SUNDIALS: SUite of Nonlinear and DIfferential / ALgebraic Equation Solvers

Radu Şerban

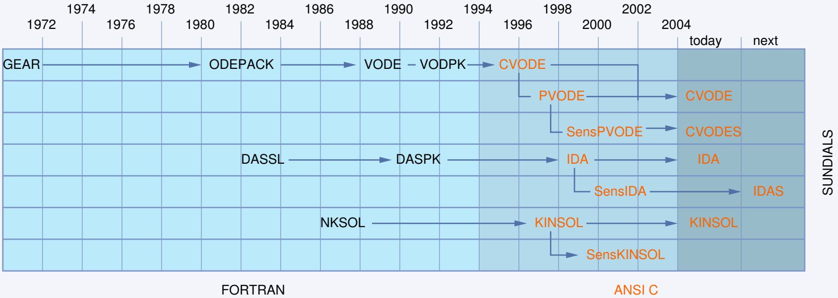
Center for Applied Scientific Computing
Lawrence Livermore National Laboratory

Sixth DOE ACTS Collection Workshop
August 25, 2005

Outline

Outline

Historical background



Solution of large systems in parallel motivated writing (or rewriting) solvers in C

- **CVODE** C rewrite of **VODE**/**VODPK** [Cohen, Hindmarsh, 1994]
- **PVODE** parallel extension of **CVODE** [Byrne, Hindmarsh, 1998]
- **KINSOL** C rewrite of **NKSOL** [Taylor, Hindmarsh, 1998]
- **IDA** C rewrite of **DASPK** [Hindmarsh, Taylor, 1999]

New sensitivity capable solvers in SUNDIALS

- **CVODES** [Hindmarsh, S., 2002]
- **IDAS** [S., in development]

The SUNDIALS solvers

CVODE - ODE solver

- Variable-order, variable-step BDF (stiff) or implicit Adams (nonstiff)
- Nonlinear systems solved by Newton or functional iteration
- Linear systems solved by direct (dense or band) or iterative solvers

IDA - DAE solver

- Variable-order, variable-step BDF
- Nonlinear system solved by Newton iteration
- Linear systems solved by direct or iterative solvers

KINSOL - nonlinear solver

- Inexact Newton method
- Krylov solver: SPGMR (Scaled Preconditioned GMRES)

CVODES

- Sensitivity-capable (forward & adjoint) version of **CVODE**

IDAS

- Sensitivity-capable (forward & adjoint) version of **IDA**

Salient features of SUNDIALS solvers

- Philosophy: **Keep codes simple to use**
- Written in C
  - Fortran interfaces: **FCVODE** and **FKINSOL** (**FIDA** in development)
  - Matlab interfaces: **SUNDIALSTB** (**CVODES** and **KINSOL**)
- Written in a **data structure neutral** manner
  - No specific assumptions about data
  - Alternative data representations and operations can be provided
- Modular implementation
  - Vector modules
  - Linear solver modules
  - Preconditioner modules
- Require minimal problem information, but offer user control over most parameters

Outline

General form of an IVP

CASC

$$F(\dot{x}, x) = 0$$
$$x(t_0) = x_0$$

Definition

If $\partial F / \partial \dot{x}$ is invertible, we can formally solve for $\dot{x}$ to obtain an *ordinary differential equation* (ODE). Otherwise, we have a *differential algebraic equation* (DAE).

DAE as differential equations on manifolds (Rheinboldt, 1984)

$$\dot{x} = v(x) \, ; \quad x \in \mathcal{M}$$

Manifold: $\mathcal{M} = \{x \in R^n \mid g(x) = 0\}$

Tangent space: $T_x\mathcal{M} = \{v \in R^n \mid g_x(x)v = 0\}$

Vector field on $\mathcal{M}$: $v : \mathcal{M} \to R^n \, ; \quad \forall x \in \mathcal{M} \Rightarrow v(x) \in T_x\mathcal{M}$

DAE index

DAEs are best classified using various concepts of their *index*.

- the index of nilpotency (for linear constant coefficient DAE): measure of numerical difficulty in solving the DAE
- the differentiation index: "departure" from ODEs
- the perturbation index: measure of sensitivity of the solutions with respect to perturbations.
- ...

Definition (Gear & Petzold, 1983)

Equation $F(x, \dot{x})$ has *differentiation* index $\text{di} = m$ if m is the minimal number of analytical differentiations

$$F(\dot{x}, x) = 0, \ \frac{dF(\dot{x}, x)}{dt} = 0, \ \ldots, \ \frac{d^m F(\dot{x}, x)}{dt^m} = 0$$

such that, by algebraic manipulations, we can extract an explicit ODE $\dot{x} = \phi(x)$ (called "underlying ODE").

Hessenberg index-1

$$\dot{x} = f(x, z)$$
$$0 = g(x, z)$$

- g_z nonsingular
- Example: singular perturbation problems (e.g. chemical kinetics)

Robertson's example (1966)

$$A \xrightarrow[\text{(slow)}]{0.04} B$$
$$B + B \xrightarrow[\text{(very fast)}]{3\cdot 10^7} C + B$$
$$B + C \xrightarrow[\text{(fast)}]{10^4} A + B$$

$$\dot{y}_A = -0.04 y_A + 10^4 y_B y_C \; ; \qquad y_A(0) = 1$$
$$\dot{y}_B = 0.04 y_A - 10^4 y_B y_C - 3 \cdot 10^7 y_B^2 \; ; \qquad y_B(0) = 0$$
$$1 = y_A + y_B + y_C$$

Hessenberg index-2

$$\dot{x} = f(x, z)$$
$$0 = g(x)$$

- $g_x f_z$ nonsingular
- Example: modeling of incompressible fluid flow by Navier-Stokes

$$u_t + uu_x + vu_y + p_x - \nu(u_x x + u_y y) = 0$$
$$v_t + uv_x + vv_y + p_y - \nu(v_x x + v_y y) = 0$$
$$u_x + v_y = 0$$

with appropriate spatial discretization.

Stiff problems

Definition (Curtiss & Hirschfelder, 1952)

Stiff equations are equations where certain implicit methods, in particular BDF, perform better, usually tremendously better, than explicit ones.

- Stiffness can be defined in terms of *multiple time scales*: If the system has widely varying time scales, and the phenomena (or *solution modes*) that change on fast scales are **stable**, then the problem is stiff (Ascher & Petzold, 1998)
- Stiffness depends on
  - Jacobian eigenvalues
  - system dimension
  - accuracy requirements
  - length of simulation
  - ...
- In general, we say a problem is *stiff* on $[t_0, t_1]$, if

$$(t_1 - t_0) \min_j \Re(\lambda_j) \ll -1$$

Forward Euler vs. Backward Euler

- **Dahlquist test equation**

$$\dot{x} = \lambda x\,, \quad x_0 = 1$$

Exact solution: $y(t_n) = y_0 e^{\lambda t_n}$

- **Absolute stability requirement**

$$|y_n| \le |y_{n-1}|\,,\ n = 1, 2, \dots$$

Reason: If $\Re(\lambda) < 0$, then $|y(t_n)|$ decays exponentially. The problem is asymptotically stable, and we cannot tolerate growth in $|y(t_n)|$.

- **Region of absolute stability**

$$S = \{ z \in \mathbb{C};\ |R(z)| \le 1 \}$$

where $y_n = R(z) y_{n-1}\,,\ z = h\lambda$

- **Forward Euler**

$$y_n = y_{n-1} + h(\lambda y_{n-1}) \ \Rightarrow\ S = \{ z \in \mathbb{C};\ |z - (-1)| \le 1 \}$$

Step size restriction: if $\lambda < 0 \ \Rightarrow\ h \le \frac{2}{-\lambda}$

- **Backward Euler**

$$y_n = y_{n-1} + h(\lambda y_n) \ \Rightarrow\ S = \left\{ z \in \mathbb{C};\ |1 - z|^{-1} \le 1 \right\}$$

Step size restriction: if $\lambda < 0 \ \Rightarrow\ h > 0$

Forward Euler vs. Backward Euler

- **Dahlquist test equation**

$$\dot{x} = \lambda x \,, \quad x_0 = 1$$

  Exact solution: $y(t_n) = y_0 e^{\lambda t_n}$

- **Absolute stability requirement**

$$|y_n| \leq |y_{n-1}| \,, \; n = 1, 2, \ldots$$

  Reason: If $\Re(\lambda) < 0$, then $|y(t_n)|$ decays exponentially. The problem is asymptotically stable, and we cannot tolerate growth in $|y(t_n)|$.

- **Region of absolute stability**

$$S = \{z \in \mathbb{C}; \; |R(z)| \leq 1\}$$

  where $y_n = R(z) y_{n-1} \,, \; z = h\lambda$

- **Forward Euler**

$$y_n = y_{n-1} + h(\lambda y_{n-1}) \; \Rightarrow \; S = \{z \in \mathbb{C}; \; |z - (-1)| \leq 1\}$$

  Step size restriction: if $\lambda < 0 \; \Rightarrow \; h \leq \frac{2}{-\lambda}$

- **Backward Euler**

$$y_n = y_{n-1} + h(\lambda y_n) \; \Rightarrow \; S = \left\{z \in \mathbb{C}; \; |1 - z|^{-1} \leq 1\right\}$$

  Step size restriction: if $\lambda < 0 \; \Rightarrow \; h > 0$

Forward Euler vs. Backward Euler

- **Dahlquist test equation**

$$\dot{x} = \lambda x \,, \quad x_0 = 1$$

Exact solution: $y(t_n) = y_0 e^{\lambda t_n}$

- **Absolute stability requirement**

$$|y_n| \le |y_{n-1}| \,, \; n = 1, 2, \ldots$$

Reason: If $\Re(\lambda) < 0$, then $|y(t_n)|$ decays exponentially. The problem is asymptotically stable, and we cannot tolerate growth in $|y(t_n)|$.

- **Region of absolute stability**

$$S = \{z \in \mathbb{C}; \; |R(z)| \le 1\}$$

where $y_n = R(z) y_{n-1}$, $z = h\lambda$

- **Forward Euler**

$$y_n = y_{n-1} + h(\lambda y_{n-1}) \;\Rightarrow\; S = \{z \in \mathbb{C}; \; |z - (-1)| \le 1\}$$

Step size restriction: if $\lambda < 0 \;\Rightarrow\; h \le \frac{2}{-\lambda}$

- **Backward Euler**

$$y_n = y_{n-1} + h(\lambda y_n) \;\Rightarrow\; S = \left\{z \in \mathbb{C}; \; |1 - z|^{-1} \le 1\right\}$$

Step size restriction: if $\lambda < 0 \;\Rightarrow\; h > 0$

Forward Euler vs. Backward Euler

- **Dahlquist test equation**

$$\dot{x} = \lambda x \,, \quad x_0 = 1$$

Exact solution: $y(t_n) = y_0 e^{\lambda t_n}$

- **Absolute stability requirement**

$$|y_n| \leq |y_{n-1}| \,, \; n = 1, 2, \ldots$$

Reason: If $\Re(\lambda) < 0$, then $|y(t_n)|$ decays exponentially. The problem is asymptotically stable, and we cannot tolerate growth in $|y(t_n)|$.

- **Region of absolute stability**

$$S = \{z \in \mathbb{C}; \; |R(z)| \leq 1\}$$

where $y_n = R(z) y_{n-1}$, $z = h\lambda$

- **Forward Euler**

$$y_n = y_{n-1} + h\left(\lambda y_{n-1}\right) \; \Rightarrow \; S = \{z \in \mathbb{C}; \; |z - (-1)| \leq 1\}$$

Step size restriction: if $\lambda < 0 \; \Rightarrow \; h \leq \frac{2}{-\lambda}$

- **Backward Euler**

$$y_n = y_{n-1} + h\left(\lambda y_n\right) \; \Rightarrow \; S = \left\{z \in \mathbb{C}; \; |1 - z|^{-1} \leq 1\right\}$$

Step size restriction: if $\lambda < 0 \; \Rightarrow h > 0$

Forward Euler vs. Backward Euler

- **Dahlquist test equation**

$$\dot{x} = \lambda x \,, \quad x_0 = 1$$

Exact solution: $y(t_n) = y_0 e^{\lambda t_n}$

- **Absolute stability requirement**

$$|y_n| \leq |y_{n-1}| \,, \; n = 1, 2, \ldots$$

Reason: If $\Re(\lambda) < 0$, then $|y(t_n)|$ decays exponentially. The problem is asymptotically stable, and we cannot tolerate growth in $|y(t_n)|$.

- **Region of absolute stability**

$$S = \{ z \in \mathbb{C}; \; |R(z)| \leq 1 \}$$

where $y_n = R(z) y_{n-1} \,, \; z = h\lambda$

- **Forward Euler**

$$y_n = y_{n-1} + h\left(\lambda y_{n-1}\right) \; \Rightarrow \; S = \{ z \in \mathbb{C}; \; |z - (-1)| \leq 1 \}$$

Step size restriction: if $\lambda < 0 \; \Rightarrow \; h \leq \frac{2}{-\lambda}$

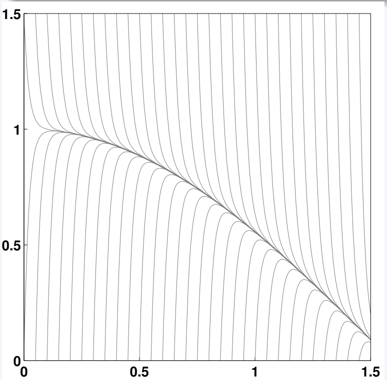
- **Backward Euler**

$$y_n = y_{n-1} + h\left(\lambda y_n\right) \; \Rightarrow \; S = \left\{ z \in \mathbb{C}; \; |1 - z|^{-1} \leq 1 \right\}$$

Step size restriction: if $\lambda < 0 \; \Rightarrow \; h > 0$

Curtiss & Hirschfelder example

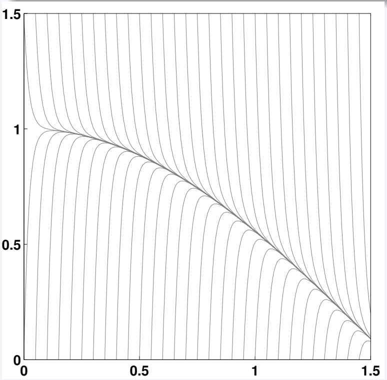
$$\dot{x} = -50\,(x - \cos(t))$$



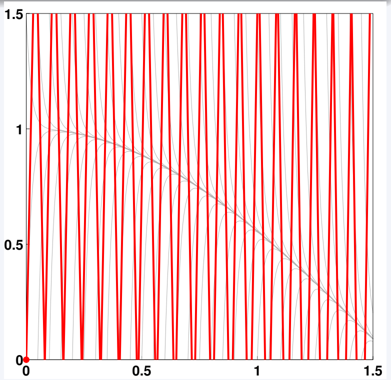
Solution curves

Curtiss & Hirschfelder example

$$\dot{x} = -50\,(x - \cos(t))$$



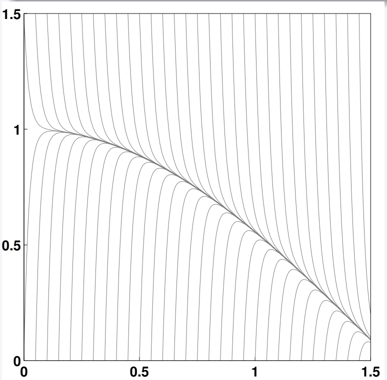
Solution curves



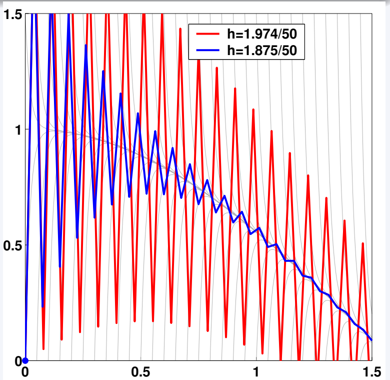
Forward Euler ($h = 2.01/50$)

Curtiss & Hirschfelder example

$$\dot{x} = -50\,(x - \cos(t))$$



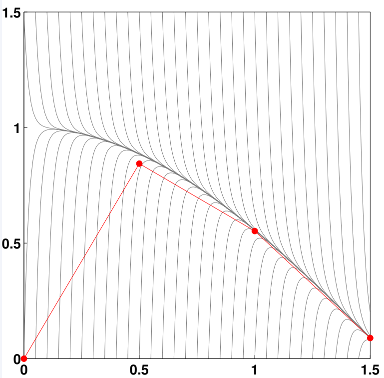
Solution curves



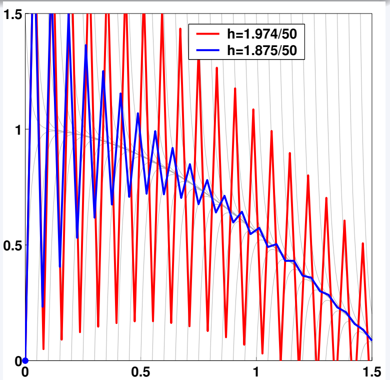
Forward Euler

Curtiss & Hirschfelder example

$$\dot{x} = -50\,(x - \cos(t))$$



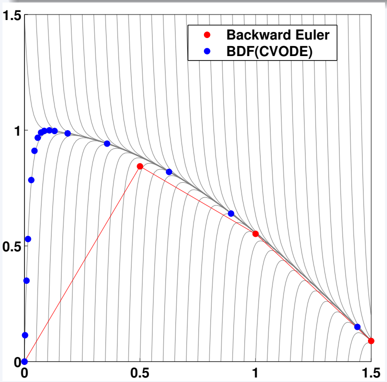
Backward Euler (3 steps)



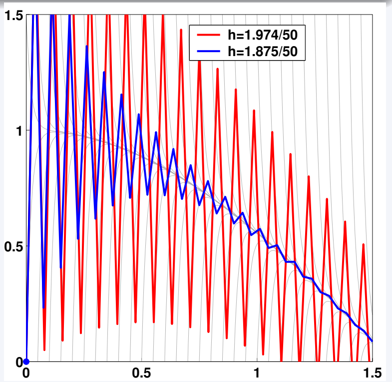
Forward Euler

Curtiss & Hirschfelder example

$$\dot{x} = -50\,(x - \cos(t))$$



CVODE solution



Forward Euler

Linear multistep methods

General form

$$\sum_{i=0}^{K_1} \alpha_{n,i} x_{n-i} + h_n \sum_{i=0}^{K_2} \beta_{n,i} \dot{x}_{n-i} = 0$$

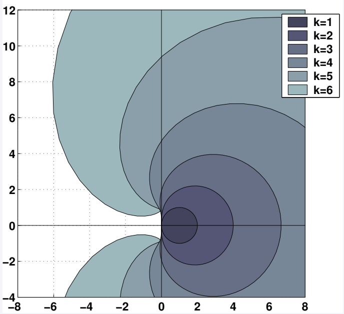
Two particular methods

- Adams-Moulton (nonstiff)
  $K_1 = 1, K_2 = k, k = 1, \ldots, 12$
- BDF (stiff)
  $K_1 = k, K_2 = 0, k = 1, \ldots, 5$

Nonlinear system (BDF)

- ODE: $\dot{x} = f(x)$
  $G(x_n) \equiv x_n - \beta_0 h_n f(x_n) - \sum_{i>0} \alpha_{n,i} x_{n-i} = 0$
- DAE: $F(\dot{x}, x) = 0$
  $G(x_n) \equiv F\left((\beta_0 h_n)^{-1} \sum_{i \geq 0} \alpha_{n,i} x_{n-i}, x_n \right) = 0$

BDF: $x_n - \beta_0 h_n \dot{x}_n = \sum_{i=1}^{k} \alpha_{n,i} x_{n-i}$



Absolute stability regions

LMM: variable-order, variable-step BDF

- **Fixed-leading coefficient form of BDF formulas**
- **Predictor-corrector implementation**
  - Predictor $x_{n(0)} = \sum_{i=1}^{k} \alpha_i^p x_{n-i} + \beta_0^p h_n \dot{x}_{n-1}$
  - Corrector $x_n = \sum_{i=1}^{k} \alpha_i x_{n-i} + \beta_0 h_n f(x_n)$
- **Use weighted residual mean square norms**

  $||x||_{\text{wrms}} := \sqrt{(x_i w_i)^2 / N} \quad w_i = \frac{1}{\text{rtol}|x_i| + \text{atol}_i}$
- **Error control mechanism**
  - Step size selection
    1. Estimate error: $E(h_n) = C \cdot (x_n - x_{n(0)})$
    2. Accept step if $||E(h_n)||_{\text{wrms}} < 1.0$
    3. Estimate error at next step $E(h'_n) \approx (h'_n / h_n)^k \, E(h_n)$
    4. Select h'_n such that $||E(h'_n)||_{\text{wrms}} < 1.0$
  - Method order selection
    1. Estimate errors for next higher and lower orders
    2. Select the order that gives the largest step size meeting the error condition

LMM: nonlinear system solution

CASC

- Use predicted value $x_n(0)$ as initial guess for the nonlinear iteration
- **Nonstiff systems: Functional iteration**

$$x_{n(m+1)} = \beta_0 h_n f\left(x_{n(m)}\right)$$

- **Stiff systems: Newton iteration**

$$M\left(x_{n(m+1)} - x_{n(m)}\right) = -G\left(x_{n(m)}\right)$$

  - ODE:
    $M \approx I - \partial f / \partial x$, $\gamma = \beta_0 h_n$
  - DAE:
    $M \approx \partial F / \partial y + \gamma \partial F / \partial \dot{x}$, $\gamma = 1/(\beta_0 h_n)$

LMM: linear system solution

- Direct dense
- Direct band
- Direct sparse
- Iterative linear solvers
  - Result in Inexact Newton nonlinear solver
  - Scaled preconditioned solvers: GMRES, Bi-CGStab, TFQMR
  - Only require matrix-vector products
  - Require preconditioner for the Newton matrix M
- Jacobian information (matrix or matrix-vector product) can be supplied by the user or estimated by difference quotients

Outline

1. SUNDIALS: overview
2. ODE and DAE integration
   - Initial value problems
   - Implicit integration methods
3. **Nonlinear systems**
   - Newton's method
   - Inexact Newton
   - Preconditioning
4. Sensitivity analysis
   - Definitions, applications, methods
   - Forward sensitivity analysis
   - Adjoint sensitivity analysis
5. SUNDIALS: usage, applications, availability
   - Usage
   - Applications
   - Availability

Basic method

CASC

$$F(x) = 0$$

x^0 : starting point

- Basis for most nonlinear solvers: Newton's method

$$J(x^k)\Delta x_k = -F(x^k) \quad \text{where } J(x) = F_x(x)$$
$$x^{k+1} = x^k + \Delta x_k$$

- Convergences if x^0 is *close enough* to x^* and $\exists J^{-1}(x^*)$
- Quadratic convergence: $\|x^{k+1} - x^*\| \leq C\|x^k - x^*\|$, for some $C > 0$

Modifications and enhancements

Two main problems with Newton's method:

- Need to calculate the Jacobian matrix
  - Matrix-free linear solvers
  - Multi-secant methods (Broyden)
    Use successive approximations B_k to the Jacobian matrix $J(x^k)$
- No guaranteed global convergence
  - Line search with backtracking
    Use only a fraction of the full Newton step: $x^{k+1} = x^k + \lambda \Delta x_k$
    Select λ to obtain
    - sufficient decrease in F relative to the step length
    - a minimum step length relative to the initial rate of decrease
    - full Newton step close to x^*.
  - Trust region methods

KINSOL provides *matrix-free linear solvers* and *line search with backtracking* capabilities.

Modifications and enhancements

Two main problems with Newton's method:

- Need to calculate the Jacobian matrix
  - Matrix-free linear solvers
  - Multi-secant methods (Broyden)
    Use successive approximations B_k to the Jacobian matrix $J(x^k)$
- No guaranteed global convergence
  - Line search with backtracking
    Use only a fraction of the full Newton step: $x^{k+1} = x^k + \lambda \Delta x_k$
    Select λ to obtain
    - sufficient decrease in F relative to the step length
    - a minimum step length relative to the initial rate of decrease
    - full Newton step close to x^*.
  - Trust region methods

KINSOL provides *matrix-free linear solvers* and *line search with backtracking* capabilities.

Inexact Newton

Solve linear systems *approximately*

$$F_x(x^k)\Delta x_k \approx -F(x^k)$$

$$\text{such that } \|F(x^k) + F_x(x^k)\Delta x_k\| \le \eta_k \|F(x^k)\|$$

$$x^{k+1} = x^k + \Delta x_k$$

Stopping tolerance η_k is selected to prevent "over-solves"

- Newton's method is based on a linear model
  - Bad approximation far from solution $\Rightarrow$ loose tolerances
  - Good approximation close to solution $\Rightarrow$ tight tolerances
- Eisenstat and Walker
  - Choice 1 $\quad \eta_k = \|F(x^k)\| - \|F(x^{k-1}) + F_x(x^{k-1})\Delta x_{k-1}\| / \|F(x^{k-1})\|$
  - Choice 2 $\quad \eta_k = 0.9\left(\|F(x^k)\| / \|F(x^{k-1})\|\right)^2$
- Constant value
  - Kelley $\quad \eta_k = 0.1$
  - ODE literature $\quad \eta_k = 0.05$

Preconditioned Krylov solver

Linear system within the Newton iteration: $Js = r$

- Krylov iterative methods find the solution in the subspace $K(J, r) = \{r, Jr, J^2r, \ldots\}$
- Their convergence rate depends on the spectral properties of J
- Preconditioning: replace the linear system with an equivalent one that has more favorable spectral properties
- Preconditioning on the right: $(JP^{-1})(Ps) = r$
- The preconditioner P must approximate the Jacobian matrix, yet be reasonably cheap to evaluate and efficient to solve
  - setup phase: evaluate and preprocess P (infrequent)
  - solve phase: solve systems $Px = b$ (frequent)
- Many preconditioner types
  - Jacobi preconditioner
  - Incomplete factorization preconditioners
  - Block preconditioners
  - Preconditioners based on the underlying problem

Outline

Definitions

Definition

Broadly speaking, *sensitivity analysis* (SA) is the study of how the variation in the output of a model (numerical or otherwise) can be apportioned, qualitatively or quantitatively, to different sources of variation.

First-order SA problem (dynamical systems)

$$F(\dot{x}, x, p) = 0$$
$$y(p) = \mathcal{O}(x, p)$$

where $x \in R^n$ and $y \in R$ ($\mathcal{O} : R^n \times R^{N_p} \to R$).
Considering the Taylor expansion of y around the nominal value p

$$y(p + \delta p) = y(p) + \nabla_p y(p) \cdot \delta p + O(\delta p^2)$$

we define the first-order SA problem as the problem of computing the gradient $\nabla_p y$.

Definitions

CASC

Definition

Broadly speaking, *sensitivity analysis* (SA) is the study of how the variation in the output of a model (numerical or otherwise) can be apportioned, qualitatively or quantitatively, to different sources of variation.

First-order SA problem (dynamical systems)

$$F(\dot{x}, x, p) = 0$$
$$y(p) = \mathcal{O}(x, p)$$

where $x \in R^n$ and $y \in R$ ($\mathcal{O} : R^n \times R^{N_p} \to R$).
Considering the Taylor expansion of y around the nominal value p

$$y(p + \delta p) = y(p) + \nabla_p y(p) \cdot \delta p + O(\delta p^2)$$

we define the first-order SA problem as the problem of computing the gradient $\nabla_p y$.

Applications of SA

- **Model evaluation**
  Finding most and least influential parameters
- **Model reduction**
  Reducing model complexity, while preserving its input-output behavior
- **Data assimilation**
  Merging observed information into a model in order to improve its accuracy
- **Uncertainty quantification**
  Characterizing (quantitatively) and reducing uncertainty in model predictions
- **Dynamically-constrained optimization**
  Improving model response (better performance, better agreement with observations, etc.)

Applications of SA

- **Model evaluation**
  Finding most and least influential parameters
- **Model reduction**
  Reducing model complexity, while preserving its input-output behavior
- **Data assimilation**
  Merging observed information into a model in order to improve its accuracy
- **Uncertainty quantification**
  Characterizing (quantitatively) and reducing uncertainty in model predictions
- **Dynamically-constrained optimization**
  Improving model response (better performance, better agreement with observations, etc.)

Applications of SA

- **Model evaluation**
  Finding most and least influential parameters
- **Model reduction**
  Reducing model complexity, while preserving its input-output behavior
- **Data assimilation**
  Merging observed information into a model in order to improve its accuracy
- **Uncertainty quantification**
  Characterizing (quantitatively) and reducing uncertainty in model predictions
- **Dynamically-constrained optimization**
  Improving model response (better performance, better agreement with observations, etc.)

Applications of SA

- **Model evaluation**
  Finding most and least influential parameters
- **Model reduction**
  Reducing model complexity, while preserving its input-output behavior
- **Data assimilation**
  Merging observed information into a model in order to improve its accuracy
- **Uncertainty quantification**
  Characterizing (quantitatively) and reducing uncertainty in model predictions
- **Dynamically-constrained optimization**
  Improving model response (better performance, better agreement with observations, etc.)

Applications of SA

- **Model evaluation**
  Finding most and least influential parameters
- **Model reduction**
  Reducing model complexity, while preserving its input-output behavior
- **Data assimilation**
  Merging observed information into a model in order to improve its accuracy
- **Uncertainty quantification**
  Characterizing (quantitatively) and reducing uncertainty in model predictions
- **Dynamically-constrained optimization**
  Improving model response (better performance, better agreement with observations, etc.)

Methods

Parameter-dependent ODE system

Model: $F(\dot{x}, x, p) = 0$

Output functional: $y(p) = \mathcal{O}(x, p)$

Methods

Parameter-dependent ODE system

Model: $F(\dot{x}, x, p) = 0$

Output functional: $y(p) = \mathcal{O}(x, p)$

DSA - discrete sensitivity analysis

$$\frac{dy}{dp_i}(p) \approx \frac{y(p + e_i \delta p_i) - y(p)}{\delta p_i}$$

or

$$\frac{dy}{dp_i}(p) \approx \frac{y(p + e_i \delta p_i) - y(p - e_i \delta p_i)}{2\delta p_i}$$

e_i is the i-th column of the identity matrix and δp is a vector of perturbations.

Methods

Parameter-dependent ODE system

Model: $F(\dot{x}, x, p) = 0$

Output functional: $y(p) = \mathcal{O}(x, p)$

FSA

$$F_{\dot{x}} \dot{s}_i + F_x s_i + F_{p_i} = 0$$

and

$$\nabla_p y(p) = [\cdots, \mathcal{O}_x s_i + \mathcal{O}_{p_i}, \cdots]$$

Cost: $(1 + N_p) \times \mathrm{cost}(\mathcal{M})$

ASA

$$(\lambda^* F_{\dot{x}})' - \lambda^* F_x = -\mathcal{O}_x^* 1$$

and

$$\nabla_p y(p) = \langle F_p, \lambda \rangle + \mathcal{O}_p$$

Cost: $(1 + N_y) \times \mathrm{cost}(\mathcal{M})$

Methods

CASC

Parameter-dependent ODE system

Model: $F(\dot{x}, x, p) = 0$

Output functional: $y(p) = \mathcal{O}(x, p)$

FSA

$$F_{\dot{x}} \dot{s}_i + F_x s_i + F_{p_i} = 0$$

and

$$\nabla_p y(p) = [\cdots, \mathcal{O}_x s_i + \mathcal{O}_{p_i}, \cdots]$$

Cost: $(1 + N_p) \times \text{cost}(\mathcal{M})$

ASA

$$(\lambda^* F_{\dot{x}})' - \lambda^* F_x = -\mathcal{O}_x^* \mathbf{1}$$

and

$$\nabla_p y(p) = \langle F_p, \lambda \rangle + \mathcal{O}_p$$

Cost: $(1 + N_y) \times \text{cost}(\mathcal{M})$

FSA for ODE and DAE systems

- Parameter dependent system: $F(\dot{x}, x, p) = 0\,, \quad x(t_0) = x_0(p)$
- Output functional: $g(x, p)$
- Sensitivity systems: $(i = 1, 2, \ldots, N_p)$

$$F_{\dot{x}} \dot{s}_i + F_x s_i + F_{p_i} = 0\,, \quad s_i(t_0) = x_{0p_i}$$

- Gradient of output functional:

$$\nabla_p g = g_x s + g_p$$

where $s = [s_1, s_2, \ldots, s_{N_p}]$ is the *sensitivity matrix*

Good: Sensitivity system does *not* depend on $\mathcal{O}$
Bad: Sensitivity system depends on p

FSA for ODE and DAE systems

- Parameter dependent system: $F(\dot{x}, x, p) = 0\,, \quad x(t_0) = x_0(p)$
- Output functional: $g(x, p)$
- Sensitivity systems: $(i = 1, 2, \dots, N_p)$

$$F_{\dot{x}} \dot{s}_i + F_x s_i + F_{p_i} = 0\,, \quad s_i(t_0) = x_{0p_i}$$

- Gradient of output functional:

$$\nabla_p g = g_x s + g_p$$

where $s = [s_1, s_2, \dots, s_{N_p}]$ is the *sensitivity matrix*

Good: Sensitivity system does *not* depend on $\mathcal{O}$
Bad: Sensitivity system depends on p

FSA: generation of the sensitivity system

$$\dot{x} = f(x,p) \quad \Rightarrow \quad \dot{s}_i = f_x s_i + f_{p_i}$$

- Analytical
- AD (ADIFOR, ADIC, ADOLC, ...)
- Directional derivative approximations

$$\begin{cases} f_x s_i \approx \frac{f(t,x+\sigma_x s_i,p) - f(t,x-\sigma_x s_i,p)}{2\sigma_x} \\ f_{p_i} \approx \frac{f(t,x,p+\sigma_i e_i p) - f(t,x,p-\sigma_i e_i)}{2\sigma_i} \end{cases} \qquad \begin{cases} \sigma_i = |\bar{p}_i| \sqrt{\max(rtol,\epsilon)} \\ \sigma_x = \frac{1}{\max(1/\sigma_i, \|s_i\|_{WRMS}/|\bar{p}_i|)} \end{cases}$$

or

$$f_x s_i + f_{p_i} \approx \frac{f(t, x+\sigma s_i, p+\sigma e_i p) - f(t, x-\sigma s_i, p-\sigma e_i)}{2\sigma}$$

where $\sigma = \min(\sigma_i, \sigma_x)$

FSA: implementation issues

Must take advantage of the shared structure with original system

Solutions (for implicit ODE/DAE integrators)

- **Staggered Direct** (Caracotsios & Stewart, 1985):
  iterate to convergence the nonlinear state system and then solve the linear sensitivity systems
  requires formation and storage of J; errors in J → errors in s
- **Simultaneous Corrector** (Maly & Petzold, 1997):
  solve simultaneously a nonlinear system for both states and sensitivity variables
  requires formation of sensitivity r.h.s. at every iteration
- **Staggered Corrector** (Feehery, Tolsma, Barton, 1997):
  iterate to convergence the nonlinear state system and then use a Newton method to solve for the sensitivity variables
  with iterative linear solvers → effectively Staggered Direct

CVODES and **IDAS** implement the *simultaneous corrector* and two flavors of the *staggered corrector* approaches.

FSA: implementation issues

Must take advantage of the shared structure with original system

Solutions (for implicit ODE/DAE integrators)

- **Staggered Direct** (Caracotsios & Stewart, 1985):
  iterate to convergence the nonlinear state system and then solve the linear sensitivity systems
  requires formation and storage of J; errors in $J \rightarrow$ errors in s
- **Simultaneous Corrector** (Maly & Petzold, 1997):
  solve simultaneously a nonlinear system for both states and sensitivity variables
  requires formation of sensitivity r.h.s. at every iteration
- **Staggered Corrector** (Feehery, Tolsma, Barton, 1997):
  iterate to convergence the nonlinear state system and then use a Newton method to solve for the sensitivity variables
  with iterative linear solvers $\rightarrow$ effectively Staggered Direct

CVODES and IDAS implement the *simultaneous corrector* and two flavors of the *staggered corrector* approaches.

FSA: implementation issues

Must take advantage of the shared structure with original system

Solutions (for implicit ODE/DAE integrators)

- **Staggered Direct** (Caracotsios & Stewart, 1985):
  iterate to convergence the nonlinear state system and then solve the linear sensitivity systems
  requires formation and storage of J; errors in $J \to$ errors in s
- **Simultaneous Corrector** (Maly & Petzold, 1997):
  solve simultaneously a nonlinear system for both states and sensitivity variables
  requires formation of sensitivity r.h.s. at every iteration
- **Staggered Corrector** (Feehery, Tolsma, Barton, 1997):
  iterate to convergence the nonlinear state system and then use a Newton method to solve for the sensitivity variables
  with iterative linear solvers $\to$ effectively Staggered Direct

CVODES and IDAS implement the *simultaneous corrector* and two flavors of the *staggered corrector* approaches.

FSA: implementation issues

CASC

Must take advantage of the shared structure with original system

Solutions (for implicit ODE/DAE integrators)

- **Staggered Direct** (Caracotsios & Stewart, 1985):
  iterate to convergence the nonlinear state system and then solve the linear sensitivity systems
  requires formation and storage of J; errors in $J \rightarrow$ errors in s
- **Simultaneous Corrector** (Maly & Petzold, 1997):
  solve simultaneously a nonlinear system for both states and sensitivity variables
  requires formation of sensitivity r.h.s. at every iteration
- **Staggered Corrector** (Feehery, Tolsma, Barton, 1997):
  iterate to convergence the nonlinear state system and then use a Newton method to solve for the sensitivity variables
  with iterative linear solvers $\rightarrow$ effectively Staggered Direct

CVODES and IDAS implement the *simultaneous corrector* and two flavors of the *staggered corrector* approaches.

FSA: implementation issues

Must take advantage of the shared structure with original system

Solutions (for implicit ODE/DAE integrators)

- **Staggered Direct** (Caracotsios & Stewart, 1985):
  iterate to convergence the nonlinear state system and then solve the linear sensitivity systems
  requires formation and storage of J; errors in $J \rightarrow$ errors in s
- **Simultaneous Corrector** (Maly & Petzold, 1997):
  solve simultaneously a nonlinear system for both states and sensitivity variables
  requires formation of sensitivity r.h.s. at every iteration
- **Staggered Corrector** (Feehery, Tolsma, Barton, 1997):
  iterate to convergence the nonlinear state system and then use a Newton method to solve for the sensitivity variables
  with iterative linear solvers $\rightarrow$ effectively Staggered Direct

CVODES and **IDAS** implement the *simultaneous corrector* and two flavors of the *staggered corrector* approaches.

Speedup results for FSA

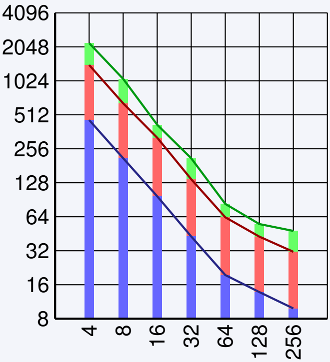
CASC

Problem 2-species 2D diurnal kinetics advection-diffusion PDE system.

Dimension $N = 2 \cdot (p_x n_x) \cdot (p_z n_z) = 2 \cdot 1600 \cdot 400 = 1280000$

Platform Parallel performance tests were performed on ASCI Frost, a 68-node, 16-way SMP system with POWER3 375 MHz processors and 16 GB of memory per node.

P ($p_x p_z$)	CPU time (s): States only	CPU time (s): Staggered partial	CPU time (s): Staggered full
4	460.31	1414.53	2208.14
8	211.20	646.59	1064.94
16	97.16	320.78	417.95
32	42.78	137.51	210.84
64	19.50	63.34	83.24
128	13.78	42.71	55.17
256	9.87	31.33	47.95



ASA - ODE derivation

- Parameter dependent system: $F(\dot{x}, x, p) = 0\,, \quad x(t_0) = x_0(p)$
- Output functional: $G(p) = \int_{t_0}^{t_f} g(x,p)\, dt$

  $\nabla_p G = \int_{t_0}^{t_f} (g_x s + g_p)\, dt + \int_{t_0}^{t_f} \lambda^* \left(F_{\dot{x}} \dot{s} + F_x s + F_p\right)\, dt$

  $= \int_{t_0}^{t_f} (g_x + (\lambda^* F_{\dot{x}})' - \lambda^* F_x)\, s\, dt + \int_{t_0}^{t_f} (g_p - \lambda^* F_p)\, dt - (\lambda^* F_{\dot{x}} s)|_{t_0}^{t_f}$
- Adjoint system:

$$(\lambda^* F_{\dot{x}})' - \lambda^* F_x + g_x = 0\,, \quad (\lambda^* F_{\dot{x}})|_{t_f} = ?$$

- Gradient of output functional:

$$\nabla_p G = \int_{t_0}^{t_f} (g_p - \lambda^* F_p)\, dt - (\lambda^* F_{\dot{x}} s)_{t=t_f} + (\lambda^* F_{\dot{x}})_{t=t_0} x_{0p}$$

Good: Sensitivity system does *not* depend on p
Bad: Sensitivity system depends on $\mathcal{O}$

ASA - ODE derivation

- Parameter dependent system: $F(\dot{x}, x, p) = 0\,, \quad x(t_0) = x_0(p)$
- Output functional: $G(p) = \int_{t_0}^{t_f} g(x,p)\, dt$

  $\nabla_p G = \int_{t_0}^{t_f} (g_x s + g_p)\, dt + \int_{t_0}^{t_f} \lambda^* \left(F_{\dot{x}} \dot{s} + F_x s + F_p \right)\, dt$

  $= \int_{t_0}^{t_f} (g_x + (\lambda^* F_{\dot{x}})' - \lambda^* F_x)\, s\, dt + \int_{t_0}^{t_f} (g_p - \lambda^* F_p)\, dt - (\lambda^* F_{\dot{x}} s)|_{t_0}^{t_f}$
- Adjoint system:

$$(\lambda^* F_{\dot{x}})' - \lambda^* F_x + g_x = 0\,, \quad (\lambda^* F_{\dot{x}})|_{t_f} = ?$$

- Gradient of output functional:

$$\nabla_p G = \int_{t_0}^{t_f} (g_p - \lambda^* F_p)\, dt - (\lambda^* F_{\dot{x}} s)_{t=t_f} + (\lambda^* F_{\dot{x}})_{t=t_0} x_{0p}$$

Good: Sensitivity system does *not* depend on p

Bad: Sensitivity system depends on $\mathcal{O}$

ASA for ODE and DAE systems

Model: $F(\dot{x}, x, p) = 0\,, \quad x(t_0) = x_0(p)$

Output functional: $G(p) = \int_{t_0}^{t_f} g(x, p)\, dt$

Gradient: $\nabla_p G = \int_{t_0}^{t_f} (g_p - \lambda^* F_p)\, dt - (\lambda^* F_{\dot{x}} x_p)\big|_{t_0}^{t_f}$

Adjoint system: $(\lambda^* F_{\dot{x}})' - \lambda^* F_x = -g_x\,, \quad \lambda^* F_{\dot{x}}\big|_{t_f} = ?$

ASA for ODE and DAE systems

Model: $F(\dot{x}, x, p) = 0\,, \quad x(t_0) = x_0(p)$

Output functional: $G(p) = \int_{t_0}^{t_f} g(x,p)\, dt$

Gradient: $\nabla_p G = \int_{t_0}^{t_f} (g_p - \lambda^* F_p)\, dt - (\lambda^* F_{\dot{x}} x_p)\big|_{t_0}^{t_f}$

Adjoint system: $(\lambda^* F_{\dot{x}})' - \lambda^* F_x = -g_x\,, \quad \lambda^* F_{\dot{x}}|_{t_f} = ?$

index-0 and index-1 DAE

$$F(\dot{x}, x) = 0 \Rightarrow (A^* \lambda)' - B^* \lambda = 0$$

$A = \partial F / \partial \dot{x}$ nonsingular, $B = \partial F / \partial x$
Can use

$$(\lambda^* A)_{t=t_f} = 0$$

and therefore

$$\nabla_p G = \int_{t_0}^{t_f} (g_p - \lambda^* F_p)\; dt + (\lambda^* F_{\dot{x}})_{t=t_0}\, x_{0p}$$

ASA for ODE and DAE systems

Model: $F(\dot{x}, x, p) = 0\,, \quad x(t_0) = x_0(p)$

Output functional: $G(p) = \int_{t_0}^{t_f} g(x,p)\, dt$

Gradient: $\nabla_p G = \int_{t_0}^{t_f} (g_p - \lambda^* F_p)\, dt - (\lambda^* F_{\dot{x}} x_p)\big|_{t_0}^{t_f}$

Adjoint system: $(\lambda^* F_{\dot{x}})' - \lambda^* F_x = -g_x\,, \quad \lambda^* F_{\dot{x}}\big|_{t_f} = ?$

Hessenberg index-2 DAE

$$\begin{aligned} \dot{x}^d &= f^d(x^d, x^a, p) \\ 0 &= f^a(x^d, p) \end{aligned} \quad \Rightarrow \quad \begin{aligned} \dot{\lambda}^d &= -A^* \lambda^d - C^* \lambda^a - g^*_{x^d} \\ 0 &= -B^* \lambda^d - g^*_{x^a} \end{aligned}$$

Search for final conditions of the form $\lambda^d(t_f) = (C^* \xi)_{t=t_f}$

$$t = t_f \Rightarrow \begin{cases} \lambda^{d*} B = -g_{x^a} \Rightarrow \xi^* CB = -g_{x^a} \Rightarrow \xi^* = -g_{x^a}(CB)^{-1} \\ f^a(x^d, p) = 0 \rightarrow Cx^d_p = -f^a_p \Rightarrow \lambda^{d*} x^d_p = -xi^* f^a_p \end{cases}$$

$$\Rightarrow \lambda^{d*}(t_f) = -\left(g_{x^a}(CB)^{-1} C\right)_{t=t_f}$$

$$\Rightarrow \nabla_p G = \int_{t_0}^{t_f} \left(g_p + \lambda^{d*} f^d_p + \lambda^{a*} f^a_p\right) dt + \lambda^{d*}(t_0) x^d_{0p} + \left(g_{x^a}(CB)^{-1} f^a_p\right)_{t=t_f}$$

ASA implementation issues

CASC

Problem

Solution of the forward problem is needed in the backward integration phase ⇒ need predictable and compact storage of state variables for the solution of the adjoint system.

Solution: checkpointing

- Simulations are reproducible from each checkpoint
- Force Jacobian evaluation at checkpoints to avoid storing it
- Store solution (and possibly first derivative) at all intermediate steps between two consecutive checkpoints
- Interpolation options: cubic Hermite, variable-order polynomial

ASA implementation issues

Problem

Solution of the forward problem is needed in the backward integration phase $\Rightarrow$ need predictable and compact storage of state variables for the solution of the adjoint system.

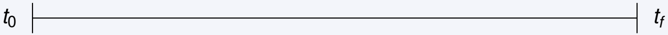
Solution: checkpointing

- Simulations are reproducible from each checkpoint
- Force Jacobian evaluation at checkpoints to avoid storing it
- Store solution (and possibly first derivative) at all intermediate steps between two consecutive checkpoints
- Interpolation options: cubic Hermite, variable-order polynomial

Checkpointing

Implementation

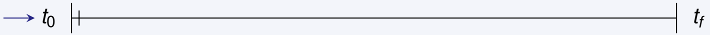
1. integrate forward step by step
2. dump checkpoint data after a given number of steps
3. continue until t_f.
4. evaluate final conditions for adjoint problem
5. store interpolation data on second forward pass
6. propagate adjoint variables backward in time
7. total cost: 2 forward passes + 1 backward pass



Checkpointing

Implementation

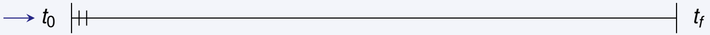
1. integrate forward step by step
2. dump checkpoint data after a given number of steps
3. continue until t_f.
4. evaluate final conditions for adjoint problem
5. store interpolation data on second forward pass
6. propagate adjoint variables backward in time
7. total cost: 2 forward passes + 1 backward pass



Checkpointing

Implementation

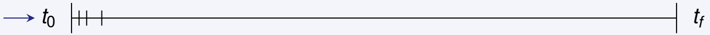
1. integrate forward step by step
2. dump checkpoint data after a given number of steps
3. continue until t_f.
4. evaluate final conditions for adjoint problem
5. store interpolation data on second forward pass
6. propagate adjoint variables backward in time
7. total cost: 2 forward passes + 1 backward pass



Checkpointing

Implementation

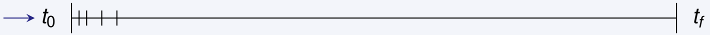
1. integrate forward step by step
2. dump checkpoint data after a given number of steps
3. continue until t_f.
4. evaluate final conditions for adjoint problem
5. store interpolation data on second forward pass
6. propagate adjoint variables backward in time
7. total cost: 2 forward passes + 1 backward pass



Checkpointing

Implementation

1. integrate forward step by step
2. dump checkpoint data after a given number of steps
3. continue until t_f.
4. evaluate final conditions for adjoint problem
5. store interpolation data on second forward pass
6. propagate adjoint variables backward in time
7. total cost: 2 forward passes + 1 backward pass



Checkpointing

Implementation

1. integrate forward step by step
2. dump checkpoint data after a given number of steps
3. continue until t_f.
4. evaluate final conditions for adjoint problem
5. store interpolation data on second forward pass
6. propagate adjoint variables backward in time
7. total cost: 2 forward passes + 1 backward pass

Checkpointing

Implementation

1. integrate forward step by step
2. dump checkpoint data after a given number of steps
3. continue until t_f.
4. evaluate final conditions for adjoint problem
5. store interpolation data on second forward pass
6. propagate adjoint variables backward in time
7. total cost: 2 forward passes + 1 backward pass

Checkpointing

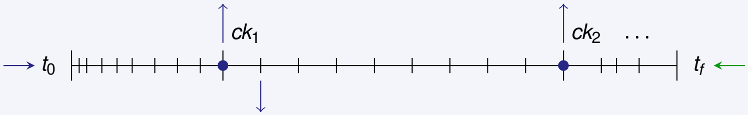
Implementation

1. integrate forward step by step
2. dump checkpoint data after a given number of steps
3. continue until t_f.
4. evaluate final conditions for adjoint problem
5. store interpolation data on second forward pass
6. propagate adjoint variables backward in time
7. total cost: 2 forward passes + 1 backward pass

Checkpointing

Implementation

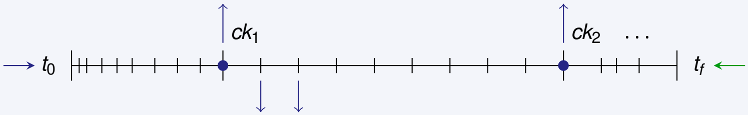
1. integrate forward step by step
2. dump checkpoint data after a given number of steps
3. continue until t_f.
4. evaluate final conditions for adjoint problem
5. store interpolation data on second forward pass
6. propagate adjoint variables backward in time
7. total cost: 2 forward passes + 1 backward pass



Checkpointing

Implementation

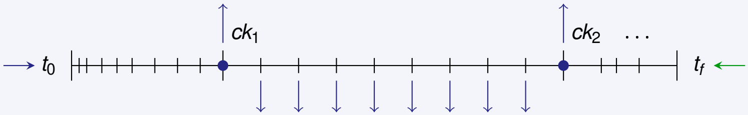
1. integrate forward step by step
2. dump checkpoint data after a given number of steps
3. continue until t_f.
4. evaluate final conditions for adjoint problem
5. store interpolation data on second forward pass
6. propagate adjoint variables backward in time
7. total cost: 2 forward passes + 1 backward pass



Checkpointing

Implementation

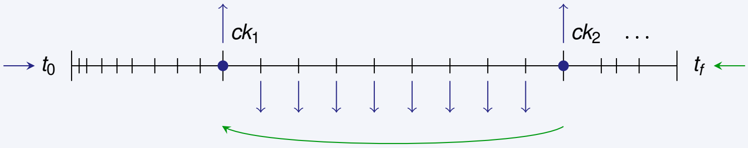
1. integrate forward step by step
2. dump checkpoint data after a given number of steps
3. continue until t_f.
4. evaluate final conditions for adjoint problem
5. store interpolation data on second forward pass
6. propagate adjoint variables backward in time
7. total cost: 2 forward passes + 1 backward pass



Checkpointing

Implementation

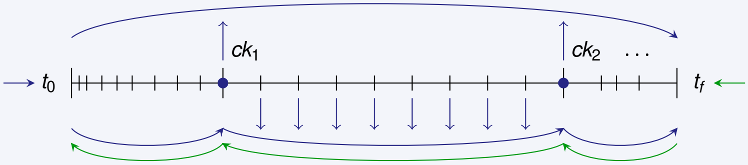
1. integrate forward step by step
2. dump checkpoint data after a given number of steps
3. continue until t_f.
4. evaluate final conditions for adjoint problem
5. store interpolation data on second forward pass
6. propagate adjoint variables backward in time
7. total cost: 2 forward passes + 1 backward pass



Checkpointing

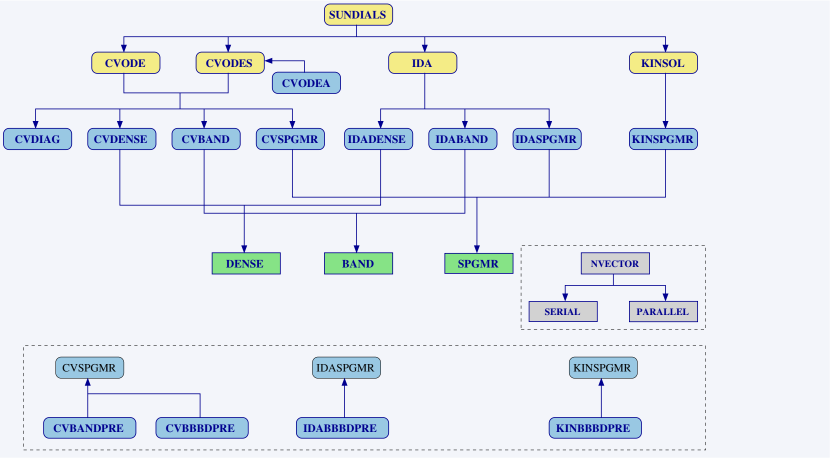
Implementation

1. integrate forward step by step
2. dump checkpoint data after a given number of steps
3. continue until t_f.
4. evaluate final conditions for adjoint problem
5. store interpolation data on second forward pass
6. propagate adjoint variables backward in time
7. total cost: 2 forward passes + 1 backward pass

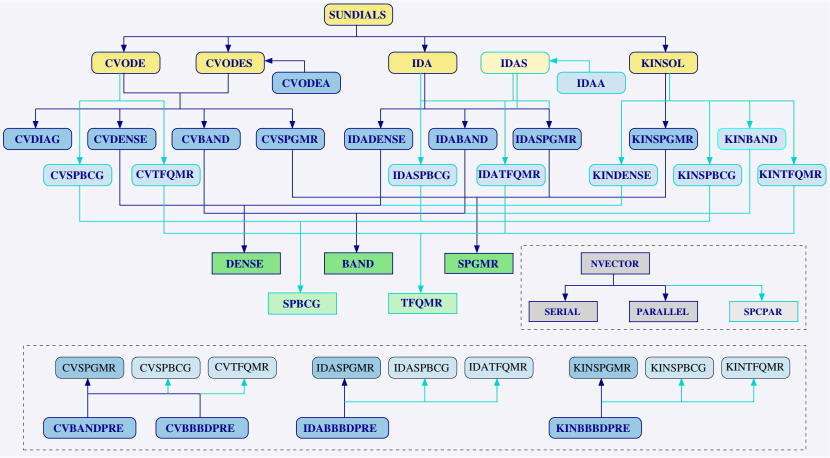


Outline

The SUNDIALS suite: v.2.1.1



The SUNDIALS suite: next release



IVP integration with CVODES

Main function

```
/* Set tolerances, initial time, etc. */
y = N_VNew_Serial(n);
/* Load I.C. into y */
cvode_mem = CVodeCreate(CV_BDF, CV_NEWTON);
flag = CVodeMalloc(cvode_mem, f, t0, y, CV_SS, rtol, atol);
flag = CVodeSetFdata(cvode_mem, my_data);
flag = CVDense(cvode_mem, n);
for (iout=1; iout<= NOUT; iout++) {
 flag = CVode(cvode_mem, tout, y, &t, CV_NORMAL);
 /* Process solution y */
}
N_VDestroy_Serial(y);
CVodeFree(cvode_mem);
```

Required functions

- right-hand side
- quadrature integrand
- g-function

Optional functions

- Jacobian data
- preconditioner
- error weights

FSA with CVODES

Main function

```
y = N_VNew*(n,...);
cvode_mem = CVodeCreate(...);
flag = CVodeMalloc(...);
flag = CVodeSet*(...);
for (iout=1; iout<= NOUT; iout++) {
 flag = CVode(...);
}
N_VDestroy*(y);
CVodeFree(cvode_mem);
```

FSA with CVODES

Main function (instrumented for FSA)

```
y = N_VNew*(n,...);
cvode_mem = CVodeCreate(...);
flag = CVodeMalloc(...);
flag = CVodeSet*(...);
yS = N_VNewVectorArray*(...);
flag = CVodeSensMalloc(...);
flag = CVodeSetSens*(...);
for (iout=1; iout<= NOUT; iout++) {
 flag = CVode(...);
 flag = CVodeGetSens(...);
}
N_VDestroy*(y);
N_VDestroyVectorArray*(...,yS);
CVodeFree(cvode_mem);
```

FSA with CVODES

CASC

Main function (instrumented for FSA)

```
y = N_VNew*(n,...);
cvode_mem = CVodeCreate(...);
flag = CVodeMalloc(...);
flag = CVodeSet*(...);
yS = N_VNewVectorArray*(...);
flag = CVodeSensMalloc(...);
flag = CVodeSetSens*(...);
for (iout=1; iout<= NOUT; iout++) {
 flag = CVode(...);
 flag = CVodeGetSens(...);
}
N_VDestroy*(y);
N_VDestroyVectorArray*(...,yS);
CVodeFree(cvode_mem);
```

FSA with CVODES

Main function (instrumented for FSA)

```
y = N_VNew*(n,...);
cvode_mem = CVodeCreate(...);
flag = CVodeMalloc(...);
flag = CVodeSet*(...);
yS = N_VNewVectorArray*(...);
flag = CVodeSensMalloc(...);
flag = CVodeSetSens*(...);
for (iout=1; iout<= NOUT; iout++) {
 flag = CVode(...);
 flag = CVodeGetSens(...);
}
N_VDestroy*(y);
N_VDestroyVectorArray*(...,yS);
CVodeFree(cvode_mem);
```

FSA with CVODES

Main function (instrumented for FSA)

```
y = N_VNew*(n,...);
cvode_mem = CVodeCreate(...);
flag = CVodeMalloc(...);
flag = CVodeSet*(...);
yS = N_VNewVectorArray*(...);
flag = CVodeSensMalloc(...);
flag = CVodeSetSens*(...);
for (iout=1; iout<= NOUT; iout++) {
 flag = CVode(...);
 flag = CVodeGetSens(...);
}
N_VDestroy*(y);
N_VDestroyVectorArray*(...,yS);
CVodeFree(cvode_mem);
```

FSA with CVODES

Main function (instrumented for FSA)

```
y = N_VNew*(n,...);
cvode_mem = CVodeCreate(...);
flag = CVodeMalloc(...);
flag = CVodeSet*(...);
yS = N_VNewVectorArray*(...);
flag = CVodeSensMalloc(...);
flag = CVodeSetSens*(...);
for (iout=1; iout<= NOUT; iout++) {
 flag = CVode(...);
 flag = CVodeGetSens(...);
}
N_VDestroy*(y);
N_VDestroyVectorArray*(...,yS);
CVodeFree(cvode_mem);
```

FSA with CVODES

Main function (instrumented for FSA)

```
y = N_VNew*(n,...);
cvode_mem = CVodeCreate(...);
flag = CVodeMalloc(...);
flag = CVodeSet*(...);
yS = N_VNewVectorArray*(...);
flag = CVodeSensMalloc(...);
flag = CVodeSetSens*(...);
for (iout=1; iout<= NOUT; iout++) {
 flag = CVode(...);
 flag = CVodeGetSens(...);
}
N_VDestroy*(y);
N_VDestroyVectorArray*(...,yS);
CVodeFree(cvode_mem);
```

FSA with CVODES

Main function (instrumented for FSA)

```
y = N_VNew*(n,...);
cvode_mem = CVodeCreate(...);
flag = CVodeMalloc(...);
flag = CVodeSet*(...);
yS = N_VNewVectorArray*(...);
flag = CVodeSensMalloc(...);
flag = CVodeSetSens*(...);
for (iout=1; iout<= NOUT; iout++) {
 flag = CVode(...);
 flag = CVodeGetSens(...);
}
N_VDestroy*(y);
N_VDestroyVectorArray*(...,yS);
CVodeFree(cvode_mem);
```

ASA with CVODES

Main function

```
y = N_VNew*(n,...);
cvode_mem = CVodeCreate(...);
flag = CVodeMalloc(...);
flag = CVodeSet*(...);
for (iout=1; iout<= NOUT; iout++) {
 flag = CVode(...);
}
N_VDestroy*(y);
CVodeFree(cvode_mem);
```

ASA with CVODES

Main function (instrumented for ASA)

```
y = N_VNew*(n,...);
cvode_mem = CVodeCreate(...);
flag = CVodeMalloc(...);
flag = CVodeSet*(...);
cvadj_mem = CVadjMalloc(...);
for (iout=1; iout<= NOUT; iout++) {
 /*flag = CVode(...);*/
 flag = CVodeF(...);
}
yB = N_VNew*(nB,...);
flag = CVodeCreateB(...);
flag = CVodeMallocB(...);
flag = CVodeSet*B(...);
flag = CVodeB(...);
N_VDestroy*(y);
CVodeFree(cvode_mem);
N_VDestroy*(yB);
CVadjFree(cvadj_mem);
```

ASA with CVODES

Main function (instrumented for ASA)

```
y = N_VNew*(n,...);
cvode_mem = CVodeCreate(...);
flag = CVodeMalloc(...);
flag = CVodeSet*(...);
cvadj_mem = CVadjMalloc(...);
for (iout=1; iout<= NOUT; iout++) {
 /*flag = CVode(...);*/
 flag = CVodeF(...);
}
yB = N_VNew*(nB,...);
flag = CVodeCreateB(...);
flag = CVodeMallocB(...);
flag = CVodeSet*B(...);
flag = CVodeB(...);
N_VDestroy*(y);
CVodeFree(cvode_mem);
N_VDestroy*(yB);
CVadjFree(cvadj_mem);
```

ASA with CVODES

Main function (instrumented for ASA)

```
y = N_VNew*(n,...);
cvode_mem = CVodeCreate(...);
flag = CVodeMalloc(...);
flag = CVodeSet*(...);
cvadj_mem = CVadjMalloc(...);
for (iout=1; iout<= NOUT; iout++) {
 /*flag = CVode(...);*/
 flag = CVodeF(...);
}
yB = N_VNew*(nB,...);
flag = CVodeCreateB(...);
flag = CVodeMallocB(...);
flag = CVodeSet*B(...);
flag = CVodeB(...);
N_VDestroy*(y);
CVodeFree(cvode_mem);
N_VDestroy*(yB);
CVadjFree(cvadj_mem);
```

ASA with CVODES

Main function (instrumented for ASA)

```
y = N_VNew*(n,...);
cvode_mem = CVodeCreate(...);
flag = CVodeMalloc(...);
flag = CVodeSet*(...);
cvadj_mem = CVadjMalloc(...);
for (iout=1; iout<= NOUT; iout++) {
 /*flag = CVode(...);*/
 flag = CVodeF(...);
}
yB = N_VNew*(nB,...);
flag = CVodeCreateB(...);
flag = CVodeMallocB(...);
flag = CVodeSet*B(...);
flag = CVodeB(...);
N_VDestroy*(y);
CVodeFree(cvode_mem);
N_VDestroy*(yB);
CVadjFree(cvadj_mem);
```

ASA with CVODES

CASC

Main function (instrumented for ASA)

```
y = N_VNew*(n,...);
cvode_mem = CVodeCreate(...);
flag = CVodeMalloc(...);
flag = CVodeSet*(...);
cvadj_mem = CVadjMalloc(...);
for (iout=1; iout<= NOUT; iout++) {
 /*flag = CVode(...);*/
 flag = CVodeF(...);
}
yB = N_VNew*(nB,...);
flag = CVodeCreateB(...);
flag = CVodeMallocB(...);
flag = CVodeSet*B(...);
flag = CVodeB(...);
N_VDestroy*(y);
CVodeFree(cvode_mem);
N_VDestroy*(yB);
CVadjFree(cvadj_mem);
```

ASA with CVODES

Main function (instrumented for ASA)

```
y = N_VNew*(n,...);
cvode_mem = CVodeCreate(...);
flag = CVodeMalloc(...);
flag = CVodeSet*(...);
cvadj_mem = CVadjMalloc(...);
for (iout=1; iout<= NOUT; iout++) {
 /*flag = CVode(...);*/
 flag = CVodeF(...);
}
yB = N_VNew*(nB,...);
flag = CVodeCreateB(...);
flag = CVodeMallocB(...);
flag = CVodeSet*B(...);
flag = CVodeB(...);
N_VDestroy*(y);
CVodeFree(cvode_mem);
N_VDestroy*(yB);
CVadjFree(cvadj_mem);
```

Some packages using SUNDIALS solvers

ARDRA Neutron and Radiation Transport
`http://www.llnl.gov/casc/Ardra/`

DELPHIN4 Coupled heat, moisture, air and salt transport
`http://www.bauklimatik-dresden.de/`

EMSO Environment for Modeling, Simulation, and Optimization
`http://vrtech.com.br/rps/emso.html`

magpar Parallel Finite Element Micromagnetics Package
`http://magnet.atp.tuwien.ac.at/scholz/magpar/`

Mathematica Wolfram Research
`http://www.wolfram.com/products/mathematica/index.html`

NEURON Empirically-based simulations of networks of neurons
`http://www.neuron.yale.edu/neuron/`

PETSc The Portable, Extensible Toolkit for Scientific Computation
`http://www-unix.mcs.anl.gov/petsc/`

SAMRAI Structured Adaptive Mesh Refinement Application Infrastructure
`http://www.llnl.gov/CASC/samrai/`

SBML Systems Biology Markup Language
`http://www.sbml.org/software/libsbml/`

Simulation applications

CASC

@LLNL

- **CVODE** is used in a 3D parallel tokamak turbulence model in LLNL's Magnetic Fusion Energy Division.
  Typical run: 7 unknowns on a 64x64x40 mesh, with 60 processors
- **KINSOL** with a hypre multigrid preconditioner is used in LLNL's Geosciences Division for an unsaturated porous media flow model.
  Fully scalable performance has been obtained on up to 225 processors on ASCI Blue.
- All solvers are being used to solve 3D neutral particle transport problems in CASC.
  Scalable performance obtained on up to 5800 processors on ASCI Red.
- Other applications: disease detection, astrophysics, magnetohydrodynamics, etc.

Other

- Many more in very different areas...

Sensitivity analysis applications

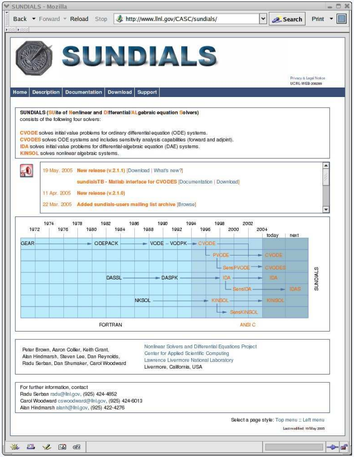
@LLNL

- Solution sensitivities in neutral particle transport applications
- Sensitivity analysis of groundwater simulations
- Sensitivity analysis of chemically reacting flows
- Sensitivity analysis of radiation transport (diffusion approximation)
- Inversion of large-scale time dependent PDEs (atmospheric releases).

Other

- Optimization of periodic adsorption processes (L.T. Biegler, CMU)
- Nonlinear model predictive control (A. Romanenko, Enginum)
- Controller design (Y. Cao, Cranfield U.)

www.llnl.gov/CASC/sundials



The SUNDIALS suite

- Open source, BSD license
- Complete documentation (HTML, PDF, PS)
- User support (mailing lists, Bugzilla bug tracking)
- (May 19, 2005): Matlab interface to **CVODES** and **KINSOL**

The SUNDIALS team

Peter Brown, Aaron Collier, Keith Grant, Alan Hindmarsh, Steven Lee, Radu Serban, Dan Shumaker, Carol Woodward

Past contributors

Scott Cohen and Allan Taylor

UCRL-PRES-213978

This work was performed under the auspices of the U.S. Department of Energy by the University of California, Lawrence Livermore National Laboratory under Contract No. W-7405-Eng-48.