Verlet integration

Verlet integration (French pronunciation: [vɛʁ'lɛ]) is a numerical method used to integrate Newton's equations of motion[1] . It is frequently used to calculate trajectories of particles in molecular dynamics simulations and video games. The verlet integrator offers greater stability than the much simpler Euler method, as well as other properties that are important in physical systems such as time-reversibility and area preserving properties. At first it may seem natural to simply calculate trajectories using Euler integration. However, this kind of integration suffers from many problems, as discussed at Euler integration. Stability of the technique depends fairly heavily upon either a uniform update rate, or the ability to accurately identify positions at a small time delta into the past. Verlet integration was used by Carl Størmer to compute the trajectories of particles moving in a magnetic field (hence it is also called **Størmer's method**) and was popularized in molecular dynamics by French physicist Loup Verlet in 1967.

Basic Verlet

Newton's equation of motion for conservative physical systems is

$$M\ddot{\mathbf{x}}(t) = F(\mathbf{x}(t)) = -\nabla V(\mathbf{x}(t))$$

or individually

$$m_k\ddot{\vec{x}}_k(t) = F_k(\mathbf{x}(t)) = -\nabla_{\vec{x}_k}V(\mathbf{x}(t))$$

where

- t is the time,
- $\mathbf{x}(t) = (\vec{x}_1(t), \ldots, \vec{x}_N(t))$ is the ensemble of the position vector of N objects,
- V is the scalar potential function,
- F is the negative gradient of the potential giving the ensemble of forces on the particles,
- M is the mass matrix, typically diagonal with blocks with mass m_k for every particle.

This setting allows to express problems in molecular dynamics and N-body planetary or stellar simulations, among others.

After a transformation to bring the mass to the right side and forgetting the structure of multiple particles, the equation may be simplified to

$$\ddot{\vec{x}}(t) = A(\vec{x}(t))$$

with some suitable vector valued function A representing the position dependent acceleration. Typically, an initial position $\vec{x}(0) = \vec{x}_0$ and an initial velocity $\vec{v}(0) = \dot{\vec{x}}(0) = \vec{v}_0$ are also given.

To discretize and numerically solve this initial value problem, a time step $\Delta t > 0$ is chosen and the sampling point sequence $t_n = n\Delta t$ considered. The task is to construct a sequence of points $\vec{x}_n$ that closely follow the points $\vec{x}(t_n)$ on the trajectory of the exact solution.

Where Euler's Method uses the forward difference approximation to the first derivative in differential equations of order one, Verlet Integration can be seen as using the central difference approximation to the second derivative:

$$\frac{\Delta^2 \vec{x}_n}{\Delta t^2} = \frac{\frac{\vec{x}_{n+1}-\vec{x}_n}{\Delta t} - \frac{\vec{x}_n-\vec{x}_{n-1}}{\Delta t}}{\Delta t} = \frac{\vec{x}_{n+1} - 2\vec{x}_n + \vec{x}_{n-1}}{\Delta t^2} = \vec{a}_n = A(\vec{x}_n)$$

The Verlet algorithm[2] uses this equation to obtain the next position vector from the previous two as

$$\vec{x}_{n+1} = 2\vec{x}_n - \vec{x}_{n-1} + \vec{a}_n \Delta t^2, \quad \vec{a}_n = A(\vec{x}_n),$$

without using the velocity. The time symmetry inherent in the method reduces the level of errors introduced into the integration by calculating the position at the next time step. The error is quantified by inserting the exact values $\vec{x}(t_{n-1}), \vec{x}(t_n), \vec{x}(t_{k+n})$ into the iteration and computing the Taylor expansions at time $t = t_n$ of the position vector $\vec{x}(t \pm \Delta t)$ in different time directions.

$$\vec{x}(t + \Delta t) = \vec{x}(t) + \vec{v}(t)\Delta t + \frac{\vec{a}(t)\Delta t^2}{2} + \frac{\vec{b}(t)\Delta t^3}{6} + \mathcal{O}(\Delta t^4)$$

$$\vec{x}(t - \Delta t) = \vec{x}(t) - \vec{v}(t)\Delta t + \frac{\vec{a}(t)\Delta t^2}{2} - \frac{\vec{b}(t)\Delta t^3}{6} + \mathcal{O}(\Delta t^4),$$

where $\vec{x}$ is the position, $\vec{v} = \dot{\vec{x}}$ the velocity, $\vec{a} = \ddot{\vec{x}}$ the acceleration and $\vec{b}$ the jerk (third derivative of the position with respect to the time) t .

Adding these two expansions gives

$$\vec{x}(t + \Delta t) = 2\vec{x}(t) - \vec{x}(t - \Delta t) + \vec{a}(t)\Delta t^2 + \mathcal{O}(\Delta t^4).$$

We can see that the first and third-order terms from the Taylor expansion cancel out, thus making the Verlet integrator an order more accurate than integration by simple Taylor expansion alone.

Caution should be applied to the fact that the acceleration here is computed from the exact solution, $\vec{a}(t) = A(\vec{x}(t))$, whereas in the iteration it is computed at the central iteration point, $\vec{a}_n = A(\vec{x}_n)$. In computing the global error, that is the distance between exact solution and approximation sequence, those two terms do not cancel exactly.

Note that at the start of the Verlet-iteration at step $n = 1$, time $t = t_1 = \Delta t$, computing $\vec{x}_2$, one already needs the position vector $\vec{x}_1$ at time $t = t_1$. At first sight this could give problems, because the initial conditions are known only at the initial time. However, from these the acceleration $\vec{a}_0 = A(\vec{x}_0)$ is known, and a suitable approximation for the first time step position can be obtained using the Taylor polynomial of degree two:

$$\vec{x}_1 = \vec{x}_0 + \vec{v}_0\Delta t + \tfrac{1}{2}\vec{a}_0\Delta t^2 \approx \vec{x}(\Delta t) + \mathcal{O}(\Delta t^3).$$

The error on the first time step calculation then is of order $\mathcal{O}(\Delta t^3)$. This is not considered a problem because on a simulation of over a large amount of timesteps, the error on the first timestep is only a negligible small amount of the total error, which at time t_n is of the order $\mathcal{O}(e^{Lt_n}\Delta t^2)$, both for the distance of the position vectors $\vec{x}_n$ to $\vec{x}(t_n)$ as for the distance of the divided differences $\frac{\vec{x}_{n+1} - \vec{x}_n}{\Delta t}$ to $\frac{\vec{x}(t_{n+1}) - \vec{x}(t_n)}{\Delta t}$. Moreover, to obtain this second order global error, the initial error needs to be of at least third order.

The velocities are not explicitly given in the Basic Verlet equation, but often they are necessary for the calculation of certain physical quantities like the kinetic energy. This can create technical challenges in molecular dynamics simulations, because kinetic energy and instantaneous temperatures at time t cannot be calculated for a system until the positions are known at time $t + \Delta t$. This deficiency can either be dealt with using the Velocity Verlet algorithm, or estimating the velocity using the position terms and the mean value theorem:

$$\vec{v}(t) = \frac{\vec{x}(t + \Delta t) - \vec{x}(t - \Delta t)}{2\Delta t} + \mathcal{O}(\Delta t^2).$$

Note that this velocity term is a step behind the position term, since this is for the velocity at time t , not $t + \Delta t$, meaning that $\vec{v}_n = \frac{\vec{x}_{n+1} - \vec{x}_{n-1}}{2\Delta t}$ is an order two approximation to $\vec{v}(t_n)$. With the same argument, but halving the time step, $\vec{v}_{n+1/2} = \frac{\vec{x}_{n+1} - \vec{x}_n}{\Delta t}$ is an order two approximation to $\vec{v}(t_{n+1/2})$, with $t_{n+1/2} = t_n + \tfrac{1}{2}\Delta t$.

One can shorten the interval to approximate the velocity at time $t + \Delta t$ at the cost of accuracy:

$$\vec{v}(t + \Delta t) = \frac{\vec{x}(t + \Delta t) - \vec{x}(t)}{\Delta t} + \mathcal{O}(\Delta t).$$

Velocity Verlet

A related, and more commonly used, algorithm is the **Velocity Verlet** algorithm [3], similar to the Leapfrog method, except that the velocity and position are calculated at the same value of the time variable (Leapfrog does not, as the name suggests). This uses a similar approach but explicitly incorporates velocity, solving the first-timestep problem in the Basic Verlet algorithm:

$$\vec{x}(t + \Delta t) = \vec{x}(t) + \vec{v}(t)\,\Delta t + \frac{1}{2}\,\vec{a}(t)\Delta t^2$$

$$\vec{v}(t + \Delta t) = \vec{v}(t) + \frac{\vec{a}(t) + \vec{a}(t + \Delta t)}{2}\Delta t$$

It can be shown that the error on the Velocity Verlet is of the same order as the Basic Verlet. Note that the Velocity algorithm is not necessarily more memory consuming, because it's not necessary to keep track of the velocity at every timestep during the simulation. The standard implementation scheme of this algorithm is:

1. Calculate: $\vec{v}\left(t + \frac{1}{2}\Delta t\right) = \vec{v}(t) + \frac{1}{2}\vec{a}(t)\,\Delta t$
2. Calculate: $\vec{x}(t + \Delta t) = \vec{x}(t) + \vec{v}\left(t + \frac{1}{2}\Delta t\right)\,\Delta t$
3. Derive $\vec{a}(t + \Delta t)$ from the interaction potential using $\vec{x}(t + \Delta t)$
4. Calculate: $\vec{v}(t + \Delta t) = \vec{v}\left(t + \frac{1}{2}\Delta t\right) + \frac{1}{2}\vec{a}(t + \Delta t)\Delta t$.

Eliminating the half-step velocity, this algorithm may be shortened to

1. Calculate: $\vec{x}(t + \Delta t) = \vec{x}(t) + \vec{v}(t)\,\Delta t + \frac{1}{2}\vec{a}(t)\,\Delta t^2$
2. Derive $\vec{a}(t + \Delta t)$ from the interaction potential using $\vec{x}(t + \Delta t)$
3. Calculate: $\vec{v}(t + \Delta t) = \vec{v}(t) + \frac{1}{2}\left(\vec{a}(t) + \vec{a}(t + \Delta t)\right)\Delta t$.

Note, however, that this algorithm assumes that acceleration $\vec{a}(t + \Delta t)$ only depends on position $\vec{x}(t + \Delta t)$, and does not depend on velocity $\vec{v}(t + \Delta t)$.

One might note that the long-term results of **Velocity Verlet**, and similarly of **Leapfrog** are one order better than the Semi-implicit Euler method. The algorithms are almost identical up to a shifted by half of a timestep in the velocity. This is easily proven by rotating the above loop to start at Step 3 and then noticing that the acceleration term in Step 1 could be eliminated by combining Steps 2 and 4. The only difference is that the midpoint velocity in **Velocity Verlet** is considered the final velocity in Semi-implicit Euler method.

The global error of all Euler methods is of order one, whereas the global error of this method is, similar to the Midpoint method, of order two. Additionally, if the acceleration indeed results from the forces in a conservative mechanical or Hamiltonian system, the energy of the approximation essentially oscillates around the constant energy of the exactly solved system, with a global error bound again of order one for semi-explicit Euler and order two for Verlet-leapfrog. The same goes for all other conservered quantities of the system like linear or angular momentum, that are always preserved or nearly preserved in a symplectic integrator.[4]

Error terms

The local error in position of the Verlet integrator is $O(\Delta t^4)$ as described above, and the local error in velocity is $O(\Delta t^2)$.

The global error in position, in contrast, is $O(\Delta t^2)$ and the global error in velocity is $O(\Delta t^2)$. These can be derived by noting the following:

$$\text{error}\big(x(t_0 + \Delta t)\big) = O(\Delta t^4)$$

and

$$x(t_0 + 2\Delta t) = 2x(t_0 + \Delta t) - x(t_0) + \Delta t^2 x''(t_0 + \Delta t) + O(\Delta t^4)$$

Therefore:

$$\text{error}\big(x(t_0 + 2\Delta t)\big) = 2\text{error}\big(x(t_0 + \Delta t)\big) + O(\Delta t^4) = 3\,O(\Delta t^4)$$

Similarly:

$$\text{error}\big(x(t_0 + 3\Delta t)\big) = 6\,O(\Delta t^4)$$
$$\text{error}\big(x(t_0 + 4\Delta t)\big) = 10\,O(\Delta t^4)$$
$$\text{error}\big(x(t_0 + 5\Delta t)\big) = 15\,O(\Delta t^4)$$

Which can be generalized to (it can be shown by induction, but it is given here without proof):

$$\text{error}\big(x(t_0 + n\Delta t)\big) = \frac{n(n+1)}{2}\,O(\Delta t^4)$$

If we consider the global error in position between $x(t)$ and $x(t + T)$, where $T = n\Delta t$, it is clear that:

$$\text{error}\big(x(t_0 + T)\big) = \left(\frac{T^2}{2\Delta t^2} + \frac{T}{2\Delta t}\right)O(\Delta t^4)$$

And therefore, the global (cumulative) error over a constant interval of time is given by:

$$\text{error}\big(x(t_0 + T)\big) = O(\Delta t^2)$$

Because the velocity is determined in a non-cumulative way from the positions in the Verlet integrator, the global error in velocity is also $O(\Delta t^2)$.

In molecular dynamics simulations, the global error is typically far more important than the local error, and the Verlet integrator is therefore known as a second-order integrator.

Constraints

The most notable thing that is now easier due to using Verlet integration rather than Eulerian is that constraints between particles are very easy to do. A constraint is a connection between multiple points that limits them in some way, perhaps setting them at a specific distance or keeping them apart, or making sure they are closer than a specific distance. Often physics systems use springs between the points in order to keep them in the locations they are supposed to be. However, using springs of infinite stiffness between two points usually gives the best results coupled with the verlet algorithm. Here's how:

$$d_1 = x_2^{(t)} - x_1^{(t)}$$
$$d_2 = \|d_1\|$$
$$d_3 = \frac{d_2 - r}{d_2}$$
$$x_1^{(t+\Delta t)} = \tilde{x}_1^{(t+\Delta t)} + \frac{1}{2}d_1 d_3$$
$$x_2^{(t+\Delta t)} = \tilde{x}_2^{(t+\Delta t)} - \frac{1}{2}d_1 d_3$$

The $x_i^{(t)}$ variables are the positions of the points i at time t, the $\tilde{x}_i^{(t)}$ are the *unconstrained* positions (*i.e.* the point positions before applying the constraints) of the points i at time t, the d variables are temporary (they are added for optimization as the results of their expressions are needed multiple times), and r is the distance that is supposed to be between the two points. Currently this is in one dimension; however, it is easily expanded to two or three. Simply find the delta (first equation) of each dimension, and then add the deltas squared to the inside of the square root of the second equation (Pythagorean theorem). Then, duplicate the last two equations for the number of dimensions there are. This is where verlet makes constraints simple - instead of say, applying a velocity to the points that would eventually satisfy the constraint, you can simply position the point where it should be and the verlet integrator takes care of the rest.

Problems, however, arise when multiple constraints position a vertex. One way to solve this is to loop through all the vertices in a simulation in a criss cross manner, so that at every vertex the constraint relaxation of the last vertex is already used to speed up the spread of the information. Either use fine time steps for the simulation, use a fixed number of constraint solving steps per time step, or solve constrains until they are met by a specific deviation.

When approximating the constraints locally to first order this is the same as the Gauss−Seidel method. For small matrices it is known that LU decomposition is faster. Large systems can be divided into clusters (for example: each ragdoll=cluster). Inside clusters the LU method is used, between clusters the Gauss−Seidel method is used. The matrix code can be reused: The dependency of the forces on the positions can be approximated locally to first order, and the verlet integration can be made more implicit.

For big matrices sophisticated solvers (look especially for "The sizes of these small dense matrices can be tuned to match the sweet spot" in [5]) for sparse matrices exist, any self made Verlet integration has to compete with these. The usage of (clusters of) matrices is not generally more precise or stable, but addresses the specific problem, that a force on one vertex of a sheet of cloth should reach any other vertex in a low number of time steps even if a fine grid is used for the cloth [6] (link needs refinement) and not form a sound wave.

Another way to solve Holonomic constraints is to use constraint algorithms.

Collision reactions

One way of reacting to collisions is to use a penalty-based system which basically applies a set force to a point upon contact. The problem with this is that it is very difficult to choose the force imparted. Use too strong a force and objects will become unstable, too weak and the objects will penetrate each other. Another way is to use projection collision reactions which takes the offending point and attempts to move it the shortest distance possible to move it out of the other object.

The Verlet integration would automatically handle the velocity imparted via the collision in the latter case, however note that this is not guaranteed to do so in a way that is consistent with collision physics (that is, changes in momentum are not guaranteed to be realistic). Instead of implicitly changing the velocity term, you would need to explicitly control the final velocities of the objects colliding (by changing the recorded position from the previous time step).

The two simplest methods for deciding on a new velocity are perfectly elastic collisions and inelastic collisions. A slightly more complicated strategy that offers more control would involve using the coefficient of restitution.

Applications

The Verlet equations can also be modified to create a very simple damping effect (for instance, to emulate air friction in computer games):

$$x(t + \Delta t) = (2 - f)x(t) - (1 - f)x(t - \Delta t) + a(t)\Delta t^2.$$

Where f is a number representing the fraction of the velocity per update that is lost to friction (0-1).

Literature

[1] Verlet, Loup (1967). "Computer "Experiments" on Classical Fluids. I. Thermodynamical Properties of Lennard−Jones Molecules" (http:// link.aps.org/doi/10.1103/PhysRev.159.98). *Physical Review* **159**: 98−103. doi:10.1103/PhysRev.159.98. .

[2] webpage (http://www.fisica.uniud.it/~ercolessi/md/md/node21.html) with a description of the method

[3] http://www.fisica.uniud.it/~ercolessi/md/md/node21.html

[4] Hairer, Ernst; Lubich, Christian; Wanner, Gerhard (2003). "Geometric numerical integration illustrated by the Störmer/Verlet method" (http:/ /citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.7.7106). *Acta Numerica* **12**: 399−450. doi:10.1017/S0962492902000144. .

[5] http://crd.lbl.gov/~xiaoye/SuperLU/superlu_ug.pdf

[6] http://www.cs.cmu.edu/~baraff/papers/index.html

External links

- Verlet Integration Demo and Code as a Java Applet (http://verlet.googlecode.com/)
- Advanced Character Physics by Thomas Jakobsen (http://www.gamasutra.com/resource_guide/20030121/jacobson_pfv.htm)
- The Verlet algorithm (http://www.fisica.uniud.it/~ercolessi/md/md/node21.html)
- Theory of Molecular Dynamics Simulations (http://www.ch.embnet.org/MD_tutorial/pages/MD.Part1.html) - bottom of page

Article Sources and Contributors

Verlet integration *Source*: http://en.wikipedia.org/w/index.php?oldid=434357461 *Contributors*: Anna Lincoln, Arnero, Beasticles, BenFrantzDale, Berland, Charles Matthews, Colinb, David Eppstein, Delldot, DivisionByZero, Encyclops, Ferrarirt, Fish-Face, Fyo, Giftlite, Harold f, Indeterminatus, JHunterJ, JackSchmidt, Janek Kozicki, Jheriko, Jitse Niesen, Justin W Smith, Kakofonous, Kwamikagami, Lingwitt, Lnemzer, LutzL, Malcohol, Mattopia, Michael Hardy, Mjhsieh, Monstrim, Nazdrovje, NickAlbright, Numsgil, Oerjan, Olaf Davis, Omnipaedista, Pedrito, Phil Boswell, Pmdboi, Pomte, Rjwilmsi, Rubik123, Sander123, Seidenstud, Selket, Sodaplayer, Thegeneralguy, TotientDragooned, Vump, Zalgo, 116 anonymous edits

License