An Exact Bitwise Reversible Integrator

Jos Stam
NVIDIA

July 15, 2022

Abstract

At a fundamental level most physical equations are time reversible. In this paper we propose an integrator that preserves this property at the discrete computational level. Our simulations can be run forward and backwards and trace the same path exactly bitwise. We achieve this by implementing theoretically reversible integrators using a mix of fixed and floating point arithmetic. Our main application is in efficiently implementing the reverse step in the adjoint method used in optimization. Our integrator has applications in differential simulations and machine learning (backpropagation).

1   Introduction

Most fundamental physical equations are time reversible. Simulations of these equations running forward in time are indistinguishable from those running in reverse. However, our intuition tells us that this is not the case in real life. Rarely do we see bits of shards of broken glass reverse to form a bottle resting on a table. However, we are familiar with bottles falling off a table and smashing on the floor into many shards of glass. This is the so called "paradox of the arrow of time." How can time reversible fundamental equations give rise to seemingly irreversible phenomena? We will not dwell on these philosophical considerations, see for example [3]. An easy way out of this conundrum is to consider that all phenomena satisfying the reversible physical laws are possible but that some are very improbable. The ones with high probability to occur are the ones we observe in practice.

Examples of reversible processes include molecular dynamics, the Euler equations of fluid motion, planetary systems, and so forth. Although these systems are ideal cases they are nonetheless ubiquitous in simulation. Or at least until dissipative effects are introduced.

In computers, physical processes are discretized and integrated using a time step h. The state of the system (particle positions and velocities for example) is some finite quantity S. Both h and S are **not** represented as idealized (mathematical) real numbers but using finite representations. Basically bits, strings of ones and zeros. For example a time step of $h = \pi$ is represented in 32 bit IEEE 754 float standard as the following string of bits:

$$0 \; 10000000 \; 10010010000111111011011,$$

or alternatively in hex notation: 40420FDB [1]. Below we will show another way of discretizing reals like π using fixed point numbers.

In this paper, we discuss a class of reversible integrators for Hamiltonian systems. The most common example are Newton's equations with conservative forces that are the gradient of a potential field. When using floating point numbers these integrators are only approximately reversible. By using fixed numbers which basically amounts to integrating with integers, we obtain exact bitwise reversible simulations. This means that if we integrate the equations forward for an arbitrary number of time steps h and then decide to integrate backwards with $-h$ we recover any previous state exactly at a bitwise level. Like $1200045 = 1200045$ not approximately like $1200045 \approx 1200044$.

Many applications can benefit from reversible integrators. Here we list a few.

- Animators can browse back and forth in time to refine a simulation's results. This is akin to how traditional animators flip through their sequence of drawings using a *flipbook*.

- In Machine Learning and Optimization of physical systems a very popular method is the *adjoint method* [7, 12, 4]. This technique requires the reverse simulation of the physical system. Usually this is achieved by storing the results of the forward solution. We will discuss this application in more length below.

- Reversible simulations do not suffer from energy drift and are therefore relatively stable for long term simulations. The energy is not exactly conserved but bounded. They are however, not unconditionally stable and have an upper bound on the time step h.

This paper was mainly inspired by the seminal work of Levesque and Verlet [8]. This works fits into the class of *Geometrical Integrators* that preserve properties of the underlying equations they simulate [5]. We are of course not the first ones to use this technique. Rein and Tamayo apply it to simulating n-body systems like our solar system [11]. This technique has also been applied to analyzing complex statistical mechanical systems [6]. In this paper we clearly present how to implement this technique, providing source code. We also provide a novel application in computing gradients in the context of reversible neural ODEs.

2 Hamiltonian Systems

2.1 A Simple Spring

One of the simplest examples of a Hamiltonian system is that of a one-dimensional spring. Not just any spring but one that has unit mass and unit stiffness. The spring's state is then defined at any point in time by two variables its position $q(t)$ and its velocity (momentum) $p(t)$. The Hamiltonian (energy) in this case is equal to the sum of kinetic and potential energies: $H(q,p) = \frac{1}{2}p^2 + \frac{1}{2}q^2$. Energy conservation then directly implies that a solution lies on a circle in phase space $q - p$ that contains the initial condition $(q(0), p(0))^T$. We know that the spring satisfies Newton's second order differential equation: $\ddot{q} = -q$. Alternatively, the equations can be written as a first order differential equation using the Hamiltonian:

$$\dot{q} = \frac{\partial H}{\partial p} \quad \text{and} \quad \dot{p} = -\frac{\partial H}{\partial q}. \tag{1}$$

For our specific Hamiltonian we have that $\frac{\partial H}{\partial q} = q$ and $\frac{\partial H}{\partial p} = p$. Introducing the complex number $z(t) = q(t) + ip(t)$, Eq. 1 becomes $\dot{z} = -iz$ and its solution is a rotation of the initial state

$$z(t) = e^{-it} z(0) \tag{2}$$

and lies on a circle. Obviously, the solution is exactly reversible.

2.2 General Case

The general case is multi-dimensional. Our state consists of n *generalized coordinates* $\mathbf{q}$ and n *generalized momenta* $\mathbf{p}$. We assume that the Hamiltonian depends solely on the state $H(\mathbf{q}, \mathbf{p})$ satisfying Eq. 1 with q and p in boldface.

Consider any property $A(\mathbf{q}, \mathbf{p})$, its evolution over time is implicitly given by its dependence on the state $(\mathbf{q}, \mathbf{p})^T$. Indeed, using the chain rule of differentiation and Eq. 1, we get that:

$$\dot{A} = \dot{\mathbf{q}}\frac{\partial A}{\partial \mathbf{q}} + \dot{\mathbf{p}}\frac{\partial A}{\partial \mathbf{p}} = \frac{\partial H}{\partial \mathbf{p}}\frac{\partial A}{\partial \mathbf{q}} - \frac{\partial H}{\partial \mathbf{q}}\frac{\partial A}{\partial \mathbf{p}} = i\mathcal{L}\, A, \tag{3}$$

where

$$i\mathcal{L} = \frac{\partial H}{\partial \mathbf{p}}\frac{\partial}{\partial \mathbf{q}} - \frac{\partial H}{\partial \mathbf{q}}\frac{\partial}{\partial \mathbf{p}} = i\mathcal{L}_q + i\mathcal{L}_p. \tag{4}$$

is called the *Liouville operator*. Note that by convention the imaginary number i is there to emphasize that this operator is Hermitian. Eq. 3 can formally be solved as follows:

$$A(t) = e^{i\mathcal{L}t}\, A(0). \tag{5}$$

Similarly to the simple spring example, we can introduce a $2n$-dimensional variable $\mathbf{z} = (\mathbf{q}, \mathbf{p})^T$ and plug it into Eq. 5 for each of its components ($A = z_i$) to get an equation for the evolution of our state:

$$\mathbf{z}(t) = e^{i\mathcal{L}t}\, \mathbf{z}(0). \tag{6}$$

We can regard this equation as a "rotation" (unitary transformation) of the initial state over time. The evolution is therefore exactly reversible and preserves energy analogously to the one dimensional spring case above. We can verify that this is indeed a special case $i\mathcal{L}z = p\frac{\partial z}{\partial q} - q\frac{\partial z}{\partial p} = p - iq = -i(q + ip) = -iz$.

2.3 Discretization

We now derive a reversible discrete integrator from the theoretical considerations. We follow the work of Tuckerman *et al.* [13]. The idea is to discretize Eq. 6 by replacing the Liouvillian by its decomposition (Eq. 4) and replace the continuous time variable t with its discrete counterpart h.

$$\mathbf{z}(h) = e^{(i\mathcal{L}_q + i\mathcal{L}_p)h}\, \mathbf{z}(0) = e^{i\mathcal{L}_q h}\, e^{i\mathcal{L}_p h}\, \mathbf{z}(0) + O(h^2). \tag{7}$$

The last equality is exact only when $i\mathcal{L}_q$ and $i\mathcal{L}_p$ commute which in general is not the case. The propagation operator on the left hand side is also non reversible. The clever idea is to replace the propagator with an equivalent symmetrical one:

$$\mathbf{z}(h) = e^{i\mathcal{L}_q h/2}\, e^{i\mathcal{L}_p h}\, e^{i\mathcal{L}_q h/2}\, \mathbf{z}(0) + O(h^3). \tag{8}$$

This integrator is unitary and invertible as desired and of order two. This is sufficient for our bitwise-accurate integrator below. However, higher order integrators can be constructed similarly if needed [5].

Now using first order approximations of the integrators in Eq. 8 becomes the well known *Position Verlet* integration step:

$$
\begin{aligned}
\mathbf{q}(h/2) &= \mathbf{q}(0) + h/2 \, \frac{\partial \hat{H}(0)}{\partial \mathbf{p}} \\
\mathbf{p}(h) &= \mathbf{p}(0) + h \, \left(-\frac{\partial \hat{H}(h/2)}{\partial \mathbf{q}} \right) \\
\mathbf{q}(h) &= \mathbf{q}(h/2) + h/2 \, \frac{\partial \hat{H}(h)}{\partial \mathbf{p}},
\end{aligned}
$$

where $\hat{H}(t) = H(\mathbf{q}(t), \mathbf{p}(t))$. We point out that by exchanging the roles of $\mathbf{q}$ and $\mathbf{p}$ in Eq. 8 we get the *Velocity Verlet* integrator which is also time reversible. This method is also more commonly referred to as *Leap Frog*.

In most application the Hamiltonian is the sum of kinetic energy and a potential V: $H = \frac{1}{2}|\mathbf{p}|^2 + V(\mathbf{q})$. In this case we can introduce the conservative force $\mathbf{f}(t) = -\frac{\partial V(\mathbf{q}(t))}{\partial \mathbf{q}}$ which results in the following Position Verlet integrator:

$$
\begin{aligned}
\mathbf{q}(h/2) &= \mathbf{q}(0) + h/2 \, \mathbf{p}(0) \\
\mathbf{p}(h) &= \mathbf{p}(0) + h \, \mathbf{f}(h/2) \\
\mathbf{q}(h) &= \mathbf{q}(h/2) + h/2 \, \mathbf{p}(h).
\end{aligned}
\tag{9}
$$

This Position Verlet formulation will be our go to reversible integrator in the rest of the paper. We note that these integrators are not unconditionally stable. A strict stability bound is hard to establish because the forces can be non-linear. A linear stability analysis however suggest the bound $h\omega < 1$, where ω is the highest "frequency" of the linear system. Before we present our implementation we discuss how real numbers are represented in computers.

3   Float versus Fixed

Once we have our reversible integrator our job seems to be done. Usually we implement the integrator in some programming language that has some built in floating point data type. We run the simulation forward in time for N steps and

then run it backwards ($h \leftarrow -h$) for N steps. Ideally, we should return to the initial state exactly. This is usually not the case. As James Gosling the creator of *Java* once famously said "95% of folks out there are completely clueless about floating-point." Floats are tricky. They do not behave like the ideal real numbers they try to represent. They have the advantage that they can handle a wide range of magnitudes as they are based on scientific notation. However, the basic properties of real numbers are often not satisfied, for example $0.1 + 0.2 \neq 0.3$. Integers do not suffer from this problem as they are discrete.

Enter the *Fixed Point* numbers. This format was popular before floats were implemented in hardware. They work well if your data is bounded. Since our integrators are bounded for reasonable time step sizes h we can assume without loss of generality that our reals lie in the interval $[1, -1]$. We can then approximate a real number R by the following integer I:

$$I = \lfloor R \times \mathrm{BIG\_INT} \rfloor, \tag{10}$$

where BIG_INT is a large integer and $\lfloor \cdot \rfloor$ is the integer part of a real number. The only operation needed in our integration algorithm is addition as shown below in the implementation. To get a real number from a fixed number, we simply "invert" Eq. 10:

$$R = I \ / \ \mathrm{BIG\_INT}.$$

Actually, our implementation is hybrid and uses floating points to compute forces and fixed points when integrating.

4   Implementation

In the following C++ code only the integration steps use the fixed format. Both the coordinates and the velocities are represented in fixed format. The time step can be any arbitrary float. The simulation is completely controlled by the force function which uses only floats. Existing simulations can therefore easily be implemented in this framework. We represent our fixed points as 64 bit integers "int64_t", a type standardized since C++11.

```
typedef int64_t fixed;
const fixed c_max_fixed = 0x1000000000000000;

static float x2f(const fixed i_f){
    return i_f / (float)c_max_fixed;
```

```
}
static fixed f2x(const float i_f) {
    return (fixed)(i_f * c_max_fixed);
}

static fixed * x, * v;
static int size;
// define your favorite force field here.
static float force(const int i, const float i_x);

void integrate(const float h) {
    const float h2 = 0.5f * h;

    for (int i = 0; i < size; i++) {
        x[i] += f2x( h2 * x2f(v[i]) );
    }
    for (int i = 0; i < size; i++) {
        float f = force( i, x2f(x[i]) );
        v[i] += f2x( h * f );
    }
    for (int i = 0; i < size; i++) {
        x[i] += f2x( h2 * x2f(v[i]) );
    }
}
```

We also implemented the integrator in *JavaScript (JS)* to create a web based demo that is easily shared. Ironically, we were faced with the problem that JS does not differentiate between integers and float numbers. It only has the monolithic *Number* type. We found a solution by treating fixed point numbers as strings. We lost some efficiency of course. The implementation is as follows:

```
var XN = 1000000000;
function X2F(x) { return (+x) / XN; }
function F2X(f) { ((f*XN)|0).toString(); }
function XADD(x1, x2) { (+x1 + +x2).toString(); }
```

The code is a bit obfuscated but JS programmers will recognize the hack to get an integer from a Number by taking the bitwise "or" with zero. The "+" operator transforms a string into a Number. See the accompanying *HTML* file from the link given below for the entire implementation. We would like to hear from readers who have a better solution. Note that this is just a quirk of the JS language. For most typed languages, like in our C++ implementation above,
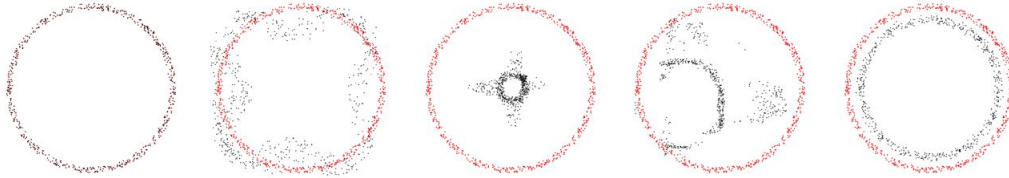
Figure 1: Gravitational collapse of a ring and back.

there is no performance hit.

5 Results

We implemented the exact bitwise integrator for various forces. In each example we run the simulation forward in time and then reverse the time step h and observe that the initial state is exactly recovered. Figure 1 shows 5 frames of a particle system which evolves under pair-wise gravitational-like forces. The initial ring configuration first collapses and after a time reversal exactly returns to its initial state.

We also implemented a chaotic pendulum simulation in JavaScript so it can easily be shared on the internet. You can find the demo on the following web page: josstam.com/reversible. The html source is self-contained and includes all the JS code. The readers should feel free to experiment with the code.

6 Application: Reversible Adjoint / Neural ODEs

Reversible integrators can play an important role in optimization and machine learning. First, we have to introduce some ideas from optimization. We provide a very concise derivation of the so called *adjoint method* and then apply it to our Hamiltonian framework.

The goal in optimization and machine learning is to find an optimal set of controls/weights θ that minimize/maximize some *cost* function J. The function to optimize usually depends on a state $u(\theta)$ which satisfies some equation/constraint $E(u, \theta)$. This is a constrained optimization problem.

$$\text{Find:} \quad \theta^* = \text{argmin}_\theta \ J(u(\theta)) \quad \text{such that} \quad E(u, \theta) = 0. \tag{11}$$

Two important examples follow. (1) A Vanilla *Neural Network* computes an output $y = f(x, \theta)$ from a set of node values x and weights θ, the state is $u = (x, y)$, the constraint is $E = -y + f(x, \theta) = 0$ and J is the cost function. (2) An *Ordinary Differential Equation (ODE)* also fits this framework. In this case the state $u(t, \theta)$ is some continuous quantity that evolves over time and depends on continuous controls $\theta(t)$ like external forces. The function J models some goal we want to achieve, like matching a keyframe. The state u satisfies an ODE: $E = -\dot{u} + f(u, \theta)$. The "Neural ODEs" paper combines these two examples in a clever way [2].

Equation 11 can be solved for the controls θ by many optimization techniques [10]. Usually they involve the gradient of the cost function with respect to the controls.

$$\delta J := \frac{dJ}{d\theta} = \frac{\partial J}{\partial u}\frac{du}{d\theta} =: J_u \, \delta u. \tag{12}$$

The gradient depends on the differential δu for which we can derive an equation by differentiating $E = 0$:

$$0 = \delta E = \frac{dE}{d\theta} = \frac{\partial E}{\partial u}\frac{du}{d\theta} + \frac{\partial E}{\partial \theta} \quad \text{or} \quad E_u \, \delta u = -E_\theta. \tag{13}$$

This is a linear equation for δu, whose solution we can then plug into Eq. 12 to get our desired δJ. This equation becomes expensive to solve in the presence of many controls like in deep learning which have many weights.

Fortunately, there is an alternative known as the *adjoint method*. Instead of solving 13 we solve an *adjoint* equation involving a new variable a, known as a *Lagrange multiplier* or *adjoint variable*.

$$(E_u)^* \, a = -J_u^* \quad \text{and} \quad \delta J' = a^* E_\theta. \tag{14}$$

Where "*" is the adjoint operation (transpose for vectors and matrices). The adjoint equation is independent of the controls. The fact that $\delta J' = \delta J$ can be established in a one liner proof:

$$\delta J = J_u \delta u = -\left(E_u^* a\right)^* \delta u = -a^* E_u \delta u = a^* E_\theta = \delta J'. \tag{15}$$

For the case of an ODE $(d/dt^* = -d/dt)$ we get in agreement with [2]:

$$-\dot{a} = \left(\frac{\partial f(u, \theta)}{\partial u}\right)^* a + \left(\frac{\partial J}{\partial u}\right)^*.$$
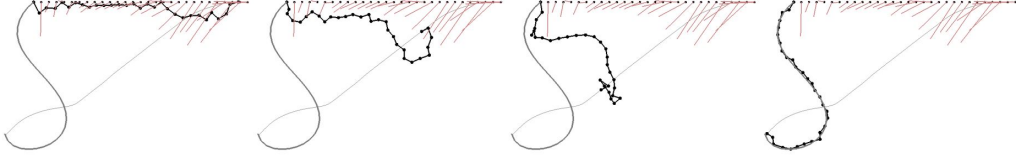
Figure 2: Chain controlled to hit the "S" target.

This is a linear ODE that runs in reverse. It depends on the values of u in the forward simulation. Usually these states are stored like in neural networks or dealt with using clever check-pointing techniques. For our reversable integrators we do not have that problem since we can retrace our path along with the adjoint equation. More concretely, let $\mathbf{a} = (\hat{\mathbf{q}}, \hat{\mathbf{p}})^T$ be the adjoint state to our coordinate/momentum coordinates. Then the adjoint Position Verlet (Eq. 9) is:

$$
\begin{aligned}
\hat{\mathbf{p}}(-h/2) &= \hat{\mathbf{p}}(-h) - h/2\, \hat{\mathbf{q}}(-h) \\
\hat{\mathbf{q}}(0) &= \hat{\mathbf{q}}(-h) - h\, \left(\frac{\partial \mathbf{f}}{\partial \mathbf{q}}(-h/2) \right)^* \hat{\mathbf{p}}(-h/2) \qquad (16) \\
\hat{\mathbf{p}}(0) &= \hat{\mathbf{p}}(-h/2) - h/2\, \hat{\mathbf{q}}(0).
\end{aligned}
$$

Interestingly, this is a Velocity Verlet integration for the adjoint. Similarly, if we had started with a Velocity Verlet integration we would have obtained a Position Verlet integration scheme for the adjoint.

We used this framework to keyframe control a chaotic pendulum to match a keyframe similarly to [9, 14]. Our controls are the initial velocities of the chain. A sequence of frames is shown in Figure 2. The controls are shown in red.

7   Conclusion and Future Work

In this paper we have introduced an exact reversible integrator for Hamiltonian systems. Hopefully, we have given enough theoretical background and actual code to make this technique understandable and useful. We also showed how to use this solver to efficiently solve a class of optimization problems using the adjoint method.

In future work we would like to explore applications in machine learning, e.g. reversible neural networks. Also we would like to extend this integrator to non-Hamiltonian reversible systems. Another challenge is to derive an exact reversible integrator for the Euler equations of fluid dynamics.

In general we hope to inspire people to explore the interchange between techniques in machine learning and simulation.

References

[1] *IEEE Standard for Floating-Point Arithmetic*. 2019.

[2] R. T. Q. Chen, Y. Rubanova, J. Bettencourt, and D. Duvenaud. Neural ordinary differential equations. In *NIPS'18: Proceedings of the 32nd International Conference on Neural Information Processing Systems*, page 6572–6583, 2018.

[3] P. Coveney. *The Arrow Of Time: A Voyage Through Science To Solve Time's Greatest Mystery*. Ballantine Books, 1992.

[4] M. B. Giles and N. A. Pierce. An introduction to the adjoint method to design. *Flow, Turbulence and Combustion*, 65:393–415, 2000.

[5] E. Hairer, G. Wanner, and C. Lubich. *Geometric Numerical Integration*. Springer Verlag, 2002.

[6] W. G. Hoover. *Time Reversibility, Computer Simulation, And Chaos (Advanced Nonlinear Dynamics)*. World Scientific, 1999.

[7] J. L. Lagrange. *Mécanique Analytique*. Desaint, Paris, 1788.

[8] D. Levesque and L. Verlet. Molecular dynamics and time reversibility. *Journal of Statistical Physics*, 72:519–537, 1993.

[9] A. McNamara, A. Treuille, Z. Popović, and J. Stam. Fluid control using the adjoint method. *Transaction on Graphics (TOG). Proceedings of ACM SIGGRAPH*, 23(3):449–456, 2004.

[10] J. Nocedal and S. J. Wrights. *Numerical Optimization (2nd ed.)*. Springer, Berlin, 2006.

[11] Hanno Rein and Daniel Tamayo. Janus: a bit-wise reversible integrator for n-body dynamics. *Monthly Notices of the Royal Astronomical Society*, 473:3351–3357, 2017.

[12] D. Rumelhart, G. Hinton, and R. Williams. Learning representations by back-propagating errors. *Nature*, 323:533–536, 1986.

[13] M. Tuckerman, B. J. Berne, and G. J. Martyna. Reversible multiple time scale molecular dynamics. *Journal of Chemical Physics*, 97(3):1990–2001, 1992.

[14] C. Wojtan, P. J. Mucha, and G. Turk. Keyframe control of complex particle systems using the adjoint method. *ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, pages 15–23, 2006.