odeint

Solving ordinary differential equations in C++

Karsten Ahnert[1,2] and Mario Mulansky[2]

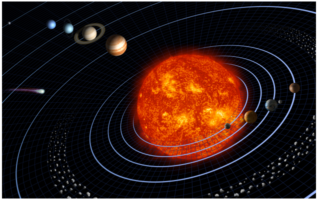
[1] Ambrosys GmbH, Potsdam
[2] Institut für Physik und Astronomie, Universität Potsdam

April 24, 2012

Outline

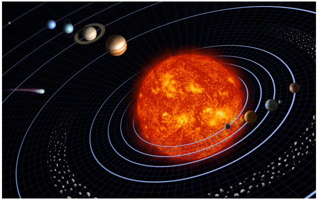
What is an ODE? – Examples

Newtons equations



What is an ODE? – Examples

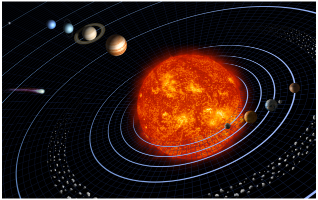
Newtons equations



Reaction and relaxation equations (i.e. blood alcohol content)

What is an ODE? – Examples

Newtons equations

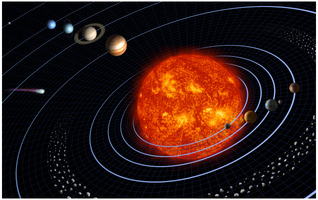


Granular systems

Reaction and relaxation equations (i.e. blood alcohol content)

What is an ODE? – Examples

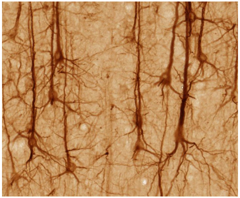
Newtons equations



Reaction and relaxation equations (i.e. blood alcohol content)

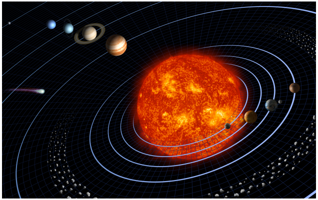
Granular systems

Interacting neurons



What is an ODE? – Examples

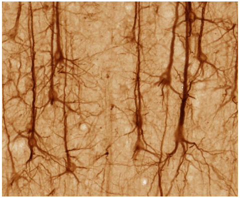
Newtons equations



Reaction and relaxation equations (i.e. blood alcohol content)

Granular systems

Interacting neurons



- Many examples in physics, biology, chemistry, social sciences
- Fundamental in mathematical modelling

What is an ODE?

$$\frac{\mathrm{d}x(t)}{\mathrm{d}t} = f\big(x(t), t\big) \qquad \text{short form} \qquad \dot{x} = f(x, t)$$

- $x(t)$ – dependent variable
- t – indenpendent variable (time)
- $f(x, t)$ – defines the ODE

Initial Value Problem (IVP):

$$\dot{x} = f(x, t), \qquad x(t = 0) = x_0$$

Numerical integration of ODEs

Find a numerical solution of an ODE an its initial value problem

$$\dot{x} = f(x,t)\ , \qquad x(t=0) = x_0$$

Example: Explicit Euler

$$x(t+\Delta t) = x(t) + \Delta t\ f(x(t),t) + \mathcal{O}(\Delta t^2)$$

General scheme of order s

$$x(t) \ \mapsto \ x(t+\Delta t) \qquad \text{, or}$$

$$x(t+\Delta t) = \mathcal{F}_t x(t) + \mathcal{O}(\Delta t^{s+1})$$

odeint

Solving ordinary differential equations in C++

Open source

- Boost license – do whatever you want do to with it

odeint

Solving ordinary differential equations in C++

Open source

- Boost license – do whatever you want do to with it

Download

- **`www.odeint.com`**

odeint

Solving ordinary differential equations in C++

Open source

- Boost license – do whatever you want do to with it

Download

- **`www.odeint.com`**

Modern C++

- Generic programming, functional programming
- Fast, easy-to-use and extendable.
- Container independent
- Portable

Who uses odeint

NetEvo

OMPL – Open Motion Planning Library

Motivation: The interface problem in C/C++

- Many frameworks exist to do numerical computations.
- Data has to be stored in containers or collections.
- GSL: `gsl_vector`, `gsl_matrix`
- NR: pointers with Fortran-style indexing
- Blitz++, MTL4, boost::ublas
- QT: `QVector`, wxWidgets: `wxArray`, MFC: `CArray`

But: All books on C++ recommend the use of the STL containers `std::vector`, `std::list`, ...

Motivation: The interface problem in C/C++

- Many frameworks exist to do numerical computations.
- Data has to be stored in containers or collections.
- GSL: `gsl_vector`, `gsl_matrix`
- NR: pointers with Fortran-style indexing
- Blitz++, MTL4, boost::ublas
- QT: `QVector`, wxWidgets: `wxArray`, MFC: `CArray`

But: All books on C++ recommend the use of the STL containers `std::vector`, `std::list`, ...

Theoretical solution of the interface mess

GoF Design Pattern: Adaptor, also known as Wrapper

Motivation: The interface problem in C/C++

- Many frameworks exist to do numerical computations.
- Data has to be stored in containers or collections.
- GSL: `gsl_vector`, `gsl_matrix`
- NR: pointers with Fortran-style indexing
- Blitz++, MTL4, boost::ublas
- QT: `QVector`, wxWidgets: `wxArray`, MFC: `CArray`

But: All books on C++ recommend the use of the STL containers `std::vector`, `std::list`, ...

Theoretical solution of the interface mess

GoF Design Pattern: Adaptor, also known as Wrapper

Alternative

Generic, container independent algorithms

Portability of your algorithm

How to run your algorithm?

- Single machine, single CPU
- Single machine, multiple CPU's (OpenMP, threads, ...)
- Multiple machines (MPI)
- GPU (Cuda, Thrust, OpenCL)

Portability of your algorithm

How to run your algorithm?

- Single machine, single CPU
- Single machine, multiple CPU's (OpenMP, threads, ...)
- Multiple machines (MPI)
- GPU (Cuda, Thrust, OpenCL)

Which data types are used by your algorithm?

- Build-in data types – `double`, `complex<double>`
- Arbitrary precision types – GMP, MPFR
- Vectorial data types `float2d`, `float3d`

Portability of your algorithm

How to run your algorithm?

- Single machine, single CPU
- Single machine, multiple CPU's (OpenMP, threads, ...)
- Multiple machines (MPI)
- GPU (Cuda, Thrust, OpenCL)

Which data types are used by your algorithm?

- Build-in data types – `double`, `complex<double>`
- Arbitrary precision types – GMP, MPFR
- Vectorial data types `float2d`, `float3d`

Theoretical solution

GoF Design Pattern: Strategy, also known as Policy

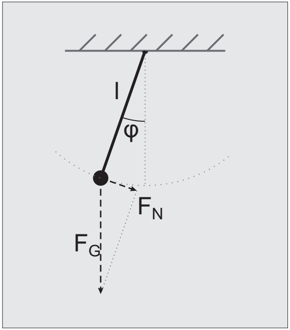
Alternative

Generic algorithms

Lets step into odeint

1. Introduction
2. Tutorial
3. Technical details
4. Discussion

Example – Pendulum



Newtons law: $ma = F$

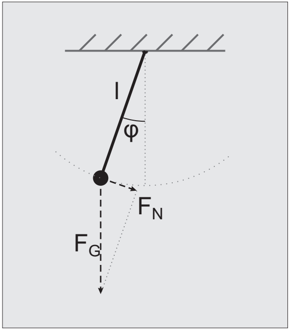
Acceleration: $a = l\ddot{\varphi}$

Force: $F = F_N = -mg\sin\varphi$

$\Longrightarrow$ **ODE for** φ

$$\ddot{\varphi} = -g/l\sin\varphi = -\omega_0^2\sin\varphi$$

Example – Pendulum



$$\ddot{\varphi} = -\omega_0^2 \sin\varphi$$

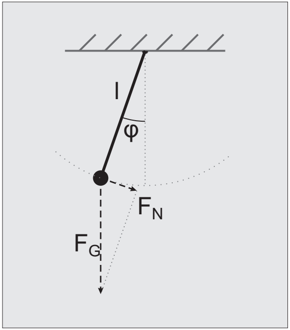
Small angle: $\sin\varphi \approx \varphi$

Harmonic oscillator $\ddot{\varphi} = -\omega_0^2 \varphi$

Analytic solution:
$\varphi = A\cos\omega_0 t + B\sin\omega_0 t$

Determine *A* and *B* from initial condition

Example – Pendulum



Full equation: $\ddot{\varphi} = -\omega_0^2 \sin \varphi$

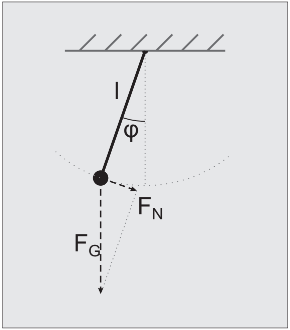
Pendulum with friction and external driving:

$$\ddot{\varphi} = -\omega_0^2 \sin \varphi - \mu \dot{\varphi} + \varepsilon \sin \omega_E t$$

No analytic solution is known

$\Longrightarrow$ **Solve this equation numerically.**

Example – Pendulum



$$\ddot{\varphi} = -\omega_0^2 \sin\varphi - \mu\dot{\varphi} + \varepsilon \sin\omega_E t$$

Create a first order ODE

$$x_1 = \varphi \ , \quad x_2 = \dot{\varphi}$$

$$\begin{aligned} \dot{x}_1 &= x_2 \\ \dot{x}_2 &= -\omega_0 \sin x_1 - \mu x_2 + \varepsilon \sin\omega_E t \end{aligned}$$

x_1 and x_2 are the state space variables

Let's solve the pendulum example numerically

```
#include <boost/numeric/odeint.hpp>

namespace odeint = boost::numeric::odeint;
```

$$\dot{x}_1 = x_2 \ , \quad \dot{x}_2 = -\omega_0 \sin x_1 - \mu x_2 + \varepsilon \sin \omega_E t$$

```
typedef std::array<double,2> state_type;
```

Let's solve the pendulum example numerically

$\dot{x}_1 = x_2$, $\dot{x}_2 = -\omega_0^2 \sin x_1 - \mu x_2 + \varepsilon \sin \omega_E t$

```
struct pendulum
{
  double m_mu, m_omega, m_eps;

  pendulum(double mu,double omega,double eps)
  : m_mu(mu),m_omega(omega),m_eps(eps) { }

  void operator()(const state_type &x,
     state_type &dxdt,double t) const
  {
    dxdt[0] = x[1];
    dxdt[1] = -sin(x[0]) - m_mu * x[1] +
        m_eps * sin(m_omega*t);
  }
};
```

Let's solve the pendulum example numerically

$\varphi(0) = 1\ , \quad \dot{\varphi}(0) = 0$

```
odeint::rk4< state_type > rk4;
pendulum p( 0.1 , 1.05 , 1.5 );

state_type x = {{ 1.0 , 0.0 }};
double t = 0.0;

const double dt = 0.01;
rk4.do_step( p , x , t , dt );
t += dt;
```

$x(0) \mapsto x(\Delta t)$

Let's solve the pendulum example numerically

```
std::cout<<t<<" "<< x[0]<<" "<<x[1]<<"\n";
for( size_t i=0 ; i<10 ; ++i )
{
  rk4.do_step( p , x , t , dt );
  t += dt;
  std::cout<<t<<" "<< x[0]<<" "<<x[1]<<"\n";
}
```

$x(0) \mapsto x(\Delta t) \mapsto x(2\Delta t) \mapsto x(3\Delta) \mapsto \ldots$

Simulation

Oscillator: $\mu = 0$, $\omega_E = 0$, $\varepsilon = 0$

Damped oscillator: $\mu = 0.1$, $\omega_E = 0$, $\varepsilon = 0$

Damped, driven oscillator: $\mu = 0.1$, $\omega_E = 1.05$, $\varepsilon = 1.5$

Different Steppers

```
runge_kutta_fehlberg78< state_type > s;
```

```
runge_kutta_dopri5< state_type > s;
```

Symplectic steppers (for Hamiltonian systems)

```
symplectic_rkn_sb3a_mclachlan< state_type > s;
```

Implicit steppers (for stiff systems)

```
rosenbrock4< double > s;
```

These steppers perform one step with constant step size!

Controlled steppers – Step size control

insert graphic

Controlled steppers

```
auto s = make_controlled(1.0e-6,1.0e6,
  runge_kutta_fehlberg78<state_type>() );
controlled_step_result r =
  s.try_step(ode,x,t,dt);
```

Tries to perform the step and updates x, t, and dt!

It works because Runge-Kutta-Fehlberg has error estimation:

```
runge_kutta_fehlberg78<state_type> s;
s.do_step(ode,x,t,dt,xerr);
```

Controlled steppers

```
auto s = make_controlled(1.0e-6,1.0e6,
  runge_kutta_fehlberg78<state_type>() );
while( t < t_end )
{
  controlled_step_result res
    = s.try_step(ode,x,t,dt);
  while( res != success )
  {
    res = s.try_step(ode,x,t,dt);
  }
}
```

Non-trivial time-stepping logic

Use integrate functions!

```
integrate_adaptive(s,ode,x,t_start,t_end,dt);
integrate_adaptive(s,ode,x,t_start,t_end,dt,
    observer);
```

Observer: Callable object `obs(x,t)`

Example (using Boost.Phoenix):

```
integrate_adaptive(s,ode,x,t_start,t_end,dt,
  cout<< arg1[0] << " " << arg1[1] << "\n" );
```

More integrate versions:

`integrate_const`, `integrate_times`, ...

```
integrate_const(s,ode,x,t,dt,obs);
```

Grafik with problem and solution

Dense output

```
auto s = make_dense_output( 1.0e-6 , 1.0e-6 ,
    runge_kutta_dopri5< state_type >() );
integrate_const( s , p , x , t , dt );
```

Interpolation between two steps with same precision as the original stepper!
Grafik!

More steppers

Stepper Concepts: Stepper, ErrorStepper, ControlledStepper, DenseOutputStepper

Stepper types:

- Implicit – `implicit_euler`, `rosenbrock4`
- Symplectic – `symplectic_rkn_sb3a_mclachlan`
- Predictor-Corrector – `adams_bashforth_moulton`
- Extrapolation – `bulirsch_stoer`
- Multistep methods – `adams_bashforth_moulton`

Some of them have step-size control and dense-output!

Small summary

- Very easy example – harmonic oscillator
- Basic features of odeint
- Different stepper – Controlled steppers, Dense output steppers
- Integrate functions

Small summary

- Very easy example – harmonic oscillator
- Basic features of odeint
- Different stepper – Controlled steppers, Dense output steppers
- Integrate functions

Now, lets look at the advanced features!

Extended systems

Lattice systems

Extended systems

Lattice systems

Discretiztations of PDEs

Extended systems

Lattice systems

Discretiztations of PDEs

Granular systems

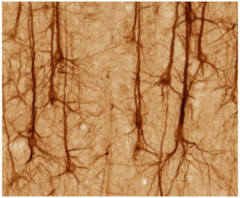
Extended systems

Lattice systems

Discretiztations of PDEs

Granular systems

ODEs on graphs



High-Performance-Computing

Phase oscillator lattices

Any oscillator can be described by one variable, its phase.
Trivial dynamics: $\dot{\varphi} = \omega\varphi$
Coupled phase oscillators
Neurosciences
Heart dynamics
Synchronization
Any weakly perturbed oscillator system
$\dot{\varphi}_k = \omega_k \varphi_k + q(\varphi_{k+1}, \varphi_k) + q(\varphi_k, \varphi_{k-1})$

Phase compacton lattice

$$\dot{\varphi}_k = \cos \varphi_{k+1} - \cos \varphi_{k-1}$$

State space contains *N* variables

```
typedef std::vector<double> state_type;
```

Animation

Space-time plot for visualization of compactons and chaos

Ensemble of phase oscillators

$$\dot{\varphi}_k = \omega_k + \sum_l \sin(\varphi_l - \varphi_k)$$

Synchronization – all oscillator oscillates with the same frequency

Synchronized state $\varphi_k = \omega_S t + \varphi_{0,k}$

Classical implementation

```
typedef std::vector<double> state_type;

struct phase_ensemble
{
    state_type m_omega;
    double m_epsilon;

    phase_ensemble(size_t n,double g=1.0,double
        epsilon=1.0)
    : m_omega(n,0.0),m_epsilon(epsilon)
    {
        create_frequencies(g);
    }

    void create_frequencies(double g) { ... }

    void operator()(const state_type &x,
        state_type &dxdt,double t) const
    {
        ...
    }
```

Solving ODEs with CUDA using thrust

What is Thrust

Thrust is a parallel algorithms library which resembles the C++ Standard Template Library (STL). Thrust's high-level interface greatly enhances developer productivity while enabling performance portability between GPUs and multicore CPUs. Interoperability with established technologies (such as CUDA, TBB and OpenMP) facilitates integration with existing software. Develop high-performance applications rapidly with Thrust!

Solving ODEs with CUDA using thrust

Where to use it

- Large systems, discretizations of ODE, lattice systems, granular systems, etc.
- Parameter studies, integrate many ODEs in parallel with different parameters
- Initial value studies, integrate the same ODE with many different initial conditions in parallel

Lorenz system – Parameter study

$$\dot{x} = \sigma(y - x) \qquad \dot{y} = Rx - y - xz \qquad \dot{z} = -bz + xy \tag{1}$$

Standard parameters $\sigma = 10$, $R = 28$, $b = 8/3$ deterministic chaos, butterfly effect
picture of Lorenz system

Lorenz system – Parameter study

Vary R from 0 to 50, for which parameters the system is chaotic?
Lyapunov exponents, perturbations of the original system

Algebras and operations

Euler method

$x_i(t + \Delta t) = x_i(t) + \Delta t * f_i(x)$

Algebras perform the iteration over i and operation the elementary addition.

Algebras and operations enter the stepper as template parameters

```
typedef runge_kutta4<state_type,value_type,
    deriv_type,time_type,
     algebra,operations,resize_policy> stepper;
```

- `default_operations`
- `range_algebra` – Boost.Ranges
- `vector_space_algebra` – Passes the state directly to the operations
- `fusion_algebra` – For compile time sequences, like $std :: tuple < double, double >$
- `thrust_algebra` and `thrust_algebra` – for thrust

Thrust example for Lorenz system,
Implementation of the system function

More advanced features, die themen kÃ¶nnen auch auf mehreren folien zusammengefasst werden

Boost::ref

boost::range

complex state types, vielleicht auch nicht arbitrary precision types

matrices as state types

graph as state types

self expanding lattices

Independent Algorithms

Goal

Container- and computation-independent implementation of the numerical algorithms.

Benefit

High flexibility and applicability, ODEINT can be used for virtually any formulation of an ODE.

Approach

Detatch the algorithm from memory management and computation detail and make each part interchangeable.

Mathematical Algorithm

Typical mathematical computation to calculate the solution of an ODE ($\dot{\vec{x}} = \vec{f}(\vec{x}, t)$):

$$
\begin{aligned}
\vec{F}_1 &= \vec{f}(\vec{x}_0, t_0) \\
\vec{x}' &= \vec{x}_0 + a_{21} \cdot \Delta t \cdot \vec{F}_1 \\
\vec{F}_2 &= \vec{f}(\vec{x}', t_0 + c_1 \cdot \Delta t) \\
\vec{x}' &= \vec{x}_0 + a_{31} \cdot \Delta t \cdot \vec{F}_1 + a_{32} \cdot \Delta t \cdot \vec{F}_2 \\
&\vdots \\
\vec{x}_1 &= \vec{x}_0 + b_1 \cdot \Delta t \cdot \vec{F}_1 + \cdots + b_s \cdot \Delta t \cdot \vec{F}_s
\end{aligned}
$$

Strucutural Requirements

$$\vec{F}_1 = \vec{f}(\vec{x}_0, t_0) \qquad \vec{x}' = \vec{x}_0 + a_{21} \cdot \Delta t \cdot \vec{F}_1$$

Types:

- vector type, mostly, but not neccessarily, some container like `vector<double>` (actually we have `state_type` and `deriv_type`)
- time type, usually `double`, but might be a multi-precision type
- value type, most likely the same as time type

Strucutural Requirements

$$\vec{F}_1 = \vec{f}(\vec{x}_0, t_0) \qquad \vec{x}' = \vec{x}_0 + a_{21} \cdot \Delta t \cdot \vec{F}_1$$

Types:

- vector type, mostly, but not neccessarily, some container like `vector<double>` (actually we have `state_type` and `deriv_type`)
- time type, usually `double`, but might be a multi-precision type
- value type, most likely the same as time type

Function Call:

```
void rhs( const vector_type &x , vector_type &
    dxdt , const time_type t )
{ /* user defined */ }

rhs( x0 , F1 , t ); //memory allocation for F1?
```

- Memory allocation for temporary results (`F`, `x'`)

Computational Requirements

$$\vec{x}_1 = \vec{x}_0 + b_1 \cdot \Delta t \cdot \vec{F}_1 + \cdots + b_s \cdot \Delta t \cdot \vec{F}_s$$

- vector-vector addition
- scalar-scalar multiplication
- scalar-vector multiplication

($\longrightarrow$ vector space)

Type Declarations

Tell ODEINT which types your are working with:

```
/* define your types */
typedef vector<double> state_type;
typedef vector<double> deriv_type;
typedef double value_type;
typedef double time_type;

/* define your stepper algorithm */
typedef runge_kutta4< state_type , value_type ,
    deriv_type , time_type > stepper_type;
```

Reasonable standard values for the template parameters allows for:

```
typedef runge_kutta4<state_type> stepper_type;
```

Memory Allocation / Resizing

Two possible situations: dynamic size / fixed size `vector_type`

dynamic size - memory allocation required

- e.g. `vector<double>`
- declare type as resizeable
- specialize resize template
- use `initially_resizer` or `always_resizer` in stepper algorithm

fixed size - memory allocation not required

- e.g. `array<double,N>`
- declare type as not resizeable
- that's it

Declare Resizeability

```
/* by default any type is not resizable */
template< class Container >
struct is_resizeable
{
    typedef boost::false_type type;
    const static bool value = type::value;
};

/* specialization for std::vector */
template< class T, class A >
struct is_resizeable< std::vector< T , A  > >
{
    typedef boost::true_type type;
    const static bool value = type::value;
};
```

To use a new dynamic sized type, this has to be specialized by the user.

Tell ODEINT how to resize

Again: only required if
`is_resizeable<state_type>::type == boost::true_type`.

Class Template responsible for resizing:

```
template< class StateOut , class StateIn >
struct resize_impl
{
    /* standard implementation */
    static void resize( StateOut &x1 , const
        StateIn &x2 )
    {
        x1.resize( boost::size( x2 ) );
    }
};
```

For anything that does not support `boost::size` or `resize` the user must provide a specialization.

1. Introduction
2. Tutorial
3. Technical details
4. Discussion