Purely Incremental Methods: Implementation
## TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>§23.1</td>
<td>Introduction</td>
<td>23–3</td>
</tr>
<tr>
<td>§23.2</td>
<td>Functional Organization of Nonlinear Analysis</td>
<td>23–3</td>
</tr>
<tr>
<td>§23.2.1</td>
<td>General Purpose FEM Code</td>
<td>23–3</td>
</tr>
<tr>
<td>§23.2.2</td>
<td>Educational Program</td>
<td>23–5</td>
</tr>
<tr>
<td>§23.3</td>
<td>User Choices</td>
<td>23–5</td>
</tr>
<tr>
<td>§23.3.1</td>
<td>Predictors</td>
<td>23–6</td>
</tr>
<tr>
<td>§23.3.2</td>
<td>Increment Control Strategies</td>
<td>23–6</td>
</tr>
<tr>
<td>§23.3.3</td>
<td>Corrector Choice</td>
<td>23–6</td>
</tr>
<tr>
<td>§23.3.4</td>
<td>Benchmark Problems</td>
<td>23–6</td>
</tr>
<tr>
<td>§23.4</td>
<td>Solution Advancing Modules</td>
<td>23–6</td>
</tr>
<tr>
<td>§23.4.1</td>
<td>Nonlinear Solution Driver</td>
<td>23–8</td>
</tr>
<tr>
<td>§23.4.2</td>
<td>Setting Up The Reference Configuration</td>
<td>23–10</td>
</tr>
<tr>
<td>§23.4.3</td>
<td>Performing An Incremental Step</td>
<td>23–11</td>
</tr>
<tr>
<td>§23.4.4</td>
<td>Incremental Step Via Forward Euler</td>
<td>23–13</td>
</tr>
<tr>
<td>§23.4.5</td>
<td>Incremental Step Via Second Order RK</td>
<td>23–13</td>
</tr>
<tr>
<td>§23.4.6</td>
<td>Incremental Step Via Third Order RK</td>
<td>23–13</td>
</tr>
<tr>
<td>§23.4.7</td>
<td>Incremental Step Via Fourth Order RK</td>
<td>23–14</td>
</tr>
<tr>
<td>§23.4.8</td>
<td>Incremental and Residual Velocities</td>
<td>23–15</td>
</tr>
<tr>
<td>§23.5</td>
<td>Problem Interface Modules</td>
<td>23–18</td>
</tr>
<tr>
<td>§23.5.1</td>
<td>Total Residual Interface</td>
<td>23–18</td>
</tr>
<tr>
<td>§23.5.2</td>
<td>Tangent Stiffness Interface</td>
<td>23–19</td>
</tr>
<tr>
<td>§23.5.3</td>
<td>Incremental Load Interface</td>
<td>23–19</td>
</tr>
<tr>
<td>§23.6</td>
<td>Solution Display Modules</td>
<td>23–20</td>
</tr>
<tr>
<td>§23.6.1</td>
<td>Solution Table Printing</td>
<td>23–21</td>
</tr>
<tr>
<td>§23.6.2</td>
<td>Solution Table 2D Plotting</td>
<td>23–25</td>
</tr>
<tr>
<td>§23.6.3</td>
<td>Solution Table 3D Plotting</td>
<td>23–28</td>
</tr>
<tr>
<td>§23.6.4</td>
<td>Auxiliary Solution Table PrintPlot Modules</td>
<td>23–29</td>
</tr>
<tr>
<td>§23.7</td>
<td>Utility Modules</td>
<td>23–29</td>
</tr>
<tr>
<td>§23.7.1</td>
<td>Butcher Table</td>
<td>23–29</td>
</tr>
<tr>
<td>§23.7.2</td>
<td>Plot Display Channel</td>
<td>23–29</td>
</tr>
<tr>
<td>§23.7.3</td>
<td>ICS Factor</td>
<td>23–30</td>
</tr>
<tr>
<td>§23.7.4</td>
<td>Matrix Frobenius Norm</td>
<td>23–31</td>
</tr>
<tr>
<td>§23.7.5</td>
<td>Problem Information</td>
<td>23–31</td>
</tr>
</tbody>
</table>
§23.1. Introduction

This Chapter describes a tiny program called *GeNoBe*, a portmanteau that stands for *Geometrically Nonlinear Benchmarking*.¹ Its purpose is to numerically solve selected benchmark problems that involve geometric nonlinearities, using incremental-iterative solution methods. The benchmarks may involve finite element models, but they also include some that are simply mathematical objects chosen to illustrate selected features of solution methods. The version described in this Chapter performs purely incremental analysis only, skipping corrections. It is expanded to include a corrector in Chapter 3.

Unlike some of its Lockheed ancestors (which are identified in its top text cell) *GeNoBe* is only a special purpose code. It can only process the subset of problems available in its benchmark library. This restriction is prompted by trying to keep simplicity for educational use. A general-purpose nonlinear FEM program is a highly complex beast that comprises millions of code lines. Commercial products of that nature are black boxes with inaccessible innards. Such stealthy behemoths are out of place in a course of this scope, in which emphasis is placed on full transparency and visibility.

*GeNoBe* is implemented in the *Mathematica* language. The description that begins in §23.4 is top down. That is, it starts from the nonlinear solution driver down to more and more specialized modules. (Script “templates” are illustrated at the end of the Chapter.) The program can support questions given in homeworks and exams if they pertain to the analysis of problems covered by this implementation. Thus this Chapter serves as an informal users manual for such questions.

§23.2. Functional Organization of Nonlinear Analysis

This section gives a quick overview of the functional organization of nonlinear FEM codes. Similarities and differences between two extremes: a general purpose commercial FEM code, and the “baby” educational code used here, are noted.

§23.2.1. General Purpose FEM Code

The overall organization of a general purpose nonlinear FEM code for static structural analysis is flowcharted in Figure 23.1. This picture is mainly intended to establish comparison with a educational. special purpose code such as *GeNoBe*. It does not represent any specific commercial program but tries to display main functional groupings and their interrelations. Boxes enclosed in dashed lines are external to the FEM program proper. Box color conventions are as follows:

*Blue Boxes.* The program user and its interface. The term “user” includes both a human user directly interacting with the FEM program, or an external “wrapper” program that uses the FEM solver as a supporting stub; for example a CAD system.²

*Green Boxes.* The main program components: preprocessor, processor and postprocessor. Pre- and postprocessors may carry elaborated interfaces to other software components, such as mesh generators or adaptors, which are not shown for clarity. Often these may be separately licensed from the same or other vendors.

---

¹ Pronunciation close to “gnome” — a diminutive being (it fits). Also sounds like the clan of Obi Wan Kenobi of *Star Wars*.

² For example, the ABAQUS code wrapped by Dassault’s CATIA.
Figure 23.1. Functional organization of a general purpose FEM program for static nonlinear analysis.

Figure 23.2. Functional organization of the educational program GeNoBe.
Orange Boxes. Driver software that advances and monitors the solution advancing process.\(^3\)  
Yellow Boxes. Computational software devoted to problem solving.\(^4\) Informmally known as the “number crunchers.”  
Grey Boxes. Utilities that support the rest of the code. Miscellaneous ones may include array utilities, for example a free-source linear algebra package such as LAPack. Architectural utilities provide interfaces to the operating system, such as file managers and graphics libraries. These isolate functional modules, such as linear solvers or mesh plotters, from platform specific details.  

Box sizes in Figure 23.1 bear no relation to software volume. For example, in typical commercial FEM codes pre- and postprocessor modules typically overwhelm the rest. Within the processor proper, the element library may represent 90% or more of the entire software.  

Programs of this nature tend to grow in size like malignant tumors until they reach millions of lines of low level code. One curious feature (also observed by software archaeologists in other long-lived software systems) is the presence of legacy “zombie code,” written by long departed developers or interns, undocumented,\(^5\) and allegedly superseded by newer software. Necrotic software tends to remain imbedded forever since removal carries fears of triggering obscure bugs.

\(\S 23.2.2.\) Educational Program

The functional organization of the GeNoBe program is flowcharted in Figure 23.2. The similarity with the chart in Figure 23.1 ought not be surprising. All FEM (or FEM-like) programs based on the Direct Stiffness Method (DSM) look functionally alike, whether they have 500 code lines or 10 million. It is a matter of scale, resources, capabilities, and goals. An educational code like GeNoBe can be learned by a bright graduate student in a few hours. A 10-million-line FEMzilla is beyond individual comprehension.\(^6\)

Figure 23.3 list names of the GeNoBe modules, grouped within the appropriate functional boxes. Names of modules in the problem problem library are not listed to reduce clutter. Only the implemented benchmarks are listed.

\(\S 23.3.\) User Choices

Before starting the description of the user scripts, it is convenient to go over the set of analysis methods and problems that have been implemented in GeNoBe. For a purely incremental solution, the user makes three decisions:

- Integration scheme (predictor)
- Increment control strategy
- Benchmark problem to solve

---

3 In programs that carry out nonlinear dynamic analysis, the “incremental driver” advances the solution over real time steps. The corrector driver, if any, “freezes” the time to execute residual iterations.

4 A separate constitutive library is clearly separated only if the program treats material nonlinearities. Modules in that library are usually invoked at element Gauss points. If the program does not consider material nonlinearities, that separation is unnecessary.

5 “Perhaps a few cryptic comments littered in the code tomb” (Kevin Davis).

6 Some of those monsters still lumber around, and are gradually expanding into multiphysics: a tribute to the enduring power of profit, as well as to the ubiquity and endurance of the DSM.
These are covered in the next subsections. If a corrective process is to be included, the user selects

- Corrector method

This specification is discussed in Chapter 25ff.

§23.3.1. Predictors

Predictors advance the first-order rate equations over one incremental step. All predictors implemented in *GenoBe* are members of the class of Explicit Runge Kutta (ERK) integrators. These are described in §21.5. A total of seven predictors are currently implemented. They are listed in Table 21.1, which is reproduced in Table 23.1 for the reader’s convenience.

§23.3.2. Increment Control Strategies

A total of four incremental control strategies (ICS) have been implemented in *GenoBe*. These are identified in the first four entries of Table 23.2. A fifth strategy: work control, has not yet been implemented but an acronym for it is reserved.

All ICS may be combined with scaling on state degrees of freedom. Details are given later in conjunction with the utility module ICSFactor.

§23.3.3. Corrector Choice

If a corrector is used, several of its attributes are specified. This option is discussed in Chapter ?.

§23.3.4. Benchmark Problems

These are listed in Table 23.3. More details are given in Chapter 22.
Table 23.1. Predictors for Advancing Over an Incremental Step

<table>
<thead>
<tr>
<th>ERK scheme</th>
<th>Acronym</th>
<th>$R$ †</th>
<th>Ref.</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Forward Euler</td>
<td>FE</td>
<td>1</td>
<td>§21.3</td>
<td>First ODE integration scheme (Euler, 1768)</td>
</tr>
<tr>
<td>Explicit Midpoint Rule</td>
<td>EMR</td>
<td>2</td>
<td>§21.4.1</td>
<td>Explicit form of the Midpoint Rule</td>
</tr>
<tr>
<td>Explicit Trapezoidal Rule</td>
<td>ETR</td>
<td>2</td>
<td>§21.4.2</td>
<td>Explicit form of the Trapezoidal Rule</td>
</tr>
<tr>
<td>Heun’s third order RK</td>
<td>RK3H</td>
<td>3</td>
<td>[452]</td>
<td>Classical third-order RK</td>
</tr>
<tr>
<td>Ralston’s third order RK</td>
<td>RK3R</td>
<td>3</td>
<td>[628]</td>
<td>Tries to minimize roundoff effects</td>
</tr>
<tr>
<td>Classical fourth order RK</td>
<td>RK4C</td>
<td>4</td>
<td>[452]</td>
<td>The original RK (Runge, 1895; Kutta, 1901)</td>
</tr>
<tr>
<td>Kutta’s fourth order RK</td>
<td>RK4K</td>
<td>4</td>
<td>[452]</td>
<td>Classical variant of RK4C</td>
</tr>
</tbody>
</table>

† $R$ is the number of evaluations per incremental step; see §21.5.1. Same as order if $R \leq 4$.
‡ By “classical” is meant that the scheme was developed in the pre-computer era.

Table 23.2. Increment Control Strategies in GeNoBe

<table>
<thead>
<tr>
<th>Stepsize control strategy</th>
<th>Acronym</th>
<th>Ref.</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load (a.k.a. $\lambda$) control</td>
<td>LC</td>
<td>Chapter 20</td>
<td>Also known as stage parameter control</td>
</tr>
<tr>
<td>State control</td>
<td>SC</td>
<td>Chapter 20</td>
<td>Also known as displacement control</td>
</tr>
<tr>
<td>Arclength control</td>
<td>AC</td>
<td>Chapter 20</td>
<td>Linearized form of arclength control</td>
</tr>
<tr>
<td>Hyperelliptic control</td>
<td>HC</td>
<td>Chapter 20</td>
<td>Includes LC and SC as special cases</td>
</tr>
<tr>
<td>Work control</td>
<td>WC</td>
<td>[57]</td>
<td>Not implemented</td>
</tr>
</tbody>
</table>

Table 23.3. Benchmark Problems in GeNoBe

<table>
<thead>
<tr>
<th>Problem</th>
<th>Acronym</th>
<th>Ref.</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mises Truss</td>
<td>MT</td>
<td>Chapters 6ff</td>
<td>Total Lagrangian formulation</td>
</tr>
<tr>
<td>Circle Game</td>
<td>CG</td>
<td>Chapter 5</td>
<td>Not a FEM problem</td>
</tr>
<tr>
<td>Space Frame</td>
<td>SF</td>
<td>[846]</td>
<td>Not implemented</td>
</tr>
<tr>
<td>Toggle’s Frame</td>
<td>TF</td>
<td>TBD</td>
<td>Not implemented</td>
</tr>
<tr>
<td>Lee’s Frame</td>
<td>LF</td>
<td>TBD</td>
<td>Not implemented</td>
</tr>
</tbody>
</table>

§23.4. Solution Advancing Modules

The modules grouped in this Section are tasked with advancing the nonlinear solution in step by step manner. Their common feature is that they do not access benchmark problem modules directly. They do that indirectly through the problem interface modules covered in §23.5.
Chapter 23: PURELY INCREMENTAL METHODS: IMPLEMENTATION

{probid, λ, ctlpar, state, numdof, nmax, λmax, umax, unorm, kmax, zv, solz, inisol, solret, nextsol, soltab, n, m, status = ""},
{inisol, status} = SetRefConfig[problem, method, geopar, matpar, fabpar, 
inipar, incpar, corpar, inictl, insta, rfload, options];
If [status != "", Return[{{inisol}, status}]]; probid = problem[[1]]; {nmax, λmax, umax} = Take[incpar, 3];
numdof = ProblemInfo[probid]; zv = Table[0, {numdof}];
solz = {0, 0, zv, zv, zv, zv, 0, {0, 0}, 0, 0, 0, " "];
soltab = Table[solzv, {nmax + 2}]; soltab[[1]] = inisol; m = Length[solz];
{n, λ, state} = Take[inisol, 3]; ctlpar = {λ}; unorm = Sqrt[state.state];
While [status == " " && n < nmax && Abs[λ] <= λmax && unorm <= umax,
lastsol = soltab[[n + 1]]; 
{predsol, status} = IncStepDriver[problem, method, geopar, matpar, 
fabpar, inipar, incpar, corpar, ctlpar, state, rfload, options, {n, m}];
{n, λ, state} = Take[predsol, 3]; unorm = Sqrt[state.state];
ctlpar = {λ}; nextsol = predsol; kmax = corpar[[1]]; 
If [n >= nmax, nextsol[[m]] = "n >= ToString[nmax]"]; 
If [unorm > umax, nextsol[[m]] = "||u|| > ToString[umax]"];
If [Abs[λ] > λmax, nextsol[[m]] = "|λ| > ToString[λmax]"];
soltab[[n + 1]] = nextsol; ];
solret = Take[soltab, n + 1]; ClearAll[soltab];
Return[{{solret, status}]};

Figure 23.4. The NonLinSolDriver module that drives the step by step computations.

§23.4.1. Nonlinear Solution Driver

The code of the driver module NonLinSolDriver is listed in Figure 23.4. It is invoked as

{solret, status} = NonLinSolDriver[problem, method, geopar, matpar, fabpar, 
inipar, incpar, corpar, refctl, refsta, rfload, options];

The arguments are

- **problem**  A list: {probid, probvar} of benchmark problem identifiers; see Table 23.4
- **method**  A list: {ics, pred, corr} of solution method specifications; see Table 23.4
- **geopar**  A list of geometric parameters (e.g. node coordinates) for the benchmark problem; see Chapter 24
- **matpar**  A list of material parameters (e.g. elastic moduli) for the benchmark problem; see Chapter 24
- **fabpar**  A list of fabrication parameters (e.g., cross section areas, plate thicknesses) for the benchmark problem; see Chapter 24
- **inipar**  A list of initialization parameters for the benchmark problem; see Chapter 24
- **incpar**  A list: {nmax, ...} of increment control parameters; see Table 23.4
- **corpar**  A list: {kmax, ...} of correction parameters; see Table 23.4
- **refctl**  Reference control value; see Table 23.4
- **refsta**  Reference state; see Table 23.4
- **rfload**  Reference load see Table 23.4
- **options**  General processing options; see Table 23.4
### Table 23.4. Configuration Details of NonLinSolDriver Arguments

<table>
<thead>
<tr>
<th>Argument*</th>
<th>Item</th>
<th>Type†</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>problem</td>
<td>probid</td>
<td>String</td>
<td>Benchmark problem identifier – see Table 23.3</td>
</tr>
<tr>
<td></td>
<td>probvar</td>
<td>String</td>
<td>Benchmark problem variant (optional, may be omitted)</td>
</tr>
<tr>
<td>method</td>
<td>ics</td>
<td>String</td>
<td>Identifier of increment control strategy — see Table 23.2</td>
</tr>
<tr>
<td></td>
<td>pred</td>
<td>String</td>
<td>Identifier of ERK integrator used as predictor — see Table 23.1</td>
</tr>
<tr>
<td>incpar</td>
<td>nmax</td>
<td>Integer</td>
<td>Maximum number of incremental steps</td>
</tr>
<tr>
<td></td>
<td>( \lambda )</td>
<td>Numeric</td>
<td>Maximum absolute value of staging parameter</td>
</tr>
<tr>
<td></td>
<td>umax</td>
<td>Numeric</td>
<td>Maximum state vector 2-norm</td>
</tr>
<tr>
<td></td>
<td>adapt</td>
<td>Logical</td>
<td>True to specify steplength adaptation (presently unused)</td>
</tr>
<tr>
<td></td>
<td>ell</td>
<td>Numeric</td>
<td>Increment steplength (initial value if adaptation specified)</td>
</tr>
<tr>
<td></td>
<td>ellmin</td>
<td>Numeric</td>
<td>Minimum increment steplength if adaptation specified</td>
</tr>
<tr>
<td></td>
<td>ellmax</td>
<td>Numeric</td>
<td>Maximum increment steplengh if adaptation specified</td>
</tr>
<tr>
<td></td>
<td>icscal</td>
<td>List</td>
<td>Scaling information used in non-LC increment control strategy</td>
</tr>
<tr>
<td></td>
<td>eigsK</td>
<td>Logical</td>
<td>True to return max/min eigenvalues of ( K ) in soltab</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td></td>
<td>Additional items may be added</td>
</tr>
<tr>
<td>corpar</td>
<td>kmax</td>
<td>Numeric</td>
<td>Max number of corrective iterations (presently a placeholder)</td>
</tr>
<tr>
<td></td>
<td>acctol</td>
<td>Numeric</td>
<td>Tolerance to stop corrections (presently a placeholder)</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td>(varies)</td>
<td>Additional items may be added</td>
</tr>
<tr>
<td>rflload</td>
<td>List</td>
<td>Numeric</td>
<td>Reference load vector ( q_0 ) if loading is proportional, in which case ( \lambda q_0 ) is external force. If loading is not proportional, this argument is ignored, and should be set to the empty list.</td>
</tr>
<tr>
<td>refctl</td>
<td>List</td>
<td>Numeric</td>
<td>Control parameter ( \lambda_0 ) at initial configuration, specified as ( \text{refctl} = { \lambda_0 } ). If an empty list, zero is assumed.</td>
</tr>
<tr>
<td>refsta</td>
<td>List</td>
<td>List</td>
<td>State vector ( u_0 ) at initial configuration specified as ( \text{refsta} = u_0 ). If an empty list, the zero vector is assumed.</td>
</tr>
<tr>
<td>options</td>
<td>numer</td>
<td>Logical</td>
<td>True for floating-point work (the usual setting)</td>
</tr>
</tbody>
</table>

* Arguments geopar, matpar and fabpar are benchmark dependent, and not described here.
† A “numeric” item may be written as integer, fraction, or real. It is internally converted to real if numer is True (as usual). For example, 2/5 and 0.4 are equivalent in such case.

As function value the module returns:

- `soltab` Solution table. A list that records computation results at the end of each incremental step. On exit from NonLinSolDriver after \( n \) steps, the table has \( n + 1 \) rows (only one if computations abort at initialization time so \( n = 0 \)). Each row contains 12 items (atomic or lists), which are specified in Table 23.5.
- `soltab` A textstring that specifies the solution status on exit. Blank if no abnormal condition occurred. If a fatal error is detected either during initialization or during it, status will indicate the reason.
Table 23.5. Solution Table Row Configuration

<table>
<thead>
<tr>
<th>#</th>
<th>Item</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>n</td>
<td>Integer</td>
<td>Step number n, starting at 0.</td>
</tr>
<tr>
<td>2</td>
<td>λₙ</td>
<td>Numeric</td>
<td>Control parameter λₙ upon completing step</td>
</tr>
<tr>
<td>3</td>
<td>u</td>
<td>List</td>
<td>State vector uₙ upon completing step</td>
</tr>
<tr>
<td>4</td>
<td>v</td>
<td>List</td>
<td>Incremental velocity vector vₙ₊α at last ERK station †</td>
</tr>
<tr>
<td>5</td>
<td>r</td>
<td>List</td>
<td>Residual vector rₙ upon completing step</td>
</tr>
<tr>
<td>6</td>
<td>q</td>
<td>List</td>
<td>Incremental load vector qₙ₊α at last ERK station †</td>
</tr>
<tr>
<td>7</td>
<td>Kdet</td>
<td>Numeric</td>
<td>Stiffness determinant det(Kₙ₊α) at last RRK station †</td>
</tr>
<tr>
<td>8</td>
<td>{Kmax,Kmin}</td>
<td>List</td>
<td>Max &amp; min eigenvalues of Kₙ₊α at last ERK station †. Computed only if eigsK in incpar is True; else an empty list</td>
</tr>
<tr>
<td>9</td>
<td>elln</td>
<td>Numeric</td>
<td>Steplength ℓₙ used in step n</td>
</tr>
<tr>
<td>10</td>
<td>{...}</td>
<td>List</td>
<td>Placeholder for step adaptation information</td>
</tr>
<tr>
<td>11</td>
<td>{...}</td>
<td>List</td>
<td>Placeholder for corrector exit information</td>
</tr>
<tr>
<td>12</td>
<td>remark</td>
<td>String</td>
<td>Informative message; e.g., termination condition on solution exit</td>
</tr>
</tbody>
</table>

* A “numeric” item may be written as integer, fraction, or real. It is internally converted to real if numer is True (as usual). For example, 2/5 and 0.4 are equivalent in such case.
† (.)ₙ₊α is the value at the last station of the RK scheme used.
α = 0 only for the Forward Euler (FE) scheme, which evaluates at (.)ₙ

Note that intermediate results computed by ERK schemes of second through fourth order are not saved in the solution table.

This module starts by calling the initialization module SetRefConfig to set up the reference configuration, as well as conducting various checks. If no error is detected, it enters a While loop during which it calls IncStepDriver to advance the solution by an incremental step. This continues until either a normal termination condition (such as max number of steps) is verified, or an error condition (signaled by a nonblank status) is returned by IncStepDriver.

§23.4.2. Setting Up The Reference Configuration

Module SetRefConfig, listed in Figure 23.5, is called by NonLinSolDriver to initialize the solution process by setting up the reference control-state configuration. The module is invoked as

```
{inisol,status}= SetRefConfig[problem,method,geopar,matpar,fabpar,inipar,incpar,conpar,refctl,refsta,rfload,options];
```

The arguments of SetRefConfig are exactly the same as those of NonLinSolDriver. Those are described in §23.4.1. The function returns are:

- **inisol** Initialized first-row of the solution table. If no error is detected, inisol is set to the reference state. This state, specified by arguments refctl and refsta, is assumed to be a solution of the nonlinear system.
- **status** A textstring that specifies the solution status on exit. If a fatal error is detected (for example, an unimplemented problem or method) status returns a descriptive error message, while sol0 returns null values.
§23.4 SOLUTION ADVANCING MODULES

```
SetRefConfig[problem_, method_, geopar_, matpar_, fabpar_, inipar_, 
inpar_, corpar_, incpar_, inictl_, inista_, rffield_, options_] :=
Module[{probid, numdof, ics, int, zv = {0}, solz, 
A, c, w, n, λ0, u0, elln, v0, vr, q0, r0, Kdet, Keigs, inisol, status = " "},
solz = {0, 0, zv, zv, zv, zv, 0, {0, 0}, 0, 0, " "};
probid = problem[[1]]; numdof = ProblemInfo[probid];
If [numdof <= 0, status = "Unimplemented problem " <> probid;
    Return[{solz, status}]];
zv = Table[0, {numdof}]; solz = {0, 0, zv, zv, zv, zv, 0, {0, 0}, 0, 0, " "};
{ics, int} = Take[method, 2]; {A, c, w} = ButcherTable[int];
If [c == Null, status = "Unimplemented int " <> int;
    Return[{solz, status}]];
If [Position["LC", "SC", "AC", "HC"], ics] == {},
    status = "Unimplemented ics " <> ics;
    Return[{solz, status}]];
n = 0; λ0 = 0; u0 = zv; elln = inpar[5];
If [Length[inictl] > 0, {λ0} = inictl];
If [Length[inista] > 0, u0 = inista]; qrsol = {True, True};
{v0, vr, q0, r0, Kdet, Keigs} = IncResVel[problem, method, geopar, 
matpar, fabpar, inipar, incpar, corpar, {λ0}, u0, rffield, options, qrsol];
If [status != " " , Return[{solz, status}]];
inisol = n, λ0, u0, v0, r0, q0, Kdet, Keigs, elln, 0, 0, "ref config"};
Return[{inisol, status}]];
```

**Figure 23.5.** The SetRefConfig module that initializes the solution process by setting up the reference control-state configuration.

Most of the logic of SetRefConfig is concerned with testing irrecoverable error conditions before the solution process is launched. For instance, the validity of the benchmark problem is checked by calling module ProblemInfo. Then it tests whether the specified methods are implemented. If no implementation errors are detected, it calls module IncResVel to initialize data at the stiffness-rate level. (Outputs of IncResVel are described in §23.4.8.) If IncResVel gives the all-clear, SetRefConfig completes setting up the first row of the solution table before return.

§23.4.3. Performing An Incremental Step

Module IncStepDriver advances the solution process over the nth incremental step. Its code is listed in Figure 23.6. The module is invoked as

```
{nextsol, status} = IncStepDriver[problem, method, geopar, matpar, fabpar, 
inipar, incpar, corpar, ctlpar, state, rffield, options, soldim];
```

Arguments problem through corpar, as well as rffield and options, are exactly the same as those of NonLinSolDriver. Those are described in §23.4.1. The new arguments are

- **ctlpar** A one-item list that stores the control parameter \( \lambda_n \) as \( \{ \lambda_n \} \).
- **state** The state vector \( \mathbf{u}_n \) obtained in the previous step.
- **soldim** A two-item integer list \( \{ n, m \} \) that provides dimensions of the solution table soltab described in §23.4.1. Here \( n \) is the number of steps performed so far (zero upon first entry to IncStepDriver), whereas \( m \) is the number of items in each soltab row. (Presently \( m = 12 \).) The latter is used to set up a null next-solution-row in case an error is detected.

The function returns are:

23–11
IncStepDriver[problem_, method_, geopar_, matpar_, fabpar_, inipar_, incpar_,
corpar_, ctlpar_, state_, rfload_, options_, soldim_] := Module[{probid,
int, A, c, w, n, m, neval, zv, numdof, nextsol, status = " ",
{n, m} = soldim;
int = method[[2]]; {A, c, w} = ButcherTable[int]; neval = Length[c];
probid = problem[[1]]; numdof = ProblemInfo[probid];
zv = Table[0, {numdof}]; nextsol = {++n, 0, zv, zv, zv, zv, 0, {0, 0}, 0, 0, 0, " "};
If [neval == 1,
  {nextsol, status} = IncStep1[problem, method, geopar, matpar, fabpar,
inipar, incpar, corpar, ctlpar, state, rfload, options, soldim];
If [neval == 2,
  {nextsol, status} = IncStep2[problem, method, geopar, matpar, fabpar,
inipar, incpar, corpar, ctlpar, state, rfload, options, soldim];
If [neval == 3,
  {nextsol, status} = IncStep3[problem, method, geopar, matpar, fabpar,
inipar, incpar, corpar, ctlpar, state, rfload, options, soldim];
If [neval == 4,
  {nextsol, status} = IncStep4[problem, method, geopar, matpar, fabpar,
inipar, incpar, corpar, ctlpar, state, rfload, options, soldim];
Return[{nextsol, status}];

Figure 23.6. The IncStepDriver module that advances the solution over one incremental step.

nextsol  The \((n+1)^{th}\) row (the next one) of the solution table. Set to all-null values if status
reports an error.

status   A textstring that specifies the solution status on exit. If not " " , an error has been
detected during this step and NonLinSolDriver will abort the solution process.

As can be observed in Figure 23.6, IncStepDriver is merely a cover routine that branches to
IncStep1, IncStep2, IncStep3 or IncStep4 according to the number of evaluations to be done
by the specified RK integrator. This number is indirectly retrieved by the call

\{A, c, w\} = ButcherTable[int]

in which textstring int specified the integration method as per Table 23.1, and \{A, c, w\} receive
the eponymous matrices and vectors of the Butcher table described in §21.5.4. Taking the length
of c gives the number of evaluations per step.\(^7\)

---

\(^7\) There is no need to check here for an unimplemented ERK method since that was previously done by SetRegConfig.
§23.4 SOLUTION ADVANCING MODULES

IncStep1[problem_, method_, geopar_, matpar_, fabpar_, inipar_, incpar_,
   corpar_, ctlpar_, state_, rfload_, options_, soldim_] := Module[
   {nmax, \[Lambda]max, umax, adapt, ell, ellmin, ellmax, acctol, icscal, eigsK,
     ics, int, nextsol, elln, qrsol, n, m, un, vl, vw, vr, f1, fw,
     q1, qw, r, Kdet, Keigs, \[Lambda]n, \[Delta]\[Lambda]n, \[Lambda]next, unext, rnext},
   {nmax, \[Lambda]max, umax, adapt, ell, ellmin, ellmax, acctol, icscal, eigsK} = incpar;
   {ics, int} = Take[method, 2]; {n, m} = soldim; \[Lambda]n = ctlpar[[1]]; un = state;
   nextsol = Table[Null, {m}]; elln = ell; qrsol = {True, False};
   {{v1, vr, q1, r, Kdet, Keigs}, status} = IncResVel[problem, method, geopar,
     matpar, fabpar, inipar, incpar, corpar, {\[Lambda]n}, un, rfload, options, qrsol];
   If [status != " ", Return[{nextsol, status}]];
   f1 = IncResFactor[ics, icscal, vl, q1]; \[Delta]\[Lambda]1 = elln/f1; vw = vl; qw = q1;
   fw = f1; \[Delta]\[Lambda]n = elln/fw; \[Lambda]next = \[Lambda]n + \[Delta]\[Lambda]n; unext = un + vw*\[Delta]\[Lambda]n; n++;
   rnext = TotalRes[problem, method, geopar, matpar, fabpar, inipar,
     incpar, corpar, {\[Lambda]next}, unext, rfload, options];
   nextsol = {n, \[Lambda]next, unext, vw, rnext, qw, rnext, Kdet, Keigs, elln, 0, 0, " "};
   Return[{nextsol, " "}]];

Figure 23.7. The IncStep1 module that advances the solution over one incremental step via Forward Euler. This is the only one-evaluation ERK scheme.

§23.4.4. Incremental Step Via Forward Euler

The code of the IncStep1 module is listed in Figure 23.7. The module is invoked as

   {nextsol, status} = IncStep1[problem, method, geopar, matpar, fabpar, inipar,
     incpar, corpar, ctlpar, state, rfload, options, soldim];

Arguments and returns are exactly the same as those of IncStepDriver, described in §23.4.3.

The only instance of an explicit, one-stage, ERK integrator is Forward Euler (FE). Accordingly a call to ButcherTable is not needed. On entry, the module computes the incremental velocity via IncResVel. It then applies the pertinent increment control strategy to advance to the next solution.

Before exit, the total force residual is evaluated via TotalRes, and returned to IncStepDriver via nextsol. This operation is typically missing from purely incremental FEM codes. It is done here for two reasons: to quantify the equilibrium drift error, thus eventually supporting a steplength adaptation process, and to initialize a corrective process if specified.

§23.4.5. Incremental Step Via Second Order RK

The code of the IncStep2 module is listed in Figure 23.8. The module is invoked as

   {nextsol, status} = IncStep2[problem, method, geopar, matpar, fabpar, inipar,
     incpar, corpar, ctlpar, state, rfload, options, soldim];

Arguments and returns are exactly the same as those of IncStepDriver, described in §23.4.3.

The solution is advanced over one incremental step via a two-evaluations-per-step, second-order ERK method. This is a one-parameter family. Two instances are implemented: the Explicit Midpoint Rule (EMR) and the Explicit Trapezoidal Rule (ETR). These are described in §21.4.1 and §21.4.2, respectively. The necessary method information is retrieved from ButcherTable. Two evaluations of the incremental velocity follow, and are combined to advance over the step. As in IncStep1, the total force residual of the advanced solution is evaluated and returned to IncStep via nextsol.

23–13
Chapter 23: PURELY INCREMENTAL METHODS: IMPLEMENTATION

IncStep2[problem_, method_, geopar_, matpar_, fabpar_, inipar_, incpar_, corpar_, ctlpar_, state_, rfload_, options_, soldim_] := Module[
{nmax, λmax, umax, adapt, ell, ellmin, ellmax, acctol, icscal, eigsK, 
nextsol, ics, int, elln, qrsol, A, c, w, n, m, un, c1, c2, 
w1, w2, A21, u1, u2, v1, v2, vw, vr, q1, q2, qw, r, 
λ1, λ2, λn, Δλn, Δλ1, Δλ2, Δλ3, f1, f2, f3, f4, 
Kdet, Keigs, ssadat, unext, λnext, rnext},
{nmax, λmax, umax, adapt, ell, ellmin, ellmax, acctol, icscal, eigsK} = incpar;
{ics, int} = Take[method, 2]; {n, m} = soldim;
λn = ctlpar[[1]]; un = state;
nextsol = Table[Null, {m}]; elln = ell; qrsol = {True, False};
{A, c, w} = ButcherTable[int]; {c1, c2} = c; {w1, w2} = w; A21 = A[[2, 1]];
{{v1, vr, q1, r, Kdet, Keigs}, status} = IncResVel[problem, method, geopar, 
matpar, fabpar, inipar, incpar, corpar, {λn}, un, rfload, options, qrsol];
If[status != " ", Return[{nextsol, status}]];
f1 = ICSFactor[ics, icscal, v1, q1]; Δλ1 = elln/f1;
λ2 = λn + c2*Δλ1; u2 = un + A21*v1*Δλ1;

{{v2, vr, q2, r, Kdet, Keigs}, status} = IncResVel[problem, method, geopar, 
matpar, fabpar, inipar, incpar, corpar, {λ2}, u2, rfload, options, qrsol];
If[status != " ", Return[{nextsol, status}]];
vw = w1 + v1 + w2 + v2; qw = w1 + q1 + w2 + q2;
fw = ICSFactor[ics, icscal, vw, qw]; ssdat = {elln, a, κ};
Δn = elln/Δn; λnext = λn + Δλn; unext = un + vw*Δλn; n++;
rnext = TotalRes[problem, method, geopar, matpar, fabpar, inipar, 
incpar, corpar, {λnext}, unext, rfload, options];
nextsol = {{n, λnext, unext, vw, rnext, qw, Kdet, Keigs, elln, 0, 0, " "}};
Return[{nextsol, " "}];

Figure 23.8. The IncStep2 module that advances over one incremental step via a two-
evaluations, second-order ERK. Two instances: EMR and ETR, are implemented.

§23.4.6. Incremental Step Via Third Order RK

The code of the IncStep3 module is listed in Figure 23.9. The module is invoked as

{nextsol, status} = IncStep3[problem, method, geopar, matpar, fabpar, inipar, 
incpar, corpar, ctlpar, state, rfload, options, soldim];

Arguments and returns are exactly the same as those of IncStepDriver, described in §23.4.3.

The solution is advanced over one incremental step via a three-evaluation-per-step, third-order RK
scheme. This is a multiparameter family. The two implemented instances are called RK3R (R for
Ralston) and RK3H (H for Heun). These are described in Table 23.1, which provides references.

The necessary information is retrieved from ButcherTable. Three evaluations of the incremental
velocity follow, and are combined to advance over the step. As in IncStep1 and IncStep2 the
total residual of the advanced solution is evaluated and returned to IncStep via nextsol.

§23.4.7. Incremental Step Via Fourth Order RK

The code of the IncStep4 module is listed in Figure 23.10. The module is invoked as

{nextsol, status} = IncStep4[problem, method, geopar, matpar, fabpar, inipar, 
incpar, corpar, ctlpar, state, rfload, options, soldim];

Arguments and returns are exactly the same as those of IncStepDriver, described in §23.4.3.

The solution is advanced over one incremental step via an explicit, four-evaluations-per-step, fourth-
order RK method. This is a multiparameter family. The two implemented instances are called RK4C
§23.4 SOLUTION ADVANCING MODULES

The IncStep3 module that advances over an incremental step via a three-evaluations, third-order ERK. Two instances: RK3R and RK3H, are implemented.

(C for Classical) and RK4K (K for Kutta). These are described in Table 23.1, which provides references. The necessary method information is retrieved from ButcherTable. Four evaluations of the incremental velocity follow, and are combined to advance over the step. As in IncStep1, IncStep2 and IncStep3, the total force residual of the advanced solution is evaluated and returned to IncStepDriver via nextsol.

§23.4.8. Incremental and Residual Velocities

Module IncResVel evaluates either the incremental velocity vector \( \mathbf{v}_q = \mathbf{K}^{-1} \mathbf{q} \), the residual velocity vector \( \mathbf{v}_r = -\mathbf{K}^{-1} \mathbf{r} \), or both. The module is listed in Figure 23.11. It is invoked as

\[
\{vq, vr, q, r, Kdet, Keigs, status\} = \text{IncResVel}[\text{problem, method, geopar, matpar, fabpar, inipar, incpar, corpar, } \lambda n, \text{un, rfload, options, qrsol}];
\]

Arguments are identical to those of the IncStepx modules, with the following exemptions.

- \text{ctlpar} Control parameter value at the current ERK evaluation point
- \text{state} State vector at the current ERK evaluation point

Both of these may differ from the eponymous arguments to the IncStepx modules if a higher order ERK integrator is used, if IncResVel is called at an intermediate evaluation point.
IncStep4[problem_,method_,geopar_,matpar_,fabpar_,inipar_,incpar_,
corpar_,ctlpar_,state_,rfload_,options_,soldim_]:=Module[
{nmax,λmax,umax,adapt,ell,ellmin,ellmax,acctol,icscal,eigsK,
ics,int,nextsol,elln,qrsol,A,c,w,n,m,un,c1,c2,c3,c4,
w1,w2,w3,w4,A21,A31,A32,A41,A42,A43,u1,u2,u3,u4,v1,v2,v3,v4,
wv,vr,vq,q,vw,qw,r,λ1,λ2,λ3,λ4,λn,Δλn,Δλ1,Δλ2,Δλ3,
λ1,λ2,λ3,λ4,Δλn,Δλ1,Δλ2,Δλ3,
f1,f2,f3,f4,fw,Kdet,Keigs,unext,λnext,rnext},
{nmax,λmax,umax,adapt,ell,ellmin,ellmax,acctol,icscal,eigsK}=incpar;
{ics,int}=Take[method,2]; {n,m}=soldim;
λn=ctlpar[[1]]; un=state;nextsol=Table[Null,{m}]; elln=ell; qrsol={True,False};
{A,c,w}=ButcherTable[int]; {c1,c2,c3,c4}=c; {w1,w2,w3,w4}=w;
A21=A[[2,1]]; A31=A[[3,1]]; A32=A[[3,2]]; A41=A[[4,1]]; A42=A[[4,2]]; A43=A[[4,3]];{(v1,vr,vq,q1,r,Kdet,Keigs),status]=IncResVel[problem,method,geopar,
matpar,fabpar,inipar,incpar,corpar,λn,un,rfload,options,qrsol];
If [status!=" ",Return[{nextsol,status}]];
f1=ICSFactor[ics,icscal,v1,q1]; λn+c2*Δλ1; u2=un+A21*v1*Δλ1;
{(v2,vr,vq,q2,r,Kdet,Keigs),status]=IncResVel[problem,method,geopar,
matpar,fabpar,inipar,incpar,corpar,λ2,u2,rfload,options,qrsol];
If [status!=" ",Return[{nextsol,status}]];
f2=ICSFactor[ics,icscal,v2,q2]; λn+c3*Δλ2; u3=un+(A31*v1+A32*v2)*Δλ2;
{(v3,vr,vq,q3,r,Kdet,Keigs),status]=IncResVel[problem,method,geopar,
matpar,fabpar,inipar,incpar,corpar,λ3,u3,rfload,options,qrsol];
If [status!=" ",Return[{nextsol,status}]];
f3=ICSFactor[ics,icscal,v3,q3]; λn+c4*Δλ3; u4=un+(A41*v1+A42*v2+A43*v3)*Δλ3;
{(v4,vr,vq,q4,r,Kdet,Keigs),status]=IncResVel[problem,method,geopar,
matpar,fabpar,inipar,incpar,corpar,λ4,u4,rfload,options,qrsol];
If [status!=" ",Return[{nextsol,status}]];
wv=w1*v1+w2*v2+w3*v3; qw=w1*q1+w2*q2+w3*q3;
fw=ICSFactor[ics,icscal,vw,qw]; ssdat={elln,a,κ};
λn=elln/fw; λnext=λn+Δλn; unext=un+vw*Δλn; n++;
rnext=TotalRes[problem,method,geopar,matpar,fabpar,inipar,incpar,corpar,λnext,unext,rfload,options];
nextsol={n,λnext,unext,vw,rnext,qw,Kdet,Keigs,elln,0,0," "};
Return[{nextsol," "}]];

Figure 23.10. The IncStep4 module that advances over one incremental step via a four-evaluations, fourth-order ERK. Two instances: RK4C and RK4K, are implemented.

Argument soldim is omitted as it is not needed. It is replaced by qrsol A list of two logical flags supplied as {qsol,rsol}, which specify what to do:
If qsol is True, evaluate \(v_q = K^{-1}q\), else omit.
If rsol is True, evaluate \(v_r = -K^{-1}r\), else omit.
If both qsol and rsol are False, an error condition is flagged since IncResVel cannot be called to do nothing.

The function returns:

vq Incremental velocity vector \(v_q\) if computed, else {}
vr Residual velocity vector \(v_r\) if computed, else {}
q Incremental load vector \(q\) if computed, else {}
r Total residual vector \(r\) if computed, else {}
Kdet Tangent stiffness matrix determinant

23–16
§23.4 SOLUTION ADVANCING MODULES

IncResVel[problem_, method_, geopar_, matpar_, fabpar_, inipar_, incpar_,
corpar_, ctlpar_, state_, rfload_, options_, qrsol_] := Module[
{K, Kdet, Knorm, Knill, Kcinv, Keigs, ev, evmin, evmax, numer=options[1],
epsing=10.^-12, epsill=10.^-6, qsol=qrsol[[1]], rsol=qrsol[[2]],
Null6=Table[Null,{6}], q={}, r={}, vq={}, vr={}, qmr,status=" "},
K=TanStiff[problem, method, geopar, matpar, fabpar, inipar, incpar, corpar, ctlpar, state, rfload, options];
If [K==Null, status="Error in K eval"; Return[{Null6,status}]];
If ![qsol&&!rsol, status="qsol&rsol False"; Return[{Null6,status}]];
Kdet=Det[K]; If ![numer, Kdet=Simplify[Kdet]; Keigs={}];
If ![numer&&Kdet==0, status="Exactly singular K"; Return[{Null6,status}]];
If [numer,
Knorm=MatrixFrobNorm[K];
If [Knorm==0, status="Null K"; Return[{Null6,status}]];
ev=Abs[Eigenvalues[N[K]]];
evmax=Max[Max[ev],epsing*Knorm]; evmin=Min[ev];
Keigs={evmax, evmin}; Kcinv=evmin/evmax;
If [Kcinv<=epsing, status="Singular K"; Return[{Null6,status}]];
If [Kcinv<=epsill, status="Warning: ill-cond K";]
];
If [qsol, q=IncLoad[problem, method, geopar, matpar, fabpar, inipar, incpar, corpar, ctlpar, state, rfload, options]];
If [rsol, r=TotalRes[problem, method, geopar, matpar, fabpar, inipar, incpar, corpar, ctlpar, state, rfload, options]];
If [qsol&&rsol, qmr=Transpose[{q,-r}];
{vq,vr}=Transpose[LinearSolve[K, qmr]];
If [qsol&&!rsol, vq=LinearSolve[K,q]];
If [rsol&&!qsol, vr=LinearSolve[K,-r]];
Return[{{vq,vr,q,r,Kdet,Keigs},status}]];

Figure 23.11. The IncResVel module that evaluates either the incremental velocity vector
\(v_q = K^{-1} q\), the residual velocity vector \(v_r = K^{-1} r\), or both.

Keigs  If flag eigsK in incpar is True, the maximum and minimum eigenvalues of the
tangent stiffness matrix as a two-item list. Else it returns the empty list \{

status  Textstring reporting status on exit

On entry, the stiffness matrix \(K\) is evaluated at the given control-state values using module
TanStiff. If an evaluation error is detected, IncResVel exits while reporting the error in status.
Next, the singularity or ill-condition of \(K\) is tested if the numer flag in options is True, which is
the usual case. If a singularity is detected, it tries to get away from it by calling SingularityFix.\(^8\)
If the singularity is deemed irrecoverable, IncResVel exits again with an appropriate status.
The remaining logic depends on the logical flags in qrsol. If qsol is True, the incremental load
vector \(q\) is evaluated, and the velocity vector \(v_q = K^{-1} q\) computed using the built-in module
LinearSolve. If rsol is True, the total residual \(r\) is evaluated and \(v_r = -K^{-1} r\) computed similarly.
If both flags are True, the two velocities are concurrently evaluated by calling LinearSolve
with two right hand sides: \(q\) and \(-r\). The results are returned as \(v_q, v_r, q, r, Kdet, Keigs, status\)
as described above. Vector quantities that are not computed return as empty lists.

\(^8\) This evasive maneuver is not yet implemented, pending tests.
§23.5. Problem Interface Modules

The problem interface modules grouped in this Section evaluate the tangent stiffness matrix, incremental load vector and total residual vector, at a given control-state pair. They serve as interfaces between the solution-advancing modules described in §23.4 and the benchmark problems covered in Chapter 24.

§23.5.1. Total Residual Interface

The code of the TotalRes interface module is listed in Figure 23.12. The module is invoked as

\[
\text{r} = \text{TotalRes}[	ext{problem, method, geopar, matpar, fabpar, inipar, incpar, corpar, ctlpar, state, rfload, options}];
\]

The arguments are the same as those of IncStep1 through IncStep4, with the following exceptions. Argument \text{soldim} is omitted as it is not needed, and

\begin{itemize}
  \item \text{ctlpar} \quad \text{Control parameter value at the current ERK evaluation point}
  \item \text{state} \quad \text{State vector at the current ERK evaluation point}
\end{itemize}

The function returns

\[
r \quad \text{Total residual vector. If an error is detected (e.g., the specified benchmark problem is not implemented) the module returns Null.}^{9}
\]

On entry TotalRes branches as per the problem benchmark identifier. It packs the geometric, material, fabrication and initialization properties required for that problem into a list called XXprop, where XX is the problem identifier. It calls the appropriate residual module and returns that vector.

---

\[9\] That kind of error is precluded if TotalRes is driven by NonLinSolDriver via the solution-advancing modules, because SetRefConfig checks for that event. But it may happen if TotalRes is called from a lower level harness driver during “bottom up” testing.
§23.5 PROBLEM INTERFACE MODULES

The code of the TanStiff interface module is listed in Figure 23.13. The module is invoked as

\[
K = \text{TanStiff}[\text{problem}, \text{method}, \text{geopar}, \text{matpar}, \text{fabpar}, \text{inipar}, \text{incpar}, \\
\text{corpar}, \text{ctlpar}, \text{state}, \text{rfload}, \text{options}];
\]

The arguments are the same as those of IncStep1 through IncStep4, with the following exceptions. Argument soldim is omitted as it is not needed, and

- \text{ctlpar} \quad \text{Control parameter value at the current ERK evaluation point}
- \text{state} \quad \text{State vector at the current ERK evaluation point}

The function returns

- \( K \) \quad \text{Tangent stiffness matrix. If an error is detected (e.g., the specified benchmark problem is not implemented) the module returns Null. (Cf. footnote for TotalRes.)}

On entry TanStiff branches according to the problem benchmark identifier. It packs the geometric, material and fabrication parameters required for the problem, along with the problem version identifier, and calls the appropriate stiffness module.

§23.5.3. Incremental Load Interface

The code of the IncLoad interface module is listed in Figure 23.14. The module is invoked as

\[
q = \text{IncLoad}[\text{problem}, \text{method}, \text{geopar}, \text{matpar}, \text{fabpar}, \text{inipar}, \text{incpar}, \\
\text{corpar}, \text{ctlpar}, \text{state}, \text{rfload}, \text{options}];
\]

The arguments are the same as those of IncStep1 through IncStep4, with the following exceptions. Argument soldim is omitted as it is not needed, and

- \text{ctlpar} \quad \text{Control parameter value at the current ERK evaluation point}
- \text{state} \quad \text{State vector at the current ERK evaluation point}

The function returns

- \( q \) \quad \text{Incremental load vector. If an error is detected (e.g., the specified benchmark problem is not implemented) the module returns Null. (Cf. footnote for TotalRes.)}
IncLoad[problem_, method_, geopar_, matpar_, fabpar_, inipar_, incpar_,
corpar_, ctipar_, state_, rfload_, options_]:=Module[
{probid=problem[[1]], prbvar=" ", λ, numer, q, modnam="IncLoad:"},
If [Length[problem]>1, prbvar=problem[[2]]];
numer=options[[1]]; λ=ctiplar[[1]];
If [probid=="MT",
MTprop=PackPropOfMisesTruss[prbvar, geopar, matpar, fabpar, inipar];
q=IncLoadOfMisesTruss[MTprop, λ, state, rfload, numer];
Return[q]];
If [probid=="CG",
CGprop=PackPropOfCircleGame[prbvar, geopar, matpar, fabpar, inipar];
q=IncLoadOfCircleGame[CGprop, λ, state, rfload, numer];
Return[q]];
Print[modnam," problem ",probid," not implemented"];
Return[Null]];

Figure 23.14. The IncLoad interface module that computes and returns the
incremental load vector at the present control-state pair.

On entry, IncLoad branches according to the problem benchmark identifier. It packs the geometric,
material and fabrication parameters required for the problem, along with the problem version
identifier, and calls the appropriate incremental load module.

§23.6. Solution Display Modules

Modules described in this section can be used to print and plot information stored in the solution
table, or derived information. Compared to the preceding ones, they have more “industrial strength”
flavor since flexibility in presenting results is important in an educational framework. Thus some
of the fancy display features of their ancestor Fortran programs (1970-1980s) are retained.

Print or plot displays usually involve only part of the solution table, especially in problems with
large number of DOF. Furthermore the information may be condensed; for example by printing or
plotting the norm of a vector instead of its components. Or printing the solution over a restricted step
range. To achieve such reductions it is necessary to understand how the solution data is symbolically
accessed through text labels. This is summarized in three Tables.

Table 23.6 lists defaults labels for all items in the solution table. By “default” is meant that they are
preset by GeNoBe on start. For example, if the problem has three DOF, the state vector components
are labeled "u1", "u2", and "u3" by default.

It is possible to replace some default labels by user-defined ones. In the 3-DOF case, "u1", "u2",
and "u3" could be replaced, for example, by "uX", "uY", and "θ". Labels amenable to replacement
are listed in Table 23.7.

It is also permissible to assign derived labels to request printing functions such as vector norms
and stiffness condition number. Those values are not stored in the solution table, but may be
obtained through auxiliary computations. For example, to output the 2-norm (Euclidean norm) of
the residual, namely ||r||2 = ||r|| = +√ r^TR, one specifies the label "||r||". Norms of vectors
u, v, r and q may be thus requested. In addition, the spectral condition number of the tangent
stiffness matrix may be printed with "Kcond" if its max/min eigenvalues have been recorded. A
complete list of derived labels is given in Table 23.8.
Table 23.6. Solution Table Default Labels

<table>
<thead>
<tr>
<th>#</th>
<th>Table item</th>
<th>Default labels *</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>n</td>
<td>&quot;step&quot;</td>
<td>Step number $n$</td>
</tr>
<tr>
<td>2</td>
<td>$\lambda$</td>
<td>&quot;(\lambda)&quot;</td>
<td>Control parameter $\lambda_n$</td>
</tr>
<tr>
<td>3</td>
<td>u</td>
<td>{&quot;u1&quot;,&quot;u2&quot;,&quot;uN&quot;}</td>
<td>State vector $\mathbf{u}_n$ †</td>
</tr>
<tr>
<td>4</td>
<td>v</td>
<td>{&quot;v1&quot;,&quot;v2&quot;,&quot;vN&quot;}</td>
<td>Incremental velocity vector $\mathbf{v}_q$ †</td>
</tr>
<tr>
<td>5</td>
<td>r</td>
<td>{&quot;r1&quot;,&quot;r2&quot;,&quot;rN&quot;}</td>
<td>Force residual vector $\mathbf{r}_q$ †</td>
</tr>
<tr>
<td>6</td>
<td>q</td>
<td>{&quot;q1&quot;,&quot;q2&quot;,&quot;qN&quot;}</td>
<td>Incremental load vector $\mathbf{q}_n$ †</td>
</tr>
<tr>
<td>7</td>
<td>Kdet</td>
<td>&quot;</td>
<td>K</td>
</tr>
<tr>
<td>8</td>
<td>Keigs</td>
<td>{&quot;Kmax&quot;,&quot;Kmin&quot;}</td>
<td>Max &amp; min eigenvalues of $\mathbf{K}$</td>
</tr>
<tr>
<td>9</td>
<td>elln</td>
<td>&quot;ell&quot;</td>
<td>Steplength $\ell_n$</td>
</tr>
<tr>
<td>10</td>
<td>{ ... }</td>
<td></td>
<td>Placeholder</td>
</tr>
<tr>
<td>11</td>
<td>{ ... }</td>
<td></td>
<td>Placeholder</td>
</tr>
<tr>
<td>12</td>
<td>remark</td>
<td>&quot;rem&quot;</td>
<td>Informative message</td>
</tr>
</tbody>
</table>

* No default labels for items 10-11, which are currently placeholders
† $N$ is the number of degrees of freedom (DOF)

Table 23.7. Solution Table Replacement Labels

<table>
<thead>
<tr>
<th>#</th>
<th>Table item</th>
<th>Possible?</th>
<th>Example † (if possible)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>n</td>
<td>yes</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>$\lambda$</td>
<td>yes</td>
<td>$\lambda_{\text{rep}}=\chi$  or $\mathbf{u}_{\text{rep}}={\chi}$ ‡</td>
</tr>
<tr>
<td>3</td>
<td>u</td>
<td>yes</td>
<td>$\mathbf{u}_{\text{rep}}={uX_1,uY_1,u\theta_1}$</td>
</tr>
<tr>
<td>4</td>
<td>v</td>
<td>yes</td>
<td>$v_{\text{rep}}={vX_1,vY_1,v\theta_1}$</td>
</tr>
<tr>
<td>5</td>
<td>r</td>
<td>yes</td>
<td>$\mathbf{r}_{\text{rep}}={rX_1,rY_1,r\theta_1}$</td>
</tr>
<tr>
<td>6</td>
<td>q</td>
<td>yes</td>
<td>$\mathbf{q}_{\text{rep}}={qX_1,qY_1,q\theta_1}$</td>
</tr>
<tr>
<td>7</td>
<td>Kdet</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>Keigs</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>elln</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>{ ... }</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>{ ... }</td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>remark</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

* Replacement labels are only possible for the indicated items.
† Example assumes three DOF.
‡ Both forms work.

§23.6.1. Solution Table Printing

The module that prints solution table information is listed in Figure 23.15. It is invoked as

```
PrintSolTable[problem,method,soltab,replab,what,digits,srange,choptol];
```

The arguments are

- problem Benchmark problem identifier; see Table 23.3.
Table 23.8. Solution Table Derived Labels

<table>
<thead>
<tr>
<th># Table item</th>
<th>Derived label*</th>
<th>Description ($N$ is number of DOF)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 n</td>
<td>$||u||$</td>
<td>Step number $n$</td>
</tr>
<tr>
<td>2 $\lambda$</td>
<td>$||v||$</td>
<td>Control parameter $\lambda_n$</td>
</tr>
<tr>
<td>3 u</td>
<td>$||u||$</td>
<td>State vector 2-norm $|u_n|$</td>
</tr>
<tr>
<td></td>
<td>$||u||\infty$</td>
<td>State vector infinity-norm $|u_n|_\infty,$ †</td>
</tr>
<tr>
<td>4 v</td>
<td>$||r||$</td>
<td>Incremental velocity vector 2-norm $|v_n|$</td>
</tr>
<tr>
<td></td>
<td>$||r||\infty$</td>
<td>Incremental velocity vector infinity-norm $|v_n|_\infty,$ †</td>
</tr>
<tr>
<td>5 r</td>
<td>$||q||$</td>
<td>Force residual vector 2-norm $|r_n|$</td>
</tr>
<tr>
<td></td>
<td>$||q||\infty$</td>
<td>Force residual vector infinity-norm $|r_n|_\infty,$ †</td>
</tr>
<tr>
<td>6 q</td>
<td>$||q||$</td>
<td>Incremental load vector 2-norm $|q_n|$</td>
</tr>
<tr>
<td></td>
<td>$||q||\infty$</td>
<td>Incremental load vector infinity-norm $|q_n|_\infty,$ †</td>
</tr>
<tr>
<td>7 $K_{\text{det}}$</td>
<td></td>
<td>Determinant of $K$</td>
</tr>
<tr>
<td>8 Keigs</td>
<td>&quot;Kcond&quot;</td>
<td>Spectral condition number of $K$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(ratio of largest to smallest eigenvalue)</td>
</tr>
<tr>
<td>9 elfn</td>
<td>$\ell_n$</td>
<td>Steplength $\ell_n$</td>
</tr>
<tr>
<td>10 { ... }</td>
<td></td>
<td>Placeholder</td>
</tr>
<tr>
<td>11 { ... }</td>
<td></td>
<td>Placeholder</td>
</tr>
<tr>
<td>12 remark</td>
<td></td>
<td>Informative message</td>
</tr>
</tbody>
</table>

* Derived labels are only available for the indicated items; else left blank.
† For the $\|\|\|\infty$ norms, the infinity symbol is formed with two lower case “o”.

method  Solution method identifier; see Tables 23.1 and 23.2.
soltab  Solution table; see Table 23.5.
replab  A list of textstrings to be assigned as replacement labels to the solution data indicated in Table 23.7. If replab is an empty list: $\{}$, all default labels are used. If nonempty, it must be a list of exactly five sublists:

$\{\lambda_{\text{rep}}, \text{urep}, \text{vrep}, \text{rrep}, \text{qrep}\}$

$\lambda_{\text{rep}}$ is a 1-item sublist storing a replacement label for the control parameter $\lambda$, whereas urep, vrep, rrep, qrep are sublists of replacement labels for the state, incremental velocity, residual and incremental load vectors, respectively. An empty sublist skips replacement. If nonempty, a sublist must have labels for all DOF. For example, to relabel the state and residual vector components of a 2-DOF problem as $u_X, u_Y, r_X$ and $r_Y$, and the control parameter as $\chi$, set

replab=$\{"\chi",\"uX","uY\"\},\{\},\{"rX","rY\"\},\{\}\};$

The empty sublists specify that the default labels for vectors $v_q$ and $q$ are not be changed. It is also OK to specify the control label directly as a textstring, so that

replab=$\{"\chi","uX","uY\"\},\{\},\{"rX","rY\"\},\{\}\};$

also works. Replacement labels make sense only if that particular data is to be printed or plotted. (replab is reused in the plotting module described in §23.6.2.)
what A list of labels that specify what is to be printed. If some have been replaced through replab, the replacement labels should be used.

The configuration of what is best illustrated through an example. Suppose that for a 2-DOF problem the replacement labels are (as in the replab example)

\[
\text{replab=\{"\chi\\", "uX","uY"},\{\},{"rX","rY"},\{\}\};}
\]

To print step, control parameter, state vector components (both), Y-residual vector component, 2-norm of incremental velocity vector, stiffness determinant, increment steplength and remarks, specify

\[
\text{what=\{"step","\chi","uX","uY","rY","||v||","|K|","ell","rem"\}}
\]

These quantities will be printed in the order in which they appear in this list.

digits This argument may be used to enter formatting specification for table columns. If an empty list: \{\}, default formats are used. If nonempty, it must be a 12-item list of individual integers or integer-pair sublists with the configuration

\[
\text{(dn,\{dλ,fλ\},\{du, fu\},\{dv,fv\},\{dr,fr\},\{dq,fq\},
\{dK,fK\},\{de,fe\},\{dl,fl\},0,0,drem)}
\]

An individual integer, generically called d, is entered for columns 1 (step number), 10 and 11 (placeholders), and 12 (remark). Here dn is the number of digits printed for step and drem the number of characters for remark. Values dn=4 and drem=8 are recommended; those being the actual defaults. If zero, that column is not printed, which is the case for placeholders.

An integer pair is generically called \{(d,f)\}. These apply to items output in floating-point format. Here d is the number of significant digits shown whereas f is the number of digits after the decimal point; the latter being padded with zeros if necessary. For example, consider printing \(\pi = 3.141592654\ldots\). If \{(d,f)\}={6,5} the output is 3.14159, whereas if \{(d,f)\}={3,5} it will print as 3.14000. If both integers are zero, that column is not printed. Relevant table content includes columns 2 (control parameter \(\lambda\)) through 9 (increment length \(\ell\)). In the display of a vector, an integer-pair specification applies to all components; e.g., \{dr,fr\} is used for all residual vector entries, as well as residual vector norms. The default format is

\[
\{4,\{5,4\},\{5,5\},\{4,3\},\{4,4\},\{5,4\},\{4,4\},\{4,3\},\{3,3\},0,0,8\}
\]

This works fine for most benchmark problems, but may have to be adjusted for other problems depending on physical units and/or scaling.

srange Step print range. If an empty list, all of the steps stored in the solution table are printed. If nonempty it may be a list of one, two or three integers:

\[
\{\text{nbeg}\}, \{\text{nbeg,nend}\}, \text{or } \{\text{nbeg,nend,ninc}\}.
\]

In the first form, print all steps in the solution table starting at nbeg. In the second form, print steps nbeg through nend. In the third form, print steps nbeg through nend with increment ninc. The reader is warned that step numbering starts at zero, which is associated with the reference configuration. For instance, suppose that the solution table stores 11 steps, numbered 0 through 10. The specification srange=\{7\} will print four steps: 7,8,9,10, while srange=\{0,5\} will print six steps: 0,1,2,3,4,5. Specifying srange=\{0,10,3\} will print four steps: 0,3,6,9, since 12 is outside range. To print step 5 only, set srange=\{5,5\}.
PrintSolTable[problem_, method_, soltab_, replab_, what_, digits_, srange_,
choptol_] := Module[{nsteps, numdof, λrep, urep, vrep, rrep, qrep, i, j, k, n,
specs, ulabs, soln, solni, val, step, λ, Kdet, Kmax, Kmin, ell,rem,
TF, TFB, thead, probid = "", prbvar = "", ics, int, cmet = "",
nprows, npcols, nbeg, nend, ninc, sralen, lab, pos, d, dig = digits, defdig,
zdigits = 0 digits for ", title = "Computed solution for problem: ",
modnam = "PrintSolTable:", pskip = "", print skipped.", stab},
TF[text_] := StyleForm[text, FontFamily -> "Courier",FontSize -> 10,
FontWeight -> "Plain", FontSlant -> "Plain"];
TFB[text_] := StyleForm[text, FontFamily -> "Courier",FontSize -> 10,
FontWeight -> "Bold", FontSlant -> "Plain"];
nsteps = nend = Length[soltab]; nbeg = ninc = 1; sralen = Length[srange];
If[nsteps <= 0, Print[modnam, " soltab empty", pskip];
Return[]]; numdof = Length[soltab[[1, 3]]];
λrep = urep = vrep = rrep = qrep = {};
If[Length[replab]>0, {λrep, urep, vrep, rrep, qrep} = replab];
tablab = SolTableLabels[numdof, λrep, urep, vrep, rrep, qrep];
defdig = {4, {5, 4}, {5, 5}, {4, 3}, {4, 4}, {5, 4}, {4, 4}, {3, 3}, 0, 0, 8};
If[Length[dig] == 0, dig = defdig]; strlen = Length[srange];
{specs, ulabs} = SolTablePOPSpecs[numdof, tablab, what, dig];
npcols = Length[specs]; If[npcols == 0, Print[modnam, " empty print specs", pskip];
Return[]];
thead = TF/@Flatten[Table[specs[[k, 1]], {k, 1, npcols}]]; If[Length[ulabs] > 0, Print[modnam, " warning - undefined label(s) ", ulabs, " wont be printed"]];
probid = problem[[1]]; If[Length[problem] > 1, prbvar = problem[[2]];
{ics, int} = Take[method, 2]; If[Length[method] > 3, cmet = method[[3]]];
piorpc = If[cmet != "", ", pred: ", ", int: "];
title = SequenceForm[TF[title], TFB[probid], ", TFB[prbvar],
TF[" with ics: ", TFB[ics], TFB[piorpc], TFB[int]]];
If[cmet != "", title = SequenceForm[title, TF[", corr: ", TFB[cmet]]];
Print[title]; Print[
TableForm[stab, TableAlignments -> {Right},
TableDirections -> {Column, Row}, TableSpacing -> {0, 1},
TableHeadings -> {None, thead}]]; ClearAll[soln, stab]];

<table>
<thead>
<tr>
<th>Problem</th>
<th>Method</th>
<th>Solution Table</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 23.15. Module that prints the computed solution table, or selected portions thereof.

23–24
Some safeguards are implemented: if \( n_{\text{beg}} \) is negative, it is set to 0; if \( n_{\text{end}} \) exceeds the largest step recorded, it is set to that value; and if \( n_{\text{inc}} \) is zero or negative, it is set to 1. Finally, if \( n_{\text{beg}} \) exceeds \( n_{\text{end}} \), nothing is printed.

choptol A “chop tolerance” for floating-point printing. If the absolute value is less that choptol, it is set to exact zero. This may be used to clean up the output of quantities that take abnormally small values. If choptol is zero, no chopping occurs.

The module does not return any function value, since its output goes to the print channel.

§23.6.2. Solution Table 2D Plotting

The display module that generates 2D list plots from information stored in the computed solution table is listed in Figure 23.15. The plot is done through Mathematica’s ListPlot library function. The module can be invoked in two forms:

\[
\text{PlotSolTable}[\text{problem}, \text{method}, \text{soltab}, \text{replab}, \text{what}, \text{scale}, \text{logplot}, \text{range}, \text{aspect}, \text{imgsiz}, \text{color}, \text{showit}];
\]

\[
pobj=\text{PlotSolTable}[\text{problem}, \text{method}, \text{soltab}, \text{replab}, \text{what}, \text{scale}, \text{logplot}, \text{range}, \text{aspect}, \text{imgsiz}, \text{color}, \text{showit}];
\]

The first form writes the plot to the standard display channel.\(^{10}\) To get it, set argument showit (a logical flag) to True. The second form does not produce a plot but returns the graphics object as function value. This can be subsequently combined with other graphic objects to produce a multiple plot, which can be displayed via Show. It is specified by setting showit to False.

The arguments are:

- **problem** Benchmark problem identifier; see Table 23.3.
- **method** Solution method identifier; see Tables 23.1 and 23.2.
- **soltab** Solution table; see Table 23.5.
- **replab** An optional list of user-defined replacement labels. See description in §23.6.1.
- **what** A list of two labels: \( \{xvar, yvar\} \) that specifies the quantities to be plotted along the \( x \) and \( y \) axes, respectively. These may be either the original labels listed in Table 23.9 (some of which may be overwritten with user-defined labels), or the derived labels listed in Table 23.10. The use of this argument is better illustrated with examples:
  - \( \text{what}=\{"u2", "\lambda" \} \): Plot second state vector entry \( u_2 \) (along horizontal axis) versus control parameter \( \lambda \) (along vertical axis). This is a typical load-deflection plot.
  - \( \text{what}=\{"step", "|r|" \} \): Plot the residual 2-norm vector \( |r| \) (along vertical axis) versus step \( n \) (along horizontal axis). This is a typical history plot.
  - \( \text{what}=\{"u1", "u2" \} \): Plot first state vector entry \( u_1 \) (along horizontal axis) versus second state vector entry \( u_2 \) (along vertical axis). This is a typical state plot.

If "u1" and "u2" have been replaced by "uX" and "uY", respectively, via replab, the first example must be changed to \( \text{what}=\{"uY", "\lambda" \} \) and the third one to \( \text{what}=\{"uX", "uY" \} \).

\(^{10}\) See function GetDisplayChannel in §23.7.2.
PlotSolTable[problem_, method_, soltab_, replab_, what_, scale_, logplot_, range_, aspect_, imgsiz_, color_, showit_]:= Module[
{nsteps, numdof, i, j, k, m, n, xsp, ysp, xvar=what[[1]], yvar=what[[2]], 
nlab, soln, xsc, ysc=1, sclen=Length[scale], prange=Automatic, 
λrep, urep, vrep, rrep, qrep, specs, ulabs, ralen=Length[range], val, 
title, joined, lincolor=RGBColor[0, 0, 0], probid="", prbvar="", 
ics, int, cmet=", TF, dfun, p, xv=Table[0, {12}], epslog=1/10.^10, 
modname="PlotSolTable:", pskip="", plot skipped.", 
TF[text_]:=StyleForm[text, FontFamily->"Courier", FontSize->9, 
FontWeight->"Plain", FontSlant->"Plain"]; 
nsteps=Length[soltab]; nlab=Length[replab]; m=Length[soltab[[1]]]; If [nsteps<=0, Print[modnam," soltab empty", plotmsg]; Return[]]; numdof=Length[soltab[[1,3]]]; λrep=urep=vrep=rrep=qrep={}; If [Length[replab]>0, {λrep, urep, vrep, rrep, qrep}=replab]; tablab=SolTableLabels[numdof, λrep, urep, vrep, rrep, qrep]; 
{specs, ulabs}=SolTablePOPSpecs[numdof, tablab, what, Table[0, {12}]]; If [Length[specs]!=2, Print[modnam," unknown label(s) ", ulabs, 
pskip]; Return[]]; {xsp, ysp}=specs; If [ralen==0&&range==All, prange=All]; If [ralen>0, prange=range]; If [sclen==1, {ysc}=scale]; If [sclen==2, {xsc, ysc}=scale]; 
xyv=Table[{0., 0.}, {nsteps}]; xpos=xsp[[2]]; ypos=ysp[[2]]; If [Length[xpos]>0, {ix, jx}=xpos, ix=xpos; jx=0]; For [n=1, n<=nsteps, n++, soln=soltab[[n,ix]]]; If [jx==0, xyv[[n,1]]=xsc*soln; Continue[]]; If [jx>0, xyv[[n,1]]=xsc*soln[[jx]]; Continue[]]; If [jx==-2, xyv[[n,1]]=xsc*Max[Abs[soln]]; Continue[]]; If [ix==3&&ix<=6, xyv[[n,1]]=xsc*Sqrt[soln*soln]; Continue[]]; {Kmax, Kmin}=soln; xyv[[n,1]]=xsc*(If [Kmin==0, 0, Kmax/Kmin]); If [Length[ypos]>0, {iy, jy}=ypos, iy=ypos; jy=0]; For [n=1, n<=nsteps, n++, soln=soltab[[n,iy]]]; If [jy==0, xyv[[n,2]]=ysc*soln; Continue[]]; If [jy>0, xyv[[n,2]]=ysc*soln[[jy]]; Continue[]]; If [jy==-2, xyv[[n,2]]=ysc*Max[Abs[soln]]; Continue[]]; If [iy==3&&iy<=6, xyv[[n,2]]=ysc*Sqrt[soln*soln]; Continue[]]; {Kmax, Kmin}=soln; xyv[[n,2]]=ysc*(If [Kmin==0, 0, Kmax/Kmin]); If [logplot, For [n=1, n<=nsteps, n++, 
val=Abs[N[xyv[[n,2]]]]; If [val<=epslog, val=epslog]; xyv[[n,2]]=Log[10,val]]]; probid=problem[[1]]; If [Length[problem]>1, prbvar=problem[[2]]]; 
{ics, int}=Take[method, 2]; If [Length[method]>=3, cmet=method[[3]]]; 
piorp=If [cmet!="", pred: "", int: "]; title=yvar<>" vs "<>xvar<>" for "<>probid<>prbvar"", ics: "<<
title<>"\n, corr: ">>cmet]; If [logplot, title="log"<>title]; If [color="Red", lincolor=RGBColor[1, 0, 0]]]; If [color="Green", lincolor=RGBColor[0,1,0]]; If [color="Blue", lincolor=RGBColor[0,0,1]]; joined=If [VersionNumber>=6.0, Joined->True, PlotJoined->True]; If [showit, 
p=ListPlot[xyv, joined, PlotRange->prange, Frame->True, 
AxesOrigin->{0,0}, ImageSize->imgsz, AspectRatio->aspect, 
PlotStyle->{AbsoluteThickness[2.0], lincolor}, 
PlotLabel->TF[title], DisplayFunction->Identity]; 
ClearAll[soln, xyv]; Return[p]]; 
ListPlot[xyv, joined, PlotRange->prange, Frame->True, 
AxesOrigin->{0,0}, ImageSize->imgsz, AspectRatio->aspect, 
PlotStyle->{AbsoluteThickness[2.0], lincolor}, 
PlotLabel->TF[title], DisplayFunction->GetDisplayFunction[]]; ClearAll[soln, xyv]; 

Figure 23.16. Module that generates list plots from information in the computed solution table.
### Table 23.9. Solution Table Plotting Labels

<table>
<thead>
<tr>
<th>#</th>
<th>Table item</th>
<th>Plot label</th>
<th>Quantity to be plotted</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>n</td>
<td>&quot;step&quot;</td>
<td>Step number n</td>
</tr>
<tr>
<td>2</td>
<td>λ</td>
<td>&quot;λ&quot;</td>
<td>Control parameter*</td>
</tr>
<tr>
<td>3</td>
<td>u</td>
<td>&quot;u1&quot;,&quot;u2&quot;,...</td>
<td>State vector entries*</td>
</tr>
<tr>
<td>4</td>
<td>v</td>
<td>&quot;v1&quot;,&quot;v2&quot;,...</td>
<td>Incremental velocity vector entries*</td>
</tr>
<tr>
<td>5</td>
<td>r</td>
<td>&quot;r1&quot;,&quot;r2&quot;,...</td>
<td>Residual vector entries*</td>
</tr>
<tr>
<td>6</td>
<td>q</td>
<td>&quot;q1&quot;,&quot;q2&quot;,...</td>
<td>Incremental load vector entries</td>
</tr>
<tr>
<td>7</td>
<td>Kdet</td>
<td>&quot;Kdet&quot;</td>
<td>Tangent stiffness determinant</td>
</tr>
<tr>
<td>8</td>
<td>Keigs</td>
<td>&quot;Keigmax&quot;,&quot;Keigmin&quot;</td>
<td>Max or min eigenvalue of stiffness matrix</td>
</tr>
<tr>
<td>9</td>
<td>elln</td>
<td>&quot;ell&quot;</td>
<td>Incremental steplength</td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td>Placeholder</td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
<td>Placeholder</td>
</tr>
<tr>
<td>12</td>
<td>remark</td>
<td></td>
<td>Informative message</td>
</tr>
</tbody>
</table>

* Default labels for control parameter, as well as state, incremental velocity, residual and incremental load vector entries, can be overwritten via `replab`; see Table 23.8.

### Table 23.10. Solution Table Derived Plotting Labels

<table>
<thead>
<tr>
<th>#</th>
<th>Table item</th>
<th>Plot label</th>
<th>Quantity to be plotted</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>u</td>
<td>&quot;</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>&quot;</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>v</td>
<td>&quot;</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>&quot;</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>r</td>
<td>&quot;</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>&quot;</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>q</td>
<td>&quot;</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>&quot;</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>Keigs</td>
<td>&quot;Kcond&quot;</td>
<td>Spectral condition number of stiffness matrix</td>
</tr>
</tbody>
</table>

* See previous table

**scale**

Argument that may be used to numerically scale the y-variable `yvar` (most common) the x-variable `xvar`, or both. To scale `yvar` by `ysc`, set `scale={ysc}`. To scale `xvar` by `xsc`, set `scale={xsc,1}`. To scale both, set `scale={xsc,ysc}`. If an empty list, no scaling is applied, which is equivalent to `scale={1,1}`. The chief use of scaling is “range equilibration” in multiple plots so all graphs are discernible in the same frame.

**logplot**

A logical flag. If `True`, the y-variable is modified by taking the log_{10} of its absolute values. If the absolute value is less than a threshold of $10^{-16}$ it is set to that threshold to avoid large negative logs. Occasionally useful for quantities that may vary across several orders of magnitude, such as `det(K)` or `||r||`. But note that sign information is lost. If set to `False`, no log is taken.
Chapter 23: PURELY INCREMENTAL METHODS: IMPLEMENTATION

SolTableLabels[numdof_, λrep_, urep_, vrep_, rrep_, qrep_] := Module[
{i, ii, λlab, ulab, vlab, rlab, qlab, labels},
ulab = vlab = rlab = qlab = Table[" ", {numdof}]; λlab = "λ";
For [i = 1, i <= numdof, i++, ii = ToString[i];
ulab[[i]] = "u" <> ii; vlab[[i]] = "v" <> ii;
rlab[[i]] = "r" <> ii; qlab[[i]] = "q" <> ii;
If [Length[λrep] == 0 & λrep != "", λlab = λrep];
If [Length[λrep] == 1, λlab = λrep[[1]]];
If [Length[urep] == numdof, ulab = urep];
If [Length[vrep] == numdof, vlab = vrep];
If [Length[rrep] == numdof, rlab = rrep];
If [Length[qrep] == numdof, qlab = qrep];
tablab = {"step", λlab, ulab, vlab, rlab, qlab, "|K|",
{"Kmax", "Kmin"}, "ell", "", "", "rem"};
Return[tablab];

SolTablePOPSpecs[numdof_, tablab_, what_, digits_] := Module[{i, j, j1, j2, jj, k, m, wi, d, wlen, derlab, dlen, dpos, tlen, tpos, lab, ulabs = {}, specs = {}},
derlab = {
{"||u||", {3, -1}}, {"||u||oo", {3, -2}}, {"||v||", {4, -1}}, {"||v||oo", {4, -2}},
{"||r||", {5, -1}}, {"||r||oo", {5, -2}}, {"||q||", {6, -1}}, {"||q||oo", {6, -2}},
{"Kcond", {8, -1}}}; dlen = Length[derlab]; wlen = Length[what];
For [i = 1, i <= wlen, i++, wi = what[[i]];
If [wi == {} || wi == " ", Continue[]];
tpos = Position[tablab, wi]; dpos = Position[derlab, wi];
tlen = Length[tpos]; dlen = Length[dpos];
If [tlen == 0 & dlen == 0, AppendTo[ulabs, wi]; Continue[]];
If [tlen == 0, m = Dimensions[tpos][[2]]; If [m == 1, ((j) = tpos; lab = tablab[[j]]; d = digits[[j]]; AppendTo[specs, {lab, j, d}]);
If [m == 2, ((j, k) = tpos; lab = tablab[[j, k]]; d = digits[[j]]; AppendTo[specs, {lab, {j, k}, d}]);
If [dlen == 0, j = dpos[[1, 1]]; {lab, jj} = derlab[[jj]]; {j1, j2} = jj;
d = digits[[j1]]; AppendTo[specs, {lab, jj, d}]];
};
Return[{specs, ulabs}];

Figure 23.17. Auxiliary modules that support printing and plotting.

range This argument can be used to control the plot range. It can take two forms:
A word: either All or Automatic, spelled as shown. Do not enclose in quotes.
A list of the form \{xrange, yrange\}, in which xrange and yrange specify the
plotting ranges for xvar and yvar, respectively.
If range is an empty list, Automatic is assumed. For details on the effect of these
various specifications, see the Mathematica documentation.

aspect Plot aspect ratio (y-dimension over x-dimension). Normally 0.5 to 1.

imgsz Horizontal plot dimension in points. Normally 300 to 400.

color Textstring that specifies color of the plot line. Legal ones are "Black", "Red",
"Green" and "Blue". If none of these, "Black" is assumed.

showit The logical flag that controls plot production. See description under calling se-
quences of PlotSolutionTable above. Setting it to False supports the preparation of multiple plots that are to appear in one frame. Usage examples can be found in the next Chapter.
§23.7 Utility Modules

§23.6.3. Solution Table 3D Plotting

Module Par3DPlotSolutionTable produces a 3D parametric plot from information stored in the solution table, using the step number as driving parameter. Its main use is to generate state-control plots that show two state vector components versus the control parameter. This module has not been implemented.

§23.6.4. Auxiliary Solution Table PrintPlot Modules

Modules SolTableLabels and SolTablePOPSpecs and provide auxiliary services to both PrintSolTable and PlotSolTable. Those modules are listed in Figure 23.17. They are only briefly described here, since they are only used internally. Their calling sequences are

\[
\text{tablab}=\text{SolTableLabels}\left[\text{numdof}, \text{\lambda rep}, \text{urep}, \text{vrep}, \text{rrep}, \text{qrep}\right]; \\
\{\text{specs}, \text{ulabs}\}=\text{SolTablePOPSpecs}\left[\text{numdof}, \text{tablab}, \text{what}, \text{digits}\right];
\]

SolTableLabels is used by PrintSolTable and PlotSolTable to set up solution table labels. It sets up default labels of Table 23.6. It then replaces those specified by the user in argument replab, which is supplied unpacked into \text{\lambda rep} through \text{qrep}. The final label set is returned in tablab.

SolTablePOPSpecs (in which “POP” stands for “Print Or Plot”) sets up a data structure that contains instructions for either printing the solution table, or for preparing a 2D plot. As inputs it receives the number of degrees of freedom numdof, the final label set tablab returned by SolTableLabels, the “what to print or plot” list in what, and the print format in digits. (If called by PlotSolTable, argument digits is only a placeholder.) The module proceeds to encode those print instructions into specs. Unmatched labels, if any are collected in ulabs; if all labels match, ulabs remains an empty list. Both specs and ulabs are returned as function value to the calling module.

§23.7. Utility Modules

Utility modules described in the following subsections are intended to provide miscellaneous information that can be retrieved by any other module. A utility module is always self-contained and occasionally can be simply implemented as a function. They are presented in alphabetical order.

§23.7.1. Butcher Table

Returns numerical data that implements the ERK integration schemes of Table 23.1. This module is listed in Figure 21.5 of Chapter 21, and described in §21.5.4.

§23.7.2. Plot Display Channel

This utility, listed in Figure 23.18, is implemented as a function (a module format is not needed). It is invoked as

\[
dfun=\text{GetDisplayChannel}[]
\]

This function takes no arguments. It returns the display function channel as per Mathematica version. If that version is less than 6.0, dfun is set to $DisplayFunction; else to Print.
GetDisplayFunction[] := If [$VersionNumber >= 6.0, Print, $DisplayFunction];

Figure 23.18. Function that returns the DisplayFunction channel name as per Mathematica version.

ICSFactor[ics_, icscal_, v_, q_] := Module[{sgn = Sign[q.v],
csc = icscal, λ scale = 0, vscale = 0, vv},
If [ics == "LC", Return [sgn]];  
If [Head[csc] == List && Length[csc] <= 0, csc = 1];
If [Length[csc] <= 0, vscale = csc];
If [Length[csc] == 1, vscale = csc[[1]]];
If [Length[csc] == 2, {λ scale, vscale} = csc];
If [Head[vscale] == List, vv = (vscale.v)^2, vv = (vscale*v)*(vscale*v)];
If [ics == "SC", Return [sgn*Sqrt[vv]]];
If [ics == "AC", Return [sgn*Sqrt[1+vv]]];
If [ics == "HC", Return [sgn*Sqrt[λ scale^2+vv]]];
Return [Null]];

Figure 23.19. Module that returns the factor $f$ associated with an Increment Constraint Strategy ICS.

§23.7.3. ICS Factor

Utility ICSFactor, listed in Figure 23.19, is used by incremental step modules such as IncStep1 through IncStep4 to get the factor $f$ used in the increment control strategy. The module is invoked as

$$f = \text{ICSFactor}[ics, icscal, v, q]$$

The arguments are

ics  Increment control strategy identifier. See Table 23.2
icscal Specifies scaling information for the computation of $f$. The goal is to homogenize physical units. Four possible configurations: a scalar, a blank list, a one-item list or a two-item list. specifying scaling coefficient for the control parameter and velocity vector. If a two-item list, the second item may be also a list. The code should be studied to understand how this scaling information is implemented.

v Incremental velocity vector
q Incremental load vector. Used for computing the sign of $q^Tv$

The function return is

f Factor $f$ to be used by the incremental step modules. If an input combination is not implemented, it returns Null.

23–30
§23.7 UTILITY MODULES

```math
\text{MatrixFrobNorm}[A_] := \text{Module}[[i,j,n=\text{Length}[A], Fnorm=0],
\text{If} [n<=0, \text{Return}[0]]; 
\text{For} [i=1,i<=n,i++, \text{Fnorm}+=A[[i]]\cdot A[[i]]]; 
\text{Return}[\text{Sqrt}[\text{Fnorm}/n]];
```

**Figure 23.20.** Module that returns the Frobenius norm of a real square matrix.

```math
\text{ProblemInfo}[problem_] := \text{Module}[(\text{numdof}=0),
\text{If} [\text{problem}=="MT", \text{numdof}=2]; 
\text{If} [\text{problem}=="CG", \text{numdof}=1]; 
\text{Return}[\text{numdof}];
```

**Figure 23.21.** Module that returns selected information for a benchmark problem.

§23.7.4. Matrix Frobenius Norm

Utility module \text{MtxFrobNorm}, listed in Figure 23.20, computes the Frobenius norm of a real square matrix. The module is invoked as

\[
\text{Anorm} = \text{MtxFrobNorm}[A]
\]

The only argument is

\begin{align*}
A & \quad \text{A square real matrix of order } n_A \times n_A.
\end{align*}

The function return is

\[
\text{Anorm} \quad \text{The Frobenius norm } ||A||_F = \left( \sqrt{\sum_{i=1}^{n_A} \sum_{j=1}^{n_A} A_{ij}^2} \right) / n_A
\]

§23.7.5. Problem Information

Utility module \text{ProblemInfoModule}, listed in Figure 23.21, returns selected information for a specified benchmark problem — for now only the number of degrees of freedom (DOF). It is invoked as

\[
\text{numdof} = \text{ProblemInfo}[\text{probid}]
\]

The only argument is

\begin{align*}
\text{probid} & \quad \text{Benchmark problem identifier.}
\end{align*}

The function return is

\begin{align*}
\text{numdof} & \quad \text{Number of DOF. If the problem is not implemented, it returns zero. This is actually the way in which } \text{SetRefConfig} \text{ checks for that error condition.}
\end{align*}