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++, NumPy (Python) and Fortran’s built-in array containers. The solvers constitute implementations of the Multidimensional Positive-Definite Adective 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-
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 Advection 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), Fortran (listings prefixed with F, red frame), Python (listings prefixed with P, blue frame) and

### 2. Implementation of the formulae

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

```c++
using real_t = double;
```

```python
Real_t = 'float64'
```

```fortran
module real_m

implicit none

integer, parameter :: real_t = kind(0.0d0)

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:

\[
\partial_t \psi = -\nabla \cdot (\vec{v} \psi)
\]  

(1)

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

\(^3\)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/.

\[// code licensed under the terms of GNU GPL v3
// copyright holder: University of Warsaw\]

\[// code licensed under the terms of GNU GPL v3
// copyright holder: University of Warsaw\]

\[// code licensed under the terms of GNU GPL v3
// copyright holder: University of Warsaw\]
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:

```cpp
// Listing C.2
using arr_t = blitz::Array<double, 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:

```cpp
// Listing C.3
#define return_macro(expr) 
  return blitz::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:

```python
# Listing P.2
import numpy
from _numpy_pypy import set_invalidation
set_invalidation(False)
except ImportError:
  pass
import numpy
try:
  numpy.seterr(all='ignore')
except AttributeError:
  pass
```

\(^5\)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/.

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 \(\mathbf{C}\) are in fact a \(\{C_{x}^{[n]}, C_{y}^{[n]}\}\) array sequence.

Using an additional array dimension to represent the sequence elements is not considered for two reasons. First, the \(C_{x}^{[n]}\) and \(C_{y}^{[n]}\) 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

\(^7\)`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:

```fortran
module arrow_vec
use real_p
implicit none

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

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

type :: arrow_vec_t
  class(arr_vec_t), allocatable :: arrs(:)
class(arr_ptr_t), allocatable :: at(:)
integer :: length
contains
    procedure :: ctor => arrow_vec_ctor
    procedure :: init => arrow_vec_init
end type

contains

subroutine arrow_vec_ctor(this, n)
  class(arr_vec_t) :: this
  integer, intent(in) :: n
  integer::length = n
  allocate(this%arrs, 0 : n-1)
  allocate(this%arrs%a(:))
end subroutine

subroutine arrow_vec_init(this, n, i, j)
  class(arr_vec_t) :: this
  integer, intent(in) :: n
  integer, intent(in) :: i, j
  integer::length = n
  allocate(this%arrs%a(i : i+1, j : j+1))
  this%arrs%a => this%arrs%a
end subroutine

end module
```

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

Type `arr_ptr_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 `arr_ptr_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 `arr_ptr_t`, the $(i,j)$ 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.

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
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. $C_{i-1/2,j}$, $C_{i+1/2,j}$, $C_{i,j-1/2}$, and $C_{i,j+1/2}$ 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:

This way, the arrays representing vector field components can be indexed using $(i + h, j)$, $(i - h, j)$ 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-in 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:

2.4. Array index permutations

Hereinafter, the $\pi^d_{ab}$ 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:

$$
\sum_{d=0}^{1} \psi_{i,j}[i+1,j] + \psi_{i,j+1}.
$$

Blitz++ ships with the RectDomain class (aliased here as idx_t) for specifying array ranges in multiple dimensions. The $\pi$ 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
template specialisations):

```cpp
template <int d> using pi = int; // C++

inline int pi(const rng_t i, const rng_t j);
inline int pi(const rng_t i, const rng_t j);
inline int pi(const rng_t i, const rng_t j);
inline int pi(const rng_t i, const rng_t j);
inline int pi(const rng_t i, const rng_t j);
inline int pi(const rng_t i, const rng_t j);
inline int pi(const rng_t i, const rng_t j);
```

NumPy uses tuples of slices for addressing multidimensional array with a single object. Therefore, the following definition of function \(pi()\) suffices to represent \(\pi\):

```python
def pi(i, j):
    return (i, j)
```

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:

```fortran
module pi
use real_p
implicit none
contains
! function pi(i, j) result(return)
integer, intent(i) :: i
target :: arri(i, :) = real(real(i), allocatable, target = arri(i, :))
integer, intent(i) :: j
target :: arrj(j, :) = real(real(j), allocatable, target = arrj(j, :))
select case (d)
case (1) return => arri(i : i + 1, j : j + 1)
case (2) return => arrj(j : j + 1, i : i + 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)]:

\[
\psi[n+1]_{i,j} = \psi^n_{i,j} - \sum_{d=0}^{N-1} \left( F[\psi^n_{i,j} + \pi_{1,0}^{d}, C[0], \psi^n_{i,j} + \pi_{1/2,0}^{d}] - F[\psi^n_{i,j} + \pi_{1/2,0}^{d}, C[0], \psi^n_{i,j} + \pi_{1,0}^{d}] \right),
\]

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

\[
F(\psi_L, \psi_R, C) = \begin{cases} 
\max(C, 0) \cdot \psi_L + \min(0, C) \cdot \psi_R \\
C + |C| \cdot \psi_L + C - |C| \cdot \psi_R.
\end{cases}
\]

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

```cpp
template <int d> using p = int;
inline p pi(const int i, const int j);
```

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

```cpp
template <int d> using p = int;
const int p = 1;
return pi(p); // C++
```
and the actual two-dimensional donor-cell formula:

In Python, the same formulae are expressed as follows:

The Fortran counterparts are:

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:

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

2.6. MPDATA formulæ

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]):

where $\psi$ and $C$ represent values from the previous iteration and where:
For positive-definite $\psi$, the $A$ and $B$ terms take the following form:\footnote{Since $\psi \geq 0$, $|A| \leq 1$ and $|B| \leq 1$. See [28, Section 4.2] for description of adaptation of the formulae for advection of fields of variable sign.}

$$
A_{[d]}[i,j] = \psi_{[i,j]} + \pi_{d,0} - \psi_{[i,j]},
$$

$$
B_{[d]}[i,j] = \frac{1}{2} \left( \frac{\psi_{[i,j]} + \pi_{d,1} + \psi_{[i,j]} + \pi_{d,-1} - \psi_{[i,j]} + \pi_{d,0} + \psi_{[i,j]} + \pi_{d,-1} + \psi_{[i,j]} + \pi_{d,0}}{\psi_{[i,j]} + \pi_{d,1} + \psi_{[i,j]} + \pi_{d,0} + \psi_{[i,j]} + \pi_{d,-1} + \psi_{[i,j]} + \pi_{d,0}} \right).
$$

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

\texttt{where} construct in all three considered languages:

\begin{verbatim}
Listing F.10 (Fortran)
integer arn_t(i, j)
real(real_t) :: return(arn_t)
where(den > 0, nom / den, 0)
end function
\end{verbatim}

\begin{verbatim}
Listing F.11 (Fortran)
function update_fraction(den) result (return)
real(real_t) :: return(den)
where(den > 0, nom / den, 0)
end function
\end{verbatim}

\begin{verbatim}
Listing F.12 (Python)
def update_fraction(den):
    return numpy.where(den > 0, nom / den, 0)
\end{verbatim}

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

\begin{verbatim}
Listing F.13 (C++)
template<int d>
inline auto update_data_t(const arr_t ipsi, const rng_t u)
    return macro[
        psi[p(i0<d-1+i, j)] - psi[p(i0+d-1+i, j)],
        psi[p(i0+d-1+i, j)] - psi[p(i0+i, j)],
    ]
\end{verbatim}

Equation (6) takes the following form:

\begin{verbatim}
Listing F.14 (C++)
template<int d>
inline auto update_data_t(const arr_t ipsi, const rng_t u)
    return macro[
        psi[p(i0+d-1+i, j)] - psi[p(i0+i, j)],
        psi[p(i0+d-1+i, j)] - psi[p(i0+i, j)],
    ]
\end{verbatim}

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

\begin{verbatim}
Listing F.15 (Python)
def update_data_t(mpsi, u):
    return mpi
\end{verbatim}

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

\begin{verbatim}
Listing F.16 (Python)
def update_data_t(ipsi, u):
    return scalar
\end{verbatim}

\begin{verbatim}
Listing F.17 (Python)
def update_data_t(ipsi, u):
    return scalar
\end{verbatim}
Equation (5) takes the following form:

\[ V_{n+1} = V_n + k \frac{\nabla^2 V_n}{l} \]

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. 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, numexpr or Numba, 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 \(64 \times 64\) to \(2048 \times 2048\). The Gaussian impulse was advected for \(nt = \frac{2^4}{(nx \cdot ny)}\) timesteps, in order to assure comparable timing accuracy for all grid sizes (\(2^4\) 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).
3.2. Results

Figure 2 presents a plot of the peak memory use\textsuperscript{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 \((\text{nx} + 2) \times (\text{ny} + 2)\) arrays representing the two time levels of \(\psi\), a \((\text{nx} + 1) \times (\text{ny} + 2)\) array representing the \(C^{[x]}\) component of the Courant number field, a \((\text{nx} + 2) \times (\text{ny} + 1)\) 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 \(1024 \times 1024\). The roughly asymptotic values reached in all four setups for grid sizes larger than \(1024 \times 1024\) 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 \(256 \times 256\);
- 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.

\textsuperscript{17}The resident set size (rss) as reported by the GNU time utility (version packaged in Debian as 1.7-24).
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.\footnote{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.}

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++/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 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\cite{15}, and are generally advised to be avoided\cite[Section 3.12]{21}.

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;

![Fig. 4. Same as Fig. 3 for an AMD Phenom™ II X6 105ST processor. (Colors are visible in the online version of the article; http://dx.doi.org/10.3233/SPR-140379.)](image-url)
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]. 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-

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).
library 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 pre-processor 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.

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 sets 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-

---

\(^{20}\)CMake is a family of open-source, cross-platform tools automating building, testing and packaging of C/C++/Fortran software, see http://cmake.org/.

\(^{21}\)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.\textsuperscript{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 formulæ “translated” into C++, Python and Fortran in the main body of the paper.

A.1. Halo regions

The MPDATA formulæ defining $\psi_{i,j}^{[n+1]}$ as a function of $\psi_{i,j}^{[n]}$ (discussed in the following sections) feature terms such as $\psi_{i-1,j-1}^{[n]}$. One way of assuring validity of these formulæ 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 $i \sim i \pm \text{halo}$.

An \texttt{ext()} function is defined in all three implementations, in order to simplify coding of array range extensions:

```python
def ext2(i, n):
    if (type(n) == int) & (n == 1):
        n = one
    return slice(i - n, i + n + 1, 1)
```

Consequently, a range depicted by $i \pm 1/2$ may be expressed in the code as \texttt{ext(i, 0.5)}. In all three implementations, the \texttt{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 $\psi$ 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 \texttt{arrvec_t}; task (iii) is handled by the accessor methods; task (iv)
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 fulfill 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 \( C[x] \) 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:

```python
class solver(object):
    #stor-like method
    def __init__(self, bx, by, nx, ny, nbi):
        self.i = 0
        self.hbis = hbi
        self.i = slice(hbi, nx + hbi)
        self.j = slice(hbi, ny + hbi)
        self.bcx = bx
        self.bcy = by

    def psi = 
        hmpy.empty()
        ext = self.i, self.hbis
        stop = self.j, self.hbis
        1, real_1
        hmpy.empty()
        ext = self.j, self.hbis
        stop = self.i, self.hbis

    def cycle(self):
        ...
```

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 \((0, 0)\). However, since the whole halo logic is hid-
den within the solver, such details are not exposed to the user. The \texttt{bcx} and \texttt{bcy} boundary-condition specifications are passed to the solver through constructor-like \texttt{__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++):

```
module bce
  implicit none
  type, abstract:: bce
  contains
    procedure(bce_fill_halos), deferred:: fill_halos
    procedure(bce_init), deferred:: init
  end type
  abstract interface
    subroutine bce_fill_halos(bce, a, j
      import:: bc_m, real_t
      class(bce_t):: this
      real(real_t), allocatable:: a(:, j)
    end subroutine
    subroutine bce_init(bce, d, n, hilo)
      import:: bc_m, class(bce_t):: this
      integer:: d, n, hilo
    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:

```
module solver
  use arraw
  use bce
  implicit none
  type, abstract:: solver_t
  class(arraw_t), allocatable:: psi, C
  integer:: n, C
  integer:: (0, 1, 2)
  class(bce_t), pointer:: bcx, bcy
contains
  procedure:: solve => solver_solve
  procedure:: state => solver_state
  procedure:: co-rant => solver_co-rant
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 \texttt{fill\_halos()} method. An implementation of periodic boundary conditions in C++ is provided in the following listing:

```cpp
// Method invoked by the solver
void fill_halos(const rng_t &a, const rng_t &b, int hlo)
{
    a[0] = a[0] - hlo;
    b[0] = b[0] - hlo;
}
```

As hinted by the member field names, the \texttt{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 \texttt{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.

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 \texttt{solver\_donorcell} implementation of the abstract \texttt{solver} class:

```python
@<template-class box_t, class boy_t>
struct solver\_donorcell : solver\_box\_t, boy\_t
{
    solver\_donorcell(int nx, int ny) :
    solver\_box\_t(nx, ny),
    boy\_t()
    { }
}

void advecvp()
{
    donorcell\_op(
        this\rightarrow\psi, this\rightarrow\psi, this\rightarrow\psi,
        this\rightarrow\psi, this\rightarrow\psi,
        this\rightarrow\psi
    );
}
```

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.
A.5. MPDATA solver

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

```cpp
template<>
const int n_iters = n_iters > 2 ? 2 : 1;
array tmp(n_mpj);

// step
solver_update.data(nt, n, ny):

for (int n = 0; n < n_mpj; ++n) {
  for (int j = 1; j < j_max; ++j) {
    for (int i = 1; i < i_max; ++i) {
      // choosing input/output for antidiff C
      const array C_corr = (step == 1)
      ? this->C
      : (step == 2) ? tmp[i + 1] // odd steps
                    : tmp[i] // even steps
      ;
      // calculating the antidiffusive C
      C_corr[i] = mdata->nadj(i, C_corr[i], this->psi, this->C, this->, this->j, this->);}
    }
  }
}
```

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 the 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 \(C_{i+\frac{1}{2}}\) 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[0]` and `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.
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.\textsuperscript{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 max-
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.)
The following listings contain the Python and Fortran counterparts to listing C.20 (with the setup and plotting logic omitted).

```python
[listing F.19 (Python)]
924 slv = solver_update(t1, cyclic, cyclic, nx, ny)
925 slv.state[1]() = read_file(fname, nx, ny)
926 slv.courant[1]() = Ca
927 slv.solve(nt)
928

[listing F.23 (Fortran)]
461 type(modedata_c) :: slv
462 type(myGslC, target) :: bxw, bxw
463 integer :: nx, ny, pt, it
464 real(real64_c) :: Cw, Cy
465 (continued)

[listing F.24 (Fortran)]
446 call s1vvelocit(bxw, bxw, nx, ny)
448 ptr -> slv.state()
449 call read_file(fname, ptr)
450 ptr -> slv.courant[1]
452 ptr = Cw
454 ptr -> slv.solve(nt)
458

References


