Specification of PSIDE

Jacques J.B. de Swart
CWI, PO Box 94079, 1090 GB Amsterdam, The Netherlands (Jacques.de.Swart@cwi.nl) &
Paragon Decision Technology, PO Box 3277, 2001 DG Haarlem, The Netherlands (jacques@paragon.nl)

Walter M. Lioen
CWI, PO Box 94079, 1090 GB Amsterdam, The Netherlands (Walter.Lioen@cwi.nl)

Wolter A. van der Veen
formerly at CWI, PO Box 94079, 1090 GB Amsterdam, The Netherlands
MacNeal-Schwendler (E.D.C.) B.V., Groningenweg 6, 2803 PV Gouda, The Netherlands (wolter@macsch.com)

edition December 15, 1998 for version 1.3

ABSTRACT
PSIDE is a code for solving implicit differential equations on parallel computers. It is an implementation of the four-stage Radau IIA method. The nonlinear systems are solved by a modified Newton process, in which every Newton iterate itself is computed by an iteration process. This process is constructed such that the four stage values can be computed simultaneously. We describe here how PSIDE is set up as a modular system and what control strategies have been chosen.

1991 Mathematics Subject Classification: Primary: 65-04, Secondary: 65L05, 65Y05.
1991 Computing Reviews Classification System: G.1.7, G.4.
Keywords and Phrases: numerical software, parallel computers, IVP, IDE, ODE, DAE.
Note: The maintenance of PSIDE belongs to the project MAS2.2: 'Parallel Software for Implicit Differential Equations'.
Acknowledgements: This work is supported financially by the 'Technologiestichting STW' (Dutch Foundation for Technical Sciences), grants no. CWI.2703, CWI.4533. The use of supercomputer facilities was made possible by the 'Stichting Nationale Computerfaciliteiten' (National Computing Facilities Foundation, NCF), with financial support from the 'Nederlandse Organisatie voor Wetenschappelijk Onderzoek' (Netherlands Organization for Scientific Research, NWO).
Note: The latest version of PSIDE and this document can always be found at [SLV98].

1. Introduction

A powerful method for the numerical solution of the system of implicit differential equations

$$\begin{array}{lll} g(t,y,\dot{y}) = 0, & g, y \in \mathbb{R}^d, & \\ t_0 \le t \le t_{\text{end}}, & y(t_0) = y_0, & \dot{y}(t_0) = \dot{y}_0, \end{array} \tag{1.1}$$

is an implicit Runge–Kutta method (IRK). In the class of IRKs, the Radau IIA methods combine high order, $2s-1$, where s is the number of stages, with the nice property of L-stability. However, implementing this method requires high computational costs. PSIDE (abbreviating Parallel Software for Implicit Differential Equations) is an implementation of the four-stage Radau IIA method, where the stages can be computed in parallel. Section 2 describes how this is done.

When implementing an IRK, a lot of decisions have to be made. How to form a prediction for the Newton process, when to refactorize the iteration matrix, when to evaluate the Jacobian, how many Newton iterations should be done, what should be the new stepsize, when to reject a step? The answers to these questions for PSIDE are mainly based on [SS97, GS97, OS99, Ben96, HW96b, Gus92].

In order to have a clear overview of these control strategies, PSIDE is set up modularly. Section 3 shows how several modules build up PSIDE. Section 4–12 each describe one of these modules in detail.

2. Parallelism in PSIDE

For solving (1.1) numerically with the four-stage Radau IIA method, we have to solve $\dot{Y}$ from the nonlinear system

$$G(\mathbb{1}\otimes y_n + h(A\otimes I)\dot{Y}, \dot{Y}) = 0\,. \tag{2.1}$$

Here, $\dot{Y} = (\dot{Y}_1^{\mathrm{T}}, \dot{Y}_2^{\mathrm{T}}, \dot{Y}_3^{\mathrm{T}}, \dot{Y}_4^{\mathrm{T}})^{\mathrm{T}}$ is the so-called stage derivative vector of dimension $4d$, where the $\dot{Y}_i$ contain approximations to the derivative values $\dot{y}(t_n + c_i h)$, the abscissa are in $c = (c_1, c_2, c_3, c_4)^{\mathrm{T}}$, the stepsize is denoted by h, the 4×4 Radau IIA matrix by A, the approximation to $y(t_n)$ by y_n, the Kronecker product by $\otimes$, the vector $(1,1,1,1)^{\mathrm{T}}$ by $\mathbb{1}$, and G stands for the stacked values of g, i.e.

$$G(\mathbb{1}\otimes y_n + h(A\otimes I)\dot{Y}, \dot{Y}) := \begin{pmatrix} g(t_n + c_1 h, y_n + h\sum_j a_{1j}\dot{Y}_j, \dot{Y}_1) \\ \vdots \\ g(t_n + c_4 h, y_n + h\sum_j a_{4j}\dot{Y}_j, \dot{Y}_4) \end{pmatrix}.$$

Here and in the sequel, I is an identity matrix of dimension either 4 or d, but its dimension will always be clear from the context. For Radau IIA, $c_s = 1$. Once we obtained $\dot{Y}$, we compute the stage vector Y and y_{n+1} from

$$Y = \mathbb{1}\otimes y_n + h(A\otimes I)\dot{Y}, \qquad y_{n+1} = (e_s^{\mathrm{T}}\otimes I)Y\,,$$

where $e_s = (0,0,0,1)^{\mathrm{T}}$. To solve (2.1) we apply a modified Newton process and apply the Butcher transformation $AT = T\Lambda$, where Λ is a block diagonal matrix containing 2 blocks of dimension 2×2, thus arriving at

Scheme:

$\dot{Y}$ `is given by predictor`
`repeat until convergence`
  $Y \leftarrow \mathbb{1}\otimes y_n + h(A\otimes I)\dot{Y}$
  `solve` $(I\otimes M + h\Lambda\otimes J)\Delta\dot{W} = -(T^{-1}\otimes I)G(Y,\dot{Y})$
  $\dot{Y} \leftarrow \dot{Y} + (T\otimes I)\Delta\dot{W}$
`end`

where $M = \partial g/\partial\dot{y}$ and $J = \partial g/\partial y$, both evaluated at some previous approximations. By $P \leftarrow Q$ we mean that Q is assigned to P.

Remark It would also have been possible to define $Z = Y - \mathbb{1}\otimes y_n$, apply a modified Newton process to

$$G(\mathbb{1}\otimes y_n + Z, ((hA)^{-1}\otimes I)Z) = 0\,,$$

and iterate on Z. Applying Butcher transformations $A^{-1}T = T\Lambda^{-1}$ would then lead to

Scheme:

```
Z is given by predictor
repeat until convergence
   Ẏ ← ((hA)^{-1} ⊗ I)Z
   solve ((hΛ)^{-1} ⊗ M + I ⊗ J)ΔW = −(T^{-1} ⊗ I)G(𝟙 ⊗ y_n + Z, Ẏ)
   Z ← Z + (T ⊗ I)ΔW
end
```

which is equally expensive as iterating on $\dot{Y}$. We prefer not to use additional quantities Z. However, for the problem class $M\dot{y} = f(y)$, iterating on Z is somewhat cheaper. ◇

To compute $\Delta\dot{W}$ we would need to solve two linear systems of dimension $2d$. Instead of doing this, we solve $\Delta\dot{W}$ by the Parallel Iterative Linear system Solver for Runge–Kutta methods (PILSRK) proposed in [HS97]. In PILSRK we split the matrix Λ in $L + (\Lambda - L)$, where L has distinct positive eigenvalues, and rewrite the equation as

$$(I \otimes M + hL \otimes J)\Delta\dot{W} = (h(L - \Lambda) \otimes J)\Delta\dot{W} - (T^{-1} \otimes I)G(Y, \dot{Y})\,.$$

Now we perform an iteration process according to

$$(I \otimes M + hL \otimes J)\Delta\dot{W}^j = (h(L - \Lambda) \otimes J)\Delta\dot{W}^{j-1} - (T^{-1} \otimes I)G(Y, \dot{Y})\,. \tag{2.2}$$

If J is a full matrix, then the computation of $h(L-\Lambda)\otimes J$ can be expensive. Since for most applications M is sparser than J (e.g., for ordinary differential equations, $M = \pm I$), we rewrite (2.2) as

$$\begin{aligned}(I \otimes M + hL \otimes J)\Big(\Delta\dot{W}^j - \big((I - L^{-1}\Lambda) \otimes I\big)\,\Delta\dot{W}^{j-1}\Big) = \\ -\big((I - L^{-1}\Lambda) \otimes M\big)\,\Delta\dot{W}^{j-1} - (T^{-1} \otimes I)G(Y, \dot{Y})\,.\end{aligned} \tag{2.3}$$

Applying again Butcher transformations $LS = SD$, where D is a diagonal matrix, and m iterations of type (2.3), leads to

Scheme:

```
Ẏ is given by predictor
repeat until convergence
   Y ← 𝟙 ⊗ y_n + h(A ⊗ I)Ẏ
   ΔV̇^0 ← 0
   do j = 1,...,m
      solve (I ⊗ M + hD ⊗ J)(ΔV̇^j − (B ⊗ I)ΔV̇^{j-1}) =
            −(B ⊗ M)ΔV̇^{j-1} − (Q^{-1} ⊗ I)G(Y, Ẏ)
   end
   Ẏ ← Ẏ + (Q ⊗ I)ΔV̇^m
end
```

where $B = I - (LS)^{-1}\Lambda S$ and $Q = TS$. We now see that the 4 components of dimension d in $\Delta\dot{V}^k$, each corresponding to one stage, can be computed in parallel.

How to form the matrices T, L and S such that PILSRK converges rapidly, can be found in [HS97]. However, the appendix of this specification lists all the matrices that play a role in the derivation above.

3. PSIDE as modular scheme

Figure 1 shows the modules of PSIDE. If a line connects two modules, then the upper module ‘calls’ the lower one. Table 1 describes what the modules basically do. The next sections describe them in more detail.

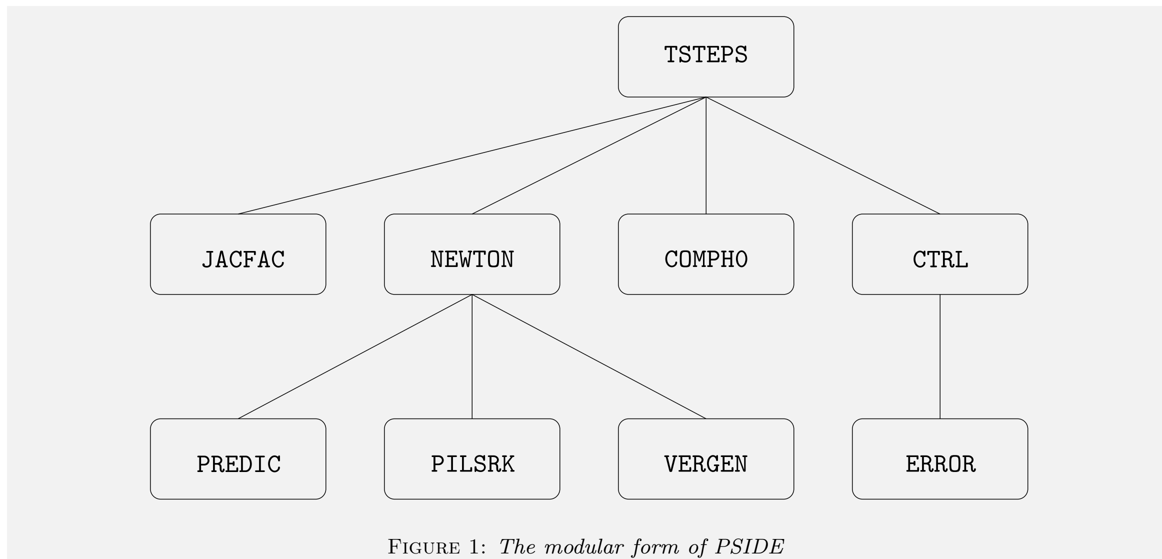

FIGURE 1: *The modular form of PSIDE*

Since in an actual implementation, we do not store the sequences $t_0, t_1, \ldots; y_0, y_1, \ldots$ and $\dot{y}_0, \dot{y}_1, \ldots,$ in the sequel the values t, y and $\dot{y}$ denote the current timepoint, and the approximations to the solution and its derivative at t, respectively.

4. Module TSTEPS

This module performs the iteration in time. It uses `COMPHO`, `JACFAC`, `NEWTON`, `CTRL`.

Scheme:

compute h by COMPHO
$h_{LU} \leftarrow h$
$t \leftarrow t_0$
$\dot{y} \leftarrow \dot{y}_0$
$\dot{Y}_p \leftarrow \mathbb{1} \otimes \dot{y}$
first $\leftarrow$ true
jacnew $\leftarrow$ true
facnew $\leftarrow$ true
while ($t < t_{\text{end}}$) do
  depending on jacnew, facnew compute M, J, LU by JACFAC
  compute $Y, \dot{Y}, \alpha$, growth, diver, conver by NEWTON
  compute $y, \dot{y}, \dot{Y}_p, t, h_p, h, h_{LU}$, first, jacnew, facnew by CTRL
end

Depending on the boolean variables `jacnew` and `facnew`, module `JACFAC` evaluates the Jacobian matrices M and J and factorizes the iteration matrix $I \otimes M + h_{LU} D \otimes J$. These booleans `jacnew` and `facnew` and the stepsize used for this factorization, h_{LU}, are determined in the control module `CTRL`.

Apart from the vectors $\dot{Y}$ and Y, module `NEWTON` gives as output the estimated rate of convergence α, and the boolean variables `growth`, `diver` and `solved`. If the growth of the current iterate has a too large increment with respect to y, then `growth` is true, whereas `diver` and `solved` tell that the Newton process is diverging or has converged. Based on α, `growth`, `diver` and `solved`, module `CTRL` decides whether the current $\dot{Y}$ and Y can be accepted. If so, then it assigns $t \leftarrow t + h$, $y \leftarrow Y_s$, $\dot{y} \leftarrow \dot{Y}_s$ and proposes a new stepsize. The old stepsize and old stage derivative vector are stored in h_p and $\dot{Y}_p$,

TABLE 1: *Overview of tasks of modules in Figure 1.*

Module	Function
TSTEPS	performs the time-stepping
JACFAC	evaluates the Jacobians and factorizes the iteration matrix
NEWTON	performs the Newton iteration
PREDIC	computes the prediction to start the Newton iteration
COMPHO	computes the initial stepsize
CTRL	decides whether the Jacobians should be updated and the iteration matrix should be factorized
PILSRK	the Parallel Iterative Linear system Solver for Runge–Kutta methods
VERGEN	checks whether the Newton iterates diverge or converge
ERROR	computes the local error estimate

respectively. If the step is not accepted, then t, y and $\dot{y}$ are not changed and h is recomputed.

5. Module JACFAC

Depending on `jacnew` and `facnew`, this module evaluates the Jacobians and factorizes the iteration matrix. If the Jacobians are re-evaluated, the boolean `jacu2d` (Jacobians up to date) is set true.

Variables:

input	jacnew, facnew, h_{LU}
output	jacu2d, M, J, LU

Scheme:

```
if (jacnew) then
  form the Jacobian matrices M, J
  jacu2d ← true
end
if (facnew) then
  factorize the iteration matrix I ⊗ M + h_LU D ⊗ J
end
```

6. Module NEWTON

Module `NEWTON`, which uses the modules `PREDIC`, `PILSRK` and `VERGEN`, performs the Newton iteration.

Variables:

input	$y, \dot{Y}_p, LU, M, t, h_p, h$
output	$Y, \dot{Y}, \alpha$, growth, diver, slow, solved, exact
local	$\Delta Y, \Delta\dot{Y}, k$, ready

Scheme:

```
ready ← false
k ← 0
compute Ẏ by PREDIC
Y ← 𝟙 ⊗ y + h(A ⊗ I)Ẏ
compute ready, growth by VERGEN
while (not(ready)) do
  k ← k + 1
  compute ΔẎ by PILSRK
  ΔY ← h(A ⊗ I)ΔẎ
  Y ← Y + ΔY
  Ẏ ← Ẏ + ΔẎ
  compute α, growth, diver, slow, solved, exact by VERGEN
end
```

The estimated rate of convergence α and the booleans `growth`, `diver`, `slow`, `solved` and `exact` are computed for use in module `CTRL`.

7. Module PREDIC

As starting value for the Newton process we use fourth order extrapolation of the previous stage derivative vector, i.e.

$$\dot{Y} = EY_p,$$

where the $s \times s$ matrix E is determined by the order conditions

$$E(c - \mathbb{1})^k = \left(\frac{h}{h_p}c\right)^k, \qquad k = 0, 1, \ldots, s-1\,.$$

(Here, for any vector $a = (a_i)$, the kth power a^k, is understood to be the vector with entries a_i^k.) This means that

$$E = VU^{-1}, \quad U := \begin{bmatrix} \mathbb{1} & c - \mathbb{1} & \cdots & (c-\mathbb{1})^{s-1} \end{bmatrix}, \quad V := \begin{bmatrix} \mathbb{1} & \frac{h}{h_p}c & \cdots & \left(\frac{h}{h_p}\right)^{s-1} \end{bmatrix}.$$

Variables:

input	$\dot{Y}_p, h_p, h$
output	$\dot{Y}$

Scheme: $\dot{Y} \leftarrow EY_p$

We experimented with a lot of other predictors than fourth order extrapolation: Extrapolation of order 3 (using only 3 of the 4 stages in $\dot{Y}_p$), a polynomial of degree 2 that fitted the four stages in Y_p in an—in least squares sense—optimal way, a last step point value predictor (i.e. $\dot{Y} = (\mathbb{1} e_s^{\mathrm{T}} \otimes I)\dot{Y}_p$), and a starting value that is one of these predictors, depending on which predictor yields the smallest residual. We also tried several rational approximations. However, numerous experiments showed that fourth-order extrapolation yields the best overall performance.

8. Module COMPH0

This module computes the initial stepsize h_0. It is similar to the strategy used in DASSL [Pet91].

Variables:

input	$\dot{y}_0, t_0, t_{\text{end}}$
output	h_0
local	$\zeta, h_{\text{def}}, f_h$

Scheme:

```
h_0 ← min{h_def, f_h |t_end − t_0|}
if ( ||ẏ_0||_scal > ζ/h_0 ) then
   h_0 ← ζ/||ẏ_0||_scal
end
h_0 ← sign(h_0, t_end − t_0)
```

Parameters:

	value	*source*
ζ	0.5	[Pet91]
h_{def}	10^{-5}	[HW96a]
f_h	10^{-5}	experience

9. Module CTRL

This module is a modified version of the one presented in [GS97, Figure 4].

Variables:

input	$y, Y, \dot{y}, \dot{Y}, LU, t, t_{\text{end}}, h, h_{LU}, \alpha,$ `first`, `growth`, `diver`, `slow`, `solved`, `jacu2d`, `jacnew`, `exact`
output	$y, \dot{y}, \dot{Y}_p, t, h_p, h, h_{LU},$ `first`, `jacnew`, `facnew`, `jacu2d`
local	$h_r, h_{\text{new}}, h_\alpha, \alpha_{\text{ref}}, \alpha_{\text{jac}}, \alpha_{LU}, f_{\text{min}}, f_{\text{max}}, f_{\text{rig}}, \xi, \omega, n_{\text{rem}}$

Scheme:

```
if (not(growth)) $h_\alpha \leftarrow h\alpha_{\rm ref}/\max\{\alpha, \alpha_{\rm ref}/f_{\max}\}$
if (solved) then
  compute $y, \dot{y}, \dot{Y}_p, t, h_p, h_r$, first by ERROR
  if ( $|t_{\rm end} - t| > 10u_{\rm round}|t|$ ) then
    if jacu2d $\wedge\, \alpha > \alpha_{\rm ref}$ ) then
      $h_{\rm new} \leftarrow \min\{f_{\max}h, \max\{f_{\min}h, \min\{h_r, h_\alpha\}\}\}$
    else
      $h_{\rm new} \leftarrow \min\{f_{\max}h, \max\{f_{\min}h, h_r\}\}$
    end
    if (not(exact) $\wedge\, \alpha - |h - h_{LU}|/h_{LU} > \alpha_{\rm jac}$ ) then
      if (jacu2d) then
        $h_{\rm new} \leftarrow h/f_{\rm rig}$
      else
        jacnew ← true
      end
    end
  end
elseif (growth) then
  $h_{\rm new} \leftarrow h/f_{\rm rig}$
elseif (diver) then
  $h_{\rm new} \leftarrow \min\{f_{\max}h, \max\{f_{\min}h, h_\alpha\}\}$
  jacnew ← not(jacu2d)
elseif (slow) then
  if (jacu2d) then
    if ( $\alpha > \xi\alpha_{\rm ref}$ ) then
      $h_{\rm new} \leftarrow \min\{f_{\max}h, \max\{f_{\min}h, h_\alpha\}\}$
    else
      $h_{\rm new} \leftarrow h/f_{\rm rig}$
    end
  else
    $h_{\rm new} \leftarrow h$; jacnew ← true
  end
end
$n_{\rm rem} \leftarrow (t_{\rm end} - t)/h_{\rm new}$
if ( $n_{\rm rem} - \lfloor n_{\rm rem} \rfloor > \omega \vee \lfloor n_{\rm rem} \rfloor = 0$ ) then
  $n_{\rm rem} \leftarrow \lfloor n_{\rm rem} \rfloor + 1$
else
  $n_{\rm rem} \leftarrow \lfloor n_{\rm rem} \rfloor$
end
$h \leftarrow (t_{\rm end} - t)/n_{\rm rem}$
facnew ← jacnew $\vee (|h - h_{LU}|/h_{LU} > \alpha_{LU})$
if (facnew) $h_{LU} \leftarrow h$
```

Parameters:

	value	*source*
α_{ref}	0.15	see below
α_{jac}	0.1	[GS97]
α_{LU}	0.2	see below
$f_{\min}$	0.2	[Ben96, p.52]
$f_{\max}$	2	experience
f_{rig}	2	[Gus92, p.154]
ξ	1.2	experience
ω	0.05	[SSV97]

In this scheme, α_{ref} is the desired rate of convergence. In [GS97] it is shown that, under reasonable assumptions, h_α is the stepsize for which the rate of convergence will be α_{ref}. If the current rate of convergence α is larger than α_{ref}, then h_α will be used for the next Newton process, unless it is greater than h_r, the stepsize proposed by module `ERROR`. [GS97] also derives that, if $\alpha - |h - h_{LU}|/h_{LU} > \alpha_{\text{jac}}$, then the convergence of the Newton process is likely to fail due to an old Jacobian. Such failures are prevented by the strategy above. If nevertheless the Jacobian is fresh, then the assumptions of the theory are not fulfilled and the stepsize is reduced by a rigid factor f_{rig}. The case where `exact` is true, refers to the situation that that (2.1) was solved exactly. This happens e.g. if the function g in (1.1) equals $\dot{y}$.

If `growth` is true (see §11), then the stepsize is reduced by a factor f_{rig}, but we do not compute new Jacobians.

For a diverging Newton process, h_α will be the new stepsize. If `slow` is true, then the Newton process is converging too slowly (see §11). In this case, the new stepsize is again identified with h_α, if $\alpha > \xi\alpha_{\text{ref}}$. The factor ξ, which has to be > 1, is built in for the case that α is only slightly larger than α_{ref}. Without this factor, the new stepsize for this case would be set equal to h_α, which is only a little bit smaller than the old stepsize, thus leading again to a Newton process that converges too slowly. If $\alpha \leq \xi\alpha_{\text{ref}}$, then the assumptions of the analysis have failed to hold, and the stepsize is rigidly reduced by a factor f_{rig}. For both the diverging and the slowly converging case, the iteration matrix will be factorized, with or without a new Jacobian, depending on `jacnew`.

The formulas of the form $h_{\text{new}} = \min\{f_{\max}h, \max\{f_{\min}h, \cdot\}\}$ prevent the new stepsize to vary from the old stepsize by a factor outside the range $[f_{\min}, f_{\max}]$.

This strategy for updating the Jacobian and refactorizing the iteration matrix is different from most strategies in ODE/DAE software, in the sense that it attempts to adjust the stepsize such that an optimal convergence rate of the Newton process is obtained. Another difference is the strategy for the case that the Jacobian is not updated. Many codes do not change the stepsize in this situation if the relative change of stepsize is in a 'dead-zone' (e.g. RADAU5 uses the dead-zone $[1; 1.2]$). However, in [Gus92, p.135] it is argued that, in order to arrive at a smooth numerical solution, it is better to remove this strategy 'since a smooth stepsize sequence leads to smoother error control'.

The role of n_{rem} is to adjust the stepsize such that the remaining integration interval is a multiple of the stepsize. This strategy was taken from [SSV97].

If a new Jacobian is not required, then the iteration matrix will only be factorized in case that the proposed stepsize h_{new} differs significantly from h_{LU}, i.e. $(|h_{\text{new}} - h_{LU}|/h_{LU} > \alpha_{LU})$.

The papers [GS97] and [Gus92] advocate values of α_{ref}, α_{jac} and α_{LU} around 0.2. However, they also suggest to use larger values if the costs of factorizing the iteration matrix are high with respect to the costs of one iteration. Since PSIDE is a parallel code aiming at problems of large dimension, we choose 0.25 for α_{ref} and 0.3 for α_{LU}. Numerous experiments confirmed that these choices yield an efficient code.

10. Module PILSRK

This module is the same as presented in §2, although an economization is made by computing the first iterate separately. Numerous experiments showed that for index 0 and index 1 problems, performing only 1 inner iteration suffices. On the other hand, [HV97] reveals that for higher-index problems, 2 inner iterations lead to a more robust and efficient behavior.

Variables:

input	$Y, \dot{Y}, LU, M, t, h$
output	$\Delta\dot{Y}$
local	$\Delta\dot{V}, \widetilde{G}, j, m$

Scheme:

$\widetilde{G} \leftarrow (Q^{-1} \otimes I)G(Y, \dot{Y})$
$\Delta\dot{V} \leftarrow -(LU)^{-1}\widetilde{G}$
`if (higher index) then`
  `do` $j = 2, \ldots, m$
    $\Delta\dot{V} \leftarrow (B \otimes I)\Delta\dot{V} - (LU)^{-1}((B \otimes M)\Delta\dot{V} + \widetilde{G})$
  `end`
`end`
$\Delta\dot{Y} \leftarrow (Q \otimes I)\Delta\dot{V}$

Parameters:

	value	*source*
m	2	experience

11. Module VERGEN

This module checks the convergence behavior of the Newton process. Most of it is based on [Gus92, §5.2].

Variables:

input	$y, Y, \Delta Y, h, k, \alpha$
output	α, `ready`, `growth`, `diver`, `slow`, `solved`, `exact`
local	$u_p, u, \tau, \kappa, k_{\max}, \gamma, \alpha_1$

Scheme:

```
growth ← false
diver ← false
slow ← false
solved ← false
exact ← false
if ( ∃ i |Y_{i,s}|/max{|y_i|, atol_i} > g_fac ∧ ind_i ≤ 1 ) then
  growth ← true
else
  if ( k = 1 ) then
    u ← ||ΔY||_scal
    α = α_1
    exact ← u = 0
    solved ← exact
  elseif ( k > 1 ) then
    u_p ← u
    u ← ||ΔY||_scal
    α ← α^θ (u/u_p)^{1−θ}
    if ( α ≥ γ ) then
      diver ← true
    elseif ( u α/(1 − α) < τ ∨ u < κ u_round ||y||_scal ) then
      solved ← true
    elseif ( k = k_max ∨ u α^{k_max−k}/(1 − α) > τ ) then
      slow ← true
    end
  end
end
ready ← growth ∨ diver ∨ slow ∨ solved ∨ exact
```

Parameters:

	value	*source*
τ	0.01	see below
κ	100	[Ben96, p.51]
$k_{\max}$	15	see below
γ	1	[Gus92, p.132]
θ	0.5	experience
g_{fac}	100	experience
α_1	0.1	experience

The boolean variable `growth` monitors whether the current iterate is too large with respect to y. This is necessary to prevent overflow. We use $\max\{|y_i|, \texttt{atol}_i\}$ instead of $|y_i|$ for the case $y_i = 0$. Experience has shown that it is not efficient to put a limit on the growth of higher-index variables. The variable u is saved for use in the next call of `VERGEN`. The case where `slow` is true, refers to a Newton process that is converging too slowly.

Here and in the sequel, the norm $\|\cdot\|_{\text{scal}}$ is defined by

$$\|X\|_{\text{scal}} = \sqrt{\frac{1}{4d}\sum_{i=0}^{3}\sum_{j=1}^{d}\left(\frac{h^{\texttt{ind}_j-1}X_{id+j}}{\texttt{atol}_j + \texttt{rtol}_j\,|y_j|}\right)^2}, \qquad \text{if} \qquad X \in \mathbb{R}^{4d},$$

and by

$$\|x\|_{\text{scal}} = \sqrt{\frac{1}{d}\sum_{j=1}^{d}\left(\frac{h^{\texttt{ind}_j-1}x_j}{\texttt{atol}_j + \texttt{rtol}_j\,|y_j|}\right)^2}, \qquad \text{if} \qquad x \in \mathbb{R}^{d}.$$

In these formulas, $\texttt{atol}_j$ and $\texttt{rtol}_j$ are the user-supplied absolute and relative error tolerance vectors, respectively, and $\texttt{ind}_j$ contains the user-supplied index of component j. For both index 0 and index 1 variables, $\texttt{ind}_j = 1$.

In the first iterate, we initialize α by α_1. The value of τ is rather small compared to termination criteria in other codes. E.g., in RADAU5 values around 10^{-1} or 10^{-2} were found to be efficient [HW96b, p.121], and DASSL uses 0.33 [BCP89, p.123]. The reason for this is that τ can be seen as the factor by which the iteration error has to be smaller than the error estimate ϵ. In PSIDE, ϵ is of local order 5 (see §12), and the steppoint value of local order 8. Consequently, in order not to let the iteration error spoil the accuracy of the steppoint value, τ has to be smaller than 1, and how much smaller than 1 should depend on the difference between the order of the error estimate and that of the method. This may in part explain why RADAU5, where this difference is 2, and DASSL, where it is 1, use larger values for τ.

The reason for the rather large value of $k_{\max}$ ([HW96b, p.121] advocates values of 7 or 10 for RADAU5) is twofold. Firstly, the order of PSIDE is seven, which is higher than that of e.g. the fifth order RADAU5, so that we need more iterations to find the solution of the non-linear system. Secondly, if PILSRK does not find the exact Newton iterate, a few additional iterations might help.

12. Module ERROR

12.1 The error estimate in PSIDE

The construction of the error estimate is based on [SS97]. In order not to have confusion between the previous values of y and $\dot{y}$ and the current ones, we use the time index n in the derivation of the error estimate.

To estimate the error, we use an implicit embedded formula of the form

$$\hat{y}_{n+1} = y_n + h(b_0\dot{y}_n + (b^{\mathrm{T}} \otimes I)\dot{Y} + d_s\dot{\hat{y}}_{n+1})\,, \tag{12.1}$$

where d_s is the lower right element in D. We eliminate $\dot{\hat{y}}_{n+1}$ by substituting (12.1) in (1.1) yielding

$$g\big(t_{n+1}, \hat{y}_{n+1}, (hd_s)^{-1}(\hat{y}_{n+1} - y_n - h(b_0\dot{y}_n + (b^{\mathrm{T}} \otimes I)\dot{Y}))\big) = 0\,. \tag{12.2}$$

Solving $\hat{y}_{n+1}$ from (12.2) by a modified Newton process, leads to the recursion

$$\begin{aligned} \hat{y}^{j+1}_{n+1} &= \hat{y}^{j}_{n+1} - hd_s(M + h_{LU}d_sJ)^{-1}g\big(t_{n+1}, \hat{y}^{j}_{n+1}, \\ &\quad (hd_s)^{-1}(\hat{y}^{j}_{n+1} - y_n - h(b_0\dot{y}_n + (b^{\mathrm{T}} \otimes I)\dot{Y}))\big)\,. \end{aligned}$$

Now we set $\hat{y}^{(0)}_{n+1} = y_{n+1}$, and consider the first Newton iterate $\hat{y}^{1}_{n+1}$ as a reference formula by itself, which is

$$\begin{aligned} \hat{y}^{1}_{n+1} &= y_{n+1} - hd_s(M + h_{LU}d_sJ)^{-1}g\big(t_{n+1}, y_{n+1}, \\ &\quad (hd_s)^{-1}(\hat{y}_{n+1} - y_n - h(b_0\dot{y}_n + (b^{\mathrm{T}} \otimes I)\dot{Y}))\big)\,. \end{aligned}$$

In this formula, we determine b such that $\hat{y}^{1}_{n+1}$ is of local order $s+1$, i.e., it satisfies

$$C\,b = (1 - b_0, 1/2, 1/3, \ldots, 1/s)^{\mathrm{T}} - d_s\mathbb{1}\,, \tag{12.3}$$

where $C = (c_{ij})$; $c_{ij} = c_j^{i-1}$. Notice that implying order conditions directly on (12.1) would also lead to (12.3). The paper [SS97] describes how to select the parameter b_0 such that the amplitude of the error estimate approximates the true error in y_{n+1}. Carrying out this procedure for the four-stage Radau IIA method yields the value 0.01 for b_0.

We now define the error estimate r by

$$\begin{aligned} r &= \hat{y}^{1}_{n+1} - y_{n+1} \\ &= -hd_s(M + h_{LU}d_sJ)^{-1}g\big(t_{n+1}, y_{n+1}, d_s^{-1}((v^{\mathrm{T}} \otimes I)\dot{Y} - b_0\dot{y}_n)\big)\,, \end{aligned} \tag{12.4}$$

where $v = (v_i)$, with $v_i = a_{si} - b_i$.

We notice that the error estimate (12.4) reduces for ODE problems to the same formula as in RADAU5 [HW96b, p.123, Formula (8.19)]. However, by choosing the reference method as being of the form (12.1), the 'filtering' with the matrix $(M + h_{LU} d_s J)^{-1}$, needed to 'remove' the stiff error components in the error estimate, arises on purely mathematical grounds.

12.2 Stepsize selection

The following module contains the predictive stepsize controller of Gustafsson [Gus92, Listing 5.1] to propose a new stepsize h_r.

Variables:

input	$y, Y, \dot{y}, \dot{Y}, LU, t, h_p, h, f_{\max}$, `first`
output	$y, \dot{y}, \dot{Y}_p, t, h_p, h_r$, `first`, `jacu2d`
local	$r, h_{\text{rej}}, \epsilon_p, \epsilon_{\text{rej}}, \epsilon, p_{\text{est}}, p_{\min}$, `sucrej`

Scheme:

```
compute r from (12.4)
ε ← ||r||_scal
if ( ε < 1 ) then
  if ( ε = 0 ) then
    h_new ← f_max h
  elseif (first ∨ sucrej) then
    first ← false
    h_r ← ζ h ε^(-1/5)
  else
    h_r ← ζ h^2/h_p (ε_p/ε^2)^(1/5)
  end
  y ← Y_s
  ẏ ← Ẏ_s
  Ẏ_p ← Ẏ
  t ← t + h
  if ( |t_end − t| < 10u_round)|t|) then t ← t_end
  h_p ← h
  ε_p ← ε
  sucrej ← false
  jacu2d ← false
else
  if (not(first) ∧ sucrej) then
    p_est ← min{5, max{p_min, log(ε/ε_rej)/log(h/h_rej)}}
    h_r ← ζ h ε^(-1/p_est)
  else
    h_r ← ζ h ε^(-1/5)
  end
  h_rej ← h
  ε_rej ← ε
  sucrej ← true
end
```

Parameters:

	value	*source*
ζ	0.8	[Gus92, p.156]
$p_{\min}$	0.1	[Gus92, p.121]

The variables h_{rej}, ϵ_p, ϵ_{rej} and `sucrej` are saved for use in the next call of `ERROR`. Notice that this

module is only called if the Newton process has converged. If $\|\epsilon\|_{\text{scal}} < 1$, it steps forward in time, i.e. it updates t, y, $\dot{y}$ and shifts $\dot{Y}_p$, h_p and ϵ_p; the Jacobians are per definition not up to date anymore.

Appendix

In this appendix we provide the method parameters in PSIDE, i.e. the abscissa vector c and the RK matrix A, that define the four-stage Radau IIA method, the matrices D, B, Q and Q^{-1}, defining the Parallel Linear system Solver for Runge–Kutta methods (PILSRK), and the scalar b_0 and the vector v, needed for the embedded reference formula. As additional information, we list the matrices Λ, L, S and T that arose in the derivation of PILSRK.

$$c^{\mathrm{T}} = \begin{pmatrix} 0.08858795951268 & 0.40946686444074 & 0.78765946176085 & 1.00000000000000 \end{pmatrix}$$

$$A = \begin{bmatrix} 0.11299947932312 & -0.04030922072350 & 0.02580237742032 & -0.00990467650726 \\ 0.23438399574737 & 0.20689257393542 & -0.04785712804857 & 0.01604742280653 \\ 0.21668178462322 & 0.40612326386742 & 0.18903651817002 & -0.02418210489982 \\ 0.22046221117674 & 0.38819346884323 & 0.32884431998002 & 0.06250000000001 \end{bmatrix}$$

$$\mathrm{diag}(D) = \begin{bmatrix} 0.15207736897658 & 0.19863166560206 & 0.17370482124555 & 0.22687976652481 \end{bmatrix}$$

$$B = \begin{bmatrix} -3.36398745680207 & -0.44654700754010 & 0 & 0 \\ 25.34203884124225 & 3.36398745680207 & 0 & 0 \\ 0 & 0 & -0.43736727682531 & -0.05805760311840 \\ 0 & 0 & 3.29483348541735 & 0.43736727682531 \end{bmatrix}$$

$$Q = \begin{bmatrix} 2.95257334306175 & 0.31594239005361 & 1.53250361857179 & 0.02760017730665 \\ -7.26634778465530 & -0.87557678542461 & -1.05525925554832 & -0.31127768044595 \\ 3.42024269744602 & 0.94929336342678 & -10.79971906268609 & -2.13491394363799 \\ 34.89702510456449 & 4.37526650476817 & -42.90392657810952 & -5.89600020104167 \end{bmatrix}$$

$$Q^{-1} = \begin{bmatrix} 0.49403714522764 & 0.26941265525930 & -0.20775393051682 & 0.06331582713183 \\ -3.53352093058280 & -2.98586378845007 & 1.75646110158256 & -0.49490947213933 \\ 0.48764145508107 & 0.12393820514650 & 0.04237703393234 & -0.01960507515011 \\ -3.24650638474176 & -1.52301305545687 & -0.23459121597752 & -0.01945253030841 \end{bmatrix}$$

$$b_0 = 0.01$$

$$v^{\mathrm{T}} = \begin{pmatrix} 0.01577537639774 & -0.00973676595201 & 0.00646138955427 & 0.22437976652485 \end{pmatrix}$$

$$\Lambda = \begin{bmatrix} 0.15207736897658 & 0.06790969403105 & 0 & 0 \\ -0.35070903457864 & 0.04202359569373 & 0 & 0 \\ 0 & 0 & 0.17370482124555 & 0.01008488557162 \\ 0 & 0 & -0.40058458777036 & 0.20362278551270 \end{bmatrix}$$

$$L = \begin{bmatrix} 0.15207736897658 & 0 & 0 & 0 \\ -0.35070903457864 & 0.19863166560206 & 0 & 0 \\ 0 & 0 & 0.17370482124555 & 0 \\ 0 & 0 & -0.40058458777036 & 0.22687976652481 \end{bmatrix}$$

$$S = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 7.53333333333333 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 7.53333333333333 & 1 \end{bmatrix}$$

$$T = \begin{bmatrix} 0.57247400465791 & 0.31594239005361 & 1.32458228286171 & 0.02760017730665 \\ -0.67033600112323 & -0.87557678542461 & 1.28969927047784 & -0.31127768044595 \\ -3.73110064036903 & 0.94929336342678 & 5.28329931272012 & -2.13491394363799 \\ 1.93668410197760 & 4.37526650476817 & 1.51260826973776 & -5.89600020104167 \end{bmatrix}$$

References

[BCP89] K.E. Brenan, S.L. Campbell, and L.R. Petzold. *Numerical Solution of Initial–Value Problems in Differential–Algebraic Equations*. North–Holland, New York–Amsterdam–London, 1989.

[Ben96] C. Bendtsen. *Parallel Numerical Algorithms for the Solution of Systems of Ordinary Differential Equations*. PhD thesis, Institute of Mathematical Modelling, Technical University of Denmark, June 1996.

[GS97] K. Gustafsson and G. Söderlind. Control strategies for the iterative solution of nonlinear equations in ODE solvers. *SIAM Journal on Scientific and Statistical Computing*, 18(1):23–40, Jan 1997.

[Gus92] K. Gustafsson. *Control of Error and Convergence in ODE Solvers*. PhD thesis, Lund Institute of Technology, 1992.

[HS97] P.J. van der Houwen and J.J.B. de Swart. Parallel linear system solvers for Runge–Kutta methods. *Advances in Computational Mathematics*, 7:157–181, 1997.

[HV97] P.J. van der Houwen and W.A. van der Veen. The solution of implicit differential equations on parallel computers. *Applied Numerical Mathematics*, 25:257–274, 1997.

[HW96a] E. Hairer and G. Wanner. *RADAU5*, July 9, 1996. Available at `ftp://ftp.unige.ch/pub/doc/math/stiff/radau5.f`.

[HW96b] E. Hairer and G. Wanner. *Solving Ordinary Differential Equations II: Stiff and Differential-algebraic Problems*. Springer-Verlag, second revised edition, 1996.

[OS99] H. Olsson and G. Söderlind. Stage value predictors and efficient Newton iterations in implicit Runge–Kutta methods. *SIAM Journal on Scientific Computing*, 20(1):185–202, 1999.

[Pet91] L.R. Petzold. *DASSL: A Differential/Algebraic System Solver*, June 24, 1991. Available at `http://www.netlib.org/ode/ddassl.f`.

[SLV98] J.J.B. de Swart, W.M. Lioen, and W.A. van der Veen. *PSIDE*, November 25, 1998. Available at `http://www.cwi.nl/cwi/projects/PSIDE/`.

[SS97] J.J.B. de Swart and G. Söderlind. On the construction of error estimators for implicit Runge–Kutta methods. *Journal of Computational and Applied Mathematics*, 86(2):347–358, 1997.

[SSV97] B.P. Sommeijer, L.F. Shampine, and J.G. Verwer. *RKC*, 1997. Available at `http://www.netlib.org/ode/rkc.f`.