on the performance.[18]

4. The tradeoffs of language choice

The timing and memory usage statistics presented in Figs 2–4 reveal that, in the presented case, no single language/library/compiler setup corresponds to both shortest execution time and smallest memory footprint. Yet, performance is not the only criterion for the selection of a given language. Presented case study allows as well to assess other language characteristics that define the tradeoffs of language choice.

4.1. *Representability of blackboard abstractions*

It was shown in Section 2 that C++11/Blitz++, Python/NumPy and Fortran 2008 provide comparable functionality in terms of matching the blackboard abstractions within the program code. Taking into account solely the part of code representing particular formulae, for instance Eq. (5) and listings C.14, P.13, F.15, all three languages allow to match (or surpass) LaTeX in its brevity of formula translation syntax. All three languages were shown to be capable of providing mechanisms to compactly represent such abstractions as:

- loop-free array arithmetics;
- functions returning array-valued expressions;
- permutations of array indices allowing dimension-independent definitions of functions (see e.g. listings C.8 and C.9, P.7 and P.8, F.7 and F.8);
- fractional indexing of arrays corresponding to employment of a staggered grid.

Making use of features such as loop-free arithmetics not only shortens the code, but also enables the compiler or library authors to relieve the user (i.e. scientific programmer) from hand-coding optimisations (e.g. loop order choice). Hand-coded optimisations – code rearrangements aimed solely at the purpose of increasing performance – were long recognised as having *a strong negative impact when debugging and maintenance are considered* [15], and are generally advised to be avoided [21, Section 3.12].

Three issues specific to Fortran that resulted in employment of a more repetitive or cumbersome syntax than in C++ or Python were observed:

- Fortran lacks support for specifying array ranges in multiple dimensions with a single entity (cf. tuples of slices in NumPy and blitz::RectDomain);
- Fortran does not feature a mechanism allowing to reuse a single piece of code (algorithm) with different data types (compare e.g. listings C.15, P.14 and F.17) such as templates in C++ and the so-called "duck typing" in Python;

[18]Both Blitz++ and NumPy support Fortran's column-major ordering as well, however this feature is still missing from PyPy's built-in NumPy implementation as of PyPy 1.9.

- Fortran does not allow a function call to appear on the left-hand side of assignment (see e.g. how the **ptr** pointers were used as a workaround in the **cyclic_fill_halos** method in listing F.20);
- Fortran lacks support for arrays of arrays (cf. Section 2.2).

Interestingly, the limitation in extendability via inheritance was found to exist partially in NumPy as well (see Footnote 10). The lack of a counterpart in Fortran to the C++ template mechanism was identified in [7] as one of the key deficiencies of Fortran when compared with C++ in context of applicability to object-oriented scientific programming.

4.2. *Developers' community and libraries*

The size of the programmers' community of a given language influences the availability of: trained personnel, reusable software components and information resources. It also affects the maturity and quality of compilers and tools. Fortran is a domain-specific language while Python and C++ are general-purpose languages with disproportionately larger users' communities. The OOP features of Fortran have not gained wide popularity among users [38].[19] Fortran is no longer routinely taught at the university computer science departments [14], in contrast to C++ and Python. An example of decreasing popularity of Fortran in academia is the discontinuation of Fortran printed editions of the "Numerical Recipes" series of Press et al. (as of the third edition, the C++ version is the only one).

Blitz++ is one of several packages that offer high-performance object-oriented array manipulation functionality with C++ (and is not necessarily optimal for every purpose [13]). In contrast, the NumPy package became a de-facto standard solution for Python. Consequently, numerous Python libraries adopted NumPy but there are apparently very few C++ libraries offering Blitz++ support out of the box (the gnuplot-iostream used in listing C.20 being a much-appreciated counterexample). However, Blitz++ allows to interface with virtually any library (including Fortran libraries), by resorting to referencing the underlying memory with raw pointers.

The availability and maturity of libraries that offer object-oriented interfaces differ among the three considered languages. The built-in standard libraries of Python and C++ are richer than those of Fortran and offer versatile data types, collections of algorithms and facilities for interaction with host operating system. In the authors' experience, the small popularity of OOP techniques among Fortran users is reflected in the library designs (including the Fortran's built-in library routines). What makes correct use of external libraries less convenient with Fortran is the lack of standard exception handling mechanism, a feature long and *much requested by the numerical community* [24, Foreword].

The three languages differ as well with regard to availability of mechanisms (either built-in or available in external libraries) for handling concurrent computations. For instance, GCC supports OpenMP with Fortran and C++ what allows to easily leverage shared-memory parallelisation possibilities of multi-core processors. There is no equivalent built-in solution for multi-threading in CPython or PyPy. Fortan 2008 standard includes the "coarray" built-in parallel programming model for which counterparts are available as external libraries in case of C++ and Python. Implementations of the Message Passing Interface (MPI) for handling communication in distributed-memory setups are available for all three languages.

4.3. *Productivity, ease of use and misuse*

The factors influencing the development and maintenance time/cost are of particular importance in scientific computing [36]. Among the three compared environments, Python gains significantly if code length or coding time is prioritised (see also discussion in [17]). Python has already been the language of choice for scientific software projects having code clarity or ease of use as the first requirement (see e.g. [4]). PyPy's capability to improve performance of unmodified Python code may make Python a favourable choice even if high performance is important, especially if a combined measure of performance and development cost is to be considered.

Using the number of lines of code or the number of distinct language keywords needed to implement a given logic as measures of syntax brevity, Python clearly surpasses its rivals. Python was developed with emphasis on code readability and object-orientation. Arguably, taking it to the extreme – Python uses line indentation to define blocks of code and treats even a single integer as an object. As a consequence, Python is relatively easy to learn and easy to teach.

Fortran's lack of an exception mechanism poses a misuse risk when using both internal and external li-

[19] An anecdotal yet significant example being the incomplete support for syntax-highlighting of modern Fortran in Vim and Emacs editors (at the time of writing).

brary calls. The lack of exceptions results in a default policy to ignore recoverable errors. With no additional error-handling code, a Fortran program may silently continue after an error – additional code is needed to detect the error. In C++ and Python, such program will stop by default, while additional code may be introduced to recover from the error condition. Python does not feature such notorious mechanisms as the preprocessor in C++ and the implicit typing in Fortran, making it less prone to misuse.

Python implementations do not expose users to compilation or linking processes. As a result, Python-written software is easier to deploy and share, especially if multiple architectures and operating systems are targeted. However, there exist tools such as CMake[20] that allow to efficiently automate building, testing and packaging of C++ and Fortran programs.

It is worth noting one advantage of the C++/ Blitz++ setup. Blitz++ ensures temporary-array-free computations by design [34] avoiding unintentional performance loss. In contrast, with both Fortran and Python, the memory footprint caused by employment of temporary objects in array arithmetics is dependant on compiler choice or the level of optimisations.

Finally, Python is definitely easiest to debug among the three languages. Great debugging tools for C++ do exist, however the debugging and development is often hindered by indecipherable compiler messages flooded with lengthy type names stemming from employment of templates. Support for the OOP features of Fortran among compilers, debuggers and other programming aids remains immature at the time of writing.

5. Summary and outlook

Three implementations of a prototype solver for the advection equation were introduced. The solvers are based on MPDATA – an algorithm of particular applicability in geophysical fluid dynamics [28]. All implementations follow the same object-oriented structure but are implemented in three different languages (or language–library pairs):

- C++ with Blitz++;
- Python with NumPy;
- Fortran.

[20]CMake is a family of open-source, cross-platform tools automating building, testing and packaging of C/C++/Fortran software, see http://cmake.org/.

Presented programs were developed making use of such recent developments as support for C++11 and Fortran 2008 in GCC, and the NumPy support in the PyPy implementation of Python. The fact that all considered standards are open and the employed tools implementing them are free and open-source is certainly an advantage ([2], [33, Section 28.2.5]).

The key conclusion is that all considered language/ library/compiler setups offer possibilities for using OOP to compactly represent the mathematical abstractions within the program code. This creates the potential to improve code readability and brevity,

- contributing to its auditability, indispensable for credible and reproducible research in computational science [19,23,30]; and
- helping to keep the programs maintainable and avoiding accumulation of the code debt[21] that besets scientific software in such domains as climate modelling [11].

The performance evaluation revealed that:

- the Fortran setup offered shortest execution times,
- it took the C++ setup less than twice longer to compute than Fortran,
- C++ and PyPy setups offered significantly smaller memory consumption than Fortran and CPython for larger domains,
- the PyPy setup was roughly twice slower than C++ and up to twice faster than CPython.

The three equally-structured implementations required ca. 200, 300 and 500 lines of code in Python, C++ and Fortran, respectively. It is the authors' impression that these figures are somehow indicative of the programming effort.

In addition to the source code presented within the text, a set of tests and build-/test-automation scripts allowing to reproduce the analysis and plots presented in Section 3 are all available at the project repository,[22] and are released under the GNU GPL license [29]. The authors encourage to use the presented codes for teaching and benchmarking purposes.

The OOP design enhances the possibilities to reuse and extend the presented code. Development is underway of an object-oriented C++ library featuring concepts presented herein, supporting integration in one to three dimensions, handling systems of equations with source terms, providing miscellaneous op-

[21]See [6] for discussion of technical/code debt.

[22]git repository at http://github.com/igfuw/mpdata-oop/.

tions of MPDATA and several parallel processing approaches.[23]

Acknowledgements

We thank Piotr Smolarkiewicz and Hanna Pawłowska for their help throughout the project. This study was partly inspired by the lectures of Lech Łobocki.

Tobias Burnus, Julian Cummings, Ondřej Čertík, Patrik Jonsson, Arjen Markus, Zbigniew Piotrowski, Davide Del Vento and Janus Weil provided valuable feedback to the initial version of the manuscript and/or responses to questions posted to Blitz++ and gfortran mailing lists.

The final version of the manuscript benefited from suggestions of two anonymous reviewers.

SA, AJ and DJ acknowledge funding from the Polish National Science Centre (Project No. 2011/01/N/ST10/01483).

Part of the work was carried out during a visit of SA to the National Center for Atmospheric Research (NCAR) in Boulder, CO, USA. NCAR is operated by the University Corporation for Atmospheric Research. The visit was funded by the Foundation for Polish Science (START programme).

Development of NumPy support in PyPy was led by Alex Gaynor, Matti Picus and MF.

Appendix A. Prototype solvers

The following sections provide a complete description of a minimal example of application of the formulae "translated" into C++, Python and Fortran in the main body of the paper.

A.1. Halo regions

The MPDATA formulae defining $\psi^{[n+1]}_{[i,j]}$ as a function of $\psi^{[n]}_{[i,j]}$ (discussed in the following sections) feature terms such as $\psi_{[i-1,j-1]}$. One way of assuring validity of these formulae on the edges of the domain (e.g. for $i = 0$) is to introduce the so-called halo region surrounding the domain. The method of populating the halo region with data depends on the boundary condition type. Employment of the halo-region logic implies repeated usage of array range extensions in the code such as $i \rightsquigarrow i \pm \mathit{halo}$.

An **ext**() function is defined in all three implementation, in order to simplify coding of array range extensions:

listing C.15 (C++)
```cpp
template<class n_t>
inline rng_t ext(const rng_t &r, const n_t &n) {
  return rng_t(
    (r - n).first(),
    (r + n).last()
  );
}
```

listing P.14 (Python)
```python
def ext(r, n):
  if (type(n) == int) & (n == 1):
    n = one
  return slice(
    (r - n).start,
    (r + n).stop
  )
```

listing F.17 (Fortran)
```fortran
module halo_m
  use arakawa_c_m
  implicit none

  interface ext
    module procedure ext_n
    module procedure ext_h
  end interface

  contains

  function ext_n(r, n) result (return)
    integer, intent(in) :: r(2)
    integer, intent(in) :: n
    integer :: return(2)

    return = (/ r(1) - n, r(2) + n /)
  end function

  function ext_h(r, h) result (return)
    integer, intent(in) :: r(2)
    type(half_t), intent(in) :: h
    integer :: return(2)

    return = (/ r(1) - h, r(2) + h /)
  end function
end module
```

Consequently, a range depicted by $i \pm 1/2$ may be expressed in the code as **ext(i, h)**. In all three implementations, the **ext**() function accept the second argument to be an integer or a "half" (cf. Section 2.3).

A.2. Prototype solver

The tasks to be handled by a prototype advection equation solver proposed herein are:

(i) storing arrays representing the ψ and $\vec{C}$ fields and any required housekeeping data,
(ii) allocating/deallocating the required memory,
(iii) providing access to the solver state,
(iv) performing the integration.

In the following C++ definition of the **solver** structure, task (i) is represented with the definition of the structure member fields; task (ii) is split between the **solver**'s constructor and the destructors of **arrvec_t**; task (iii) is handled by the accessor methods; task (iv)

[23]Git repository at http://github.com/igfuw/libmpdataxx.

is handled within the **solve**() method:

listing C.16 (C++)
```cpp
template<class bcx_t, class bcy_t>
struct solver
{
  // member fields
  arrvec_t psi, C;
  int n, hlo;
  rng_t i, j;
  bcx_t bcx;
  bcy_t bcy;

  // ctor
  solver(int nx, int ny, int hlo) :
    hlo(hlo),
    n(0),
    i(0, nx-1),
    j(0, ny-1),
    bcx(i, j, hlo),
    bcy(j, i, hlo)
  {
    for (int l = 0; l < 2; ++l)
      psi.push_back(new arr_t(ext(i, hlo), ext(j, hlo)));
    C.push_back(new arr_t(ext(i, h), ext(j, hlo)));
    C.push_back(new arr_t(ext(i, hlo), ext(j, h)));
  }

  // accessor methods
  arr_t state() {
    return psi[n](i,j).reindex({0,0});
  }

  arr_t courant(int d)
  {
    return C[d];
  }

  // helper methods invoked by solve()
  virtual void advop() = 0;

  void cycle()
  {
    n = (n + 1) % 2 - 2;
  }

  // integration logic
  void solve(const int nt)
  {
    for (int t = 0; t < nt; ++t)
    {
      bcx.fill_halos(psi[n], ext(j, hlo));
      bcy.fill_halos(psi[n], ext(i, hlo));
      advop();
      cycle();
    }
  }
};
```

The **solver** structure is an abstract definition (containing a pure virtual method) requiring its descendants to implement at least the **advop**() method which is expected to fill **psi**[**n** + **1**] with an updated (advected) values of **psi**[**n**]. The two template parameters **bcx_t** and **bcy_t** allow the solver to operate with any kind of boundary condition structures that fulfil the requirements implied by the calls to the methods of **bcx** and **bcy**, respectively.

The donor-cell and MPDATA schemes both require only the previous state of an advected field in order to advance the solution. Consequently, memory for two time levels ($\psi^{[n]}$ and $\psi^{[n+1]}$) is allocated in the constructor. The sizes of the arrays representing the two time levels of ψ are defined by the domain size ($nx \times ny$) plus the halo region. The size of the halo region is an argument of the constructor. The **cycle**() method is used to swap the time levels without copying any data.

The arrays representing the $C^{[x]}$ and $C^{[y]}$ components of $\vec{C}$, require $(nx+1) \times ny$ and $nx \times (ny+1)$ elements, respectively (being laid out on the Arakawa-C staggered grid).

Python definition of the **solver** class follows closely the C++ structure definition:

listing P.15 (Python)
```python
class solver(object):
  # ctor-like method
  def __init__(self, bcx, bcy, nx, ny, hlo):
    self.n = 0
    self.hlo = hlo
    self.i = slice(hlo, nx + hlo)
    self.j = slice(hlo, ny + hlo)

    self.bcx = bcx(0, self.i, hlo)
    self.bcy = bcy(1, self.j, hlo)

    self.psi = (
      numpy.empty((
        ext(self.i, self.hlo).stop,
        ext(self.j, self.hlo).stop
      ), real_t),
      numpy.empty((
        ext(self.i, self.hlo).stop,
        ext(self.j, self.hlo).stop
      ), real_t)
    )

    self.C = (
      numpy.empty((
        ext(self.i, hlf).stop,
        ext(self.j, self.hlo).stop
      ), real_t),
      numpy.empty((
        ext(self.i, self.hlo).stop,
        ext(self.j, hlf).stop
      ), real_t)
    )

  # accessor methods
  def state(self):
    return self.psi[self.n][self.i, self.j]

  # helper methods invoked by solve()
  def courant(self,d):
    return self.C[d][:]

  def cycle(self):
    self.n  = (self.n + 1) % 2 - 2

   # integration logic
  def solve(self, nt):
    for t in range(nt):
      self.bcx.fill_halos(
        self.psi[self.n], ext(self.j, self.hlo)
      )
      self.bcy.fill_halos(
        self.psi[self.n], ext(self.i, self.hlo)
      )
      self.advop()
      self.cycle()
```

The key difference stems from the fact that, unlike Blitz++, NumPy does not allow an array to have arbitrary index base – in NumPy the first element is always addressed with 0. Consequently, while in C++ (and Fortran) the computational domain is chosen to start at ($i = 0$, $j = 0$) and hence a part of the halo region to have negative indices, in Python the halo region starts at $(0, 0)$.[24] However, since the whole halo logic is hid-

[24] The reason to allow the domain to begin at an arbitrary index is mainly to ease debugging in case the code would be used in parallel

den within the solver, such details are not exposed to the user. The **bcx** and **bcy** boundary-condition specifications are passed to the solver through constructor-like **__init__**() method as opposed to template parameters in C++.

The above C++ and Python prototype solvers, in principle, allow to operate with any boundary condition objects that implement methods called from within the solver. This requirement is checked at compile-time in the case of C++, and at run-time in the case of Python. In order to obtain an analogous behaviour with Fortran, it is required to define, prior to definition of a solver type, an abstract type with deferred procedures having abstract interfaces (sic!, see Table 2.1 in [26], for a summary of approximate correspondence of OOP nomenclature between Fortran and C++):

listing F.18 (Fortran)

```fortran
module bcd_m
  use arrvec_m
  implicit none

  type, abstract :: bcd_t
    contains
    procedure(bcd_fill_halos), deferred :: fill_halos
    procedure(bcd_init), deferred :: init
  end type

  abstract interface
    subroutine bcd_fill_halos(this, a, j)
      import :: bcd_t, real_t
      class(bcd_t ) :: this
      real(real_t), allocatable :: a(:,:)
      integer :: j(2)
    end subroutine

    subroutine bcd_init(this, d, n, hlo)
      import :: bcd_t
      class(bcd_t) :: this
      integer :: d, n, hlo
    end subroutine
  end interface
end module
```

Having defined the abstract type for boundary-condition objects, a definition of a solver class following closely the C++ and Python counterparts may be provided:

listing F.19 (Fortran)

```fortran
module solver_m
  use arrvec_m
  use bcd_m
  use arakawa_c_m
  use halo_m
  implicit none

  type, abstract :: solver_t
    class(arrvec_t), allocatable :: psi, C
    integer :: n, hlo
    integer :: i(2), j(2)
    class(bcd_t), pointer :: bcx, bcy
    contains
    procedure :: solve   => solver_solve
    procedure :: state   => solver_state
    procedure :: courant => solver_courant
    procedure :: cycle   => solver_cycle
    procedure(solver_advop), deferred :: advop
  end type

  abstract interface
    subroutine solver_advop(this)
      import solver_t
      class(solver_t), target :: this
    end subroutine
  end interface

  contains

  subroutine solver_ctor(this, bcx, bcy, nx, ny, hlo)
    use arakawa_c_m
    use halo_m
    class(solver_t) :: this
    class(bcd_t), intent(in), target :: bcx, bcy
    integer, intent(in) :: nx, ny, hlo

    this%n = 0
    this%hlo = hlo
    this%bcx => bcx
    this%bcy => bcy

    this%i = (/ 0, nx - 1 /)
    this%j = (/ 0, ny - 1 /)

    call bcx%init(0, nx, hlo)
    call bcy%init(1, ny, hlo)

    allocate(this%psi)
    call this%psi%ctor(2)
    block
      integer :: n
      do n=0, 1
        call this%psi%init(                                 &
          n, ext(this%i, hlo), ext(this%j, hlo)             &
        )
      end do
    end block

    allocate(this%C)
    call this%C%ctor(2)
    call this%C%init(                                       &
      0, ext(this%i, h), ext(this%j, hlo)                   &
    )
    call this%C%init(                                       &
      1, ext(this%i, hlo), ext(this%j, h)                   &
    )
  end subroutine

  function solver_state(this) result (return)
    class(solver_t) :: this
    real(real_t), pointer :: return(:,:)
    return => this%psi%at(this%n)%p%a(                      &
      this%i(1) : this%i(2),                                &
      this%j(1) : this%j(2)                                 &
    )
  end function

  function solver_courant(this, d) result (return)
    class(solver_t) :: this
    integer :: d
    real(real_t), pointer :: return(:,:)
    return => this%C%at(d)%p%a
  end function

  subroutine solver_cycle(this)
    class(solver_t) :: this
    this%n = mod(this%n + 1 + 2, 2) - 2
  end subroutine

  subroutine solver_solve(this, nt)
    class(solver_t) :: this
    integer, intent(in) :: nt
    integer :: t

    do t = 0, nt-1
      call this%bcx%fill_halos(                             &
        this%psi%at(this%n)%p%a, ext(this%j, this%hlo)      &
      )
      call this%bcy%fill_halos(                             &
        this%psi%at(this%n)%p%a, ext(this%i, this%hlo)      &
      )
      call this%advop()
      call this%cycle()
    end do
  end subroutine
end module
```

computations using domain decomposition where each subdomain could have its own index base corresponding to the location within the computational domain.

A.3. Periodic boundaries

The solver definition described in Section A.2 requires a given boundary condition object to implement a **fill_halos**() method. An implementation of periodic boundary conditions in C++ is provided in the following listing:

listing C.17 (C++)
```
template<int d>
struct cyclic
{
  // member fields
  rng_t left_halo, rght_halo;
  rng_t left_edge, rght_edge;;

  // ctor
  cyclic(
    const rng_t &i, const rng_t &j, int hlo
  ) :
    left_halo(i.first()-hlo, i.first()-1),
    rght_edge(i.last()-hlo+1, i.last()  ),
    rght_halo(i.last()+1, i.last()+hlo  ),
    left_edge(i.first(), i.first()+hlo-1)
  {}

  // method invoked by the solver
  void fill_halos(const arr_t &a, const rng_t &j)
  {
    a(pi<d>(left_halo, j)) = a(pi<d>(rght_edge, j));
    a(pi<d>(rght_halo, j)) = a(pi<d>(left_edge, j));
  }
};
```

As hinted by the member field names, the **fill_halos**() methods fill the left/right halo regions with data from the right/left edges of the domain. Thanks to employment of the function **pi**() described in Section 2.4 the same code may be applied in any dimension (the dimension being a template parameter).

The following listings contain the Python and Fortran counterparts to listing C.17.

listing P.16 (Python)
```python
class cyclic(object):
  # ctor
  def __init__(self, d, i, hlo):
    self.d = d
    self.left_halo = slice(i.start-hlo, i.start    )
    self.rght_edge = slice(i.stop -hlo, i.stop     )
    self.rght_halo = slice(i.stop,      i.stop +hlo)
    self.left_edge = slice(i.start,     i.start+hlo)

  # method invoked by the solver
  def fill_halos(self, psi, j):
    psi[pi(self.d, self.left_halo, j)] = (
      psi[pi(self.d, self.rght_edge, j)]
    )
    psi[pi(self.d, self.rght_halo, j)] = (
      psi[pi(self.d, self.left_edge, j)]
    )

```

listing F.20 (Fortran)
```fortran
module cyclic_m
  use bcd_m
  use pi_m
  implicit none

  type, extends(bcd_t) :: cyclic_t
    integer :: d
    integer :: left_halo(2), rght_halo(2)
    integer :: left_edge(2), rght_edge(2)
    contains
    procedure :: init => cyclic_init
    procedure :: fill_halos => cyclic_fill_halos
  end type

  contains

  subroutine cyclic_init(this, d, n, hlo)
    class(cyclic_t) :: this
    integer :: d, n, hlo

    this%d = d
    this%left_halo = (/ -hlo, -1 /)
    this%rght_halo = (/ n, n-1+hlo /)
    this%left_edge = (/ 0, hlo-1 /)
    this%rght_edge = (/ n-hlo, n-1 /)
  end subroutine

  subroutine cyclic_fill_halos(this, a, j)
    class(cyclic_t) :: this
    real(real_t), pointer :: ptr(:,:)
    real(real_t), allocatable :: a(:,:)
    integer :: j(2)
    ptr => pi(this%d, a, this%left_halo, j)
    ptr =  pi(this%d, a, this%rght_edge, j)
    ptr => pi(this%d, a, this%rght_halo, j)
    ptr =  pi(this%d, a, this%left_edge, j)
  end subroutine
end module
```

A.4. Donor-cell solver

As mentioned in the previous section, the donor-cell formula constitutes an advection scheme, hence we may use it to create a **solver_donorcell** implementation of the abstract **solver** class:

listing C.18 (C++)
```
template<class bcx_t, class bcy_t>
struct solver_donorcell : solver<bcx_t, bcy_t>
{
  solver_donorcell(int nx, int ny) :
    solver<bcx_t, bcy_t>(nx, ny, 1)
  {}

  void advop()
  {
    donorcell_op(
      this->psi, this->n, this->C,
      this->i, this->j
    );
  }
};
```

The above definition is given as an example only. In the following sections, an MPDATA solver with the same interface is defined.

The following listings contain the Python and Fortran counterparts to listing C.18.

listing P.17 (Python)
```python
class solver_donorcell(solver):
  def __init__(self, bcx, bcy, nx, ny):
    solver.__init__(self, bcx, bcy, nx, ny, 1)

  def advop(self):
    donorcell_op(
      self.psi, self.n,
      self.C, self.i, self.j
    )
```

listing F.21 (Fortran)
```fortran
module solver_donorcell_m
  use donorcell_m
  use solver_m
  implicit none

  type, extends(solver_t) :: donorcell_t
    contains
    procedure :: ctor => donorcell_ctor
    procedure :: advop => donorcell_advop
  end type

  contains

  subroutine donorcell_ctor(this, bcx, bcy, nx, ny)
    class(donorcell_t) :: this
    class(bcd_t), intent(in), target :: bcx, bcy
    integer, intent(in) :: nx, ny
    call solver_ctor(this, bcx,bcy, nx,ny, 1)
  end subroutine

  subroutine donorcell_advop(this)
    class(donorcell_t), target :: this
    class(arrvec_t), pointer :: C
    C => this%C
    call donorcell_op(                                 &
      this%psi, this%n, C, this%i, this%j              &
    )
  end subroutine
end module
```

A.5. MPDATA solver

An MPDATA solver may be now constructed by inheriting from the **solver** class with the following definition in C++:

listing C.19 (C++)

```cpp
template<int n_iters, class bcx_t, class bcy_t>
struct solver_mpdata : solver<bcx_t, bcy_t>
{
  // member fields
  static const int n_tmp = n_iters > 2 ? 2 : 1;
  arrvec_t tmp[n_tmp];
  rng_t im, jm;

  // ctor
  solver_mpdata(int nx, int ny) :
    solver<bcx_t, bcy_t>(nx, ny, 1),
    im(this->i.first() - 1, this->i.last()),
    jm(this->j.first() - 1, this->j.last())
  {
    for (int n = 0; n < n_tmp; ++n)
    {
      tmp[n].push_back(new arr_t(
        this->C[0].domain()[0], this->C[0].domain()[1])
      );
      tmp[n].push_back(new arr_t(
        this->C[1].domain()[0], this->C[1].domain()[1])
      );
    }
  }

  // method invoked by the solver
  void advop()
  {
    for (int step = 0; step < n_iters; ++step)
    {
      if (step == 0)
        donorcell_op(
          this->psi, this->n, this->C, this->i, this->j
        );
      else
      {
        this->cycle();
        this->bcx.fill_halos(
          this->psi[this->n], ext(this->j, this->hlo)
        );
        this->bcy.fill_halos(
          this->psi[this->n], ext(this->i, this->hlo)
        );

        // choosing input/output for antidiff C
        const arrvec_t
          &C_unco = (step == 1)
            ? this->C
            : (step % 2)
              ? tmp[1]  // odd steps
              : tmp[0], // even steps
          &C_corr = (step  % 2)
            ? tmp[0]    // odd steps
            : tmp[1];   // even steps

        // calculating the antidiffusive C
        C_corr[0](im+h, this->j) = mpdata_C_adf<0>(
          this->psi[this->n], im, this->j, C_unco
        );
        this->bcy.fill_halos(C_corr[0], ext(this->i,h));

        C_corr[1](this->i, jm+h) = mpdata_C_adf<1>(
          this->psi[this->n], jm, this->i, C_unco
        );
        this->bcx.fill_halos(C_corr[1], ext(this->j,h));

        // donor-cell step
        donorcell_op(
          this->psi, this->n, C_corr, this->i, this->j
        );
      }
    }
  }
};
```

The array of sequences of temporary arrays **tmp** allocated in the constructor is used to store the antidiffusive velocities from the present and optionally previous timestep (if using more than two iterations).

The **advop**() method controls the MPDATA iterations within one timestep. The first (step = 0 iteration) of MPDATA is an unmodified donor-cell step. Subsequent iterations begin with calculation of the antidiffusive Courant fields using formula (5). In order to calculate values spanning an $(i-\frac{1}{2},\ldots,i+\frac{1}{2})$ range using a formula for $C_{[i+1/2,\ldots]}$ only, the formula is evaluated using extended index ranges **im** and **jm**. In the second (step = 1 iteration), the uncorrected Courant field (**C_unco**) points to the original **C** field, and the antidiffusive Courant field is written into **C_corr** which points to **tmp[1]**. In the third (step = 2) iteration **C_unco** points to **tmp[1]** while **C_corr** points to **tmp[0]**. In subsequent iterations **tmp[0]** and **tmp[1]** are alternately swapped.

The following listings contain the Python and Fortran counterparts to listing C.19.

listing P.18 (Python)

```python
class solver_mpdata(solver):
  def __init__(self, n_iters, bcx, bcy, nx, ny):
    solver.__init__(self, bcx, bcy, nx, ny, 1)
    self.im = slice(self.i.start-1, self.i.stop)
    self.jm = slice(self.j.start-1, self.j.stop)

    self.n_iters = n_iters

    self.tmp = [(
      numpy.empty(self.C[0].shape, real_t),
      numpy.empty(self.C[1].shape, real_t)
    )]
    if n_iters > 2:
      self.tmp.append((
        numpy.empty(self.C[0].shape, real_t),
        numpy.empty(self.C[1].shape, real_t)
      ))

  def advop(self):
    for step in range(self.n_iters):
      if step == 0:
        donorcell_op(
          self.psi, self.n, self.C, self.i, self.j
        )
      else:
        self.cycle()
        self.bcx.fill_halos(
          self.psi[self.n], ext(self.j, self.hlo)
        )
        self.bcy.fill_halos(
          self.psi[self.n], ext(self.i, self.hlo)
        )
        if step == 1:
          C_unco, C_corr = self.C, self.tmp[0]
        elif step % 2:
          C_unco, C_corr = self.tmp[1], self.tmp[0]
        else:
          C_unco, C_corr = self.tmp[0], self.tmp[1]

        C_corr[0][self.im+hlf, self.j] = mpdata_C_adf(
          0, self.psi[self.n], self.im, self.j, C_unco
        )
    self.bcy.fill_halos(C_corr[0], ext(self.i, hlf))

    C_corr[1][self.i, self.jm+hlf] = mpdata_C_adf(
      1, self.psi[self.n], self.jm, self.i, C_unco
    )
    self.bcx.fill_halos(C_corr[1], ext(self.j, hlf))

    donorcell_op(
      self.psi, self.n, C_corr, self.i, self.j
    )
```

listing F.22 (Fortran)

```
module solver_mpdata_m
  use solver_m
  use mpdata_m
  use donorcell_m
  use halo_m
  implicit none

  type, extends(solver_t) :: mpdata_t
    integer :: n_iters, n_tmp
    integer :: im(2), jm(2)
    class(arrvec_t), pointer :: tmp(:)
    contains
    procedure :: ctor => mpdata_ctor
    procedure :: advop => mpdata_advop
  end type

  contains

  subroutine mpdata_ctor(this, n_iters, bcx, bcy, nx, ny)
    class(mpdata_t) :: this
    class(bcd_t), target :: bcx, bcy
    integer, intent(in) :: n_iters, nx, ny
    integer :: c

    call solver_ctor(this, bcx, bcy, nx, ny, 1)

    this%n_iters = n_iters
    this%n_tmp = min(n_iters - 1, 2)
    if (n_iters > 0) allocate(this%tmp(0:this%n_tmp))

    associate (i => this%i, j => this%j, hlo => this%hlo)
      do c=0, this%n_tmp - 1
        call this%tmp(c)%ctor(2)
        call this%tmp(c)%init(0, ext(i, h), ext(j, hlo))
        call this%tmp(c)%init(1, ext(i, hlo), ext(j, h))
      end do

      this%im = (/ i(1) - 1, i(2) /)
      this%jm = (/ j(1) - 1, j(2) /)
    end associate
  end subroutine

  subroutine mpdata_advop(this)
    class(mpdata_t), target :: this
    integer :: step

    associate (i => this%i, j => this%j, im => this%im,&
      jm => this%jm, psi => this%psi, n => this%n,      &
      hlo => this%hlo, bcx => this%bcx, bcy => this%bcy&
    )
      do step=0, this%n_iters-1
        if (step == 0) then
          block
            class(arrvec_t), pointer :: C
            C => this%C
            call donorcell_op(psi, n, C, i, j)
          end block
        else
          call this%cycle()
          call bcx%fill_halos(                             &
            psi%at( n )%p%a, ext(j, hlo)                   &
          )
          call bcy%fill_halos(                             &
            psi%at( n )%p%a, ext(i, hlo)                   &
          )

          block
            class(arrvec_t), pointer :: C_corr, C_unco
            real(real_t), pointer :: ptr(:,:)

            ! chosing input/output for antidiff. C
            if (step == 1) then
              C_unco => this%C
              C_corr => this%tmp(0)
            else if (mod(step, 2) == 1) then
              C_unco => this%tmp(1) ! odd step
              C_corr => this%tmp(0) ! even step
            else
              C_unco => this%tmp(0) ! odd step
              C_corr => this%tmp(1) ! even step
            end if

            ! calculating the antidiffusive velo
            ptr => pi(0, C_corr%at( 0 )%p%a, im+h, j)
            ptr = mpdata_C_adf(                            &
              0, psi%at( n )%p%a, im, j, C_unco            &
            )
            call bcy%fill_halos(                           &
              C_corr%at(0)%p%a, ext(i, h)                  &
            )

            ptr => pi(0, C_corr%at( 1 )%p%a, i, jm+h)
            ptr = mpdata_C_adf(                            &
              1, psi%at( n )%p%a, jm, i, C_unco            &
            )
            call bcx%fill_halos(                           &
              C_corr%at(1)%p%a, ext(j, h)                  &
            )

            ! donor-cell step
            call donorcell_op(psi, n, C_corr, i, j)
          end block
        end if
      end do
    end associate
  end subroutine
end module
```

Appendix B. Usage example

The following listing provides an example of how the MPDATA solver defined in Section A.5 may be used together with the cyclic boundary conditions defined in Section A.3. In the example, a Gaussian signal is advected in a 2D domain defined over a grid of 24×24 cells. The program first plots the initial condition, then performs the integration for 75 timesteps with three different settings of the number of iterations used in MPDATA. The velocity field is constant in time and space (although it is not assumed in the presented implementations). The signal shape at the end of each simulation is plotted as well. Plotting is done with the help of the gnuplot-iostream library.[25]

The resultant plot is presented herein as Fig. 5. The top panel depicts the initial condition. The three other panels show a snapshot of the field after 75 timesteps. The donor-cell solution is characterised by strongest numerical diffusion resulting in significant drop in the signal amplitude. The signals advected using MPDATA show smaller numerical diffusion with the solution obtained with more iterations preserving the signal altitude more accurately. In all of the simulations the signal maintains its positive definiteness. The domain periodicity is apparent in the plots as the maxi-

[25]gnuplot-iostream is a header-only C++ library allowing gnuplot to be controlled from C++, see http://stahlke.org/dan/gnuplot-iostream/. Gnuplot is a portable command-line driven graphing utility, see http://gnuplot.info/.

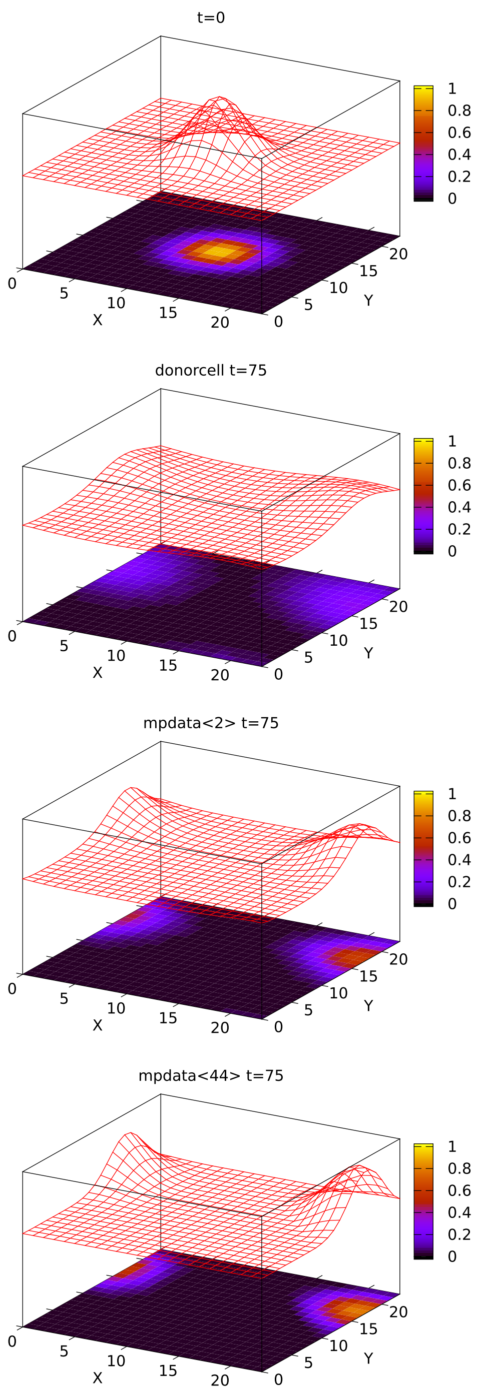

Fig. 5. Plot generated by the program given in listing C.20. The top panel shows initial signal shape (at time $t = 0$). The subsequent panels show snapshots of the advected field after 75 timesteps from three different simulations: donorcell (or 1 MPDATA iteration), MPDATA with two iterations and MPDATA with 44 iterations. The colour scale and the wire-frame surface correspond to signal amplitude. See Appendix B for discussion. (Colors are visible in the online version of the article; http://dx.doi.org/10.3233/SPR-140379.)

mum of the signal after 75 timesteps is located near the domain walls.

listing C.20 (C++)

```cpp
#include "listings.hpp"
#define GNUPLOT_ENABLE_BLITZ
#include <gnuplot-iostream/gnuplot-iostream.h>

enum {x, y};

template <class T>
void setup(T &solver, int n[2])
{
  blitz::firstIndex i;
  blitz::secondIndex j;
  solver.state() = exp(
    -sqr((.5+i)-n[x]/2.) / (2*pow(n[x]/10., 2))
    -sqr((.5+j)-n[y]/2.) / (2*pow(n[y]/10., 2))
  );
  solver.courant(x) = -.5;
  solver.courant(y) = -.25;
}

template <class T>
void plot(T &solver, Gnuplot &gp)
{
  gp << "splot '-' binary"
     << gp.binfmt(solver.state())
     << " origin=(.5,.5,-1)"
     << " with image notitle"
     << ", '-' binary"
     << gp.binfmt(solver.state())
     << " origin=(.5,.5,0)"
     << " with lines notitle\n";
  gp.sendBinary(solver.state().copy());
  gp.sendBinary(solver.state().copy());
}

int main()
{
  int n[] = {24, 24}, nt = 75;
  Gnuplot gp;
  gp << "set term pdf size 10cm, 30cm\n"
     << "set output 'figure.pdf'\n"
     << "set multiplot layout 4,1\n"
     << "set border 4095\n"
     << "set xtics out\n"
     << "set ytics out\n"
     << "unset ztics\n"
     << "set xlabel 'x/dx'\n"
     << "set ylabel 'y/dy'\n"
     << "set xrange [0:" << n[x] << "]\n"
     << "set yrange [0:" << n[y] << "]\n"
     << "set zrange [-1:1]\n"
     << "set cbrange [-.025:1.025]\n"
     << "set palette maxcolors 42\n";
  {
    solver_donorcell<cyclic<x>, cyclic<y>>
      slv(n[x], n[y]);
    setup(slv, n);
    gp << "set title 't/dt=0'\n";
    plot(slv, gp);
    slv.solve(nt);
    gp << "set title 'donorcell t/dt="<<nt<<"'\n";
    plot(slv, gp);
  }
  {
    const int it = 2;
    solver_mpdata<it, cyclic<x>, cyclic<y>>
      slv(n[x], n[y]);
    setup(slv, n);
    slv.solve(nt);
    gp << "set title 'mpdata<" << it << "> "
       << "t/dt=" << nt << "'\n";
    plot(slv, gp);
  }
  {
    const int it = 44;
    solver_mpdata<it, cyclic<x>, cyclic<y>>
      slv(n[x], n[y]);
    setup(slv, n);
    slv.solve(nt);
    gp << "set title 'mpdata<" << it << "> "
       << "t/dt=" << nt << "'\n";
    plot(slv, gp);
  }
}
```

The following listings contain the Python and Fortran counterparts to listing C.20 (with the setup and plotting logic omitted).

listing P.19 (Python)

```
slv = solver_mpdata(it, cyclic, cyclic, nx, ny)
slv.state()[:] = read_file(fname, nx, ny)
slv.courant(0)[:] = Cx
slv.courant(1)[:] = Cy
slv.solve(nt)
```

listing F.23 (Fortran)

```
  type(mpdata_t) :: slv
  type(cyclic_t), target :: bcx, bcy
  integer :: nx, ny, nt, it
  real(real_t) :: Cx, Cy
  real(real_t), pointer :: ptr(:,:)
```

listing F.24 (Fortran)

```
  call slv%ctor(it, bcx, bcy, nx, ny)

  ptr => slv%state()
  call read_file(fname, ptr)

  ptr => slv%courant(0)
  ptr = Cx

  ptr => slv%courant(1)
  ptr = Cy

  call slv%solve(nt)
```

References

[1] B.J. Abiodun, W.J. Gutowski, A.A. Abatan and J.M. Prusa, CAM-EULAG: A non-hydrostatic atmospheric climate model with grid stretching, *Acta Geophys.* **59**(6) (2011), 1158–1167.

[2] J.A. Añel, The importance of reviewing the code, *Comm. ACM* **54**(5) (2011), 40–41.

[3] A. Arakawa and V.R. Lamb, Computational design of the basic dynamical process of the UCLA general circulation model, in: *Methods in Computational Physics*, Vol. 17, Academic Press, New York, 1977, pp. 173–265.

[4] N. Barnes and D. Jones, Clear climate code: Rewriting legacy science software for clarity, *IEEE Software* **28**(6) (2011), 36–42.

[5] C.F. Bolz, A. Cuni, M. Fijałkowski, M. Leuschel, S. Pedroni and A. Rigo, Runtime feedback in a meta-tracing JIT for efficient dynamic languages, in: *ICOOOLPS'11: Proceedings of the 6th Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems*, 2011.

[6] F. Buschmann, To pay or not to pay technical debt, *IEEE Software* **28**(6) (2011), 29–31.

[7] J.R. Cary, S.G. Shasharina, J.C. Cummings, J.V.W. Reynders and P.J. Hinker, Comparison of C++ and Fortran 90 for object-oriented scientific programming, *Comp. Phys. Comm.* **105**(1) (2011), 20–36.

[8] P. Charbonneau and P.K. Smolarkiewicz, Modeling the solar dynamo, *Science* **340**(6128) (2013), 42–43.

[9] B. Einarsson (ed.), *Accuracy and Reliability in Scientific Computing*, SIAM, Philadelphia, PA, USA, 2005.

[10] T. Ezer, H. Arango and A.F. Shchepetkin, Developments in terrain-following ocean models: intercomparisons of numerical aspects, *Ocean Model.* **4**(3) (2002), 249–267.

[11] S.M. Freeman, T.L. Clune and R.W. Burns III, Latent risks and dangers in the state of climate model software development, in: *Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research*, ACM, 2010, pp. 111–114.

[12] S.M. Griffies, C. Boning, F.O. Bryan, E.P. Chassignet, R. Gerdes, H. Hasumi, A. Hirst, A.-M. Treguier and D. Webb, Developments in ocean climate modelling, *Ocean Model.* **2**(3,4) (2000), 123–192.

[13] K. Iglberger, G. Hager, J. Treibig and U. Rüde, Expression templates revisited: a performance analysis of current methodologies, *SIAM J. Sci. Comput.* **34**(2) (2012), C42–C69.

[14] R. Kendall, J.C. Carver, D. Fisher, D. Henderson, A. Mark, D. Post, C.E. Rhoades, Jr. and S. Squires, Development of a weather forecasting code: A case study, *IEEE Software* **25**(4) (2008), 59–65.

[15] D.E. Knuth, Structured programming with go to statements, *Comput. Surv.* **6**(4) (1974), 261–301.

[16] S. Legutke, Building Earth system models, in: *Earth System Modelling – Volume 5: Tools for Configuring, Building and Running Models*, R. Ford, G. Riley, R. Budich and R. Redler, eds, Springer, 2012, pp. 45–54.

[17] J.W.-B. Lin, Why python is the next wave in earth sciences computing, *Bull. Amer. Meteor. Soc.* **93**(12) (2012), 1823–1824.

[18] A. Markus, *Modern Fortran in Practice*, Cambridge Univ. Press, 2012.

[19] Z. Merali, Computational science: ...Error... why scientific programming does not compute, *Nature* **467** (2010), 775–777.

[20] C.D. Norton, V.K. Decyk, B.K. Szymanski and H. Gardner, The transition and adoption to modern programming concepts for scientific computing in Fortran, *Sci. Prog.* **15**(1) (2007), 27–44.

[21] S. Paoli, C++ coding standard specification, Technical report, CERN European Laboratory for Particle Physics, 2000, available at: http://pst.web.cern.ch/PST/HandBookWorkBook/Handbook/Programming/CodingStandard/c++standard.pdf.

[22] M. Pilgrim, *Dive Into Python*, Apress, 2004.

[23] D.E. Post and L.G. Votta, Computational science demands a new paradigm, *Phys. Today* **58**(1) (2005), 35–41.

[24] W.H. Press, S.A. Teukolsky, W.T. Vetterling and B.P. Flannery, *Numerical Recipes in Fortran 90. The Art of Parallel Scientific Computing*, 2nd edn, Cambridge Univ. Press, 1996.

[25] W.H. Press, S.A. Teukolsky, W.T. Vetterling and B.P. Flannery, *Numerical Recipes. The Art of Scientific Computing*, 3rd edn, Cambridge Univ. Press, 2007.

[26] D. Rouson, J. Xia and X. Xu, *Scientific Software Design. The Object-Oriented Way*, Cambridge Univ. Press, 2012.

[27] P.K. Smolarkiewicz, A fully multidimensional positive definite advection transport algorithm with small implicit diffusion, *J. Comp. Phys.* **54**(2) (1984), 325–362.

[28] P.K. Smolarkiewicz, Multidimensional positive definite advection transport algorithm: an overview, *Int. J. Numer. Meth. Fluids* **50**(10) (2006), 1123–1144.

[29] R. Stallman et al., *GNU General Public License*, Free Software Foundation, 2007, available at: http://gnu.org/licenses/gpl, version 3.

[30] V. Stodden, I. Mitchell and R. LeVeque, Reproducible research for scientific computing: Tools and strategies for changing the culture, *Comput. Sci. Eng.* **14**(4) (2012), 13–17.

[31] B. Stroustrup, *The C++ Programming Language*, 4th edn, Addison Wesley, 2013.

[32] M. Sundberg, The everyday world of simulation modeling: The development of parameterizations in meteorology, *Sci. Technol. Hum. Val.* **34**(2) (2009), 162–181.

[33] J.P.M. Syvitski, S.D. Peckham, O. David, J.L. Goodall, C. Deluca and G. Theurich, Cyberinfrastructure and community environmental modeling, in: *Handbook of Environmental Fluid Dynamics: Systems, Pollution, Modeling, and Measurements*, Vol. 2, H.J.S. Fernando, ed., CRC Press/Taylor & Francis Group, Boca Raton, FL, USA, 2013, pp. 399–410.

[34] T. Veldhuizen and M. Jernigan, Will C++ be faster than Fortran?, in: *Scientific Computing in Object-Oriented Parallel Environments*, Y. Ishikawa, R. Oldehoeft, J. Reynders and M. Tholburn, eds, Lecture Notes in Computer Science, Vol. 1343, Springer, Berlin/Heidelberg, 1997, pp. 49–56.

[35] H.G. Weller, G. Tabor, H. Jasak and C. Fureby, A tensorial approach to computational continuum mechanics using object-oriented techniques, *Comput. Phys.* **12**(6) (1998), 620–631.

[36] G. Wilson, Where's the real bottleneck in scientific computing?, *Am. Sci.* **94**(1) (2006), 5–6.

[37] G. Wilson, D.A. Aruliah, C. Titus Brown, N.P. Chue Hong, M. Davis, R.T. Guy, S.H.D. Haddock, K. Huff, I.M. Mitchell, M. Plumbley, B. Waugh, E.P. White and P. Wilson, Best practices for scientific computing, *PLoS Biol.* **12**(1) (2014), e1001745.

[38] D.J. Worth, State of the art in object oriented programming with Fortran, Technical Report RAL-TR-2008-002, Science and Technology Facilities Council, 2008.

[39] M.Z. Ziemiański, M.J. Kurowski, Z.P. Piotrowski, B. Rosa and O. Fuhrer, Toward very high horizontal resolution NWP over the Alps: Influence of increasing model resolution on the flow pattern, *Acta Geophys.* **59**(6) (2011), 1205–1235.