DOI 10.1007/s10958-020-05096-2
Journal of Mathematical Sciences, Vol. 251, No. 3, December, 2020

LANDAU: A LANGUAGE FOR DYNAMICAL SYSTEMS WITH AUTOMATIC DIFFERENTIATION

I. Dolgakov* **and D. Pavlov***

UDC 519.682.6, 517.95, 521.3

Most numerical solvers used to determine the free variables of dynamical systems rely on first-order derivatives of the state of the system with respect to the free variables. The number of free variables can be fairly large. One of the approaches to obtaining these derivatives is the integration of the derivatives simultaneously with the dynamical equations, which is best done with automatic differentiation techniques. Even though there exist many automatic differentiation tools, none has been found to be scalable and usable for practical purposes of modeling dynamical systems. Landau is a Turing incomplete statically typed domain-specific language aimed to fill this gap. The Turing incompleteness allows for a sophisticated source code analysis and, as a result, a highly optimized compiled code. Among other things, the language syntax supports functions, compile-time ranged `for` *loops,* `if/else` *branching constructions, real variables and arrays, and allows for manually discarding calculations where the automatic derivative values are expected to be negligibly small. In spite of reasonable restrictions, the language is rich enough to express and differentiate any cumbersome equation with practically no effort. Bibliography:* 11 *titles.*

1. Introduction

In the modeling of dynamical systems, various systems from different application domains can be represented by an autonomous system of first-order ODEs:

$$\dot{\vec{x}}(t) = f(\vec{x}(t), \vec{p}), \tag{1}$$

where $\vec{p} = \{p_i\}_{i=1}^m$ is a vector of m fixed parameters. An instance of the model is determined by parameter values and also initial conditions:

$$\vec{x}(t_0) = \vec{x}_0. \tag{2}$$

For example, in the case of an N-body dynamical system, the parameters are masses, and the initial conditions are positions and velocities at a certain moment of time. In practice, e.g., in planetary ephemerides, the precise values of the initial conditions are unknown, while some approximate values are determined from observations.

The task is to solve the initial value problem (IVP) (1), (2) for a range of t covering all the t_i and to minimize the discrepancy between the observed and computed values. The IVP is most often solved numerically.

Let $\vec{P} = (x_0^{(0)}, \ldots, x_0^{(n)}, p_1, \ldots, p_m)$ be the full set of free variables to be fit to observations by (as is usual with dynamical systems) the nonlinear least squares method. The first-order derivatives $\frac{\mathrm{d}\vec{x}}{\mathrm{d}\vec{P}}$ are required for the method.

One way to obtain $\frac{\mathrm{d}\vec{x}}{\mathrm{d}\vec{P}}$ is to include it into our system of ODEs, together with $\vec{x}$ itself. Accordingly, the initial conditions $\frac{\mathrm{d}\vec{x}_0}{\mathrm{d}\vec{P}}$ and the time derivative $\frac{\mathrm{d}}{\mathrm{d}t}\frac{\mathrm{d}\vec{x}}{\mathrm{d}\vec{P}}$ are needed to solve the IVP for the new system. While the initial conditions are trivial, the time derivative must be obtained by substituting (1):

$$\frac{\mathrm{d}}{\mathrm{d}t}\frac{\mathrm{d}\vec{x}}{\mathrm{d}\vec{P}} = \frac{\mathrm{d}f(\vec{x}, \vec{p})}{\mathrm{d}\vec{P}}. \tag{3}$$

*Institute of Applied Astronomy of the Russian Academy of Sciences, St.Petersburg, Russia, e-mail: `ia.dolgakov@iaaras.ru, dpavlov@iaaras.ru`.

Published in *Zapiski Nauchnykh Seminarov POMI*, Vol. 485, 2019, pp. 78–89. Original article submitted October 14, 2019.

Thus, in order to estimate the free variables, one needs to compute the derivative of the right-hand side of the ODE with respect to $\vec{P}$. There are three ways to perform such a computation:

- Full symbolic differentiation, which requires a computer algebra system and can be quite costly computationally.
- Numeric differentiation using the finite difference technique, which is prone to truncation errors.
- Automatic differentiation.

Automatic differentiation (AD) is a technique of obtaining numerical values of derivatives of a given $\mathbb{R}^n \to \mathbb{R}^m$ function (Listing 1). In contrast to symbolic differentiation, AD not only reduces the computation time by using memoization techniques, but also provides more flexibility, as it can deal with complicated structures from programming languages, such as conditions and loops. Because of the associativity of the chain rule, there are at least two ways (modes) of memoization: forward and reverse.

The forward mode of AD is based on the concept of dual numbers and on traversing the computational graph (Fig. 1) in the natural forward order. Each variable of the original program is associated with its derivative counterpart(s), which is (are) computed along with the original variable value (see Listing 2). The computational complexity of the forward mode is proportional to the number n of independent input variables, so it is most effective when $n \ll m$. In our practice, the number of output values is far greater than the number of input values, therefore, we used forward accumulation.

```
1 func M(E, e):
2     w1 = E
3     w2 = e
4     w3 = sin(w1)
5     w4 = w3 * w2
6     w5 = w1 - w4
7     M = w5
8     return(M)
```

Listing 1. A function with $n = 2$, $m = 1$: the Kepler equation $M = E - e\sin(E)$.

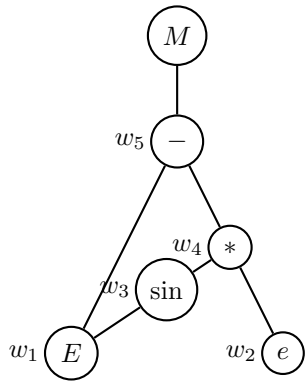

Fig. 1. The computational graph of the function from Listing 1.

In the case of the reverse mode, the values of the derivatives are accumulated starting from the root(s) of the computational graph, each assignment is accompanied with m accumulations (see Listing 3). Hence, the computational complexity of the reverse mode is proportional to the number m of outputs; thus, it is most effective when $n \gg m$, which is often the case in the computation of gradients of many-to-one functions so widely used in neural networks.

2. Related work and motivation

There exists a large number of forward-mode AD software tools for differentiating functions written in general-purpose programming languages, such as Fortran (ADIFOR) [2], C (ADIC) [3], or C++ (ADOL-C) [7]. Rich features of the "host" languages, such as arrays, loops, conditions, and recursion, often make it difficult to implement a practically usable AD

```
func dM(E, e):
  ∂w1/∂E = 1
  ∂w1/∂e = 0
  w1 = E

  ∂w2/∂E = 0
  ∂w2/∂e = 1
  w2 = e

  ∂w3/∂E = cos(w1) * ∂w1/∂E
  ∂w3/∂e = cos(w1) * ∂w1/∂e
  w3 = sin(w1)

  ∂w4/∂E = ∂w3/∂E * w2 + w3 * ∂w2/∂E
  ∂w4/∂e = ∂w3/∂e * w2 + w3 * ∂w2/∂e
  w4 = w3 * w2

  ∂w5/∂E = ∂w1/∂E - ∂w4/∂E
  ∂w5/∂e = ∂w1/∂e - ∂w4/∂e
  w5 = w1 - w4

  ∂f1/∂E = ∂w5/∂E
  ∂f2/∂E = ∂w5/∂e
  M = w5
  return(M, ∂M/∂E, ∂M/∂e)
```

Listing 2. Computing the derivatives $\frac{\partial M}{\partial E}$ and $\frac{\partial M}{\partial e}$ using the forward AD mode.

```
func dM(E, e):
  w1 = E
  w2 = e
  w3 = sin(w1)
  w4 = w3 * w2
  w5 = w1 - w4
  M = w5

  ∂M/∂w5=1
  ∂M/∂w4=∂M/∂w5 * (-1)
  ∂M/∂w3=∂M/∂w4 * w2
  ∂M/∂w2=∂M/∂w4 * w3
  ∂M/∂w1=∂M/∂w5*1+∂f/∂w3*cos(w1)
  ∂M/∂e=∂M/∂w2 * 1
  ∂M/∂E= ∂M/∂w1 * 1
  return(M, ∂M/∂E, ∂f/∂e)
```

Listing 3. Computing the derivatives $\frac{\partial M}{\partial E}$ and $\frac{\partial M}{\partial e}$ using the reverse AD mode.

system without imposing limitations on the language and/or extra technical work when specifying the function, especially in presence of multi-dimensional functions with many independent variables.

On the other hand, there is a number of languages developed specially for AD tasks, such as Taylor [8] and VLAD [9,10]. Taylor's syntax, while very simple and natural, is very limited (no conditionals, loops, arrays, or subprocedures). VLAD, a functional Scheme-like language, has conditionals, loops, recursion, and subprocedures, but does not have arrays or mutability.

Finally, there are tools for differentiating functions specified as mathematical expressions in mathematical computation systems, such as MATLAB (ADMAT) [5] or Mathematica (TIDES) (see [1, 2]). Such tools often require a bigger effort (as compared with general-purpose languages) to input a practical dynamical system of large dimension with a lot of free variables.

In this paper, we propose a new language, Landau, designed specially for dynamical systems. Other examples of such a design are TIDES and Taylor. However, TIDES and Taylor obtain high-precision solutions using the Taylor method and high-order derivatives, while Landau provides only first-order derivatives and is supposed to be used with numerical integrators yielding an acceptable approximate solution (such as the Runge–Kutta or Adams methods) with better performance than high-precision methods.

Like VLAD, Landau is a domain-specific language designed with automatic differentiation in mind. Like TIDES and Taylor, Landau offers C code generation. Like general-purpose

languages, Landau has common control-flow constructs, arrays, and mutability; but unlike general-purpose languages, Landau uses Turing incompleteness to perform static source analysis (see Sec. 4) and generate efficient code.

Landau also has the ability not only to derive dependencies between derivatives from the source code (e.g., if $x = y + z$, then $\frac{\partial x}{\partial y} = 1 + \frac{\partial z}{\partial y}$), but also to manually set values of derivatives of other variables belonging to the dynamical system (e.g., $\frac{\partial y}{\partial a} = \text{var}$).

3. Syntax

The language syntax includes functions, mutable real and integer variables, mutable real arrays, constants, `if/else` statements, and `for` loops. A special type `parameter` is used to express Jacobian denominator variables which are not used in expressions (the right-hand sides of assignments) themselves. In the case of dynamical equations, the derivatives with respect to such parameters can be used to express initial condition vectors. A special derivative operator `'` is used to annotate or assign a value of a derivative. Even with branching constructions (`if/else` statements), the function is guaranteed to be continuously differentiable thanks to the prohibition to use real arguments inside the condition body. Moreover, it is allowed to manually omit negligibly small derivatives using the `discard` keyword (e.g., if $x(a) = y(a) + z(a) + t(a)$ and the command `discard y ' a` is typed, then $\frac{\partial x}{\partial a} = \frac{\partial x}{\partial a} + \frac{\partial t}{\partial a}$).

Listing 4 demonstrates a Landau program for a dynamical system describing the motion of a spacecraft. The state of the system, i.e., the three-dimensional position and velocity of the spacecraft, obeys Newton's laws. The derivatives of the state with respect to six initial conditions (position and velocity) and one parameter (the gravitational parameter of the central body) are calculated using AD.

```
1  #lang landau
2
3  #Annotated parameters.  Function does not have them directly
4  #as arguments, but has derivatives w.r.t.  them in the state vector.
5  parameter[6] initial
6
7  real[6 + 36 + 6] x_dot (
8    real[6+36+6] x, # state+derivatives w.r.t.  initial and GM
9    real GM)
10 {
11   real[36] state_derivatives_initial=x[6 : 6+36]
12   real[6] state_derivatives_gm = x[6 + 36 : ]
13   real[6] state = x[ : 6]
14
15   # Set the state vector's Jacobian values.
16   state[ : ]'initial[ : ]=state_derivatives_initial[ : ]
17   state[ : ] ' GM = state_derivatives_gm
18
19   real[6] state_dot
20   #Transfer the time derivatives from x to their xdot counterparts,
21   # because ẋ = v_x.
22   state_dot[ : 3] = state[3 : ]
23
24   # Write the state_dot part to the function output.
25   x_dot[ : 3] = state_dot[ : 3]
26
```

```
27    # Apply Newtonian laws.
28    real dist2=sqr(state[0])+sqr(state[1])+sqr(state[2])
29    real dist3inv = 1 / (dist2 * sqrt(dist2))
30
31    state_dot[3 : ] = GM * (-state[ : 3]) * dist3inv
32
33    # Write the state_dot part to the function output.
34    x_dot[3 : ] = state_dot[3 : ]
35
36    # Write the state_dot derivatives to the function output.
37    x_dot[6 : 6 + 36] = state_dot[ : ] ' initial[ : ]
38    x_dot[6 + 36 : 6 + 36 + 6] = state_dot[ : ] ' GM
39  }
```

Listing 4. A Landau program for modeling the motion of a spacecraft around a planet. The initial position and velocity of the spacecraft, as well as the gravitational parameter of the planet, are supposed to be determined by the nonlinear least squares method.

4. Implementation

Automatic differentiation can be implemented in one of two ways: operator overloading and source code transformation. The first approach is based on describing the dual number data structure and overloading arithmetic operators and functions to operate on them. The second one involves an analysis of the function source and generation of the differentiation code. It was found [11] that the latter approach generally produces a more efficient differentiation code. Landau is written in Racket [6], and it uses the source code transformation approach to produce Racket or ANSI C differentiation code.

Let `lvalue` be the variable in the left-hand side of an assignment and `rvalues` be the variables in the right-hand side. The differentiation is performed in the following way: each real `lvalue` is associated with an array containing the values of the derivatives. The right-hand side of the assignment is differentiated symbolically[1], the result is stored in an accumulator array.

Each real variable assignment in a forward scheme is accompanied with n assignments to the accumulators of the derivatives, but in practice there is no need to compute and store all of them, because some values of the derivatives are never used afterwards. This means that the computed Jacobians are often sparse.

To illustrate the sparsity problem and keep things simple, we consider an artificial migration problem over N areas with a simplified diffusion model of migration:

$$\frac{\mathrm{d}p_i}{\mathrm{d}t} = \sum_{\substack{j=0 \\ j \neq i}}^{N} m_{ij} p_j, \quad i \in [0, N),$$

$$p_j(t_0) = p_j^{(0)}, \quad j \in [0, N),$$

where the initial condition vector $\mathbf{p}^{(0)} = \{p_j\}^{(0)}$ is supposed to be determined from observational data. Say, there are k regions with $l = \frac{N}{k}$ strongly interconnected areas whose population

[1]Even though the reverse mode is truly preferable if $n > m$, which is the case in a term-level assignment, because there is only one output in each assignment (i.e., $m = 1$), the computation overhead is negligibly small in the case of small expressions.

at an arbitrary moment of time depends on the initial conditions of other areas within the same region. Following the logic described in the introduction, we need to find the derivatives of the solution with respect to the initial conditions. A Landau program for solving this problem is presented in Listing 5.

```
#lang landau
const int N = 1000
const int k = 10
const int l = N / k
const int L2 = l * l
parameter[N] p0

real[N + L2 * k] f
  (real[N * N] m,
   real[N]     p,
   real[N * N] derivatives_p0) {

  p[ : ] ' p0[ : ] = derivatives_p0[ : ]

  real[N] p_dot
  for i = [0 : N]
    for j = [0 : N]
      if (i != j) {
        p_dot[i] += m[N * i + j] * p[j]
      }

  f[0 : N] = p_dot[ : ]

  for i = [0 : k]
    f[N + L2 * i : N + L2 * i + L2] =
      p_dot[l * i : l * i + l] ' p0[l * i : l * i + l]
}
```

Listing 5. An example Landau program demonstrating sparsity. One-to-many migration in $k = 10$ regions over $N = 1000$ areas.

The fact that the population depends on the initial conditions only within the region makes the Jacobian $\frac{\mathrm{d}\mathbf{p}}{\mathrm{d}\mathbf{p}_0}$ sparse:

$$\begin{bmatrix} J_{0,0} & \cdots & J_{0,99} & & & & & \\ \vdots & \ddots & \vdots & & & 0 & & \\ J_{99,0} & \cdots & J_{99,99} & & & & & \\ & & & \ddots & & & & \\ & & & & J_{900,900} & \cdots & J_{900,999} \\ & 0 & & & \vdots & \ddots & \vdots \\ & & & & J_{999,900} & \cdots & J_{999,999} \end{bmatrix}. \tag{4}$$

In this simple example, the sparsity pattern is represented by square blocks on the main diagonal, but it can be randomly sparse in general. Accumulating the derivatives of f in a straightforward manner would require to compute and store N^2 values, while only $\frac{N^2}{k}$ of them are needed.

There are at least two approaches to dealing with sparsity. The first approach is to generate a code where each useful[2] Jacobian matrix element is stored in a separate variable. This involves unrolling loops into assignmant sequences and, as a result, facing performance penalty due to CPU cache misses. Another approach is to store useful Jacobian values in arrays and preserve the ability to use loops for traversing. The sparsity is handled by packing useful Jacobian elements to smaller arrays and generating mappings from the packed derivative indices to the original ones and inverse mappings, which map the original indices to the packed ones. Listing 6 demonstrates the differentiation of the loops from lines 16–20 of Listing 5.

Compilation is performed in two stages. During the first stage, the information about dependencies, used variables, and derivatives is gathered for each variable or array cell. The Turing incompleteness guarantees that all loops and conditions can be unrolled and computed at compile time, so the initial Landau function can be transformed into a list of actions (Listing 7): derivative annotation, variable assignment, and storing derivatives in the output value. The list is then traversed to build the dependency graph of the derivatives, which is used at the second compilation stage to generate the mappings, inverse mappings, and differentiation code.

```
1  for i in [0 : N]:
2    for j in [0 : N]:
3      if i ≠ j:
4        for k in mappings_{p_dot, p0}(i):
5          dp_dot_dp0[k] = dp_dot_dp0[k]
6            + m[N * i + j] * dp_dp0[inv_mapping_{p, p0}(j,k)]
7        p_dot[i] = p_dot[i] + m[N * i + j] * p[j]
```

Listing 6. A pseudocode of the differentiation of the loops (lines 16–20 of Listing 5).

```
1  need-this-derivative p_dot[999] ' p0[999]
2  need-this-derivative p_dot[999] ' p0[998]
3  need-this-derivative p_dot[999] ' p0[997]
4  need-this-derivative ...
5  ...
6  p_dot[999] depends-from {p_dot[999], p[998]}
7  p_dot[999] depends-from {p_dot[999], p[997]}
8  p_dot[999] depends-from {p_dot[999], p[996]}
9  ...        depends-from ...
10 ...
11 have-this-derivative p[0] ' p0[2]
12 have-this-derivative p[0] ' p0[1]
13 have-this-derivative p[0] ' p0[0]
```

Listing 7. The list of reversed actions generated from Listing 5.

Let H be the length of the parameter vector and h_x be the number of derivatives (e.g., the number of nonzero elements in a row of the Jacobian) needed for the variable x. When $h_x \ll H$, most Jacobian values are not used and thus should not be computed. Using the mappings technique described above, we store only h_x derivative values and use the mappings $l \to k$

[2]We do not use the term "nonzero" here, because it may happen that a useful Jacobian matrix element is equal to zero.

and the inverse mappings $k \to l$, where $l \in [0, h_x)$, $k \in [0, H)$, to set and get derivative values. The mappings can be easily implemented as arrays of length h_x by storing the original indices of the parameter vector. But it is challenging to implement effectively the inverse mappings, because storing them in an array directly will result in $\Theta(H_{\max})$ memory consumption, where $H_{\max}$ is the largest used index of the parameter vector. For example, even if one needs to compute the derivative with respect to the last parameter index, the resulting mapping is an array of size 1, but the length of the inverse mapping is H.

A more sophisticated way to implement the inverse mappings is to use minimal perfect hash functions (MPHF). A perfect hash function maps a static set of h keys to a set of g integer numbers without collisions, where $g \geq h$. If $g = h$, the function is said to be minimal. There exist various asymptotically effective algorithms for generating such functions [4], but it is not clear whether the constant factors are small enough to make the generation of many MPHFs during a single compilation practically effective. In the current version of Landau, the inverse mappings are implemented as integer arrays and thus are not quite memory-efficient.

5. Conclusion

A new language called Landau has been invented to fill the niche of a domain-specific language designed for practically usable forward-mode AD for estimating the values of free parameters of a complex dynamical system.

A compiler that translates Landau code into either Racket or high-performance C code has been implemented, making the overall procedure of estimating free variables fast and fluent.

Further work is required to implement the inverse mappings more effectively. Such an implementation should clearly be possible thanks to the Turing incompleteness of Landau code, which allows for complete static analysis.

The authors are grateful to their colleague Dan Aksim for reading drafts of this paper and to Matthew Flatt and Matthias Felleisen from the PLT Racket team for their help with the Racket programming platform.

REFERENCES

1. A. Abad, R. Barrio, M. Marco-Buzunariz, and M. Rodríguez, "Automatic implementation of the numerical Taylor series method: A *Mathematica* and *Sage* approach," *Appl. Math. Comput.*, **268**, 227–245 (2015).
2. Ch. Bischof, A. Carle, G. Corliss, A. Griewank, and P. Hovland, "ADIFOR–generating derivative codes from Fortran programs," *Sci. Programming*, **1**, No. 1, 11–29 (1992).
3. Ch. Bischof, L. Roh, and A. J. Mauer-Oats, "ADIC: an extensible automatic differentiation tool for ANSI-C," *Software: Practice and Experience*, **27**, No. 12, 1427–1456 (1997).
4. F. C. Botelho, R. Pagh, and N. Ziviani, "Simple and space-efficient minimal perfect hash functions," in: *Workshop on Algorithms and Data Structures* (2007), pp. 139–150.
5. T. F. Coleman and A. Verma, "ADMAT: An automatic differentiation toolbox for MATLAB," in: *Proceedings of the SIAM Workshop on Object Oriented Methods for Inter-Operable Scientific and Engineering Computing*, **2**, SIAM, Philadelphia (1998).
6. M. Felleisen, R. B. Findler, M. Flatt, Sh. Krishnamurthi, E. Barzilay, J. McCarthy, and S. Tobin-Hochstadt, "A programmable programming language," *Comm. ACM*, **61**, No. 3, 62–71 (2018).
7. A. Griewank, D. Juedes, and J. Utke, "Algorithm 755: ADOL-C: a package for the automatic differentiation of algorithms written in C/C++," *ACM Trans. Math. Software*, **22**, No. 2, 131–167 (1996).

8. À. Jorba and M. Zou, "A software package for the numerical integration of ODEs by means of high-order Taylor methods," *Experiment. Math.*, **14**, No. 1, 99–117 (2005).
9. J. M. Siskind and B. A. Pearlmutter, "Nesting forward-mode AD in a functional framework," *Higher-Order Symbol. Comput.*, **21**, No. 4, 361–376 (2008).
10. J. M. Siskind and B. A. Pearlmutter, "Efficient implementation of a higher-order language with built-in AD," in: 7*th International Conference on Algorithmic Differentiation*, Oxford (2016).
11. M. Tadjouddine, Sjh. A. Forth, and J. D. Pryce, "AD tools and prospects for optimal AD in CFD flux Jacobian calculations," in: *Automatic Differentiation of Algorithms*, Springer (2002), pp. 255–261.