OwlDE: making ODEs first-class `Owl` citizens

Marcello Seri[1] **and Ta-Chu Kao**[2]

1 Bernoulli Institute for Mathematics, Computer Science and Artificial Intelligence, University of Groningen **2** Computational and Biological Learning Lab, Department of Engineering, University of Cambridge

DOI: 10.21105/joss.01812

Software

- Review
- Repository
- Archive

Editor: Matthew Sottile

Reviewers:

- @xoolive
- @rljacobson
- @mjsottile

Submitted: 04 October 2019
Published: 20 December 2019

Summary

After only three years of intensive development and continuous optimisation, `Owl` has emerged in the `OCaml` ecosystem as a versatile and powerful scientific programming library, competitive with mainstream libraries such as `SciPy` and `NumPy`. What sets `Owl` apart is that it brings under one umbrella the flexibility of a dynamical language and the power and safety of the `OCaml` type system (Wang, 2017).

Today, `Owl` can be used to solve a wide range of scientific problems: it provides efficient types for handling multidimensional arrays and linear algebra operations built on top of `BLAS` and `LAPACK`; it supports machine learning applications with a powerful computational graph engine and automatic differentiation pipeline. To improve efficiency, `Owl` allows offloading computations to distributed systems and GPUs. With the recent addition of `dataframes` and integration with Jupyter Notebooks provided by `ocaml-jupyter`, `Owl` has the chance to become an excellent framework for exploratory mathematical analysis.

A notable omission in `Owl`'s ecosystem, when compared to similar solutions in `python` and `Julia`, was a package for solving ordinary differential equations. To fill this need, we designed `OwlDE`, a flexible and efficient ODE engine for `Owl`.

Design of the core module

The lack of automation around type classes and dynamic typing in `OCaml` may seem at first a huge impediment to designing a flexible and ergonomic ODE integrator library. Indeed, such a library should ideally allow the users to seamlessly use different algorithms that require different kinds of inputs and options, and return varying kinds of outputs. Such constraints on input/output types pose a major problem for the strong static type system of `OCaml`, even if they are in principle implementable exploiting advanced language features, and were one of the central issues in designing `OwlDE`.

We iterated various options and settled on the use of first-class modules. These have been introduced in OCaml since version 3.12, and further improved in subsequent releases. With first-class modules the user can parametrise functions over modules, allowing us to find a middle ground between the verbosity of functorial code and the composability of OCaml functions. The flexibility of this API allowed us to provide integration and bindings to external frameworks like `SUNDIALS` or `ODEPACK` in a completely seamless way. Such an API allowed us to compare native implementations with industry-grade solvers, both for mathematical exploration, testing and benchmark purposes.

To give an idea of the interface, the following code allows to integrate the initial value problem

$$\dot{x} = f(x,t) = Ax, \qquad x(t_0) = x_0$$

where A is the 2x2 matrix $(1, -1; 2, -3)$, $x_0 = (-1; 1)$ and $t_0 = 0$.

```
open Owl
open Owl_ode
open Owl_ode.Types

(* f(x,t) *)
let f x t =
   let a = [|[|1.; -1.|];
             [|2.; -3.|]|]
           |> Mat.of_arrays in
   Mat.(a *@ x)

(* temporal specification:
   construct a record using the constructor T1 and
   includes information of start time, duration,
   and step size.*)
let tspec = T1 {t0 = 0.; duration = 2.; dt=1E-3}

(* initial state of the system *)
let x0 = Mat.of_array [|-1.; 1.|] 2 1

(* ts and xs will contain the integrated times and the values of the state
   x at each of those times *)
let ts, xs = Ode.odeint Native.D.rk4 f x0 tspec ()
```

The tight integration with the OCaml and Owl ecosystem allows us also to benefit from some of their strengths. The strong static type system made refactoring and code analysis an immediate task, and greatly reduced the necessary test surface. The powerful functorised ndarray subsystem exposed by Owl made the library trivially extensible also in somewhat unexpected directions. Indeed, it is possible to take the integrators exposed by OwlDE and use them to reproduce the work of (Chen, Rubanova, Bettencourt, & Duvenaud, 2018) without the need to rewrite any of the core functions, as done in adjoint_ode. Similarly, it is possible to extend the range of integrators and introduce new ones rather seamlessly, as done in cviode, an implementation of the integrators introduced in (Vermeeren, Bravetti, & Seri, 2019).

One further strength of this library, comes from its native OCaml component, which can be compiled to JavaScript and used for interactive simulations in the browser, as demoed during the OCaml Workshop at ICFP 2019 in Berlin. The demo and the usage instructions are freely available at owlde-demo-icfp2019.

Conclusion

When it comes to scientific programming, fast prototyping and ease of use have long been considered the province of dynamically-typed languages like python. However, our experience developing Owl and OwlDE, and the feedback from users who primarily use these tools for research, suggests otherwise. In fact, the OCaml type system often ensures correctness and speeds up computations without increasing verbosity, or hindering usability and readability. In many cases, we discovered that porting python code to OCaml code via Owl was nearly trivial and the resulting OCaml code was often comparable in length, but with the added benefits of fewer runtime errors and improved performance. We look forward to developing OwlDE and Owl further with inputs from the OCaml community.

Acknowledgements

The authors would like to thank Liang Wang, Guillaume Hennequin, the Cambridge OCaml Labs and all the contributors to `Owl` and `Owl_ode` for their support and help throughout the development of the project.

References

Chen, T. Q., Rubanova, Y., Bettencourt, J., & Duvenaud, D. K. (2018). Neural ordinary differential equations. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, & R. Garnett (Eds.), *Advances in neural information processing systems 31* (pp. 6571–6583). Curran Associates, Inc. Retrieved from http://papers.nips.cc/paper/7892-neural-ordinary-differential-equations.pdf

Vermeeren, M., Bravetti, A., & Seri, M. (2019). Contact variational integrators. *Journal of Physics A: Mathematical and Theoretical*. doi:10.1088/1751-8121/ab4767

Wang, L. (2017). Owl: A general-purpose numerical library in ocaml. Retrieved from https://arxiv.org/abs/1707.09616