I. Dolgakov, D. Pavlov

LANDAU: LANGUAGE FOR DYNAMICAL SYSTEMS WITH AUTOMATIC DIFFERENTIATION

ABSTRACT. Most numerical solvers used to determine free variables of dynamical systems rely on first-order derivatives of the state of the system w.r.t. the free variables. The number of the free variables can be fairly large. One of the approaches of obtaining those derivatives is the integration of the derivatives simultaneously with the dynamical equations, which is best done with the automatic differentiation technique.

Even though there exist many automatic differentiation tools, none have been found to be scalable and usable for practical purposes of dynamic systems modelling. Landau is a Turing incomplete statically typed domain-specific language aimed to fill this gap. The Turing incompleteness provides the ability of 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 the ability to manually discard calculation where the automatic derivatives values are expected to be negligibly small. In spite of reasonable restrictions, the language is rich enough to express and differentiate any cumbersome paper-equation with practically no effort.

§1. INTRODUCTION

In dynamical system modeling, 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. One instance of the model is based on the values of the parameters, and also the 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 observed and computed values. The IVP is most often solved numerically.

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

One way to obtain $\frac{d\vec{x}}{dP}$ is to include it into our system of ODEs, together with $\vec{x}$ itself. Accordingly, the initial conditions $\frac{d\vec{x_0}}{dP}$ and the time derivative $\frac{d}{dt}\frac{d\vec{x}}{dP}$ 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):

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

Thus, in order to estimate the free variables, one needs to compute the derivative of the ODE's right-hand side w.r.t. $\vec{P}$. There are three ways to perform such a computation:

- Full symbolic differentiation, which requires a computer algebra system and can be quite computationally costly.
- Numeric differentiation using the 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). As opposite to the 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 chain rule associativity 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 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 forward mode is proportional

to the number of independent input variables n, thus it is most effective when $n \ll m$. In our practice the number of the function output values is far greater then the number of the input ones, 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 $M = E - e\sin(E)$
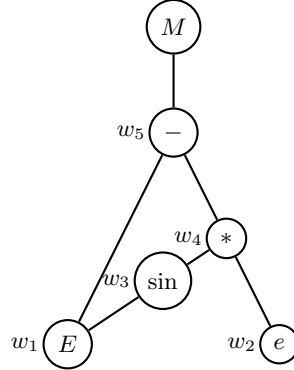

Figure 1. Computational graph of function from listing 1.

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

§2. Related work and motivation

There exist a large number of forward-mode AD software tools for differentiating functions that are written in general-purpose programming languages, like Fortran (ADIFOR) [3], C (ADIC) [4] or C++ (ADOL-C) [8]. Rich features of the "host" languages, like arrays, loops, conditions, and recursion, often make it difficult to implement a practically usable AD system without imposing limitations on the language and/or extra technical work

```
1   func dM(E, e):
2       ∂w₁/∂E = 1
3       ∂w₁/∂e = 0
4       w₁ = E
5
6       ∂w₂/∂E = 0
7       ∂w₂/∂e = 1
8       w₂ = e
9
10      ∂w₃/∂E = cos(w₁) * ∂w₁/∂E
11      ∂w₃/∂e = cos(w₁) * ∂w₁/∂e
12      w₃ = sin(w₁)
13
14      ∂w₄/∂E = ∂w₃/∂E * w₂ + w₃ * ∂w₂/∂E
15      ∂w₄/∂e = ∂w₃/∂e * w₂ + w₃ * ∂w₂/∂e
16      w₄ = w₃ * w₂
17
18      ∂w₅/∂E = ∂w₁/∂E - ∂w₄/∂E
19      ∂w₅/∂e = ∂w₁/∂e - ∂w₄/∂e
20      w₅ = w₁ - w₄
21
22      ∂f₁/∂E = ∂w₅/∂E
23      ∂f₂/∂E = ∂w₅/∂e
24      M = w₅
25      return(M, ∂M/∂E, ∂M/∂e)
```

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

```
1   func dM(E, e):
2       w₁ = E
3       w₂ = e
4       w₃ = sin(w₁)
5       w₄ = w₃ * w₂
6       w₅ = w₁ - w₄
7       M = w₅
8
9       ∂M/∂w₅ = 1
10      ∂M/∂w₄ = ∂M/∂w₅ * (-1)
11      ∂M/∂w₃ = ∂M/∂w₄ * w₂
12      ∂M/∂w₂ = ∂M/∂w₄ * w₃
13      ∂M/∂w₁ = ∂M/∂w₅ * 1 + ∂f/∂w₃ * cos(w₁)
14      ∂M/∂e = ∂M/∂w₂ * 1
15      ∂M/∂E = ∂M/∂w₁ * 1
16      return(M, ∂M/∂E, ∂f/∂e)
```

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

when specifying the function, especially in presence of multi-dimensional functions with many independent variables.

On the other hand, there exist a number of languages developed specially for AD tasks, like Taylor [9] and VLAD [10,11]. Taylor 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 computing systems, like MATLAB (ADMAT) [6] or Mathematica (TIDES) [1, 2]. Such tools often require a bigger effort (as compared to a general-purpose languages) to input a practical dynamical system of large dimension with a lot of free variables.

In this work, a new language, Landau, is proposed, designed specially for dynamical systems. Other examples of such design are TIDES and Taylor. However, TIDES and Taylor obtain high-precision solutions using Taylor method and high-order derivatives, while Landau provides only first-order derivatives and is supposed to be used with numerical integrators that obtain an acceptable approximate solution (like 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 embraces Turing incompleteness to perform static source analysis (see section 4) and generate efficient code.

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

§3. SYNTAX

The language syntax offers functions, mutable real and integer variables, mutable real arrays, constants, `if/else` statements and `for` loops. Special type `parameter` is used to express Jacobian denominator variables which are not used in expressions (the right-hand sides of the assignments) itself. In case of dynamical equations differentiation such parameters could express initial conditions vectors. Special derivative operator `'` is used to annotate or assign the value of the derivative. Even with branching constructions (`if/else` statements) the function is guaranteed to be continuously differentiable thanks to the prohibition of the 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 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 3-dimensional position and velocity of the spacecraft, obeys Newtonian laws. The derivatives of the state w.r.t. 6 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 = vₓ.
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. Landau program for modeling spacecraft movement around a planet. Spacecraft's initial position and velocity, as well as the gravitational parameter of the planet, are supposed to be determined by nonlinear least-squares method

§4. IMPLEMENTATION

Automatic differentiation can be implemented in one of two ways: the operator overloading and the 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 analysis of function source and generation of the differentiation code. It was found [12] that the latter approach generally produces more efficient derivative code. Landau is written in Racket [7] and it uses source code transformation approach to produce Racket or ANSI C differentiation code.

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

Each real variable assignation in a forward scheme is augmented with n assignations to the derivatives' accumulators, but in practice there is no need to compute and store all of them because some derivatives' values

---

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

are never used afterwards. That means that the computed Jacobians are often sparse.

To illustrate the sparsity problem and keep things simple let us 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 that there are k regions with $l = \frac{N}{k}$ strongly interconnected areas whose population at an arbitrary moment of time depends from the initial conditions of other areas within that region. Following the logic from the introduction, we need to find the solution derivatives with respect to the initial conditions. The Landau program for solving that problem is presented in listing 5.

```
1   #lang landau
2   const int N = 1000
3   const int k = 10
4   const int l = N / k
5   const int L2 = l * l
6   parameter[N] p0
7
8   real[N + L2 * k] f
9     (real[N * N] m,
10     real[N]     p,
11     real[N * N] derivatives_p0) {
12
13     p[ : ] ' p0[ : ] = derivatives_p0[ : ]
14
15     real[N] p_dot
16     for i = [0 : N]
17       for j = [0 : N]
18         if (i != j) {
19           p_dot[i] += m[N * i + j] * p[j]
20         }
```

```
21
22    f[0 : N] = p_dot[ : ]
23
24    for i = [0 : k]
25      f[N + L2 * i : N + L2 * i + L2] =
26        p_dot[l * i : l * i + l] ' p0[l * i : l * i + l]
27  }
```

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

The fact that the population depends on the initial conditions only within the region makes the Jacobian $\frac{d\mathbf{p}}{d\mathbf{p}_0}$ 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 the following simple example the sparsity pattern is presented with square blocks on the main diagonal but it could be randomly sparse in general. Accumulating the p's derivatives in a straightforward manner will require one to compute and store N^2 values while only $\frac{N^2}{k}$ are needed.

There are at least two approaches to deal with sparsity. The first approach is to generate the code where each useful[2] Jacobian matrix element is stored in a separate variable. That involves unrolling loops to the assignation sequences and, as a result, facing the performance penalty due to the 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 indexes to the original ones and inverse mappings, which map the original indexes to the packed ones.

---

[2]We are not using term nonzero here, because it can happen that the useful Jacobian matrix element is equal to zero.

The listing 6 demonstrates the differentiation of the loops from the lines 16–20 of the listing 5.

The 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, thus the initial Landau function can be transformed to a list of actions (listing 7): derivative annotation, variable assignation and storage of the derivative in the output value. The list is then traversed to gather the dependency graph of the derivatives, which is used in the second compilation stage to generate 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. Pseudocode of the loops (lines 16–20 of the listing 5) differentiation

```
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. Reversed actions list generated from the listing 5

Let H be the length of parameter vector and h_x be the number of derivatives (e.g. Jacobian row's elements) needed for the x variable. 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 mappings $l \to k$ and inverse mappings $k \to l$, where $l \in [0, h_x), k \in [0, H)$ to set and get derivative values. Mappings can be easily implemented as arrays with length h_x by storing the original indexes of the parameter vector. But it is challenging to implement effective inverse mappings, because storing them in array directly will result to the $\Theta(H_{\max})$ memory consumption, where $H_{\max}$ is the maximum used parameter vector index. For example, even if one needs to compute derivative with respect to the last parameter index, the resulting mapping is array of size 1, but inverse mapping's length is H.

More sophisticated way to implement inverse mappings is to use minimal perfect hash functions (MPHF). A perfect hash function maps a static set of h keys into a set of g integer numbers without collisions, where $g \geq h$. If $g = h$, the function is called minimal. Various asymptotically effective algorithms for generating such functions exist [5], but it is not clear if the constant factors are small enough to make the generation of many MPHFs during the 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 for more effective implementation of the inverse mappings. Such an implementation clearly should be possible thanks to Turing-incompleteness of Landau code that allows for complete static analysis.

Felleisen of the PLT Racket team for their help with the Racket programming platform.

References

1. A. Abad, R. Barrio, M. Marco-Buzunariz, M. Rodríguez, *Automatic implementation of the numerical Taylor series method.* — Appl. Math. Comput., **268** (2015), 227–245.

2. *Automatic implementation of the numerical Taylor series method: A Mathematica and Sage approach. — Automatic implementation of the numerical Taylor series method.* — Appl. Math. Comput., **268** (2015), 227–245.

3. Ch. Bischof, A. Carle, G. Corliss, A. Griewank, P. Hovland, ADIFOR–*generating derivative codes from Fortran programs.* — Scientific Programming, **1**, No. 1, (1992), 11–29.

4. Ch. Bischof, L. Roh, A. J. Mauer-Oats, ADIC*: an extensible automatic differentiation tool for* ANSI-C. — Software: Practice and Experience, **27**, No. 12 (1997), 1427–1456.

5. F. C. Botelho, R. Pagh, N. Ziviani, *Simple and space-efficient minimal perfect hash functions.* — In: Workshop on Algorithms and Data Structures, (2007), 139–150.

6. T. F. Coleman, A. Verma, ADMAT*: An automatic differentiation toolbox for* MAT-LAB. — In: Proceedings of the SIAM Workshop on Object Oriented Methods for Inter-Operable Scientific and Engineering Computing, SIAM, Philadelphia, PA, **2** (1998).

7. M. Felleisen, R. B. Findler, M. Flatt, Sh. Krishnamurthi, E. Barzilay, J. McCarthy, S. Tobin-Hochstadt, *A Programmable Programming Language.* — Commun. ACM, **61**, No. 3 (2018), 62–71.

8. A. Griewank, D. Juedes, J. Utke, *Algorithm* 755: ADOL-C: *a package for the automatic differentiation of algorithms written in* C/C++. — ACM Transactions on Mathematical Software (TOMS), **22**, No. 2 (1996), 131–167.

9. À. Jorba, M. Zou, *A Software Package for the Numerical Integration of ODEs by Means of High-Order Taylor Methods.* — Experimental Mathematics, **14**, No. 1 (2005), 99–117.

10. J. M. Siskind, B. A. Pearlmutter, *Nesting forward-mode AD in a functional framework.* — Higher-Order and Symbolic Computation, **21**, No. 4 (2008), 361–376.

11. J. M. Siskind, B, A. Pearlmutter, *Efficient Implementation of a Higher-Order Language with Built-In* AD. — AD2016 – 7th International Conference on Algorithmic Differentiation, Oxford, UK (2016).

12. M. Tadjouddine, Sjh. A. Forth, J. D. Pryce, AD *tools and prospects for optimal* AD *in* CFD *flux Jacobian calculations.* — Automatic differentiation of algorithms,(2002), 255–261.

Institute of Applied Astronomy
of the Russian Academy of Sciences

E-mail: `ia.dolgakov@iaaras.ru`