Sundials/ML: Connecting OCaml to the Sundials Numeric Solvers

Timothy Bourke
Inria Paris
École normale supérieure, PSL University
tim@tbrk.org

Jun Inoue
National Institute of Advanced Industrial Science and Technology
jun.inoue@aist.go.jp

Marc Pouzet
Sorbonne Universités, UPMC Univ Paris 06
École normale supérieure, PSL University
Inria Paris
marc.pouzet@ens.fr

This paper describes the design and implementation of a comprehensive OCaml interface to the Sundials library of numeric solvers for ordinary differential equations, differential algebraic equations, and non-linear equations. The interface provides a convenient and memory-safe alternative to using Sundials directly from C and facilitates application development by integrating with higher-level language features, like garbage-collected memory management, algebraic data types, and exceptions. Our benchmark results suggest that the interface overhead is acceptable: the standard examples are rarely twice as slow in OCaml than in C, and often less than 50% slower. The challenges in interfacing with Sundials are to efficiently and safely share data structures between OCaml and C, to support multiple implementations of vector operations, matrices, and linear solvers through a common interface, and to manage calls and error signalling to and from OCaml. We explain how we overcame these difficulties using a combination of standard techniques such as phantom types and polymorphic variants, and carefully crafted data representations.

1 Introduction

Sundials [9] is a suite of six numeric solvers: *CVODE*, for ordinary differential equations, *CVODES*, adding support for quadrature integration and sensitivity analysis, *IDA*, for differential algebraic equations, *IDAS*, adding support for quadrature integration and sensitivity analysis, *ARKODE*, for ordinary differential equations using adaptive-step additive Runge-Kutta methods, and *KINSOL*, for non-linear equations. The six solvers share data structures and operations for vectors, matrices, and linear solver routines. They are implemented in the C language.

In this article we describe the design and implementation of a comprehensive OCaml [13] interface to the Sundials library, which we call *Sundials/ML*. The authors of Sundials, Hindmarsh *et al.* [10], describe "a general movement away from Fortran and toward C in scientific computing" and note both the utility of C's pointers, structures, and dynamic memory management for writing such software, and also the availability, efficiency, and relative ease of interfacing to it from Fortran. So, why bother writing an interface from OCaml? We think that OCaml interfaces to libraries like Sundials are ideal for (i) programs that mix numeric computation with symbolic manipulation, like interpreters and simulators for hybrid modelling languages; (ii) rapidly developing complex numerical models; and (iii) incorporating

Kenichi Asai and Mark Shinwell (Eds.)
ML/OCaml 2016
EPTCS 285, 2018, pp. 101–130, doi:10.4204/EPTCS.285.4

numerical approximation into general-purpose applications. Compared to C, OCaml detects many mistakes through a combination of static analyses (strong typing) and dynamic checks (for instance, array bounds checking), manages memory automatically using garbage collection, and propagates errors as exceptions rather than return codes. Not only does the OCaml type and module system detect a large class of programming errors, it also enables rich library interfaces that clarify and enforce correct use. We exploit this possibility in our library; for example, algebraic data types are used to structure solver configuration as opposed to multiple interdependent function calls, polymorphic typing ensures consistent creation and use of sessions and linear solvers, and phantom types are applied to enforce restrictions on vector and matrix use. On the other hand, all such interfaces add additional code and thus runtime overhead and the possibility of bugs—we discuss these issues in section 4.

The basic techniques for interfacing OCaml and C are well understood [13, Chapter 20][15][5, Chapter 12][14, Chapters 19–21], and it only takes one or two weeks to make a working interface to one of the solvers using the basic array-based vector data structure. But not all interfaces are created equal!

It takes much longer to treat the full range of features available in the Sundials suite, like the two solvers with quadrature integration and sensitivity analysis, the different linear solver modules and their associated matrix representations, and the diverse vector data structures. In particular, it was not initially clear how to provide all of these features in an integrated way with minimal code duplication and good support from the OCaml type system. Section 3 presents our solution to this problem. The intricacies of the Sundials library called for an especially careful design, particularly in the memory layout of a central vector data structure which took several iterations to perfect. The key challenges are to limit copying by modifying data in place, to interact correctly with the garbage collector to avoid memory leaks and data corruption, and to exploit the type and module systems to express documented constraints and provide a convenient interface. Our interface employs higher-order functions and currying, but such functional programming techniques are incidental; ultimately, we cannot escape the imperative nature of the underlying library. Similarly, in several instances static types are unable to encode constraints that change with the solver's state, and we are obliged to add dynamic checks to provide a safe and truly high-level interface.

Otherwise, a significant engineering effort is required to create a robust build system able to support the various configurations and versions of Sundials on different platforms, and to translate and debug the 100-odd standard examples that were indispensable for designing, evaluating, and testing our interface. Section 4 summarizes the performance results produced through this effort.

The Sundials/ML interface is used in the runtime of the Zélus programming language [2]. Zélus is a synchronous language [1] extended with ordinary differential equations (ODEs) for programming embedded systems and modelling their environments. Its compiler generates OCaml code to be executed together with a simulation runtime that orchestrates discrete computations and the numerical approximation of continuous trajectories.

The remainder of the paper is structured as follows. First, we describe the overall design of Sundials and Sundials/ML from a library user's point-of-view with example programs (section 2). Then we explain the main technical challenges that we overcame and the central design choices (section 3). Finally, we present an evaluation of the performance of our binding (section 4) before concluding (section 5).

The documentation and source code—approximately 15 000 lines of OCaml, and 17 000 lines of C, not counting a significant body of examples and tests—of the interface described in this paper is available under a 3-clause BSD licence at `http://inria-parkas.github.io/sundialsml/`. The code has been developed intermittently over eight years and adapted for successive Sundials releases through to the current 3.1.0 version.

2 Overview

In this section, we outline the Application Programming Interfaces (APIs) of Sundials and Sundials/ML and describe their key data structures. We limit ourselves to the elements necessary to explain and justify subsequent technical points. A complete example is presented at the end of the section.

2.1 Overview of Sundials

A mathematical description of Sundials can be found in Hindmarsh *et al.* [9]. The user manuals[1] give a thorough overview of the library and the details of every function. The purposes of the four basic solvers are readily summarized:

- *CVODE* approximates $Y(t)$ from $\dot{Y} = f(Y)$ and $Y(t_0) = Y_0$.
- *IDA* approximates $Y(t)$ from $F(Y,\dot{Y}) = 0$, $Y(t_0) = Y_0$, and $\dot{Y}(t_0) = \dot{Y}_0$.
- *ARKODE* approximates $Y(t)$ from $M\dot{Y} = f_E(Y) + f_I(Y)$ and $Y(t_0) = Y_0$.
- *KINSOL* calculates U from $F(U) = 0$ and initial guess U_0.

The problems are stated over vectors Y, $\dot{Y}$, and U, and expressed in terms of a function f from states to derivatives, a function F from states and derivatives to a residual, or the combination of an explicit function f_E and an implicit function f_I, both from states to derivatives, together with a mapping matrix M. The first three solvers find solutions that depend on an independent variable t, usually considered to be the simulation time. The two solvers that are not mentioned above, *CVODES* and *IDAS*, essentially introduce a set of parameters P—giving models $f(Y,P)$ and $F(Y,\dot{Y},P)$—and permit the calculation of parameter sensitivities, $S(t) = \frac{\partial Y(t)}{\partial P}$ using a variety of different techniques.

Four features are most relevant to the OCaml interface: solver sessions, vectors, matrices, and linear solvers.

2.1.1 Solver sessions

Using any of the Sundials solvers involves the same basic pattern: (*i*) a session object is created; (*ii*) several inter-dependent functions are called to initialize the session; (*iii*) "set$*$" functions are called to give parameter values; (*iv*) a "solve" or "step" function is called repeatedly to approximate a solution; (*v*) "get$*$" functions are called to retrieve results; (*vi*) the session and related data structures are freed. The sequence in figure 1, extracted from an example distributed with Sundials, is typical of steps *i* to *iii*. Line 1 creates a session with the (*CVODE*) solver, that is, an abstract type implemented as a pointer to a structure containing solver parameters and state that is passed to and manipulated by all subsequent calls. This function call, and all the others, are followed by statements that check return codes. Line 3 specifies a "user data" pointer that is passed by the solver to all callback functions to provide session-local storage. Line 5 specifies a callback function `f` that defines the problem to solve, here a function from a vector of variable values to their derivatives ($\dot{X} = f(X)$), an initial value `T0` for the independent variable, and a vector of initial variable values `u` that implicitly defines the problem size ($X_0 = u$). The other calls specify tolerance values (line 7), instantiate an iterative linear solver (line 9), attach it to the solver session (line 11), and set callback functions `jtv`, defining the problem Jacobian (line 13), and `Precond` and `PSolve`, defining preconditioning (line 15). The loop that calculates values of X over time, and the functions that retrieve solver results and statistics, and free memory are not shown.

[1] https://computation.llnl.gov/casc/sundials/

```
1  cvode_mem = CVodeCreate(CV_BDF, CV_NEWTON);
2  if(check_flag((void *)cvode_mem, "CVodeCreate", 0)) return(1);

3  flag = CVodeSetUserData(cvode_mem, data);
4  if(check_flag(&flag, "CVodeSetUserData", 1)) return(1);

5  flag = CVodeInit(cvode_mem, f, T0, u);
6  if(check_flag(&flag, "CVodeInit", 1)) return(1);

7  flag = CVodeSStolerances(cvode_mem, reltol, abstol);
8  if (check_flag(&flag, "CVodeSStolerances", 1)) return(1);

9  LS = SUNSPGMR(u, PREC_LEFT, 0);
10 if(check_flag((void *)LS, "SUNSPGMR", 0)) return(1);

11 flag = CVSpilsSetLinearSolver(cvode_mem, LS);
12 if (check_flag(&flag, "CVSpilsSetLinearSolver", 1)) return 1;

13 flag = CVSpilsSetJacTimes(cvode_mem, jtv);
14 if(check_flag(&flag, "CVSpilsSetJacTimes", 1)) return(1);

15 flag = CVSpilsSetPreconditioner(cvode_mem, Precond, PSolve);
16 if(check_flag(&flag, "CVSpilsSetPreconditioner", 1)) return(1);
```

Figure 1: Extract from the cvDiurnal_kry example in C, distributed with Sundials [9].

While the *IDA* and *KINSOL* solvers follow the same pattern, using the *CVODES* and *IDAS* solvers is a bit more involved. These solvers provide additional calls that augment a basic solver session with features for more efficiently calculating certain integrals and for analyzing the sensitivity of a solution to changes in model parameters using either so called forward methods or adjoint methods. The adjoint methods involve solving an ODE or differential algebraic equation (DAE) problem by first integrating normally, and then initializing new "backward" sessions that are integrated in the reverse direction.

The routines that initialize and configure solver sessions are subject to rules constraining their presence, order, and parameters. For instance, in the example, the call to `CVodeSStolerances` must follow the call to `CVodeInit` and precede calls to the step function; calling `CVodeCreate` with the parameter `CV_NEWTON` necessitates calls to configure a linear solver; and the `CVSpilsSetLinearSolver` call requires an iterative linear solver such as `SUNSPGMR` which, in turn, requires a call to `CVSpilsSetJacTimes` and, since the `PREC_LEFT` argument is given, a call to `CVSpilsSetPreconditioner` with at least a `PSolve` value.

2.1.2 Vectors

The manipulation of vectors is fundamental in Sundials. Vectors of floating-point values are represented as an abstract data type termed an *nvector* which combines a data pointer and 26 function pointers to operations that manipulate the data. Typical of these operations are *nvlinearsum*, which calculates the scaled sum of nvectors X and Y into a third vector Z, that is, $Z = aX + bY$, and *nvmaxnorm*, which returns the maximum absolute value of an nvector X. Nvectors must also provide an *nvclone* operation that produces a new nvector of the same size and with the same operations as an existing one. Solver

sessions are seeded with an initial nvector that they clone internally and manipulate solely through the abstract operations—they are thus defined in a data-independent manner.

Sundials provides eight instantiations of the nvector type: serial nvectors, parallel nvectors, OpenMP nvectors, Pthreads nvectors, Hypre ParVector nvectors, PETSC nvectors, RAJA nvectors, and CUDA nvectors. Serial nvectors store and manipulate arrays of floating-point values. Parallel nvectors store a local array of floating-point values and a Message Passing Interface (MPI) communicator; some operations, like *nvlinearsum*, simply loop over the local array, while others, like *nvmaxnorm*, calculate locally and then synchronize via a global reduce operation. OpenMP nvectors and Pthreads nvectors operate concurrently on arrays of floating-point values. Despite the lack of real multi-threading support in the OCaml runtime, binding to these nvector routines is unproblematic since they do not call back into user code. The serial, OpenMP, and Pthreads nvectors provide operations for accessing the underlying data array directly; this feature is notably exploited in the implementation of certain linear solvers. The Hypre ParVector, PETSC, and RAJA nvectors interface with scientific computing libraries that have not been ported to OCaml; they are not supported by Sundials/ML. We have not yet attempted to support CUDA nvectors. Finally, library users may also provide their own *custom nvectors* by implementing the set of basic operations over a custom representation.

2.1.3 Matrices

In recent versions of Sundials, matrices are treated like nvectors: they are implemented as an abstract data type combining a data pointer with pointers to 9 abstract operations [10, §7]. There are, for instance, operations to clone, to destroy, to scale and add to another matrix, and to multiply by an nvector.

Implementations are provided for two-dimensional dense, banded, and sparse matrices, and it is possible for users to define custom matrices by implementing the abstract operations. The *matrix content* of dense matrices consists of fields for the matrix dimensions and data length, a data array of floating-point values, and an array of pre-calculated column pointers into the data array. The content of banded matrices, which only contain the main diagonal and a certain number of diagonals above and below it, are represented similarly but with extra fields to record the numbers of diagonals. The content of sparse matrices includes the number of non-zero elements that can potentially be stored, a data array of non-zero elements, and two additional integer arrays. The interpretation of the integer arrays depends on a storage format field. For Compressed-Sparse-Column (CSC) matrices, an *indexptrs* array maps column numbers (from zero) to indices of the data array and an *indexvals* array maps indices of the data array to row numbers (from zero). So, for instance, the non-zero elements of the $(j-1)$th column are stored consecutively in the data array at indices $indexptrs[j] \leq k < indexptrs[j+1]$, and the row of each element is $indexvals[k]+1$. For Compressed-Sparse-Row (CSR) matrices, *indexptrs* maps row numbers to data indices and *indexvals* maps data indices to column numbers.

In the interests of calculation speed, the different Sundials routines often violate the abstract interface and access the underlying representations directly. This induces compatibility constraints: for instance, dense matrices may only be multiplied with serial, OpenMP, and Pthreads nvectors, and the KLU linear solver only manipulates sparse matrices. In section 3.3, we explain how these rules are expressed as typing constraints in the OCaml interface.

2.1.4 Linear solvers

Each of the solvers must resolve non-linear algebraic systems. For this, they use either "functional iteration" (*CVODE* and *CVODES* only), or, usually, 'Newton iteration'. Newton iteration, in turn, requires

the solution of linear systems.

Several linear solvers are provided with Sundials. They are instantiated by generic routines, like `SUNSPGMR` in the example, and attached to solver sessions. Generic linear solvers factor out common algorithms that solver-specific linear solvers specialize and combine with callback routines, like `jtv`, `Precond`, and `Psolve` in the example. Internally, generic linear solvers are implemented, similarly to nvectors and matrices, as an abstract data type combining a data pointer with 13 function pointers. Solver-specific linear solvers are invoked through a generic interface comprising four function pointers in a session object: *linit*, *lsetup*, *lsolve*, and *lfree*. Users may provide their own *custom* linear solvers by specifying subsets of the 13 operations, or even their own *alternate* linear solvers by directly setting the four pointers.

Sundials includes three main linear solver families: a diagonal approximation of system Jacobians using difference quotients, Direct Linear Solvers (DLS) that perform LU factorization on Jacobian matrices, and Scaled Preconditioned Iterative Linear Solvers (SPILS) based on Krylov methods. The DLS modules require callback routines to calculate an explicit representation of the Jacobian matrix of a system. The SPILS modules require callback routines to multiply vectors by an (implicit) Jacobian matrix and to precondition.

As was the case for solver sessions, the initialization and use of linear solvers is subject to various rules. For instance, the DLS modules exploit the underlying representation of serial, OpenMP, and Pthreads nvectors, and they cannot be used with other nvectors. The SPILS modules combine a method (SPGMR, SPFGMR, SPBCGS, SPTFQMR, or PCG) with an optional preconditioner. There are standard preconditioners (left, right, or both) for which users supply a solve function and, optionally, a setup function, and which work for any nvector, and also a banded matrix preconditioner that is only compatible with serial nvectors, and a Band-Block-Diagonal (BBD) preconditioner that is only compatible with parallel nvectors. The encoding of these restrictions into the OCaml interface is described in section 3.4.

2.2 Overview of Sundials/ML

The structure of the OCaml interface mostly follows that of the underlying library and values are consistently and systematically renamed: module-identifying prefixes are replaced by module paths and words beginning with upper-case letters are separated by underscores and put into lower-case. For instance, the function name `CVSpilsSetGSType` becomes `Cvode.Spils.set_gs_type`. This makes it easy to read the original documentation and to adapt existing source code, like, for instance, the examples provided with Sundials. We did, however, make several changes both for programming convenience and to increase safety, namely: (i) solver sessions are mostly configured via algebraic data types rather than multiple function calls; (ii) errors are signalled by exceptions rather than return codes; (iii) user data is shared between callback routines via partial function applications (closures); (iv) vectors are checked for compatibility using a combination of static and dynamic checks; and (v) explicit free commands are not necessary since OCaml is a garbage-collected language.

The OCaml program extract in figure 2 is functionally equivalent to the C code of figure 1. But rather than specifying Newton iteration by passing a constant (`CV_NEWTON`) and later calling linear solver routines (like `CVSpilsSetLinearSolver`), a solver session is configured by passing a value that contains all the necessary parameters. This makes it impossible to specify Newton iteration without also properly configuring a linear solver. The given value is translated by the interface into the correct sequence of calls to Sundials. The interface checks the return code of each function call and raises an exception if necessary. In the extract, the `~setup` and `~jac_times_vec` markers denote labelled arguments [13, §4.1]; we use them for optional arguments, as in this example, and also to clarify the

```
let cvode_mem =
  Cvode.(init BDF
    (Newton Spils.(solver (spgmr u)
                          ~jac_times_vec:(None, jtv data)
                          (prec_left ~setup:(precond data) (psolve data))))
    (SStolerances (reltol, abstol))
    (f data) t0 u)
in
```

Figure 2: Extract from our OCaml adaptation of cvDiurnal_kry. The definitions of the `precond`, `jtv`, `psolve`, and `f` functions and those of the `data`, `reltol`, `abstol`, `t0`, and `u` variables are not shown.

meaning of multiple arguments of the same type. The callback functions—`precond`, `jtv`, `psolve`, and `f`—are all applied to `data`. This use of partial application over a shared value replaces the "user data" mechanism of Sundials (`CVodeSetUserData`) that provides session-local storage; it is more natural in OCaml and it frees the underlying user data mechanism for use by the interface code. As in the C version, `t0` is the initial value of the independent variable, and `u` is the vector of initial values. Figure 2 uses the OCaml local open syntax [13, §7.7.7], `Cvode.(···)` and `Spils.(···)`, to access the functions `Cvode.init`, `Cvode.Spils.solver`, `Cvode.Spils.spgmr`, and `Cvode.Spils.prec_left`, and also the constructors `Cvode.BDF`, `Cvode.Newton`, and `Cvode.SStolerances`.

Not all options are configured at session creation, that is, by the call to `init`. Others are set via later calls; for instance, the residual tolerance value could be fine tuned by calling:

```
Cvode.Spils.set_eps_lin cvode_mem e
```

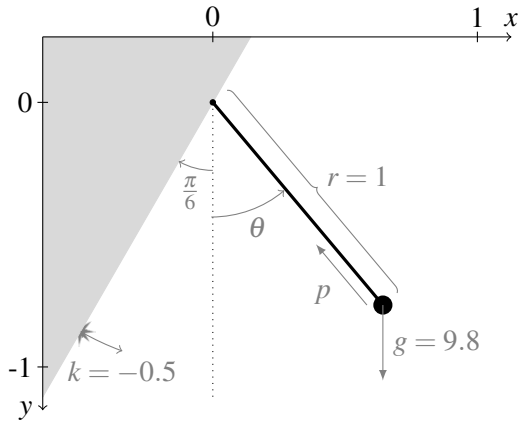
In choosing between the two possibilities, we strove for a balance between enforcing correct library use and providing a simple and natural interface. For instance, bundling the choice of Newton iteration with that of a linear solver and its preconditioner exploits the type system both to clarify how the library works and to avoid runtime errors. Calls like that to `set_eps_lin`, on the other hand, are better made separately since the default values usually suffice and since it may make sense to change such settings between calls to the solver.

The example code treats a session with *CVODE*, but the *IDA* and *KINSOL* interfaces are similar. The *CVODES* and *IDAS* solvers function differently. One of the guiding principles behind the C versions of these solvers is that their extra features be accessible simply by adding extra calls to existing programs and linking with a different library. We respect this choice in the design of the OCaml interface. For example, additional "quadrature" equations are added to the session created in figure 2 by calling:

```
Cvodes.Quadrature.init cvode_mem fq yq
```

which specifies a function `fq` to calculate the derivatives of the additional equations, and also their initial values in `yq`. While it would have been possible to signal such enhancements in the session type, we decided that this would complicate rather than clarify library use, especially since several enhancements—namely quadratures, forward sensitivity, forward sensitivity with quadratures, and adjoint sensitivity—and their combinations are possible. We must thus sometimes revert to runtime checks to detect whether features are used without having been initialized. These checks are not always performed by the underlying library and misuse of certain features can give rise to segmentation faults. The choice of whether to link Sundials/ML with the basic solver implementations or the enhanced ones is made during installation.

To calculate sensitivities using the adjoint method, a library user must first "enhance" a solver session using `Cvodes.Adjoint.init`, then calculate a solution by taking steps in the forward direction,



Polar ODEs

$$\theta(0) = \frac{\pi}{2}$$
$$\frac{d\theta}{dt}(0) = 0$$
$$\frac{d^2\theta}{dt^2} = -g \sin\theta$$

Cartesian DAEs

$$x(0) = 1 \qquad y(0) = 0$$
$$\frac{dx}{dt}(0) = 0 \qquad \frac{dy}{dt}(0) = 0$$
$$\frac{d^2x}{dt^2} = px \qquad \frac{d^2y}{dt^2} = py - g$$
$$x^2 + y^2 = r^2$$

Figure 3: A simple pendulum model.

before attaching "backward" sessions, and then taking steps in the opposite direction. Such backward sessions are identified in Sundials by integers and the functions dedicated to manipulating them take as arguments a forward session and an integer. But other more generic functions are shared between normal and backward sessions and it is necessary to first acquire a backward session pointer (using `CVodeGetAdjCVodeBmem`) before calling them. Our interface hides these details by introducing a `Cvodes.Adjoint.bsession` type which is realized by wrapping a standard session in a constructor (to avoid errors in the interface code) and storing the parent session and integer code along with other backward-specific fields as explained in section 3.2.

The details of the interfaces to nvectors and linear solvers are deferred to sections 3.1 and 3.4 since the choices made and the types used are closely tied to the technical details of their representations in memory. We mention only that, unlike Sundials, the interface enforces compatibility between nvectors, sessions, and linear solvers. Such controls become more important when sensitivity enhancements are used, since then the vectors used are likely of differing lengths.

Sundials is a large and sophisticated library. It is thus important to provide high-quality documentation. For this, ocamldoc [13] and its ability to define new markup tags are invaluable. For instance, we combine the existing tags for including LaTeX with the MathJax library[2] to render mathematical descriptions inline, and we introduce custom tags to link to the extensive Sundials documentation.

2.3 Complete examples

We now present two complete programs that use the Sundials/ML interface. Consider the simple pendulum model shown in figure 3: a unit mass at the end of a steel rod is attached to an inclined wall by a friction-less hinge. The rod and mass are raised parallel to the ground and released at time $t_0 = 0$. The only forces acting on the mass are gravity and the tension from the rod. Our aim is to plot the position of the mass as t increases. We consider two equivalent models for the dynamics: ODEs in terms of the angle θ and DAEs in terms of the Cartesian coordinates x and y. When the mass hits the wall, its velocity is multiplied by a (negative) constant k.

[2] `http://www.mathjax.org/`

2.3.1 Polar coordinates in *CVODE*

The ODE model in terms of polar coordinates can be simulated with *CVODE*. We start by declaring constants from the model and replacing the basic arithmetic operators with floating-point ones.

```
let r, g, k = 1.0, 9.8, -0.5
and pi = 4. *. atan (1.)
and ( + ), ( - ), ( * ), ( / ) = ( +. ), ( -. ), ( *. ), ( /. )
```

The solver manipulates arrays of values, whose 0th elements track θ and whose 1st elements track $\frac{\mathrm{d}\theta}{\mathrm{d}t}$. We declare constants for greater readability.

```
let theta, theta' = 0, 1
```

The dynamics are specified by defining a right-hand-side function that takes three arguments: `t`, the time, `y`, a big array of state values, and `yd`, a big array to fill with instantaneous derivative values.

```
let rhs t y yd =
  yd.{theta}  <- y.{theta'};
  yd.{theta'} <- -. g * sin y.{theta}
```

Apart from the imperative assignments to `yd`, side-effects are not allowed in this function, since it will be called multiple times with different estimates for the state values.

Interesting events are communicated to the solver through zero-crossing expressions. The solver tracks the value of these expressions and signals when they change sign. We define a function that takes the same inputs as the last one and that fills a big array `r` with the values of the zero-crossing expressions.

```
let roots t y r =
  r.{0} <- -. pi / 6 - y.{theta}
```

The single zero-crossing expression is negative until the mass collides with the wall.

We create a big array,[3] initialized with the initial state values, and "wrap" it as an nvector.

```
let y = RealArray.of_list [ pi/2. ; 0. ]
let nv_y = Nvector_serial.wrap y
```

The big array, `y`, and the nvector, `nv_y`, share the same underlying storage. We will access the values through the big array, but the Sundials functions require an nvector.

We can now instantiate the solver, using the Adams-Moulton formulas, functional iteration, and the default tolerance values, and staring at $t_0 = 0$. We pass the `rhs` function defining the dynamics, the `roots` function defining state events, and the nvector of initial states (which the solver uses for storage).

```
let s = Cvode.(init Adams Functional default_tolerances
                    rhs ~roots:(1, roots) 0.0 nv_y)
```

Some solver settings are not configured through the `init` routine, but rather by calling functions that act imperatively on the session. Here, we set the maximum simulation time to 10 s and specify that we only care about "rising" zero-crossings, where a negative value becomes zero or positive.

```
Cvode.set_stop_time s 10.0;
Cvode.set_all_root_directions s Sundials.RootDirs.Increasing
```

There are two main reasons for not adding these settings as optional arguments to `init`. First and foremost, these settings may be changed during a simulation, for instance, to implement mode changes, so separate functions are required in any case. Second, calls to `init` can already become quite involved, especially when specifying a linear solver. Providing optional settings in separate functions divides the

[3] `RealArray` is a helper module for `Bigarray.Array1`s of `float`s.

interface into smaller pieces. Unlike the features handled by `init`, the only ordering constraint imposed on these functions is that they be called after session initialization.

The simulation is advanced by repeatedly calling `Cvode.solve_normal`. We define a first function to advance the simulation time `t` to a given value `tnext`. When a zero-crossing is signalled, it updates the array element for θ', reinitializes the solver, and re-executes.

```
let rec stepto tnext t =
  if t >= tnext then t else
  match Cvode.solve_normal s tnext nv_y with
  | (tret, Cvode.RootsFound) ->
      y.{theta'} <- k * y.{theta'};
      Cvode.reinit s tret nv_y;
      stepto tnext tret
  | (tret, _) -> tret
```

A second function calls a routine to display the current state of the system (and pause slightly) before advancing the simulation time by `dt`.

```
let rec showloop t = if t < t_end then begin
  show (r * sin y.{theta}, -. r * cos y.{theta});
  showloop (stepto (t + dt) t)
end
```

The simulation is started by calling this function.

```
showloop 0.0
```

Despite tail-recursive calls in `stepto` and `showloop`, this program is undeniably imperative: callbacks update arrays in place, mutable memory is shared between arrays and nvectors, and sessions are progressively updated. This is a consequence of the structure of the underlying library and works well in an ML language like OCaml. We nevertheless benefit from a sophisticated type system (which infers all types in this example automatically), abstract data types and pattern matching, and exceptions.

2.3.2 Cartesian coordinates in *IDA*

The DAE model in terms of Cartesian coordinates can be simulated with *IDA* once it is rearranged into the form $F(t, X, \frac{\mathrm{d}X}{\mathrm{d}t}) = 0$. We introduce auxiliary variables v_x and v_y to represent the velocities and arrive at the following system with five equations and five unknowns.

$$v_x - \frac{\mathrm{d}x}{\mathrm{d}t} = 0 \qquad v_y - \frac{\mathrm{d}y}{\mathrm{d}t} = 0 \qquad \frac{\mathrm{d}v_x}{\mathrm{d}t} - px = 0 \qquad \frac{\mathrm{d}v_y}{\mathrm{d}t} - py + g = 0 \qquad x^2 + y^2 - 1 = 0$$

The variable p accounts for the "pull" of the rod. It is the only *algebraic* variable (its derivative does not appear); all the others are *differential* variables. The system above is of index 3, whereas an index 1 system is preferred for calculating the initial conditions. The index is lowered by differentiating the algebraic constraint twice, giving the following equation.

$$x\frac{\mathrm{d}v_x}{\mathrm{d}t} + y\frac{\mathrm{d}v_y}{\mathrm{d}t} + v_x^2 + v_y^2 = 0$$

Different substitutions of v_x and v_y with, respectively, $\frac{\mathrm{d}x}{\mathrm{d}t}$ and $\frac{\mathrm{d}y}{\mathrm{d}t}$ make for fourteen possible reformulations of the constraint that are equivalent in theory but that may influence the numeric solution. Here, we choose to implement the following form.

$$x\frac{\mathrm{d}v_x}{\mathrm{d}t} + y\frac{\mathrm{d}v_y}{\mathrm{d}t} + v_x\frac{\mathrm{d}x}{\mathrm{d}t} + v_y\frac{\mathrm{d}y}{\mathrm{d}t} = 0$$

Our second OCaml program begins with the same constant declarations as the first one. This time the X and $\frac{dX}{dt}$ arrays track five variables which we index,

```
let x, y, vx, vy, p = 0, 1, 2, 3, 4
```

and there are also five residual equations to index:

```
let vx_x, vy_y, acc_x, acc_y, constr = 0, 1, 2, 3, 4
```

The residual function itself is similar in principle to the `rhs` function of the previous example. It takes four arguments: `t`, the time, `vars`, a big array of variable values, `vars'`, a big array of variable derivative values, and `res`, a big array to fill with calculated residuals. We encode fairly directly the system of equations described above.

```
let residual t vars vars' res =
  res.{vx_x}   <- vars.{vx}   -  vars'.{x};
  res.{vy_y}   <- vars.{vy}   -  vars'.{y};
  res.{acc_x}  <- vars'.{vx}  -  vars.{p} * vars.{x};
  res.{acc_y}  <- vars'.{vy}  -  vars.{p} * vars.{y}  +  g;
  res.{constr} <- vars.{x} * vars'.{vx} + vars.{y} * vars'.{vy}
                  + vars.{vx} * vars'.{x} + vars.{vy} * vars'.{y}
```

The previous example applied functional iteration which does not require solving linear constraints. This is not possible in *IDA*, so we will use a linear solver for which we choose to provide a function to calculate a Jacobian matrix for the system. The function receives a record containing variables and their derivatives, a coefficient `c` that is proportional to the step size, and other values that we do not require. It also receives a dense matrix `out`, which we "unwrap" into a two-dimensional big array and fill with the non-zero partial derivatives of residuals relative to variables.

```
let jac Ida.({ jac_y = vars ; jac_y' = vars'; jac_coef = c }) out =
  let out = Matrix.Dense.unwrap out in
  out.{x,  vx_x}   <- -.c;             out.{y,  vy_y}  <- -.c;
  out.{vx, vx_x}   <- 1.;              out.{vy, vy_y}  <- 1.;
  out.{x,  acc_x}  <- -. vars.{p};     out.{y,  acc_y} <- -.vars.{p};
  out.{vx, acc_x}  <- c;               out.{vy, acc_y} <- c;
  out.{p,  acc_x}  <- -.vars.{x};      out.{p,  acc_y} <- -.vars.{y};
  out.{x,  constr} <- c * vars.{vx}  +  vars'.{vx};
  out.{y,  constr} <- c * vars.{vy}  +  vars'.{vy};
  out.{vx, constr} <- c * vars.{x}   +  vars'.{x};
  out.{vy, constr} <- c * vars.{y}   +  vars'.{y}
```

The zero-crossing function is now defined in terms of the Cartesian variables.

```
let roots t vars vars' r =
  r.{0} <- vars.{x} - vars.{y} * (sin (-. pi / 6.) / -. cos (-. pi / 6.))
```

We create two big arrays initialized with initial values and a guess for the initial value of p, and wrap them as nvectors.

```
let vars  = RealArray.of_list [x0; y0; 0.; 0.; 0.]
let vars' = RealArray.make 5 0.
let nv_vars, nv_vars' = Nvector.(wrap vars, wrap vars') in
```

We can now instantiate the solver session. We create and pass a generic linear solver on 5-by-5 dense matrices (`nv_vars` is only provided for compatibility checks) and specialize it for an *IDA* session with the Jacobian function defined above. We also specify scalar relative and absolute tolerances, the residual function, the zero-crossing function, and initial values for the time, variables, and variable derivatives.

```
let s = Ida.(init Dls.(solver ~jac (dense nv_vars (Matrix.dense 5)))
                  (SStolerances (1e-9, 1e-9))
                  residual ~roots:(1, roots) 0. nv_vars nv_vars')
```

If it were necessary to configure the linear solver or to extract statistics from it, we would have to declare a distinct variable for it, for example, `let ls = Ida.Dls.dense nv_vars (Matrix.dense 5)`.

Since we will be asking Sundials to calculate initial values for algebraic variables, that is, for p, we declare an nvector that classifies each variable as either differential or algebraic.

```
let d, a = Ida.VarId.differential, Ida.VarId.algebraic in
let var_types = Nvector.wrap (RealArray.of_list [ d; d; d; d; a ])
```

This information is given to the solver and algebraic variables are suppressed from local error tests.

```
Ida.set_id s var_types;
Ida.set_suppress_alg s true
```

Initial values for the algebraic variables and their derivatives are then calculated and updated in the appropriate nvectors for a first use at time `dt`.

```
Ida.calc_ic_ya_yd' s ~y:nv_vars ~y':nv_vars' ~varid:var_types dt
```

Then, as in the first program, we define a function to advance the simulation and check for zero-crossings. If a zero-crossing occurs, the variables and derivatives are updated, the solver is reinitialized, and the values of algebraic variables and their derivatives are recalculated.

```
  let rec stepto tnext t =
    if t >= tnext then t else
    match Ida.solve_normal s tnext nv_vars nv_vars' with
    | (tret, Ida.RootsFound) ->
        vars.{vx} <- k * vars.{vx};
        vars.{vy} <- k * vars.{vy};
        Ida.reinit s tret nv_vars nv_vars';
        Ida.calc_ic_ya_yd' s ~y:nv_vars ~y':nv_vars' ~varid:var_types (t + dt);
        stepto tnext tret
    | (tret, _) -> tret
```

The final function is almost the same as in the first program, except that now the state values can be passed directly to the display routine.

```
let rec showloop t = if t < t_end then begin
  show (vars.{x}, vars.{y});
  showloop (stepto (t + dt) t)
end in
showloop 0.0
```

This second program involves more technical details than the first one, even if, in this case, it solves the same problem. No new concepts are required from a programming point-of-view.

3 Technical details

We now describe and justify the main typing and implementation choices made in the Sundials/ML interface. For the important but standard details of writing stub functions and converting to and from OCaml values we refer readers to Chapter 20 of the OCaml manual [13]. We focus here on the representation in OCaml of nvectors (section 3.1), sessions (section 3.2), and linear solvers (section 3.4), describing some of the solutions we tried and rejected, and presenting the solutions. We also describe the treatment of Jacobian matrices (section 3.3).

3.1 Nvectors

Nvectors combine an array of floating-point numbers (`doubles`) with implementations of the 26 vector operations. The OCaml big array library [13] provides arrays of floating-point numbers that can be shared directly between OCaml and C. It is the natural choice for interfacing with the payloads of nvectors. Both Sundials' nvectors and OCaml's big arrays have a flag to indicate whether or not the payload should be freed when the nvector or big array is destroyed.

3.1.1 An attempted solution

A first idea for an interface is to work only with big arrays on the OCaml side of the library, and to convert them automatically *to* nvectors on calls into Sundials, and *from* nvectors on callbacks from Sundials. We did exactly this in early versions of our interface that only supported serial nvectors.[4] For calls into Sundials, like `Cvode.init` from the earlier example, the interface code creates a temporary serial nvector to pass into *CVODE* by calling the Sundials function

```
N_VMake_Serial(Caml_ba_array_val(b)->dim[0], (realtype *)Caml_ba_data_val(b))
```

which creates an nvector from an existing array. The two arguments extract the array size and the address of the underlying data from the big array data structure. Operations on the resulting nvector directly manipulate the data stored in the big array. This nvector is destroyed by calling *nvdestroy*—one of the 26 abstract operations defined for nvectors—before returning from the interface code.

For callbacks into OCaml, we create a big array for each nvector argument `v` passed in from Sundials by calling the OCaml function

```
caml_ba_alloc(CAML_BA_FLOAT64|CAML_BA_C_LAYOUT, 1, NV_DATA_S(v), &(NV_LENGTH_S(v)))
```

From left to right, the arguments request a big array of `double` values in row-major order indexed from 0, specify the number of dimensions, pass a pointer to the nvector's payload extracted using a Sundials macro, and give the length of the array using another Sundials macro (the last argument is an array with one element for each dimension). Again, operations on the big array modify the underlying array directly. After a callback, we set the length of the big array `b` to 0,

```
Caml_ba_array_val(b)->dim[0] = 0
```

as a precaution against the possibility, albeit unlikely, that a callback routine could keep a big array argument in a reference variable that another part of the program later accesses after the nvector memory has been freed. The OCaml runtime will eventually garbage collect the emptied big array. This mechanism can, however, be circumvented by creating a subarray which will have a distinct header, and hence dimension field, pointing to the same underlying memory [4].

This approach has two advantages: it is simple and library users need only work with big arrays. As for performance, calls into Sundials require `malloc` and `free` but they are outside critical solver loops, and, anyway, "wrapper" vectors can always be cached within the session value if necessary. Callbacks from Sundials do occur inside critical loops, but we can expect `caml_ba_alloc` to allocate on the relatively fast OCaml heap and a `malloc` is not required since we pass a data pointer. This approach has two drawbacks: it does not generalize well to OpenMP, Pthreads, parallel, and custom nvectors, and a big array header is always allocated on the major heap which may increase the frequency and cost of garbage collection. We tried to generalize this approach to handle parallel nvectors by using preprocessor macros, but the result was confusing for users, tedious to maintain, and increasingly unwieldy as we extended the interface to treat *CVODES* and *IDAS*.

[4] This approach is also taken in the NMAG library [7] for interfacing with *CVODE* for both serial and parallel nvectors. The Modelyze implementation [3] provides a similar interface to *IDA*, but explicitly copies values to and from standard OCaml arrays and nvectors.

3.1.2 The adopted solution

The solution we finally adopted exploits features of the nvector abstract datatype and polymorphic typing to treat nvectors more generically and without code duplication. The idea is straightforward: we pair an OCaml representation of the contents with a (wrapped) pointer to a C nvector structure, and we link both to the same underlying data array as in the previous solution. The difference is that we maintain both the OCaml view and the C view of the structure at all times.

The memory layout of our nvector is shown in figure 4. The OCaml type for accessing this structure is defined in the `Nvector` module as:

```
type ('data, 'kind) t = 'data * cnvec * ('data -> bool)
```

and used abstractly as `('data, 'kind) Nvector.t` throughout the interface. The `'data` points to the OCaml view of the payload (labelled "`payload`" in figure 4). For serial nvectors, `'data` is instantiated as a big array of `floats`. The phantom type [11] argument `'kind` is justified subsequently. The `cnvec` component is a custom block pointing to the `N_Vector` structure allocated in the C heap. The last component of the triple is a function that tests the compatibility of the nvector with another one: for serial nvectors, this means one of the same length, while for parallel nvectors, global sizes and MPI communicators are also checked. This component is used internally in our binding to ensure that, for instance, only compatible nvectors are added together. Since the compatibility check only concerns the payload, we use a function from type `'data` rather than type `Nvector.t`. This check together with the type arguments prevents nvectors being used in ways that would lead to invalid memory accesses. The two mechanisms help library users: types document how the library is used and dynamic checks signal problems at their point of occurrence (as an alternative to long debugging sessions).

With this representation, the `Nvector` module can provide a generic and efficient function for accessing the data from the OCaml side of the interface:

```
let unwrap ((payload, _, _) : ('data, 'kind) t) = payload
```

with the type `('data, 'kind) Nvector.t -> 'data`. Calls from OCaml into C work similarly by obtaining a pointer to the underlying Sundials nvector from the `cnvec` field.

Callbacks from C into OCaml require another mechanism. Stub functions are passed `N_Vector` values from which they must recover corresponding OCaml representations before invoking a callback. While it would be possible to modify the nvector contents field to hold both an array of data values and a pointer to an OCaml value, we wanted to use the original nvector operations without any additional overhead. Our solution is to allocate more memory than necessary for an `N_Vector` so as to add a "backlink" field that references the OCaml representation. The approach is summarised in figure 4. At left are the values in the OCaml heap: an `Nvector.t` and its `'data` payload. The former includes a pointer into the C heap to an `N_Vector` structure extended, hence the '+', with a third field that refers back to the data payload on the OCaml side. Callbacks can now easily retrieve the required value:

```
#define NVEC_BACKLINK(nvec) (((struct cnvec *)nvec)->backlink)
```

The backlink field must be registered as a global root with the garbage collector to ensure that it is updated if the payload is moved and also that the payload is not destroyed inopportunely. Pointing this global root directly at the `Nvector.t` would create a cycle across both heaps and necessitate special treatment to avoid memory leaks. We thus decided to pass payload values directly to callbacks, with the added advantage that callback functions have simpler types in terms of `'data` rather than `('data, 'kind) Nvector.t`. When there are no longer any references to an `Nvector.t`, it is collected and its finalizer frees the `N_Vector` and associated global root. This permits the payload to be collected when no other references to it exist. We found that this choice works well in practice provided

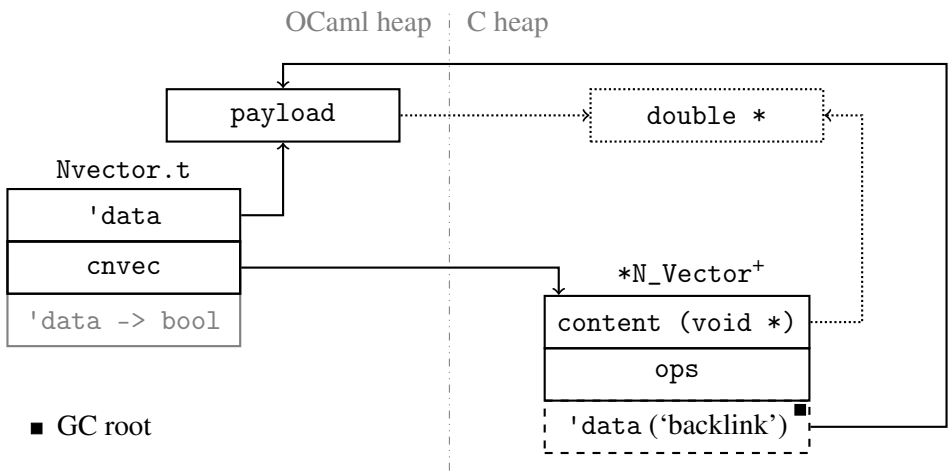


Figure 4: Interfacing nvectors (dotted lines are for serial nvectors).

the standard vector operations are also defined for payload values—that is, directly on values of type `'data`—since callbacks can no longer use the nvector operations directly. The only drawback we experienced was in implementing custom linear solvers. Such linear solvers take nvector arguments, which thus become payload values within OCaml, but they also work with callbacks into Sundials that take nvector arguments. The OCaml code must thus 'rewrap' payload values in order to use the callbacks.

The "backlink" field is configured when an `N_Vector` is created by a function in the OCaml `Nvector` module. The operations of serial and other `N_Vector`s are unchanged but for `nvclone`, `nvdestroy`, and `nvcloneempty` which are replaced by custom versions. The replacement `nvclone` allocates and sets up the backlink field and associated payload. For serial nvectors, it aliases the contents field to the data field of the big array payload which is itself allocated in the C heap, as shown in dotted lines in figure 4, and registers the backlink as a global root. The replacement `nvdestroy` removes the global root and frees memory allocated directly in the C heap but leaves the payload array to be garbage collected when it is no longer accessible. There are thus two classes of nvectors: those created on the OCaml side with a lifetime linked to the associated `Nvector.t` and determined by the garbage collector, and those cloned within Sundials, for which there is no `Nvector.t` and whose lifetime ends when Sundials explicitly destroys them, though the payload may persist.

Overriding the clone and destroy operations is more complicated than the first attempted solution, and does not work with nvectors created outside the OCaml interface (which would not have the extra backlink field). This means, in particular, that we forgo the possibility of code mixing Fortran, C, and OCaml, but otherwise this approach is efficient and generalizes well.

Within the OCaml interface, the serial, OpenMP, and Pthreads nvectors carry a big array payload, but at the C level each is represented by a different type of struct: those for the last two, for example, track the number of threads in use. OpenMP and Pthreads nvectors can be used anywhere that Serial nvectors can—since they are all manipulated through a common set of operations—except when the underlying representation is important, as in direct accesses to nvector data or through functions like `Nvector_pthreads.num_threads`. We enforce these rules using the `'kind` type variable introduced above and polymorphic variants [8][13, §4.2]. We use three polymorphic variant constructors, marked with backticks, and declare three type aliases as (closed) sets of these constructors:

```
type Nvector_serial.kind =   [`Serial]
```

```
type Nvector_pthreads.kind = [`Pthreads | Nvector_serial.kind]
type Nvector_openmp.kind =   [`OpenMP   | Nvector_serial.kind]
```

Here we abuse the syntax of OCaml slightly: in the real implementation, each kind is declared in the indicated module. The first line declares `Nvector_serial.kind` as a type whose only (variant) constructor is `` `Serial ``. The second line declares `Nvector_pthreads.kind` as a type whose only constructors are `` `Pthreads `` and `` `Serial ``, and likewise for the third line. In fact, the constructors are never used as values, since the `'kind` argument is a 'phantom' [11]: it only ever occurs on the left-hand side of type definitions. They serve only to express typing constraints. Functions that accept any kind of nvector are polymorphic in `'kind`, and those that only accept a specific kind of nvector are constrained with a specific `kind`, like one of the three listed above or others introduced specifically for parallel or custom nvectors. Functions that accept any of the three kinds listed above but no others, since they exploit the underlying serial data representation, take a polymorphic nvector whose `'kind` is constrained by

```
constraint 'kind = [>Nvector_serial.kind]
```

Such an argument can be instantiated with any type that includes at least the `` `Serial `` constructor. The fact that `Nvector.t` is opaque means that it can only be one of the nvector types `Nvector_serial.t`, `Nvector_pthreads.t`, or `Nvector_openmp.t`. An example is given in section 3.4.

For parallel nvectors, the payload is the triple:

```
(float, float64_elt, c_layout) Bigarray.Array1.t * int * Mpi.communicator
```

where the first element is a big array of local `floats`, the second gives the global number of elements, and the third specifies the MPI processes that communicate together.[5] We instantiate the `'data` type argument of `Nvector.t` with this triple and provide creation and clone functions that create aliasing for the big array and duplicate the other two elements between the OCaml and C representations. A specific kind is declared for parallel nvectors.

Fur custom nvectors, we define a record type containing a field for each nvector operation, and a compatibility check (`n_vcheck`), over an arbitrary payload type `'d`:

```
type 'd nvector_ops = {
  n_vcheck     : 'd -> 'd ->  bool;
  n_vclone     : 'd -> 'd;
  n_vlinearsum : float -> 'd -> float -> 'd -> 'd -> unit;
  n_vmaxnorm   : 'd -> float;
  ⋮
}
```

Such a record can then be used to create a wrapper function that turns payload values of type `'d` into custom nvectors, by calling:

```
val make_wrap : 'd nvector_ops -> 'd -> ('d, Nvector_custom.kind) Nvector.t
```

The resulting `Nvector.t` carries the type of payload manipulated by the given operations and a kind, `Nvector_custom.kind`, specific to custom nvectors. The kind permits distinguishing, for instance, between a custom nvector whose payload is a big array of `floats` and a standard serial nvector. While there is little difference from within OCaml—both have the same type of payload—the differences in the underlying representations are important from the C side. In the associated `N_Vector` data structure, we point the `ops` fields at generic stub code that calls back into OCaml and store the closures defining the operations in the `content` field. This field is registered as a global root with the garbage collector. The

[5] We use the OCaml MPI binding: `https://github.com/xavierleroy/ocamlmpi/`.

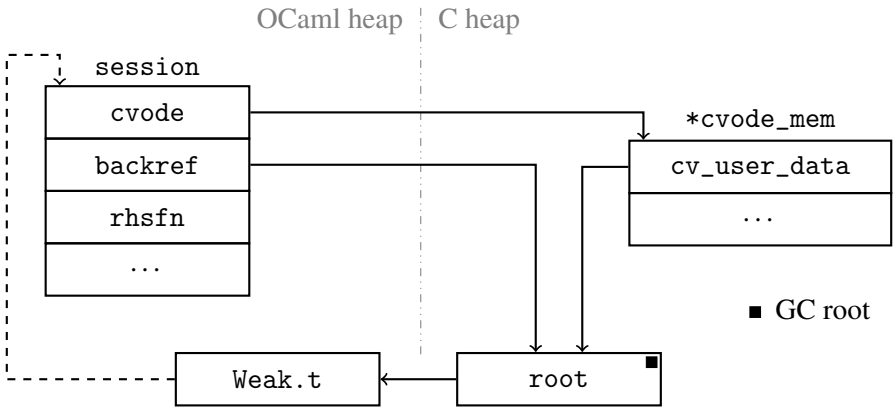


Figure 5: Interfacing (*CVODE*) sessions.

payload is stored using the backlink technique described earlier and depicted in figure 4. It is possible to create an OCaml-C reference loop by referring back to an `Nvector.t` value from within a custom payload or the set of custom operations, and thus to inhibit correct garbage collection. This is unlikely to happen in normal use and is difficult to detect, so we simply rule it a misuse of the library.

3.2 Sessions

OCaml session values must track an underlying C session pointer and also maintain references to callback closures and some other bookkeeping details. We exploit the user data feature of Sundials to implement callbacks. The main technical challenges are to avoid inter-heap loops and to smoothly accommodate sensitivity analysis features.

The solution we implemented is sketched in figure 5 and described below for *CVODE* and *CVODES*. The treatment of *IDA*, *IDAS*, *ARKODE*, and *KINSOL* is essentially the same. As for nvectors, OCaml session types are parametrized by `'data` and `'kind` type variables. They are represented internally as records:

```
type ('data, 'kind) session = {
  cvode           : cvode_mem;
  backref         : c_weak_ref;
  rhsfn           : float -> 'data -> 'data -> unit;
  ⋮
  mutable sensext : ('data, 'kind) sensext;
}
```

The type variables are instantiated from the nvector passed to the `init` function that creates session values. The type variables ensure a coherent use of nvectors, which is essential in operations that involve multiple nvectors, like *nvlinearsum*, since the code associated with one of the arguments is executed on the data of all of the arguments.

The `cvode` field of the session record contains a pointer to the associated Sundials session value `*cvode_mem`. The `cv_user_data` field of `*cvode_mem` is made to point at a `malloc`ed `value` that is registered with the garbage collector as a global root. This root value cannot be stored directly in `*cvode_mem` because Sundials only provides indirect access to the `cv_user_data` field through the functions `CvodeSetUserData` and `CvodeGetUserData`. We would have had to violate the interface to acquire the address of the field in order to register it as a global root.

```
1 static int rhsfn(realtype t, N_Vector y, N_Vector ydot, void *user_data)
2 {
3   CAMLparam0();
4   CAMLlocal2(session, r);
5   CAMLlocalN(args, 3);

6   WEAK_DEREF (session, *(value*)user_data);

7   args[0] = caml_copy_double(t);
8   args[1] = NVEC_BACKLINK(y);
9   args[2] = NVEC_BACKLINK(ydot);

10  r = caml_callbackN_exn(Field(session, RECORD_CVODE_SESSION_RHSFN), 3, args);

11  CAMLreturnT(int, CHECK_EXCEPTION (session, r, RECOVERABLE));
12 }
```

Figure 6: Typical Sundials/ML callback stub.

The root value must refer back to the `session` value since it is used in C level callback functions to obtain the appropriate OCaml closure. The `rhsfn` field shown in the record above is, for instance, the closure for the *CVODE* 'right-hand side function', the `f` of section 2. The root value stores a weak reference that is updated by the garbage collector if `session` is moved but which does not prevent the destruction of `session`. This breaks the cycle which exists across the OCaml and C heaps: storing a direct reference in the root value would prevent garbage collection of the `session` but the root value itself cannot be removed unless the `session` is first finalized. The `backref` field is used only by the finalizer of `session` to unregister the global root and to free the associated memory.

Figure 6 shows the callback stub for the `session.rhsfn` closure described above. The C function `rhsfn()` is registered as the right-hand side function for every *CVODE* session created by the interface. Sundials calls it with the value of the independent variable, `t`, an nvector containing the current value of the dependent variable, `y`, an nvector for storing the calculated derivative, `ydot`, and the session-specific pointer registered in `cv_user_data`. Lines 3 to 5 contain standard boilerplate for an OCaml stub function. Line 6 follows the references sketched in figure 5 to retrieve a `session` record: the `WEAK_DEREF` macro contains a call to `caml_weak_get`. The weak reference is guaranteed to point to a value since Sundials cannot be invoked from OCaml without passing the session value used in the callback. Line 7 copies the floating-point argument into the OCaml heap. Lines 8 and 9 recover the nvector payloads using the macro described in section 3.1. Line 10 retrieves and invokes the `rhsfn` closure from the session object. Finally, at line 11, the return value is determined by checking whether or not the callback raised an exception, and if so, whether it was the distinguished `RecoverableFailure` that signals to Sundials that recovery is possible.

3.2.1 An alternative session interface.

Another approach for linking the C `*cvode_mem` to an OCaml `session` value is outlined in figure 7. Since for a callback to occur, control must already have passed into the Sundials library through the interface, there will be a reference to the `session` value on the OCaml stack. It is thus possible to pass the reference to `CvodeSetUserData` before calling into Sundials. The reference will be updated

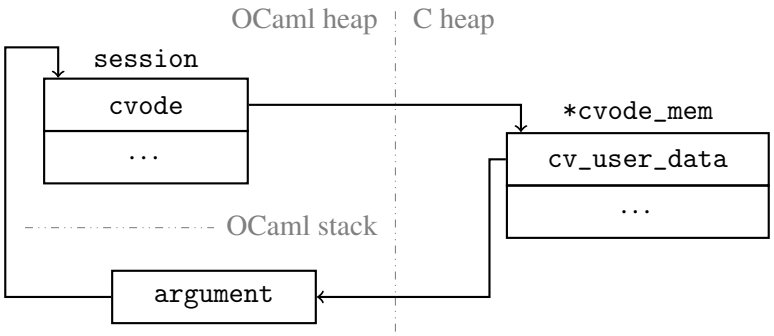


Figure 7: Alternative session interface (not adopted).

by the garbage collector as necessary, but not moved itself during the call. This approach is appealing as it requires neither global roots nor weak references. It also requires fewer interfacing instructions in performance critical callback loops due to fewer indirections and because there is no need to call `caml_weak_get`. Although this implementation is uncomplicated for functions like `rhsfn` that are only called during solving, it is more invasive for the error-handling functions which can, in principle, be triggered from nearly every call; either updates must be inserted everywhere or care must be taken to avoid an incorrect memory access. When using an adjoint sensitivity solver, the user data references of all backward sessions must be updated before solving, but the error-handling functions do not require special treatment since they are inherited from the parent session. We chose the approach based on weak references to avoid having to think through all such cases and also because our testing did not reveal significant differences in running time between the two approaches.

3.2.2 Quadrature and Sensitivity features.

Although the *CVODES* solver conceptually extends the *CVODE* solver, it is implemented in a distinct code base (and similarly for *IDAS* and *IDA*). For the OCaml library, we wanted to maintain the idea of an extended interface without completely duplicating the implementation. The library thus provides two modules, `Cvode` and `Cvodes`, that share the `session` type. As both modules need to access the internals of `session` values, we declare this type, and all the types on which it depends, inside a third module `Cvode_impl` that the other two include. To ensure the opacity of session types in external code, we simply avoid installing the associated cvode_impl.cmi compiled OCaml interface file. The mutable `sensext` field of the `session` record tracks the extra information needed for the sensitivity features. It has the type:

```
type ('data, 'kind) sensext =
    NoSensExt
  | FwdSensExt of ('data, 'kind) fsensext
  | BwdSensExt of ('data, 'kind) bsensext
```

The `NoSensExt` value is used for basic sessions without sensitivity analysis. The `FwdSensExt` value is used to augment a session with calculations of quadratures, forward sensitivities, and adjoint sensitivities. It contains additional callback closures and also a list of associated backward session values to prevent their garbage collection while they may still be required by C-side callback stubs which only hold weak references. The `BwdSensExt` value is used in backward sessions created for adjoint sensitivity analysis. It contains callback closures and a link to the parent session and an integer identifier sometimes required

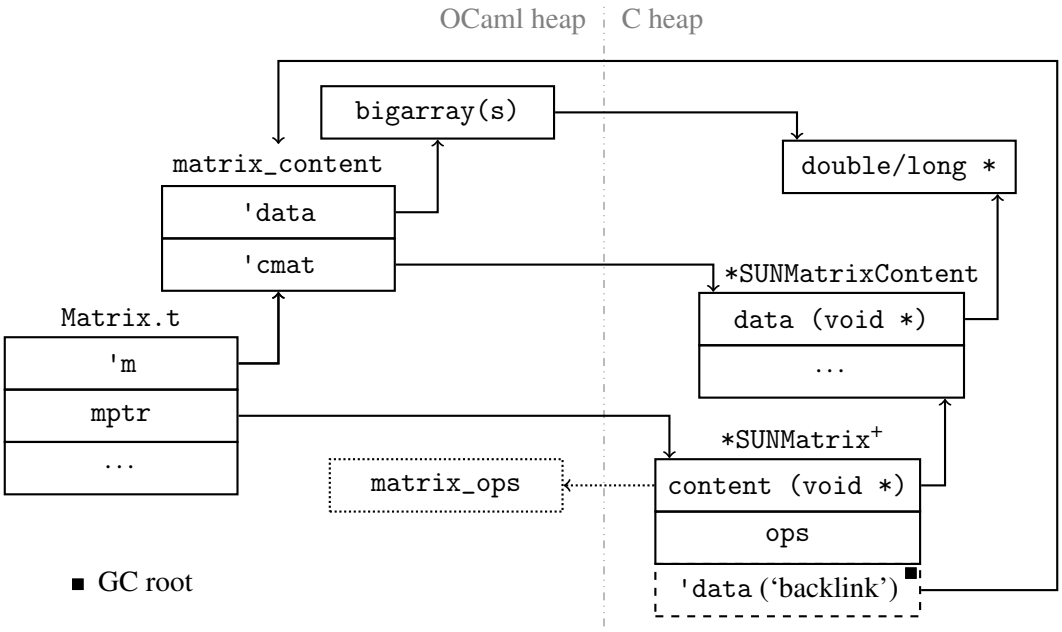


Figure 8: Interfacing matrices (dotted elements are for custom matrices only).

by Sundials functions. Reusing the basic session interface for backward sessions mirrors the approach taken in Sundials and simplifies the underlying implementation at the cost of some redundant fields—the normal callback closures are never used,—and an indirection to access the `bsensext` fields.

The only other complication in interfacing to *CVODES* (and *IDAS*) is that they often work with arrays of nvectors. For calls into Sundials, given OCaml arrays of OCaml nvectors, we allocate short-lived C arrays in the interface code and extract the corresponding C nvector from each element. For callbacks from Sundials, we maintain cached OCaml `arrays` in the `sensext` values and populate them with OCaml payloads extracted from the C nvectors.

3.3 Matrices

The interface for matrices must support callbacks from C into OCaml and allow direct access to the underlying data through big arrays. We adapt the solution adopted for nvectors, albeit with an extra level of 'mirroring'. An outline of our approach is shown in figure 8.

In C, a `SUNMatrix` record pairs a pointer to content with function pointers to matrix operations. The content of each matrix type is represented by a specific record type (`SUNMatrixContent_Dense`, `SUNMatrixContent_Band`, or `SUNMatrixContent_Sparse`), which is not exposed to users. The content records contain the fields described in section 2.1.3 and pointers to underlying data arrays.

In OCaml, we implement the following types for matrices.

```
type cmat
type ('mk, 'm, 'data, 'kind) t = {
  payload : 'm;
  rawptr  : cmat;
  ...
}
```

The second type is used abstractly as `('k, 'm, 'data, 'kind) Matrix.t`. The `cmat` component is a custom block pointing to the `SUNMatrix` structure allocated in the C heap; when garbage collected, its finalizer destroys the structure. The `payload` field of type `'m` refers to the OCaml representation of the matrix content. The `'mk` phantom type signifies whether the underlying representation is the standard one provided by Sundials, in which the C-side content points to a `SUNMatrixContent_*` structure, or a special form for custom matrices, in which the C-side content field is a global root referring to a set of OCaml closures for matrix operations. The other two type arguments, `'data` and `'kind`, are used to express restrictions on the matrix-times-vector operation. For instance, the built-in matrix types may only be multiplied by serial, OpenMP, and Pthreads nvectors.

In callbacks from C into OCaml, stub functions are passed `SUNMatrix` values from which they must recover a corresponding OCaml representation before invoking OCaml code. We reuse the mechanism described in section 3.1.2 for nvectors by adding a backlink field on the C side. As before, this requires overriding the clone operation to recreate the backlink and OCaml-side structures for new matrices, and the destroy operation to unregister the global root. We again prefer to avoid cross-heap cycles by not referring directly to the `Matrix.t` wrapper. Referring back to a big array would work well enough for dense matrices, but banded matrices also require tracking size and numbers of diagonals, and sparse matrices require the *indexptrs* and *indexvals* arrays described in section 2.1.3.

Our solution is to introduce an intermediate structure:

```
type ('data, 'cmat) matrix_content = {
  mutable payload : 'data;
  rawptr          : 'cmat;
}
```

which is not exposed directly by the interface, but which is rather instantiated across submodules for dense, banded, sparse, and custom matrices. For instance the `Matrix.Dense.t` type is implemented by the following definitions.

```
type data = (float, Bigarray.float64_elt, Bigarray.c_layout) Bigarray.Array2.t
type cmat
type t = (data, cmat) matrix_content
```

The `cmat` represents a custom block containing a `SUNMatrixContent_Dense` pointer to the content linked (or not) from a `SUNMatrix` structure and the `payload` field refers to a big array that wraps the data underlying the content structure. Similar instantiations are used for the `Matrix.Band.t` and `'s Matrix.Sparse.t` types, the latter includes a phantom type argument that tracks the underlying format (CSC or CSR).

Unfortunately, the scheme described here is made more complicated by the fact that certain banded and sparse operations sometimes reallocate matrix storage—for instance, if more non-zero elements are needed to store a result. The only solution we found to this problem was to override those operations by duplicating the original source code and adjusting them to create new big arrays and link their payloads back into the C-side structures.

In a custom matrix, the matrix operations are simply overloaded by stubs that retrieve an OCaml closure via the `SUNMatrix` content field and invoke it with content retrieved through the backlink.

3.4 Linear solvers

Sundials provides several different linear solvers and various options for configuring them. One of our design goals was to clarify and ensure valid configurations using the OCaml module and type systems. Our interface allows both custom and alternate linear solvers written in OCaml and cleanly accommodates parallel preconditioners without introducing a mandatory dependency on MPI.

3.4.1 Generic linear solvers

A generic linear solver essentially tries to find a vector x to satisfy an equation $Ax = b$, where A is a matrix and b is a vector. Sundials provides a single type for generic linear solvers that encompasses instances from two families, DLS and SPILS. The two families, however, are essentially implemented using different operations and attached to sessions by different functions with different supplementary arguments, for instance, *CVODE* provides `CVDlsSetLinearSolver` and `CVSpilsSetLinearSolver`. We thus found it more natural to define two distinct types, each with their own associated module.

Instances of the DLS family manipulate explicit representations of the matrix A and the vectors b and x. The type for a generic DLS is exposed as (slightly abusing syntax):

```
type ('m, 'data, 'kind, 'tag) LinearSolver.Direct.linear_solver
```

where `'m` captures the type of A, `'data` and `'kind` constrain the nvectors for b and x, and `'tag` is used to restrict the use of operations that require KLU, SuperLUMT, or custom DLS instances. Internally, the type is realized by a record that contains a pointer to the C-side structure, a reference to the matrix used within Sundials for storage and cloning (to prevent it being prematurely garbage collected), and a boolean to dynamically track and prevent associations with multiple sessions.

DLS implementations are provided for dense, banded, and sparse matrices. The function that creates a generic linear solver over dense matrices is typical:

```
val dense : 'kind Nvector_serial.any
            -> 'kind Matrix.dense
            -> (Matrix.Dense.t, 'kind, tag) serial_linear_solver
```

The resulting generic linear solver is restricted to serial, OpenMP, or Pthreads nvectors:

```
type ('m, 'kind, 'tag) serial_linear_solver
  = ('m, Nvector_serial.data, [>Nvector_serial.kind] as 'kind, 'tag) linear_solver
```

The `[>Nvector_serial.kind]` constraint only allows the type variable `'kind` to be instantiated by a type that includes the constructor `Nvector_serial.kind`, which was presented on page 115.

A custom DLS implementation is created by defining a record of OCaml functions:

```
type ('m, 'data, 's) ops = {
  init  : 's -> unit;
  setup : 's -> 'm -> unit;
  solve : 's -> 'm -> 'data -> 'data -> unit;
  get_work_space : ('s -> int * int) option;
}
```

where `'s` is the type of the internal state of an instance. The following two functions are provided.

```
val make : ('m, 'data, 's) ops -> 's -> ('mk, 'm, 'data, 'kind) Matrix.t
           -> ('m, 'data, 'kind, [`Custom of 's]) linear_solver
val unwrap : ('m, 'data, 'kind, [`Custom of 's]) linear_solver -> 's
```

The first takes a set of operations, an initial state, and a storage matrix, and returns a DLS instance. The matrix kind `'mk`, indicating whether the matrix implementation is standard or custom, need not be propagated to the result type since callbacks only receive the matrix contents of type `'m`. The tag type argument indicates a custom linear solver whose internal state has the given type. This tag allows for a generic `unwrap` function and thereby avoids requiring users to maintain both a custom state—to set properties or get statistics—and an actual instance.

Instances of the SPILS family rely on a function that approximates a matrix-vector product to model the (approximate) effect of the matrix A without representing it explicitly. The success of this approach typically requires the solving a preconditioned system that results from scaling the matrix A and vectors b and x, and multiplying them by problem-specific matrices on the left, right, or both sides.

The type for a generic SPILS is exposed as:

```
type ('data, 'kind, 'tag) LinearSolver.Iterative.linear_solver
```

where `'data` and `'kind` constrain the nvectors used and `'tag` is used to restrict operations for specific iterative methods. The internal realization of this type and the creation of custom linear solvers is essentially the same as for the DLS module. The following SPILS instantiation function is typical.

```
val spgmr : ?maxl:int
            -> ?max_restarts:int
            -> ?gs_type:gramschmidt_type
            -> ('data, 'kind) Nvector.t
            -> ('data, 'kind, [`Spgmr]) linear_solver
```

It takes three optional arguments to configure the linear solver and an nvector to specify the problem size and compatibility constraints.

3.4.2 Associating generic linear solvers to sessions

Generic linear solvers are associated with sessions after having created the session and before simulating it. In the OCaml interface, we incorporated these steps into the `init` functions that return solver sessions, as shown in figure 2 and applied in section 2.3.2. This allows us to enforce required typing constraints and ensure that calls to the underlying library are made in the correct order. We introduce intermediate types to represent the combination of a generic linear solver with its session-specific parameters and to group diagonal approximation, DLS, SPILS, and alternate modules. For instance, in the *CVODE* solver, we declare:

```
type ('data, 'kind) session_linear_solver
```

which is realized internally by a function over a session value and an nvector, and acts imperatively to configure the session. Values of this type are provided by solver-specific submodules whose particularities we now summarize. For our purposes, the important thing is not what the different modules do, but rather how the constraints on their use are expressed in Sundials/ML.

Diagonal linear solvers. The *CVODE* diagonal linear solver is interfaced by the submodule:

```
module Diag : sig
  val solver : ('data, 'kind) session_linear_solver
  val get_work_space : ('data, 'kind) session -> int * int
  val get_num_rhs_evals : ('data, 'kind) session -> int
end
```

A `Cvode.Diag.solver` value is passed to `Cvode.init` or `Cvode.reinit` where it is invoked and makes calls to the Sundials `CVodeSetIterType` and `CVDiag` functions that set up the diagonal linear solver. The `get_*` functions retrieve statistics specific to the diagonal linear solver—here the memory used by the diagonal solver in terms of real and integer values, or the number of times that the right-hand side callback has been invoked. Other linear solvers also provide `set_*` functions. As the underlying implementations of these functions sometimes typecast memory under the assumption that the associated linear solver is in use, we implement dynamic checks that throw an exception when a session is configured with one linear solver and passed to a function that assumes another. This constraint cannot be adequately expressed using static types since a session may be dynamically reinitialized with a different linear solver.

Direct Linear Solvers (DLS). Interfacing DLS requires treating nvector compatibility and the matrix data structures passed to callback functions. For instance, the `Cvode.Dls` submodule contains the value:

```
val solver : ?jac:'m jac_fn ->
             ('m, 'kind, 'tag) LinearSolver.Direct.serial_linear_solver ->
             'kind serial_session_linear_solver
```

where `serial_session_linear_solver` is another abbreviation for restricting nvectors.

```
type 'kind serial_session_linear_solver =
  (Nvector_serial.data, [>Nvector_serial.kind] as 'kind) session_linear_solver
```

The `?jac` label marks a named optional argument. It is used to pass a function that calculates an explicit representation of the system Jacobian matrix. The only relevant detail here is the use of `'m` to ensure that the same matrix type is used by the callback function and the generic linear solver. Similarly, `'kind` is propagated to the result type to ensure nvector compatibility when a session is created.

Each solver has its own `Dls` submodule into which the types and values of `LinearSolver.Direct` are imported to maximize the effect of the local open syntax—for instance, as in the call to `Ida.init` in the example of section 2.3.2.

Scaled Preconditioned Iterative Linear Solvers (SPILS). A SPILS associates an iterative method with a preconditioner. Iterative methods are exposed as functions that take an optional Jacobian multiplication function and a preconditioner, for example,

```
val solver :
  ('data, 'kind, 'tag) LinearSolver.Iterative.linear_solver
  -> ?jac_times_vec:'data jac_times_setup_fn option * 'data jac_times_vec_fn
  -> ('data, 'kind) preconditioner
  -> ('data, 'kind) session_linear_solver
```

As is clear from the type signature, it is the preconditioner that constrains nvector compatibility. Internally the `preconditioner` type pairs a preconditioning "side" (left, right, both, or none) with a function that configures a preconditioner given a session and an nvector. Functions are provided to produce elements of this type. For instance, the `Cvode.Spils` module provides:

```
val prec_none : ('data, 'kind) preconditioner
val prec_left  : ?setup:'data prec_setup_fn
                 -> 'data prec_solve_fn
                 -> ('data, 'kind) preconditioner
val prec_right : ?setup:'data prec_setup_fn
                 -> 'data prec_solve_fn
                 -> ('data, 'kind) preconditioner
val prec_both  : ?setup:'data prec_setup_fn
                 -> 'data prec_solve_fn
                 -> ('data, 'kind) preconditioner
```

The last three produce preconditioners from optional setup functions and mandatory solve functions over the nvector payload type. These preconditioners are compatible with any type of nvector.

Banded preconditioners, on the other hand, are only compatible with serial, OpenMP, and Pthreads nvectors. We group them into a submodule `Cvode.Spils.Banded`:

```
val prec_left  : bandrange
                 -> (Nvector_serial.data, [> Nvector_serial.kind]) preconditioner
val prec_right : bandrange
                 -> (Nvector_serial.data, [> Nvector_serial.kind]) preconditioner
val prec_both  : bandrange
                 -> (Nvector_serial.data, [> Nvector_serial.kind]) preconditioner
```

The banded preconditioners provide their own setup and solve functions.

The Band-Block-Diagonal (BBD) preconditioner is only compatible with parallel nvectors. Its `'data` type variable is instantiated to `Nvector_parallel.data`, the payload of parallel nvectors, and its `'kind` type variable to `Nvector_parallel.kind`. The declarations are made in a separate module `Cvode_bbd`, which is simply not compiled when MPI is not available.

Each solver has a `Spils` submodule into which the types and values of `LinearSolver.Iterative` are imported to maximize the effect of the local open syntax—as shown, for instance, in figure 2.

4 Evaluation

An interface layer inevitably adds run-time overhead: there is extra code to execute at each call to, or callback from the library. This section presents our attempt to quantify this overhead. Since we are interested in the cost of using Sundials from programs written in OCaml, rather than count the number of additional instructions per call or callback we think it more relevant to compare the performance of programs written in OCaml with equivalent programs written directly in C. We consider two programs equivalent when they produce identical sequences of output bytes using the same sequence of solver steps in the Sundials library. Here we compare wall clock run times, which, despite the risk of interference from other processes and environmental factors, have the advantages of being relatively simple to measure and directly relevant to users.

Sundials is distributed with example programs (71 with serial, OpenMP, or Pthreads nvectors and 21 with parallel nvectors—not counting duplicates) that exercise a wide range of solver features in numerically interesting ways. We translated them all into OCaml.[6] Comparing the outputs of corresponding OCaml and C versions with `diff` led us to correct many bugs in our interface and example implementations, and even to discover and report several bugs in Sundials itself. We also used `valgrind` [16] and manual invocations of the garbage collector to reveal memory-handling errors in our code.

After ensuring the equivalence of the example programs, we used them to obtain and optimize performance results. As we explain below, most optimizations were in the example programs themselves, but we were able to validate and evaluate some design choices, most notably the alternative proposal for sessions described in section 3.2. The bars in figures 9 and 10 show the ratios of the execution times of the OCaml code against the C code.[7] A value of 2.0 on the horizontal axis means that the OCaml version takes twice as long as the C version to calculate the same result.

The extent of the bars show 99.5% confidence intervals for the OCaml/C ratio, calculated according to Chen et al.'s technique [6]. Formally, if O and C are random variables representing the running time of a given test in OCaml and C, respectively, then the bars show the range of all γ for which the Mann-Whitney U-test (also known as the Wilcoxon rank-sum test) does *not* reject the null hypothesis $P(\gamma C > O) = P(\gamma C < O)$ at the 99.5% confidence level. Intuitively, if we handicap the C code by scaling its time by γ, then the observed measurements are consistent with the assumption that the handicapped C code beats the non-handicapped OCaml code exactly half of the time: the observed data may be skewed in one direction or the other, but the deviation from parity is smaller than random noise would give 99.5% of the time if they really were equally competitive.

The results include customized examples: the kinFoodWeb_kry_custom example uses custom

[6] The translations aim to facilitate direct comparison with the original code, to ease debugging and maintenance. They are not necessarily paragons of good OCaml style.

[7] The Sundials/ML source code distribution includes all examples along with the scripts and build targets necessary to reproduce the experiments described in this paper.

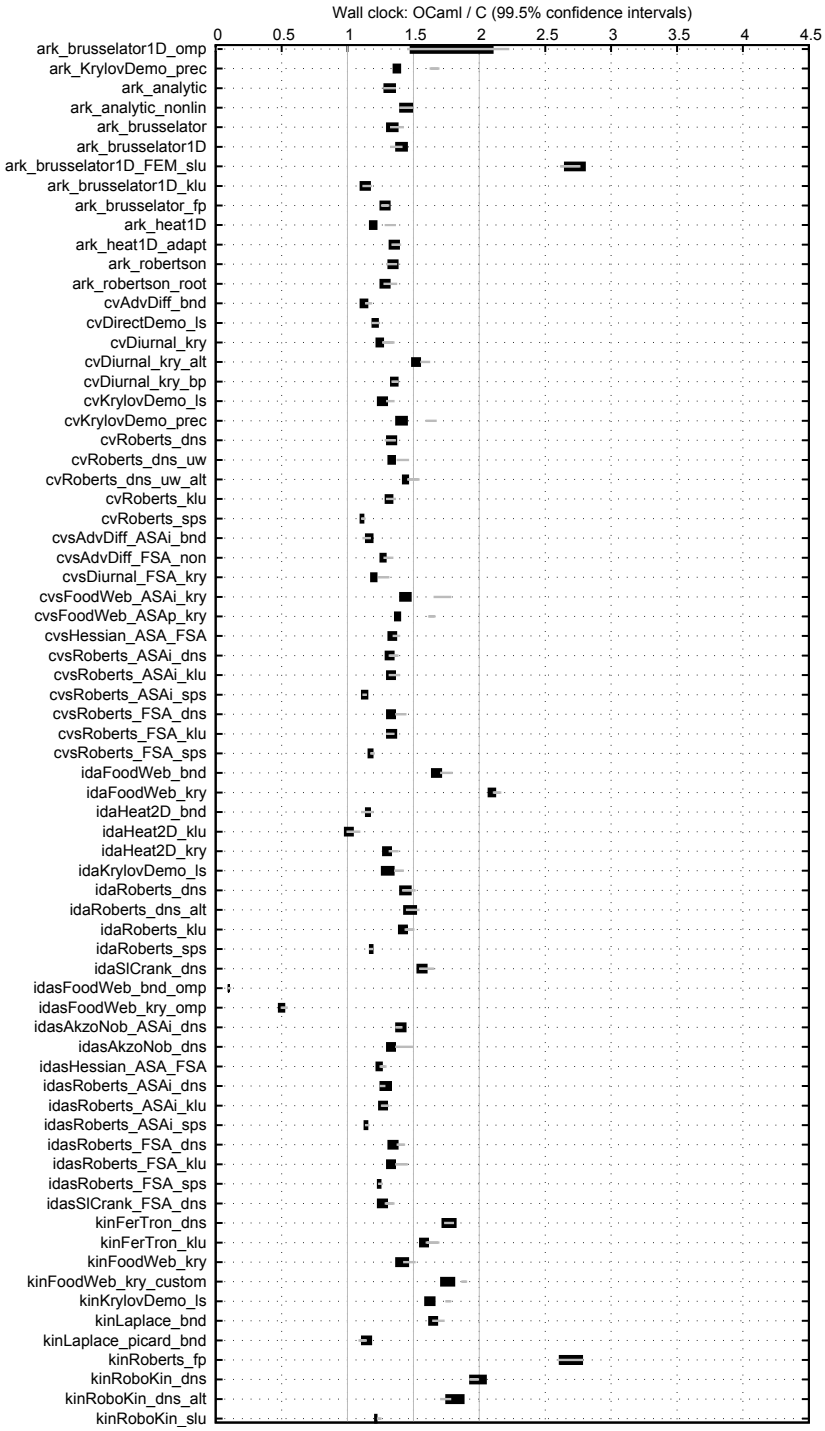


Figure 9: Serial examples: C (gcc 6.3.0 with -O3) versus OCaml native code (4.07.0). The black bars show results obtained with the `-unsafe` option that turns off array bounds checking and other dynamic checks. The grey lines show the results with dynamic checks. Results were obtained under Linux 4.9.0 on an Intel i7 running at 2.60 GHz with a 1 MB L2 cache, a 6 MB L3 cache, and 8 GB of RAM. Sundials was compiled with `CMAKE_BUILD_TYPE=Release`.

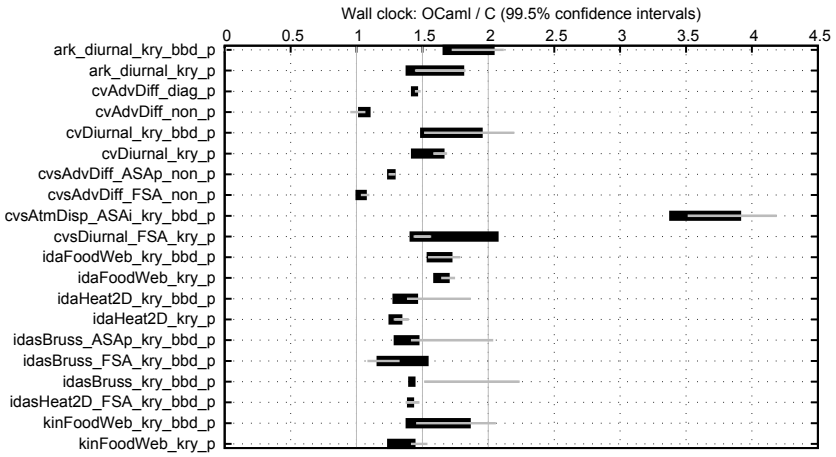


Figure 10: Parallel examples: C (gcc 6.3.0 with -O3) versus OCaml native code (4.07.0). The black bars show results obtained with the `-unsafe` option that turns off array bounds checking and other dynamic checks. The grey lines show the results with dynamic checks. Results were obtained under Linux 4.9.0 on an Intel i7 running at 2.60 GHz with a 1 MB L2 cache, a 6 MB L3 cache, and 8 GB of RAM. Sundials was compiled with `CMAKE_BUILD_TYPE=Release`.

nvectors with low-level operations implemented in OCaml on `float arrays`; the `*_alt` examples use an alternate linear solver reimplemented in OCaml using the underlying `Dls` binding. This involves calls from OCaml to C to OCaml to C. Each custom example produces the same output as the corresponding original (their predecessors in the graph). Two OCaml versions, idasFoodWeb_bnd_omp and idasFoodWeb_kry_omp are faster than the C versions—we explain why below. The black bars in figures 9 and 10 give the ratios achieved when the OCaml versions are compiled without checks on array access, nvector compatibility, or matrix validity (since the C code does not perform these checks). The grey lines show the results with dynamic checks; the overhead is typically negligible. The graph suggests that the OCaml versions are rarely more than 50% slower than the original ones and that they are often less than 20% slower. That said, care must be taken extrapolating from the results of this particular experiment to the performance of real applications, which will not be iterated 1000s of times and where the costs of garbage collection can be minimized given sufficient memory.

The actual C run times are not given in figures 9 and 10. Most of them are less than 1 ms, nearly all of them are less than 1 s, the longest is on the order of 4 s (ark_diurnal_kry_bbd_p). We were not able to profile such short run times directly: the time and gprof commands simply show 0 s. The figures in the graph were obtained by modifying each example (in both C and OCaml) to repeatedly execute its `main` function. Since the C code calls `free` on all of its data, we also manually trigger a full major collection and heap compaction in OCaml at the end of the program (the ratios are smaller otherwise). The figure compares the median ratios over 10 such executions. The fastest examples require 1000s of iterations to produce a measurable result, so we must vary the number of repetitions per example to avoid the slower examples taking too long (several hours each). Iterating the examples so many times sometimes amplifies factors other than interface overhead.

For the two examples where OCaml apparently performs better than C, the original C code includes OpenMP pragmas around loops in the callback functions. This actually slows them down and the OCaml code does better because this feature is not available.

In general, we were able to make the OCaml versions of the examples up to four times faster by following three simple and unsurprising guidelines.

1. We added explicit type annotations to all vector arguments. For instance, rather than declare a callback with

   ```
   let f t y yd = ...
   ```

   we follow the standard approach of adding type annotations,

   ```
   type rarray = (float, float64_elt, c_layout) Bigarray.Array1.t
   let f t (y : rarray) (yd : rarray) = ...
   ```

   so that the compiler need not generate polymorphic code and can optimize for the big array layout.

2. We avoided the functions `Bigarray.Array1.sub`, `Bigarray.Array2.slice_left` that allocate fresh big arrays on the major heap and thereby increase the frequency and cost of garbage collection. They can usually be avoided by explicitly passing and manipulating array offsets. We found that when part of an array is to be passed to another function, as for some MPI calls, it can be faster to copy into and out of a temporary array.

3. We wrote numeric expressions and loops according to Leroy's [12] advice to avoid float 'boxing'.

As a side benefit of the performance testing, iterating each example program 1000s of times with some control over the garbage collector revealed several subtle memory corruption errors in our interface. We investigated and resolved these using manual code review and a C debugger.

In summary, the results obtained, albeit against an admittedly small set of examples, indicate that OCaml code using the Sundials solvers should rarely be more than 50% slower than equivalent code written in C, provided the guidelines above are followed, and it may be only about 20% slower. One response to the question "Should I use OCaml to solve my numeric problem?" is to rephrase it in terms of the cost of calculating results ("How long must I wait?") against the cost of producing and maintaining programs ("How much effort will it take to write, debug, and later modify a program?"). This section provides insight into the former cost. The latter cost is difficult to quantify, but it is arguably easier to write and debug OCaml code thanks to automatic memory management, strong static type checking, bounds checking on arrays, and higher-order functions. This combined with the other features of OCaml—like algebraic data types, pattern matching, and polymorphism—make the Sundials/ML library especially compelling for programs that combine numeric calculation and symbolic manipulation.

5 Conclusion

We present a comprehensive OCaml interface to the Sundials suite of numeric solvers. We outline the main features of the underlying library, demonstrate its use via our interface, describe the implementation in detail, and summarize extensive benchmarking.

Sundials is an inherently imperative library. Data structures like vectors and matrices are reused to minimize memory allocation and deallocation, and modified in place to minimize copying. It works well in an ML-style language where it is possible to mix features of imperative programming—like sequencing, loops, mutable data structures, and exceptions—with those of functional programming—like higher-order functions, closures, and algebraic data types. An interesting question that we did not treat is how to build efficient numerical solvers in a more functional style.

It turns out that the abstract data types used to structure the Sundials library are also very useful for implementing a high-level interface. By overriding elements of these data structures, namely the clone

and destroy operations, we are able to smoothly integrate them with OCaml's automatic memory management. Designers of other C libraries that intend to support high-level languages might also consider this approach. For Sundials, some minor improvements are possible—for instance, adding (i) a user data field to nvectors and matrices that could be exploited for 'backlinks', (ii) a function to return the address of the session user data field so that it can be registered directly with the garbage collector, and (iii) a mechanism for overriding the reallocation mechanism within banded and sparse matrices to eliminate the need to reimplement certain operations,—but the approach works well overall.

In our interface, C-side structures are mirrored by OCaml structures that combine low-level pointers, their associated finalize functions, and high-level values. This is a standard approach that we adapted for two particular requirements, namely, to give direct access to low-level arrays and to treat callbacks efficiently. The first requirement is addressed by combining features of the OCaml big array library with the ability to override the Sundials clone and destroy operations. The second requirement necessitates a means to obtain the OCaml "mirror" of a given instance of a C data structure. The backlink fields solve this problem well provided care is taken to avoid inter-heap cycles. Besides the usual approach of using weak references, we demonstrate an alternative for when the mirrored structures are "wrappers". In this case, the pointer necessary to recover an OCaml structure from C drops a level in the wrapper hierarchy. This is a simple solution to the cycle problem that is also convenient for library users. We note that it engenders two kinds of instance: those created in OCaml and those cloned in C. For instances created in OCaml and used in C, care must be taken to maintain a reference from OCaml until the instance is no longer needed as the backlink itself does not prevent garbage collection and ensuing instance destruction. For instances created in C and passed back into OCaml, there is no such problem provided the destruction function only unregisters the global root and does not actually destroy the instance.

Our work relies heavily on several features of the OCaml runtime, most notably flexible functions for creating and manipulating big arrays, the ability to register global garbage collection roots in structures on the C heap, features for invoking callbacks and recovering exceptions from them, and the ability to associate finalize functions to custom blocks. We found phantom types combined with polymorphic variants are a very effective way to express constraints on the use and combination of data structures that arise from the underlying library. While Generalized Abstract Data Types (GADTs) are used in our implementation, they are not exposed through the interface, probably because in this library we prefer opaque types to variant types since they permit a better division into submodules.

The evaluation results show that the programming convenience provided by OCaml and our library has a measurable runtime cost. The overall results, however, are not as bad as may perhaps be feared. We conjecture that, in most cases, the time gained in programming, debugging, and maintenance will outweigh the cumulative time lost to interface overhead. Since programs that build on Sundials typically express sophisticated numerical models, ensuring their readability through high-level features and good style is an important consideration. Ideally, programming languages and their libraries facilitate reasoning about models, even if only informally, and precisely communicating techniques, ideas, and knowledge. We hope our library contributes to these aims.

Acknowledgements

We thank Kenichi Asai and Mark Shinwell for their persistence and thoroughness, and the anonymous reviewers for their helpful suggestions. The work described in this paper was supported by the ITEA 3 project 11004 MODRIO (Model driven physical systems operation).

References

[1] Albert Benveniste, Paul Caspi, Stephen Edwards, Nicolas Halbwachs, Paul Le Guernic & Robert de Simone (2003): *The Synchronous Languages 12 Years Later*. *Proc. IEEE* 91(1), pp. 64–83, doi:10.1109/JPROC.2002.805826.

[2] Timothy Bourke & Marc Pouzet (2013): *Zélus: A Synchronous Language with ODEs*. In Calin Belta & Franjo Ivancic, editors: *Proc. 16th Int. Conf. on Hybrid Systems: Computation and Control (HSCC 2013)*, ACM Press, Philadelphia, USA, pp. 113–118, doi:10.1145/2461328.2461348.

[3] David Broman & Jeremy G. Siek (2012): *Modelyze: a Gradually Typed Host Language for Embedding Equation-Based Modeling Languages*. Technical Report UCB/EECS-2012-173, University of California, Berkeley, United States. Available at `https://www2.eecs.berkeley.edu/Pubs/TechRpts/2012/EECS-2012-173.html`.

[4] Daniel Bünzli (2005): *Bigarrays and temporar[y] C pointers*. Caml-list mailing list. Available at `http://caml.inria.fr/pub/ml-archives/caml-list/2005/01/ce57c7689f1b7d0fa60514937757d9da.en.html`.

[5] Emmanuel Chailloux, Pascal Manoury & Bruno Pagano (2000): *Développement d'applications avec Objective Caml*. O'Reilly France. Available at `https://caml.inria.fr/pub/docs/oreilly-book/`.

[6] T. Chen, Q. Guo, O. Temam, Y. Wu, Y. Bao, Z. Xu & Y. Chen (2015): *Statistical Performance Comparisons of Computers*. *IEEE Transactions on Computers* 64(5), pp. 1442–1455, doi:10.1109/TC.2014.2315614.

[7] Hans Fangohr, Thomas Fischbacher, Matteo Franchin, Giuliano Bordignon, Jacek Generowicz, Andreas Knittel, Michael Walter & Maximilian Albert (2012): *NMAG User Manual Documentation*. University of Southhampton, England. Available at `http://nmag.soton.ac.uk/nmag/index.html`. Release 0.2.1.

[8] Jacques Garrigue (1998): *Programming with polymorphic variants*. In: *The 1998 ACM SIGPLAN Workshop on on ML*, ACM, Baltimore, MD, USA. Available at `https://caml.inria.fr/pub/papers/garrigue-polymorphic_variants-ml98.pdf`.

[9] Alan C. Hindmarsh, Peter N. Brown, Keith E. Grant, Steven L. Lee, Radu Serban, Dan E. Shumaker & Carol S. Woodward (2005): *SUNDIALS: Suite of Nonlinear and Differential/Algebraic Equation Solvers*. *ACM Trans. Mathematical Software* 31(3), pp. 363–396, doi:10.1145/1089014.1089020.

[10] Alan C. Hindmarsh, Radu Serban & Daniel R. Reynolds (2017): *User Documentation for* CVODE *v3.1.0*, v3.1.0 edition. Lawrence Livermore National Laboratory, Livermore, CA, USA. Available at `https://computation.llnl.gov/sites/default/files/public/cv_guide.pdf`.

[11] Daan Leijen & Erik Meijer (1999): *Domain Specific Embedded Compilers*. In: *2nd Conference on Domain-Specific Languages (DSL'99)*, ACM, Austin, TX, USA, pp. 109–122, doi:10.1145/331960.331977.

[12] Xavier Leroy (2002): *Writing efficient numerical code in Objective Caml*. `http://caml.inria.fr/pub/old_caml_site/ocaml/numerical.html`.

[13] Xavier Leroy, Damien Doligez, Alain Frisch, Jacques Garrigue, Didier Rémy & Jérôme Vouillon (2018): *The OCaml system: Documentation and user's manual*, 4.07 edition. Inria. Available at `http://caml.inria.fr/pub/docs/manual-ocaml/`.

[14] Yaron Minsky, Anil Madhavapeddy & Jason Hickey (2013): *Real World OCaml: Functional programming for the masses*. O'Reilly. Available at `https://v1.realworldocaml.org`.

[15] Florent Monnier (2013): *How to mix OCaml and C code*. Web page. Available at `https://www.linux-nantes.org/~fmonnier/OCaml/ocaml-wrapping-c.html`.

[16] Nicholas Nethercote & Julian Seward (2007): *Valgrind: a framework for heavyweight dynamic binary instrumentation*. In: *Proc. 2007 ACM SIGPLAN Conf. on Programming Language Design and Implementation*, ACM Press, San Diego, CA, USA, pp. 89–100, doi:10.1145/1273442.1250746.