“D.4.2.16 – OCaml interface to the Sundials suite of numerical solvers”

WP 4.2 – Tool support

WP4 – Systems with multiple operating modes

MODRIO (11004)

Version 1.1

Date 19/01/2016

Authors

Timothy Bourke	Inria
Jun Inoue	Inria
Marc Pouzet	Inria

Executive summary

This document summarizes the status at M42 of deliverable D4.2.16 "OCaml interface to the Sundials suite of numerical solvers". The objective of this deliverable is to implement an open source library that permits OCaml programmers to build upon numerical solvers developed at the Lawrence Livermore National Laboratory in the United States.

Parts of this new library are used in the Inria prototype language. But it would greatly facilitate any development (in OCaml) of software that mixes symbolic manipulations and numerical code; like compilers and interpreters for hybrid modeling languages like Modelica.

This document describes the technical choices and innovations that were made during the development of the OCaml interface in the context of the MODRIO project.

The Sundials documentation and source-code is available at http://inria-parkas.github.io/sundialsml/ under a BSD license.

Summary

1. Introduction

Sundials [1] is a library of five numerical solvers: CVODE, for Ordinary Differential Equations (ODEs), CVODES, for ODEs with sensitivity analysis, IDA, for Differential Algebraic Equations (DAEs), IDAS, for DAEs with sensitivity analysis, and KINSOL, for non-linear equations. The five solvers share data structures and operations for vectors and linear algebra routines. They are implemented in C.

In this document, we describe the design and implementation of a comprehensive OCaml [2] interface to the Sundials library: *Sundials/ML*.

The authors of Sundials describe a "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 [3](§1.1). So, why bother writing an interface from OCaml? We think that OCaml is ideal for

1. programs that mix numeric computation with symbolic manipulation, like interpreters for hybrid modeling languages, and perhaps also for prototype compilers—even if 'real' compilers must likely generate C;
2. directly prototyping numerical models, and;
3. for incorporating numerical approximation into more complicated applications.

Compared to C, OCaml detects many mistakes through a combination of static (strong typing) and dynamic (array bounds, vector compatibility) checks, manages memory automatically using garbage collection, and propagates errors as exceptions rather than return codes. In addition, Sundials/ML streamlines solver configuration using algebraic data types, as opposed to inter-dependent function calls, and clarifies some library features using features of OCaml's module and type systems. On the other hand, any such interface adds additional code and thus runtime overhead and the possibility of bugs—we discuss these issues in section 3, —and OCaml is less portable than C.

1.1. Overview of Sundials

This section briefly outlines the basic structure of the Sundials library. We limit ourselves to the elements necessary to explain and justify subsequent technical points. A mathematical description of the library can be found in [1]. The user manuals[1] give a thorough overview of the library and the details of every function.

The purposes of the three 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 $\dot{Y}(t_0) = \dot{Y}_0$.

KINSOL calculates U from $F(U) = 0$ and initial guess U_0.

The problems are stated over vectors Y and U.

The first two solvers find solutions that depend on the variable t, usually considered to be the simulation time. The remaining solvers, 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 $S(t) = \frac{\partial Y(t)}{\partial p}$.

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

Solver sessions. Using any of the Sundials solvers involves the same basic pattern:

1. a session object is created;
2. several inter-dependent functions are called to initialize the session;
3. 'set' functions are called to give parameter values;
4. a 'solve' or 'step' function is called (iteratively) to find or approximate a solution;
5. 'get' functions are called to retrieve results;

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

6. the session and related data structures are freed.

The sequence in Figure 1, extracted from an example distributed with Sundials, is typical. The first line creates a session with the (CVODE) solver, an abstract type implemented as a pointer to a structure containing solver parameters and state, which is passed to all subsequent calls.

This function call, and all the others, are followed by statements that check return codes. Line 4 specifies a 'user data' pointer that is passed by the solver to all callback functions to provide session-local storage. Line 7 specifies a callback function *f*, that defines the problem to solve, here a function from a vector of variable values to their derivatives ($\dot{Y} = f(Y)$), and a vector of initial variable values *u* that implicitly defines the problem size ($Y_0 = u$). The other calls specify tolerance values, configure an iterative linear solver with callback functions defining the problem Jacobian, *jtv*, and preconditioning, *Precond* and *PSolve*. The loop that calculates the value of X over time, and the functions that retrieve solver results and statistics, and free memory are not shown.

While the IDA and KINSOL solvers follow almost exactly the same pattern, the CVODES and IDAS solvers are a bit more involved. They 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 DAE problem by first integrating normally, and then initializing new 'backward' sessions that are integrated in the reverse direction.

The function calls that initialize and configure solver sessions are subject to rules constraining their presence, order, and parameters. For instance, in the example of Figure 1, the call to *CVodeSStolerances* must follow the call to *CVodeInit* and obligatorily precede any call to the step function, calling *CVodeCreate* with the parameter *CV_NEWTON* necessitates later calls to configure a linear solver, and the *CVSpgmr* linear solver requires a call to *CVSpilsSetPreconditioner* with at least a *PSolve* value.

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 with pointers to 25 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. Solvers 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.

Two instantiations of the nvector type are provided by Sundials and shared across solvers: serial nvectors and parallel nvectors. Serial nvectors store and manipulate arrays of floats. Parallel nvectors store a local array of floats and a *Message Passing Interface* (MPI) communicator; some operations, like nvlinearsum, simply loop over the local array, others, like nvmaxnorm, calculate locally and then synchronize via a global reduce. Users[2] may also provide their own nvector implementations.

Linear solvers. Each of the solvers must resolve nonlinear algebraic systems. For this, they use either 'functional iteration' (CVODE and CVODES only), or, more usually, 'Newton iteration' [3](§2.1). Newton iteration, in turn, requires the solution of linear systems.

Several linear solver modules are provided with Sundials. Internally, they are called through a generic interface comprising four function pointers in a session object: *linit*, *lsetup*, *lsolve*, and *lfree* [3](§7). Users may also provide their own *alternate* linear solver modules by directly setting the four pointers.

Sundials includes three main linear solver modules: a diagonal approximation of system Jacobians using difference quotients, *Direct Linear Solver* (DLS) modules that perform LU factorization on dense and banded matrices, and *Scaled Preconditioning Iterative Linear Solver* (SPILS) modules based on Krylov methods. Distinct implementations are provided for each solver, since the types of user-supplied callback functions and internal details change, but they are all based on centralized implementations of basic linear algebra and SPILS routines.

[2] In this document, a 'user' is a programmer using the presented OCaml interface.

Just as 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 nvectors—they cannot be used with parallel or custom nvectors. The SPILS modules combine a specific method (SPGMR, SPBCG, or SPTFQMR) with an optional preconditioner. There are standard preconditioners (left, right, or both) for which users supply a solve function, and, optionally, setup and Jacobian-times-vector functions, and which work for any type of nvector, and also a band matrix preconditioner that is only compatible with serial nvectors, and a *Band Block Diagonal* (BBD) preconditioner that is only compatible with parallel nvectors.

```
1  cvode_mem = CVodeCreate(CV_BDF, CV_NEWTON);
2  if(check_flag((void *)cvode_mem, "CVodeCreate", 0)) return(1);
3
4  flag = CVodeSetUserData(cvode_mem, data);
5  if(check_flag(&flag, "CVodeSetUserData", 1)) return(1);
6
7  flag = CVodeInit(cvode_mem, f, T0, u);
8  if(check_flag(&flag, "CVodeInit", 1)) return(1);
9
10 flag = CVodeSStolerances(cvode_mem, reltol, abstol);
11 if (check_flag(&flag, "CVodeSStolerances", 1)) return(1);
12
13 flag = CVSpgmr(cvode_mem, PREC_LEFT, 0);
14 if(check_flag(&flag, "CVSpgmr", 1)) return(1);
15
16 flag = CVSpilsSetJacTimesVecFn(cvode_mem, jtv);
17 if(check_flag(&flag, "CVSpilsSetJacTimesVecFn", 1)) return(1);
18
19 flag = CVSpilsSetGSType(cvode_mem, MODIFIED_GS);
20 if(check_flag(&flag, "CVSpilsSetGSType", 1)) return(1);
21
22 flag = CVSpilsSetPreconditioner(cvode_mem, Precond, PSolve);
23 if(check_flag(&flag, "CVSpilsSetPreconditioner", 1)) return(1);
```

Figure 1: Initialization C version (from cvode/serial/cvDiurnal_kry)

```
1  let cvode_mem =
2    Cvode.init Cvode.BDF
3      (Cvode.Newton (Cvode.Spils.spgmr
4          (Cvode.Spils.prec_left ~setup:(precond data)
5                                 ~jac_times_vec:(jtv data)
6                                 (psolve data))))
7      (Cvode.SStolerances (reltol, abstol))
8      (f data) t0 u
9  in
10 Cvode.Spils.set_gs_type cvode_mem Spils.ModifiedGS;
```

Figure 2: Initialization OCaml version (from cvode/serial/cvDiurnal_kry)

1.2. Overview of Sundials/ML

The structure of the OCaml interface mostly follows that of the underlying library. 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:

1. solver sessions are mostly configured via algebraic data types rather than multiple function calls;
2. errors are signaled by exceptions, not return codes (also from within user-supplied callback routines);
3. user data is shared between callback routines via closures (partial applications of functions);
4. vectors are checked for compatibility using a combination of static and dynamic checks; and
5. explicit free commands are not necessary since OCaml is a garbage-collected language.

The features described above are evident in the program extract of Figure 2, which is equivalent to the C code of Figure 1. Note that functions 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.

In the example, for instance, *CVSpilsSetGSType* becomes *Cvode.Spils.set_gs_type*. Similar changes are made for constants and return codes (which usually become exceptions). In one sense, such naming details are mundane, but handling them poorly can complicate the maintenance and use of the interface and make it difficult to refer to materials like documentation and example code written for the original library.

In the OCaml version of Figure 2, rather than specifying Newton iteration by passing a constant (*CV_NEWTON*) and later calling linear solver routines (*CVSpgmr*, etcetera), a solver session is configured by passing a value that contains all the necessary parameters. This makes it impossible, for instance, 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 callback functions—*precond*, *jtv*, *psolve*, and *f*—are all applied to *data*. This use of partial application over a shared data value replaces the 'user data' mechanism of Sundials (*CVodeSetUserData*); it is more natural in OCaml and it frees the underlying user data mechanism for use by the interface code.

Not all options are configured at session creation, that is, by the call to *init*. Others are set via later calls, like that to *Cvode.Spils.set_gs_type* in Figure 2. In choosing between the two, 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_gs_type*, on the other hand, are better made separately since the default values usually suffice and since it often makes sense to change such settings between calls to the solver.

The example code treats a session with CVODE, but the IDA and KINSOL interfaces work similarly. The CVODES and IDAS solvers are fundamentally different. One of the guiding principles of these solvers is that their extra features be accessible simply by adding extra calls to existing programs [4](§5) and linking with a different library, since the underlying code base is distinct even if much of the interface and functionality is unchanged. We respected 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 to calculate their derivatives, *fq*, and their initial values, *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. Like Sundials itself, we rely on runtime checks to detect when features are used without having been correctly initialized.

To calculate sensitivities using the adjoint method, a user first 'enhances' a solver session (*Cvodes.Adjoint.init*) then calculates a solution by taking steps in the 'forward' direction, before attaching and initializing 'backward' sessions, and then taking steps in the opposite direction. A backward session is identified in Sundials by an integer. Most backward-specific functions take as arguments the forward session and the integer, but other features are accessed by acquiring a backward session object with *CVodeGetAdjCVodeBmem* [4](§6.2.9) and then passing it to the standard functions. The interface hides these details behind 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 2.2.

The details of the interfaces to nvectors and linear solvers are deferred to sections 2.1 and 2.3 since the choices made and the types used are closely tied to the technical details of their respective 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. They also help users by formalizing the rules for using linear solvers.

Sundials is a large and sophisticated library. We thus made a special effort to provide high-quality documentation. The existing features of ocamldoc [2] and the ability to add new tags were invaluable. For instance, we combined the tags for including LaTeX markup with the MathJax library[3] to render mathematical descriptions inline. And, rather than duplicate the extensive Sundials documentation, we used custom tags to insert hyperlinks back to it.

2. Technical details

We now describe the typing and implementation technicalities of 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 the OCaml manual [2](Chapter 19). We focus here on the nvector (section 2.1), session (section 2.2), and linear solver (section 2.3) datatypes, describing some of our early attempts to treat them and presenting the solutions we finally arrived at and their justifications. We also describe the treatment of Jacobian matrices (section 2.4) and some technicalities of linking with the underlying libraries (section 2.5).

2.1. Vectors

Serial nvectors combine an array of floating-point numbers (*double*s) with implementations of the 25 vector operations. The OCaml big array library [2](Chapter 28) provides arrays of floating-point numbers that can be shared directly between OCaml and C, and it is thus the natural choice for interfacing with the contents of serial nvectors.

A first idea for an interface is to work only with big arrays on the OCaml side of the library, and to transparently convert them to nvectors on calls into Sundials, and from nvectors on calls from Sundials. We did exactly this in early versions of our interface. For calls into Sundials, like *Cvode.init* from the earlier example, we defined a macro to take a big array *b* and create a new serial nvector:

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

The first argument expression extracts the array size, the second returns the address of the underlying data array. Operations on the resulting nvector directly manipulate the data stored in the big array. Another macro, called just before returning from the wrapper code, frees the

nvector argument *v*:

```
N_VDestroy(v)
```

For callbacks into OCaml, we defined a macro to create a big array from an nvector argument *v*:

```
caml_ba_alloc(BIGARRAY_FLOAT, 1, NV_DATA_S(v), &(NV_LENGTH_S(v)))
```

The first argument specifies the layout; the second the number of dimensions, the third passes a pointer to the nvector data, and the fourth gives the length of the array.[4] Again, changes to the big array are made directly on the underlying array. After a callback, another macro sets the length of the big array *b* to 0:

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

[3] http://www.mathjax.org

[4] The last argument must be an array with one element for each dimension.

This is a precaution against the possibility, albeit unlikely, that a callback routine could keep a big array argument in a reference variable for another part of the program to later access after the nvector memory had been freed. Note, however, that this mechanism can be circumvented using subarrays (as noted by [5]). The OCaml runtime will eventually garbage collect the wrapper big array.

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 be cached with the session value if necessary. Callbacks from Sundials do occur inside critical loops, but we can expect *caml_ba_alloc* to allocate on the minor heap and a *malloc* is not required since we pass a data pointer. The real problem with this approach is that it does not generalize well to parallel and custom nvectors. We tried using conditional compilation and code duplication (copy-paste-modify), but the result was confusing for users, tedious to maintain, and increasingly unwieldy as we extended the interface to include CVODES and IDAS.

The solution we finally adopted exploits features of the nvector abstract datatype and polymorphic types to treat nvectors more generically and without code duplication. The idea is straightforward: we pair an OCaml representation of the contents with a pointer to an nvector structure, and we link both to the same underlying data array as in the previous solution. This gives the type

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

which becomes the opaque *Nvector.t* used throughout the interface. For serial nvectors, *'data* is instantiated as a big array of *float*s. The phantom type argument *'kind* is justified subsequently. The *cnvec* component is an nvector (a pointer to a structure allocated in the C heap). The last component of the triple returns true only when passed a compatible nvector; for serial nvectors, this means one of the same length. This function together with the type arguments precludes using nvectors in a way that would lead to invalid memory accesses.

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, _, _) = payload

and, for calls into Sundials, the C side of the interface can also readily access the associated nvector:

#define NVEC_CVAL(v) ((N_Vector *)Data_custom_val(v))*

#define NVEC_VAL(v) (NVEC_CVAL(Field(v, 1)))

without any need to destroy it before returning.

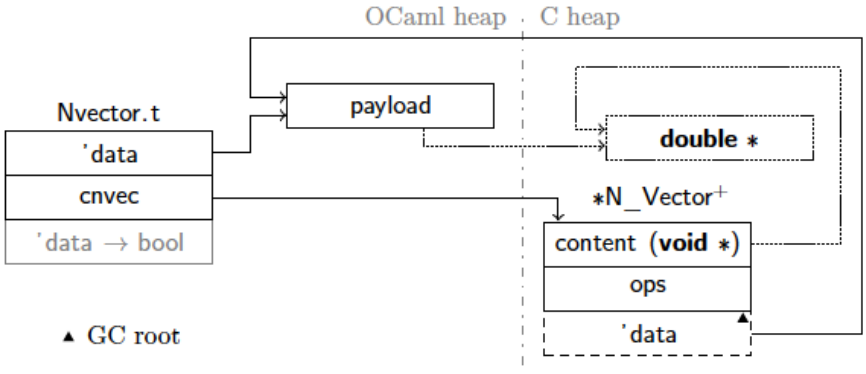

Figure 3: Interfacing nvectors (the dotted lines are for serial nvectors)

Callbacks into OCaml require more work. Callback stub functions must recover the OCaml representations of the *N_Vector* values they are passed before invoking OCaml code. 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 summarized in Figure 3. 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 both that it is updated if the payload is moved and that the payload is not destroyed inopportunely. Pointing this global root directly at the *Nvector.t* would give 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, which permits the payload to be collected when no other references to it exist. Provided nvector operations on payload values are also exposed, this choice works well in practice.

The operations of serial (and parallel) *N_Vector*s are unchanged but for *nvclone* and *nvdestroy*.[5] 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 3. The replacement *nvdestroy* removes the global root and frees memory allocated directly in the C heap but it 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 thus 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.

The solution just described is more complicated than the original one, and it does not work with nvectors created outside the OCaml interface, but it is efficient and it generalizes well. For parallel nvectors, the payload is the triple:

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

where the first element is an array of local elements, the second gives the total (global) number of elements, and the third specifies the MPI processes that communicate together.[6] 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.

For custom nvectors, we provide a module that turns OCaml functions defining the nvector operations and a value of the type manipulated by them into an *Nvector.t* with the value as the payload and *ops* that call back into OCaml. In this case, we use the *N_Vector content* field to store the closures defining the operations (registered as a global root), since the *ops* point to generic stub code.[7] The reason for the *'kind* argument in the definition of *Nvector.t* now becomes evident: to distinguish between a serial nvector with an aliased *content* field and a custom nvector whose payload is also a big array of *float*s. There is no difference from the OCaml side of the interface, but the differences in the underlying representations are important from the C side. We declare an uninterpreted type called *kind* in each nvector module, giving three in all: *Nvector_serial.kind*, *Nvector_parallel.kind*, and *Nvector_custom.kind*.

2.2. Sessions

OCaml session values must track an underlying Sundials session pointer and also maintain references to callback closures and some bookkeeping details. We exploit the Sundials user data

[5] And also *nvcloneempty*, which is optional and not provided.

[6] Using the OCaml MPI binding: https://forge.ocamlcore.org/projects/ocamlmpi/.

[7] In theory, it is possible to create an OCaml-C reference loop, and thus prevent garbage collection, by referring back to an \ocaml{Nvector.t} value from within a custom payload or operations. We rule this a misuse of the library.

feature to implement callbacks. The technical challenges are to avoid reference loops (using weak pointers), and to smoothly accommodate sensitivity analysis features (using extra session fields).

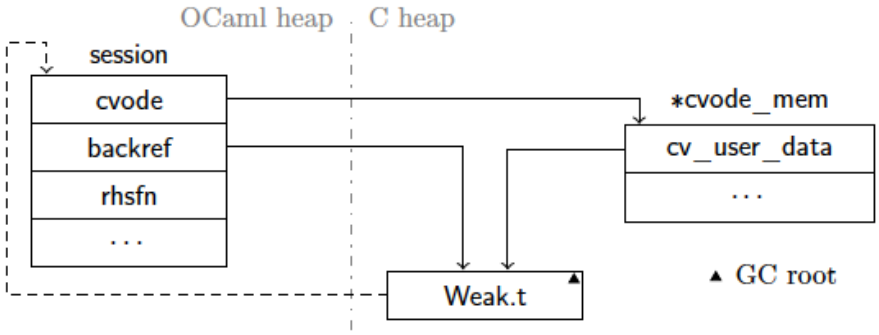

Figure 4: Interfacing CVODE sessions

The solution we implemented is sketched in Figure 4 and described below for the CVODE and CVODES solvers. The treatment of IDA, IDAS, and KINSOL is essentially the same. OCaml session types are parameterized by *'data* and *'kind* type variables, they are represented internally as records:

type ('data, 'kind) session = { ... }

The type variables are instantiated from the nvector passed to the *init* functions that create session values—this initial nvector is cloned several times by Sundials for internal calculations.

Each session value contains a pointer, labeled *cvode* in Figure 4, to a Sundials memory block, labeled **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. The root value cannot be stored directly in **cvode_mem* without violating the interface provided by Sundials; we prefer to simply call *CVodeSetUserData*. The root value must refer back to the *session* value since it is used to retrieve callback closures. This poses the problem of cycles across the two heaps: a direct reference from the root value would prevent garbage collection of the *session* but the root cannot be removed unless the *session* is finalized. The solution is to break the cycle with a weak reference that is updated when *session* is moved but which does not prevent its destruction. The *backref* field is used only by the finalizer of *session* to unregister the global root and to free the associated memory.

```
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
7      WEAK_DEREF (session, *(value*)user_data);
8
9      args[0] = caml_copy_double(t);
10     args[1] = NVEC_BACKLINK(y);
11     args[2] = NVEC_BACKLINK(ydot);
12
13     r = caml_callback3_exn(Field(session, RECORD_CVODE_SESSION_RHSFN),
14                            args[0], args[1], args[2]);
15
16     CAMLreturnT(int, CHECK_EXCEPTION (session, r, RECOVERABLE));
17 }
```

Figure 5: Typical callback stub

The code extract in Figure 5 shows a typical callback stub—for the CVODE 'right-hand side function', the *f* of section 1.1. The function *rhsfn* is registered as the right-hand side function for every CVODE session created by the interface, that is, in every call to *CVode*. 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 pointer registered by *CVodeSetUserData*.

Lines 3 to 5 contain standard boilerplate for an OCaml stub function. Line 7 follows the references sketched in Figure 4 to retrieve the *session* record.[8] The weak reference is guaranteed to point to a value since Sundials cannot be invoked from OCaml without passing a session. Line 9 copies the floating-point argument into the OCaml heap. Lines 10 and 11 recover the nvector payloads using the macro defined in section 2.1. Line 13 retrieves and invokes the appropriate closure from the session object. Finally, at line 16, 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.

An alternative approach for linking the C **cvode_mem* to an OCaml *session* value is outlined in Figure 6. Since for a callback to occur, control must already have passed into the Sundials library through the

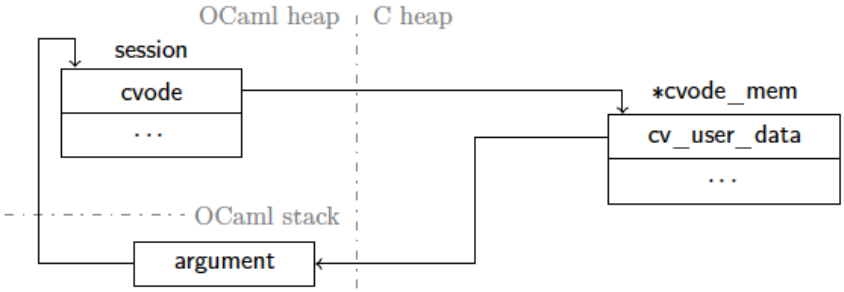

Figure 6: Alternative session interface (not adopted)

interface, there will be a reference to the *session* value on the OCaml stack. It is thus possible to pass the address of the reference to *CvodeSetUserData* before calling into Sundials. The reference will be updated 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 due to fewer indirections and because there is no need to call *caml_weak_get*. But although the implementation is uncomplicated for functions like *rhsfn*, which are only called during solving, it is more invasive for the error-handling functions that can, in principle, be triggered from nearly every call. Even if, in practice, the error-handling function is called more rarely under the OCaml interface[9] 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.

While in Sundials the CVODES solver conceptually extends the CVODE solver, it is implemented in a distinct code base.[10] For the OCaml library, we wanted to maintain the idea of an extended interface but avoid 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 compiled *cvode_impl.cmi* file—although ideally such abstractions would be expressible within the module system. The session record has an extra mutable field of type

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

[8] The *WEAK_DEREF* macro contains a call to *caml_weak_get*.

[9] For instance, *cvode_mem* can never be *NULL*.

[10] This description applies equally well to the IDAS and IDA solvers.

that extends the scheme in Figure 4 to handle sensitivity features. The *fsensext* and *bsensext* records essentially contain fields for the additional quadrature and sensitivity callback closures. The *fsensext* record also contains a list of associated backward session values to prevent their garbage collection while they may still be required by C-side callback stubs that only hold weak references. The *bsensext* also contains a link to the parent session and the integer identifier sometimes required by Sundials functions. Reusing the basic session interface for backward sessions simplifies the underlying implementation at the cost of some redundant fields—the forward callback closures are never used, —and an indirection.

The only other technical complication in interfacing with CVODES (and IDAS) is that they often work with arrays of nvectors. For calls into Sundials, we allocate short-lived C arrays in the interface code and invoke the *NVEC_VAL* macro, described on page 10, in a loop. For the performance critical callbacks from Sundials, we maintain cached *array*s in the *sensext* values and populate them by iterating the *NVEC_BACKLINK* macro from page 11.

2.3. Linear solvers

The different linear solver modules and the options available for configuring them offer a great many possibilities. In our interface we tried to clarify the valid combinations and their configuration in the OCaml module and type systems. We also wanted the result to be open to extension by alternate modules written in OCaml, and to cleanly accommodate the built-in parallel preconditioners without introducing a mandatory dependency on MPI.

An obvious first idea is to introduce a variant type with one tag per module, which is exactly what we did in early versions of the interface. This approach has the advantage of centralizing all possibilities. Externally, users simply construct values from a list of possibilities, and internally the implementation is built around a single *match*/*with* that makes the appropriate calls into Sundials. Compatibility constraints between sessions, linear solver modules, and preconditioners are elegantly expressed using the *kind* types (section 2.2) together with Generalized Abstract Data Types (GADTs) [2](§7.18). Incorporating parallel preconditioners is problematic however, since the values that specify them should also carry the types of callback functions that incorporate the *MPI.communicator* type. While this problem could be resolved with conditional compilation, we wanted to avoid complicating the code, build system, and testing.

The solution we finally adopted is to declare a distinct, opaque *linear_solver* type in each solver module, that is, alongside each *session* type. Internally, these types abbreviate functions that configure a given session, possibly using information obtained from an initial nvector:

```
type ('data, 'kind) linear_solver =
  ('data, 'kind) session -> ('data, 'kind) nvector -> unit
```

For example, the CVODE diagonal linear solver is interfaced by the submodule:

```
module Diag : sig
  val solver : ('data, 'kind) 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 can be passed to the *Cvode.init* or *Cvode.reinit* functions where it is invoked and makes calls to the Sundials *CVodeSetIterType* and *CVDiag* functions that setup the diagonal linear solver. The *get_** functions retrieve statistics specific to the diagonal module, other linear solver modules 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 to throw an exception when a session is configured with one linear solver and passed to a function that assumes another. This constraint cannot be expressed using types since a session may be reinitialized with a different linear solver.

Interfacing the DLS modules requires treating nvector compatibility and callback functions. The linear

solver values are declared in a *Dls* submodule:

```
val dense : ?jac:dense_jac_fn -> unit -> serial_linear_solver
val lapack_dense : ?jac:dense_jac_fn -> unit -> serial_linear_solver
val band : ?jac:band_jac_fn  -> bandrange -> serial_linear_solver
val lapack_band : ?jac:band_jac_fn  -> bandrange -> serial_linear_solver
```

where the restriction to serial nvectors is expressed in the type synonym:

```
type serial_linear_solver =
  (Nvector_serial.data, Nvector_serial.kind) linear_solver
```

The optional callback functions pose a new problem since they are passed a Sundials *DlsMat* value [3](§8.1.1). The representation of these values and the control of their lifetime by Sundials is deferred to section 2.4.

A SPILS linear solver results from the combination of an iterative method with a preconditioner. Iterative methods are exposed functions that take an optional parameter and a preconditioner and that return a linear solver:

```
val spgmr : ?maxl:int -> ('data, 'kind) preconditioner -> ('data, 'kind) linear_solver
```

The *spbcg* and *sptfqmr* methods have identical types. As is clear from the type signature, it is the preconditioner that constrains which nvectors may be used. Internally the *preconditioner* type pairs the preconditioning 'side' (left, right, both, or none) with a function that configures a preconditioner given a session and an nvector. For instance *Cvode.Spils* provides:

```
val prec_none : ('d, 'k) preconditioner
val prec_left : ?setup : 'data prec_setup_fn
        -> ?jac_times_vec : 'data jac_times_vec_fn
        -> 'data prec_solve_fn
        -> ('data, 'kind) preconditioner
val prec_right : (* as above *)
val prec_both : (* as above *)
```

where the last three functions produce preconditioners from optional setup and Jacobian multiplication functions and mandatory solve functions. The banded preconditioners are only compatible with serial nvectors. We group them into a submodule *Cvode.Spils.Banded*:

```
val prec_left : ?jac_times_vec : (Nvector_serial.data jac_times_vec_fn)
  -> bandrange -> (Nvector_serial.data, Nvector_serial.kind) preconditioner
val prec_right : (* as above *)
val prec_both : (* as above *)
```

The banded preconditioner provides its own setup and solve functions. The BBD preconditioner is interfaced similarly but with type variables instantiated to *Nvector_parallel.data* and *Nvector_parallel.kind* in a separate *Cvode_bbd* module. When MPI is not available, this module is simply not compiled.

Finally, we provide a way for users to exploit the *linit*, *lsetup*, and *lsolve* hooks from OCaml. For this, a *Cvode.Alternate* submodule provides the function *make*:

```
type ('data, 'kind) callbacks = {
  linit  : ('data, 'kind) linit option;
  lsetup : ('data, 'kind) lsetup option;
  lsolve : ('data, 'kind) lsolve; }
```

```
val make :
  (('data, 'kind) session -> ('data, 'kind) Nvector.t -> ('data, 'kind) callbacks)
  -> ('data, 'kind) linear_solver
```

which creates a linear solver given a function from a session and initial nvector to a record containing optional and required callbacks. This interface works naturally enough, but its utility is uncertain since linear solvers normally need access to some internal session fields, and since it adds overhead inside performance-critical loops. That said, we try to mitigate the former limitation by exposing some internal fields within *Alternate*, and the experiments described in section 3, albeit limited, have reasonable execution times.

2.4. Jacobian Matrices

In Sundials, two-dimensional matrices are represented by a *DlsMat* type [3](8.1.1). They are used in the DLS modules to hold and manipulate Jacobian data either in dense, that is, all M by N elements, or band form, that is, the main diagonal, the ml diagonals below it, and the mu diagonals above it. *DlsMat* values are implemented by records containing form (dense or band), dimensions, an array of floats containing the raw data (in column-order), and an array of column pointers into the data.

The DLS modules pass *DlsMat* values to user-supplied callback functions to be filled with approximations to the Jacobian matrix of a system in a given state. Interfacing with these values is awkward because their lifetimes are controlled by Sundials, and, unlike for nvectors, they are manipulated directly and not through a table of abstract operations. The solution we apply is more pragmatic than ideal.

OCaml types for matrices are declared in *Dls.DenseMatrix* and *Dls.BandMatrix* modules. The two modules differ only in that the latter must map row and column coordinates into the stored diagonals. The *Dls.DenseMatrix.t* type is realized by a record:

```
type t = {
  payload : (float, float64_elt, c_layout) Bigarray.Array2.t;
  dlsmat  : Obj.t;
  mutable valid : bool;
}
```

where *payload* is a two-dimensional big array, *dlsmat* is a *DlsMat* (a C pointer to a record), and *valid* is a Boolean that indicates whether *dlsmat* is valid (it is *true* initially). The *payload* and *dlsmat* values share the same underlying matrix data. The *get* and *set* functions can be implemented either by wrapping the underlying Sundials macro, which costs a call into C and a *caml_copy_double*, or by exploiting the big array wrapper:

```
let get { payload; valid } i j =
  if Sundials_config.safe && not valid then raise Invalidated;
  payload.{j, i}
```

The validity check is optimized away when compiling without safety checks and the compiler has a chance to avoid boxing the float.

Our callback stubs exploit the fact that each DLS module allocates only two *DlsMat* values, and that only one of them is ever passed to callback functions. When first called, the stubs wrap a given *DlsMat* value in one of the types described above and cache it in the *session* record. When a linear solver module is reinitialized, or when a session is finalized, the *valid* fields of any cached values are set to *false*. This solution is not pretty, but we do not know of a better one.

2.5. Linking

As stated earlier, while conceptually CVODES and IDAS extend CVODE and IDA with new functionality, each solver is implemented in a distinct code base. There are thus five, counting KINSOL, distinct *libsundials_** libraries, two pairs of which share a common subset of symbols.

To simplify as much as possible basic use of the interface, we produce a *sundials.cma* library that includes modules containing common data types (*Sundials*, *Dls*, and *Spils*), serial and custom nvector implementations (*Nvector*, *Nvector_serial*, and *Nvector_custom*), and all of the solvers (*Cvode*, *Cvodes*, *Ida*, *Idas*, and *Kinsol*). We link it with *libsundials_cvodes*, *libsundials_idas*, *libsundials_kinsol*, and *libsundials_nvecserial* Sundials libraries. A program using the library is compiled as follows:

ocamlc -o myprog.byte -I +sundialsml bigarray.cma sundials.cma myprog.ml

We also provide an alternate *sundials_no_sens.cma* library that includes the same common and nvector modules, but only the *Cvode*, *Ida*, and *Kinsol* solver modules. It is linked with the *libsundials_cvode*, *libsundials_ida*, *libsundials_kinsol*, and *libsundials_nvecserial* Sundials libraries. Evidently, this library only provides a subset of the solvers and it executes different underlying code. The difference can be seen in the results of functions like *Cvode.get_work_space* and *Ida.get_work_space* that return different results depending on which Sundials libraries are linked. The alternate library is thus at least important for the testing described in section 3 where the outputs of different implementations are expected to match precisely.

All of the modules with dependencies on MPI (*Nvector_parallel*, *Cvode_bbd*, *Cvodes_bbd*, *Ida_bbd*, and *Idas_bbd*) are compiled into the *sundials_mpi.cma* library, which is linked with *libsundials_nvecparallel*. To compile OCaml programs that use parallel nvectors requires adding *sundials_mpi.cma* after *sundials.cma* (or *sundials_no_sens.cma*) in calls to *ocamlc*.

3. Evaluation

Interface code inevitably adds run-time overhead: there is extra code to execute at each call. Rather than count the number of additional instructions, it is 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. In this paper 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 calculate and directly relevant to users. We do not compare memory or energy use as these are less relevant to the application domain—memory use is anyway proportional to problem size.

Sundials is distributed with many example programs (53 with serial nvectors and 24 with parallel nvectors) that exercise a wide range of solver features in numerically interesting ways. We reimplemented all of these examples in OCaml. Comparing the outputs of corresponding OCaml and C versions with *diff* led us to correct many bugs in our interface code and example implementations, and even to discover and report several bugs in Sundials itself. Beyond these comparisons we also used *valgrind* [6] and manual invocations of the garbage collector to reveal memory-handling errors in our code.

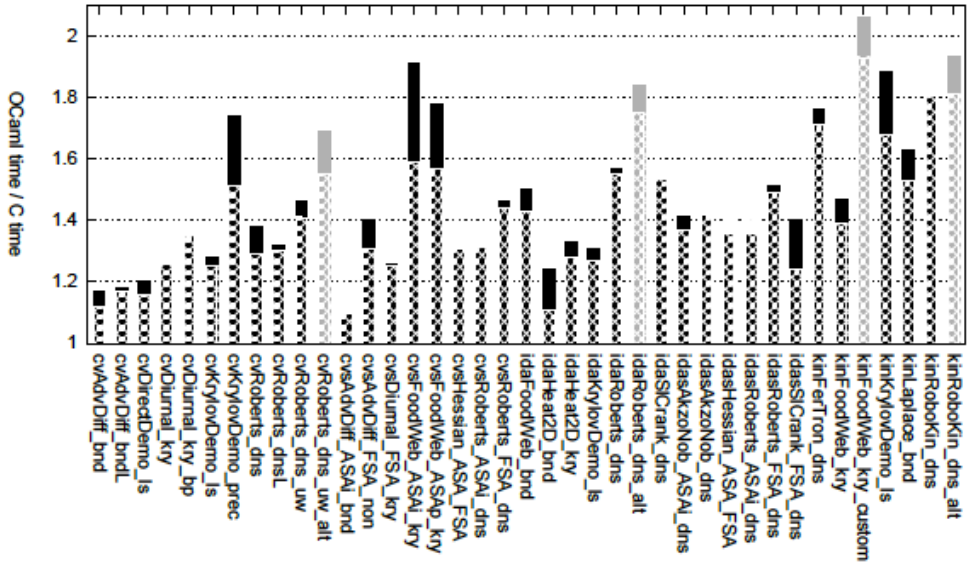

Figure 7: Serial examples: C (gcc 4.9.1 with -O2) versus OCaml (4.02.01). The hashed sub-bars show the results when compiled with --unsafe. Results were obtained under Linux 3.16.0 on an Intel i7 running at 2.60GHz with a 6Gb L2 cache and 8Gb of RAM.

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 sketched in section 2.2. The bars in Figure 7 show the ratios of the execution times of the OCaml code against the C code. A value of 2 on the left axis means that OCaml takes twice as long to calculate the same result. We only present the results for the serial examples since those for the parallel examples are dominated by process setup and communication costs, that is, their ratios are mostly close to 1.0 and never more than 1.1. The gray bars show the results for 'customized' examples: the *kinFoodWeb_kry_custom* example uses custom nvectors with low-level operations implemented in OCaml on *float array*s; the **_alt* examples use an alternate linear solver reimplemented in OCaml through the *Dls* binding.[11] Each customized example produces the same output as the corresponding original but takes longer, see the corresponding left-hand neighbors. Ignoring the gray bars, the graph suggests that the OCaml versions are rarely more than 80% slower than the original ones and that they are often less than 50% slower. The hashed sub-bars in Figure 7 give the ratio achieved when the OCaml versions are compiled without checks on array access, nvector compatibility, or matrix validity.

The actual C run times are not given in Figure 7. Nearly all of them are less than 1 ms, the two longest are on the order of 2 s (*cvsFoodWeb_ASA*_kry*), the next longest is about 300 ms (*cvsDiurnal_FSA_kry*). 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 *malloc* and *free* at each iteration, we also manually trigger a full major collection and heap compaction in OCaml at each iteration. The fastest examples require 25 000 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 instance, we found using *valgrind* (*--tool=callgrind* and *kcachegrind*) that, for some examples, the OCaml versions spend a greater fraction of their time in *printf* than do the C versions.[12] We were thus able to lower their ratios by instead using the *print_string* and *print_int* functions. Finally,

[11] This involves calls from OCaml to C to OCaml to C.

[12] The difference was more marked prior to OCaml 4.02.01.

the run times we obtain often vary significantly from one execution to the next. The figure compares the mean run times over 3 executions.

We made the OCaml versions of the examples up to four times faster using three simple conventions.

1. We added explicit type annotations to all vector arguments. For instance, rather than declare a callback with
   let f t y yd = ...
   we take the standard approach of declaring
   let f t (y : Sundials.RealArray.t) (yd : Sundials.RealArray.t) = ...
   since then the compiler need not generate polymorphic code and can optimize for the big array layout.[13]
2. We avoided functions like *Bigarray.Array1.sub* and *Bigarray.Array2.slice_left*. These functions allocate new big arrays on the major heap, which increases the frequency of major GCs. 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, it can be faster to copy into and out of a temporary array.
3. We write numeric expressions and loops according to the advice in [7] to avoid float 'boxing'.

In summary, the results we obtained, against a small set of examples, indicate that OCaml code using the Sundials solvers should rarely be more than twice as slow as equivalent code written in C, provided certain guidelines are followed, and it may typically be only about 50% 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?") with the cost of producing and maintaining programs ("How much effort will it take to write, debug, and later modify a program?"). This section provides some insight into the former cost. The latter cost is more difficult to quantify. But, arguably, it is faster to write and debug OCaml code thanks to automatic memory management, bounds checking on arrays, strong static type checking, higher-order functions, etcetera. The Sundials/ML library is especially compelling for programs that combine numeric calculation and symbolic manipulation.

4. References

[1] Alan C. Hindmarsh et al., "SUNDIALS: Suite of Nonlinear and Differential/Algebraic Equation Solvers," *ACM Transactions on Mathematical Software*, vol. 31, no. 3, pp. 363-396, September 2005.

[2] Xavier Leroy et al., "The OCaml system: Documentation and user's manual," Inria, 2014.

[3] Alan C. Hindmarsh and Serban Radu, "User Documentation for CVODE v2.6.0," Lawrence Livermore National Laboratory, 2009.

[4] Alan C. Hindmarsh and Radu Serban, "User Documentation for CVODES v2.7.0," Lawerence Livermore National Laboratory, 2012.

[5] Daniel Bünzli. (2005, January) Caml-list mailing list: Bigarrays and temporar[y] C pointers.

[6] Nicholas Nethercote and Julian Seward, "Valgrind: a framework for heavyweight dynamic binary instrumentation," in *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, San Diego, CA, USA, 2007, pp. 89-100.

[7] Xavier Leroy. (2002, July) Writing efficient numerical code in Objective Caml. [Online]. http://caml.inria.fr/pub/old_caml_site/ocaml/numerical.html

[13] *type Sundials.RealArray.t = (float, float64_elt, c_layout) Bigarray.Array1.t*