Ordinary Differential Equations

Ramses van Zon

PHY1610 Winter 2025

ADVANCED RESEARCH COMPUTING at the UNIVERSITY OF TORONTO

Ordinary Differential Equations

- Are equations with derivatives with respect to 1 variable, e.g.

$$\frac{dx}{dt} = f(x,t)$$

- There can be more than one such equation, e.g.

$$\frac{dx^{(1)}}{dt} = f^{(1)}(x^{(1)}, x^{(2)}, t); \quad \frac{dx^{(2)}}{dt} = f^{(2)}(x^{(1)}, x^{(2)}, t)$$

- The derivative can be of higher order to, e.g. $\dfrac{d^2x}{dt^2} = f(x,t)$

  But this can be written, by setting $x^{(1)} = x$, $x^{(2)} = dx/dt$, to

$$\frac{dx^{(1)}}{dt} = x^{(2)}; \qquad \frac{dx^{(2)}}{dt} = f(x^{(1)}, t)$$

ODE Examples

Lotka–Volterra (predator/pray)

$$\frac{dx^{(1)}}{dt} = x^{(1)}(\alpha - \beta x^{(2)})$$
$$\frac{dx^{(2)}}{dt} = -x^{(2)}(\gamma - \delta x^{(1)})$$

Harmonic oscillator

$$\frac{dx^{(1)}}{dt} = x^{(2)}$$
$$\frac{dx^{(2)}}{dt} = -x^{(1)}$$

Rate equations (chemistry))

$$\begin{aligned}
\frac{dx^{(1)}}{dt} &= -2k_1[x^{(1)}]^2x^{(2)} + 2k_2[x^{(3)}]^2 \\
\frac{dx^{(2)}}{dt} &= -k_1[x^{(1)}]^2x^{(2)} + k_2[x^{(3)}]^2 \\
\frac{dx^{(3)}}{dt} &= 2k_1[x^{(1)}]^2x^{(2)} - 2k_2[x^{(3)}]^2
\end{aligned}$$

Lorenz system (weather)

$$\begin{aligned}
\frac{dx^{(1)}}{dt} &= \sigma(x^{(2)} - x^{(1)}) \\
\frac{dx^{(2)}}{dt} &= x^{(1)}(\rho - x^{(3)}) - x^{(2)} \\
\frac{dx^{(3)}}{dt} &= x^{(1)}x^{(2)} - \beta x^{(3)}
\end{aligned}$$

Numerical approaches

Start from the general form:

$$\frac{dx^{(i)}}{dt} = f(x^{(1)}, x^{(2)}, ..., t)$$

- Algorithms for numerically solving ODEs are called integrators.
- All integrators will evaluate f at discrete points t_0, t_1, ...
- Initial conditions: specify $x^{(i)}(t_0)$.
- The time step is typically denoted with h.
- Consecutive points may have a fixed step size $h = t_{k+1} - t_k$ or may be adaptive.

Desirable qualities for an integrator

- *Accuracy*
- *Efficiency*
- *Stability*
- Respect physical laws, e.g.

  Time reversal symmetry
  Conservation of energy
  Conservation of linear momentum
  Conservation of angular momentum
  Conservation of phase space volume

The most efficient algorithm is then the one that allows the largest possible time step for a given level of **accuracy**, while maintaining **stability** and preserving **conservation laws**.

ODE solvers: Forward Euler

To solve:

$$\frac{dx}{dt} = f(t, x)$$

we could take the simple approximation:

$$x_{n+1} \approx x_n + hf(x_n, t_n) \qquad \text{"forward Euler"}$$

Why?

$$x(t_n + h) = x(t_n) + h\frac{dx}{dt}(t_n) + \mathcal{O}(h^2)$$

So:

$$x(t_n + h) = x(t_n) + hf(x_n, t_n) + \mathcal{O}(h^2)$$

So when taking small time steps, this should be accurate.

Accuracy of the forward Euler method

$$x(t_n + h) = x(t_n) + hf(x_n, t_n) + \mathcal{O}(h^2)$$

- $\mathcal{O}(h^2)$ is the local error, i.e., the error in each time step.
- For given trajectory from $t = t_1$ to t_2, we need $n = (t_2 - t_1)/h$ steps.
- The global error, i.e., the error accumulated over the trajectory, is therefore: $n \times \mathcal{O}(h^2) = \mathcal{O}(h)$
- Not very accurate.

Stability of the forward Euler method

To solve harmonic oscillator:

$$\frac{dx^{(1)}}{dt} = x^{(2)}$$

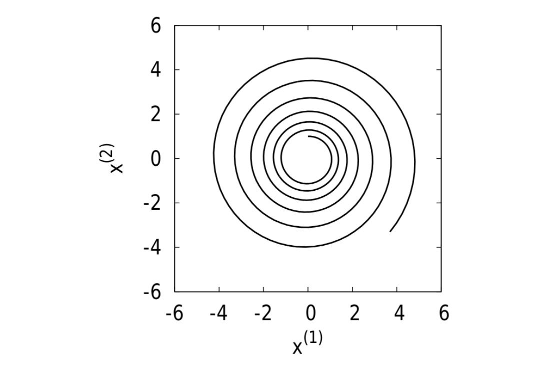
$$\frac{dx^{(2)}}{dt} = -x^{(1)}$$

with forward Euler gives:

$$\begin{pmatrix} x^{(1)}_{n+1} \\ x^{(2)}_{n+1} \end{pmatrix} = \begin{pmatrix} 1 & h \\ -h & 1 \end{pmatrix} \begin{pmatrix} x^{(1)}_{n} \\ x^{(2)}_{n} \end{pmatrix}$$

Stability governed by eigenvalues.

$\lambda_\pm = 1 \pm ih$ of that matrix.

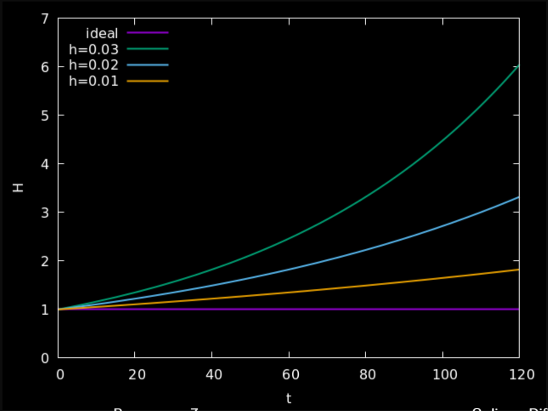


$$|\lambda_\pm| = \sqrt{1+h^2} > 1$$

Unstable for any h!

Monitoring Stability

- For the harmonic oscillator, we know the exact answer, so it's easy to see that the forward Euler integrator is unstable.
- For systems without an exact solution, one may still know that some quantities should be bounded.
- Many physical systems have conserved energy, so we can monitor the energy as a function of time.



Harmonic oscillator:

$$H = \frac{1}{2}[x^{(1)}]^2 + \frac{1}{2}[x^{(1)}]^2$$

So smaller h does help, but in the long run ($t \sim \mathcal{O}(1/h)$), unstable.

ODE solvers: implicit mid-point Euler

Equation to solve:

$$\frac{dx}{dt} = f(x,t)$$

Symmetric simple approximation:

$$x_{n+1} \approx x_n + hf((x_n + x_{n+1})/2, t_n) \qquad \textit{"mid-point Euler"}$$

This is an implicit formula, i.e., has to be solved for x_{n+1}.

Example: Harmonic oscillator

$$\begin{bmatrix} 1 & -\frac{h}{2} \\ \frac{h}{2} & 1 \end{bmatrix} \begin{bmatrix} x^{[1]}_{n+1} \\ x^{[2]}_{n+1} \end{bmatrix} = \begin{bmatrix} 1 & \frac{h}{2} \\ -\frac{h}{2} & 1 \end{bmatrix} \begin{bmatrix} x^{[1]}_{n} \\ x^{[2]}_{n} \end{bmatrix} \Rightarrow \begin{bmatrix} x^{[1]}_{n+1} \\ x^{[2]}_{n+1} \end{bmatrix} = M \begin{bmatrix} x^{[1]}_{n} \\ x^{[2]}_{n} \end{bmatrix}$$

Eigenvalues M are $\lambda_\pm = \frac{(1 \pm ih/2)^2}{1+h^2/4}$ so $|\lambda_\pm| = 1$

Stable for all h!

Implicit methods often more stable and allow larger step size h.

ODE solvers: implicit mid-point Euler

Equation to solve:

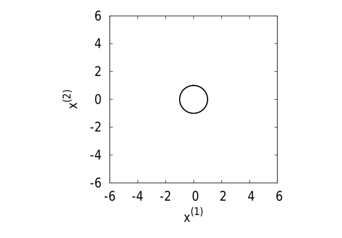
$$\frac{dx}{dt} = f(x,t)$$

Symmetric simple approximation:

$$x_{n+1} \approx x_n + hf((x_n + x_{n+1})/2, t_n) \qquad \text{"mid-point Euler"}$$

This is an implicit formula, i.e., has to be solved for x_{n+1}.

Example: Harmonic oscillator



ODE solvers: Predictor-Corrector

- Computation of new point
- Correction using that new point
- Gear P.C.: keep previous values of x to do higher order Taylor series (predictor), then use f in last point to correct.

  Can suffer from catastrophic cancellation at very low h.
- Runge-Kutta: Refines by using mid-points. 4th order version:

$$
\begin{aligned}
k_1 &= hf(t, x) \\
k_2 &= hf(t + h/2, x + k_1/2) \\
k_3 &= hf(t + h/2, x + k_2/2) \\
k_4 &= hf(t + h, x + k_3) \\
x' &= y + \frac{k_1}{6} + \frac{k_2}{3} + \frac{k_3}{3} + \frac{k_4}{6}
\end{aligned}
$$

Adaptive Step-size Control

Rather than taking a fixed h, we can vary h such that the solution has a certain accuracy.

Methods that adjust the time step as the computation proceeds are known as adaptive methods.

Such an approach needs four components:

1. The basic algorithm for a single h time step,
2. An algorithm to determine the best h time step based on given absolute or relative precision,
3. A evolution algorithm combining these two to take the best possible single time step.
4. A driver routine to step forward in time, using the evolution, for the desired time points.

Don't code this yourself (except for the 'driver')!

Adaptive schemes are implemented in libraries such as the `gsl` and `boost::numeric::odeint`.

ODE example: Van der Pol equation

The Van der Pol oscilator satisfies the following equation:

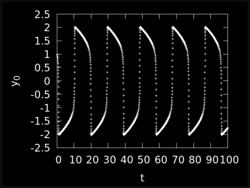
$$\frac{d^2y}{dt^2} - \mu(1-y^2)\frac{dy}{dt} + y = 0$$

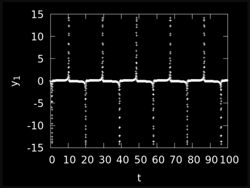
or, writing $y_0 = y =$, $y_1 = dy/dt$,

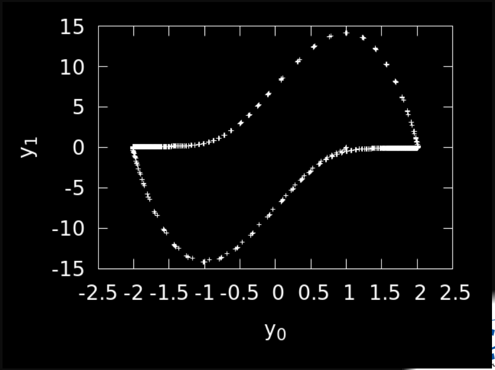
$$\frac{dy_0}{dt} = y_1$$

$$\frac{dy_1}{dt} = -y_0 - \mu(y_0^2 - 1)y_1$$

Solution for $t = 0..100$ starting from $(y_0, y_1) = (1, 0)$







GSL ODE example: Van der Pol equation

```
#include <iostream>
#include <iomanip>
#include <memory>
#include <gsl/gsl_errno.h>
#include <gsl/gsl_odeiv2.h>

const unsigned vdpdim = 2;

int vdprhs(double t, const double y[],
           double f[], void *params)
{
 double mu = *reinterpret_cast<double*>(params);
 f[0] = y[1];
 f[1] = -y[0] - mu*y[1]*(y[0]*y[0] - 1);
 return GSL_SUCCESS;
}
```

```
int main() {
 const gsl_odeiv2_step_type*
   step_type = gsl_odeiv2_step_rk8pd;
 double abstol = 1.e-8;
 auto stepper = std::shared_ptr<gsl_odeiv2_step>
   (gsl_odeiv2_step_alloc(step_type, vdpdim),
    gsl_odeiv2_step_free);
```

```
 auto control = std::shared_ptr<gsl_odeiv2_control>
   (gsl_odeiv2_control_y_new (abstol, 0.0),
    gsl_odeiv2_control_free);
 auto evolver = std::shared_ptr<gsl_odeiv2_evolve>
   (gsl_odeiv2_evolve_alloc(vdpdim),
    gsl_odeiv2_evolve_free);
```

```
 double mu = 10;
 gsl_odeiv2_system sys = {vdprhs, 0, vdpdim, &mu};
```

```
 double t = 0.0;
 double maxt = 100.0;
 double h = 1.e-6;
 double y[vdpdim] = { 1.0, 0.0 };
```

```
 while (t < maxt) {
  int status = gsl_odeiv2_evolve_apply
   (evolver.get(), control.get(), stepper.get(),
    &sys, &t, maxt, &h, y);
```

```
  if (status != GSL_SUCCESS) break;
  std::cout<<std::scientific<<std::setprecision(5)
           <<t<<" "<<y[0]<<" "<<y[1]<<"\n";
 }
}
```

Boost ODE example: Van der Pol equation

```cpp
#include <iostream>
#include <array>
#include <boost/numeric/odeint.hpp>

const unsigned vdpdim = 2;
typedef std::array<double,vdpdim> State;

struct van_der_pol {
 double mu;
 van_der_pol(double mu) : mu(mu) {}
 void operator()(const State &y,
                 State &f, double t) const {
  f[0] = y[1];
  f[1] = -y[0] - mu*y[1]*(y[0]*y[0] - 1);
 }
};

void write_state(const State &y, double t) {
 std::cout<<std::scientific<<std::setprecision(5)
         <<t<<" "<<y[0]<<" "<<y[1]<<"\n";
}
```

```cpp
int main(int argc, char* argv[])
{
 using namespace boost::numeric::odeint;

 double abstol = 1.e-8;
 auto control = make_controlled(abstol, 0.0,
                  runge_kutta_dopri5<State>());
 double mu = 10.0;
 auto system = van_der_pol(mu);

 double t = 0.0;
 double maxt = 100.0;
 double h = 1.e-6;
 State y = { 1.0, 0.0 };

 integrate_adaptive(control, system,
                    y, t, maxt, h,
                    write_state);
}
```

Compilation on Teach

GSL example:

```
$ module load gcc gsl
$ g++ -Wall -Wfatal-errors -g -O3 -c -o gslvdp.o gslvdp.cpp
$ g++ -g -o gslvdp.cpp gslvdp.o -lgsl -lgslcblas
```

Boost:

```
$ module load gcc boost
$ g++ -Wall-Wfatal-errors -g -O3 -c -o boostvdp.o boostvdp.cpp
$ g++ -g -o boostvdp.cpp boostvdp.o
```

(note that boost's ode implementation is header-only).

Special case: Molecular Dynamics

Molecular Dynamics Simulations

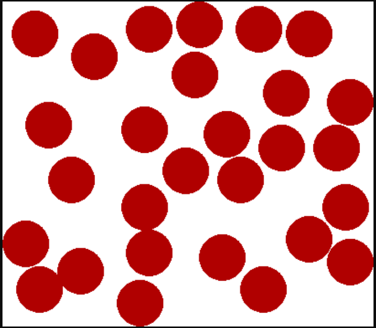
Used in chemical physics, materials science and the modelling of bio-molecules.

- N interacting particles
- $m_i \ddot{r}_i = F_i(r_1, r_2, ;t)\}$

  $+$ initial conditions

What makes this different from other ODEs?

- Hamiltonian dynamics
- Very expensive evaluation of F if N is large
- Large simulation times needed



N-body simulation fall within this class as well; the numerics does not care whether the particles are atoms or stars.

Hamiltonian dynamics

- Molecular Dynamics aims to compute *equilibrium*, *thermodynamic* and *transport* properties of *classical many body systems*.
- Often, the energy is of the form $H = \frac{|p|^2}{2m} + \Phi(r)$ (a.k.a. the Hamiltonian), and is conserved under the dynamics.
- In that case, the systems follows Newton's equations of motion:

$$\dot{r} = \frac{1}{m} p \qquad \dot{p} = F = -\frac{\partial \Phi}{\partial r},$$

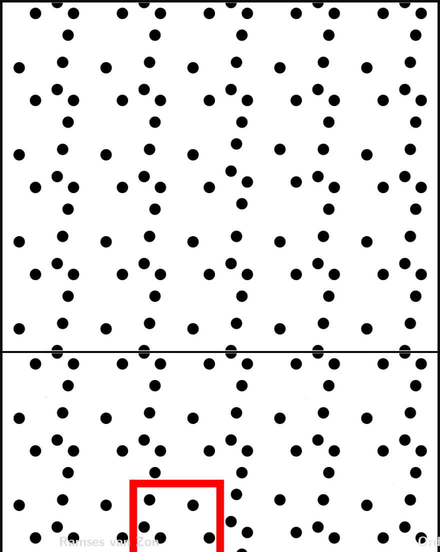
- Potential energy Φ is typically a sum of pair potentials:

$$\Phi(r) = \sum_{(i,j)} \varphi(r_{ij}) = \sum_{i=1}^{N} \sum_{j=1}^{i-1} \varphi(r_{ij}),$$

which entails the following expression for the forces F:

$$\mathbf{F}_i = \sum_{j \neq i} \varphi'(r_{ij}) \frac{\mathbf{r}_j - \mathbf{r}_i}{r_{ij}}$$

Boundary conditions



- Cannot simulate infinite systems.
- Add a **wall**; the box with thick red boundaries is our simulation box.
- But wall gives finite size effects.
- More benign: *Periodic Boundary Conditions*
- Wall becomes a **simulation box**.
- A particle exiting simulation box is put back at the other end.
- Other boxes are periodic images.
  We can *compute* their position when their effect is needed, instead of *storing*.
- Sometimes call "checker board boundary conditions".

Force calculations: cut-off

- A common pair potential between neutral, spherical particles (atoms) is the Lennard-Jones potential

$$\varphi(r) = 4\varepsilon\left[\left(\frac{\sigma}{r}\right)^{12} - \left(\frac{\sigma}{r}\right)^{6}\right],$$

σ is a measure of the range of the potential.

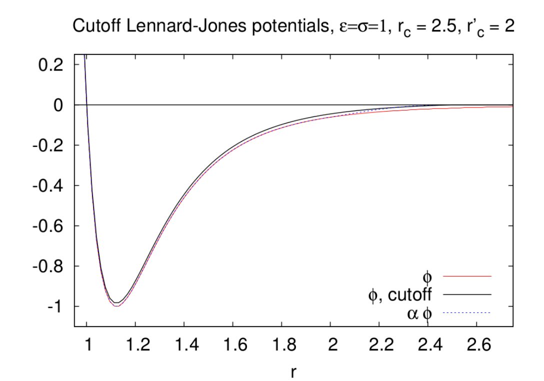
ε is its strength.

The potential is positive for small r: repulsion.

The potential is negative for large r: attraction.

The potential goes to zero for large r: short-range.

The potential has a minimum of $-\varepsilon$ at $2^{1/6}\sigma$.



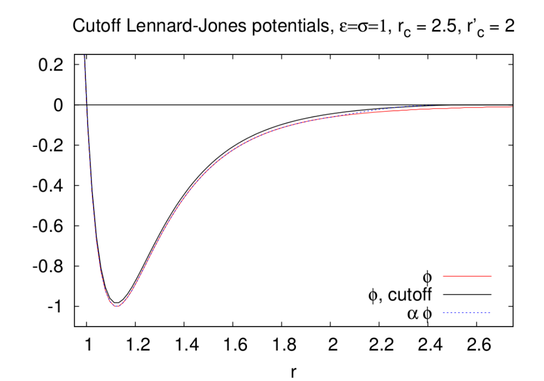
Force calculations

- Avoid infinite sums: modify the potential such that it becomes zero beyond a certain *cut-off* distance r_c:

$$\varphi'(r) = \begin{cases} \varphi(r) - \varphi(r_c) & \text{if } r < r_c \\ 0 & \text{if } r \geq r_c \end{cases}$$

  where the subtraction of $\varphi(r_c)$ prevents discontinuities.

- Computing all forces in an N-body system requires the computation of $N(N-1)/2$ forces $\mathbf{F}_{ij}$
- Force computation often the most demanding part of MD.



Streamlining the force evaluation

Cell divisions

- Divide the simulation box into cells larger than the cutoff r_c.
- Make a list of all particles in each cell.
- In the sum over pairs in the force computation, only sum pairs of particles in the same cell or in adjacent cells.

Neighbour lists

- Make a list of pairs of particles that are closer than $r_c + \delta r$.
- Sum over the list of pairs to compute the forces.
- The neighbour lists are to be used in subsequent force calculations as long as the list is still valid.
- Invalidation criterion: a particle has moved more than $\delta r/2$.

For systems with short-range interactions: $\mathcal{O}(N^2) \rightarrow \mathcal{O}(N)$.

Symplectic integrators

MD applications typically contain their own specialized integrator(s).

Reaching long times is paramount, so the stability of the integrator is the most important criteria.

So-called **symplectic integrators** turn out to be particularly stable.

These consists of substeps, each generated by its own Hamiltonian.

Verlet Scheme (first version)

$$r_{n+1} = r_n + \frac{p_n}{m}h + \frac{F_n}{2m}h^2$$

$$p_{n+1} = p_n + \frac{F_{n+1} + F_n}{2}h$$

The momentum rule appears to make this an implicit rule since F_{n+1} is required, but not if F only depends on r!

Verlet Scheme (second version)

The extra storage step can be avoided by introducing the half step momenta as intermediates:

$$p_{n+1/2} = p_n + \frac{1}{2}F_n h$$

$$r_{n+1} = r_n + \frac{p_{n+1/2}}{m}h$$

$$p_{n+1} = p_{n+1/2} + \frac{1}{2}F_{n+1}h$$

SciNet ADVANCED RESEARCH COMPUTING at the UNIVERSITY OF TORONTO

Where are the MD libraries?

MD packages are usually applications with a lot of parameters, that used other libraries. Examples:

- Gromacs
- NAMD
- LAMMPS

which all differ in intended usages, available force fields, serial speed (platform dependent), parallel scalability, etc.

OpenMM

Some MD packages come more as frameworks, which could be used as a library, within e.g. a C++ program.

OpenMM out of Stanford is a prime example which is actively maintained.

You can even setup the simulations from python with it.

https://simtk.org/home/openmm