Numerik Blatt 1

Kathrin Ronellenfitsch, Thorsten Beier, Christopher Pommrenke

Aufgabe 1

1.1

1.2

1.3

Aufgabe 2

2.1

- Wie lautet der Algorithmus zur Einbeziehung des globalen Fehlers, wenn statt einer Schrittweitenhalbierung eine Schrittweitenviertelung vorngenommen wird?

Es gilt analog zu Rannacher Script 2.3.1:

$$y_{n+1}^{H} - u(t_{n+1}) = \big(1 + O(H)\big)e_n - (H)^{m+1}\tau_{n+1}^{m} + O(H^{m+2}\big)$$

$$y_{n+1}^{H/4} - u(t_{n+1}) = \left(1 + O(H)\right)e_n - 4(\frac{1}{4}H)^{m+1}\tau_{n+1}^{m} + O(H^{m+2}\right)$$

Subtraktion ergibt:

$$y_{n+1}^{H} - y_{n+1}^{H/4} = O(H)e_n - \tau_{n+1}^{m}\left(4(\frac{1}{4}H)^{m+1}) - H^{m+1}\right) + O(H^{m+2}$$

daraus folgt (wenn man die O-Terme weglaesst):

$$\tau_{n+1}^{m} = \frac{y_{n+1}^{H} - y_{n+1}^{H/4}}{H^{m+1}(1 - 4^{-m})}$$

der Algorithmus zur adaptiven Schrittweitensteuerung ist nun analog zu Rannacher Script 2.3.2: (Schrittweitenkontrolle durch Schrittweitenhalbierung)

2.2

Schrittweitensteuerung durch Schrittweitenviertelung ist in Prinzip fuer jede Einschrittmehtode anwendbar. Fuer implizite Einschrittmethoden ist eine lokale Extrapolation noetig.

Aufgabe 3

Mit der Folgenden Implementierung ergibt sich kein nennenswerter vorteil durch die adaptive schrittweitenkontrolle (evt bug.) Der selbe maximale fehler ueber das intervall laesst sich mit auch ohne adaptive schrittweitenkontrolle erreichen(mit h=0.0001)

```cpp
#ifndef INITIAL_VALUE_PROBLEM_H_
#define INITIAL_VALUE_PROBLEM_H_

template<class T, class N=T>
class InitialValueProblem {
private:
  const N u0;
  const N t0;

public:
  /** \brief export size_type */
  typedef std::size_t size_type;

  /** \brief export time_type */
  typedef T time_type;

  /** \brief export number_type */
  typedef N number_type;

  //! constructor stores parameter lambda
  InitialValueProblem ( const N& u0_, const N& t0_)
    :  u0(u0_), t0(t0_)
  {}

  //! return number of componentes for the model
  std::size_t size () const
  {
    return 1;
  }

  //! set initial state including time value
  void initialize (T& t0, hdnum::Vector<N>& x0) const
  {
    t0 = this->t0;
    x0[0] = u0;
  }

  //! model evaluation
  void f (const T& t, const hdnum::Vector<N>& x, hdnum::Vector<N>& result
     ) const
  {
    result[0] = -200.0*t*x[0]*x[0];
  }

  //! jacobian evaluation needed for implicit solvers
  void f_x (const T& t, const hdnum::Vector<N>& x, hdnum::DenseMatrix<N>&
       result) const
  {
    throw std::string("Jacobian evaluation not implemented!");
  }
};

#endif /* INITIAL_VALUE_PROBLEM_H_ */
```

```cpp
#include <iostream>

#include "hdnum.hh"
#include "initial_value_problem.h"

int main() {
   typedef double Number; // define a number type

   const Number t0 = -3.0; // initial time
   const Number tStep0 = 0.0001; // delta t
   const Number tMax = -1.0; // end time
   const Number u0 = 1.0 / 901.0; // initial state
   const Number T= tMax-t0;

   typedef InitialValueProblem<Number> Model; // Model type
   Model model( u0, t0); // instantiate model
   typedef hdnum::Kutta3<Model> Solver; // solver
   Solver solver(model); // instantiate solver


   hdnum::Vector<Number> times; // store time values here
   hdnum::Vector<hdnum::Vector<Number> > states; // store states here
   times.push_back(solver.get_time()); // initial time
   states.push_back(solver.get_state()); // initial state
   Number h_n=tStep0;
   Number maxError=0;
   while (solver.get_time() <= tMax) // the time loop
   {
      std::cout<<solver.get_time()<<"\n";
      //do 2 steps with half stepsize
      solver.set_dt(h_n/2.0); // set initial time step
      solver.step(); // advance model by a half time step
      solver.step(); // advance model by a half time step
      Number yHalfStep=solver.get_state()[0];
      //
      solver.set_dt(h_n); // set normal time step
      solver.set_state(times.back(),states.back());
      solver.step(); // advance model by one time step
      times.push_back(solver.get_time()); // save time
      states.push_back(solver.get_state()); // and state

      Number yFullStep=solver.get_state()[0];
      Number absDiffHalfFullStep=std::fabs(yHalfStep-yFullStep);

      const size_t m=3; //kutta 3
      Number epsilon=std::pow(10.0,-10.0);
      Number TOL=epsilon*std::fabs(solver.get_state()[0])/h_n;
      Number TermA=std::pow(2.0*h_n,m+1)*(1.0-std::pow(2.0,-1.0*m));
      Number TermB=T*std::fabs(absDiffHalfFullStep);
      Number h_opt=((TermA*TOL)/TermB);
      h_opt=std::pow(h_opt, Number(1)/Number(m));

      //h_n=h_opt  is fine with  1/2 h_n <=h_opt
      //do the "real" step with the optimized h_n
      h_n=h_opt;
      solver.set_dt(h_n);
      solver.step(); // advance model by one time step
      times.push_back(solver.get_time()); // save time
      states.push_back(solver.get_state()); // and state
      const Number error=fabs(1.0/(1.0+100.0*std::pow(solver.get_time()
          ,2))-solver.get_state()[0]);
      if(error>maxError){
         maxError=error;
      }
   }

   std::cout<<"max error: "<<maxError<<"\n";
```

```
    solver.set_state(t0,states.front());
    solver.set_dt(tStep0);
    h_n=tStep0;
    while (solver.get_time() <= tMax) // the time loop
    {
       std::cout<<solver.get_time()<<"\n";
       solver.set_dt(h_n);
       solver.step();
       const Number error=fabs(1.0/(1.0+100.0*std::pow(solver.get_time()
           ,2))-solver.get_state()[0]);
       if(error>maxError){
         //std::cout<<" error h_old: "<<h_n<<"h_new"<<h_n/2<<"\n";
         h_n*=0.99;
         solver.set_dt(h_n);
         solver.set_state(t0,states.front());
       }
    }
    std::cout<<"fixed h_n for same error : "<<h_n<<"\n";

    return 0;
}
```