Intel® Ordinary Differential Equation Solver Library

Reference Manual

May 2008

World Wide Web: http://developer.intel.com

Disclaimer and Legal Information

Contents

Introduction

The Intel® Ordinary Differential Equation Solver Library (Intel® ODE Solver Library) provides an application programming interface (API) for solving systems of Ordinary Differential Equations (ODE). The API includes universal and specialized ODE solver routines (see **ODE Routines**). These solvers are optimized for Intel® processors. Direct use of the *universal* ODE routine may be helpful to those who would like to find the method that best suits their problem. *Specialized* routines can be of interest to those who already know the property of their ODE systems and the proper method to solve them. All ODE routines can be called from C and Fortran, although the description of the input and output parameters uses Fortran conventions. C users can find routine calls specifics in the **Calling ODE Routines from C** section.

ODE Implemented

Intel® ODE Solver Library is intended for the numerical solution of initial value problems for a system of n ordinary differential equations with an arbitrary stiffness: *y* and *f* are vectors of dimension *n*, and

$$dy / dt = f(t,y),\ \ t>t_0,\ \ y(t_0) \text{ is a given initial vector.}$$

Depending on the estimation of the problem stiffness, it is possible to use an explicit or implicit solver, as well as the solver with automatic choice of the scheme, in every integration step. The explicit part of the solver, intended to solve non-stiff and middle-stiff problems, is based on the 4th order Merson's method and on the 1st order multistage methods with extended stability domains. The main feature of the explicit method is the stability control without additional computations, which enables using the method with an extended domain of stability at almost no cost. This feature makes the explicit method efficient for the middle-stiff problems as well. The explicit solver can be used as the 1st order method with the fixed number of stages from 2 through 9 or as the solver with automatic choice of the number of stages in every integration step. The technique is described in

> *Novikov E.A.* Application of explicit Runge-Kutta methods to solve stiff ODE's. Advances in Modeling & Analysis, A, AMSE Press, v.16, №1, 1992, p.23 -35.

The implicit part of the solver is based on the *L*-stable *(5,2)*-method of the 4th order of accuracy. This method uses 2 right-hand side computations, a computation (numerical or analytical) of the Jacobi matrix, a decomposition of the matrix into triangular factors, and 5 solutions of the linear system. The algorithm suggests an option to freeze the Jacobi matrix and the corresponding matrix decomposition for a few integration steps. This feature helps to improve performance for slowly varying Jacobi matrices. In the Intel® ODE Solver Library, the frozen Jacobi matrix can be used for not more than 20 consecutive successful steps. The class of *(m.k)*-methods is described in

> *Novikov E.A.* Construction of the (m,k)–methods for the solution of linear systems of ordinary differential equations. Mathematical models and tools for Chemical Kinetics, AMSE Transactions 'Scientific Siberian', Series A, v. 9, 1993, p.88 -103.

ODE Routines

Intel® MKL ODE Solver Library provides an interface to explicit, implicit, and mixed ODE solvers. In this manual, the interface is referred to as ODE interface. It implements a group of routines (ODE routines) used to compute the solution to stiff, non-stiff, and middle-stiff ODE systems. The ODE interface provides much flexibility of use: you can adjust routines to your particular needs at the cost of manual tuning routine parameters. To describe the Intel ODE interface, Fortran conventions are used. C users should refer to **Calling ODE Routines from C**.

NOTE. Please pay attention to the difference between Fortran and C when addressing arrays**:** **`ipar(k)`** in Fortran corresponds to **`ipar[k-1]`** in C.

The library contains a number of routines intended to solve non-stiff, middle-stiff, stiff problems, and problems with a variable or *a priori* unknown stiffness. There are 5 basic routines: a routine based on the explicit Runge-Kutta type method of a variable order and stages in the 1st order scheme, 2 routines based on the implicit one-step *L*-stable *(m.k)*-method of the 4th order with a numerical or user-defined Jacobi matrix, and 2 routines with automatic choice of the scheme and a numerical or user-defined Jacobi matrix for the implicit part. The library also contains a universal routine, which incorporates all the above methods in a single interface. This routine is mainly intended for experienced and research users.

Below is the list of ODE routines and brief description of their purpose.

`dodesol`	A universal routine for solving ODE systems with an arbitrary stiffness; incorporates the functionality of all the five routines described below.
`dodesol_rkm9st`	A specialized routine for solving non-stiff and middle-stiff ODE systems using the explicit method, which is based on the 4th order Merson's method and the 1st order multistage method of up to and including 9 stages with stability control.
`dodesol_mk52lfn`	A specialized routine for solving stiff ODE systems using the implicit method based on L-stable (5,2)-method with the numerical Jacobi matrix, which is computed by the routine.
`dodesol_mk52lfa`	A specialized routine for solving stiff ODE systems using the implicit method based on L-stable (5,2)-method with numerical or analytical computation of the Jacobi matrix. The user must provide a routine for this computation.
`dodesol_rkm9mkn`	A specialized routine for solving ODE systems with a variable or a priori unknown stiffness; automatically chooses the explicit or implicit scheme in every step and computes the numerical Jacobi matrix when necessary.
`dodesol_rkm9mka`	A specialized routine for solving ODE systems with a variable or a priori unknown stiffness; automatically chooses the explicit or implicit scheme in every step. The user must provide a routine for numerical or analytical computation of the Jacobi matrix.

dodesol

A universal routine for solving ODE systems with an arbitrary stiffness.

NOTE: Each routine described below can be invoked from this interface through proper choice of parameters. This routine is recommended for research users who have gained sufficient experience in using the Intel® ODE Solver Library.

Syntax

FORTRAN:

```
CALL dodesol(ipar,n,t,t_end,y,rhs,jacmat,h,hm,ep,tr,dpar,kd,ierr)
```

C:

```
dodesol(ipar,&n,&t,&t_end,y,rhs,jacmat,&h,&hm,&ep,&tr,dpar,kd,&ierr);
```

Input / Output Parameters

Parameter **Description**

Parameter	Description
ipar INTEGER	(input/output) Array of length 128 containing control flags and some statistics. **NOTE:** Avoid tuning the ***ipar*** array unless you have sufficient experience in using the Intel® ODE Solver Library.
ipar(1)	First entry flag: default initial value is ***ipar(1)****=0*. After the first successful step, ***ipar(1)=1.***
ipar(2)	Integration scheme flag: ***ipar(2)****=0* – the explicit or implicit scheme is chosen automatically, ***ipar(2)****=1* – the Merson's method and the 1st order explicit up to 9-stage methods are used, ***ipar(2)****=2* – the implicit *L*-stable *(5,2)*-method of the 4th order is used. Default value is 0.
ipar(3)	Exit flag: ***ipar(3)****=0* – exit at the end of the integration interval, ***ipar(3)****=1* – exit after every successful step. Default value is 0.
ipar(4)	Jacobi matrix computation flag: ***ipar(4)****=0* – the routine computes the numerical Jacobi matrix. In this case ***jacmat*** is a dummy parameter that will not be used. ***ipar(4)****=1* – a user-defined routine computing the Jacobi matrix must be provided in the ***jacmat*** parameter. Can be used to provide an analytical Jacobi matrix. Default value is 0.
ipar(5)	Jacobi matrix freezing flag: ***ipar (5)****=0* – freezing is not used. ***ipar(5)****=1* – freezing is used. Default value is 0.
ipar(6)	Fixed explicit method flag: ***ipar(6)****=0* – the explicit part works as the method with a variable order of accuracy and a variable number of stages in the 1st order method, ***ipar(6)****=1* – the explicit part works as the Merson's method, ***ipar(6)****=k, 1<k<10* – the explicit part works as fixed k-stage 1st order method. Default initial value is 0.
ipar(7)	Stability control flag: ***ipar (7)****=0* – the stability is under control, ***ipar(7)****=1* – if ***ipar(6)*** is not equal to 0, the stability is not controlled. Default value is 0.
ipar(8)	***ipar(8)****=k, 1<k<9* – the maximal number of stages in the 1st order k-stage method, ***ipar(8)****=0* – the maximal number of stages is 9. Default value is 0.
ipar(k)*, 8<k<35*	Parameters for internal use only. Initialize them with zeros prior to the first call to the routine.
ipar (k)*, 34<k<129*	Reserved for future use. Initialize these parameters with zeros prior to the first call to the routine.
n INTEGER	(input) Number of equations to be integrated.
t DOUBLE PRECISION	(input/output) Independent variable. By the end of integration, ***t*** is equal to ***t_end***.
t_end DOUBLE PRECISION	(input) The end of integration interval.
y DOUBLE PRECISION	(input/output) Array of dimension ***n*** containing the solution vector at a given moment ***t***. Before the integration, ***y*** must contain the user-defined initial data for the problem.

Parameter **Description**

rhs (input) Pointer to the user-defined routine that computes the right-hand side of the ODE system. This routine has the following template:

```
subroutine <name>(n, t, y, f)
integer n
double precision t, y(n), f(n)
 .................
f(i) = .....
 .................
return
end
```

where parameters `n, t,` and `y` are input and the array `f` is output; *`<name>`* must be declared as external in the calling program. C users may consult C example to create their own routine.

jacmat (input) Pointer to the user-defined routine that computes the Jacobi matrix (can be analytical) for the right-hand side of the ODE system. This routine has the following template:

```
subroutine <name>(n, t, y, a)
integer n
double precision t, y(n), a(n,n)
 .......................
a(i,j) = < df(i) / dy(j) >
 .......................
return
end
```

where parameters `n, t,` and `y` are input and the array `a` is output; *`<name>`* must be declared as external in the calling program. C users may consult C example to create their own routine.
NOTE: You may provide a dummy parameter instead of the routine if ***ipar(4)=0.***

Parameter	Description
h DOUBLE PRECISION	(input/output) Step size. During the integration, ***h*** contains the size of the last successful step.
hm DOUBLE PRECISION	(input) Minimal step size. If the step control requires ***h*** for the next step to be less than ***hm***, then ***h:=hm***. The value of ***hm*** depends on physical sizes involved in the problem. For the normalized problem statements, it is recommended to use ***hm****=1.d-12*.
ep DOUBLE PRECISION	(input) Relative error tolerance, which must be small enough. The code cannot ensure the requested accuracy for ***ep****<1.d-9*. This parameter is used to control the step size.
tr DOUBLE PRECISION	(input) Threshold for control of the relative error. If **|*y(i)*|>*tr***, then the relative error is controlled. Otherwise, the absolute error ***tr*ep*** is controlled.
dpar DOUBLE PRECISION	(output) Work array containing all intermediate stages of the methods. Allocate memory for ***dpar*** as double-precision array of length *max{13*n,(7+2*n)*n}* if ***ipar(2)=0,*** *13*n* if ***ipar(2)=1,*** *(7+2*n)*n* if ***ipar(2)=2.***
kd INTEGER	(output) Work array of length ***n***, which is used in the implicit scheme only.
ierr INTEGER	(output) Error flag.

NOTE: The routine `dodesol` is recommended for research purposes of experienced users of the library. If you are searching the best routine that suites your needs, you can try different routines by appropriate varying of the flags in array ***ipar***, which provides many specialized options. As a simpler alternative, it is recommended to use one or several routines described below, which can be applied to a specific problem.

dodesol_rkm9st

A specialized routine for solving non-stiff and middle-stiff ODE systems using the explicit method based on the 4th order Merson's method and the 1st order multistage method of up to and including 9 stages with stability control.

Syntax

FORTRAN:

```
CALL dodesol_rkm9st(ipar,n,t,t_end,y,rhs,h,hm,ep,tr,dpar,ierr)
```

C:

```
dodesol_rkm9st(ipar,&n,&t,&t_end,y,rhs,&h,&hm,&ep,&tr,dpar,&ierr);
```

Input / Output Parameters

Parameter	**Description**
ipar INTEGER	(input/output) Array of length 128 containing control flags and some statistics. **NOTE:** Avoid tuning the ***ipar*** array unless you have sufficient experience in using the Intel® ODE Solver Library.
ipar(1)	First entry flag: default initial value is ***ipar(1)****=0*. After the first successful step, ***ipar(1)*=1.**
ipar(2)	For internal use only.
ipar(3)	Exit flag: ***ipar(3)*=0** – exit at the end of the integration interval, ***ipar(3)*=1** – exit after every successful step. Default value is 0.
ipar(4), ipar(5)	For internal use only.
ipar(6)	Fixed explicit method flag: ***ipar(6)*=0** – the explicit part works as the method with a variable order of accuracy and a variable number of stages in the 1st order method, ***ipar(6)*=1** – the explicit part works as the Merson's method, ***ipar(6)*=k, 1<k<10** – the explicit part works as fixed the k-stage 1st order method. Default initial value is 0.
ipar(7)	Stability control flag: ***ipar (7)*=0** – the stability is under control, ***ipar(7)*=1** – if ***ipar(6)*** is not equal to 0, the stability is not controlled. Default value is 0.
ipar(8)	***ipar(8)*=k, 1<k<9** – the maximal number of stages in the 1st order k-stage method, ***ipar(8)*=0** – the maximal number of stages is 9. Default value is 0.
ipar(k), 8<k<35	Parameters for internal use only. Initialize them with zeros prior to the first call to the routine.
ipar (k), 34<k<129	Reserved for future use. Initialize these parameters with zeros prior to the first call to the routine.

The other parameters are the same as in `dodesol`. The exceptions are ***jacmat*** and ***kd***, which are absent in `dodesol_rkm9st` (see **Syntax**), and the length of array ***dpar***, which is *13****n***.

NOTE. `dodesol_rkm9st` can be called from `dodesol` by setting ***ipar(2)=1.***

dodesol_mk52lfn

A specialized routine for solving stiff ODE systems using the implicit method based on L-stable (5,2)-method with automatic numerical computation of the Jacobi matrix.

Syntax

FORTRAN:

```
CALL dodesol_mk52lfn(ipar,n,t,t_end,y,rhs,h,hm,ep,tr,dpar,kd,ierr)
```

C:

```
Dodesol_mk52lfn(ipar,&n,&t,&t_end,y,rhs,&h,&hm,&ep,&tr,dpar,kd,&ierr);
```

Input / Output Parameters

Parameter	Description
ipar INTEGER	(input/output) Array of length 128 containing control flags and some statistics. **NOTE:** Avoid tuning the ***ipar*** array unless you have sufficient experience in using the Intel® ODE Solver Library.
ipar(1),ipar(2)	For internal use only.
ipar(3)	Exit flag: ***ipar(3)=0*** – exit at the end of the integration interval, ***ipar(3)=1*** – exit after every successful step. Default value is 0.
ipar(4)	For internal use only.
ipar(5)	Jacobi matrix freezing flag: ***ipar(5)=0*** – freezing is not used. ***ipar(5)=1*** – freezing is used. Default value is 0.
ipar(k), *5<k<35*	Parameters for internal use only. Initialize them with zeros prior to the first call to the routine.
ipar(k), *34<k<129*	Reserved for future use. Initialize these parameters with zeros prior to the first call to the routine.

The other parameters are the same as in `dodesol`. The exception is ***jacmat***, which is absent in `dodesol_mk52lfn` (see **Syntax**), and the length of array ***dpar***, which is *(7+2****n)*n***.

NOTE. `dodesol_mk52lfn` can be called from `dodesol` by setting ***ipar(2)=2*** and ***ipar(4)=0.***

dodesol_mk52lfa

A specialized routine for solving stiff ODE systems using the implicit method based on L-stable (5,2)-method with a user-defined routine for numerical or analytical computation of the Jacobi matrix.

Syntax

FORTRAN:

```
CALL dodesol_mk52lfa(ipar,n,t,t_end,y,rhs,jacmat,h,hm,ep,tr,dpar,kd,ierr)
```

C:

```
dodesol_mk52lfa (ipar,&n,&t,&t_end,y,rhs,jacmat,&h,&hm,&ep,&tr,dpar,kd,&ierr);
```

Input / Output Parameters

Parameter	**Description**
ipar INTEGER	(input/output) Array of length 128 containing control flags and some statistics. **NOTE:** Avoid tuning the ***ipar*** array unless you have sufficient experience in using the Intel® ODE Solver Library.
ipar(1),ipar(2)	For internal use only.
ipar(3)	Exit flag: ***ipar(3)=0*** – exit at the end of the integration interval, ***ipar(3)=1*** – exit after every successful step. Default value is 0.
ipar(4)	For internal use only.
ipar(5)	Jacobi matrix freezing flag: ***ipar (5)=0*** – freezing is not used. ***ipar(5)=1*** – freezing is used. Default value is 0.
ipar(k), *5<k<35*	Parameters for internal use only. Initialize them with zeros prior to the first call to the routine.
ipar (k), *34<k<129*	Reserved for future use. Initialize these parameters with zeros prior to the first call to the routine.

The other parameters are the same as in `dodesol` (see **Syntax**). The length of array ***dpar*** is *(7+2**n*)**n*.

NOTE. `dodesol_mk52lfa` can be called from `dodesol` by setting ***ipar(2)=2*** and ***ipar(4)=1.***

dodesol_rkm9mkn

A specialized routine for solving ODE systems with a variable or a priori unknown stiffness. Automatically chooses the explicit or implicit scheme and computes the numerical Jacobi matrix.

Syntax

FORTRAN:

```
CALL dodesol_rkm9mkn(ipar,n,t,t_end,y,rhs,h,hm,ep,tr,dpar,kd,ierr)
```

C:

```
dodesol_rkm9mkn (ipar,&n,&t,&t_end,y,rhs,&h,&hm,&ep,&tr,dpar,kd,&ierr);
```

Input / Output Parameters

Parameter **Description**

Parameter	**Description**
ipar INTEGER	(input/output) Array of length 128 containing control flags and some statistics.
	NOTE: Avoid tuning the ***ipar*** array unless you have sufficient experience in using the Intel® ODE Solver Library.
ipar(1)	First entry flag: default initial value is ***ipar****(1)=0*. After the first successful step**,** ***ipar****(1)=1.*
ipar(2)	For internal use only.
ipar(3)	Exit flag:
	ipar*(3)***=***0* – exit at the end of the integration interval,
	ipar*(3)***=***1* – exit after every successful step.
	Default value is 0.
ipar(4)	For internal use only.
ipar(5)	Jacobi matrix freezing flag:
	ipar *(5)***=***0* – freezing is not used.
	ipar*(5)***=***1* – freezing is used.
	Default value is 0.
ipar(6)	Fixed explicit method flag:
	ipar*(6)***=***0* – the explicit part works as the method with a variable order of accuracy and a variable number of stages in the 1st order method,
	ipar*(6)***=***1* – the explicit part works as the Merson's method,
	ipar*(6)***=***k, $1<k<10$* – the explicit part works as the fixed k-stage 1st order method.
	Default initial value is 0.
ipar(7)	Stability control flag:
	ipar *(7)***=***0* – the stability is under control,
	ipar*(7)***=***1* – if ***ipar(6)*** is not equal to 0, the stability is not controlled.
	Default value is 0.
ipar(8)	***ipar****(8)***=***k, $1<k<9$* – the maximal number of stages in the 1st order k-stage method,
	ipar*(8)***=***0* – the maximal number of stages is 9.
	Default value is 0.
ipar*(k)***, *$8<k<35$*	Parameters for internal use only. Initialize them with zeros prior to the first call to the routine.
ipar *(k)***, *$34<k<129$*	Reserved for future use. Initialize these parameters with zeros prior to the first call to the routine.

The other parameters are the same as in dodesol. The exception is ***jacmat***, which is absent in dodesol_rkm9mkn (see **Syntax**), and the length of array ***dpar*** , which is *max{13*****n****,(7+2*****n****)********n}*.

NOTE. dodesol_rkm9mka can be called from dodesol by setting ***ipar(2)=****0* and ***ipar(4)=****0.*

dodesol_rkm9mka

A specialized routine for solving ODE systems with a variable or a priori unknown stiffness. Automatically chooses the explicit or implicit scheme and accepts a user-defined routine for numerical or analytical computation of the Jacobi matrix.

Syntax

FORTRAN:

```
CALL dodesol_rkm9mka(ipar,n,t,t_end,y,rhs,jacmat,h,hm,ep,tr,dpar,kd,ierr)
```

C:

```
dodesol_rkm9mka(ipar,&n,&t,&t_end,y,rhs,jacmat,&h,&hm,&ep,&tr,dpar,kd,&ierr);
```

Input / Output Parameters

Parameter	**Description**
ipar INTEGER	(input/output) Array of length 128 containing control flags and some statistics. **NOTE:** Avoid tuning the ***ipar*** array unless you have sufficient experience in using the Intel® ODE Solver Library.
ipar(1)	First entry flag: default initial value is ***ipar(1)****=0*. After the first successful step, ***ipar(1)=1****.*
ipar(2)	Parameter for internal use only.
ipar(3)	Exit flag: ***ipar(3)****=0* – exit at the end of the integration interval, ***ipar(3)****=1* – exit after every successful step. Default value is 0.
ipar(4)	Parameter for internal use only.
ipar(5)	Jacobi matrix freezing flag: ***ipar(5)****=0* – freezing is not used. ***ipar(5)****=1* – freezing is used. Default value is 0.
ipar(6)	Fixed explicit method flag: ***ipar(6)****=0* – the explicit part works as the method with a variable order of accuracy and a variable number of stages in the 1st order method, ***ipar(6)****=1* – the explicit part works as the Merson's method, ***ipar(6)****=k, 1<k<10* – the explicit part works as the fixed k-stage 1st order method. Default initial value is 0.
ipar(7)	Stability control flag: ***ipar(7)****=0* – the stability is under control, ***ipar(7)****=1* – if ***ipar(6)*** is not equal to 0, the stability is not controlled. Default value is 0.
ipar(8)	***ipar(8)****=k, 1<k<9* – the maximal number of stages in the 1st order k-stage method, ***ipar(8)****=0* – the maximal number of stages is 9. Default value is 0.
ipar(k)*, 8<k<35*	Parameters for internal use only. Initialize them with zeros prior to the first call to the routine.
ipar(k)*, 34<k<129*	Reserved for future use. Initialize these parameters with zeros prior to the first call to the routine.

The other parameters are the same as in `dodesol` (see **Syntax**). The length of array ***dpar*** is *max{13****n****,(7+2****n****)****n****}*.

NOTE. `dodesol_rkm9mka` can be called from `dodesol` by setting ***ipar(2)****=0* and ***ipar(4)****=1.*

Return Values

ierr = *0* – The routine has completed calculations normally.
ierr = *-100* – Error occurred: the number of equations ***n*** is less than zero.

ierr = *-201* – Error occurred: the initial value of ***t*** is greater than the end of integration interval ***t_end***.
ierr = *-202* – Error occurred: the initial step size ***h*** or minimal step size ***hm*** is non-positive.
ierr = *-203* – Error occurred: the relative error tolerance ***ep*** is non-positive.
ierr = *-204* – Error occurred: the threshold for control of the relative error is non-positive.

Interfaces

FORTRAN:

```
SUBROUTINE dodesol(ipar,n,t,t_end,y,rhs,jacmat,h,hm,ep,tr,dpar,kd,ierr)
INTEGER ipar(128), n, kd, ierr
DOUBLE PRECISION t,t_end,y(*),h,hm,ep,tr,dpar(*)
EXTERNAL rhs, jacmat

SUBROUTINE dodesol_rkm9st(ipar,n,t,t_end,y,rhs,h,hm,ep,tr,dpar,ierr)
INTEGER ipar(128), n, ierr
DOUBLE PRECISION t,t_end,y(*),h,hm,ep,tr,dpar(*)
EXTERNAL rhs

SUBROUTINE dodesol_mk52lfn(ipar,n,t,t_end,y,rhs,h,hm,ep,tr,dpar,kd,ierr)
INTEGER ipar(128), n, kd, ierr
DOUBLE PRECISION t,t_end,y(*),h,hm,ep,tr,dpar(*)
EXTERNAL rhs

SUBROUTINE dodesol_mk52lfa(ipar,n,t,t_end,y,rhs,jacmat,h,hm,ep,tr,dpar,kd,ierr)
INTEGER ipar(128), n, kd, ierr
DOUBLE PRECISION t,t_end,y(*),h,hm,ep,tr,dpar(*)
EXTERNAL rhs, jacmat

SUBROUTINE dodesol_rkm9mkn(ipar,n,t,t_end,y,rhs,h,hm,ep,tr,dpar,kd,ierr)
INTEGER ipar(128), n, kd, ierr
DOUBLE PRECISION t,t_end,y(*),h,hm,ep,tr,dpar(*)
EXTERNAL rhs

SUBROUTINE dodesol_rkm9mka(ipar,n,t,t_end,y,rhs,jacmat,h,hm,ep,tr,dpar,kd,ierr)
INTEGER ipar(128), n, kd, ierr
DOUBLE PRECISION t,t_end,y(*),h,hm,ep,tr,dpar(*)
EXTERNAL rhs, jacmat
```

C:

```
void dodesol(int*ipar, int*n, double*t, double*t_end, double*y, void*rhs(int*n,
                   double*t, double*y,double*f),
                   void*jacmat(int*n, double*t, double*y, double**a), double*h,
                   double*hm, double*ep, double*tr, double*dpar, int*kd,
                   int*ierr)

void dodesol_rkm9st(int*ipar, int*n, double*t, double*t_end, double*y,
                   void*rhs(int*n, double*t, double*y,double*f),double*h,
                   double*hm, double*ep, double*tr, double*dpar,  int*ierr)

void dodesol_mk52lfn(int*ipar, int*n, double*t, double*t_end, double*y,
                   void*rhs(int*n, double*t, double*y,doble*f), double*h,
                   double*hm, double*ep, double*tr, double*dpar, int*kd,
                   int*ierr)

void dodesol_mk52lfa(int*ipar, int*n, double*t, double*t_end, double*y,
                   void*rhs(int*n, double*t, double*y,double*f),
                        void*jacmat(int*n, double*t, double*y, double**a), double*h,
                        double*hm, double*ep, double*tr, double*dpar, int*kd,
                        int*ierr)

void dodesol_rkm9mkn(int*ipar, int*n, double*t, double*t_end, double*y,
                        void*rhs(int*n, double*t, double*y,double*f), double*h,
                        double*hm, double*ep, double*tr, double*dpar, int*kd,
                        int*ierr)

void dodesol_rkm9mka(int*ipar, int*n, double*t, double*t_end, double*y,
                        void*rhs(int*n, double*t, double*y,double*f),
                        void*jacmat(int*n, double*t, double*y, double**a), double*h,
                        double*hm, double*ep, double*tr, double*dpar, int*kd,
                        int*ierr)
```

Calling ODE Routines from C

The calling interface for all Intel ODE routines is designed to be easily used in Fortran. However, you can invoke each ODE routine directly from C if you are familiar with the inter-language calling conventions of your platform. The inter-language calling conventions include, but are not limited to, the argument passing mechanisms for the language, the data type mappings from Fortran to C, and decoration of Fortran external names on the platform. To promote portability and relieve a user of dealing with specifics of the calling conventions, C header file **`intel_ode.h`** declares a set of macros and introduces type definitions intended to hide the inter-language calling conventions and provide an interface to the routines that looks more natural (although not fully yet) in C. One of the key differences between C and Fortran is the language argument-passing mechanism: C programs use pass-by-value semantics, and Fortran programs use pass-by-reference semantics. The Intel® ODE Solver Library retains pass-by-reference Fortran semantics for C calls.

NOTE. Please pay attention to the difference between Fortran and C when addressing arrays**:** ***`ipar`*****`(k)`** in Fortran corresponds to ***`ipar`*****`[k-1]`** in C.

Code Examples

Code presented in this section computes solutions of initial value problem for the system of two ODEs describing nonlinear oscillations in Van der Pol generator. The 1st order ODE system has the following form:

$$\begin{aligned} y_1' &= y_2 \\ y_2' &= \lambda[(1-y_1^2)y_2 - y_1] \end{aligned} \qquad 0 < t \le 160,$$

with initial values $y_1(0) = 2,\ \ y_2(0) = 0$, parameter λ defines the frequency of nonlinear oscillations. In the code below, $\lambda = 10^6$. For these values of the parameter and the length of integration interval, the Van der Pol equations provide an example of the problem with a variable stiffness. This example demonstrates the usage of all ODE routines. The Jacobi matrix for this system, which is used in some ODE routines, has the form:

$$J = \begin{pmatrix} 0 & 1 \\ -\lambda(1+2y_1y_2) & \lambda(1-y_1^2) \end{pmatrix}.$$

NOTE. Usually the Van der Pol model is represented as a 2nd order oscillatory equation:

$$y'' - \alpha(1-y^2)y' + \omega^2 y = 0\,.$$

For the system with periodic solution, it is almost impossible to find an approximate solution with a large accuracy after many periods. For practical needs, it is enough to have two significant digits in the solution. The problem under consideration describes approximately 100 periods. An accuracy about 5% is achieved by setting error tolerance ***ep=1.e-6*** for all solvers.

At ***t=t_end=160***, the solution with 3 significant digits is: ***y(1)=1.878*** ***y(2)=-0.7436;***

The following results were obtained in the examples (up to rounding errors):
solution from **dodesol_rkm9st** : ***y(1)=1.88041*** ***y(2)=-0.74150;***
solution from **dodesol_mk52lfn**: ***y(1)=1.87779*** ***y(2)=-0.74325;***
solution from **dodesol_mk52lfa**: ***y(1)=1.87779*** ***y(2)=-0.74325;***
solution from **dodesol_rkm9mkn**: ***y(1)=1.87788*** ***y(2)=-0.74320;***
solution from **dodesol_rkm9mka**: ***y(1)=1.87788*** ***y(2)=-0.74320.***

All computations were performed with default values in the ***ipar*** array.

Example F below implements the computations in Fortran, and Example C provides C code for the same computations.

Example F

```
!*******************************************************************************
!                               INTEL CONFIDENTIAL
!   Copyright(C) 2007-2008 Intel Corporation. All Rights Reserved.
!   The source code contained  or  described herein and all documents related to
!   the source code ("Material") are owned by Intel Corporation or its suppliers
!   or licensors.  Title to the  Material remains with  Intel Corporation or its
!   suppliers and licensors. The Material contains trade secrets and proprietary
!   and  confidential  information of  Intel or its suppliers and licensors. The
!   Material  is  protected  by  worldwide  copyright  and trade secret laws and
!   treaty  provisions. No part of the Material may be used, copied, reproduced,
!   modified, published, uploaded, posted, transmitted, distributed or disclosed
!   in any way without Intel's prior express written permission.
!   No license  under any  patent, copyright, trade secret or other intellectual
!   property right is granted to or conferred upon you by disclosure or delivery
!   of the Materials,  either expressly, by implication, inducement, estoppel or
!   otherwise.  Any  license  under  such  intellectual property  rights must be
!   express and approved by Intel in writing.
!
!*******************************************************************************
!  This example gives the solution of initial value problem for the Van der
!  Pol equation:
!
!        y"-1.d6*[(1-y*y)*y'+1.d6*y=0,  0<t<160,    y(0)=2,  y'(0)=0.
!
!*******************************************************************************

      PROGRAM ODE_EXAMPLE_F

      IMPLICIT NONE

      INTEGER n, ierr, i
! It is higly recommended to declare ipar array of size 128
! for compatibility with future versions of ODE solvers
      INTEGER kd(2), ipar(128)
      DOUBLE PRECISION t, t_end, h, hm, ep, tr
! As ODE system has size n=2, than the size of dpar array is equal to
! max{13*n,(7+2*n)*n}=max{26,22}=26. More details on dpar array can be
! found in the Manual
```

```
      DOUBLE PRECISION y(2), dpar(26)
      EXTERNAL rhs_v_d_p, jacmat_v_d_p
      REAL time_begin, time_end

! global parameter settings suitable for all 6 dodesol routines
! minimal step size for the methods
         hm=1.d-12
! relative tolerance. The code cannot guarantee the requested accuracy for ep<1.d-9
         ep=1.d-6
! absolute tolerance
         tr=1.d-3

c***************************** dodesol ********************************
! Please don't forget to initialize ipar array with zeros before the first call
! to dodesol routines
         DO i=1,128
            ipar(i)=0
         END DO

         t=0.d0
         h=1.d-7

! setting size of the system n, end of integration interval t_end, and initial
! value y at t=0
         CALL example_v_d_p(n,t_end,y)

         CALL CPU_TIME(time_begin)
! universal solver
         CALL dodesol(ipar,n,t,t_end,y,rhs_v_d_p,jacmat_v_d_p,
     &                h,hm,ep,tr,dpar,kd,ierr)

         CALL CPU_TIME(time_end)

         IF(ierr.ne.0) THEN
            PRINT*,'======================='
            PRINT*,'DODESOL FORTRAN example FAILED'
            PRINT*,'dodesol routine exited with error code',ierr
            STOP 1
         END IF

         PRINT*
         PRINT*, 'dodesol results'
         PRINT*
         PRINT*, 'ipar(2)=',ipar(2),'ipar(4)=',ipar(4)
         PRINT*, 't=',t
         PRINT*, 'Solution','  y1=',y(1),'  y2=',y(2)
         PRINT*, '-----------------------------------------------------'
         PRINT*, 'CPU time=',time_end-time_begin,' seconds'
         PRINT*, '====================================================='
         PRINT*
         IF(dabs(y(1)-1.878d0)+dabs(y(2)+0.7436d0).gt.1.d-2) THEN
            PRINT*,'Solution seems to be inaccurate. Probably, ',
     &                                              'example FAILED...'
            STOP 1
        END IF

c***************************** dodesol_rkm9st ****************************
! Please don't forget to initialize ipar array with zeros before the first call
! to dodesol routines
         DO i=1,128
            ipar(i)=0
         END DO

         t=0.d0
         h=1.d-7
```

```
! setting size of the system n, end of integration interval t_end, and initial
! value y at t=0
         CALL example_v_d_p(n,t_end,y)

         CALL CPU_TIME(time_begin)
! explicit solver
         CALL dodesol_rkm9st(ipar,n,t,t_end,y,rhs_v_d_p,h,hm,ep,tr,
     &                       dpar,ierr)

         CALL CPU_TIME(time_end)

         IF(ierr.ne.0) THEN
            PRINT*,'======================='
            PRINT*,'DODESOL FORTRAN example FAILED'
            PRINT*,'dodesol_rkm9st routine exited with error code',ierr
            STOP 1
         END IF

         PRINT*
         PRINT*, 'dodesol_rkm9st results'
         PRINT*
         PRINT*, 't=',t
         PRINT*, 'Solution','  y1=',y(1),'  y2=',y(2)
         PRINT*, '-----------------------------------------------------'
         PRINT*, 'CPU time=',time_end-time_begin,' seconds'
         PRINT*, '====================================================='
         PRINT*
         IF(dabs(y(1)-1.878d0)+dabs(y(2)+0.7436d0).gt.1.d-2) THEN
           PRINT*,'Solution seems to be inaccurate. Probably, ',
     &                                            'example FAILED...'
            STOP 1
        END IF

c****************************** dodesol_mk52lfn *******************************
! Please don't forget to initialize ipar array with zeros before the first call
! to dodesol routines
         DO i=1,128
            ipar(i)=0
         END DO

         t=0.d0
         h=1.d-7

! setting size of the system n, end of integration interval t_end, and initial
! value y at t=0
         CALL example_v_d_p(n,t_end,y)

         CALL CPU_TIME(time_begin)
! implicit solver with automatic numerical Jacobi matrix computations
         CALL dodesol_mk52lfn(ipar,n,t,t_end,y,rhs_v_d_p,h,hm,ep,tr,
     &                        dpar,kd,ierr)

         CALL CPU_TIME(time_end)

         IF(ierr.ne.0) THEN
            PRINT*,'======================='
            PRINT*,'DODESOL FORTRAN example FAILED'
            PRINT*,'dodesol_mk52lfn routine exited with error code',ierr
            STOP 1
         END IF

         PRINT*
         PRINT*, 'dodesol_mk52lfn results'
         PRINT*
         PRINT*, 't=',t
         PRINT*, 'Solution','  y1=',y(1),'  y2=',y(2)
         PRINT*, '------------------------------------------------------'
         PRINT*, 'CPU time=',time_end-time_begin,' seconds'
         PRINT*, '======================================================'
         PRINT*
         IF(dabs(y(1)-1.878d0)+dabs(y(2)+0.7436d0).gt.1.d-2) THEN
           PRINT*,'Solution seems to be inaccurate. Probably, ',
     &                                              'example FAILED...'
            STOP 1
       END IF

c****************************** dodesol_mk52lfa *******************************
! Please don't forget to initialize ipar array with zeros before the first call
! to dodesol routines
         DO i=1,128
            ipar(i)=0
         END DO

         t=0.d0
         h=1.d-7

! setting size of the system n, end of integration interval t_end, and initial
! value y at t=0
         CALL example_v_d_p(n,t_end,y)

         CALL CPU_TIME(time_begin)
! implicit solver with user-defined Jacobi matrix computations
         CALL dodesol_mk52lfa(ipar,n,t,t_end,y,rhs_v_d_p,jacmat_v_d_p,
     &                        h,hm,ep,tr,dpar,kd,ierr)

         CALL CPU_TIME(time_end)

         IF(ierr.ne.0) THEN
            PRINT*,'======================='
            PRINT*,'DODESOL FORTRAN example FAILED'
            PRINT*,'dodesol_mk52lfa routine exited with error code',ierr
            STOP 1
         END IF

         PRINT*
         PRINT*, 'dodesol_mk52lfa results'
         PRINT*
         PRINT*, 't=',t
         PRINT*, 'Solution','  y1=',y(1),'  y2=',y(2)
         PRINT*, '------------------------------------------------------'
         PRINT*, 'CPU time=',time_end-time_begin,' seconds'
         PRINT*, '======================================================'
         PRINT*
         IF(dabs(y(1)-1.878d0)+dabs(y(2)+0.7436d0).gt.1.d-2) THEN
           PRINT*,'Solution seems to be inaccurate. Probably, ',
     &                                              'example FAILED...'
            STOP 1
       END IF

c****************************** dodesol_rkm9mkn *******************************
! Please don't forget to initialize ipar array with zeros before the first call
! to dodesol routines
         DO i=1,128
            ipar(i)=0
         END DO

         t=0.d0
         h=1.d-7

! setting size of the system n, end of integration interval t_end, and initial
! value y at t=0
         CALL example_v_d_p(n,t_end,y)
         CALL CPU_TIME(time_begin)
! hybrid solver with automatic numerical Jacobi matrix computations
         CALL dodesol_rkm9mkn(ipar,n,t,t_end,y,rhs_v_d_p,h,hm,ep,tr,
     &                        dpar,kd,ierr)

         CALL CPU_TIME(time_end)

         IF(ierr.ne.0) THEN
            PRINT*,'======================='
            PRINT*,'DODESOL FORTRAN example FAILED'
            PRINT*,'dodesol_rkm9mkn routine exited with error code',ierr
            STOP 1
         END IF

         PRINT*
         PRINT*, 'dodesol_rmk9mkn results'
         PRINT*
         PRINT*, 't=',t
         PRINT*, 'Solution',' y1=',y(1),' y2=',y(2)
         PRINT*, '-----------------------------------------------------'
         PRINT*, 'CPU time=',time_end-time_begin,' seconds'
         PRINT*, '====================================================='
         PRINT*
         IF(dabs(y(1)-1.878d0)+dabs(y(2)+0.7436d0).gt.1.d-2) THEN
           PRINT*,'Solution seems to be inaccurate. Probably, ',
     &                                           'example FAILED...'
            STOP 1
        END IF

c****************************** dodesol_rkm9mka *******************************
! Please don't forget to initialize ipar array with zeros before the first call
! to dodesol routines
         DO i=1,128
            ipar(i)=0
         END DO

         t=0.d0
         h=1.d-7

! setting size of the system n, end of integration interval t_end, and initial
! value y at t=0
         CALL example_v_d_p(n,t_end,y)

         CALL CPU_TIME(time_begin)
! hybrid solver with user-defined Jacobi matrix computations
         CALL dodesol_rkm9mka(ipar,n,t,t_end,y,rhs_v_d_p,jacmat_v_d_p,
     &                        h,hm,ep,tr,dpar,kd,ierr)

         CALL CPU_TIME(time_end)

         IF(ierr.ne.0) THEN
            PRINT*,'======================='
            PRINT*,'DODESOL FORTRAN example FAILED'
            PRINT*,'dodesol_rkm9mka routine exited with error code',ierr
            STOP 1
         END IF

         PRINT*
         PRINT*, 'dodesol_rkm9mka results'
         PRINT*
         PRINT*, 't=',t
         PRINT*, 'Solution',' y1=',y(1),' y2=',y(2)
         PRINT*, '-----------------------------------------------------'
         PRINT*, 'CPU time=',time_end-time_begin,' seconds'
         PRINT*, '====================================================='
         PRINT*
          IF(dabs(y(1)-1.878d0)+dabs(y(2)+0.7436d0).gt.1.d-2) THEN
           PRINT*,'Solution seems to be inaccurate. Probably, ',
     &                                              'example FAILED...'
            STOP 1
        END IF
        PRINT*, '======================='
         PRINT*, 'DODESOL FORTRAN example successfully PASSED through',
     &                                   ' all steps of computations'
         PRINT*

      STOP 0
      END

c********************** Example for Van der Pol equations **************
      SUBROUTINE example_v_d_p(n,t_end,y)
! The routine initializes the size of the system n, the end of
! integration interval t_end, and inital data y at t=0.0
      IMPLICIT NONE

      INTEGER n
      DOUBLE PRECISION t_end,y(*)

         n=2
         t_end=160.d0

         y(1)=2.d0
         y(2)=0.d0

      RETURN
      END

c******************* Right hand side of Van der Pol equations ******************
      SUBROUTINE rhs_v_d_p(n,t,y,f)

      IMPLICIT NONE

      INTEGER n
      DOUBLE PRECISION t,y(*),f(*)

         f(1)=y(2)
         f(2)=1.d6*((1.d0-y(1)*y(1))*y(2)-y(1))

      RETURN
      END

c************* analytical Jacobi matrix for Van der Pol equations **************
      SUBROUTINE jacmat_v_d_p(n,t,y,a)

      IMPLICIT NONE

      INTEGER n
      DOUBLE PRECISION t,y(*),a(n,*)

         a(1,1)=0.d0
         a(1,2)=1.d0
         a(2,1)=-1.d6*(1.d0+2.d0*y(1)*y(2))
         a(2,2)= 1.d6*(1.d0-y(1)* y(1))

      RETURN
      END

C************************* End of Fortran code example ************************
```

Example C

```
/*******************************************************************************
!                               INTEL CONFIDENTIAL
!   Copyright(C) 2007-2008 Intel Corporation. All Rights Reserved.
!   The source code contained  or  described herein and all documents related to
!   the source code ("Material") are owned by Intel Corporation or its suppliers
!   or licensors.  Title to the  Material remains with  Intel Corporation or its
!   suppliers and licensors. The Material contains trade secrets and proprietary
!   and  confidential  information of  Intel or its suppliers and licensors. The
!   Material  is  protected  by  worldwide  copyright  and trade secret laws and
!   treaty  provisions. No part of the Material may be used, copied, reproduced,
!   modified, published, uploaded, posted, transmitted, distributed or disclosed
!   in any way without Intel's prior express written permission.
!   No license  under any  patent, copyright, trade secret or other intellectual
!   property right is granted to or conferred upon you by disclosure or delivery
!   of the Materials,  either expressly, by implication, inducement, estoppel or
!   otherwise.  Any  license  under  such  intellectual property  rights must be
!   express and approved by Intel in writing.
!
!******************************************************************************
!
!  This example gives the solution of initial value problem for the Van der
!  Pol equation:
!
!        y"-1.d6*[(1-y*y)*y'+1.d6*y=0,  0<t<160,    y(0)=2,  y'(0)=0.
!
!*******************************************************************************/

#include <stdio.h>
#include <time.h>
#include "math.h"
#include "intel_ode.h"

extern void example_v_d_p(int*,double*,double*);
extern void rhs_v_d_p(int*,double*,double*,double*);
extern void jacmat_v_d_p(int*,double*,double*,double*);

int main(void)
{

       int n, ierr, i;
/* It is higly recommended to declare ipar array of size 128
   for compatibility with future versions of ODE solvers */
       int kd[2], ipar[128];
       double t, t_end, h, hm, ep, tr;
/* As ODE system has size n=2, than the size of dpar array is equal to
   max{13*n,(7+2*n)*n}=max{26,22}=26. More details on dpar array can be
   found in the Manual */
       double y[2], dpar[26];
       clock_t time_begin,time_end;

/* global parameter settings suitable for all 6 dodesol routines */
       hm=1.e-12; /* minimal step size for the methods */
       ep=1.e-6;  /* relative tolerance. The code cannot guarantee
                    the requested accuracy for ep<1.d-9 */
       tr=1.e-3;  /* absolute tolerance */

/****************************** dodesol ********************************/

/* Please don't forget to initialize ipar array with zeros before the first
call to dodesol routines */
       for (i=0;i<128;i++) ipar[i]=0;

       t=0.e0;
       h=1.e-7;
```

```
/* setting size of the system n, end of integration interval t_end, and
initial value y at t=0 */
      example_v_d_p(&n,&t_end,y);

      time_begin=clock();
/* universal solver */
      dodesol(ipar,&n,&t,&t_end,y,rhs_v_d_p,jacmat_v_d_p,&h,&hm,&ep,&tr,dpar,kd,&ierr);
      time_end=clock();

      if(ierr!=0)
      {
            printf("\n=======================\n");
            printf("DODESOL C example FAILED\n");
            printf("dodesol routine exited with error code %4d\n",ierr);
            return -1;
      }

      printf("\ndodesol results\n\n");
      printf("ipar[1]=%4d, ipar[3]=%4d\n",ipar[1],ipar[3]);
      printf("t=%5.1f\n",t);
      printf("Solution      y1=%17.14f,   y2=%17.14f\n",y[0],y[1]);
      printf("--------------------------------------------------------\n");
      printf("CPU time=%f seconds\n", ((double)(time_end-time_begin))/CLOCKS_PER_SEC);
      printf("========================================================\n\n");
      if(fabs(y[0]-1.878e0)+fabs(y[1]+0.7436e0)>1.e-2)
            printf("Solution seems to be inaccurate. Probably example FAILED\n");

/************************** dodesol_rkm9st ***************************/

/* Please don't forget to initialize ipar array with zeros before the first
call to dodesol routines */
      for (i=0;i<128;i++) ipar[i]=0;

      t=0.e0;
      h=1.e-7;

/* setting size of the system n, end of integration interval t_end, and
initial value y at t=0 */
      example_v_d_p(&n,&t_end,y);

      time_begin=clock();
      /* explicit solver */
      dodesol_rkm9st(ipar,&n,&t,&t_end,y,rhs_v_d_p,&h,&hm,&ep,&tr,dpar,&ierr);
      time_end=clock();

      if(ierr!=0)
      {
            printf("\n=======================\n");
            printf("DODESOL C example FAILED\n");
            printf("dodesol_rkm9st routine exited with error code %4d\n",ierr);
            return -1;
      }

      printf("\ndodesol_rkm9st results\n\n");
      printf("t=%5.1f\n",t);
      printf("Solution      y1=%17.14f,   y2=%17.14f\n",y[0],y[1]);
      printf("--------------------------------------------------------\n");
      printf("CPU time=%f seconds\n", ((double)(time_end-time_begin))/CLOCKS_PER_SEC);
      printf("========================================================\n\n");
      if(fabs(y[0]-1.878e0)+fabs(y[1]+0.7436e0)>1.e-2)
      {
            printf("Solution seems to be inaccurate. Probably, example FAILED...\n");
            return -1;
      }
/************************** dodesol_mk52lfn ***************************/
```

```
/* Please don't forget to initialize ipar array with zeros before the first
call to dodesol routines */
      for (i=0;i<128;i++) ipar[i]=0;

      t=0.e0;
      h=1.e-7;

/* setting size of the system n, end of integration interval t_end, and
initial value y at t=0 */
      example_v_d_p(&n,&t_end,y);

      time_begin=clock();
/* implicit solver with automatic numerical Jacobi matrix computations */
      dodesol_mk52lfn(ipar,&n,&t,&t_end,y,rhs_v_d_p,&h,&hm,&ep,&tr,dpar,kd,&ierr);
      time_end=clock();

      if(ierr!=0)
      {
            printf("\n========================\n");
            printf("DODESOL C example FAILED\n");
            printf("dodesol_mk52lfn routine exited with error code %4d\n",ierr);
            return -1;
      }

      printf("\ndodesol_mk52lfn results\n\n");
      printf("t=%5.1f\n",t);
      printf("Solution      y1=%17.14f,   y2=%17.14f\n",y[0],y[1]);
      printf("--------------------------------------------------------\n");
      printf("CPU time=%f seconds\n", ((double)(time_end-time_begin))/CLOCKS_PER_SEC);
      printf("========================================================\n\n");
      if(fabs(y[0]-1.878e0)+fabs(y[1]+0.7436e0)>1.e-2)
      {
            printf("Solution seems to be inaccurate. Probably, example FAILED...\n");
            return -1;
      }
/************************** dodesol_mk52lfa ****************************/

/* Please don't forget to initialize ipar array with zeros before the first
call to dodesol routines */
      for (i=0;i<128;i++) ipar[i]=0;

      t=0.e0;
      h=1.e-7;

/* setting size of the system n, end of integration interval t_end, and
initial value y at t=0 */
      example_v_d_p(&n,&t_end,y);

      time_begin=clock();
/* implicit solver with user-defined Jacobi matrix computations */
      dodesol_mk52lfa(ipar,&n,&t,&t_end,y,rhs_v_d_p,jacmat_v_d_p,&h,&hm,&ep,&tr,dpar,kd,&
ierr);
      time_end=clock();

      if(ierr!=0)
      {
            printf("\n========================\n");
            printf("DODESOL C example FAILED\n");
            printf("dodesol_mk52lfa routine exited with error code %4d\n",ierr);
            return -1;
      }

      printf("\ndodesol_mk52lfa results\n\n");
      printf("t=%5.1f\n",t);
      printf("Solution      y1=%17.14f,   y2=%17.14f\n",y[0],y[1]);
      printf("-----------------------------------------------------\n");
      printf("CPU time=%f seconds\n", ((double)(time_end-time_begin))/CLOCKS_PER_SEC);
      printf("=====================================================\n\n");
      if(fabs(y[0]-1.878e0)+fabs(y[1]+0.7436e0)>1.e-2)
      {
            printf("Solution seems to be inaccurate. Probably, example FAILED...\n");
            return -1;
      }
/************************** dodesol_rkm9mkn ****************************/

/* Please don't forget to initialize ipar array with zeros before the first
call to dodesol routines */
      for (i=0;i<128;i++) ipar[i]=0;

      t=0.e0;
      h=1.e-7;

/* setting size of the system n, end of integration interval t_end, and
initial value y at t=0 */
      example_v_d_p(&n,&t_end,y);

      time_begin=clock();
/* hybrid solver with automatic numerical Jacobi matrix computations */
      dodesol_rkm9mkn(ipar,&n,&t,&t_end,y,rhs_v_d_p,&h,&hm,&ep,&tr,dpar,kd,&ierr);
      time_end=clock();

      if(ierr!=0)
      {
            printf("\n========================\n");
            printf("DODESOL C example FAILED\n");
            printf("dodesol_rkm9mkn routine exited with error code %4d\n",ierr);
            return -1;
      }

      printf("\ndodesol_rkm9mkn results\n\n");
      printf("t=%5.1f\n",t);
      printf("Solution      y1=%17.14f,   y2=%17.14f\n",y[0],y[1]);
      printf("-----------------------------------------------------\n");
      printf("CPU time=%f seconds\n", ((double)(time_end-time_begin))/CLOCKS_PER_SEC);
      printf("=====================================================\n\n");
      if(fabs(y[0]-1.878e0)+fabs(y[1]+0.7436e0)>1.e-2)
      {
            printf("Solution seems to be inaccurate. Probably, example FAILED...\n");
            return -1;
      }
/************************** dodesol_rkm9mka ****************************/

/* Please don't forget to initialize ipar array with zeros before the first
call to dodesol routines */
      for (i=0;i<128;i++) ipar[i]=0;

      t=0.e0;
      h=1.e-7;

/* setting size of the system n, end of integration interval t_end, and initial
value y at t=0 */
      example_v_d_p(&n,&t_end,y);

      time_begin=clock();
/* hybrid solver with user-defined Jacobi matrix computations */
      dodesol_rkm9mka(ipar,&n,&t,&t_end,y,rhs_v_d_p,jacmat_v_d_p,&h,&hm,&ep,&tr,dpar,kd,&
ierr);
      time_end=clock();

      if(ierr!=0)
      {
            printf("\n========================\n");
            printf("DODESOL C example FAILED\n");
            printf("dodesol_rkm9mka routine exited with error code %4d\n",ierr);
            return -1;
        }

        printf("\ndodesol_rkm9mka results\n\n");
        printf("t=%5.1f\n",t);
        printf("Solution      y1=%17.14f,   y2=%17.14f\n",y[0],y[1]);
        printf("-------------------------------------------------------\n");
        printf("CPU time=%f seconds\n", ((double)(time_end-time_begin))/CLOCKS_PER_SEC);
        printf("=======================================================\n\n");
        if(fabs(y[0]-1.878e0)+fabs(y[1]+0.7436e0)>1.e-2)
        {
            printf("Solution seems to be inaccurate. Probably, example FAILED...\n");
            return -1;
        }
        printf("\n========================\n");
        printf("DODESOL C example successfully PASSED through all steps of
computations\n");
        return 0;
}

/*************** Data for Van der Pol equations ****************/
void example_v_d_p(int*n,double*t_end,double*y)
/* The routine initializes the size of the system n, the end of
integration interval t_end, and inital data y at t=0.0 */
{
        *n=2;
        *t_end=160.e0;

        y[0]=2.e0;
        y[1]=0.e0;
}

/************* Right hand side of Van der Pol equations ***********/
void rhs_v_d_p(int*n,double*t,double*y,double*f)
{
        double c;

        c=1.e0-y[0]*y[0];

        f[0]=y[1];
        f[1]=(c*y[1]-y[0])*1.0e6;
}

/******* analytical Jacobi matrix for Van der Pol equations *******/
void jacmat_v_d_p(int*n,double*t,double*y,double*a)
{
/* Please make sure that Jacobi matrix is stored in column-wise order:
a[j*n+i]=df(i)/dx(j) */
        a[0]=0.e0;
        a[1]=-1.e6*(1.e0+2.e0*y[0]*y[1]);
        a[2]=1.e0;
        a[3]=1.e6*(1.e0-y[0]*y[0]);
}
/******************** End of C code example ********************/
```