Documentation for the SDIRK C++ solver

by Erik Østergaard, IMM, DTU, Denmark

July 2, 1998

Abstract

This document serves as documentation for the SDIRK C++ ODE solver for the UNIX environment.

The solver was originally implemented using the Borland C++ compiler version 1.0 in 1991 by Michael Jeppesen. The author of this document has ported the source to UNIX together with minor changes in structure and features.

This document presents the stucture of the source and lists the calls and functions that are avaliable. Also a little example will be given.

The solver is located at

`http://www.gbar.dtu.dk/~c938790/app/`,

as `sdirk.tar.gz`, where both the source and this documentation can be found. The source is compiled and tested in the HP-UNIX environment as well as in the Red Hat Linux environment.

Keywords:
Sdirk ODE solver, UNIX, C++.

Contents

Chapter 1

Introduction

This chapter is ment to give a brief introduction to the SDIRK method and why it mattered porting the code to UNIX.

Also the changes made during porting are mentioned in order to enable updates of applications using the previous version of SDIRK.

1.1 Brief on the SDIRK methods

An often applied method to integrate differential equations is using one of the Runge-Kutta (RK) methods. These methods can be separated into three groups characterized by their coefficients, the Butcher Tableau.

In general, the RK method for the ODE problem

$$\dot{y} = f(t, y), \quad y(a) = c, \quad f : \mathbb{R} \times \mathbb{R}^m \to \mathbb{R}^m \tag{1.1}$$

is defined by (see e.g. [H66, p.34] or [Lambert, chapter 5])

$$y_{n+1} = y_n + h \sum_{i=1}^{s} b_i k_i \tag{1.2}$$

where

$$k_i = f\left(t_n + c_i h,\ y_n + h \sum_{i=1}^{s} a_i k_i \right),\ i = 1, ..., s \tag{1.3}$$

The coefficients a, b and c together form the Butcher Tableau

$$\begin{array}{c|cccc} c_1 & a_{11} & a_{12} & \cdots & a_{1s} \\ c_2 & a_{21} & a_{22} & \cdots & a_{2s} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ c_s & a_{s1} & a_{s2} & \cdots & a_{ss} \\ \hline & b_1 & b_2 & \cdots & b_s \end{array} = \begin{array}{c|c} c & A \\ \hline & b \end{array} \tag{1.4}$$

The family of RK methods can be categorized into three groups:
Explicit methods:

$$a_{ij} = 0 \text{ for } j \geq i, j = 1, ..., s \Leftrightarrow \text{A stricty lower triangular}$$

Semi-implicit methods:

$$a_{ij} = 0 \text{ for } j > i, j = 1, ..., s \Leftrightarrow \text{A lower triangular}$$

Implicit methods:

$$a_{ij} \neq 0 \text{ for some } j > i \Leftrightarrow \text{A not lower triangular}$$

The far most used RK method is the explicit one. It is called explicit since the calculations (1.2) and (1.3) can be performed explicitly. In contrast, the semi-implicit and implicit methods introduce implicit assignments in (1.2) and (1.3), thereby demanding iterative solutions.

Methods, that are semi-implicit, are called DIRK methods (Diagonally Implicit Runge Kutta). DIRK methods having identical diagonal elements are called SDIRK methods (Singly Diagonally Implicit Runge Kutta) [1]. Methods, that are implicit, are called SIRK methods (Singly Implicit Runge Kutta).

The SDIRK method used in the implementation is called the NT1 method after Nielsen and Thomsen, Institute of Mathematical Modelling, Technical University of Denmark. The method is characterized by the Butcher Tableau

$$\begin{array}{c|cccc} 0 & 0 & & & \\ 5/6 & 5/12 & 5/12 & & \\ 10/21 & 95/588 & -5/49 & 5/12 & \\ 1 & 59/600 & -31/75 & 539/600 & 5/12 \\ \hline & 59/600 & -31/75 & 539/600 & 5/12 \\ & -37/600 & -37/75 & 1813/6600 & 37/132 \end{array} = \begin{array}{c|c} c & A \\ \hline & b \\ & d \end{array} \tag{1.5}$$

which is supplied by a d vector allowing an error estimate D to be calculated in order to perform step length control. See chapter 2 about chossing the desired step length control. We notice, that element a_{11} is 0 and not 5/12 as the other diagonal elements. This is done in order to allow explicit assignment of the first elements in (1.2) and (1.3).

Using the formulae (1.2) and (1.3) implies use of some kind of iterative methode. Here is used the Newton-Raphson method to perform these iterations. Since the diagonal elements are identical (exept for a_{11}), the Jacobian used in the Newton-Raphson iteration only needs to be calculated if the step length h is changed.

[1] There is some confusion about this nomenclature. [Lambert] e.g. uses the term DIRK of Singly Diagonally Implicit methods

1.2 Why C++? Why UNIX?

There are several motivations for choosing C++ for SDIRK. First of all, the "hands-on" memory allocation and the thereby following handy vector and matrix definition/manipulation possibilities are very usefull in constructing complex implementations. C++ is an OOP language (object orienteted programming), thereby offering a high degree of structure with a vast amount of application possibilities.

UNIX is implemented in C, thereby substantiating the choice of language. Furthermore, the far most used operating system (OS) in the campus areas is UNIX (in any version), and applications supporting this OS are in great demand.

1.3 Changes made during porting

Only minor changes have been made during porting the source from MS-DOS to UNIX. The overall structure has been left untouched as well as the classes are the same.

A new class `IVector` has been added and the file `sdirk.h` containing both headers for classes `Sdirk` and `SdirkNewtonRaphson` has been split into two headers `sdirk.h` respectively `sdirknewt.h` containing the classes `Sdirk` respectively `SdirkNewtonRaphson`.

The main integration call found in the class `Sdirk` har been changed to allow restart of integration without loss of obtained step length corrections. The previous call

```
Integrate(double &t_lo, double &t_hi, long Nstep, DVector &y);
```

has been replaced by the call

```
Integrate(DVector &t, double &h, DVector &y, long &Nstep);
```

which is a change, that affects implementations using the late version of the Sdirk class.

1.4 Contents of the `sdirk` directory

In the root directory named `sdirk`, you'll find:

1. File `Makefile` - contains commands to create `lib/libsdirk.a`. See section 2.3.1 for further details.
2. Sub-directory `doc` - contains this document.
3. Sub-directory `source` - contains all SDIRK source (`.cc`) files.
4. Sub-directory `include` - contains all SDIRK include (`.h`) files.
5. Sub-directory `example` - contains source and `Makefile` for a SDIRK example program. See sections 2.3.2 and 4 for further details.

Chapter 2

Usage of the SDIRK solver

In this chapter the usage of the implementation is explained. The supported methods will be explained, and linking and compiling in the UNIX environment is shown.

2.1 Class overview

The `Sdirk` class is defined in `sdirk.h` and coded in `sdirk.cc`. The class provides access to integrating an ODE system given the specific system $\dot{y} = f(t, y)$, its Jacobian $J(t, y)$ in any form (exact or approximated) and initial conditions $y(a)$. The system can be integrated through an interval $[a, b]$ or single stepwise using the internal calculated steplength as next step. The tolerance can be set externally, and finally, statistics can be shown listing the performance for the most resent integration.

Access to the class is granted by including `sdirk.h` in the source file

```
#include "sdirk.h"
```

An instance of the class is made by the line

```
Sdirk *MyInstance
```

and it is initialized by the line

```
MyInstance = new Sdirk(1e-3, 2, &fun, &jac, SC_PI)
```

Here we introduce the use of the `Sdirk` constructor, which in general has the following syntax

```
Sdirk(double accur,
      int num_ode,
      void(* fun)(double t, DVector &y, DVector &f),
      void(* jac)(double t, DVector &y, DMatrix &j),
      StepControlType ctrl);
```

The constructor takes five arguments, which are

`accur`	The desired accuracy
`num_ode`	Number of ODE's (the size of the system)
`fun`	Pointer to a function implementing the system f
`jac`	Pointer to a function implementing the Jacobian J
`ctrl`	Parameter indicating the type of step control to be performed

Accuracy is recommend to be between 10^{-12} and 10^{-3} hereby normally ensuring acceptable results. The lower limit allowed for accuracy is 100 times the unit roundoff ϵ_M (usually $\epsilon_M = 2.2204 \cdot 10^{-16}$), since the Newton iteration uses one 100th of the tolerance as stop criterion. If the unit roundoff is not specified on your system, it is calculated automatically using Malcolms Methode.

`sdirk` provides four different step controls, which is chosen by inserting the right parameter for `ctrl`. The step controls are:

`SC_PRIM`	Primary step control for initial step length
`SC_WATTS`	Watts step control for initial step length
`SC_ORDINARE`	The ordinary step control
`SC_PI`	Proportional-Integral step control

2.2 Avaliable methods/calls

Apart from the `Sdirk` constructor, the following methods are avaliable from the `Sdirk` class:

- `~Sdirk();`
  Destructor. This can be called explicitly in order to free the memory allocated to the instance by the constructor. Normally this memory is freed anyway, but use of the destructor ensures correct memory handling.

- `Integrate(double &t, double &h, DVector &y);`
  Integrate a single step from $t =$`t` to $t =$`t+h`. The input `y` is the state $y(t)$ of the system at $t =$`t`. The output `y` is the state $y(t + h)$ of the system at $t =$`t+h`. If the system forces/allows the solver to change the steplength, `h` reflects the new steplength on output.

- `Integrate(DVector &T, double &h, DVector &y, long Nstep);`
  Integrate through the interval [`T[1]`, `T[2]`], with initial step length `h`. On return, `Nstep` contains the number of steps taken. Actually, this integration just calls the above mentioned integration till the end of the interval is reached. `h` holds the most recent used steplength on return.

- `Reset();`
  Resets the integrator, allowing a new integration to be performed without having to destruct the instance.

- `SetEps(double eps);`
  Sets a new value for the tolerance ϵ.

- `GetInfo(SdirkInfoType &p);`
  On return, `p` contains statistical information on the integration:

`p.NumOfGoodStep`	Number of accepted steps
`p.NumOfBadStep`	Number of rejected steps
`p.NumOfNewtonDivergens`	Number of divergent steps in the Newton iteration
`p.MaxError`	The maximal estimated error

- `ShowInfo();`
  Calls `GetInfo` and displays the information in a nice way.

All these methods are accessed the usual way. Setting the tolerance to $\epsilon = 10^{-6}$ is done by

```
MyInstance->SetEps(1e-6);
```

and integration of a given system (assuming `fun` and `jac` are defined) from $t = 0$ to $t = 2\pi$, printing the solution and statistics and finally terminating the instance is done by

```
DVector T(2);
T[1] = 0.0;
T[2] = 2*M_PI;
double h = 0.1;
long Nstep = 0;
DVector y(2);
y[1] = 1.0;
y[2] = 0.0;
MyInstance->Integrate(T, h, Nstep, y);
cout << "Solution found after " << Nstep << " steps:\n" << y;
MyInstance->ShowInfo();
MyInstance->~Sdirk();
```

2.3 Linking and compiling

2.3.1 The Sdirk library

By typing `make` in the `sdirk` directory, the library `libsdirk.a` is compiled and linked and placed at `sdirk/lib/libsdirk.a`. The different object files (`.o`) compiled temporarily are removed afterwards. See the `Makefile` in the `sdirk` directory for details. A compiled version is already present, but also the the source and headers are avaliable allowing changes to be made.

2.3.2 Using the Sdirk library

Access to the class `sdirk` is granted by including the header in your application and linking to the library during compiling.

Including the header is done by

```
#include "sdirk.h"
```

Linking to the library `libsdirk.a` is done by setting the include options in the `Makefile` for the application. This can be done by constructing a `Makefile` looking like this:

```
# Makefile for implementation linking to libsdirk.a
#
# Source to make: MyApplication.cc
TARGET = MyApplication

# Libraries and headers
LIB = ./sdirk/lib
INC = ./sdirk/include
# Compiler and options
CC = g++
CCOPTS = -Wall -I$(INC) -c

# Rules
all : $(TARGET)

$(TARGET): $(TARGET).o
        $(CC) -L$(LIB) -o $(TARGET) $(TARGET).o -lsdirk -lm
$(TARGET).o: $(TARGET).cc
        $(CC) $(CCOPTS) $(TARGET).cc
```

Chapter 3

File structure

This chapter is devoted to give an overview of the structure and class hierachy of the implementation. All text in `typewriter` is code related, and is either reserved words in C (C++) or items from the implementation (classes, methods, calls, structures e.g.).
All mentioned include files (`.h`) are placed in the `include` dir.
All mentioned source files (`.cc`) are placed in the `source` dir.

3.1 Hierachy and include files

In order to give an overview of the structure and hierachy of the files used in the implementation, both the table 3.1 and the figure 3.1 is provided.

In the figure, arrows indicate that root items are included in the outpointed item. It is important to notice that `sdirk.h` and `sdirknewt.h` are included in each other. This is due to a close cooperation between the two, which will be described further in section 3.3.

In the table, `cc` indicates that the header in the corresponding row is included in the source file (`.cc`) of the corresponding column. An `h` indicates that the header in the row is included in the include file (`.h`) of the column.

3.2 Plain include files

- `types.h` A collection of the userdefined types and definitions used in the implementation. If needed, calculation of unit roundoff.
- `dmatrix.h` Definition of a `double` matrix class. Overloading the =, +=, -=, +, -, $(\cdot,\cdot)$ and << operators.
  Class name: `DMatrix`.
- `dvector.h` Definition of a `double` vector class. Overloading the =, +=, -=, +, -, $[\cdot]$ and << operators.
  Class name: `DVector`.

Figure 3.1: Graphical representation of the file hierachy

	divctrl	dmatrix	dvector	ivector	lufac	newtbase	rkbase	sdirk	sdirknewt	step	stepbase	types
divctrl.h in	cc									h		
dmatrix.h in		cc			h		h					
dvector.h in			cc		h		h				h	
ivector.h in				cc	h		h					
lufac.h in					cc	h						
newtbase.h in						cc			h			
rkbase.h in							cc	h				
sdirk.h in								cc	h			
sdirknewt.h in								h	cc			
step.h in								h		cc		
stepbase.h in										h	cc	
types.h in	h	h	h	h								cc
<iostream.h> in												h
<math.h> in												h
<stdlib.h> in												h

Table 3.1: Table of connections between files and headers.

- `ivector.h` Definition of an `int` vector class. Overloading the =, +=, -=, +, -, [·] and << operators.
  Class name: `IVector`.
- `lufac.h` Definition of a linear equation solver, using Doolittles methode.
  Class name: `LUfactorize`.
- `divctrl.h` Implementation of a divergens control used in the `Sdirk` class.
  Class name: `DivergensStepControl`.

3.3 Derived classes

Three base classes are used, from which derivation is done. These are

- `stepbase.h` Implementation of steplength control base.
  Class name: `StepControlBase`.
- `newtbase.h` Implementation of base for iterative solving the linear equations using Newton-Raphson iteration.
  Class name: `NewtonRaphsonBase`.
- `rkbase.h` Implementation of base and coefficients for the Runge-Kutta iteration.
  Class name: `RKbase`.

In `step.h`, four steplength controls are implemented; all being first or second derived classes from `StepControlBase`, shown here:

Base: `StepControlBase`
  Derived: `OrdStepControl : public StepControlBase`
  Derived: `PrimStepControl : public StepControlBase`
  Derived: `PiStepControl : public StepControlBase`
    2nd Derived: `WattsStepControl : public PiStepControl`

In the `public` part of `StepControlBase`, four `virtual` methods are defined, hence they are defined locally in the derived classes. Source for the four derived classes (as mentioned above) is in `step.cc`.

The toplevel class `Sdirk` (defined in `sdirk.h`) has a `friend`, namely the class `SdirkNewtonRaphson` (defined in `sdirknewt.h`). Hence they include each other in order to obtain this close connection:

Base: `RKbase`
  Derived: `SDirk : public RKbase`
Base: `NewtonRaphsonbase`
  Derived: `SDirkNewtonRaphson : public NewtonRaphsonbase`

Futrhermore, `Sdirk` contains an instance of `SDirkNewtonRaphson` in its `private` part, and `SdirkNewtonRaphson` contains an instance of `SDirk` in its `private` part. Finally, the constructor for `SdirkNewtonRaphson` takes this `Sdirk` instance as an argument. This explains the doublepointing arrow at figure 3.1.

Chapter 4

Small example

In this section, a small example of using the SDIRK solver is given. This example can be found in the `sdirk/example` directory along with it's `Makefile`.

4.1 van der Pol's equation

We consider van der Pol's equation

$$\ddot{y} = \mu(1 - y^2)\dot{y} - y \tag{4.1}$$

which written as first order ODE's looks like

$$\dot{y} = f(t, y) = \begin{bmatrix} \dot{y}_1 \\ \dot{y}_2 \end{bmatrix} = \begin{bmatrix} y_2 \\ -y_1 + \mu(1 - y_1^2)y_2 \end{bmatrix}, \tag{4.2}$$

with initial conditions

$$\begin{bmatrix} y_1(0) \\ y_2(0) \end{bmatrix} = \begin{bmatrix} 2 \\ 0 \end{bmatrix}. \tag{4.3}$$

The Jacobian is found as

$$J(t, y) = \left[\frac{\partial(\dot{y}_1, \dot{y}_2)}{\partial(y_1, y_2)}\right] = \begin{bmatrix} 0 & 1 \\ -2\mu y_1 y_2 - 1 & \mu(1 - y_1^2) \end{bmatrix}. \tag{4.4}$$

We implement these functions like this:

```
// van der Pol
double mu = 20.0;
void fun(double t, DVector &y, DVector &f)
{
  f[1] =  y[2];
  f[2] = -y[1] + mu*(1.0 - y[1]*y[1])*y[2];
}

// Jacobian
void jac(double t, DVector &y, DMatrix &J)
{
  J(1, 1) = 0.0;
  J(1, 2) = 1.0;
  J(2, 1) = -2.0*mu*y[1]*y[2] - 1.0;
  J(2, 2) = mu*(1.0-y[1]*y[1]);
}
```

where we have set the parameter $\mu = 20$.

We now wish to integrate this system. This is done by constructing the following little application, named `vanderPol.cc`:

```
/************************************************
 * Example of usage of the Sdirk integration class *
 * The ODE system is van der Pol's equation        *
 ************************************************/

#include "sdirk.h"   // Include the Sdirk integration class
#include <fstream.h>

// Prototypes for the ODE system
void fun(double, DVector &, DVector &);
void jac(double, DVector &, DMatrix &);

int main()
{
  // Vectors for the state (y) and for the integration interval (T)
  DVector y(2), T(2);

  // Initializing the Sdirk instance MyInstance
  Sdirk *MyInstance;
  MyInstance = new Sdirk(1e-9, 2, &fun, &jac, SC_PI);

  // Initial values
  y[1] = 1.0;                   // y_1(0)
  y[2] = 0.0;                   // y_2(0)
  double t_lo = 0.0, t_hi = 20; // Integration interval:
  int m = 1500;                 // Number of outputs
  double dt = (t_hi-t_lo)/m;    // Distance between outputs
  double h=0.1;                 // Initial steplength
  long Nstep=0;                 // Number of steps used in each integration
```

```
  ofstream out("out.dat");              // Output file
  out << y[1] << " " << y[2] << "\n";   // Write initial state
  for (int i=0; i<m; i++){
    T[1] = t_lo+i*dt;                   // Start time of integration step
    T[2] = T[1]+dt;                     // End time of integration step
    MyInstance->Integrate(T,h,y,Nstep); // Integration
    out << y[1] << " " << y[2] << "\n"; // Write current state
  }
  out.close();          // Close the output file
  MyInstance->~Sdirk(); // Free the instance
  return 0;
}
```

We see, that we integrate the system through the interval $[0, 20]$ with 1500 outputs in the interval. The application is compiled using the `Makefile` listed in section 2.3.2. The simulation is written to a file named `"out.dat"`, and the wellknown graph is seen at figure 4.1.

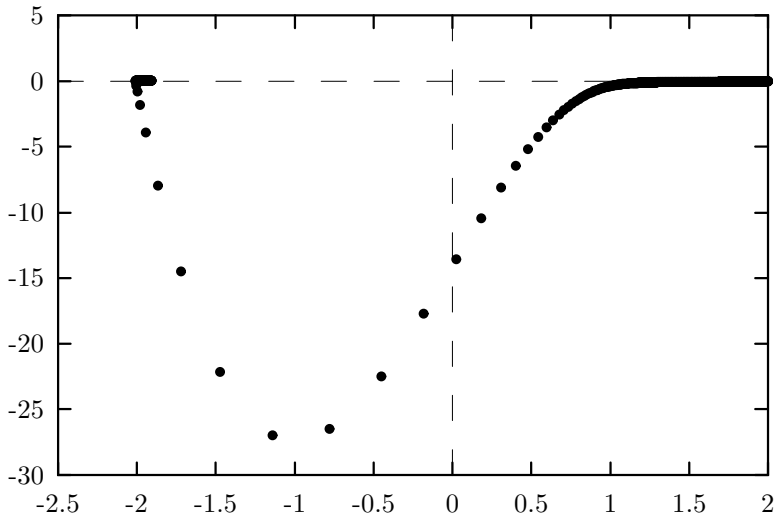

Figure 4.1: Plot of simulation

Bibliography

[Lambert] "Numerical Methods for Ordinary Differential Systems"
John Denholm Lambert,
John Wiley & Sons Ltd., 1991.

[H66] "Hæfte 66 - Numeriske Metoder for Sædvanlige differentialligninger"
Hans Bruun Nielsen & Per Grove Thomsen,
Numerisk Institut, DTH, 1993.

[Jeppesen] Source and headers for the SDIRK Borland C++ version 1.0 implementation
Michael Jeppesen,
Numerisk Institut, DTH, 1991.