The Taylor series method for ordinary differential equations

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

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

December, 8, 2011

Outline

Solving Ordinary differential equations numerically

Find a numerical solution of the initial value problem for an ODE

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

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

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

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

Numerical methods – Overview

Methods:

- Steppers: $x(t) \mapsto x(t + \Delta t)$
- Methods with embedded error estimation
- Adaptive step size control
- Dense output

Examples:

- Explicit Runge Kutta methods
- Implicit methods for stiff systems
- Symplectic methods for Hamiltonian systems
- Multistep methods
- **Taylor series method**

Software for ordinary differential equations

- GNU Scientific library – gsl, C
- Numerical recipes, C and C++
- **www.odeint.com**, C++
- odeint, Python
- apache.common.math, Java

Taylor series method

$$\dot{x} = f(x)$$

Taylor series of the solution

$$x(t + \Delta t) = x(t) + \Delta t \dot{x}(t) + \frac{\Delta t^2}{2!} \ddot{x}(t) + \frac{\Delta t^3}{3!} x^{(3)}(t) + \dots$$

- Auto Differentiation to calculate $\dot{x}(t), \ddot{x}(t), x^{(3)}(t), \dots$
- Applications: Problems with high accuracy
  astrophyical application, chaotic dynamical systems
- Arbitrary precision types
- Interval arithmetics

Software for the Taylor method

Most software packages use generators

- ATSMCC, ATOMFT – Fortran generator

  George F. Corliss and Y. F. Chang. ACM Trans. Math. Software, 8(2):114-144, 1982.

  Y. F. Chang and George F. Corliss, Comput. Math. Appl., 28:209-233, 1994

- Taylor – C Generator

  Ángel Jorba and Maorong Zou, Experiment. Math. Volume 14, Issue 1 (2005), 99-117.

- TIDES – arbitrary precision, Mathematica generator

  Rodríguez, M. et. al, TIDES: A free software based on the Taylor series method, 2011

Operator overloading

- Adol-C
- cppAD

Expression templates

- **Taylor**

Notation

Taylor series of the ODE

$$x(t+\Delta t) = x(t) + \Delta t\dot{x}(t) + \frac{\Delta t^2}{2!}\ddot{x}(t) + \frac{\Delta t^3}{3!}x^{(3)}(t) + \ldots$$

Introduce the reduced derivatives X_i

$$F_i = \frac{1}{i!}\Big(f\big(x(t)\big)\Big)^{(i)} \quad , \qquad X_i = \frac{1}{i!}x^{(i)}(t) \quad , \qquad X_{i+1} = \frac{1}{i+1}F_i$$

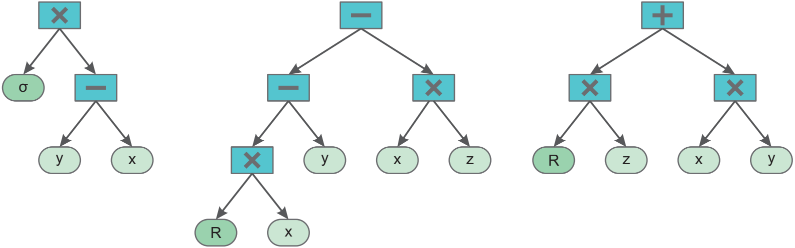
Taylor series

$$x(t+\Delta t) = X_0 + \Delta t X_1 + \Delta t^2 X_2 + \Delta t^3 X_3 + \ldots$$

Example: Lorenz system

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

Expression trees



The algorithm – Evaluation of the expression tree

Recursive determination of the Taylor coefficients

1. Initialize $X_0 = x(t)$
2. Calculate $X_1 = F_0(X_0)$
3. Calculate $X_2 = \frac{1}{2}F_1(X_0, X_1)$
4. Calculate $X_3 = \frac{1}{3}F_2(X_0, X_1, X_2)$

…

Calculate $X_s = \frac{1}{s}F_{s-1}(X_0, X_1, \ldots, X_{s-1})$

Finally $x(t + \Delta t) = X_0 + \Delta t X_1 + \Delta t^2 X_2 + \ldots$

The algorithm – Evaluation of the expression tree

Recursive determination of the Taylor coefficients

1. Initialize $X_0 = x(t)$
2. Calculate $X_1 = F_0(X_0)$
3. Calculate $X_2 = \frac{1}{2}F_1(X_0, X_1)$
4. Calculate $X_3 = \frac{1}{3}F_2(X_0, X_1, X_2)$

...

Calculate $X_s = \frac{1}{s}F_{s-1}(X_0, X_1, \ldots, X_{s-1})$

Finally $x(t + \Delta t) = X_0 + \Delta t X_1 + \Delta t^2 X_2 + \ldots$

In every iteration the expression tree is evaluated!

Algebraic operations

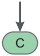
At every iteration the nodes in the expression tree have to be evaluated

Formulas for iteration i:

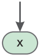
- Constants: $C_i = c\delta_{i,0}$
- Dependend variable x: X_i
- Summation $s = l + r$: $S_i = L_i + R_i$
- Multiplication $m = l \times r$: $M_i = \sum_{j=0}^{i} L_j R_{i-j}$
- Division $d = l/r$: $D_i = 1/R_0(L_i - \sum_{j=0}^{i-1} D_j R_{i-j})$
- Formulas for special functions exist: exp, log, cos, sin, $\ldots$

Algebraic operations – Formulas

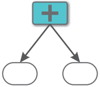
At every iteration the nodes in the expression tree have to be evaluated. Formulas for iteration i:



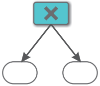
Constants: $C_i = c\delta_{i,0}$



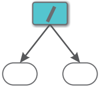
Dependend variable: x X_i



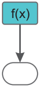
Summation $s = l + r$: $S_i = L_i + R_i$



Multiplication $m = l \times r$: $M_i = \sum_{j=0}^{i} L_j R_{i-j}$



Division $d = l/r$: $D_i = 1/R_0(L_i - \sum_{j=0}^{i-1} D_j R_{i-j})$



Formulas for special functions exist

Nice side effects

Step size control

- Error estimate: $err = X_s$
- $v = \left|\left|\frac{err_k}{\epsilon_{rel} + \epsilon_{abs}|x_k|}\right|\right|$
- $\Delta t = v^{-1/s}$
- No acceptance or rejection step is required

Dense output is trivially present

- $x(t+\tau) = X_0 + \tau X_1 + \tau^2 X_2 + \tau^3 X_3 + \ldots, \qquad 0 < \tau < \Delta t$

Methods for order estimation exist

Taylor

Download

- `https://github.com/headmyshoulder/taylor`

Taylor

Download

- `https://github.com/headmyshoulder/taylor`

Taylor will be integrated into **odeint**

- Implements some odeint - stepper

Taylor

Download

- `https://github.com/headmyshoulder/taylor`

Taylor will be integrated into **odeint**

- Implements some odeint - stepper

Modern C++

- Heavy use of the C++ template systems
- Expression templates for expression trees
- Template meta-programming

C++ Templates

- Here, C++ templates will be used to create the expression tree – **Expression Templates**
- Template Metaprogramming to evaluate the expression templates
- It basically means using the template engine to generate a program from which the compiler creates then the binary
- C++ compilers always use the template engine (no additional compile step required)

C++ Templates

- Here, C++ templates will be used to create the expression tree – **Expression Templates**
- Template Metaprogramming to evaluate the expression templates
- It basically means using the template engine to generate a program from which the compiler creates then the binary
- C++ compilers always use the template engine (no additional compile step required)

- Template engine and templates are a functional programming language
- Templates form a Turing-complete programming language
- $\longrightarrow$ You can solve any problem with the template engine

Expression templates

```
template< class L , class R > struct binary_expression
{
    binary_expression( string name , L l , R r )
    ...
    L m_l;
    R m_r;
};

struct terminal_expression
{
    terminal_expression( string name ) ...
};

const terminal_expression arg1( "arg1" ) , arg2( "arg2" );

template< class L , class R >
binary_expression< L , R > operator+( L l , R r )
{
    return binary_expression< L , R >( "Plus" , l , r );
}
...

template< class Expr > void print( Expr expr ) { ... }

print( arg1 );
print( arg1 + ( arg2 + arg1 - arg2 ) );
```

Expression templates

- Expression template are constructed during compile-time
- Strong optimization – no performance loss
- Lazyness
- Applications: Linear algebra systems, AD, (E)DSL

Example MTL4

```
mtl4::dense_matrix< double > m1( n , n ) , m2( n , n ) , m3( n , n ) ;

// do something useful with m1, m2, m3

mtl4::dense_matrix< double > result = m1 + 5.0 * m2 + 0.5 * m3 ;
```

Last line creates an expression template which is evaluated to

```
for( int i=0 ; i<n ; ++i )
   for( int j=0 ; j<n ; ++j )
      result( i , j ) = m1( i , j ) + 5.0 * m2( i , j ) + 0.5 * m3( i , j );
```

First example – Lorenz system

```
taylor_direct_fixed_order< 25 , 3 > stepper;
state_type x = {{ 10.0 , 10.0 , 10.0 }} ;
double t = 0.0;
double dt = 1.0;
while( t < 50000.0 )
{
    stepper.try_step(
        fusion::make_vector(
            sigma * ( arg2 - arg1 ) ,
            R * arg1 - arg2 - arg1 * arg3 ,
            arg1 * arg2 - b * arg3
            ) , x , t , dt );
    cout << t << "\t" << x << "\t" << dt << endl;
}
```

- ODE is a compile time sequence of expression templates
- The expression template is defined with `boost::proto`
- **No preprocessing step is necessary!**

Expression templates and ODEs

Boost.Proto (C++ library)

- Creation, manipulation and evaluation of the syntax tree
- Grammar = Allowed expression + (optional) Transformation

Boost.Proto (C++ library)

- Creation, manipulation and evaluation of the syntax tree
- Grammar = Allowed expression + (optional) Transformation

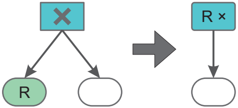
Taylor library uses Proto as front end:

- The ODE is a set of Proto expression templates
- ODE is transformed into a custom expression template

```
struct tree_generator :
    proto::or_
    <
        variable_generator< proto::_ > ,
        constant_generator ,
        plus_generator< tree_generator > ,
        minus_generator< tree_generator > ,
        multiplies_generator< tree_generator > ,
        divides_generator< tree_generator > ,
        unary_generator< tree_generator >
    > { };
```

Expression templates and ODEs

- Evaluation of the custom template – iteration several times of the template
- The nodes implement the rules for the algebraic expressions
- Optimzation of the expression tree



Example: The plus node

```
template< class Left , class Right >
struct plus_node : binary_node< Left , Right >
{
    plus_node( const Left &left , const Right &right )
    : binary_node< Left , Right >( left , right ) { }

    template< class State , class Derivs >
    Value operator()( const State &x , const Derivs &derivs , size_t which )
    {
        return m_left( x , derivs , which ) + m_right( x , derivs , which );
    }
};
```

The interface

Interface for easy implementation of arbitrary ODEs exist

```
taylor_direct_fixed_order< 25 , 3 > stepper;
stepper.try_step( sys , x , t , dt );
```

`sys` represents the ODE, Example:

```
fusion::make_vector(
            sigma * ( arg2 - arg1 ) ,
            R * arg1 - arg2 - arg1 * arg3 ,
            arg1 * arg2 - b * arg3 )
```

`x` is the state of the ODE is in-place transformed

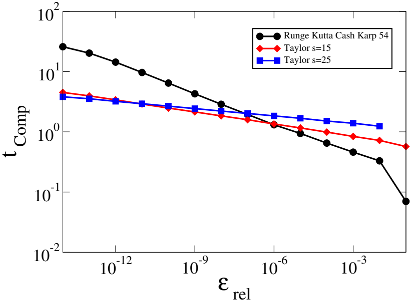
`t,dt` are the time and the step size

Results

Performance comparison against full Fortran code

- The Lorenz system as test system
- Benchmarking: a Fortran code with a non-AD implementation
- Both codes have the same performance, run-time deviation is less 20%
- Exact result depends strongly on the used compiler (gcc 4.5, gcc 4.6, gfortran, ...)

Comparison against other mehtods



- Taylor has good performance for high precision
- Outperforms the classical Runge-Kutta steppers

Conclusion

- Taylor – A C++ library for the Taylor series method of ordinary differential equations
- Uses expression templates as basis for the automatic differentiation
- Fast
- Uses modern C++ methods
- Template Metaprogramming is the main programming technique

Outlook

The library is not complete

- Implementation of special functions
- Implementation of stencils for lattice equations
- Implementation of variable order
- Implementation of dense output functionality
- Portability layer for arbitrary precision types
- Integration into odeint

Resources

Download and development

`https://www.github.com/headmyshoulder/taylor`

Odeint

`odeint.com`

Contributions and feedback

are highly welcome