odeint
An advanced C++ framework for numerical integration of ordinary differential equations

Karsten Ahnert\textsuperscript{1,2} and Mario Mulansky\textsuperscript{2}

\textsuperscript{1} Ambrosys GmbH, Potsdam
\textsuperscript{2} Institut für Physik und Astronomie, Universität Potsdam

September 21, 2011
Outline

1. Programming issues in scientific computing
2. odeint introduction
3. odeint internals
4. Outlook and conclusion
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`,...
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

*Gamma, Holm, Johnson, Vlissides: Design Patterns, Elements of Reuseable Object-Oriented Software, 1998.*
Example

```
void CrankNicolsonEvolution::prepareVector(gsl_vector_complex* phi) {
    gsl_vector_complex* phi_temp = gsl_vector_complex_alloc(dim);
    // we need a copy of phi for this
    gsl_vector_complex_memcpy(phi_temp, phi);
    for (int i=1; i<dim-1; i++) {
        // phi_n = phi_n - i*dt/2 * (phi_n-1 + phi_n+1 + pot[n]*phi_n)
        gsl_vector_complex_set(phi, i, gsl_complex_add(
            gsl_vector_complex_get(phi_temp, i),
            gsl_complex_mul_imag(
                gsl_complex_add(
                    gsl_vector_complex_get(phi_temp, i-1),
                    gsl_vector_complex_get(phi_temp, i+1)),
                gsl_complex_mul_real(gsl_vector_complex_get(phi_temp, i),
                    potential[i]))),
            -dt/2.0));
    }
    if (periodic) {
        // periodic boundaries: i=0
        gsl_vector_complex_set(phi, 0, gsl_complex_add(
            gsl_vector_complex_get(phi_temp, 0),
            gsl_complex_mul_imag(
                gsl_complex_add(
                    gsl_vector_complex_get(phi_temp, dim-1),
                    gsl_vector_complex_get(phi_temp, 1)),
                gsl_complex_mul_real(gsl_vector_complex_get(phi_temp, 0),
                    potential[0])),
            -dt/2.0));
        // periodic boundaries: i=dim-1
        gsl_vector_complex_set(phi, dim-1, gsl_complex_add(
            gsl_vector_complex_get(phi_temp, dim-1),
            gsl_complex_mul_imag(
                gsl_complex_add(
                    gsl_vector_complex_get(phi_temp, dim-2),
                    gsl_vector_complex_get(phi_temp, 0)),
                gsl_complex_mul_real(gsl_vector_complex_get(phi_temp, dim-1),
                    potential[dim-1])),
            -dt/2.0));
    }
    // else {
```
Scalability 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)
How to run your algorithm?

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

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`
Scalability 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)

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
Gamma, Holm, Johnson, Vlissides: Design Patterns, Elements of Reuseable Object-Oriented Software, 1998.
Find a numerical solution of an ODE and its initial value problem

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

Example: Explicit Euler

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

General scheme of order $s$

$$x(t) \mapsto x(t + \Delta t), \text{ or } x(t + \Delta t) = \mathcal{F}_t x(t) + \mathcal{O}(\Delta t^{s+1})$$
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
  - https://github.com/headmyshoulder/odeint-v2
odeint

Solving ordinary differential equations in C++

Open source

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

Download

- https://github.com/headmyshoulder/odeint-v2

Modern C++

- Generic programming, functional programming
- Heavy use of the C++ template system
- Fast, easy-to-use and extendable.
- Container independent
- Scalable
# First example – Lorenz system

```cpp
#include <boost/numeric/odeint.hpp>
#include <tr1/array>

using namespace boost::numeric::odeint;

typedef std::tr1::array<double,3> state_type;

void lorenz(const state_type &x, state_type &dxdt, double t) {
    // ...
}

int main(int argc, char **argv) {
    state_type x = {{10.0, 10.0, 10.0}};
    typedef dense_output_runge_kutta<
        controlled_runge_kutta<
            runge_kutta_dopri5<state_type> > > stepper_type;
    integrate_const(stepper_type(), lorenz, x, 0.0, 10.0, 0.01);
    return 0;
}
```

- The r.h.s. of the ODE is a simple function
- Methods with dense-output and/or step-size control
- Integrate functions
- Templates
typedef std::vector<double> state_type;

struct fpu {
    double m_beta;
    fpu(double beta) : m_beta(beta) { }
    void operator()(const state_type &q, state_type &dpdt) const {
        // ...
    }
};

void statistics_observer ( const state_type &x , double t ) {
    // write the statistics
}

int main(int argc, char **argv) {
    state_type q(256), p(256);
    // initialize q,p
    integrate_const(symplectic_rkn_sb3a_mclachlan<state_type>(), fpu(1.0),
    make_pair(q,p), 0.0, 10.0, 0.01, statistics_observer());
    return 0;
}

- Symplectic solver
- The ODE is now a functor, it can have parameters.
- Automatic memory managements
- Observer
Structure of odeint

Stepper Classes
- Stepper
- ErrorStepper
- ControlledStepper
- DenseOutputStepper

Integrate Functions
- integrate()
- integrate_adaptive()
- integrate_const()
- integrate_n_steps
- integrate_times

Utils
- state_wrapper
- resize()
- ...

Operations
- default_operations
- mkl_operations
- thrust_operations

Algebra
- range_algebra
- fusion_algebra
- thrust_algebra
- vector_space_algebra
- ...

Internals – Example Euler’s method

User provides

\[ y_i = f_i(x(t), t) \]

odeint provides

\[ x_i(t + \Delta t) = x_i(t) + \Delta t \cdot y_i \]

(In general vector operations like \[ z_i = a_1 x_{1,i} + a_2 x_{2,i} + \ldots \])

Instantiation

```cpp
euler<state_type, value_type, deriv_type, time_type, algebra, operations > stepper;
```
User provides

\[ y_i = f_i(x(t), t) \]

odeint provides

\[ x_i(t + \Delta t) = x_i(t) + \Delta t \cdot y_i \]

(In general vector operations like \( z_i = a_1 x_{1,i} + a_2 x_{2,i} + \ldots \))

Instantiation

```cpp
euler<state_type, value_type, deriv_type, 
    time_type, algebra, operations > stepper;
```

All elements for container independence and scalability are already included in this line!
Internals – Example Euler’s method

\[ y_i = f_i(x(t)) \]
\[ x_i(t + \Delta t) = x_i(t) + \Delta t \cdot y_i \]

euler<state_type, value_type, deriv_type, time_type, algebra, operations > stepper;

General goal: Separation
- of how an vector is iterated
- of how the basic computations are performed
from the stepper
Internals – Example Euler’s method

\[ y_i = f_i(x(t)) \]
\[ x_i(t + \Delta t) = x_i(t) + \Delta t \cdot y_i \]

```c
  euler<state_type, value_type, deriv_type,
       time_type, algebra, operations > stepper;
```

Data types

- `state_type` – the type of \( x \)
- `value_type` – the basic numeric type, e.g. double
- `deriv_type` – the type of \( y \)
- `time_type` – the type of \( t, \Delta t \)
\[ y_i = f_i(x(t)) \]
\[ x_i(t + \Delta t) = x_i(t) + \Delta t \cdot y_i \]

```cpp
euler<state_type, value_type, deriv_type,
    time_type, algebra, operations > stepper;
```

Algebra do the iteration

Algebra must be a class with public methods

- `for_each1(x, op)` – Performs \( op(x_i) \) for all \( i \)
- `for_each2(x1, x2, op)` – Performs \( op(x1_i, x2_i) \) for all \( i \)
- `...`
Internals – Example Euler’s method

\[ y_i = f_i(x(t)) \]
\[ x_i(t + \Delta t) = x_i(t) + \Delta t \cdot y_i \]

```cpp
euler<state_type, value_type, deriv_type, 
    time_type, algebra, operations > stepper;
```

Operations do the basic computation

Operations must be a class with the public classes (functors)

- `scale_sum1` – Calculates \( x = a1 \cdot y1 \)
- `scale_sum2` – Calculates \( x = a1 \cdot y1 + a2 \cdot y2 \)
- ...

\[ y_i = f_i(x(t)) \]
\[ x_i(t + \Delta t) = x_i(t) + \Delta t \cdot y_i \]

```cpp
euler<state_type, value_type, deriv_type,
    time_type, algebra, operations > stepper;
```

All together

```cpp
m_algebra.for_each3(xnew ,xold, y ,
    operations_type::scale_sum2<value_type,time_type>(1.0,dt));
```
Stepper concepts

Concepts

“... In generic programming, a concept is a description of supported operations on a type...”
Stepper concepts

Concepts

“... In generic programming, a concept is a description of supported operations on a type...”

odeint provides

- **Stepper concept**
  
  ```cpp
  stepper.do_step(sys, x, t, dt);
  ```

- **ErrorStepper concept**
  
  ```cpp
  stepper.do_step(sys, x, t, dt, xerr);
  ```

- **ControlledStepper concept**
  
  ```cpp
  stepper.try_step(sys, x, t, dt);
  ```

- **DenseOutputStepper concept**
  
  ```cpp
  stepper.do_step(sys);
  stepper.calc_state(t, x);
  ```
<table>
<thead>
<tr>
<th>Method</th>
<th>Class name</th>
<th>Concept</th>
</tr>
</thead>
<tbody>
<tr>
<td>Euler</td>
<td>euler</td>
<td>SD</td>
</tr>
<tr>
<td>Runge-Kutta 4</td>
<td>runge_kutta4</td>
<td>S</td>
</tr>
<tr>
<td>Runge-Kutta Cash-Karp</td>
<td>runge_kutta_cash_karp54</td>
<td>SE</td>
</tr>
<tr>
<td>Runge-Kutta Fehlberg</td>
<td>runge_kutta_runge_fehlberg78</td>
<td>SE</td>
</tr>
<tr>
<td>Runge-Kutta Dormand-Prince</td>
<td>runge_kutta_dopri5</td>
<td>SED</td>
</tr>
<tr>
<td>Runge-Kutta controller</td>
<td>controlled_runge_kutta</td>
<td>C</td>
</tr>
<tr>
<td>Runge-Kutta dense output</td>
<td>dense_output_runge_kutta</td>
<td>D</td>
</tr>
<tr>
<td>Symplectic Euler</td>
<td>symplectic_euler</td>
<td>S</td>
</tr>
<tr>
<td>Symplectic RKN</td>
<td>symplectic_rkn_sb3a_mclachlan</td>
<td>S</td>
</tr>
<tr>
<td>Rosenbrock 4</td>
<td>rosenbrock4</td>
<td>ECD</td>
</tr>
<tr>
<td>Implicit Euler</td>
<td>implicit_euler</td>
<td>S</td>
</tr>
<tr>
<td>Adams-Bashforth-Moulton</td>
<td>adams_bashforth_moulton</td>
<td>S</td>
</tr>
<tr>
<td>Bulirsch-Stoer</td>
<td>bulirsch_stoer</td>
<td>CD</td>
</tr>
</tbody>
</table>

S – fulfills stepper concept  
E – fulfills error stepper concept  
C – fulfills controlled stepper concept  
D – fulfills dense output stepper concept
Integrate functions

- integrate_const
- integrate_adaptive
- integrate_times
- integrate_n_steps

Perform many steps, use all features of the underlying method
Integrate functions

- integrate_const
- integrate_adaptive
- integrate_times
- integrate_n_steps

Perform many steps, use all features of the underlying method

An additional observer can be called

integrate_const(stepper, sys, x, t_start, t_end, dt, obs);
More internals

- Header-only, no linking → powerful compiler optimization
- Memory allocation is managed internally
- No virtual inheritance, no virtual functions are called
- Different container types are supported, for example
  - STL containers (vector, list, map, trl::array)
  - MTL4 matrix types, blitz++ arrays, Boost.Ublas matrix types
  - thrust::device_vector
  - Fancy types, like Boost.Units
  - ANY type you like
- Explicit Runge-Kutta-steppers are implemented with a new template-metaprogramming method
- Different operations and algebras are supported
  - MKL
  - Thrust
  - gsl
ODEs on GPUs

Graphical processing units (GPUs) are able to perform up to $10^6$ operations at once in parallel.

Frameworks
- CUDA from NVIDIA
- OpenCL
- Thrust a STL-like library for CUDA and OpenMP

Applications:
- Parameter studies
- Large systems, like ensembles or one- or two dimensional lattices
- Discretizations of PDEs

odeint supports CUDA, through Thrust
Example: Parameter study of the Lorenz system

typedef thrust::device_vector<double> state_type;
typedef runge_kutta4<state_type ,value_type ,state_type ,value_type ,
    thrust_algebra ,thrust_operations > stepper_type;

struct lorenz_system {

    lorenz_system(size_t N ,const state_type &beta)
        : m_N(N) , m_beta(beta) {}

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

    size_t m_N;
    const state_type &m_beta;
};

int main( int arc , char* argv[] )
{
    const size_t N = 1024;

    vector<value_type> beta_host(N);
    for( size_t i=0 ; i<N ; ++i )
    
        beta_host[i] = 56.0 + value_type( i ) * ( 56.0 ) / value_type( N - 1 );
    state_type beta = beta_host;
    state_type x( 3 * N , 10.0 );
    integrate_const( stepper_type() , lorenz(N,beta) , x , 0.0 , 10.0 , 0.01 );

    return 0;
}
odeint provides a fast, flexible and easy-to-use C++ library for numerical integration of ODEs.

Its container independence is a large advantage over existing libraries.

Scalable

Generic programming is the main programming technique.
Submission to the boost libraries

Dynamical system classes for easy implementation of interacting dynamical systems

More methods: implicit methods and multistep methods.

Implementation of the Taylor series method

taylor_fixed_order< 25 , 3 > taylor_type stepper;

stepper.do_step(
    fusion::make_vector
    (                 
        sigma * ( arg2 - arg1 ) ,
        R * arg1 - arg2 - arg1 * arg3 ,
        arg1 * arg2 - b * arg3
    ) , x , t , dt );
Resources

An article about the used techniques exists at

Download and documentation
http://headmyshoulder.github.com/odeint-v2/

Development
https://github.com/headmyshoulder/odeint-v2

Contributions and feedback
are highly welcome