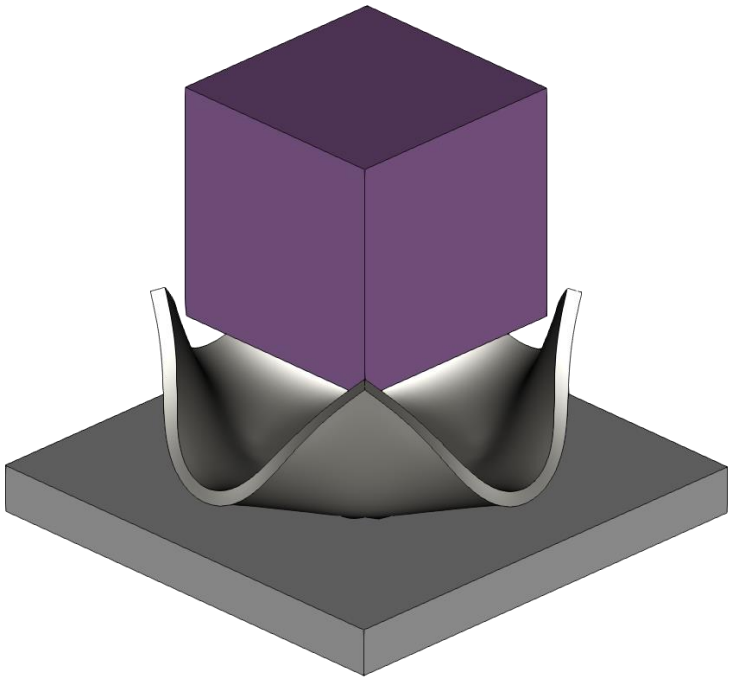
Best Practice

Guideline for User Defined Interfaces in Ansys LS-DYNA Software



2025R1

Table of Content

Abstract

In this document, some of the possibilities for user-defined features in Ansys LS-DYNA software will be presented. The objective is to provide a basic foundation for users with previous experience of the Ansys LS-DYNA software that are interested in starting to develop customized functionality. Both Fortran code examples of user subroutines and accompanying simulation models (keyword files) are provided

This document is under continuous development, and future improved revisions will be released.

Disclaimer

By using this Guideline, you hereby consent to this disclaimer and agree to its terms.

All the information in this Guideline, comprised of this document and the accompanying simulation models, is published in good faith and for general information purposes only. Neither Ansys nor the authors make any warranties about the completeness, reliability, and accuracy of the information in this Guideline. Any action you take upon the information you find in this Guideline is strictly at your own risk. Neither Ansys nor the authors will be liable for any losses and/or damages in connection with the use of the Guideline. It is always up to the user of this Guideline to verify the results.

1 Introduction

The multi-physics solver software Ansys LS-DYNA [1][3] has many pre-defined building blocks (traditional finite elements and other spatial discretization options, over 250 material models [2], many possible contact interactions etc.) for a wide range of different analysis types. For these pre-defined building blocks, the user can input parameter values, for example the hardening curve for a material model, while the behavior based on the given parameter values will be determined by the software.

In addition to these pre-defined modelling capabilities, Ansys LS-DYNA software also offers many possibilities for the user to define fully customized building blocks, like material models, elements, friction models and loadings, see Ref. [20] for an overview. These customer defined features plug into LS-DYNA via user interfaces, involving (amongst others) writing Fortran code. However, most analysis demands are met by corresponding built-in keywords, and it shall be stressed that writing subroutines in Fortran is rarely required for standard analysis.

The wide range of user interfaces offers great possibilities for researchers, either academic or company research, to implement their own developments as building-blocks within the already available LS-DYNA solver environment (rather than developing a complete FE-solver from scratch for research purposes). It is possible to develop a more-or-less fully customized FE-solver in a stepwise fashion: for example, starting with a user defined material model in combination with pre-defined elements, then writing user defined element routines, in the next step combining with user defined friction, and in the end adding even a user defined linear equation solver for implicit analysis.

The purpose of the present document is to provide an overview of some of the possibilities for user-defined features. Traditionally, the user defined features have been seen as a very advanced topic, mostly used by senior researchers and highly experienced specialists. Hopefully, this guideline can shed some light on the development of user defined features from a more applied viewpoint, opening possibilities for experienced LS-DYNA users to also work with user defined features. The everyday user should be able to gain insight into the possibilities that user defined features offer, should the pre-defined building blocks of LS-DYNA seem insufficient for solving a specific task.

It is assumed that the reader is familiar with common engineering terms and has knowledge of continuum mechanics and finite element theory. In addition, some years of experience of the Ansys LS-DYNA software is assumed. Basic keywords or FE modelling will not be discussed.

For most user-defined features treated in this Guideline, examples will be given both in the form of Fortran code and keyword files. Please note that the provided examples are intended for demonstrational purposes only. They should not be used in any type of daily production analysis. The user should review all provided examples with critical eyes.

This Guideline assumes that version R11 or later of the Ansys LS-DYNA software is used. Note that user defined features may be changed, removed, or added in later versions. Versions prior to R11 partially had a different lay-out of the source code for the user defined features. This will not be discussed in any detail in the present document.

This Guideline is currently focused on developing user features in a Linux environment, but some details regarding user feature development in Windows will also be mentioned.

For general support, see lsdyna.ansys.com/knowledge-base/. Useful publications from LS-DYNA users and developers may be found on lsdyna.ansys.com/conference-papers/. Example keyword files can be found at lsdyna.ansys.com/. For further questions, or if errors are found in this document, please contact your local Ansys LS-DYNA supplier.

The present document is based on the course notes [4][5] developed by Dr. Thomas Borrvall, Dr. Jesper Karlsson, and others. Course notes [6][7] developed by Dr. Tobias Erhart have also been of great help. The present document was developed by Dr. Anders Jonsson.

1.1 Co-simulation and interaction with other software

There are potentially many applications for running Ansys LS-DYNA software together with some other software, as a part of co-simulation, for example for applying hydraulic pressure from a system-level simulation tool to a detailed structure model [32]. Even though it may be possible to interact directly with other software via user-defined interfaces [33] the preferable approach is to use the Functional Mock-up Interface (see https://fmi-standard.org/) standard for exchanging data and synchronization of the solutions. This is available in Ansys LS-DYNA since R12, via the FMU Manager (see Ref. [1] for details and download instructions) and the built-in `*COSIM` [30][31] keywords. It offers many possibilities for transferring loads, pressures, displacements etc. both ways between LS-DYNA and other software (Python, Matlab/Simulink, Adams, etc.). In that sense, FMI can provide functionality similar to user defined loadings. For a general introduction to the co-simulation capabilities of the Ansys LS-DYNA software, see the Webinar available from the Ansys Training Center.

2 Overview

See Table 1 for a quick guide to the user features, corresponding subroutine, which Fortran source code file it can be found in, and the section of the Guideline where it is discussed.

In Section 3, some prerequisites required to get started are discussed, including some instructions regarding the LS-DYNA usermat package, and some hints to Fortran programming.

In Section 4, user defined material models are discussed. The user defined materials are accessed via the keyword `*MAT_USER_DEFINED_MATERIAL_MODELS`, and the related subroutines `umat`*`XX`* are found in the Fortran files `dyn21umats.f` and `dyn21umatv.f` of the usermat package.

In Section 5, user defined friction models are presented. The related keyword is `*USER_INTERFACE_FRICTION`, and the subroutines `usrfrc` and `mortar_usrfrc` are found in the Fortran file `dyn21cnt.f` of the usermat package.

In Section 6, user defined weld tie conditions are described. The related keyword is `*CONTACT_AUTOMATIC_SURFACE_TO_SURFACE_MORTAR_TIED_WELD_THERMAL`, and the subroutines `mortar_usrtie` are found in the Fortran file `dyn21cnt.f` of the usermat package.

In Section7 the (almost) inverse, namely user defined tiebreak conditions for Mortar contact are described. The related keyword is `*CONTACT_AUTOMATIC_SURFACE_TO_SURFACE_TIEBREAK_USER_MORTAR`, and the subroutines `mortar_usrtbrk` are found in the Fortran file `dyn21cnt.f` of the usermat package.

Section 8 briefly treats user defined loadings. The related keyword is `*USER_LOADING_{SET}`, and the subroutines `loadud` and `loadsetud` are found in the Fortran file `dyn21.f` of the usermat package.

Table 1. Overview of the user defined features

User defined feature	Subroutine	Fortran file	Section
Material models	umat*XX*, utan*XX*	dyn21umats.f, dyn21utan.f	4
	umat*XX*v	dyn21umatv.f	
Friction	usrfrc, mortar_usrfrc	dyn21cnt.f	5
Weld tie	mortar_usrtie	dyn21cnt.f	6
Tiebreak	mortar_usrtbrk	dyn21cnt.f	7
Loading	loadud, loadsetud	dyn21.f	8

Provided examples (keyword files, Fortran code) were tested with mpp/LS-DYNA R13.1, R14.1.0 and R15.0.2 double precision, sse2 (also avx2 for examples not involving contacts) under Linux, with acceptable results in all cases. All examples except the user defined friction for non-Mortar contacts (usrfrc) were also tested with smp/LS-DYNA R14.1.0 and R15.0.2, with acceptable results.

3 Prerequisites

This Section describes what is required to get started with creating user defined features for Ansys LS-DYNA software. The first step is to download a programming environment (the usermat package) from your LS-DYNA provider, see Section 3.1. Since all interaction with the user defined features will require Fortran programming, a Fortran compiler is a fundamental requirement for getting started. This is not included in the LS-DYNA usermat package. An overview of recommended compilers is presented in Section 3.2, and a very brief introduction to Fortran programming is given in Section 3.6. How to build a user defined module is described in Section 3.3, and how to integrate it in a simulation model using the appropriate LS-DYNA keywords is described in Section 3.4.

It is mainly assumed that programming and user feature development is done in a Linux environment, but some special considerations when working in Windows environment are mentioned in Section 3.5.

Appendix A of Ref. [1] also contains a general overview of how to get started working with user defined features.

Do not hesitate to contact your local Ansys LS-DYNA provider for questions regarding download of required files, or setup of compilers for different environments, or other issues related to the user defined features.

3.1 Download the Ansys LS-DYNA usermat package

It is possible to integrate the user defined features into LS-DYNA using either static or dynamic[1] linking. Static linking means that a special version of LS-DYNA is built, which contains the user defined features. This was the traditional way of working with user-defined features. It is straight-forward but offers little flexibility. It is hard to integrate more features from different sources, for example 3rd party material routines with in-house development friction models.

Dynamic linking, see also Figure 1, means that a shared object (a file with extension `.so` in Linux) is built which then can be dynamically linked to a sharelib – version of LS-DYNA using the `*MODULE` - keywords, see Section 3.4. The dynamic linking approach offers more flexibility since many shared objects from different sources can be linked to the same (standard) LS-DYNA main binary. Building a shared object may also be less resource intensive in terms of compiler and linking time. Currently[2], the dynamic linking approach is only supported by LS-DYNA under Linux.

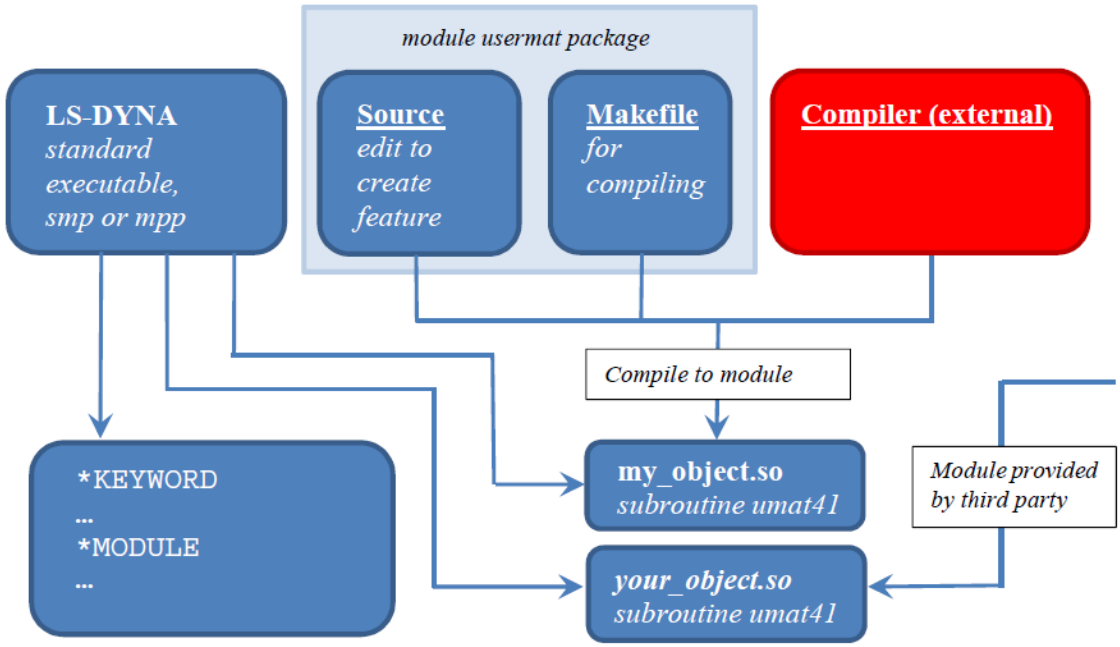


Figure 1. Integrating the user defined features into LS-DYNA using the shared object approach. Image from Ref. [7].

The first step to get started with the development of user-defined features for LS-DYNA is to decide which approach to use, either static or dynamic linking, and then download the required files from your local Ansys LS-DYNA provider. The Linux-version package for dynamic linking will typically be named

`ls-dyna_mpp....sharelib.usermat.tar.gz` or `ls-dyna_mpp....sharelib.usermat.tar.gz_extractor.sh`

and for static linking

[1] From R9. Currently only supported for LS-DYNA under Linux.
[2] By official LS-DYNA versions available up to 2024-12-15.

`ls-dyna_....usermat.tar.gz` or `ls-dyna_....usermat.tar.gz_extractor.sh`

The files will be packaged in a compressed archive format, so after downloading unpacking will be required.

Just as for "standard" LS-DYNA, it is important to distinguish between SMP/MPP/hybrid and single/double precision. This means that the user defined features must be compiled and linked for the right "flavor" and version of LS-DYNA; for example, a shared object compiled for single precision LS-DYNA can in general not be plugged into double-precision LS-DYNA executable, and vice versa. Also, the sse2/avx2/avx512 extensions must be taken into consideration, so that the shared object is built for the corresponding extension – a shared object file based on sse2 will not work properly when used with LS-DYNA with the avx2 - extension.

When working with static linking, it is obvious that re-compilation and new linking of the user-defined features will be required in order to use the user defined features with a new LS-DYNA version. It will also be required for the dynamic linking approach: the shared object file must be built with the corresponding environment for a specific LS-DYNA version, since for example interfaces and arguments to subroutines may have changed between versions. This means that also when working with dynamic linking, the user features must be re-compiled in order to be used with a new LS-DYNA version.

The Fortran files of the usermat package come with some example material models and related subroutines, which may be used as templates for further developments.

3.2 Fortran compilers

Intel's Fortran compiles are in general recommended for both Linux and Windows. More specifically [4], for

- Linux Redhat or CentOS, use
  - Intel Fortran Compiler [10] 2013 for LS-DYNA R9,
  - Intel Fortran Compiler 2016 for LS-DYNA R11 and R12,
  - Intel Fortran Compiler 2019 for LS-DYNA R13, R14 and R15.
- Linux Suse, use PGI Fortran Compiler 16.5 (10.5 for LS-DYNA R9) for LS-DYNA R11 and R12,
- Windows x64, see Section 3.5.

Some special setup of the system environment will most likely be required. For example, when working with the Intel Fortran compiler under Linux, the command

`compilervars.sh -arch intel64 -platform linux`
found in the compiler installation directory, under (for Intel Fortran 2016)
`compilers_and_libraries_2016/linux/bin/` must be issued in order to apply appropriate settings before compilation can be performed. For compiling mpp/LS-DYNA, or shared objects, also the appropriate mpi-Fortran wrapper (for example `mpiifort` for Intel MPI) must be configured. For Intel MPI under linux, the command

`compilervars.sh -arch intel64 -platform linux`

found in the MPI installation directory, under (for Intel MPI 2019) `compilers_and_libraries_2019/linux/bin/` must be issued in order to apply appropriate settings before compilation for mpp, using `mpiifort`, can be performed.

For further details on this, consult the documentation of the compiler in question, or contact your local Ansys LS-DYNA provider for support.

3.3 Compiling the subroutines for using Ansys LS-DYNA with user defined features

Once the LS-DYNA usermat package has been downloaded and unpacked, the next step is to set up the compiler environment and test-compile the files as-is, without any modifications. In Linux, the `Makefile` contains the instructions for compiling and linking that are required to build either a shared object file (`libmppdyna_ … .so`) or a statically linked customized LS-DYNA-version (the result will in the latter case be a monolithic executable called `mppdyna` or `lsdyna`). In Linux, open the text file `Makefile` and edit the Fortran compiler command to the appropriate for the present installation (see Section 3.2). The next step is to build the desired shared object, which is obtained by executing the Linux `make` command. In some cases, it may be required to re-compile all objects, this can be achieved by first issuing `make clean` and then the `make` command.

Finally, run a small test model, for example tension of some solid elements (see for example, Section 4.6.1), without any user defined features active in the model but still including the shared object (see Section 3.4 for details) to verify that that LS-DYNA runs as expected.

The purpose of this initial testing stage is to establish a basis for further development of user defined features for LS-DYNA. It is good practice to have sorted out any problems directly related to compiling and linking the shared object files, or problems related to loading them into the LS-DYNA simulation model, before starting to work with the development of advanced user-defined features. Then, if errors should occur at later stages, troubleshooting can be more efficiently focused directly on the likely cause.

Once this testing stage is completed, the development of user defined features can commence, preferably in an incremental and iterative way, as outlined in Section 3.6.5.

3.4 Plugging a user-defined shared object into Ansys LS-DYNA

The shared object file must be built with the corresponding environment (usermat package) for a specific LS-DYNA version. This means that also when working with dynamic linking, the user features must be re-compiled in order to be used with a new LS-DYNA version. Note also that specific double / single precision versions of the `.so` – files must be built. Also, the hardware acceleration extension (sse2, avx2 or avx512) must match between the usermat package and the LS-DYNA version. For example, let us assume that a user-defined material model has been developed for LS-DYNA R11.1.0 sse 2, and this has been compiled to the shared object file `libmppdyna_R111.so`. In order to use this user-defined material model with LS-DYNA R12.2.2 avx2, a new shared object file (which can be named for example `libmppdyna_R121_avx2.so`) must be built using the corresponding usermat package. This

means adapting the user contributed source code to the dyn21-files in the new package. The dyn21-files should **never** be copied between versions.

The LS-DYNA keywords required to dynamically link a user-defined shared object to the main LS-DYNA binary all begin with `*MODULE` [1]. The path to a shared object can be specified by the keyword `*MODULE_PATH`. A shared object is loaded by the keyword `*MODULE_LOAD`. In order to map the user subroutines loaded in shared objects to the model, the keyword `*MODULE_USE` is applied, for example in situations where user defined materials for different 3rd party suppliers should be combined in the same simulation model. If only a single shared object is used, it can also be linked using the environment variable `LD_LIBRARY_PATH` or the command line argument "`module=`", instead of `*MODULE`.

An example of the use of these keywords is also presented in Section 4.1.

3.5 User-defined feature development in a Windows environment

In this section, some special details regarding user defined feature development for LS-DYNA running in a Windows environment will be mentioned. For the Windows versions of LS-DYNA, the only option currently available is to work with static linking. This means that a special version of LS-DYNA is built, including the user defined features. The Windows-version package for static linking will typically be named `ls-dyna_..._winx64_.._lib.zip`, or `ls-dyna_..._winx64_.._lib_installer.exe`, see also Figure 2.

A brief instruction on how to build a LS-DYNA executable is given in the `readme.txt` – file provided in the Windows usermat package.

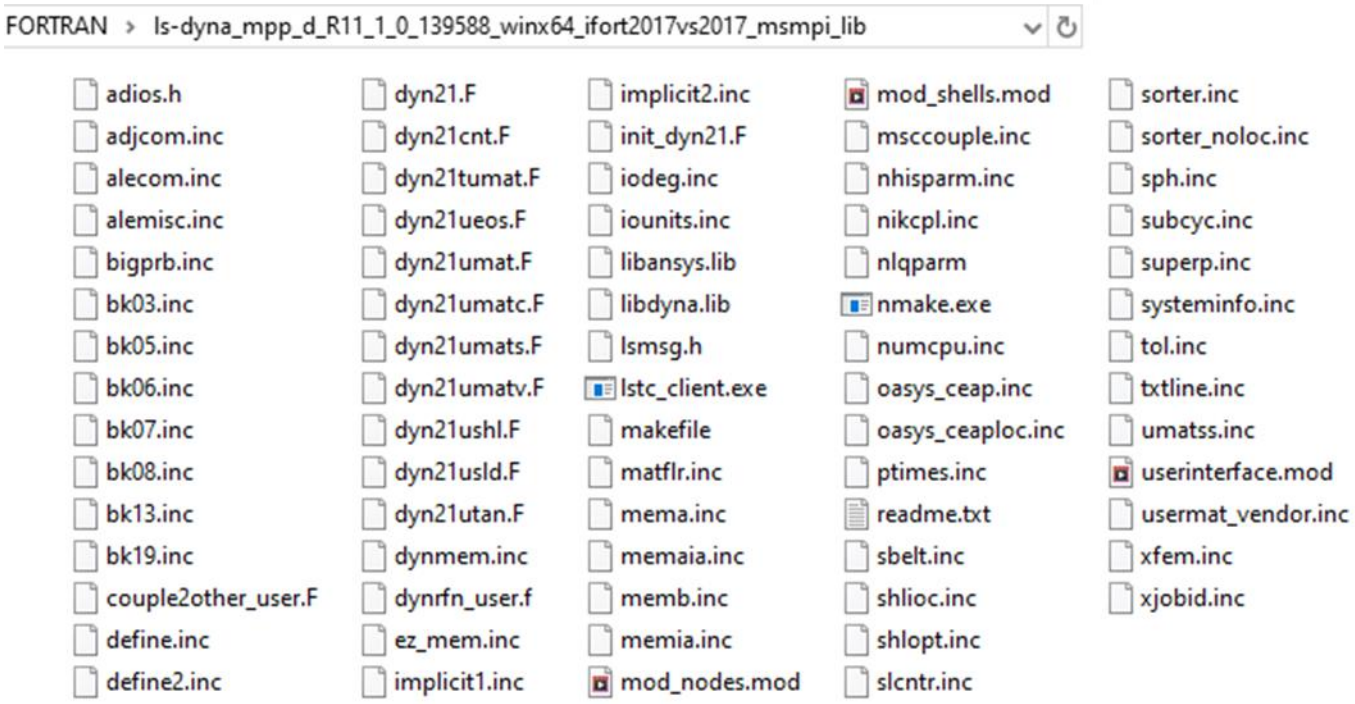


Figure 2. Example of contents of the usermat package for Windows.

The recommended Fortran compiler for LS-DYNA R11 and R12 is

Intel Parallel Studio XE 2017

and the corresponding Microsoft application environment, required for linking and access to standard libraries, is

Microsoft Visual C++ 2017 x64 Cross Tools.

For LS-DYNA R13, R14 and R15, the recommended Fortran compiler is

Intel Parallel Studio XE 2019 (Update 6, Composer Edition)

and the corresponding Microsoft application environment, required for linking and access to standard libraries, is

Microsoft Visual C++ 2019 x64 Cross Tools.

Note that the recommended compilers and tools may change for coming LS-DYNA versions, please see the information provided in the `readme.txt` – file for updated information.

In addition, the MPI for mpp/LS-DYNA under Windows is required. Note that it is also required to install the MS MPI Software Development Kit (SDK). Recommended are Microsoft MPI v8.1 for R11, and Microsoft MPI v10 for R12 of LS-DYNA, which are available for free download from www.microsoft.com.

The command `nmake.exe` [26] is used for building the LS-DYNA executable, based on the information in the `makefile` text file. Note that it may be required to update the search paths in the `makefile` text file depending on the local installation.

In order to run the customized LS-DYNA version, the executables `ansyscl.exe` and `lstc_client.exe` and the library `libiomp5md.dll` (provided with the Intel Fortran compiler installation) need to be in the same folder as the LS-DYNA executable. In some cases, Windows security settings or antivirus programs may prevent execution of a customized LS-DYNA version. How this may be remedied is discussed further in Appendix A.

3.6 A very brief introduction to Fortran programming

All interaction with the user defined features of LS-DYNA will involve some amount of Fortran programming. This section is taken from the course material [4] and the purpose is to give a very brief introduction to programming in Fortran. Other sources are for example, from Intel. Fortran's official home page is https://fortran-lang.org/.

Another good starting point seems to be
https://www.tutorialspoint.com/fortran/fortran_useful_resources.htm

3.6.1 Input format

Fortran is an imperative programming language. A typical program consists of a set of statements, which are executed sequentially. In pseudo-code, a typical lay-out looks something like this:

```
PROGRAM name
declarations
executable statements
END
```

The input is case-insensitive, and recommended practice is to always use lower case (upper case only use for emphasis here). Often fixed-format input is used, which means that each Fortran statement is written in positions 7 – 72 on a line. Position 6 is reserved as a continuation marker; in order to continue a statement on the next line, put a character in position 6 of the continued line. If a 'C' or 'c' is found in position 1 of a line, that line is taken as a comment. This is a convenient feature, making it easy to write explanations about what certain parts of the code are supposed to do, or simply de-activating certain lines of code. Position 1 – 5 are reserved for statement numbers or labels.

3.6.2 Variables and arrays

Variables can be of type INTEGER, REAL (floating point numbers) or REAL*8 (double precision floating point numbers), CHAR (characters or strings) or LOGICAL (Boolean). Variable names can be up to 31 characters long. For example:

```
INTEGER i, j,k
REAL a,b,c
REAL*8 d,e,f
```

If the variable declarations are omitted, the compiler will make certain assumptions regarding types of variables, for example, that variables beginning with the letters I, J, K, L, M, N are INTEGER. The lack of specification often leads to program errors, and it is strongly recommended that variable types are always declared. The statement IMPLICIT NONE means that all variables must be declared, and use of an undeclared variable will cause a compilation error.

Arrays are fields of variables, and are declared as for example

```
INTEGER ii(10), ndx(100)
REAL*8 sigma(6), avec(100)
```

Variables can then be assigned values by the = statement and be used for arithmetic operations (summation by +, multiplication by *, division by / and subtraction by -), for example:

```
j=20
b=2.49
avec(j) = b
```

The Fortran 90 standard offers convenient input for operations on arrays, where a sequence of elements can be accessed in one line, reducing the need for `DO` – loops, for example:

```
s(1:3) = sig(1:3)-sum(sig(1:3))/3.0
```

3.6.3 Subroutines

Subroutines are used to simplify coding and should be used for code that must be executed many times.

```
SUBROUTINE subname(parameters)
IMPLICIT NONE
declarations
statements
RETURN
END
```

In the main program, or from other subroutines, the subroutine is called by

```
CALL subname(parameters)
```
The list of parameters in the call and the subroutine declaration must match. In subroutines, arrays in the parameter list can be declared with assumed size, by use of an asterisk / star within parenthesis to declare its size as flexible, for example

```
SUBROUTINE setzero(a, la)
IMPLICIT NONE
REAL*8 a(*)
INTEGER la
…
statements
RETURN
END
```
In Fortran, all parameters to a subroutine are passed by reference. This means that if a parameter is changed (assigned a new value) in the subroutine, it will also be changed in the context where it was called from. This also means that special care must be taken when programming, so that only variables that are intended as output from a subroutine are updated.

3.6.4 Some basic Fortran statements

To repeat statements iteratively a specified number of times, the `DO` / `ENDDO` construct can be used, for example

```
DO var=first, last
  statements
ENDDO
```
The variable `var` will take values *`first, first + 1, … , last`*. In order to break a `DO` – loop, the statement `EXIT` can be used, for example

```
DO iter=1, maxiter
  statements
  IF(residual.LE.tol)THEN
    EXIT
  ENDIF
ENDDO
```
To control the program flow based on logical conditions, the `IF` / `THEN` / `ELSE` construct can be used, for example

```
IF ABS(s) .GT. 1.E-15 THEN
  si = 1./s
ELSE
  si = 1.E16
ENDIF
```
The `GOTO` statement can be used to make the execution continue on the line with a specified statement number, for example

```
GOTO 10
```
jumps to statement number 10.

3.6.5 Programming for user-defined features

In this Section, some tips specific for programing user-defined features in Ansys LS-DYNA follow.

When developing the user-defined feature, it is recommended to add some debug printouts in order to make sure that the user-defined feature actually is active and called from the main LS-DYNA binary. The user can get access to LS-DYNA's message files (`messag`, `mes0*`) and highspeed-printout file (`d3hsp`) via the standard Fortran `write` – command. This is simplified by including the file `iouinits.inc` in the subroutine, which contains the unit number for the message files (`mes0*`) in the variable `iomsg` and the `d3hsp` – file in the variable `iohsp`. An example follows:

```
.
INCLUDE 'iounits.inc'
.
write(iomsg, *) 'user defined feature writes to the message file'
.
RETURN
END
```

It is recommended to work with the development in a stepwise fashion, making small modifications and checking that each modification has the intended effect. Also starting with small (a couple of hundred elements for example) FE-models is recommended. It is good practice to verify the results from the user defined feature by comparing to handbook solutions or built-in LS-DYNA functions.

Once the developed feature is considered ready for use in a larger scale context, it is recommended to disable or limit debug printouts, since these may slow down performance significantly and create big output files.

4 Material models interface

The over 250 built-in material models in LS-DYNA [2] cover many applications, from linear elasticity to orthotropic plasticity models, foams, and composites. A web-based material model selection guide is available from https://lsdyna.ansys.com/dynamat/ [24]. Still, there may be cases where specific customer demands cannot be perfectly matched by the existing material models, and perhaps the most common customization of Ansys LS-DYNA is the development of a specialized material model. Third-party companies, like MatFEM [16] and E-Extreme / Digimat [17], deliver material models for specific purposes, like advanced failure modelling and analysis of composites. From R13 of LS-DYNA, also the option to add plasticity, viscoelasticity, creep etc. to already existing material models is available by the keyword `*MAT_ADD_INELASTICITY` [29] in a quite general manner. Fairly advanced damage and failure models can be added to many of the built-in material models, by the keywords [2] `*MAT_ADD_DAMAGE_{GISSMO/DIEM}` see for example Refs. [14][15] for examples and background.

This section will not focus specifically on material modelling as such, since it is a very wide field, with many specializations for metals, composites etc. A vast amount of published research is available, and it is currently a very active research field. For a background to material modelling and continuum mechanics, the reader is referred to for example Refs. [11][12][13].

In the following, it is assumed that an existing valid material model is to be implemented as a user defined material model in LS-DYNA. The user-defined material subroutine `umat`*XX* (where 41 ≤ XX ≤ 50) is called in the solution sequence, with the strain rate (or deformation gradient) as main input, and its objective is to update the (Cauchy) stress σ (and history variables, if required) for the next time step, in order that nodal forces can be computed, see Figure 3. All input quantities are passed to the user

subroutine in the local element coordinate system, so no additional transformations are required by the user. In the case of an anisotropic material model, the same options for specifying material directions as for the built-in LS-DYNA materials are available (see the remarks related to `*MAT_2` of Ref. [2]). Some care must be taken when implementing the material model for shells and beam elements, since it is then required that the material model will deliver stresses that are consistent with the assumptions related to structural elements (for example $\sigma_{33} = 0$ for shells). In the implicit solution sequence, also the material stiffness matrix is required, and the corresponding subroutine `utanXX` is called when assembling the global stiffness matrix.

It is also possible to create user defined thermal material models, see Appendix H of Ref. [1], but this option will not be discussed further in the present release of this Guideline.

The keyword interface for passing parameter values etc. to the user defined material model is described in Section 4.1. Post-processing the results from user subroutines is outlined in Section 4.2. The Fortran interface to the user subroutines is described in Section 4.3. Some useful pre-defined subroutines for common operations, such as push-forward, are described in Section 4.4. Finally, in Section 4.5, some examples of material model implementation are presented.

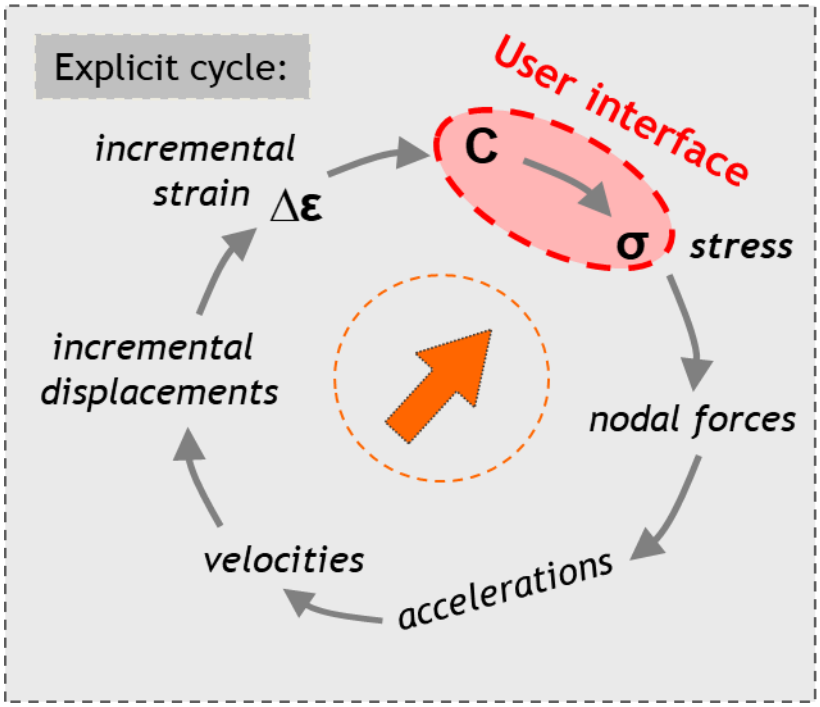


Figure 3. In the explicit solution sequence, the objective of the user-defined material interface is to output the updated stress based on provided incremental strain and current stress. Image from Ref. [7].

See also Appendix A of the Keyword manual [1] for more details on user defined material models. A dedicated course in User Defined Material Models is available from the Ansys Training Center.

4.1 Keyword interface to the user defined material models

The keyword interface to the user defined material models is given by `*MAT_USER_DEFINED_MATERIAL_MODELS`. By this, a user defined material can be assigned a Material ID (MID) that in turn can be referenced by a `*PART` – definition. One example of the keyword syntax follows:

```
*MAT_USER_DEFINED_MATERIAL_MODELS_TITLE
User defined J2 plasticity UMAT41
$#1    MID        RO        MT       LMC       NHV    IORTHO     IBULK        IG
         3  2.700E-9        41         8         1         0         3         4
$#2  IVECT     IFAIL    ITHERM    IHYPER      IEOS      LMCA    UNUSED    UNUSED

$#       P1        P2        P3        P4        P5        P6        P7        P8
     70.E3      0.31   61404.0   26718.      103.
```

The variables of this keyword are:

- *MID*: The Material ID.
- *RO*: Mass density. This, in combination with the bulk and shear moduli, is required input for LS-DYNA to be able to compute the critical time step size for explicit analysis, and penalty factors for contacts and joints.
- *MT*: The user material type. In this case, *MT* = 41 means that the subroutine `umat41` (and also `utan41` in the case of an implicit analysis) will be called.
- *LMC*: The number of material parameters to be input on the keyword and passed to the subroutine. In this case 8, meaning that the variables P1 – P8 will be read during initialization and passed to the subroutine `umat41` in the `cm` array. Fields left blank in the keyword input will be passed as zero.
- *NHV*: The number of history variables to be stored on integration-point level (max. 200). In this case, one history variable is defined.
- *IORTHO*: Set to 1 if material is orthotropic (necessary for definition of material axes). The default is an isotropic material model.
- *IBULK*, *IG*: The addresses of the bulk and shear moduli, respectively, in the parameters array. In this case, `IBULK` = 3 means that LS-DYNA can find the bulk modulus as the variable `P3`, and `IG` = 4 means that LS-DYNA can find the shear modulus as the variable `P4`.
- *IVECT*: If set to 1 the vectorized version of the material routine is called. Default is that the scalar version is called, once for each integration point. The vectorized vs. scalar version of a user subroutine is discussed mode in Section 4.3.
- *IFAIL*: Set to 1 if the material routine should be able to control element erosion of solids and shells.
- *ITHERM*: Set to 1 if the integration point temperature shall be computed and passed to the user subroutine.
- *IHYPER*: Flag for hyperelastic materials. By setting *IHYPER* = 1, the deformation gradient will be passed to the user subroutine. The implementation of a hyperelastic material is presented as an example in Section 4.5.1.
- *IEOS*: Flag for equation of state.
- *LMCA*: Length of additional material constants array (unlimited).
- `P1 … Pn`: Material parameters passed to the user defined material subroutine in the array cm. In this case, material parameters for aluminum are defined. P2 is the Poisson's ratio in this case, and P5 refers to a Load curve ID (103 in this case), which specifies the yield stress as a piecewise linear function of the accumulated effective plastic strain.

Should more than 10 different user defined material models be required, the *MODULE – keywords provide a handy solution. The following example illustrates this: first, the keywords *MODULE_PATH and *MODULE_LOAD read in two different shared objects,

```
*MODULE_PATH
 … path_to_modules …
*MODULE_LOAD
$#1            MDLID                                                TITLE
                   1 Library 1
$#2 FILENAME
sharedobje01.so
*MODULE_LOAD
$#1            MDLID                                                TITLE
                   2 Library 2
$#2 FILENAME
sharedobje02.so
```

By this, the shared object sharedobje01.so is loaded and assigned the module ID 1, and the shared object sharedobje02.so is loaded and assigned the module ID 2. By the *MODULE_USE keyword, the user material types (typically 41 ≤ MT ≤ 50) can be re-mapped to new ID numbers:

```
*MODULE_USE
$#1            MDLID
                   1
$#2             TYPE              PARAM1                          PARAM2
UMAT                                  41                            1001
*MODULE_USE
$#1            MDLID
                   2
$#2             TYPE              PARAM1                          PARAM2
UMAT                                  41                            1002
```

This means that the umat41 of sharedobj01.so can be referred to as material type 1001, and the umat41 of sharedobj02.so can be referred to as material type 1002 when creating a user defined material model:

```
*MAT_USER_DEFINED_MATERIAL_MODELS
$#1    MID        RO        MT
         1   7.85e-9      1001
…
*MAT_USER_DEFINED_MATERIAL_MODELS
$#1    MID        RO        MT
         2   2.70e-9      1002
…
```

Then these material models can be assigned to *PARTS, as any built-in LS-DYNA material model:

```
*PART
first part
$#     PID     SECID       MID
         1         1         1
*PART
second part
$#     PID     SECID       MID
```

2 2 2

By a similar procedure, the limitation to 10 user defined material models can be overcome by separating the material models and placing 1 – 10 in the sharedobje01.se, and then 11 – 20 in the sharedobje02.so, and so on.

Note that one user defined material type, for example `MT` = 41, may be referenced by an unlimited number of `*MATERIAL`s (with different Material IDs). However, in that case only the parameters `RO`, and `P1...Pn` may be changed on each `*MATERIAL`.

4.2 Post processing user defined material models

The number of additional material history variables to be output to the binary 3D databases `d3plot`, `d3part` and `d3drlf` is controlled by the *NEIPH* (output for solid elements) and *NEIPS* (for shell elements) variables of the keyword *`DATABASE_EXTENT_BINARY`. For example, in order to post-process 5 material history variables for a user defined material model, set *NEIPH* = *NEIPS* = 5. This will store the first 5 history variables (`hisv`(1:5), see Section 4.3) in the binary databases. This means that the history variables of interest for post-processing should appear in the beginning of the `hisv` array. Interactive post-processing of the binary 3D databases `d3plot`, `d3part` and `d3drlf`, including extra history results, should be possible using LS-PrePost [22], META [23] or other third-party post-processors.

The number of additional material history variables to be output to the `elout` file (for history / 2D curve plotting) is controlled by the `OPTION1` (for solid elements) and `OPTION2` (for shell elements) variables of the keyword `*DATABASE_ELOUT`. For example, in order to post-process 5 material history variables for a user defined material model, set *OPTION1* = *OPTION2* = 5.

For the history output, note that also `*DATABASE_HISTORY_...` keywords are required, in order to specify for which elements the data should be output.

One motivation for writing a user material subroutine may be that is makes it possible to output additional history results from a “standard” material model, for example back stress terms, or the maximum effective stress during a simulation.

The user defined subroutines can also write text output to the message (`mes0*`) and `d3hsp` – files. This can be very useful, for example in case some initial parameter fitting to a given test curve is done, some quality measure indicating the validity of the fit can be output. It is at least during the development phase warmly recommended to add output of some text message to confirm which subroutine(s) that are called during the solution of a specific FE model.

4.3 Interface to the user-defined material models in the subroutine umat

There are two[3] basic categories of subroutines for user defined material models:

- If the subroutine should be called once for each element and integration point, it is denoted as a *scalar* (or *serial*) material subroutine. This corresponds to *IVECT* = 0 on the *MAT_USER_DEFINED_MATERIAL_MODELS keyword. The subroutines are named umat*XX*, where 41 ≤ XX ≤ 50, and can be found in the dyn21umats.f – file.
- If the subroutine can process an array of elements for each integration point, it is denoted as a *vectorized* material subroutine. This corresponds to *IVECT* = 1 on the *MAT_USER_DEFINED_MATERIAL_MODELS keyword. The subroutines are named umat*XX*v, where 41 ≤ XX ≤ 50, and can be found in the dyn21umatv.f – file.

While the scalar approach may be easier to code – since only scala data needs to be handled – the vectorized version will probably lead to faster solution timing; the reduced number of subroutine calls will lead to less overhead processing. It is in all cases to be expected that a user-defined version of even a simple material model will not be as fast as a built-in equivalent material model. If a user defined material model is to be implemented, it should be motivated by other benefits, for example new material behavior or enhanced post-processing, than improved solution speed.

The main subroutine for interfacing between LS-DYNA and the user defined material models is usrmat and can be found in the dyn21umat.f – file. It should normally not require editing, even though it is possible in order to pass additional information to the user defined subroutines [7].

As input to the material subroutine, LS-DYNA provides the

- Incremental strain,
- Current stress,
- History variables,
- Material parameters,
- Element type (solid, shell, beam, discrete beam …) and
- Temperature.

The subroutine should, based on this and according to the material law, provide

- The stress in the next time step,
- strain corrections for structural elements (beams / shells), and if required also
- updated history variables.

[3] In addition, there is also the possibility to define cohesive user material models, by the subroutines umatXXc of the dyn21umatc.f – file. This will not be discussed further in the present revision of this Guideline.

To be more specific, the parameter list for a scalar implementation of a user defined material model (in this case `umat41`) is:

```
      subroutine umat41(cm,eps,sig,epsp,hsv,dt1,capa,etype,tt,
     1 temper,failel,crv,nnpcrv,cma,qmat,elsiz,idele,reject)
```

An overview and brief description of the parameters to the subroutine is shown in Table 2.

Table 2. The arguments to the subroutine `umatXX`

Argument	Description	Input / Output
`cm`	Material constants	Input
`eps`	Local strain increment in Voigt(1) notation	Input /Output(2)
`sig`	Local stress in Voigt notation	Input / Output
`epsp`	Accumulated effective plastic strain	Input / Output
`hsv`	History variables	Input / Output
`dt1`	Current time step size	Input
`capa`	Reduction factor for transverse shear of shells and beams(3)	Input
`etype`	String describing element type	Input
`tt`	Current problem time	Input
`temper`	Current temperature	Input / Output
`failel`	Failure flag, set to .true. to indicate failure of an integration point (Requires *IFAL* = 1, see Section 4.1)	Input / Output
`crv`	Array representation of curves in the keyword deck	Input
`nnpcrv`	Number of discretization points(4) per curve	Input
`qmat`	Transformation matrix in case of *IHYPER* =3	Input
`cma`	Additional memory for material data	Input
`elsiz`	Characteristic element size	Input
`idele`	Element id	Input
`reject`	For implicit analysis: set to .true. if the current implicit iterate should be rejected for some reason	Output

Notes: (1) This means that the symmetric stress and strain 3 × 3 tensors are represented as 6 × 1 vectors. (2) For shell elements, the through-thickness strain eps(3) shall be updated corresponding to the plane stress assumption. (3) This corresponds to *SHRF* on the `*SECTION_SHELL` or `*SECTION_BEAM` card. (4) The curves are internally re-discretized (to the number of points specified by *LCINT* on the `*CONTROL_SOLUTION` - keyword) in order to speed up the curve/table look up.

In LS-DYNA, the symmetric 3 × 3 stress tensor (σ) and the symmetric 3 × 3 strain rate tensor $\dot{\varepsilon}$ are represented as 6 × 1 vectors, stored in arrays `sig` and `eps`, using Voigt notation:

$$\texttt{sig}(1{:}6) = (\sigma_{11}, \sigma_{22}, \sigma_{33}, \sigma_{12}, \sigma_{23}, \sigma_{13})$$

and

$$\texttt{eps}(1{:}6) = (\dot{\varepsilon}_{11}, \dot{\varepsilon}_{22}, \dot{\varepsilon}_{33}, 2\dot{\varepsilon}_{12}, 2\dot{\varepsilon}_{23}, 2\dot{\varepsilon}_{13}).$$

Note that this representation differs from for example Ref. [11].

The accumulated effective plastic strain (clearly a history variable) has its own parameter `epsp` outside the history variables array `hsv`. This means that for example basic J_2 – plasticity can be defined (at least for explicit analysis) without any additional history variables (leaving *NHV* = 0 on `*MAT_USER_DEFINED_MATERIAL_MODELS`). In cases where both built-in materials (for example MAT_24) and user-defined materials are used in the same FE-model, post-processing (for example fringe plotting the accumulated effective plastic strain) is easier if the user-defined materials also use the parameter `epsp` to store the accumulated effective plastic strain (if applicable).

The `etype` parameter is a string, describing the element type, see Table 3. If a material model should be generally valid also for structural elements (shells, beams) this will require special care with respect to different types of stress / strain updates in order to fulfill the assumptions (for example updating the through-thickness strain `eps(3)` for shells in order to fulfill `sig(3) = 0.`). This is illustrated further in the examples of Section 4.5. If is in any case good practice to add element type checks in the subroutine and stop LS-DYNA with a reasonable error description (see Section 4.4) in case there is an attempt to apply a user-defined material model to element types that are not properly supported.

Table 3. The element types and their string values

`etype`.eq.	Description	Coment
`'solid'`	3D solid elements	
`'sph'`	SPH, smoothed particle hydrodynamics	Not covered in this Guideline
`'sldax'`	2d solids, axisymmetric	Shells elform 14 and 15
`'shl_t'`	Shells with thickness stretch	Shells elform 25, 26, 27
`'shell'`	Shells without thickness stretch	All other shell elforms, and thick shell elform 1, 2
`'tshel'`	For thick shells	Thick shell elforms 3, 5. May use same material formulation as 3D solid elements
`'hbeam'`	For beam elements	Beam elform 1, 11
`'tbeam'`	For trusses	Beam elform 3
`'dbeam'`	Discrete element beams (springs, dashpots etc.) see also Section 4.3.1	Beam elform 6. Not supported in implicit
`'beam'`	For beams	All other beam elforms

If the variable *ITHERMAL* = 1 on the `*MAT_USER_DEFINED_MATERIAL_MODELS` – keyword, the material temperature at the current integration point is available in the `temper` parameter. By this, temperature dependent material behavior can be implemented. The conversion of deformation to heat is taken care of by LS-DYNA outside the user-defined material models, so there is no need for the user subroutine to update the temperature.

The user-defined material model can also involve a damage/failure model, and if the variable *IFAIL* = 1 on the corresponding `*MAT_USERDEFINED_MATERIAL_MODELS` -keyword, the material model can also indicate failure of an integration point, by setting the Boolean parameter `failel = .true.` The parameter is also input to the subroutine, which means that the logic of the subroutine must handle what to do with output of for example of stress and plastic strain after an integration point is failed: should the data be set to zero, or kept constant?

The `capa` parameter is the transverse shear reduction factor, corresponding to the value of *SHRF* on `*SECTION_SHELL`. How to use it will be demonstrated in the examples of Section 4.5.

For implicit analysis, the material routine can set the parameter `reject = .true.` to indicate that the current iterate should be rejected, for example if the increment of plastic strain (`depsp` in the example below) is above some tolerance threshold (`epsinctol`):

```
if(depsp.gt.epsinctol)then
  reject = .true.
  return
endif
```

This will trigger a `RETRY` in the non-linear implicit solver, meaning that the time step will be tried again with a smaller time increment.

For a vectorized implementation, which is invoked by setting *IVECT* = 1 on the `*MAT_USER_DEFINED_MATERIAL_MODELS` -keyword, data for a vector block of elements is passed to the user-defined material subroutine. It is required to include the file `'nlqparm'` in the subroutine. The length of the vector block of integration point data is given by the parameter `nlq`, and the objective of the subroutine is to update the element data in the range from `lft` to `llt`.

To be more specific, the parameter list for a vectorized implementation of a user defined material model (in this case `umat41v`) is:

```
subroutine umat41v(cm,d1,d2,d3,d4,d5,d6,sig1,sig2,
. sig3,sig4,sig5,sig6,epsps,hsvs,lft,llt,dt1siz,capa,
. etype,tt,temps,failels,nlqa,crv,nnpcrv,cma,qmat,elsizv,idelev,
. reject)
```

A brief overview of the parameters is presented in Table 4. The main body of a vectorized implementation is outlined in pseudo – code below:

```
subroutine umat41v (...,lft,llt,...)
include 'nlqparm'
. . . declare varibles and parametes …

do k=lft,llt
  . . .
  process element point k, update sig1(k), sig2(k)… sig6(k) etc.
  . . .
enddo
return
end
```

The implementation of an orthotropic material model can be simplified by setting *IORTHO* = 1 on the material card (`*MAT_USER_DEFINED_MATERIAL_MODELS`). By this, the local (material) coordinate system is defined by two additional cards, specifying how the coordinate system is formed and updated (this is described in some detail under `*MAT_ORTHOTROPIC_ELASTIC` of Ref. [2]). With *IORTHO* = 1, all data passed to the constitutive routine `umatXX` (`umatXXv`) is in the local system and the transformation back to the global system is done outside this user defined routine.

Table 4. The arguments to the subroutine `umatXXv`

Argument	Description	Input / Output
`cm`	Material constants	Input
`dX`(1)`(nlq)`	The strain vector in element *k* is `(d1(k), d2(k), d3(k), d4(k), d5(k), d6(k))`	Input /Output
`sigX(nlq)`	Local stress in Voigt notation, components of the stress vector is `(sig1(k), sig2(k) … sig6(k))`	Input / Output
`epsp(nlq)`	Accumulated effective plastic strains	Input / Output
`hsvs(nlq, *)`	History variables	Input / Output
`lft ,llt`	Loop over vectors, from `lft` to `llt`	
`dt1siz(nlq)`	Current time step sizes	Input
`capa`	Reduction factor for transverse shear of shells and beams	Input
`etype`	String describing element type	Input
`tt`	Current problem time	Input
`temps(nlq)`	Current temperature in element point *k* is `temps(k)`	Input / Output
`failels(nlq)`	Failure flag, set to .true. to indicate failure of an integration point	Input / Output
`crv`	Array representation of curves in the keyword deck	Input
`nnpcrv`	Number of discretization points per curve	Input
`cma`	Additional memory for material data	Input
`qmat`	Transformation matrices in case of *IHYPER* =3	
`elsizv(nlq)`	Characteristic element sizes	Input
`idelev(nlq)`	Element ids	Input
`reject`	For implicit analysis: set to `.true.` if the current implicit iterate should be rejected for some reason	Output

Notes: (1) `X` goes from 1 to 6.

4.3.1 Interface for discrete beam elements

Since discrete beam elements (*ELFORM* = 6 on `*SECTION_BEAM`), like springs, dashpots etc., normally work based on changes in element length rather than strains, this data is passed in the `eps`(1:6) array, for each degree of freedom, instead of strains. The objective of the user material subroutine for discrete elements is then to update the (generalized) forces, which are stored in the `sig`(1:6) array, for each degree of freedom. Thus, for discrete beam elements,

`sig`(1:6) = (F_1, F_2, F_3, M_1, M_2, M_3)

and

`eps`(1:6) = (du_1, du_2, du_3, dr_1,dr_2, dr_3),

both given in the local element coordinate system, see Figure 4. It is recommended to set `scoor` = ± 12 on `*SECTION_BEAM`, for correct update of the beam orientation.

User defined material models for discrete beams are currently not supported in implicit.

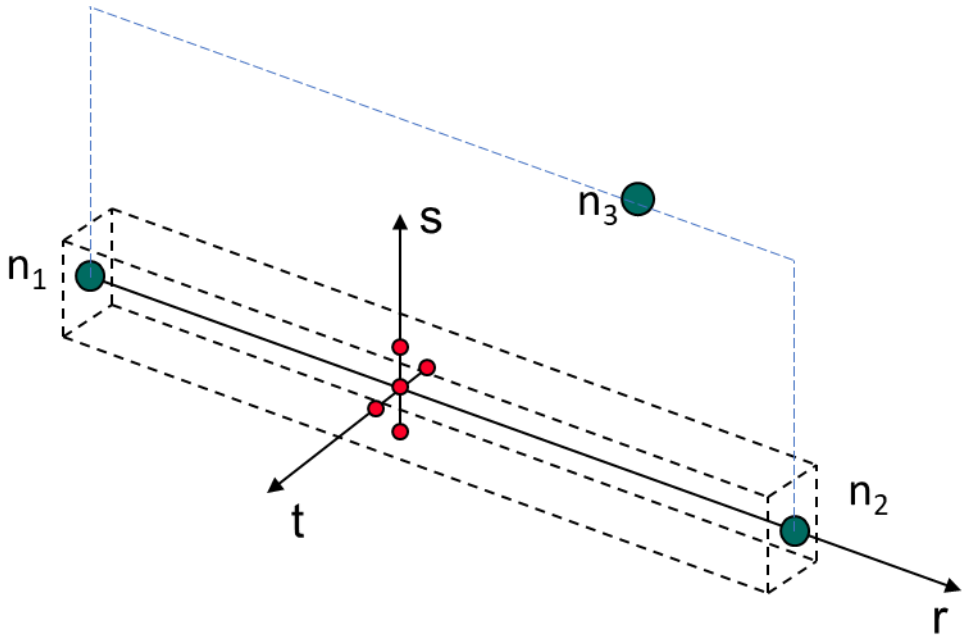


Figure 4. Orientation of discrete beams by a third node (n_3). The (r,s) – plane is defined by the three points (n_1, n_2, n_3). See Ref [1] under `*SECTION_BEAM` for other options of orientation.

4.3.2 Material tangent modulus subroutine utan for implicit analysis

For implicit analysis, also the material tangent modulus tensor for a user defined material must be provided. An algorithmically consistent implementation is required in order to achieve quadratic convergence rate in the global equilibrium iterations.

For the user material type *XX*, the material tangent modulus should be provided by the subroutine `utan`*XX* in the case of a scalar material subroutine and by `utanXXv` if a vectorized implementation of the material routine is given. These subroutines are found in the file `dyn21utan.f`. The input to the material tangent stiffness subroutine is similar the input to the material routine itself, but LS-DYNA also provides a flag `unsym` in case the unsymmetrical linear equation solver is active (by *LCPACK* = 3 on `*CONTROL_IMPLICIT_SOLVER`).

The subroutine `utanXX(v)` should, based on the input and according to the material law, provide the consistent[4] material tangent modulus. This is to be stored in the 6 × 6 matrix `es` (or matrices `dsave(nlq, 6, 6)` for a vectorized implementation). If the local coordinate system option for orthotropic materials (*IORTHO = 1*) is invoked for solid elements, then it should be expressed in this local system. For shell elements, it should be expressed in the co-rotational system defined for the current shell element. All transformations back to the global system are made by LS-DYNA after exiting the user-defined routine.

[4] Or ”best possible”

The parameter list for a scalar implementation of a user defined material tangent routine (in this case `utan41`) is:

```
      subroutine utan41(cm,eps,sig,epsp,hsv,dt1,unsym,capa,etype,tt,
     1 temper,es,crv,nnpcrv,failel,cma,qmat)
```

An overview of the parameters can be found in Table 2, with the main difference that for `utan`, almost all parameters except for `es` are purely input.

Since the subroutine `umatXX` has no information if an implicit or explicit analysis is taking place from the default parameters passed to it, one way of communicating this could be to let the `utanXX` subroutine set a material history variable (one entry in the `hsv` – array) as a flag. Then, in case some non-linear equation is solved (for example iterative radial return) different tolerances and iteration limits may be set to account for the higher accuracy requirements of an implicit analysis.

To be more formal, LS-DYNA computes the material stiffness matrix based on the constitutive modulus $C_{ikjl}^{\sigma T}$, relating the rate-of-deformation tensor to the Truesdell rate of Cauchy stress. The material stiffness matrix K^{mat} is expressed as

$$K_{iIjJ}^{\text{mat}} = \int_{\Omega} \frac{\partial N_I}{\partial x_k} C_{ikjl}^{\sigma T} \frac{\partial N_J}{\partial x_l} \mathrm{d}\Omega$$

where *N* denotes the Finite Element basis functions. For more details, see Ref. [3].

4.4 Useful predefined subroutines

In this Section, some of the pre-defined subroutines available within the LS-DYNA usermat package that may come in handy for different common tasks in user defined material subroutines are described. See also Appendix A of Ref. [1] for detailed descriptions of some of the subroutines. In addition, some generally useful subroutines for other user-defined interfaces are presented.

Often, related to the material models, evaluation of curves input in the keyword deck via the `*DEFINE_CURVE` or `*DEFINE_TABLE` keywords, for example hardening curves, will be required. For this, the subroutines `crvval` and `tabval` may be called, in order to conveniently and consistently retrieve the values from the curve (or table) data to the subroutine. For example, in order to evaluate a curve to retrieve the yield stress corresponding to the current accumulated effective strain, the following could be used:

```
      call crvval(crv,nnpcrv,lcid,epsp,sigy,h)
```

which will look up the curve given by ID `lcid` and return the yield stress at plastic strain `epsp` in the variable `sigy`, and in addition the slope of the curve (hardening modulus) in the variable `h`. Note that the curve ID should be passed as a float (`lcid` should be declared as `real`) which is convenient if for example element 5 of the material constants array `cm` should be the ID of a curve which specifies the yield stress as a function of plastic strain, then `cm(5)` can be passed directly to the subroutine `crvval`. The `crv` is the curve array and `nnpcrv` is the number of discretization points per curve; these data are

passed as parameters to the user material subroutine for the only purpose of evaluating curves, and should just be passed on to the look-up subroutines. The vectorized version of the call would be

```
call crvval_v(crv,nnpcrv,lcid,epsp_v,sigy_v,h_v,lft,llt)
```

where the plastic strain `epsp_v`, the yield stress `sigy_v`, and slopes `h_v` will be arrays of dimension `nlq`.

The `curvval` and `tabval` subroutines will extrapolate y – values for x-values beyond the last curve entry based on the slope of the last segment on the curve. See also Table 5 for an overview of the parameters to the subroutine `curvval`.

Table 5. The arguments to the subroutine `crvval`

Argument	Description	Input / Output
`crv`	Array representation of curves in the keyword deck	Input
`nnpcrv`	Number of discretization points per curve	Input
`lcid`	Curve id (from `*DEFINE_CURVE`) as float	Input
`epsp`	x – value	Input
`sigy`	y – value	Output
`h`	Slope of the curve in point epsp	Output

The parameter list for a scalar implementation of the table-lookup subroutine `tabval` is:

```
subroutine tabval(crv,nnpcrv,lcid,dxval,yval,dslope,xval,slope)
```

see Table 6 for an overview.

The curves are internally re-discretized (to the number of points specified by *`LCINT`* on the `*CONTROL_SOLUTION` – keyword, default is 100 points) in order to speed up the curve/table look up. If an evaluation based on the user-input curve data for curve (or table) ID `lcid` is desired, this can be requested by passing `-1.*lcid` to the subroutine.

Table 6. The arguments to the subroutine `tabval`

Argument	Description	Input / Output
`crv`	Array representation of curves in the keyword deck	Input
`nnpcrv`	Number of discretization points per curve	Input
`lcid`	Table id (from `*DEFINE_TABLE`) as float	Input
`dxval`	x_2 – value	Input
`yval`	y – value	Output
`dslope`	Slope $\partial y / \partial x_2$ of the curve in point (x1, x2)	Output
`xval`	x_1 – value	Input
`slope`	Slope $\partial y / \partial x_1$ of the curve in point (x1, x2)	Output

When working with hyperelastic[5] materials, there are some useful pre-defined subroutines for common operations (which will also be discussed in more detail related to the example of Section 4.5.1). By setting *IHYPER* = 1 on *MAT_USER_DEFINED_MATERIAL_MODELS, the deformation gradient **F** will be passed to the user subroutine in the history variables array, as 9 components right after the requested number of history variables (*NHV*).

The transformation of a tensor from the reference configuration to the current configuration by use of the deformation gradient **F** is commonly denoted as a push-forward operation (see for example Section 3 of Ref. [13]). The push-forward of a 2nd order tensor is performed by the subroutine `push_forward_2s` (or the vectorized version `subroutine push_forward_2`) which is useful when going from a material model formulated with respect to the 2nd Piola – Kirchhoff stress tensor **S** to Cauchy stress **σ**, which is the expected output from the user defined material subroutine. In that case, it is important to remember that

$$\boldsymbol{\sigma} = \frac{\mathbf{1}}{\det \mathbf{F}} \mathbf{F}\mathbf{S}\mathbf{F}^{T}$$

which means that after the subroutine `push_forward_2s` is called, also a division by $\det \mathbf{F}$ should be performed. The push-forward of a 4th order tensor (for example the material stiffness tensor) is performed by the subroutine `push_forward_4s` (or the vectorized version `subroutine push_forward_4`). Also, for working with shell elements and hyper-elastic materials, the subroutine `compute_f3s` updates the third row of the deformation gradient considering the through-thickness stretch (`eps(3)` or ε_{33} in the local coordinate system).

For solids elements, the LS-DYNA code will make the stress transformations required to obtain the objective Jaumann stress rate outside the user subroutine, but in cases where history variables also are stresses, for example the back stress tensor in a kinematic hardening model, the user must take care to apply the transformation to the history variables inside the user subroutine.

It is good practice to add some checking to the user subroutine, for example if the user tries to apply the material model to an unsupported element type, or if some of the input parameters are invalid, or if the material model is intended for explicit only. One option is to simply write a text message in the `mes0*` - files to inform the user, using

```
write(iomsg,*) 'Warning message …'
```

as outlined in Section 3.6.5. Another option is to issue an error message and stop the analysis. This can be done using the subroutine `lsmsg`. Calling this subroutine also requires that the file iouinits.inc is included. An example follows:

```
      include 'iounits.inc'
…
      if(etype.eq.'beam')then
         cerdat(1)=etype
         call lsmsg(3,MSG_SOL+1150,ioall,ierdat,rerdat,cerdat,0)
```

[5] Or a material using a deformation-gradient based formulation

```
         return
      endif
      if(cm(3).gt.0.5)then
         cerdat(1)='Illegal Poissons ratio'
         call lsmsg(3,MSG_SOL+1447,ioall,ierdat,rerdat,cerdat,0)
         return
      endif
```

which will output

```
*** Error 41151 (SOL+1151)
     element type beam can not be
     run with the current material model.
```

in case an attempt is made to apply the user defined material model to a beam element, and

```
*** Error 41447 (SOL+1447)
     Illegal Poissons ratio.
```

in case an invalid value is passed by the user as parameter 3 from the keyword file.

The relevant parameters to modify in the subroutine call are:

- `MSG_SOL + XXX`, for changing the message ID / type
- `cerdat(1)` to specify what error message to print.

The error message ID MSG_SOL + 1447 will display the text passed to `cerdat(1)` as in the second part of the above example.

Entities of the analysis model (nodes, parts, elements, etc.) will be stored using internal ID:s within the LS-DYNA code. These internal ID:s may differ from those specified by the user in the keywordfile. For example, if the user defines a node with ID 103, it may be assigned an internal ID of 1. In many cases, when handling user input to the user-defined interfaces, it will be required to convert between internal/external entity ID:s. Some useful functions for this purpose are listed in Table 7. In some cases, for example beam orientation nodes[6], an offset may be applied to the internal node numbers. An extra check is then required before converting internal to external node numbers:

```
if (i3.gt.1000000000) i3=i3-1000000000
```

[6] To obtain useful internal numbering of beam orientation nodes, it is required to set *NREFUP* = 1 on `*CONTROL_OUTPUT`.

Table 7. List of some functions for converting entity ID:s between internal and external (user defined) numbering

	Internal → External ID	External → Internal ID	
Entity		mpp	smp
Node	`lqfinv(ix,1)` (1)	`lqfe(ex,1)` (2)	`lqf8(ex,1)`
Element solid	`lqfinv(ix,2)`	`lqfe(ex,2)`	`lqf8(ex,2)`
Element beam	`lqfinv(ix,3)`	`lqfe(ex,3)`	`lqf8(ex,3)`
Element shell	`lqfinv(ix,3)`	`lqfe(ex,4)`	`lqf8(ex,4)`
Part	`lqfmiv(ipid)`	`lqfm(epid)`	
Curve	`ilcid(iid)`	`lcids(eid)`	

Notes: (1) There is also a function `lqfinv8` returing INTERGER*8. (2) For mpp/LS-DYNA, lqfe will return -1 in case an entity is not found in the current mpi thread.

4.5 Subroutine examples

In this Section, two examples of user defined material models are presented, with quite extensive descriptions. Both Fortran code and keyword input are presented, while the complete examples, see also Section 4.6, can be found as attachments to this document. It shall be stressed that these examples are not intended for use in any kind of production analysis, and they may very well contain errors or flaws.

The Fortran files (`dyn21umats.f`, `dyn21umatv.f` and `dyn21utan.f`) of the usermat package already come with some examples of subroutines for user defined materials (which may vary slightly depending on version) and some general routines that can be used as a starting point for user defined subroutines, for example `metalshl`, `metalsld` (in `dyn21umats.f`) and `metaltan` (in `dyn21utan.f`) for J_2 – plasticity. Also, Appendix A of Ref. [1] has some examples, including descriptions.

4.5.1 The Saint-Venant Kirchhoff model for solids and shells

This section describes the implementation of the Saint-Venant Kirchhoff model, a simple compressive isotropic hyperelastic material, for solids and shells. Using the Green-Lagrange strain tensor,

$$\mathbf{E} = \frac{1}{2}(\mathbf{F}^{\mathrm{T}}\mathbf{F} - \mathbf{I})$$

this model gives the 2nd Piola-Kirchhoff stress tensor **S** via a linear relation,

$$\mathbf{S} = \mathbb{C}\colon \mathbf{E}$$

where $\mathbb{C}$ is a fourth order stiffness tensor (this is basically what the subroutine `utanXX` should return). Using the Lamé constants μ and λ, the strain-energy function for the Saint-Venant Kirchhoff model is (see for example Section 6.5 of Ref. [13])

$$\Psi(\mathbf{E}) = \frac{\lambda}{2}(\mathrm{tr}\mathbf{E})^2 + \mu tr\mathbf{E}^2$$

which, by

$$\mathbf{S} = \frac{\partial \Psi}{\partial \mathbf{E}}$$

gives

$$S_{ij} = 2\mu E_{ij} + \lambda E_{kk}\delta_{ij}$$

The first step in implementing a material model, once the theory is known, is to consider the parameters to be input by the user on the `*MAT_USER_DEFINED_MATERIAL_MODELS` – card. In the present implementation, it was decided to let the user input Youngs modulus E and Poisson's ratio ν, since these elasticity parameters are often used in engineering applications, rather than the Lamé constants. By setting *IHYPER* = 1 on the `*MAT_USER_DEFINED_MATERIAL_MODELS` – card, the deformation gradient (in Voight – notation, with components $F11$, $F21$, $F31$, $F12$, $F22$, $F32$, $F13$, $F23$ and $F33$) will be passed to the user subroutine in the history variables array `hsv`, on positions `hsv(`*NHV*`+1:`*NHV*`+9)`. In this case, for a basic implementation of an elastic material, no history variables are required, so **F** can be retrieved from `hsv(1:9)`. There is no indication by the interface parameters to the user subroutine (see Table 2) if the *IHYPER* variable actually is set to one, so the user subroutine will depend on correct user input.

In this case, the keyword interface to the Sain-Venant Kirchhoff model will be

```
*MAT_USER_DEFINED_MATERIAL_MODELS_TITLE
Simple hyperelastic material UMAT43
$#1    MID        RO        MT       LMC       NHV    IORTHO     IBULK        IG
    mat_id         ρ        43         8                             3         4
$#2  IVECT     IFAIL    ITHERM    IHYPER      IEOS      LMCA    UNUSED    UNUSED
                                       1
$#      P1        P2        P3        P4        P5        P6        P7        P8
   Young's Poisson's         K         G    el.ID
```

where blue text indicates that the user should input sensical data, and red text indicates values that should not be changed (since these fixed values also will be assumed by the Fortran implementation). By specifying an element ID for `P5`, some debug output (written to the `mes0*` - files) will be activated.

The user subroutine `umat43` will be used to implement the material model in Fortran code. The first part of the subroutine follows:

```
      subroutine umat43 (cm,eps,sig,epsp,hsv,dt1,capa,etype,tt,
     1 temper,failel,crv,nnpcrv,cma,qmat,elsiz,idele,reject)
      include 'nlqparm'
      include 'bk06.inc'
      include 'iounits.inc'
      dimension cm(*),eps(*),sig(*),hsv(*),crv(lq1,2,*),cma(*)
      character*5 etype
      logical failel
C --
      real S(6), defgrad(3,3), green(3,3), detF, g, g2
      real p, davg, lam, sigold, epsold, tol, deps
      integer iter, limiter
      tol=1.E-7
      limiter=10
C - initial output
      if (ncycle.le.1) then
        if(cm(2).ge.5.00000E-01)then
           write(iomsg,*) 'mat43 --- illegal possions number,',cm(2)
           cerdat(1)='Illegal Poissons number'
           call lsmsg(3,MSG_SOL+1447,ioall,ierdat,rerdat,cerdat,0)
```

```
        endif
        if(idele.eq.int(cm(5)))then
          write(iomsg,*) 'mat43 --- my hyperelastic code. E=',cm(1),
     1      'nu=',cm(2),'capa=',capa
        endif
       endif
C -- Material parameters
       g2 =.5*abs(cm(1))/(1.+cm(2))
       lam = g2*cm(2)/(1.-2.*cm(2))
```

This part starts with the declaration of the subroutine, as described in Section 4.3, then some variable declarations follow. The local variables of the subroutine are also declared, for example `S(6)` is an array for storing the 2nd Piola-Kirchhoff stress tensor, `defgrad(3,3)` is an array for storing the deformation gradient **F**, and `detF` for its determinant. The variable `g2` is the shear modulus G or μ, and `lam` is λ/2. Some initial checking of input parameters is done, and if `cm(5)` gives an element ID, also a message will be printed in the `mes0*` - files, to confirm that the `umat43` is active.

The next part of the subroutine performs the stress update for solid elements. It starts with storing the deformation gradient in the matrix `defgrad` from the `hsv` array and computing its determinant.

```
C --  for solids
      if(etype.eq.'solid')then
C -- extract deformation gradient
        defgrad(1,1) = hsv(1)
        defgrad(2,1) = hsv(2)
        defgrad(3,1) = hsv(3)
        defgrad(1,2) = hsv(4)
        defgrad(2,2) = hsv(5)
        defgrad(3,2) = hsv(6)
        defgrad(1,3) = hsv(7)
        defgrad(2,3) = hsv(8)
        defgrad(3,3) = hsv(9)
c ---
        detF = defgrad(1,1)*defgrad(2,2)*defgrad(3,3)+
     1   defgrad(1,2)*defgrad(2,3)*defgrad(3,1)+
     2   defgrad(1,3)*defgrad(2,1)*defgrad(3,2)-
     3   defgrad(1,3)*defgrad(2,2)*defgrad(3,1)-
     4   defgrad(1,2)*defgrad(2,1)*defgrad(3,2)-
     5   defgrad(1,1)*defgrad(2,3)*defgrad(3,2)
```

In the final part of the stress update for solid elements, the Green-Lagrange (or to be exact 2**E**) strain is computed, and based on this the 2nd Piola-Kirchhoff stress, which is finally transformed to Cauchy stress via a push-forward operation followed by division by det **F**.

```
C -- compute 2*Green strain
        do j=1,3
          do i=1,j
            green(i,j)= sum(defgrad(:,i)*defgrad(:,j))
            green(j,i)= green(i,j)
          enddo
        enddo
        green(1,1) = green(1,1) - 1
        green(2,2) = green(2,2) - 1
```

```
        green(3,3) = green(3,3) - 1

        davg=lam*(green(1,1)+green(2,2)+green(3,3))
C -- Piola-Kirchhoff
        S(1)=g2*green(1,1)+davg
        S(2)=g2*green(2,2)+davg
        S(3)=g2*green(3,3)+davg
        S(4)=g2*green(1,2)
        S(5)=g2*green(2,3)
        S(6)=g2*green(1,3)
C --- push forward for Cauchy stress
        call push_forward_2s(S(1),S(2),S(3),S(4),S(5),S(6),hsv(1),
     1   hsv(2),hsv(3),hsv(4),hsv(5),hsv(6),hsv(7),hsv(8),hsv(9))
        sig(1:6) = S/detF
```

where the symmetry of **E** has been utilized in the nested do – loops. This concludes the stress update for solid elements. The strain and stress computations for shell elements are much more involved, since the condition $\sigma_{33} = 0$ (`sig(3) = 0`) needs to be fulfilled. An iterative procedure, following the "Sample user subroutine 45" of Ref. [1] Appendix A, is applied. The secant method is used, which requires two starting guesses. The first one is given by plane stress elasticity, with

$$\epsilon_{33} = -\frac{\nu}{1-\nu}(\epsilon_{11} + \epsilon_{22})$$

and the second staring guess is simply $\epsilon_{33} = 0$. The Fortran code follows:

```
      else if(etype.eq.'shell')then
C --- for shells
C --- secant iterations for zero z-stress, find eps(3)
      deps = 0.
      do iter=1,limiter
C first thickness strain increment initial guess
c assuming Poisson's ratio different from zero
c
        if (iter.eq.1) then
          eps(3)=-cm(2)*(eps(1)+eps(2))/(1.-cm(2))
c
c second thickness strain increment initial guess
c
        else if (iter.eq.2) then
          sigold=sig(3)
          epsold=eps(3)
          eps(3)=0.
c
c --- secant update of thickness strain increment
c
        else if (abs(sig(3)-sigold).gt.0.0) then
          deps=-(eps(3)-epsold)/(sig(3)-sigold)*sig(3)
          sigold=sig(3)
          epsold=eps(3)
          eps(3)=eps(3)+deps
        endif
c
c --- update last row of F
        call compute_f3s(hsv(3),hsv(6),hsv(9),eps(3))
```

After that follows computation of the Green-Lagrange strain and 2nd Piola-Kirchhoff stress, just as for solids:

```
C --- compute strain and stress
        defgrad(1,1) = hsv(1)
        defgrad(2,1) = hsv(2)
        defgrad(3,1) = hsv(3)
        defgrad(1,2) = hsv(4)
        defgrad(2,2) = hsv(5)
        defgrad(3,2) = hsv(6)
        defgrad(1,3) = hsv(7)
        defgrad(2,3) = hsv(8)
        defgrad(3,3) = hsv(9)
        detF = defgrad(1,1)*defgrad(2,2)*defgrad(3,3)+
     1   defgrad(1,2)*defgrad(2,3)*defgrad(3,1)+
     2   defgrad(1,3)*defgrad(2,1)*defgrad(3,2)-
     3   defgrad(1,3)*defgrad(2,2)*defgrad(3,1)-
     4   defgrad(1,2)*defgrad(2,1)*defgrad(3,2)-
     5   defgrad(1,1)*defgrad(2,3)*defgrad(3,2)
C -- compute Green strain
        do j=1,3
          do i=1,j
            green(i,j)= sum(defgrad(:,i)*defgrad(:,j))
            green(j,i)= green(i,j)
          enddo
        enddo
        green(1,1) = green(1,1) - 1
        green(2,2) = green(2,2) - 1
        green(3,3) = green(3,3) - 1

        davg=lam*(green(1,1)+green(2,2)+green(3,3))
C -- Piola-Kirchhoff stress
        S(1)=g2*green(1,1)+davg
        S(2)=g2*green(2,2)+davg
        S(3)=g2*green(3,3)+davg
        S(4)=g2*green(1,2)
        S(5)=g2*green(2,3)
        S(6)=g2*green(1,3)
```

The final transformation to Cauchy stress differs slightly, since the shear stress components σ_{23}, σ_{13} (`sig(5:6)`) are multiplied by `capa`, the reduction factor for transverse shear in shells (corresponding to *SHRF* of the `*SECTION_SHELL` – keyword). Also, in order to save some floating-point operations, the update of these stress components is moved outside the secant iterations `do` – loop. The final part of the code for shell elements follows:

```
C --- push forward to get Cauchy stress
        call push_forward_2s(S(1),S(2),S(3),S(4),S(5),S(6),hsv(1),
     1   hsv(2),hsv(3),hsv(4),hsv(5),hsv(6),hsv(7),hsv(8),hsv(9))
        sig(1:4) = S(1:4)/detF
C --- termination criteria
        if(abs(sig(3)).lt.tol*(abs(sig(1))+abs(sig(2))+abs(sig(4))))
     1    exit
      enddo
      sig(5:6) = capa*S(5:6)/detF
```

Finally, an error message will be issued in case attempts are made to apply this user defined material model to other element types than solids or shells:

```
      else
        cerdat(1)=etype
        call lsmsg(3,MSG_SOL+1151,ioall,ierdat,rerdat,cerdat,0)
      endif
      return
      end
```

which concludes the user defined material subroutine. For implicit analysis, also the tangent modulus must be computed. In this case, a push-forward of $\mathbb{C}$ is (more or less) what is required. The user subroutine `utan43` is used to implement the tangent modulus in Fortran code. The first part of the subroutine, which starts with the subroutine declaration according to Section 4.3.1, and some variable declarations follows:

```
   subroutine utan43(cm,eps,sig,epsp,hsv,dt1,unsym,capa,etype,tt,
     1 temper,es,crv,nnpcrv,failel,cma,qmat)
c
      include 'nlqparm'
      dimension cm(*),eps(*),sig(*),hsv(*),crv(lq1,2,*),cma(*)
      integer nnpcrv(*)
      dimension es(6,*),qmat(3,3)
      logical failel,unsym
      character*5 etype
      real*8 f1,f2,f3, defgrad(3,3), detF, detFinv
      real*8 dmx(6,6)
```

Just as for `umat43`, the deformation gradient **F** is stored in the `defgrad` matrix, and the determinant is computed:

```
c
      defgrad(1,1) = hsv(1)
      defgrad(2,1) = hsv(2)
      defgrad(3,1) = hsv(3)
      defgrad(1,2) = hsv(4)
      defgrad(2,2) = hsv(5)
      defgrad(3,2) = hsv(6)
      defgrad(1,3) = hsv(7)
      defgrad(2,3) = hsv(8)
      defgrad(3,3) = hsv(9)
      detF = defgrad(1,1)*defgrad(2,2)*defgrad(3,3)+
     1   defgrad(1,2)*defgrad(2,3)*defgrad(3,1)+
     2   defgrad(1,3)*defgrad(2,1)*defgrad(3,2)-
     3   defgrad(1,3)*defgrad(2,2)*defgrad(3,1)-
     4   defgrad(1,2)*defgrad(2,1)*defgrad(3,2)-
     5   defgrad(1,1)*defgrad(2,3)*defgrad(3,2)
      detFinv = 1.0 / max(detF, 1.E-12)
```

In the last line, some extra caution is taken when computing 1/det **F**, in order to avoid division by zero. The next step is to populate the `dmx` – matrix, which holds the stiffness tensor $\mathbb{C}$ (in the reference configuration). This is achieved by the following lines of code:

```
      f1 = cm(1)/(1.0+cm(2))/(1.0-2.0*cm(2))
```

```
      f2 = 1.0 - cm(2)
      f3 = 0.5 - cm(2)
      dmx = 0
      dmx(1,1) = f1*f2
      dmx(2,2) = dmx(1,1)
      dmx(3,3) = dmx(1,1)
      dmx(1,2) = f1*cm(2)
      dmx(1,3) = dmx(1,2)
      dmx(2,3) = dmx(1,2)
      dmx(2,1) = dmx(1,2)
      dmx(3,1) = dmx(1,3)
      dmx(3,2) = dmx(2,3)
C ---
      dmx(4,4) = f1*f3
      dmx(5,5) = dmx(4,4)
      dmx(6,6) = dmx(4,4)
```

The transformation to the current configuration is done by

```
C --- push forward
      call push_forward_4s(dmx,hsv(1),
     1   hsv(2),hsv(3),hsv(4),hsv(5),hsv(6),hsv(7),hsv(8),hsv(9))
      es(1:6, 1:6) = dmx * detFinv
```

In addition, in order to account for the transverse shear reduction `capa` for shells the modification

```
c --- special for shell
      if(etype.eq.'shell')then
        es(5,5) = capa * dmx(5,5) * detFinv
        es(6,6) = capa * dmx(6,6) * detFinv
      endif
      return
      end
```

is made, which also concludes the tangent modulus subroutine.

4.5.2 J_2-plasticity for solids and shells

In this Section, the implementation of a hypoelastic-plastic material model with isotropic hardening for solids and shells is described. For a detailed theoretical background, see for example Ref. [11] and especially Section 17.4.1 for details on the derivation. The yield condition is

$$f(\boldsymbol{\sigma}, \bar{\epsilon}_p) = \sigma_{vM} - \sigma_y(\bar{\epsilon}_p) = 0$$

where $\bar{\epsilon}_p$ is the accumulated effective plastic strain, and σ_{vM} is the von Mises effective stress, which is directly proportional to the norm of the stress deviator **s** given by

$$s_{ij} = \sigma_{ij} - \delta_{ij}\frac{\sigma_{kk}}{3}$$

The von Mises effective stress is then given by

$$\sigma_{vM} = \sqrt{\frac{3}{2}s_{ij}s_{ij}}$$

The hardening function $\sigma_y(\bar{\epsilon}_p)$ is often given as a piecewise linear curve, determined from material testing.

An efficient implementation of J_2 – plasticity is obtained by the radial return algorithm [4] [7] [11][21]. The starting point is the stress and strain at state 1, and the objective is to compute stress and increment of effective plastic strain at stat 2, given the strain increment $\Delta\epsilon_{ij}$. First, an elastic trial stress is computed according to

$$\boldsymbol{\sigma}^t = \boldsymbol{\sigma}^{(1)} + \mathbb{C}\colon \Delta\boldsymbol{\epsilon}$$

where $\mathbb{C}$ is the isotropic elastic stiffness tensor. From the elastic trial stress, the von Mises effective stress is calculated as

$$\sigma_{vM}^t = \sqrt{\frac{3}{2} s_{ij}^t s_{ij}^t}$$

If the effective trial stress is below the yield limit, $f(\boldsymbol{\sigma}^t, \bar{\epsilon}_p) \leq 0$, the elastic trial stress is accepted, and no further action is needed. In case yielding is indicated, the increment in effective plastic strain $\Delta\bar{\epsilon}_p$ to satisfy the yield criterion must be determined. For solid elements, this can be done, following Box 17.5 of Ref. [11], by solving

$$f_p = \sigma_{vM}^t - 3G\Delta\bar{\epsilon}_p - \sigma_y\left(\bar{\epsilon}_p^{(1)} + \Delta\bar{\epsilon}_p\right) = 0$$

using Newton's method, that is

1. Set $\Delta\bar{\epsilon}_p^1 = 0$
2. Compute $\Delta\bar{\epsilon}_p^{k+1} = \Delta\bar{\epsilon}_p^k - \frac{f_p^k}{df_p^k / d\bar{\epsilon}_p^k} = \Delta\bar{\epsilon}_p^k + \frac{f_p^k}{3G+H}$ where $H = \frac{d\sigma_y(\bar{\epsilon}_p^k)}{d\bar{\epsilon}_p^k}$
3. Compute f_p^{k+1}
4. If $\left|f_p^{k+1}\right| > tol$ then let $k = k+1$ and go to 2.
5. Update stress and accumulated effective plastic strain:

$$\bar{\epsilon}_{\mathrm{p}}^{(2)} = \bar{\epsilon}_{\mathrm{p}}^{(1)} + \Delta\bar{\epsilon}_p$$
$$s_{ij}^{(2)} = \frac{\sigma_y^{(2)}}{\sigma_{vM}^t} s_{ij}^t$$
$$\sigma_{ij}^{(2)} = s_{ij}^{(2)} + \frac{1}{3}\sigma_{kk}^t \delta_{ij}$$

In the case of linear hardening, where *H* = constant, this procedure will converge exactly in one iteration. From step 2 of the Newton scheme, it can also be noted that the slope of the hardening curve in practice will be limited by

$$3G + H > 0 \Leftrightarrow H > -3G$$

which means that from a mathematical viewpoint, a negative slope of the hardening curve can be tolerated, as long as it fulfills this condition.

For shell elements, some additional modifications are needed to fulfill $\sigma_{33} = 0$ and account for the reduction factor for transverse shear. This will be discussed in more detail in the context of the Fortran coding for shell elements below.

The keyword interface for this material model should allow for input of elastic constants (Young's modulus E and Poisson's ration ν) and a hardening curve. In addition, the user is given some control over the Newton iterations outlined above, with respect to the required tolerance and the maximum allowed number of iterations. Since `epsp` already is a separate parameter in the interface to the user subroutine, see Table 2, no history variables are, strictly speaking, required in this case. Still, one history variable will be used to indicate if yielding takes place or not. This will be used by the tangent stiffness routine `utan` later on. In all, this means that the keyword interface for the J_2 – plasticity model will be

```
*MAT_USER_DEFINED_MATERIAL_MODELS_TITLE
J2 plasticity by UMAT41
$#1    MID        RO        MT       LMC       NHV    IORTHO     IBULK        IG
    mat_id         ρ        41         8                             3         4
$#2  IVECT     IFAIL    ITHERM    IHYPER      IEOS      LMCA    UNUSED    UNUSED

$#      P1        P2        P3        P4        P5        P6        P7        P8
   Young's Poisson's         K         G      LCID       tol   limiter     el.ID
```

where blue text indicates that the user should input sensical data, and red text indicates values that should not be changed (since these fixed values also will be assumed by the Fortran implementation). The curve ID of the hardening curve (`*DEFINE_CURVE`) is input as `P5`. The tolerance for the Newton iterations is optionally input as `P6` and the maximum number of allowed iterations is optionally input as `P7`. For additional debug output, an element ID may be specified as `P8`.

It shall be mentioned that the LS-DYNA usermat package already contains the subroutines `metalshl`, `metalsld` and `metaltan` (in `dyn21umats.f` and `dyn21utan.f`) which are ready-to-use subroutines for J2 – plasticity (but have no Newton iterations for the radial return).

The user subroutine `umat41` will be used to implement the material model in Fortran code. The first part of the subroutine follows:

```
      subroutine umat41 (cm,eps,sig,epsp,hsv,dt1,capa,etype,tt,
     1 temper,failel,crv,nnpcrv,cma,qmat,elsiz,idele,reject)
c
c************************************************************
c|  Livermore Software Technology Corporation  (LSTC)       |
c|  ---------------------------------------------------     |
c|  Copyright 1987-2008 Livermore Software Tech. Corp       |
c|  All rights reserved                                     |
c************************************************************
c
c     isotropic elastic-plastic material
c
c     Variables
c
c     cm(1)=first material constant, here young's modulus
c     cm(2)=second material constant, here poisson's ratio
c        .
c        .
c        .
c     cm(n)=nth material constant
c
```

Here, many lines of detailed comments of the original Fortran file are omitted. The comments of the files of the usermat package in general provide valuable information in connection with the respective subroutine.

After this follows some variable declarations and initializations. The array `sshl`(6) will hold temporary values of the stress tensor. The arrays `s(6)` and `s2(6)` will be used for stress deviator values. The von Mises effective stress will be stored in the variable `vonMises`. The history variable `hsv(1)` will indicate if yielding takes place or not, as a way of communicating this to the `utan41` subroutine.

```
      include 'nlqparm'
      include 'bk06.inc'
      include 'iounits.inc'
      dimension cm(*),eps(*),sig(*),hsv(*),crv(lq1,2,*),cma(*),qmat(3,3)
      integer nnpcrv(*)
      logical failel,reject
      character*5 etype
      integer idele
c
      real vonMises,h,sigy,s(6),depsp,gc,tol, ep_tr
      real sig3e,sig3p, eps3e, eps3p, sshl(6), f1, f2
      real s2(6),p2,f3, strainlim
      integer iter, limiter
```

Then tolerances (`tol`) and iteration limit (`limiter`) are initialized, with their default values, but if the user provides reasonable input in `cm(6)` and `cm(7)`, it replaces the default values. The `strainlim` is a (presently hard-coded) limit on the increment of accumulated effective plastic strain ($\Delta\bar{\epsilon}_p$ above); for implicit, a reject will be issued in case this limit is exceeded. After that, an initial message is written to the `mes0*` - files, to confirm that the `umat41` is active. Finally, some material constants are computed, and the yield limit for the current value of effective plastic strains is evaluated by `crvval`.

```
      tol=1.E-4
      limiter=10
      if(cm(6).gt.0.)
     1  tol=cm(6)
      if(cm(7).gt.0.)
     1  limiter = int(cm(7))
      strainlim = 5.E-2
c
      if (ncycle.le.1) then
        if(idele.eq.int(cm(8)))then
          write(iomsg,*) 'mat41:iterative elastoplastic code. E=',cm(1),
     1      'nu=',cm(2),'lcid=',cm(5)
          write(iomsg,*) 'mat41:limiter =',limiter,'tol=',tol
        endif
      endif
c
c     compute shear modulus, g
c E is cm(1), pr is cm(2)
c
      g2 =abs(cm(1))/(1.+cm(2))
      g  =.5*g2
      gc =capa*g
```

```
      hsv(1) = 0.
C --  Yield curve val and tangent h : starting vals
      call crvval(crv,nnpcrv,cm(5),epsp,sigy,h)
```

Next follows the implementation for solid elements, starting with computation of the elastic trial stress:

```
      if(etype.eq.'solid')then
C
C     --- For solids (iterative) ---
C     Compute elastic trial stress
c
        davg=-sum(eps(1:3))/3.
        p=-davg*abs(cm(1))/(1.-2.*cm(2))
        sig(1:3)=sig(1:3)+p+g2*(eps(1:3)+davg)
        sig(4:6)=sig(4:6)+g*eps(4:6)
C --- Effective stress, first the deviatoric
        s(1:3) = sig(1:3)-sum(sig(1:3))/3.
        s(4:6) = sig(4:6)
c     compute the von Mises stress
        vonMises = s(1)**2+s(2)**2+s(3)**2+2.*(s(4)**2+s(5)**2+s(6)**2)
        vonMises = sqrt(1.5*vonMises)
        if (vonMises.le.sigy) then
          return
        else
          hsv(1)= 1.
        endif
```

At this point, the subroutine will return in case of an elastic response. In the following, the iterative radial return scheme outlined above is implemented. In the convergence check, the tolerance is multiplied by the current yield stress to obtain a relative measure. The iterations will be aborted if the convergence criterion is met, otherwise `limiter` iterations will be performed, and a message is printed. Also, an extra message is written for user feedback.

```
C - radial return, iterative
      ep_tr=epsp
      depsp = 0.
      f1 = vonMises - sigy
      do iter=1,limiter
        depsp = depsp + f1/(h+3.*g)
C     re-eval hardening curve
        ep_tr = epsp + depsp
        call crvval(crv,nnpcrv,cm(5),ep_tr,sigy,h)
        f1 = vonMises - 3.*g*depsp - sigy
        if(abs(f1).lt.tol*sigy)
     1    exit
      enddo
C --- debug
      if(iter.ge.limiter)then
        write(iomsg,*) 'mat41:iter=',iter,'idele=',idele,
     1     'stressdiff=',abs(sigy-vonMises)
      endif
      if(idele.eq.int(cm(8)))then
        write(iomsg,*) 'mat41:iter=',iter,'idele=',idele,
     1     'stressdiff=',abs(sigy-vonMises),'f1*sigy=',f1*sigy,
     2     'depsp=',depsp
```

```
      endif
```

Finally, the stress is updated, and this concludes the part of the subroutine for solid elements. The check for the increment in effective plastic strain is performed after the stress update, since the reject option is only active for implicit analysis.

```
      epsp = ep_tr
      s2 = sigy*s/vonMises
      p2 = sum(sig(1:3))/3.
      sig(1:3) = s2(1:3)+p2
      sig(4:6) = s2(4:6)
      if(depsp.gt.strainlim)then
        reject = .true.
      endif
```

Then comes the part of the implementation for shell elements. Due to the requirement of zero normal stress, this becomes more involved. In addition, considering the reduction factor for transverse shear `capa` also complicates the calculations. The present approach uses a two-step scheme, where first the through-thickness strain ε_{33} is estimated by linear interpolation between two extrema. Then, the stress state is determined using a slightly different approach for the radial return algorithm based on the estimate of ε_{33}. This solution approach for shells is adopted from the subroutine `metalshl` already present in the usermat package.

First, the elastic trial stress is computed, and yielding is checked.

```
      elseif(etype.eq.'shell')then
C
C     --- For shells (iterative) ---
C     Compute elastic trial stress and eps3
c
        eps(3) = -cm(2)*(eps(1)+eps(2))/(1.-cm(2))
        davg=-sum(eps(1:3))/3.
        p=-davg*abs(cm(1))/(1.-2.*cm(2))
        sshl(1:2)=sig(1:2)+p+g2*(eps(1:2)+davg)
        sshl(3)=0.
        sshl(4)=sig(4)+g*eps(4)
        sshl(5:6)=sig(5:6)+gc*eps(5:6)
C --- Effective stress, first the deviatoric
        s(1:3) = sshl(1:3)-sum(sshl(1:2))/3.
        s(4:6) = sshl(4:6)
c     compute the von Mises stress
        vonMises = s(1)**2+s(2)**2+s(3)**2+2.*(s(4)**2+s(5)**2+s(6)**2)
        vonMises = sqrt(1.5*vonMises)
C --  check yield
        if(vonMises.le.sigy)then
          sig(1:6) = sshl(1:6)
          return
        endif
```

Again, if the response is elastic the subroutine returns. In the following, the required updates of stresses, effective plastic strain and through-thickness strain due to the plastic deformation are performed. The first step is to estimate the through-thickness strain. This is done from linear interpolation between the first elastic estimate, and a fully plastic estimate.

```
C --- secant iterations for eps(3) estimate
C     radial return from elastic
        f1=4.5*g*(capa-1.)*(sshl(5)**2+sshl(6)**2)/(vonMises**2)
        depsp = (vonMises - sigy)/(h+3.*g+f1)
        sig3e =         - 3.*g*depsp*s(3)/vonMises
        eps3e = eps(3)
C --- first point is (eps3e, sig3e)
C     second point, radial return from plastic
        eps3p =  -eps(1)-eps(2)
        sshl(1:2)=sig(1:2)+g2*eps(1:2)
        sshl(3) = g2*eps3p
C --- Effective stress, first the deviatoric
        s(1:3) = sshl(1:3)-sum(sshl(1:3))/3.
c     compute the von Mises stress
        vonMises = s(1)**2+s(2)**2+s(3)**2+2.*(s(4)**2+s(5)**2+s(6)**2)
        vonMises = sqrt(1.5*vonMises)
C --  check yield, radial return if required
        if(vonMises.ge.sigy)then
          f1=4.5*g*(capa-1.)*(sshl(5)**2+sshl(6)**2)/(vonMises**2)
          depsp = (vonMises - sigy)/(h+3.*g+f1)
          sshl(3) = sshl(3) - 3.*g*depsp*s(3)/vonMises
        endif
C --- second point is (eps3p, sshl(3))
C     linear interpolation between p.1 and p.2
       if (abs(sig3e-sshl(3)).gt.tol*
     1      max(abs(sig3e),abs(sig3p))) then
         eps(3)=eps3e-sig3e*(eps3e-eps3p)/(sig3e-sshl(3))
       else
         eps(3)=eps3p
       endif
```

In this case, the variable `f1` holds extra terms of $\frac{d\sigma_{vM}}{d\Delta\bar{\epsilon}_p}$ due to the transverse shear reduction factor. It vanishes in the case `capa` = 1. After this, it assumed that ε_{33} is known, and it remains to determine the corresponding stress state. This starts with again computing an elastic trial stress:

```
c
c     now we have estimate for eps(3): update stresses
c
       davg=-sum(eps(1:3))/3.
       p=-davg*abs(cm(1))/(1.-2.*cm(2))
       sig(1:2)=sig(1:2)+p+g2*(eps(1:2)+davg)
       sig(3)=0
       sig(4)=sig(4)+g*eps(4)
       sig(5:6)=sig(5:6)+gc*eps(5:6)
C --- Effective stress, first the deviatoric
       s(1:3) = sig(1:3)-sum(sig(1:3))/3.
       s(4:6) = sig(4:6)
c     compute the von Mises stress
       vonMises = s(1)**2+s(2)**2+s(3)**2+2.*(s(4)**2+s(5)**2+s(6)**2)
       vonMises = sqrt(1.5*vonMises)
       if (vonMises.le.sigy) then
         return
       else
          hsv(1)= 1.
```

```
       endif
      ep_tr=epsp
      depsp = 0.
      f2 = 3.*g/vonMises
      f3 = f2*capa
```

Then, as the final step for shell elements, iterative radial return is performed. For shells, a different approach than for solids is used, where the stresses are updated in each iteration. This makes it easier to account also for the transverse shear reduction factor. If the `limiter` iterations are not enough to reach the specified tolerance, a message is printed.

```
      do iter=1,limiter
C --- radial return
        f1=4.5*g*(capa-1.)*(sig(5)**2+sig(6)**2)/(vonMises**2)
        depsp = depsp + (vonMises - sigy)/(h+3.*g+f1)
        sshl(1:2)= sig(1:2)-f2*depsp*s(1:2)
        sshl(3)=0
        sshl(4)= sig(4)-f2*depsp*s(4)
        sshl(5:6)= sig(5:6)-f3*depsp*s(5:6)
c --  compute the von Mises stress
        s2(1:3) = sshl(1:3)-sum(sshl(1:3))/3.
        vonMises = sum(s2(1:3)**2)+2.*sum(sshl(4:6)**2)
        vonMises = sqrt(1.5*vonMises)
C     re-eval hardening curve
        ep_tr = epsp + depsp
        call crvval(crv,nnpcrv,cm(5),ep_tr,sigy,h)
C --- check convergence
        if(abs(sigy-vonMises).lt.tol*abs(sigy))
     1    exit
       enddo
C --- debug
       if(iter.ge.limiter)then
         write(iomsg,*) 'mat41:iter=',iter,'idele=',idele,
     1     'stressdiff=',abs(sigy-vonMises)
       endif
       sig(1:6)=sshl(1:6)
       epsp = ep_tr
       if(depsp.gt.strainlim)then
         reject = .true.
       endif
```

Most likely, the above implementation for shell element can be improved, both for increased efficiency and accuracy. The subroutine ends with printing an error message, in case attempts are made to apply the material model to other element types than solids or shells.

```
C --- unsupported element formulation ==> Error termination
      else
        cerdat(1)=etype
        call lsmsg(3,MSG_SOL+1151,ioall,ierdat,rerdat,cerdat,0)
      endif
      end
```

For implicit analysis, also the tangent modulus is required. The user subroutine `utan41` is used to implement the tangent modulus in Fortran code. The implementation for solids is based on the results presented in Section 12.2 of Ref. [11], that is

$$C_{ijkl}^{ep} = C_{ijkl} - \frac{9G^2}{H+3G}\frac{s_{ij}s_{kl}}{\sigma_{vM}^2}$$

For shells, an approximate tangent modulus is applied: basically, the same as for solid elements is used, only some minor corrections for the transverse shear reduction are made. The first part of the subroutine, which starts with the subroutine declaration according to Section 4.3.1, and some variable declarations follows:

```
  subroutine utan41(cm,eps,sig,epsp,hsv,dt1,unsym,capa,etype,tt,
     1 temper,es,crv,nnpcrv,failel,cma,qmat)
c
c*************************************************************
c|  Livermore Software Technology Corporation  (LSTC)         |
c|  ------------------------------------------------------    |
c|  Copyright 1987-2008 Livermore Software Tech. Corp         |
c|  All rights reserved                                       |
c*************************************************************
c
      include 'nlqparm'
      include 'bk06.inc'
      include 'iounits.inc'
      dimension cm(*),eps(*),sig(*),hsv(*),crv(lq1,2,*),cma(*)
      integer nnpcrv(*)
      dimension es(6,*),qmat(3,3)
      logical failel,unsym
      character*5 etype
c
      real*8 factor, g,b, bg23,bg43
      real*8 vonMises,h,sigy,s(6),sf(6),depsp,gc,tol, ep_tr
      real*8 sig3e, sig3p, eps3e, eps3p, sshl(6), f1
      real*8 dpfac, A, shrf
      integer k, l
c
      factor=1.
      if (failel) factor=1.e-8
```

This section ends with a stiffness reduction for elements that are indicated as failed. Then, some elastic constants are computed, element type is checked, and an attempt to make a small modification for shells is made:

```
      g=factor*.5*abs(cm(1))/(1.+cm(2))
      b=factor*abs(cm(1))/3./(1.-2.*cm(2))
      bg23=b-2.*g/3.
      bg43=b+4.*g/3.
c
      if(etype.eq.'solid')then
        shrf=1.
      elseif(etype.eq.'shell')then
        shrf=capa
      else
        cerdat(1)=etype
        call lsmsg(3,MSG_SOL+1151,ioall,ierdat,rerdat,cerdat,0)
        return
      endif
```

This means that if the routine is applied to other element types than solids or shells, LS-DYNA will stop with an error message. The next step is the elastic part of the tangent stiffness modulus:

```
es(1,1)=bg43
es(2,2)=bg43
es(3,3)=bg43
es(2,1)=bg23
es(3,1)=bg23
es(3,2)=bg23
es(1,2)=es(2,1)
es(1,3)=es(3,1)
es(2,3)=es(3,2)
es(4,4)=g
es(5,5)=g*shrf
es(6,6)=g*shrf
```

and finally, the plastic part, which only is required in case the material routine `umat41` indicated that yielding takes place, by setting `hsv(1)` = 1.

```
if(hsv(1).gt.0.)then
  call crvval(crv,nnpcrv,cm(5),epsp,sigy,h)
  s(1:3) = sig(1:3) - sum(sig(1:3))/3.
  s(4:6) = sig(4:6)
  vonMises =s(1)**2+s(2)**2+s(3)**2+2.*(s(4)**2+s(5)**2+s(6)**2)
  vonMises = sqrt(1.5*vonMises)
  A = h + 3.*g
  dpfac = (9.*g**2)/A/(vonMises**2)
  sf = dpfac*s
  do k=1,6
    do l=k,6
      es(k,l) = es(k,l) - sf(k)*s(l)
      es(l,k) = es(k,l)
    enddo
  enddo
endif
```

In this implementation, the symmetry of the tangent stiffness matrix has been utilized in the reduced inner `DO` – loop. This concludes the subroutine `utan41` for the tangent modulus. Clearly, since the tangent is based on the implementation for solid elements, convergence properties for shell elements are not as good.

4.5.3 Non-linear spring

This section describes the implementation of a simple material for discrete beam elements (*ELFORM* = 6 on `*SECTION_BEAM`) representing a (optionally non-linear) spring. The material can either represent a linear relationship between force and beam elongation,

$$F_r = k\delta$$

or a non-linear relationship given by a curve,

$$F_r = f(\delta).$$

In case the curve is only defined for positive elongation, the material model will use

$$F_r = f(|\delta|)sign(\delta).$$

This material model is similar to a mix of the built-in materials `*MAT_LINEAR_ELASTIC_DISCRETE_BEAM` (MAT_66) and `*MAT_NONLINEAR_ELASTIC_DISCRETE_BEAM` (MAT_67).

The keyword interface for this material model should allow for input of either a constant stiffness *k* or a curve ID. This is done by the parameter P1: if P1 > 0, it is assumed to be a constant stiffness value, and if P1 < 0, it is assumed that the curve ID is |P1|. The user should also provide reasonable values for *K* and *G*, which are important for calculating the explicit time step.

```
*MAT_USER_DEFINED_MATERIAL_MODELS_TITLE
Spring material by UMAT44
$#1    MID        RO        MT        LMC       NHV     IORTHO     IBULK        IG
    mat_id         ρ        44          8         4                  3         4
$#2  IVECT     IFAIL    ITHERM    IHYPER      IEOS      LMCA    UNUSED    UNUSED

$#      P1        P2        P3        P4        P5        P6        P7        P8
        p1                  K         G
```

where blue text indicates that the user should input sensical data, and red text indicates values that should not be changed (since these fixed values also will be assumed by the Fortran implementation).

The user subroutine `umat44` will be used to implement the material model in Fortran code. The first part of the subroutine follows:

```
      subroutine umat44 (cm,eps,sig,epsp,hsv,dt1,capa,etype,tt,
     1 temper,failel,crv,nnpcrv,cma,qmat,elsiz,idele,reject)
c
c************************************************************************
c|  Livermore Software Technology Corporation  (LSTC)                    |
c|  ---------------------------------------------------------------     |
c|  Copyright 1987-2008 Livermore Software Tech. Corp                    |
c|  All rights reserved                                                  |
c************************************************************************
c
      include 'nlqparm'
      include 'bk06.inc'
      include 'iounits.inc'
      dimension cm(*),eps(*),sig(*),hsv(*),crv(lq1,2,*),cma(*),qmat(3,3)
      integer nnpcrv(*)
      character*5 etype
      logical failel,reject
      integer*8 idele
c
      real*8 stiff,lcid,yfval
      integer*8 iid
```

The first rows are the standard declarations for a user-defined material subroutine. After that the declaration of the model-specific local variables follow. The variable `stiff` holds the constant stiffness value, the variable `lcid` holds the curve ID and `yfval` is the current force at the current elongation, evaluated from the curve. The integer variable `iid` is the interval ID of the curve.

```
      if (ncycle.eq.1) then
        write(iomsg, *) 'User defined mat44 for discrete beams'
        if(cm(1)>0.d0)then
          write(iomsg, *) '---discrete beam found.Constant stiffness'
          write(iomsg, *) '   useing k factor:',cm(1)
        else
          write(iomsg, *) '---discrete beam found.Curve'
          lcid=abs(cm(1))
          iid=lcids(nint(lcid))
          hsv(4)=crv(1,1,iid)
          write(iomsg, *) '   using Curve ID:',nint(abs(cm(1)))
          if(hsv(4).lt.0.d0)then
            write(iomsg, *) '   curve exists also for negative x vals'
          else
            write(iomsg, *) '   reflected curve will be used for ',
     1           'negative x vals'
          endif
        endif
      endif
```

Then follows some initial checks and output of messages to the user, performed at cycle 1. The main check is if a constant stiffness or a curve ID is to be used, and if the curve also is defined for negative (compressive) elongation. This latter check is done by directly inspecting the first ordinate value of the curve, `crv(1,1,iid)`, and storing it to history variable #4. The internal ID of the curve is found by `lcids(nint(lcid))`.

Then follows the force calculations, for discrete beams only:

```
      if (etype.eq.'dbeam') then
```

In case a constant stiffness is used, the force update is uncomplicated.

```
        if(cm(1).ge.0.d0)then
          stiff=cm(1)
          sig(1)=sig(1)+eps(1)*stiff
        else
```

If a curve is used, different actions must be taken in case the elongation is negative or compressive. In case of a positive elongation, the curve can be evaluated directly:

```
          lcid=abs(cm(1))
          hsv(1)=hsv(3)+eps(1)
          if(hsv(1).gt.0.d0)then
            call crvval(crv,nnpcrv,lcid,hsv(1),yfval,stiff)
          else
```

The current elongation of the beam is stored in history variable #1, based on the previous elongation which is stored in history variable #3. Using history variables for storage of the elongation is also useful if a continuation of analysis using a `dynain` – file is to be performed.

If the elongation is compressive, we must check if the curve also exists for compression. If so, the curve can be evaluated directly:

```
            if(hsv(4).lt.0.d0)then
```

```
            call crvval(crv,nnpcrv,lcid,hsv(1),yfval,stiff)
          else
```

Otherwise, the absolute value of the elongation is passed to `crvval`, and the sign of the obtained force is reversed.

```
            call crvval(crv,nnpcrv,lcid,abs(hsv(1)),yfval,stiff)
            yfval=-1.d0*yfval
          endif
        endif
        sig(1)=yfval
      endif
```

The remaining force components are set to zero. The spring only gives a force in the r – direction, between n_1 and n_2, see Figure 4.

```
      sig(2)=0.0
      sig(3)=0.0
      sig(4)=0.0
      sig(5)=0.0
      sig(6)=0.0
```

The accumulated elongation is stored in history variable #3, based on the current time `tt` and the time from the previous call to the routine stored in history variable #2.

```
      if(tt.ne.hsv(2))then
         hsv(3)=hsv(1)
      endif
      hsv(2)=tt
```

In case the user tries to apply the material model to other elements than discrete beams, an error message is issued.

```
    else
      cerdat(1)=etype
      call lsmsg(3,MSG_SOL+1150,ioall,ierdat,rerdat,cerdat,0)
    endif
    return
    end
```

Since user-defined material models cannot be used for discrete beam elements in implicit, the computation of a corresponding tangent modulus is not required.

4.6 Ansys LS-DYNA simulation examples

In this Section, some simulation examples to demonstrate and verify the user defined material models of Sections 4.5.1 - 4.5.3 are presented. Comparisons to results obtained using the pre-defined material models of LS-DNYA are made. All LS-DYNA keyword files are supplied as attachments to this guide.

4.6.1 Examples of the Saint-Venant Kirchhoff material model

The first example applies the hyperelastic material model of `umat45` to an explicit analysis using solid elements. Material parameters typical for steel were used, that is a Young's modulus of 200 GPa and a

Poisson's ratio of 0.5. A ¼ - model of a bar (20x20x100 mm) is subjected to alternating prescribed displacement, see Figure 5. The stress vs. strain in response in one element is shown in Figure 6, compared to results using LS-DYNA's built-in hypoelastic material `*MAT_ELASTIC` and to the hyperelastic material `*MAT_ORTHOTROPIC_ELASTIC`. For stresses above 1 %, the response from `umat45` differ noticeable from `*MAT_ELASTIC`, which also is expected [2], while the agreement to the hyperelastic material `*MAT_ORTHOTROPIC_ELASTIC` (red dots in Figure 6) is very good throughout the whole strain range.

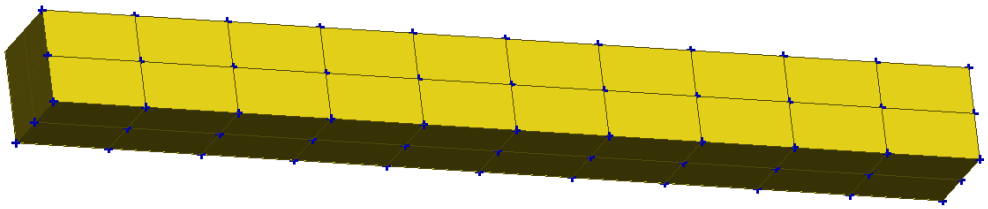


Figure 5. A ¼ model of a solid bar. The model consists of 40 solid elements. The blue symbols indicate symmetry boundary conditions applied at the nodes. One short end is fixed, while the other short end is subjected to a pulsating prescribed displacement.

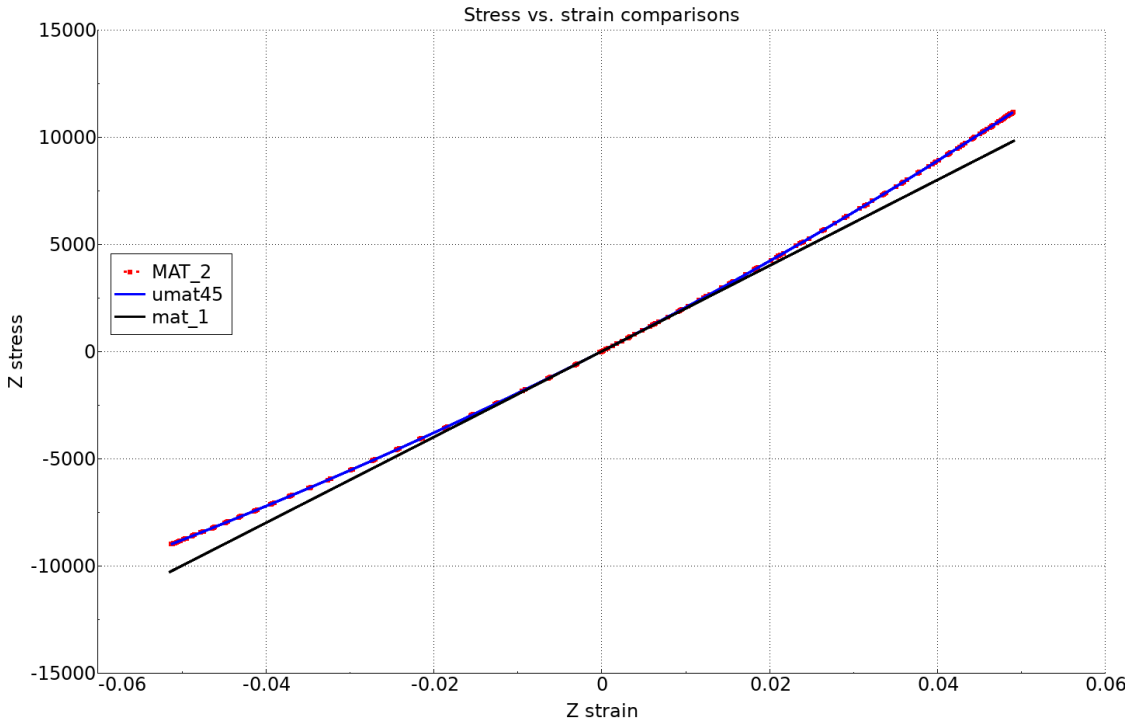


Figure 6. Stress vs. strain response for the hyperelastic Saint-Venant Kirchhoff model compared to the hypoelastic material `*MAT_ELASTIC` and the hyperelastic `*MAT_ORTHOTROPIC_ELASTIC`, in an explicit analysis.

In the second example, shell elements are used to model a cantilever beam, see Figure 7. Elastic material properties typical for aluminum (E = 70 GPa, ν = 0.31) are used. A prescribed displacement of 140 mm is applied at the bolt holes of the end bracket. Contact is considered between the square beam and the cylindrical rigid support, using `*CONTACT_AUTOMATIC_SINGLE_SURFACE_MORTAR`. Results from implicit analyses using `umat45` and `*MAT_ELASTIC` are compared in Figure 8 and Figure 9. Similar results are obtained. Differences are expected due to the differences in material model formulations. Also, similar performance with respect to iteration count and solution time is obtained for `umat45` and `*MAT_ELASTIC` for the implicit case.

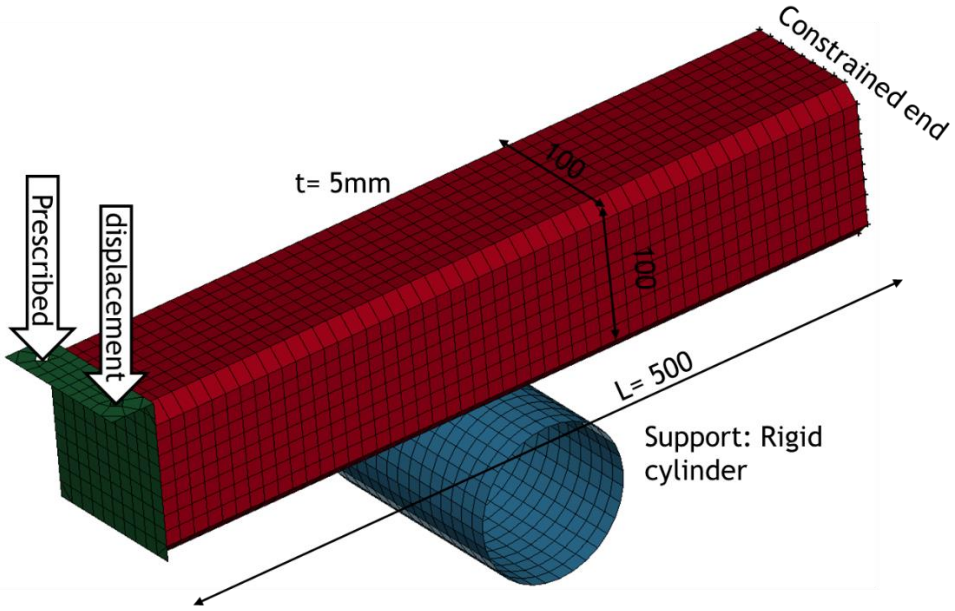


Figure 7. A square (100 × 100 mm, t = 5 mm) cantilever beam is subjected to prescribed displacement at the end bracket (green in the image) and contact with a rigid cylindrical support.

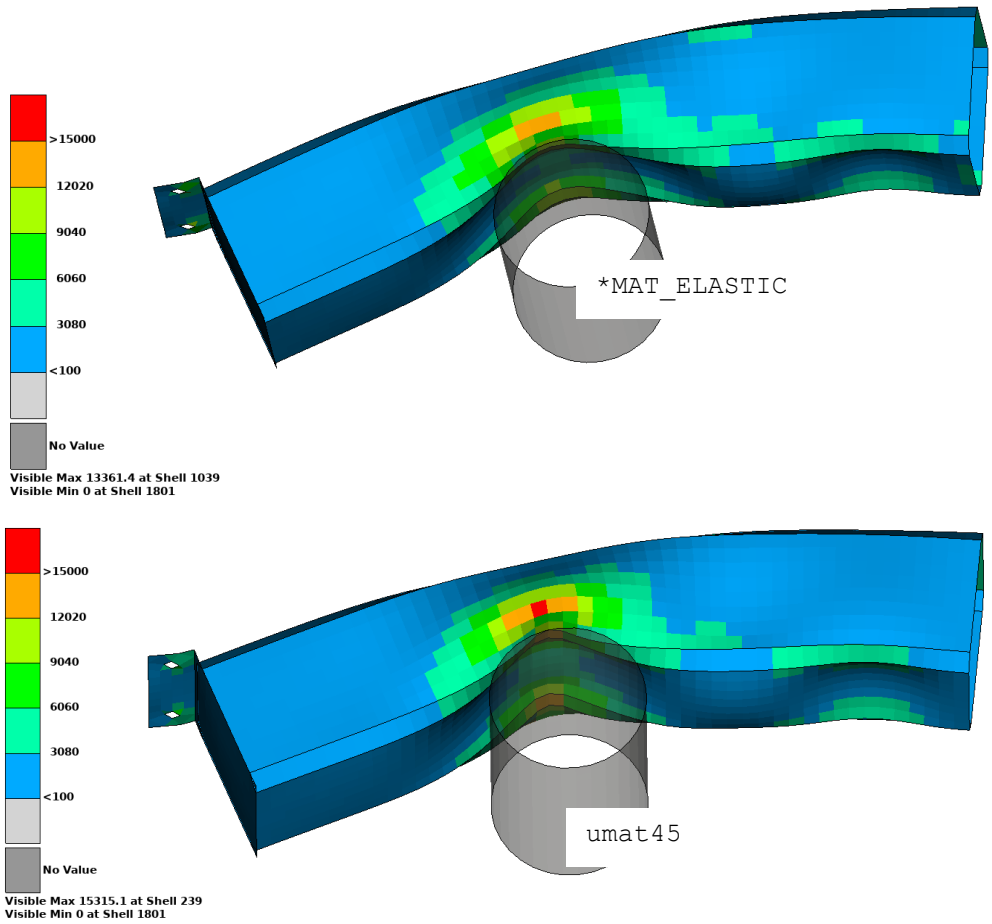


Figure 8. Fringe plot of von Mises effective at 140 mm displacement. The top image shows results using `*MAT_ELASTIC`, and the bottom image shows results using `umat45`.

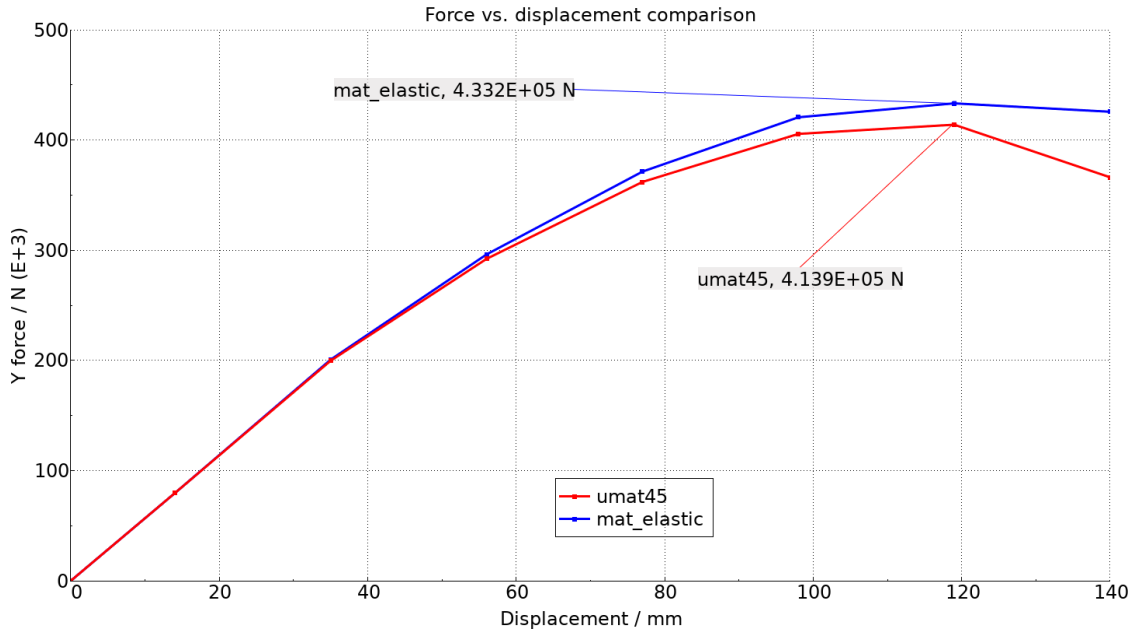


Figure 9. Comparison of global force vs. displacement response using `*MAT_ELASTIC` (blue curve) and `umat45` (red curve).

4.6.2 Examples of the J_2 – plasticity model

The first example is analysis of a tensile specimen, using shell elements, see Figure 10. Analyses were performed both using the explicit and implicit solver of Ansys LS-DYNA. Force vs. displacement results are compared in Figure 11. The peak forces for MAT_24 and `umat41` are very similar, both for implicit and explicit, while after necking some differences are found. Since necking is a local instability, small differences will be magnified after this point. The results for the explicit analyses are quite close up to about 20 mm of displacement, while the implicit umat41 results differ more. This may be due to the approximate nature of the implemented tangential stiffness matrix for shells, or insufficient accuracy in the through-thickness strain calculation. These analyses were performed with the update of shell thickness active (by *ISTUPD* = 4 on `*CONTROL_SHELL`).

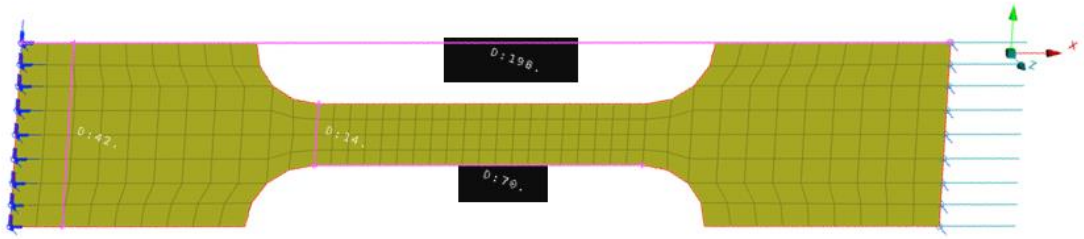


Figure 10. A tensile test specimen with gauge length ~ 70 mm, and width 14 mm. Thickness is 1.5 mm.

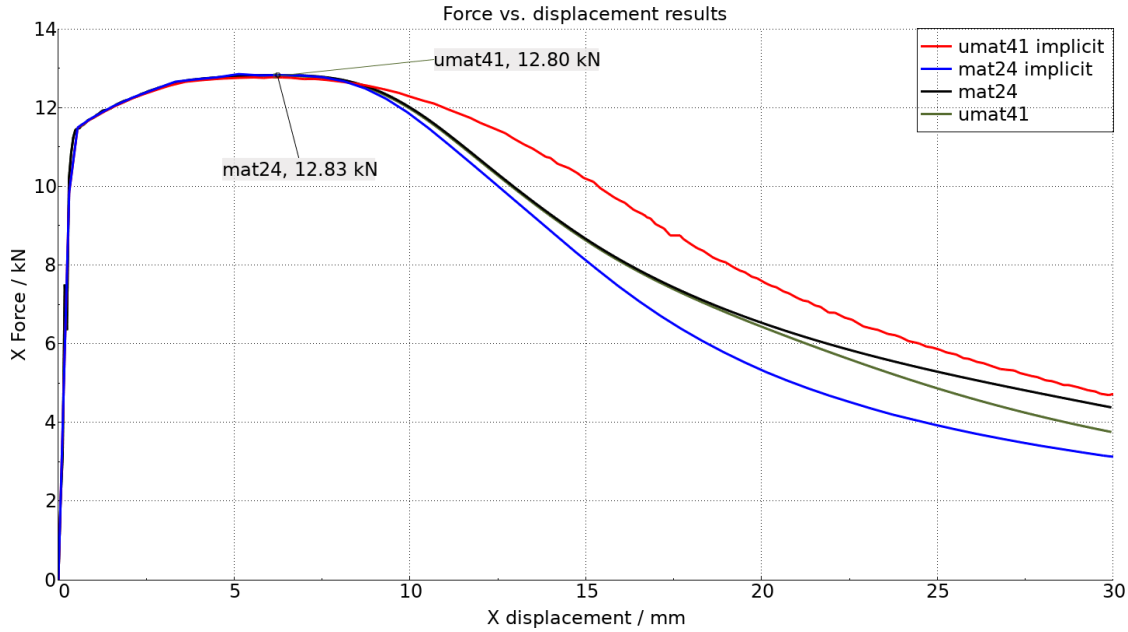


Figure 11. Force vs. displacement results for the tensile test specimen using different material models and the implicit (blue and red curves) or explicit (green and black curves) solver of LS-DYNA.

The second example is axial crushing of a crash box, see Figure 12. The thickness of the profiles is 1.5 mm, and a hardening curve corresponding to steel HX420LAD is applied. For `umat41`, it was necessary to disable the shell thickness update (by *ISTUPD* = 0 on `*CONTROL_SHELL`) in order to obtain useful results. Instabilities caused premature termination when the shell thickness update flag was active, most likely due to too low accuracy when solving the through-thickness strain. The crushing force

using MAT_24 and `umat41` are compared in Figure 13. Due to the indeterminate nature of this load case, it is far from ideal for a benchmark, and exact agreement is hardly to be expected. However, the large deformations and high plastic strain values pose a great challenge to the material model, and with some modification of the control card settings, also the umat41 manages to handle this load case. With respect to the crushing force, reasonable agreement is obtained, while the final deformed configuration differs quite substantially, see Figure 14. The solution time using mpp/LS-DYNA with 8 processes[7] is 1 h 42 min using MAT_24 and 2 h 25 min using the umat41, corresponding to an increase of about 42 % in this example. This should be seen as a rough indication of the effect of using a user defined material model on the solution time for an explicit analysis; probably the implementation of J2-plasticity presented here as umat41 is not optimal, while MAT_24 is one of the most efficient material models in LS-DYNA.

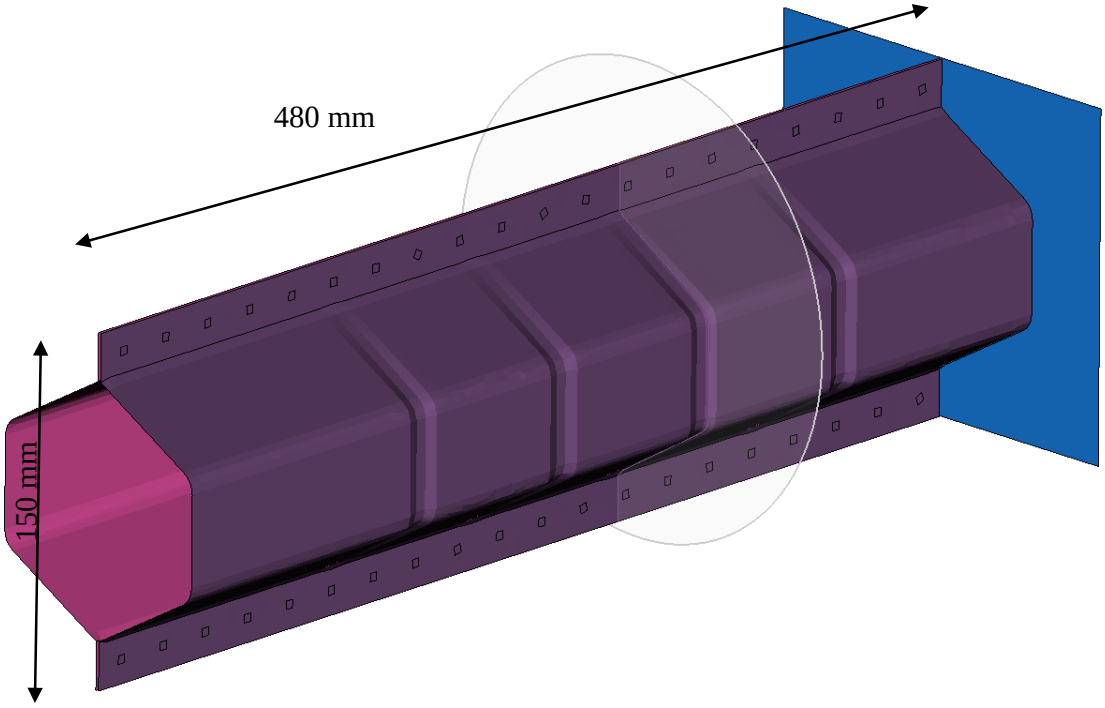


Figure 12. The crash box model. The profiles are made out of 1.5 mm thick HX420LAD steel, and the end plate (blue in the image) is made out of 2.7 mm HX340LAD. The open end of the profile is fully constrained.

[7] Intel Xeon E5-2687W v4 CPU (from 2017).

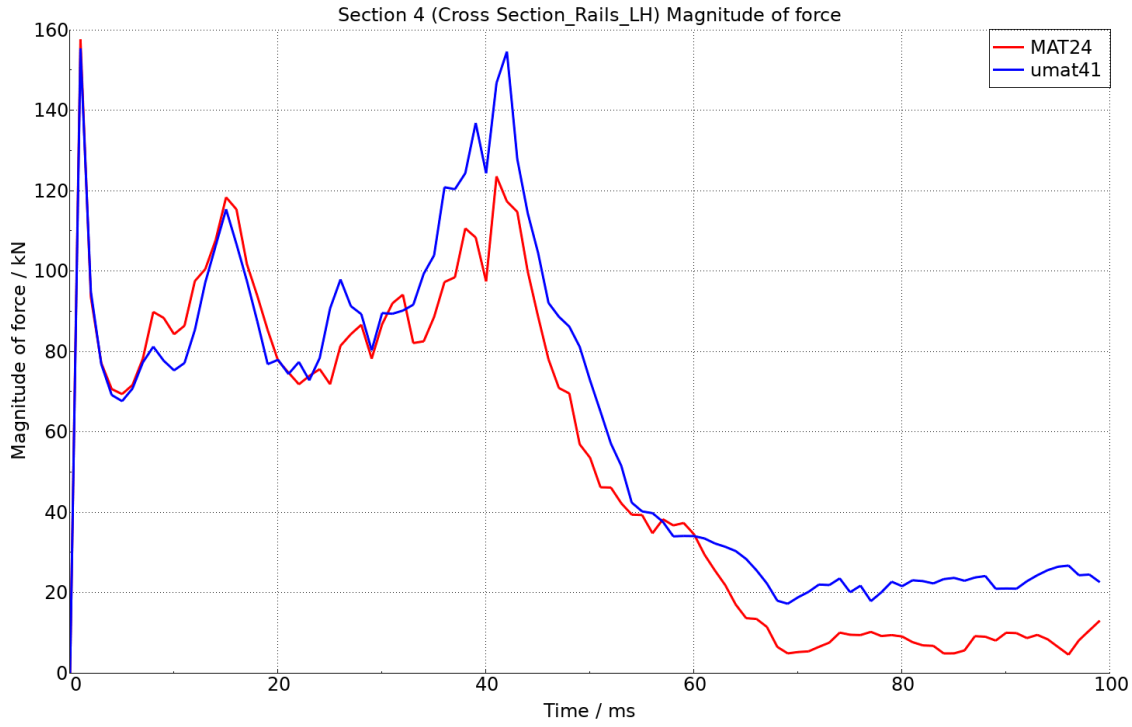


Figure 13. Comparison of the crushing force of the crash box using MAT_24 and `umat41`.

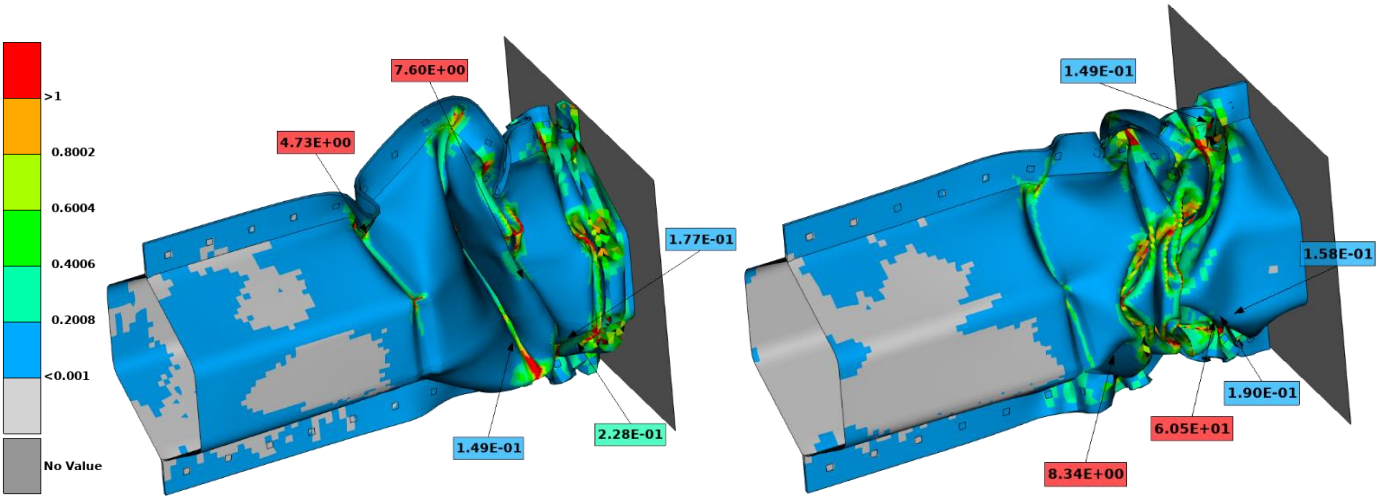


Figure 14. The final deformation of the crash box. The fringe colors show the accumulated effective plastic strain using `umat41` (left image) and MAT_24 (right image).

The final example of J_2 – plasticity consists of two pipes (∅ 90 mm, t = 10 mm) connected by a flange joint with five bolts (M10, strength class10.9), see Figure 15. The geometry is meshed using solid elements, and the example involves bolt pre-tensioning and contacts. Pipe 2 is fully constrained at the free end (black dots to the left in Figure 15). After the bolt pre-tensioning is completed, a prescribed

displacement is applied to the CNRB (blue, to the right Figure 15) of Pipe 1. This load case is solved using the implicit solver in LS-DYNA.

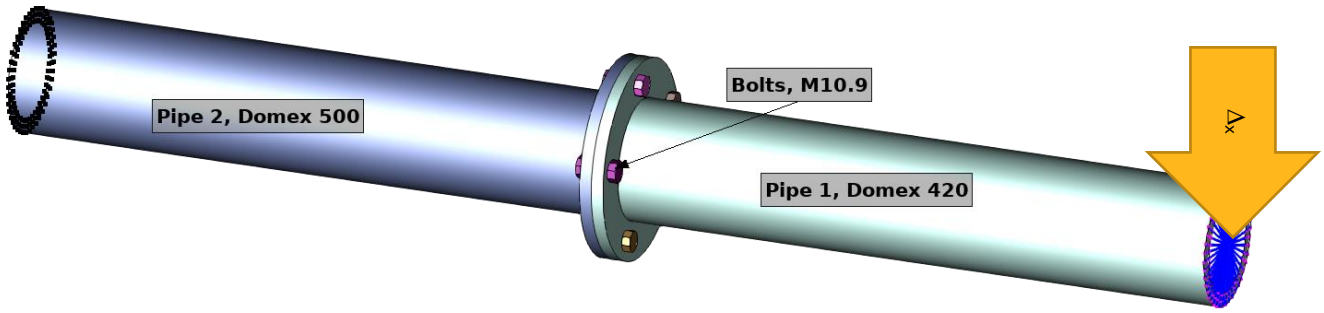
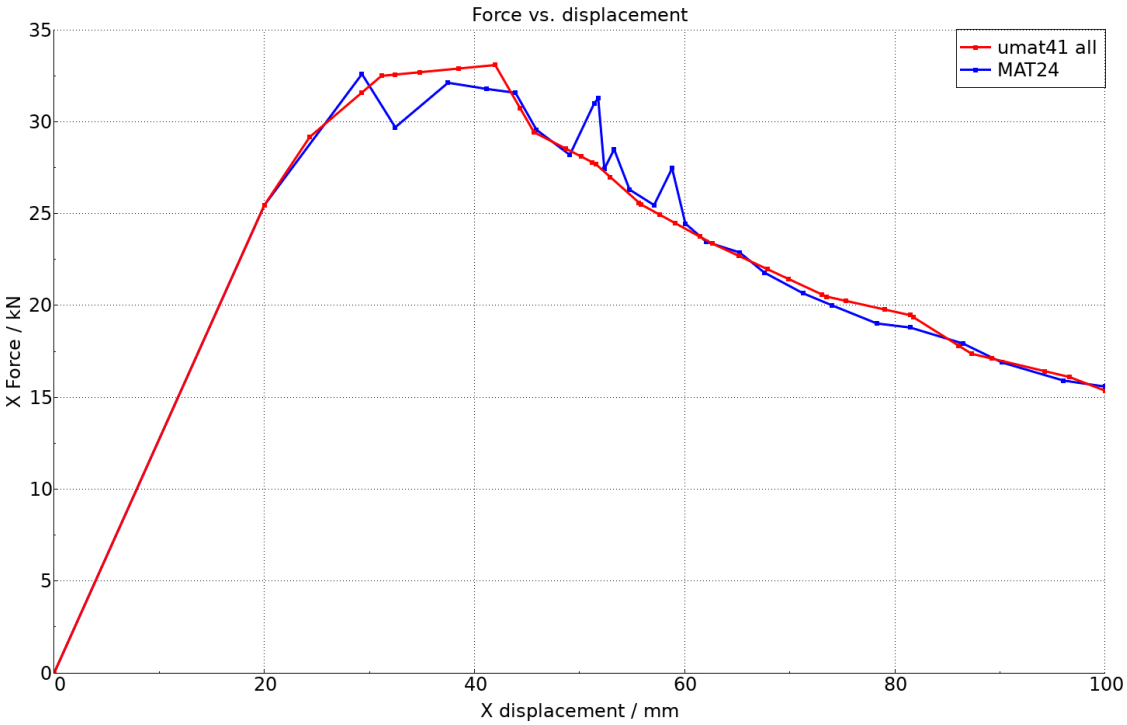


Figure 15. Two pipes are connected by five bolts in a flange joint.

Analyses were performed using `umat41` and MAT_24 for reference. The global force vs. displacement results are compared in Figure 16. The results are in general agreement, with a slightly smoother response for the umat41. This is probably because more time steps are taken at critical stages (which is most likely triggered by the choice of `strainlim = 5.E-2` in the user subroutine, see Section 4.5.2). The final configurations are compared in Figure 17. The peak accumulated effective plastic strain occurs in one of the bolts, and it is 1.73 using MAT_24 and 1.71 using `umat41`. The solution time using mpp/LS-DYNA with 4 processes[8] is approximately the same, 10 minutes, for both MAT_24 and umat41.



[8] Intel Xeon SP 6148 CPU (from 2018).

Figure 16. Force vs. displacement for the pipe joint model.

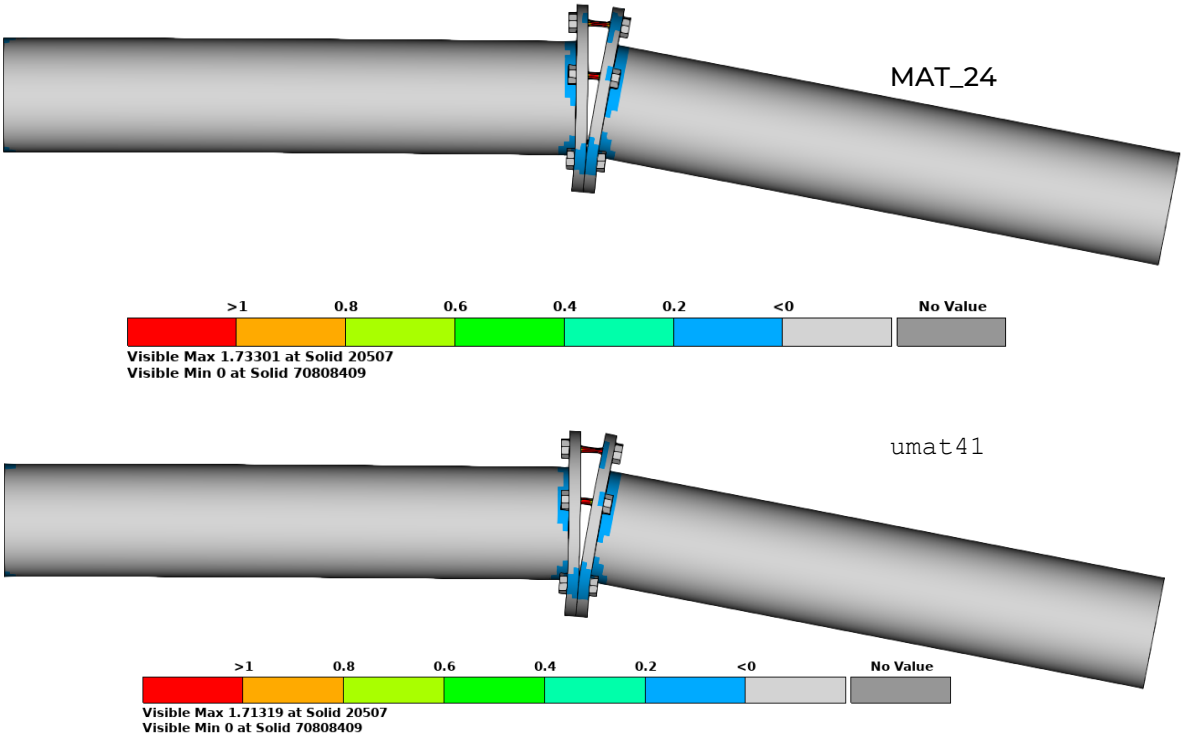


Figure 17. Final configuration of the pipe joint model. The fringe colors show accumulated effective plastic strain. The top image shows results using MAT_24, and the bottom image shows results using `umat41`.

4.6.3 Example of the non-linear spring material model

This basic example applies the non-linear spring material model of Section n4.5.3 to a seesaw-like model, see Figure 18. Five non-linear springs (discrete beams) are attached to one end of the solid beam (brown in Figure 18, 1350 hexas) via constrained nodal rigid bodies. The input force-displacement curve for the non-linear springs is shown in Figure 19. A linearly increasing loading is applied to a constrained nodal rigid body at the other end of the beam during 100 ms. The peak applied loading is 1 kN. The solid beam is constrained at 2/3 between the ends.

The force vs. displacement response in one of the springs is shown in Figure 20, compared to the response from LS-DYNA's built-in non-linear spring material model MAT_67. The two models show good agreement.

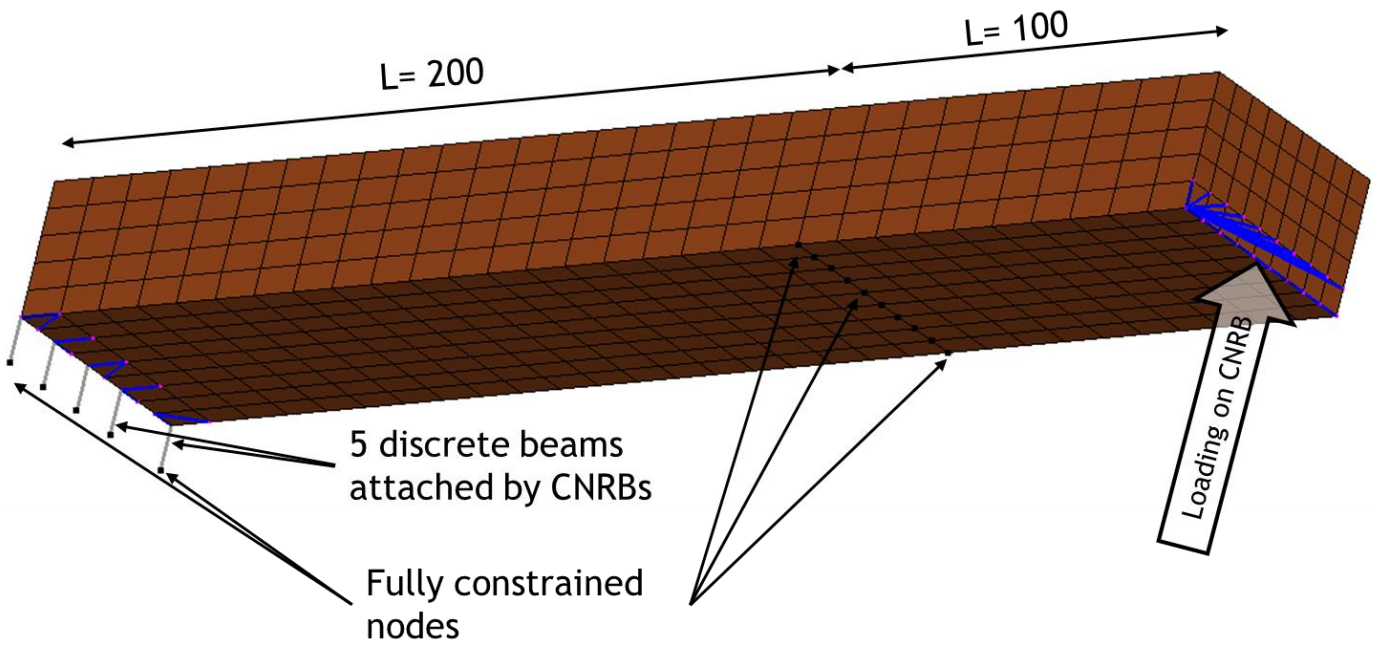


Figure 18. A simple FE-model for demonstrating the user-defined material model for discrete beams.

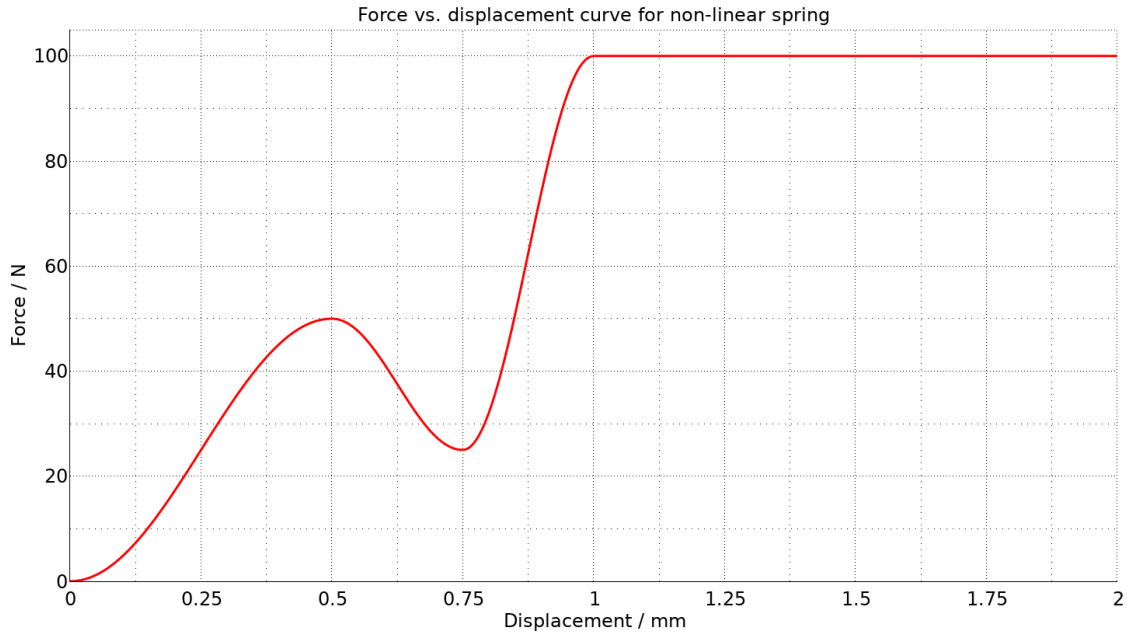


Figure 19. Non-linear force-displacement curve, input to the discrete beam material model.

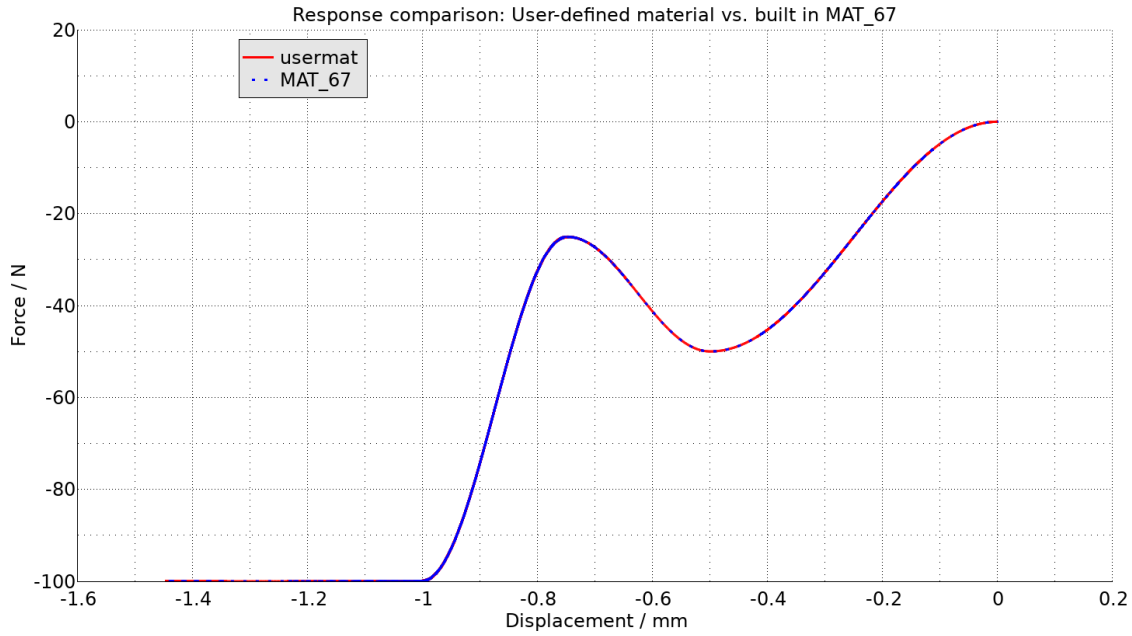


Figure 20. Comparison of the force-displacement response from the user-defined material model and the built-in material model MAT_67.

5 Friction models interface

Ansys LS-DYNA offers a variety of different ways to handle contact between model entities, see for example Ref. [8] [9]. The related keywords start with `*CONTACT_`, see Ref. [1]. A crucial component in any sliding contact is the definition of friction. The standard friction models in LS-DYNA for 3D contacts include:

- Static and dynamic friction coefficients
- Viscous damping
- A cap shear stress, typically related to the yield stress of the materials involved, limiting the peak tangential force in a contact
- Friction coefficients depending on contact pressure and/or temperature
- Orthotropic friction (for some contacts only)

The offering for 2D contacts is limited to static and dynamic friction coefficients with a cap shear stress.

Friction coefficients can be defined per contact interface, per part using `*PART_CONTACT`, or via interaction tables (`*DEFINE_FRICTION`).

The user defined friction interface makes it possible to develop customized and general friction models for some of the different 3D contacts in LS-DYNA. Due to the internal architecture of the LS-DYNA code, different subroutines are required for the smp[9], mpp and Mortar contact formulations. User defined friction does not apply to the non-Mortar segment-based contacts (*SOFT* = 2) and also not for the non-

[9] Since the subroutine for smp/LS-DYNA is more complex than the other options, it is left out of the current presentation. It will be included in coming revisions of this document.

Mortar single surface contacts (but *CONTACT_AUTOMATIC_SINGLE_SURFACE_MORTAR is supported). Customized friction models are also provided by some third-party companies, for example Triboform [28].

It is currently not possible to define a user friction law for 2D contacts (*CONTACT_2D_).

Note that the user subroutines for user defined friction are called in each explicit time step or implicit iteration, but only for the segments in contact at that particular time.

See also Appendix G of the LS-DYNA Keyword manual [1].

5.1 Keyword interface to the user defined friction models

To activate a user defined friction model from a keyword file, three steps are required:

1. On Card 2 of *CONTROL_CONTACT, set *USRFRC* to the number of parameters passed to the user defined friction model, plus the number of history variables that are stored.
2. To make the user defined friction model active for a specific contact ID, the keyword *USER_INTERFACE_FRICTION is used.
    a. This also implies that the option *CONTACT_..._ID must be used, in order to assign an ID to a specific contact definition.
    b. Set the variable *IFID* (interface number) to the Contact ID of the *CONTACT definition
    c. The extent of material history variables passed to the user friction routines is determined by the *NEHIS* variable. By setting *NEHIS* = 0 (which is the default), the plastic strain, yield stress and material directions will be passed. By setting *NEHIS* > 0, the plastic strain and the element history variables up to *NEHIS* -1 (in original order) will be passed.
3. For the user defined friction to have effect on a specific contact interface, a non-zero static friction (*FS*) must be defined for that contact interface. Also, for the non-automatic contacts, the shell thickness offset must be activated. This is done by setting *SHLTHK* = 1 or 2 on *CONTROL_CONTACT, or on Optional Card B of the *CONTACT_... keyword.

To specify different friction models for different contact interfaces in a subroutine (usrfrc or mortar_usrfrc), it might be convenient to let the first user defined input parameter denote a reference number to a specific friction model.

A keyword example follows:

```
*USER_INTERFACE_FRICTION
$#     IFID       NOC      NOCI      NHSV     NEHIS      MHSV
        35         2         2         1         0         0
$#      UC1       UC2       UC3       UC4       UC5       UC6       UC7       UC8
         1.    3501.0
*CONTACT_AUTOMATIC_SURFACE_TO_SURFACE_MORTAR_ID
$#      cid                                                                title
        35Block to base
$#1   SURFA     SURFB  SURFATYP SURFBSTYP                           SAPR      SBPR
         2         5         3         3                              1         1
$#       fs        fd        dc        vc       vdc    penchk        bt        dt
       0.15
$#     sfsa      sfsb ...
```

The variables of the *USER_INTERFACE_FRICTION keyword are:

- *IFID*: The ID of the *CONTACT to be affected. In this case, the user defined friction shall apply to Contact ID 35 (Block to base).
    - If a Mortar contact is referred by the ID, the subroutine mortar_usrfrc will be called, otherwise the subroutine usrfc will be called.
- *NOC*: The number of variables to be stored for the interface.
- *NOCI*: The number of variables to be initialized by the user (variables *UC1*, *UC2* ... etc.). *NOCI* must be smaller or equal to *NOC*.
- *NHSV*: The number of history variables per interface node. For Mortar contact it is the number of history variables per tracked[10] segment.
- *NEHIS*: The number of material history variables to be passed to the subroutine usrfrc.
- *MHSV*: The number of history variables per reference[11] segment for Mortar contact (ignored by *CONTACT_AUTOMATIC_SINGLE_SURFACE_MORTAR_ID).
- *UC1, UC2 etc.* : Parameters to be passed to the user subroutine.

5.2 Post processing user defined friction models

The history variables of the user defined friction models can be post-processed from the intfor[12] – file using LS-PrePost 4.8 (or later). To be able to fringe plot the history variables, it is required to set *SPR* = 1 on the *CONTACT_ - card in question (setting also *MPR* = 1 is recommended). On the keyword *DATABASE_EXTENT_INTFOR, specify the number of friction history variables to be written to the intfor file using the *NHUF* parameter. Finally, the keyword *DATABASE_BINARY_INTFOR_FILE is required to specify the filename (intfor is recommended) and output frequency of the contact data. See Figure 25 for an example of a fringe plot of a user defined friction history variable.

5.3 Interfaces to the user defined friction subroutines

Depending on the type of *CONTACT_ (non-Mortar or Mortar) that gets a user defined friction model associated with it, either the subroutine usrfrc or mortar_usrfrc is called. They are both found in the Fortran file dyn21cnt.f. In both cases, curve data as defined by the keyword input is passed via the parameters crv and nnpcrv, in a similar way as for the user defined material routines.In order to evaluate curves, the subroutine crvval as described in Section 4.4 may be used. The smp and mpp versions of the subroutine usrfric differ quite substantially. Currently, only the mpp version will be discussed in this presentation.

The subroutine usrfrc is called for defining the friction coefficients in non-Mortar contacts (*SOFT* = 0 or 1). The subroutine definition was extended with additional arguments, adding input of more detailed temperature information, starting with R14 of LS-DYNA. For versions R11 to R13, the subroutine definition for implementing a user defined friction model for non-Mortar contacts in mpp/LS-DYNA is

```
      subroutine usrfrc(fstt,fdyn,uc,nc,prs,temp,v,vx,vy,vz,uh,nh,
     . crv,nnpcrv,nosl,
     . ictype,side,time,ncycle,dt2,fric1,fric2,fric3,fric4,lsv,idele8,
```

[10] Previously denoted as ”slave”.
[11] Previously denoted as ”master”.
[12] This is a file for 3D visualization of contact results, for example contact pressure.

```
     . sfac1,sfac2,insv,fni,areas,stfk,ix1,ix2,ix3,ix4,aream,
     . rn1,rn2,rn3,ue,ne,uhnew)
```

For R14 and R15, the subroutine definition instead is

```
    subroutine usrfrc(fstt,fdyn,uc,nc,prs,temp,v,vx,vy,vz,uh,nh,
     . crv,nnpcrv,nosl,
     . ictype,side,time,ncycle,dt2,fric1,fric2,fric3,fric4,lsv,idele8,
     . sfac1,sfac2,intr,fni,areat,stfk,ix1,ix2,ix3,ix4,arear,
     . rn1,rn2,rn3,ue,ne,uhnew,ttrs,trfs,flxrfs)
```

and an overview of the parameters to the subroutine is shown in Table 8.

The main objective of the subroutine is to compute user defined frictional coefficients, that shall be output in the variables `fstt` (static) and `fdyn` (dynamic).

Table 8. Overview of the arguments for the `usrfrc` subroutine

Argument	Description	Input / Output
`fstt`	static frictional coefficient	Output
`fdyn`	dynamic frictional coefficient	Output
`uc(nc)`	user defined friction parameters	Input
`nc`	number of user defined friction parameters	Input
`prs`	interface pressure on reference side	Input
`temp`	temperature[1]	Input
`v`	magnitude of relative tangential velocity	Input
`vx,vy,vz`	components of relative tangential velocity	Input
`uh(nh)`	user defined friction history variables	Input/Output
`uhnew(nh)`	user defined friction history variables	Input/Output
`nh`	number of user defined friction history variables	Input
`crv`	curve array	Input
`nnpcrv`	# of discretization points per crv	Input
`nosl`	number of the sliding interface	Input
`ictype`	contact type	Input
`side`	Info on which side of the contact is being processed	Input
`time`	current solution time	Input
`ncycle`	number of current cycle	Input
`dt2`	time step size at n+1/2	Input
`fric1`	static friction coefficient FS from keyword	Input
`fric2`	dynamic friction coefficient FD from keyword	Input
`fric3`	decay constant DC from keyword	Input
`fric4`	viscous friction coefficient VC from keyword	Input
`lsv`	reference segment number	Input
`idele8`	external user element number of reference segment	Input
`sfac1`	Coulomb friction scale factor *FSF* from keyword	Input
`sfac2`	Viscous friction scale factor *VSF* from keyword	Input
`insv`	SURFA[5] node user id	Input
`fni`	normal force	Input
`areas`	SURFA node area[2]	Input
`stfk`	penalty stiffness	Input
`ix1, ix2, ix3, ix4`	SURFB[6] segment nodes - internal node numbers[3]	Input
`aream`	SURFB segment area	Input
`rn1,rn2,rn3`	x, y, z components of SURFB segment normal	Input
`ue`	Element history data[4]	Input
`ne`	number of element history variables	Input
`ttrs`[7]	temperature of the tracked node	Input, **from R14**
`trfs`[7]	averaged temperature on the reference segment	Input, **from R14**
`flxrfs`[7]	averaged nodal flux vector on reference segment	Input, **from R14**

Notes: (1) Available for coupled analysis, as the average value between SURFA and SURFB. Not yet supported for SOFT = 4 contact. (2) The SURFB node pressure is obtained from fni / areas. (3) To go from internal to external (user) node numbers, use ix1ext=lqfinv8(ix1,1). (4) The extent of the element history data, ue, is determined by the parameter NEHIS on the keyword `*USER_INTERFACE_FRICTION`, see Section 5.1. (5) Previously denoted “slave”. (6) Previously denoted “master”. (7) Additional thermal input, available from R14 of LS-DYNA.

From these arguments, it is possible to define a friction coefficient for non-Mortar contacts dependent on, for example,

- the effective plastic strain, or other history variables, of the involved materials (Note that this functionality is available from rev. 144575 of LS-DYNA),
- also, history variables associated with the contact segments,
- time, temperature, contact pressure and sliding velocity.

Note that in order to include the segment history variables in the `infor` file, the parameters `uhnew` should be used. The parameters `uh` will not be output in the `intfor` file.

Since only one subroutine `usrfric` is defined, accommodating for several friction models can be achieved by letting one of the user-defined parameters, for example `uc(1)`, denote the ID of a friction model.

The subroutine `mortar_userfrc` is called for defining the friction coefficient in Mortar contacts. The subroutine definition for implementing a user defined friction model for Mortar contacts, up until R13 of LS-DYNA is:

```
      subroutine mortar_usrfrc(init,mfrc,nprm,cprm,shst,mhst,icnt,
     1     selm,sprt,styp,stmp,seps,shis,
     2     melm,mprt,mtyp,mtmp,meps,mhis,
     3     cprs,vtan,crv,nnpcrv,dt)
```

For R14 and later versions of LS-DYNA, names of some of the parameters have changed:

```
      subroutine mortar_usrfrc(init,mfrc,nprm,cprm,thst,rhst,icnt,
     1     telm,tprt,ttyp,ttmp,teps,this,
     2     relm,rprt,rtyp,rtmp,reps,rhis,
     3     cprs,vtan,crv,nnpcrv,dt)
```

But the number of parameters is the same.

An overview of the parameters to the subroutine is shown in Table 9. The main objective of the subroutine is to compute a user defined frictional coefficient, that shall be output in the variables `mfrc`.

Table 9. The arguments to the subroutine `mortar_usrfric`

Argument, R13	R14 →	Description	Input/Output
`init`		Initialization phase (.true. or .false.) (currently not active)	Input
`mfrc`		User defined friction coefficient	Output
`nprm`		number of user friction parameters	Input
`cprm`		list of user friction parameters	Input
`shst`	`thst`	SURFA friction history variables	input/output
`mhst`	`rhst`	rference friction history variables	input/output
`icnt`		contact interface id	Input
`selm`	`telm`	element id for SURFA segment	Input
`sprt`	`tprt`	part id for SURFA segment	Input
`styp`	`ttyp`	element type for SURFA segment ('beam ','solid','shell' or 'tshel')	Input
`stmp`	`ttmp`	temperature of SURFA segment (n/a during initialization)	Input
`seps`	`teps`	effective plastic strain on SURFA side	Input
`shis`	`this`	material history variables for the SURFA segment	Input
`melm`	`relm`	element id for SURFB segment (n/a during initialization)	Input
`mprt`	`rprt`	part id for SURFB segment (n/a during initialization)	Input
`mtyp`	`rtyp`	element type for SURFB segment ('beam ','solid','shell' or 'tshel', n/a during initialization)	Input
`mtmp`	`rtmp`	temperature of SURFB segment (n/a during initialization)	Input
`meps`	`reps`	effective plastic strain on SURFB side	Input
`mhis`	`rhis`	material history variables for the SURFB segment	Input
`cprs`		contact interface pressure (n/a during initialization)	Input
`vtan`		tangential relative sliding velocity (n/a during initialization)	Input
`crv`		curve object (to be used in evaluating curve/table, n/a during initialization)	Input
`nnpcrv`		curve parameters (to be used in evaluating curve/table, n/a during initialization)	Input
`dt`		time step	Input

From these arguments, it is possible to define a friction coefficient for Mortar contacts dependent on, for example,

- the effective plastic strain, or other history variables, of the involved materials,
- history variables associated to the contact segments,
- temperature, contact pressure and sliding velocity.

The current solution time is not passed to the `mortar_usrfric` subroutine. The time in contact can be obtained by using a history variable for summing up the time steps `dt`.

5.4 Subroutine examples

In this Section, examples of user defined friction models for mpp[13]/LS-DYNA are presented, for both Mortar and non-Mortar contacts. The related subroutines are found in the Fortran file `dyn21cont.f`. Both Fortran code and keyword input are presented, while the complete examples, see also Section 5.5, can be found as attachments to this document. It shall be stressed that these examples are not intended for use in any kind of production analysis, and there may very well contain errors or flaws.

5.4.1 Time dependent friction coefficient for Mortar contact

For Mortar contact, the implementation of a time dependent friction coefficient is described. Two methods for computing the time in contact will be implemented, either by

- friction model 1: `dt` is summed up and stored in a history variable for the tracked segment, or
- friction model 2: `dt` is summed up and stored in history variables for both tracked and reference segment, and the time is taken as the maximum value of these.

The user will have to select a friction model and give a curve ID specifying the coefficient of friction as a function of time. Also, a default friction coefficient can be given, to be use in case the curve evaluation should result in a negative value. The coefficient of friction will be stored as the second tracked side history variable. The keyword interface will be

```
*USER_INTERFACE_FRICTION
$#     IFID        NOC       NOCI       NHSV      NEHIS       MHSV
contact ID           3          3          2                     1
$#      UC1        UC2        UC3        UC4        UC5        UC6        UC7        UC8
    1 or 2       LCID default μ
```

where blue text indicates that the user should input sensical data, and red text indicates text that should not be changed.

The fist part of the subroutine `mortar_usrfrc`, involving subroutine and variable declarations, follows:

```
      subroutine mortar_usrfrc(init,mfrc,nprm,cprm,shst,mhst,icnt,
     1     selm,sprt,styp,stmp,seps,shis,
     2     melm,mprt,mtyp,mtmp,meps,mhis,
     3     cprs,vtan,crv,nnpcrv,dt)
      implicit none
      include 'nlqparm'
      include 'iounits.inc'
      logical init
      real mfrc,cprm(*),shst(*),mhst(*),cprs,vtan,stmp,mtmp,crv(lq1,2,*)
      real shis(*),mhis(*),seps,meps,dt
      integer selm,sprt,melm,mprt,nprm,icnt,nnpcrv(*)
      character*5 styp,mtyp
C
      real dmdp,dmdv
```

[13] An example for user defined friction in smp/LS-DYNA will be provided in coming versions of this Guideline.

```
      real epsfac,prsfac,tdum, maxeps,cfrc
```

Here, the `implicit none` statement will require all variables to be declared explicitly, reducing the risk for programming errors.

In the original `dyn21cnt.f` Fortran file of the usermat package, the comments after the subroutine declaration provide some documentation regarding the parameters of the user friction subroutine. These comments are omitted here.

The coding for the friction model 1 follows:

```
C --- Law 1: use curve to define friction coefficient, time is time in
C            contact for the tracked segment
c ----       cprm(1) = Law id, cprm(2)=lcid, cprm(3)=default friction
      if (cprm(1).eq.1.) then
         if(shst(1).eq.0)then
           write(iomsg,*) ' --- mortar usrfric law ',cprm(1),' using ',
     1      'lcid',cprm(2),' default friction is ',cprm(3)
         endif
         shst(1)=shst(1)+dt
         call crvval(crv,nnpcrv,cprm(2),shst(1),mfrc,dmdp)
         if(mfrc.ge.0)then
            shst(2)=mfrc
         else
            mfrc=cprm(3)
            shst(2)=mfrc
         endif
      endif
```

It starts by writing a debug output message, in case the tracked side history variable is zero. Then the time in contact is updated and stored in the tracked side history variable `shst(1)`. The subroutine `crvval` is then called to evaluate the curve and obtain the coefficient of friction in the `mfrc` variable. Finally, checking is done and in case a negative value was returned, it is replaced by the default coefficient of friction given as *UC3* from `*USER_INTERFACE_FRICTION` in the variable `cprm(3)`. The applied coefficient of friction is stored in the 2nd history variable for visualization purposes.

The second friction model is similar:

```
C --- Law 2: use curve to define friction coefficient, time is max
C            time in contact for the tracked or reference segment
c ----       cprm(1) = Law id, cprm(2)=lcid, cprm(3)=default friction
      if (cprm(1).eq.2.) then
         if(shst(1).eq.0)then
           write(iomsg,*) ' --- mortar usrfric law ',cprm(1),' using ',
     1      'lcid',cprm(2),' default friction is ',cprm(3)
         endif
         shst(1)=shst(1)+dt
         mhst(1)=mhst(1)+dt
         dmdv=max(shst(1), mhst(1))
         shst(1)=dmdv
         call crvval(crv,nnpcrv,cprm(2),dmdv,mfrc,dmdp)
         if(mfrc.ge.0)then
            shst(2)=mfrc
```

```
          else
             mfrc=cprm(3)
             shst(2)=mfrc
          endif
       endif
```

The difference to the friction model 1 is that also the reference side history variable `mhst(1)` is used for storing the time in contact. Then the maximum value of the time of the tracked and reference side is taken as the time in contact, and this value is then also stored in the 1st reference side history variable `mhst(1)`. Again, in case a negative coefficient of friction should be obtained from the curve, it is replaced by the default value input as *UC3* from `*USER_INTERFACE_FRICTION`.

An example of a simulation using this `mortar_usrfrc` subroutine is provided in Section 5.5.1.

5.4.2 Friction depending on contact pressure and plastic strain

To illustrate the implementation of a user defined friction model for non-Mortar contacts, a model using curves to scale the friction as a function of contact pressure and accumulated effective plastic strain is described in this section. The user interface will involve two curve ID:s, and a max and min value to limit the coefficient of friction. The coefficient of friction will be stored as the 1st history variable, the scale factor related to plastic strain as the 2nd, and the scale factor related to contact pressure as the 3rd. The keyword interface will be

```
*USER_INTERFACE_FRICTION
$#     IFID        NOC       NOCI       NHSV      NEHIS       MHSV
contact ID          11          5          3                     3
$#      UC1        UC2        UC3        UC4        UC5        UC6        UC7        UC8
          1      LCID1      LCID2 min.frict max.frict
```

where blue text indicates that the user should input sensical data, and red text indicates text that should not be changed. In this case, *UC1* is reserved for (future) use as a friction model ID, but since only one model will be implemented, the only sensical input is 1. The *LCID1* should correspond to a curve ID scaling the coefficient of friction by a factor depending on the accumulated effective plastic strain, and *LCID2* should correspond to a curve ID scaling the coefficient of friction as a function of the contact pressure. The lower and upper bounds should be input as *UC4* and *UC5* respectively, in order to keep the applied coefficient within reasonable limits (making it possible to avoid unrealistic extrapolations in curves, which may be caused by for example very high values of plastic strain locally).

The fist part of the subroutine `usrfrc`, involving subroutine and variable declarations, follows:

```
      subroutine usrfrc(fstt,fdyn,uc,nc,prs,temp,v,vx,vy,vz,uh,nh,
     . crv,nnpcrv,nosl,
     . ictype,side,time,ncycle,dt2,fric1,fric2,fric3,fric4,lsv,idele8,
     . sfac1,sfac2,insv,fni,areas,stfk,ix1,ix2,ix3,ix4,aream,
     . rn1,rn2,rn3,ue,ne,uhnew)
c
      implicit none
      include 'nlqparm'
      include 'iounits.inc'
      real fstt,fdyn
      integer nc,nh,nosl,ictype,ncycle,lsv,insv,ix1,ix2,ix3,ix4
      integer ne
```

```
      real uc(nc),uh(nh),uhnew(nh),ue(ne)
      real prs,temp,v,time,dt2,fric1,fric2,fric3,fric4,sfac1,sfac2,
     .     fni,areas,stfk,aream,rn1,rn2,rn3,vx,vy,vz
      real crv(lq1,2,*)
      integer nnpcrv(*)
      character*(*) side
      integer*8 idele8
C
      real cfrc, epsfac, prsfac, tdum
```

Here, the `implicit none` statement will require all variables to be declared explicitly, reducing the risk for programming errors.

In the original `dyn21cnt.f` Fortran file of the usermat package, the comments after the subroutine declaration provide some documentation regarding the parameters of the user friction subroutine. These comments are omitted here.

In the next section of the subroutine, the output variables `fstt` and `fdyn` are initialized to the values input on the `*CONTACT_ …` keyword card, and a debug message is written in the `mes0*` - files.

```
c
c     set coefficients to keyword values
c
      fstt=fric1
      fdyn=fric2
C --- Law 1: use curves to scale fric1 as a function of plastic strain
C     and contact pressure
c ----       uc(1) = Law id, uc(2)=lcid for plascit strain, uc(3)=lcid
C            for pressure, uc(4) = min friction, uc(5) = max friction
      if (uc(1).eq.1.) then
         if(time.le.1E-3)then
           write(iomsg,*) ' --- forming usrfrc law ',uc(1),' using ',
     1      'lcid for plastic strain:',uc(2),
     2      'lcid for contact pressure:',uc(3),
     3      'min friction is ',uc(4),'max friction is ',uc(5)
         endif
```

In the final part of the subroutine, the curves are evaluated to extract the scale factors for plastic strain and contact pressure respectively, and a candidate friction coefficient `crfc` is computed by scaling the input static friction coefficient *FS* of the `*CONTACT_ …` card, which is passed in the variable `fric1`. Finally, this candidate coefficient is checked against the min and max allowed values. The history variables `uhnew` are updated, for post-processing purposes, with the coefficient of friction as the 1st history variable, the scale factor for plastic strain as the 2nd history variable and the scale factor for contact pressure as the 3rd.

```
         call crvval(crv,nnpcrv,uc(2),ue(1),epsfac,tdum)
         call crvval(crv,nnpcrv,uc(3),prs,prsfac,tdum)
         cfrc = fric1*epsfac*prsfac
         fstt = cfrc
         if(cfrc.lt.uc(4))then
            fstt = uc(4)
         else if(cfrc.gt.uc(5))then
```

```
            fstt = uc(5)
         endif
         uhnew(1) = fstt
         uhnew(2) = epsfac
         uhnew(3) = prsfac
      endif
```

An example of a simulation using this `usrfrc` subroutine is provided in Section 5.5.2. This friction model is also implemented for Mortar contact, as friction model 3 (*UC1* = 3) in the `mortar_usrfc` subroutine provided as an attachment to this Guideline.

5.5 LS-DYNA simulation examples

In this section, two LS-DYNA simulation examples of user defined friction are presented.

5.5.1 Mortar contact: a cube on a tilting plane

This is an implicit simulation of a classical set-up for determining the coefficient of friction between two bodies, see Figure 21. A cube (50×50×50 mm) is placed on a plane, and during the first second, gravity is ramped up. Then, the plane is tilted to an angle of 8.3° from t = 2 to t = 7 seconds, see Figure 23. The coefficient of friction is ramped down according to Figure 22, causing the cube to start sliding. The final configurations, using friction model 1 and 2 respectively, are compared in Figure 24. Using friction model 1, the time in contact is defined only on the tracked segment side. This means that once the cube starts sliding as the coefficient of friction is reduced by the time in contact (following the curve of Figure 22), contact will be made with “new” segments with (initially) zero time in contact, causing a high coefficient of friction again, which in turn reduces the sliding velocity and, as the cube falls of the plane, causing it to rotate. Using the friction model 2, this phenomenon is reduced. This is also illustrated by the comparison of the final coefficient of friction (output as contact history variable #2) in Figure 25. Note that the contact history variables are found in the `intfor` file for fringe plotting.

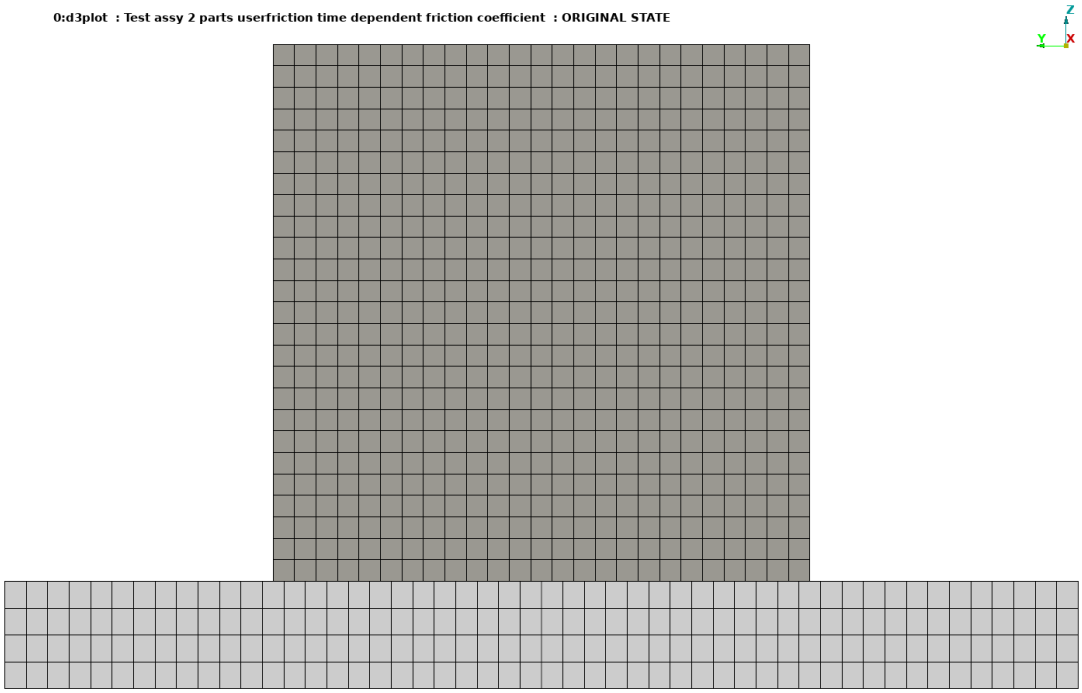


Figure 21. A cube (gray) is resting on a plane (light gray).

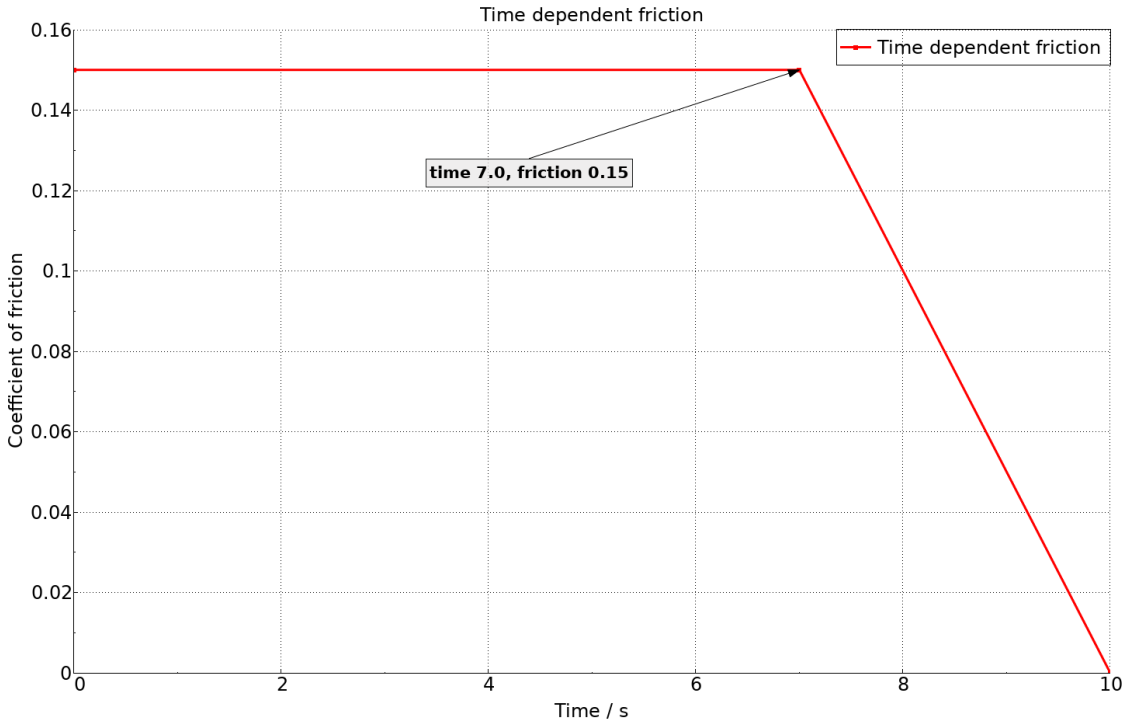


Figure 22. Time-dependent coefficient of friction used in the Mortar contact example.

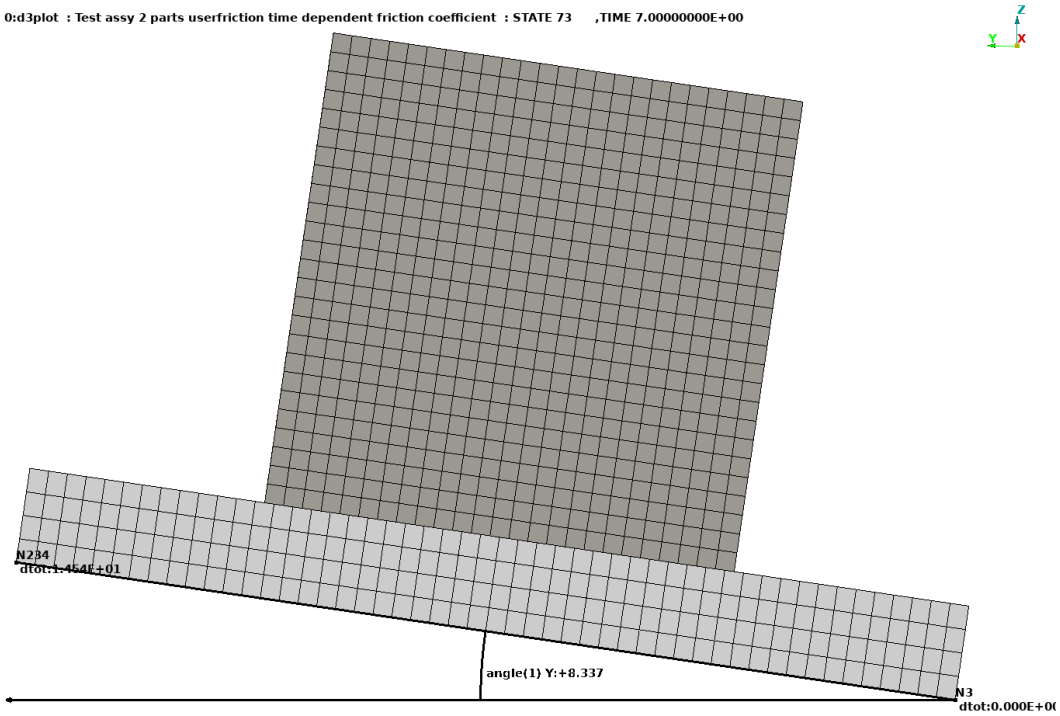


Figure 23. The plane is tilted to an angle of 8.3°, and then the coefficient of friction is decreased.

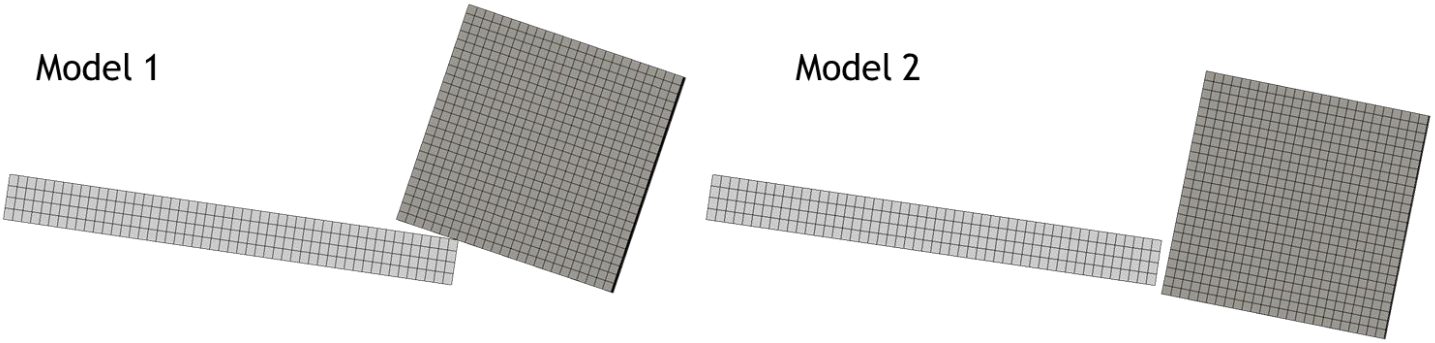


Figure 24. The left image shows the configuration at t = 10 s using friction model 1. The right image shows the configuration at t = 10 s using friction model 2.

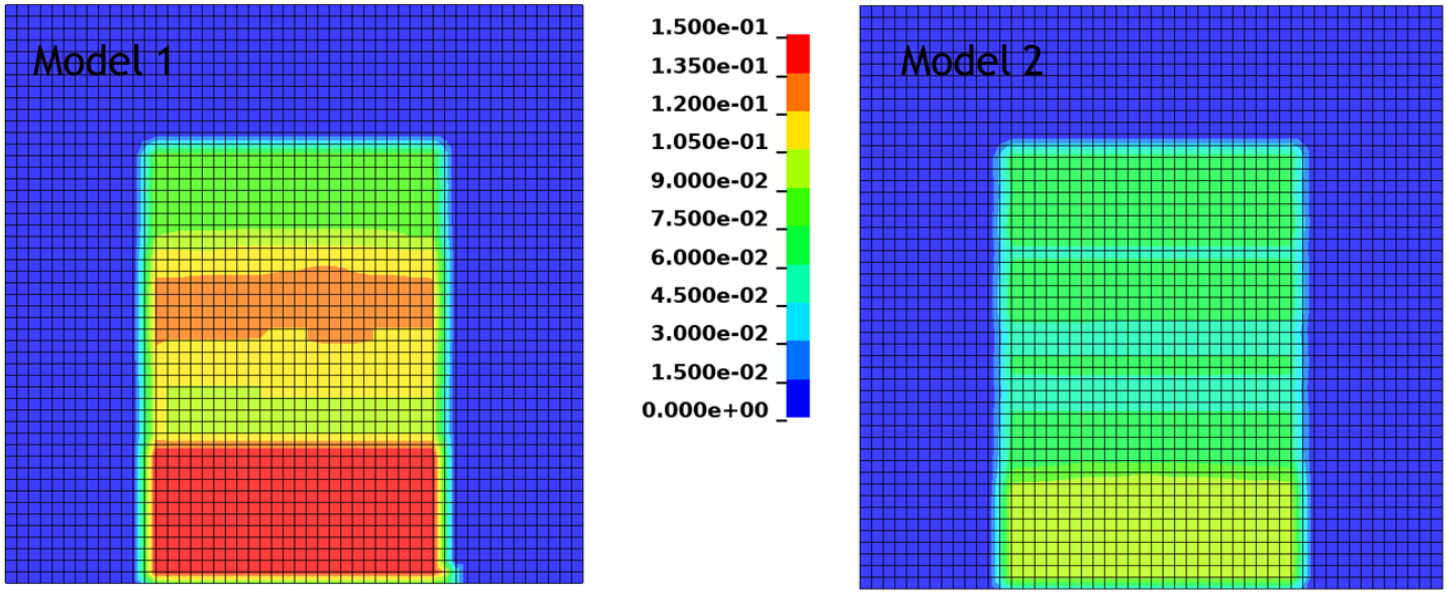


Figure 25. The fringe colors show the final coefficient of friction from 0 (blue) to 0.15 (red) using friction model 1 (left image) compared to the friction model 2 (right image).

5.5.2 Forming analysis using pressure and plastic strain dependent friction

In this example, the forming of a so-called S-rail [25] is studied, see

Figure 26. The S-rail forming simulation case.

. The analysis is performed using the explicit solver in LS-DYNA, with adaptive mesh refinement and three forming contacts (`*CONTACT_FORMING_ONE_WAY_SURFACE_TO_SURFACE_ID`). The assumed curves used for scaling the friction coefficient as a function of plastic strain and contact pressure are shown in Figure 27. In the reference simulation, a constant coefficient of friction (μ = 0.125) was used. The shell thickness results are compared in Figure 28, where some minor differences in shell thickness due to the different friction models can be found. The distribution of the coefficient of friction at an intermediate state is shown in Figure 29.

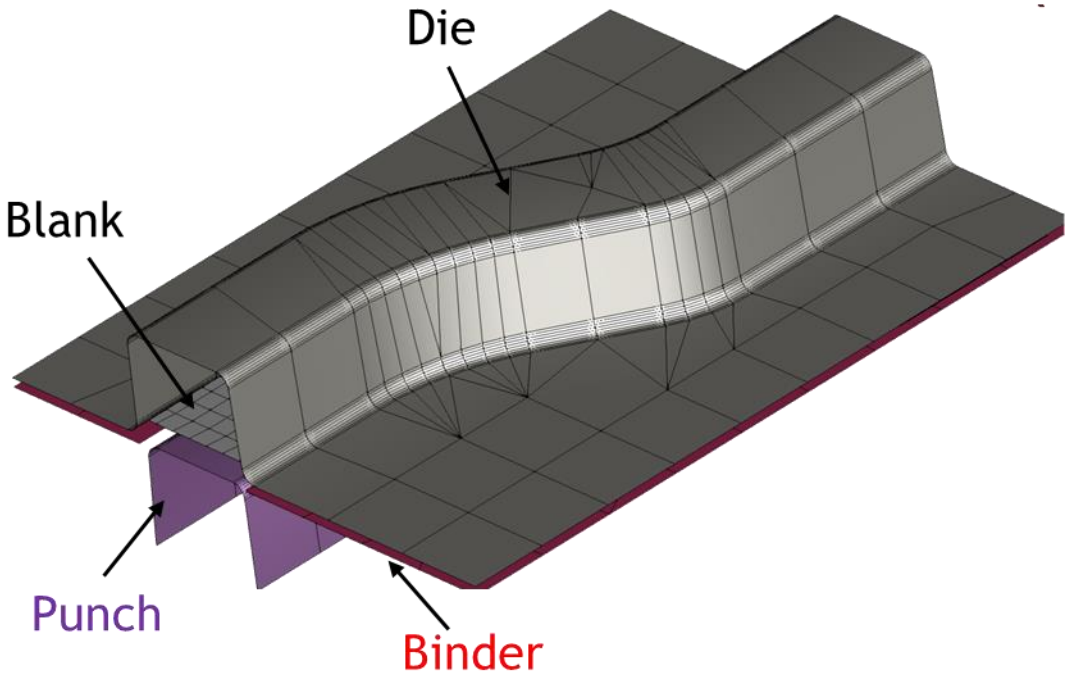


Figure 26. The S-rail forming simulation case.

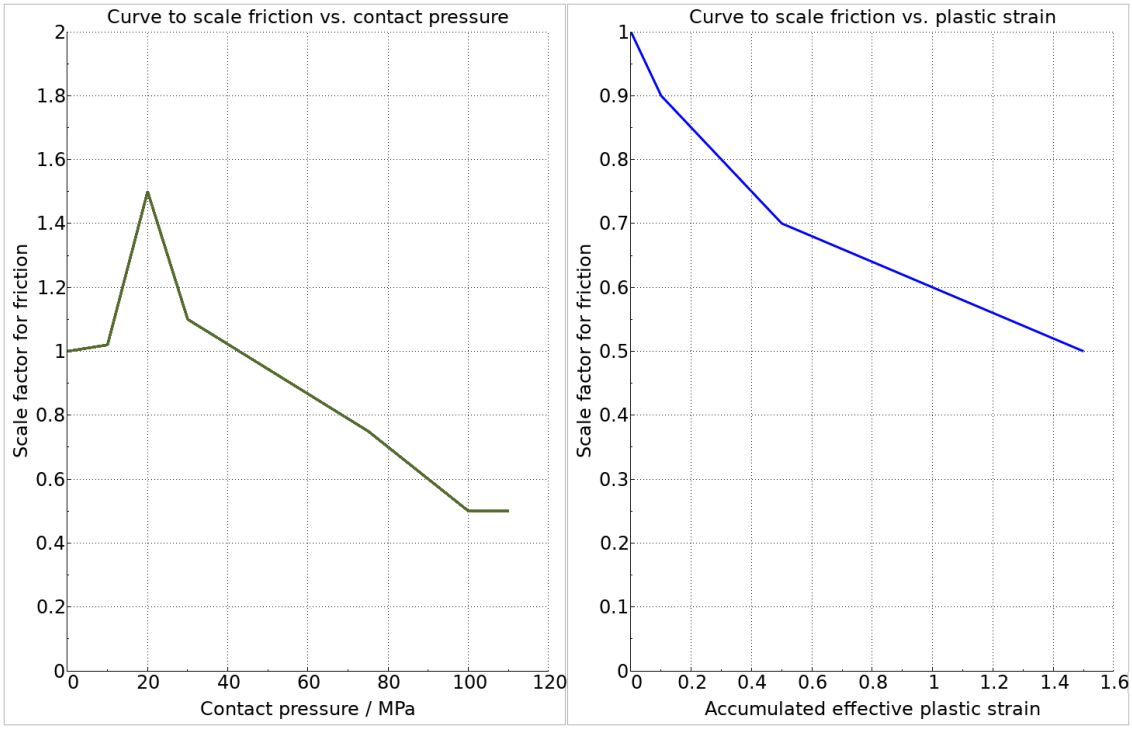


Figure 27. The left image shows the curve for scaling the friction coefficient as a function of contact pressure. The right image shows the curve for scaling the friction as a function of plastic strain.

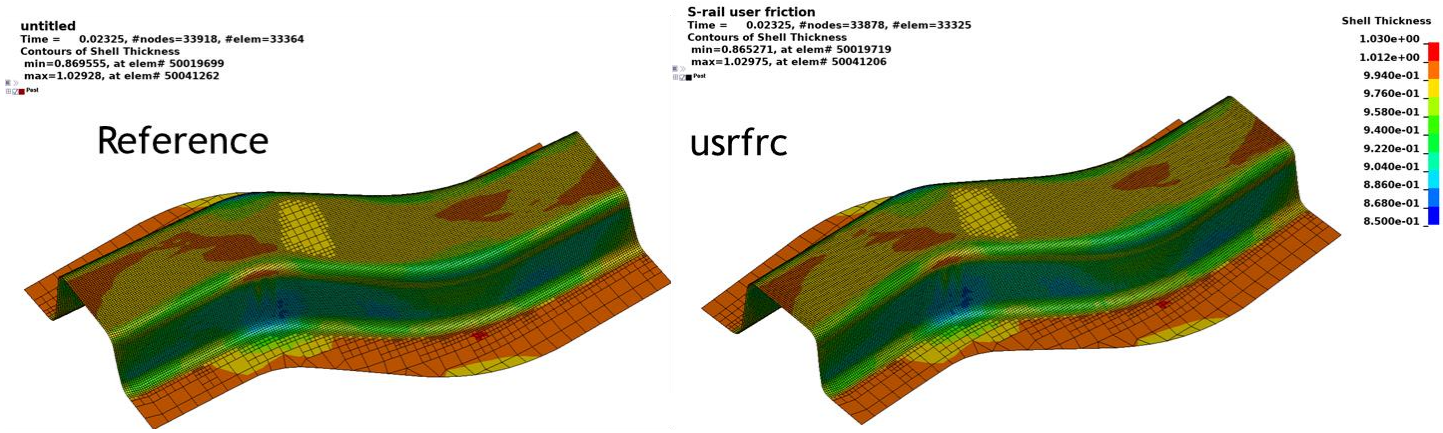


Figure 28. The resulting shell thickness distribution using constant friction (left image) and the user defined friction model.

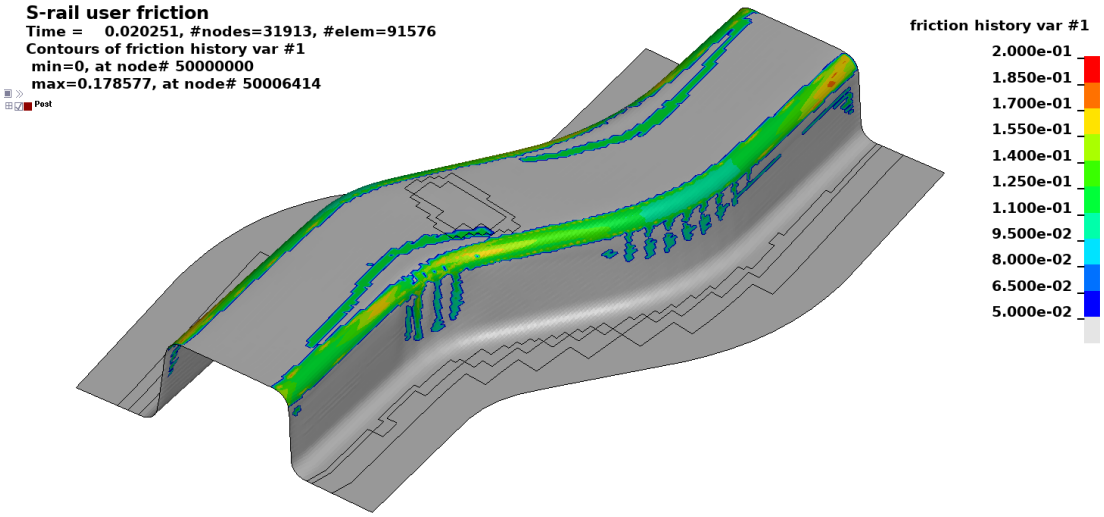


Figure 29. The friction history variable #1 for the contact between the bank and the punch, which is the current static friction coefficient, for the S-rail example.

6 Tied contact using Mortar weld tie

The original purpose of the tied weld options (`*CONTACT_AUTOMATIC_SURFACE_TO_SURFACE_..._TIED_WELD`) is for simulating a welding process: a sliding contact is irreversibly transformed to a (penalty based) tied contact when the temperature exceeds a user-specified value. This behavior is useful for simulating for example two materials that are joined by partial melting of the surfaces by an external heat source (weld torch) and when the melted metal cools down, the parts remain joined.

The user defined tie condition is only available for the Mortar formulation of this contact. It also requires that a coupled thermomechanical analysis (*SOLN* = 2 on `*CONTROL_SOLUTION`) is performed. The user defined tied condition is available from revision 143414 of LS-DYNA.

Another possibility for tying surfaces which are originally separated together is to use the *OPTION* = 1 of `*CONTACT_AUTOMATIC_SURFACE_TO_SURFACE_TIEBREAK`. By this, the surfaces will be permanently tied together as soon as they make contact, independent of the temperature.

6.1 Keyword interface to the user defined weld tie condition

The keyword interface to the user defined tie weld condition is activated by specifying a negative value of *TEMP* on the contact defined via *CONTACT_AUTOMATIC_SURFACE_TO_SURFACE_MORTAR_THERMAL_TIED_WELD_ID. A keyword example follows:

```
*CONTACT_AUTOMATIC_SURFACE_TO_SURFACE_MORTAR_TIED_WELD_THERMAL_ID
        32Base to film
$#1   SURFA     SURFB   SURFATYP SURFBSTYP                            SAPR       SBPR
         3         2          3         3                               1          1
$#2     FS
      0.15
$#3

$#4    TEMP     CLOSE    HCLOSE     NTPRM     NMHIS     NSTWH     NMTWH
       -1.               1.E+3         3                   2
$#    TPRM1     TPRM2     TPRM3     TPRM4     TPRM5     TPRM6     TPRM3     TPRM4
       0.5      102.        5.
$# Thermal properties
     0.000     0.000     1.E+3 1.0000E-3 5.0000E-3       0.5         1
```

The variables related to the user defined tie condition are:

- *TEMP*: Set -1000 < TEMP < 0 to activate a user defined tie condition. The absolute value of TEMP will be passed to the user subroutine mortar_usrtie as the tied weld ID, which then can be used for specifying different user defined conditions.
- *CLOSE*: Segments within this distance are considered for tying. The default is 1 % of the characteristic mesh length scale.
- *HCLOSE*: The thermal contact conductivity when tied.
- *NTPRM*: Number of user defined weld tie parameters (*TPRM1*, *TPRM2* etc.)
- *NMHIS*: Number of material history variables (in addition to plastic strain) to be accessible in the subroutine mortar_usrtie
- *NSTWH*: Number of tracked side tied weld history variables.
- *NMTWH*: Number of reference side tied weld history variables.
- *TPRM1, TPRM2,* etc: User parameters.

Note that the accumulated effective plastic strain for reference and tracked side is accessible even if *NMHIS* = 0.

6.2 Post processing user weld tie condition

The history variables of the user defined weld tied condition subroutine can be post-processed from the intfor file using LS-PrePost 4.8 (or later). To be able to fringe plot the history variables, it is required to set *SPR* = 1 on the *CONTACT_...TIED_WELD_ID – card (setting also *MPR* = 1 is recommended). On the keyword *DATABASE_EXTENT_INTFOR, specify the number of SURFA and SURFB weld tie history variables to be written to the intfor file using the 7th and 8th positions on Card 2, respectively. It is also recommended to activate the output of a tie indicator by setting *NTIED* =1 on Card 2. The keyword *DATABASE_BINARY_INTFOR_FILE is required to specify the filename (intfor is recommended) and output frequency of the contact data. See Figure 32 for examples of fringe plots of user defined tied weld history variables.

6.3 Interface to the user defined tie condition in the subroutine mortar_usrtie

The condition for switching from sliding to tied contact is defined in the subroutine `mortar_usrtie`, which is found in the `dyn21cnt.f` Fortran file. The subroutine definition for implementing a user defined tie condition, up until R13 of LS-DYNA, is:

```
      subroutine mortar_usrtie(tid,init,tie,nprm,cprm,
     .     shst,mhst,icnt,
     1     selm,sprt,styp,stmp,seps,shis,
     2     melm,mprt,mtyp,mtmp,meps,mhis,
     3     cprs,cshr,crv,nnpcrv,time,dt)
```

For R14 and later versions of LS-DYNA, names of some of the parameters have changed:

```
      subroutine mortar_usrtie(tid,init,tie,nprm,cprm,
     .     thst,rhst,icnt,
     1     telm,tprt,ttyp,ttmp,teps,this,
     2     relm,rprt,rtyp,rtmp,reps,rhis,
     3     cprs,cshr,crv,nnpcrv,time,dt)
```

The subroutine is called for each pair of reference-tracked contact segments. Note that the user subroutine is only called for the segments in consideration for contact, which is also related to the *CLOSE* parameter value. For those segments further away than the *CLOSE* value, the contact is not considered as active, and the subroutine is not called. The objective of the subroutine is to indicate if the segments are to be tied together or not, by the logical parameter `tie`. Note that `tie` must be initialized to `.false.` in the subroutine in order to make sure that undesired tying is avoided. An overview of the parameters to the subroutine is shown in Table 10. Curve data as defined by the keyword input is passed via the parameters `crv` and `nnpcrv`, in a similar way as for the user defined material routines, and in order to evaluate curves, the subroutine `crvval` as described in Section 4.4 may be used.

Table 10. Overview of the arguments for the `mortar_usrtie` subroutine

Argument	From R14 →	Description	Input / Output
`tid`		tie interface id	Input
`tie`		set to .true. if tie condition is met, otherwise .false.	Output
`init`		initialization phase (.true. or .false.) (currently not active)	(Input)
`nprm`		number of user tie parameters	Input
`cprm`		list of user tie parameters, use only nprm	Input
`shst`	`Thst`	tie history variables SURFA side	Input/output
`mhst`	`Rhst`	tie history variables SURFB side	Input/output
`icnt`		contact interface id	Input
`selm`	`selm`	element id for SURFA segment	Input
`sprt`	`sprt`	part id for SURFA segment	Input
`styp`	`styp`	element type for SURFA segment ('beam ','solid','shell' or 'tshel')	Input
`stmp`	`stmp`	temperature of SURFA segment (n/a during initialization)	Input
`seps`	`seps`	effective plastic strain on SURFA side	Input
`shis`	`shis`	material history variables for the SURFA segment	Input
`melm`	`melm`	element id for SURFB segment (n/a during initialization)	Input
`mprt`	`mprt`	part id for SURFB segment (n/a during initialization)	Input
`mtyp`	`mtyp`	element type for SURFB segment ('beam ','solid','shell' or 'tshel')	Input
`mtmp`	`mtmp`	temperature of SURFB segment (n/a during initialization)	Input
`meps`	`meps`	effective plastic strain on SURFB side	Input
`mhis`	`mhis`	material history variables for the SURFB segment	Input
`cprs`		contact interface pressure (n/a during initialization)	Input
`cshr`		contact interface shear stress (n/a during initialization)	Input
`crv`		curve object (to be used in evaluating curve/table)	Input
`npcrv`		curve parameters (to be used in evaluating curve/table)	Input
`time`		simulation time	Input
`dt`		time step size	Input

From these arguments, it is possible to define a tie condition depending on, for example,

- the effective plastic strain, or other history variables, of the involved materials,
- also history variables associated with the contact segments on the tracked (SURFA) and reference (SURFB) side,
- time, temperature, and contact pressure.

Once the tied condition is met, and `tie` is set to `.true.`, the segment pair is removed from the checking loop, and will not be passed to the subroutine again. This means that the segment history variables cannot be updated once the tied condition is met.

6.4 Subroutine example

In this section, a basic tie condition depending on the time in contact, contact pressure and temperature will be implemented. It shall be stressed that this example is not intended for use in any kind of production analysis, and it may very well contain errors or flaws.

This may have some similarity with hot glue that cures (or solidifies) below a certain temperature, which in combination with a certain contact pressure being applied for a specified amount of time creates a glued bond. For this, three parameters would be required: a critical minimum contact

pressure *p0*, a critical transition temperature *tthresh*, and the time *ctime* required to create the bond. One history variable for storing the time in contact, during which these requirements are fulfilled, is a minimum, but in addition a second history variable for storing the first time of contact will be created for post-processing and visualization.

The keyword interface will be:

```
*CONTACT_AUTOMATIC_SURFACE_TO_SURFACE_MORTAR_TIED_WELD_THERMAL_ID
CID       Title
$#1  SURFA     SURFB  SURFATYP SURFBSTYP                          SAPR       SBPR
Specify what should be in contact
$#2     FS
Specify friction
$#3

$#4    TEMP     CLOSE    HCLOSE     NTPRM     NMHIS     NSTWH     NMTWH
       -1.          therm.cond         3                   2
$#    TPRM1     TPRM2     TPRM3     TPRM4     TPRM5     TPRM6     TPRM3     TPRM4
   tthresh        p0    ctime
$# Thermal properties
Specify thermal properties
```

where blue text indicates that the user should input sensical data, and red text indicates values that should not be changed.

The first part of the subroutine `mortar_usrtie` involving subroutine and variable declarations, follows:

```
      subroutine mortar_usrtie(tid,init,tie,nprm,cprm,
     .     shst,mhst,icnt,
     1     selm,sprt,styp,stmp,seps,shis,
     2     melm,mprt,mtyp,mtmp,meps,mhis,
     3     cprs,cshr,crv,nnpcrv,time,dt)
      implicit none
      include 'nlqparm'
      include 'iounits.inc'
      real dt
      logical init,tie
      real cprm(*),shst(*),mhst(*),cprs,cshr,stmp,mtmp,crv(lq1,2,*)
      real shis(*),mhis(*),seps,meps,time
      integer tid,selm,sprt,melm,mprt,nprm,icnt,nnpcrv(*)
      character*5 styp,mtyp
```

In the original `dyn21cnt.f` Fortran file of the usermat package, the comments after the subroutine declaration provide some documentation of the user tie subroutine, regarding for example parameters. These comments are omitted here.

The main part of the subroutine starts by initializing the tie indicator to `.false.` and writing a message to the `mes0*` files, to confirm what subroutine is active.

```
      tie = .false.
      if (tid.eq.1) then
         if(time.lt.0.1) then
           write (iomsg,*) "Using Test Mortar usertie Law:1"
           write (iomsg,*) "   tie will be active if temperature < ",
     1                     cprm(1)," and contact pressure is > ",
```

```
     2                        cprm(2)," and time in contact is > ",
     3                        cprm(3)
         endif
```

Then the history variables are updated. The first time in contact, with a contact pressure above the critical value *p0*, is stored in the first history variable `shst(1)`. If the contact pressure is above *p0* and temperature is below *tthresh*, then the time in contact, stored in the 2nd history variable `shst(2)`, is incremented by the current time step size `dt`.

```
          if((stmp.le.cprm(1)).and.(cprs.ge.cprm(2))) then
            shst(2) = shst(2) + dt
          endif
          if((shst(1).lt.time).and.(cprs.lt.cprm(2)))then
             shst(1) = 0.
          endif
          if((shst(1).eq.0.).and.(cprs.ge.cprm(2)))then
             shst(1) = time
          endif
```

Finally, the tie condition is checked: if the contact pressure is above the critical value *p0* and the temperature is below *tthresh* and the time in contact, stored in the 2nd history variable `shst(2)`, exceeds the required *ctime*, the tie flag is set to true, and a message is printed to confirm which element that got tied.

```
c     tie when tracked temperature is below cprm(1) and contact
c     pressure is above cprm(2)  and time in contact is greater than
C     cprm(3)
         if((stmp.le.cprm(1)).and.(cprs.ge.cprm(2))) then
           if(shst(2).gt.cprm(3)) then
             tie=.true.
             write (iomsg,*) '--- Test Mortar usertie 1:elem ',
     1       selm,' now tied at temp ',stmp, ' cpress ',
     2       cprs,' time in contact ',shst(2)
           endif
         endif
       endif
```

This is a rather basic example. Much more involved criteria are possible based on the accessible model quantities, and the history variables of the contact segments can of course have more intricate evolution laws. An example of a simulation using this `mortar_usrtie` subroutine is provided in Section 6.5.

6.5 LS-DYNA simulation example

In this example, a rubber sheet is pressed against an aluminum plate by a steel cube, see Figure 30 for geometries and initial temperatures. The test case is analyzed using the implicit solver of LS-DYNA, as a coupled thermomechanical simulation.

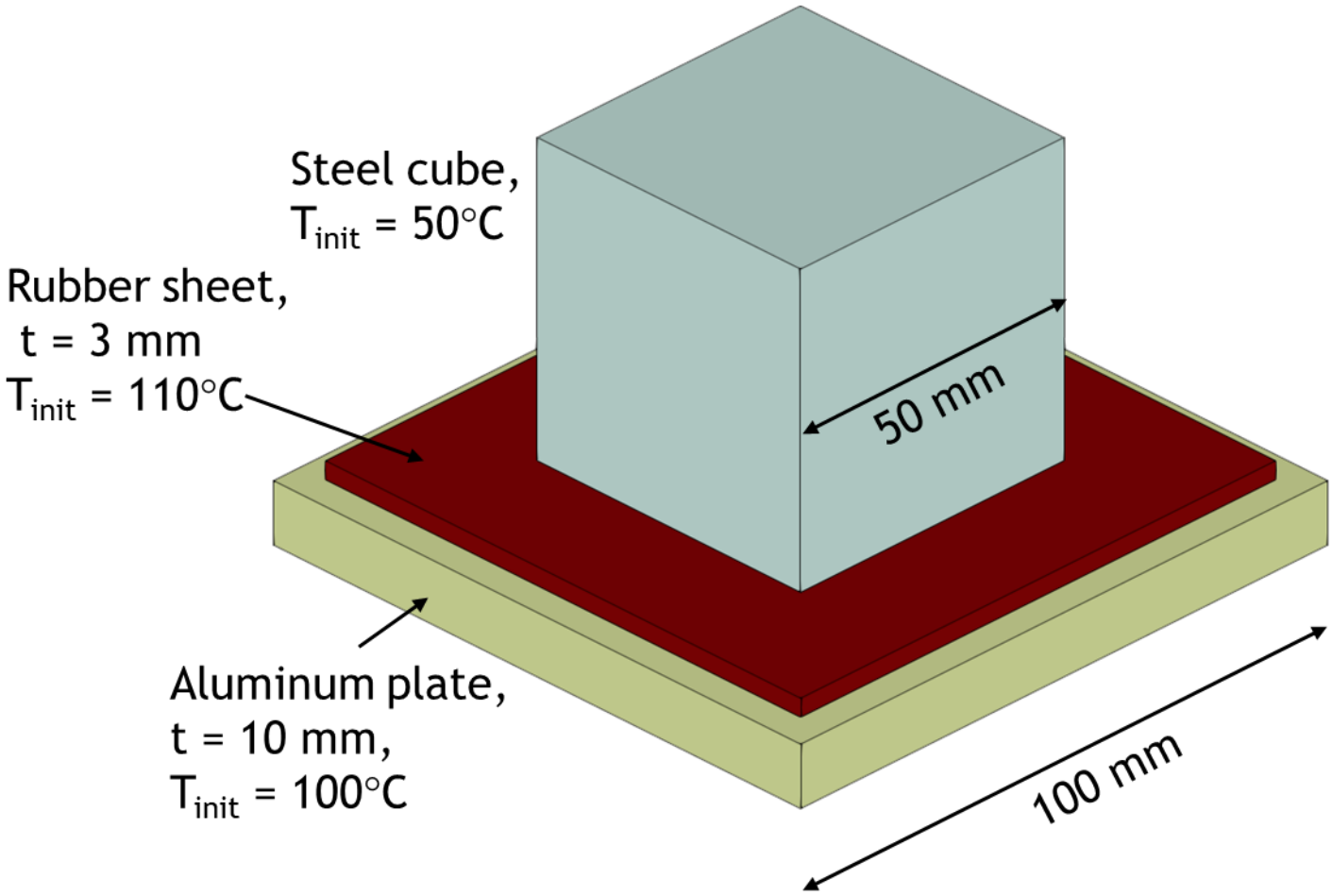


Figure 30. The geometry and initial temperatures for the test case.

A user defined weld tie condition is applied between the rubber sheet and the aluminum plate, using the subroutine as described in Section 6.4, with a critical pressure *p0* = 0.5 MPa, a critical transition temperature *tthresh*, = 102°C and a required time *ctime* = 5 seconds.

Mechanical loading is applied to the set-up in two steps:

1. A distributed loading is applied to the topside of the steel cube: ramped up for 1 second, kept constant for 9 seconds, and then ramped down.
2. A distributed loading is applied to the topside of the rubber sheet, in order to illustrate which segments that got bonded with the aluminum plate.

The loading history is also illustrated in Figure 31. The tie indicator and contact history variable #2 (time in contact) is shown in Figure 32. The final deformed configuration is shown in Figure 33, from which it is concluded that the tied condition between the rubber sheet and the aluminum plate is enforced correctly.

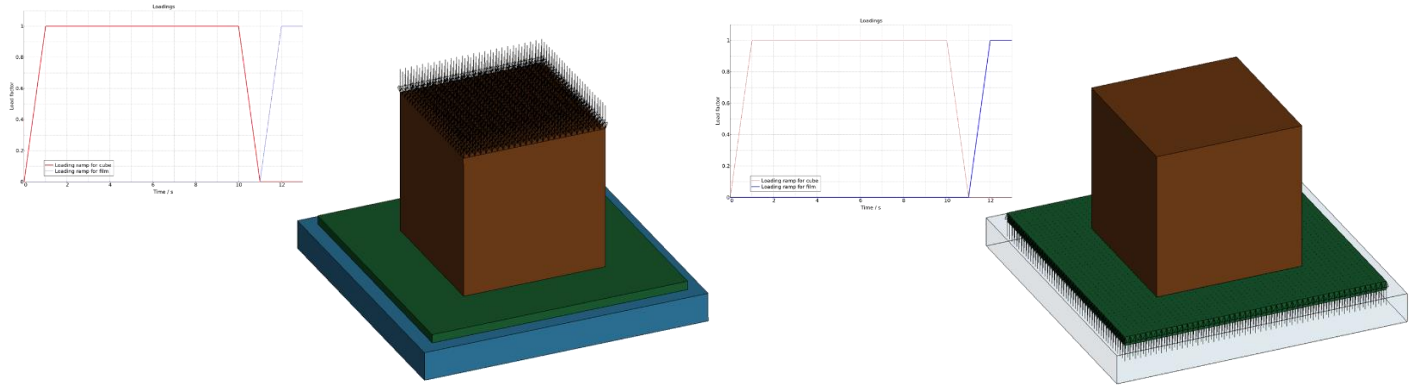


Figure 31. Load case description. First the cube is pushed down (left image) then the sheet is lifted (right image).

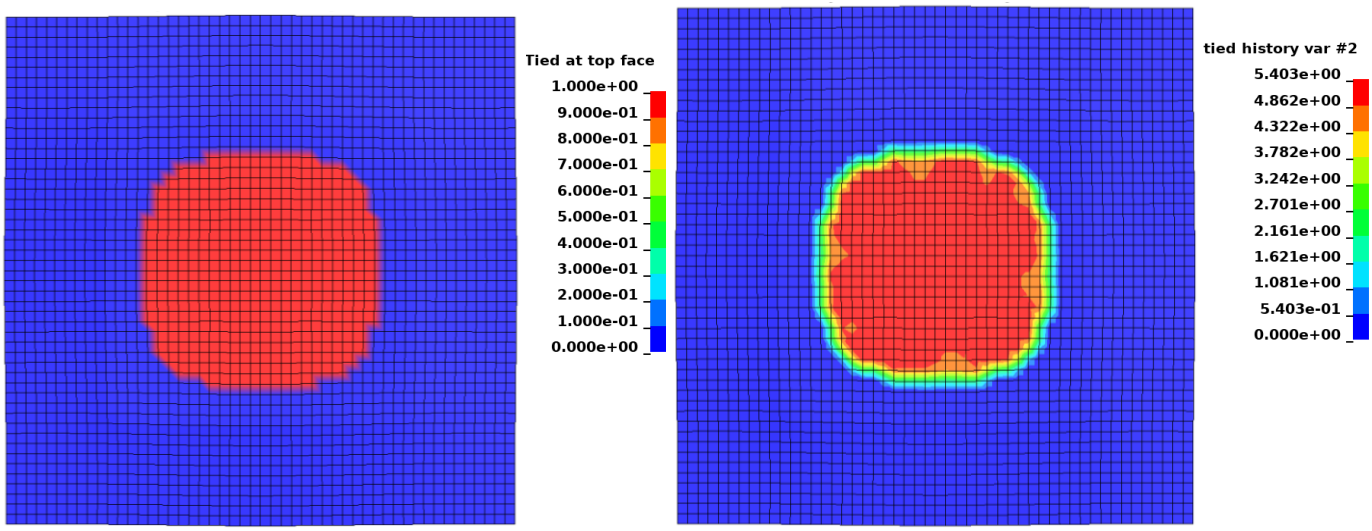


Figure 32. The left image shows a fringe plot of the tied indicator from the `intfor` file. The right image shows the 2nd history variable, time in contact.

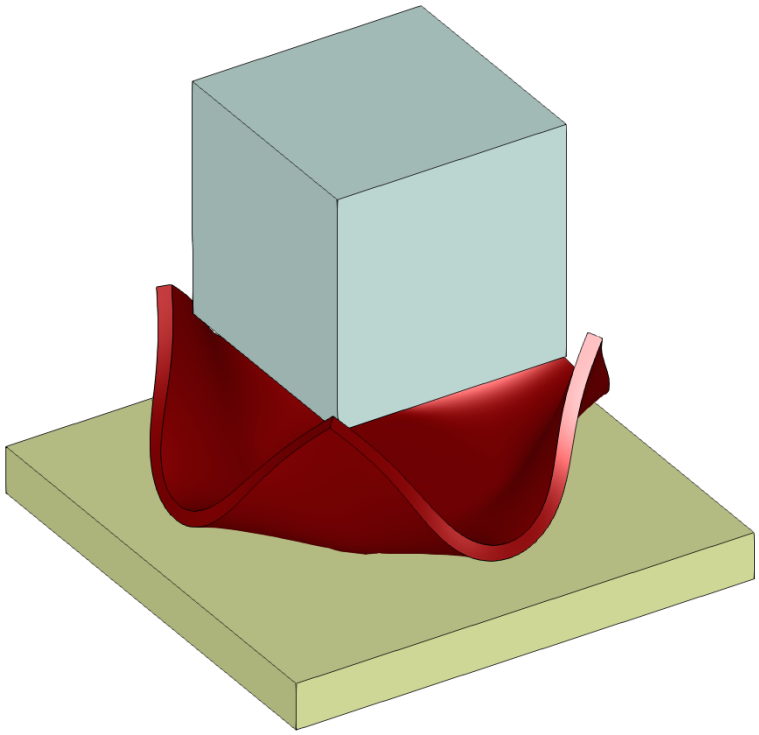


Figure 33. Deformed configuration during load step 2, application of the distributed loading to the rubber sheet.

7 Mortar tiebreak contact

The main purpose of a tiebreak contact is to model surfaces that are initially connected (bonded, glued, welded etc.) but due to high loading or other effects may separate during the analysis. An initially tied contact is irreversibly transformed into a sliding contact. Alternatively, cohesive material models, for example `*MAT_COHESIVE_MIXED_MODE` or `*MAT_COHESIVE_GENERAL` may be applied in an interface layer with cohesive elements between parts that may separate.

In this section only the Mortar formulation of the tiebreak contacts is described. The relevant keyword for the tiebreak functionality is `*CONTACT_AUTOMATIC_SURFACE_TO_SURFACE_TIEBREAK_USER_MORTAR`. The tiebreak contacts have many predefined criteria for damage and failure of the tied interface, see Ref. [1], based on stress, energy release rate or cohesive models, see for example Figure 34.

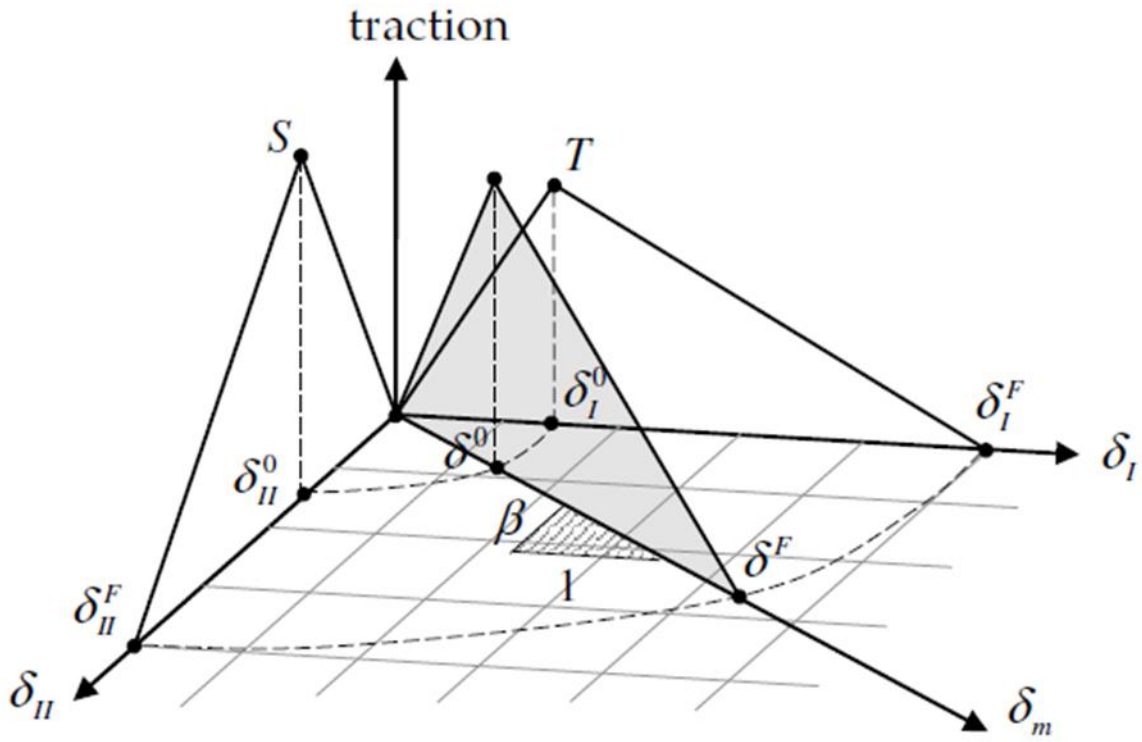


Figure 34. Cohesive mixed-mode law (according to `*MAT_COHESIVE_MIXED_MODE`) for traction-separation, invoked by option = 9 in the tiebreak contacts.

In a sense, the tiebreak contact is the inverse of the Mortar weld tie contact (see Section 6). It is possible to inherit history variables from a joining simulation where a `*CONTACT_AUTOMATIC_SURFACE_TO_SURFACE_MORTAR_THERMAL_TIED_WELD_ID` was used and formulate a separation criterion based on the adhesion process results.

The user defined mortar tiebreak condition is fully supported from revision R13-1674-g3b7bda8165 of LS-DYNA, but with exception of the access to material history variables and plastic strain is available already in R13.0.0.

For the contact segments where the tied contact is released, a “normal” Mortar sliding contact remains, keeping the segments from penetrating each other.

7.1 Keyword interface to the user defined tiebreak condition

The keyword interface to the user defined tiebreak condition is activated by the keyword `*CONTACT_AUTOMATIC_SURFACE_TO_SURFACE_TIEBREAK_USER_MORTAR_ID`. A keyword example follows:

```
*CONTACT_AUTOMATIC_SURFACE_TO_SURFACE_TIEBREAK_USER_MORTAR_ID
        32Base to film
$#1   SURFA     SURFB   SURFATYP SURFBSTYP                                SAPR      SBPR
```

```
         3         2         3         3                              1         1
$#2     FS
      0.15
$#3

$#4 OPTION       NHV     CT2CN        CN    OFFSET     NHMAT     NHWLD
       103         3        1.                              3         2
$#     UP1       UP2       UP3       UP4       UP5       UP6       UP7       UP8
       0.5     102.         5.
$#     UP9      UP10      UP11      UP12      UP13      UP14      UP15      UP16

$#    SOFT    SOFSCL    LCIDAB    MAXPAR     SBOPT     DEPTH     BSORT    FRCFRQ

$#  PENMAX    THKOPT    SHLTHK     SNLOG      ISYM     I2D3D    SLDTHK    SLDSTF
       1.0
```

The variables related to the user defined tiebreak condition are:

- *OPTION*: User tiebreak type. 101 ≤ *OPTION* ≤ 105.
- *NHV*: Number of history variables
- *CT2CN*: Ratio of tangential stiffness to normal stiffness.
- *CN*: Normal stiffness. If left blank, the penalty stiffness divided by the segment area is used (default).
- *OFFSET*: Not applicable to Mortar contact.
- *NHMAT*: Number of material history variables (in addition to plastic strain) to be accessible in the subroutine `mortar_usrtbrk`
- *NHWLD*: Number of tied weld history variables to be read in the user tiebreak routine, assuming they have been carried over from a previous simulation.
- *UP1..UP16*: User parameters.

Note that the accumulated effective plastic strain for reference and tracked side is accessible even if *NHMAT* = 0.

Surfaces that are initially close enough will be tied. The tolerance distance for tying can be set by the *PENMAX* parameter (1.0 on the last row of the keyword example above).

7.2 Post processing user tiebreak condition

The tied status, as an indicator from 1 (meaning perfectly tied) to 0 (meaning completely released) can be output by setting *NTIED* =1 on Card 2 of `*DATABASE_EXTENT_INTFOR`, and visualized in LS-PrePost, see Figure 38 for an example. The keyword `*DATABASE_BINARY_INTFOR_FILE` is required to specify the filename (`intfor` is recommended) and output frequency of the contact data.

7.3 Interface to the user defined tiebreak condition in the subroutine mortar_usrtbrk

The condition for releasing the tied contact is defined in the subroutine `mortar_usrtbrk`, which is found in the `dyn21cnt.f` Fortran file. The subroutine definition for implementing a user defined tiebreak condition is[14]:

```
      subroutine mortar_usrtbrk(tid,dmg,pn,ps,iconv,dpndn,dpnds,dpsdn,
     1     dpsds,istif,prm,hst,whst,selm,melm,sprt,mprt,styp,mtyp,
     2     dn,ds,en,es,ts,tm,seps,meps,shis,mhis,time,dt)
```

For R14 and later versions of LS-DYNA, names of some of the parameters have changed:

```
subroutine mortar_usrtbrk(tid,dmg,pn,ps,iconv,dpndn,dpnds,dpsdn,
     1     dpsds,istif,prm,hst,whst,telm,relm,tprt,rprt,ttyp,rtyp,
     2     dn,ds,en,es,tt,tr,teps,reps,this,rhis,time,dt)
```

The subroutine is called for each pair of reference-tracked contact segments. Note that the user subroutine is only called for the segments in consideration for tied contact, which is also related to the *PENMAX* parameter value. For those segments that are not tied, the subroutine is not called. The objective of the subroutine is to update the damage parameter `dmg`, where 0 indicates no damage (a complete tied contact) and 1 indicates a complete release of the tied contact. If required for implicit calculations (indicated by `istif.ne.0`), also the tangent stiffness matrix should be calculated.

An overview of the parameters to the subroutine is shown in Table 11.

[14] Re-formatted from the original in dyn21cnt.f (where 12 continuation lines are used).

Table 11. Overview of the arguments for the `mortar_usrtbrk` subroutine

Argument	From R14→	Description	Input / Output
tid		tiebreak interface id, integer between 101 and 105	Input
dmg		damage. should be increased between 0 (completely tied) and dmg = 1. indicates complete release and 1 (completely released)	Input/Output
pn		normal traction (unit: pressure)	Output
ps		tangential traction (unit: pressure)	Output
iconv		flag for converged step in implicit (check iconv.eq.0)	Input
dpndn		tangent of normal traction wrt normal separation (unit: pressure / length) needed when istiff.ne.0	Output
dpnds		tangent of normal traction wrt to tangential separation (unit: pressure / length) needed when istif.ne.0 NOTE: dpnds = dpsdn is required	Output
dpsdn		tangent of tangential traction wrt to normal separation (unit: pressure / length) needed when istif.ne.0 NOTE: dpsdn = dpnds is required	Output
dpsds		tangent of tangential traction wrt to tangential separation (unit: pressure / length) needed when istif.ne.0	Output
istif		flag if stiffness is needed for implicit, check istif.ne.0	Input
prm		input parameters	Input
hst		history variables	Input/Output
whst		Weld tie history variables from previous tied weld analysis	Input
selm	telm	element id for SURFA segment	Input
melm	relm	element id for SURFB segment	Input
sprt	tprt	part id for SURFA segment	Input
mprt	rprt	part id for SURFB segment	Input
styp	ttyp	Element type for SURFA segment ('beam ','solid','shell' or 'tshel')	Input
mtyp	rtyp	Element type for SURFB segment ('beam ','solid','shell' or 'tshel')	Input
dn		Normal separation (unit: length, positive means tensile)	Input
ds		Tangential separation (unit: lengt, always positive)	Input
en		Normal stiffness (unit: pressure / length)	Input
es		Tangential stiffness (unit: pressure / length)	Input
ts	tt	Temperature of SURFA segment	Input
tm	tr	Temperature of SURFB segment	Input
seps	teps	Effective plastic strain on SURFA side	Input
meps	reps	Effective plastic strain on SURFB side	Input
shis	this	material history variables for the SURFA segment	Input
mhis	rhis	material history variables for the SURFB segment	Input
time		simulation time	Input
dt		time step size	Input

From these arguments, it is possible to define a tiebreak condition depending on, for example,

- the effective plastic strain, or other history variables, of the involved materials,
- also, history variables associated with the contact segments on tracked and reference side, which can be inherited from a previous analysis using MORTAR_TIED_WELD.
- time, temperature, and contact pressure.

Note that it is required that the tangent stiffness matrix (stored in variables dpndn, dpnds, dpsdn, dpsds)

$$\boldsymbol{D} = \begin{pmatrix} \frac{\partial p_n}{\partial d_n} & \frac{\partial p_n}{\partial d_s} \\ \frac{\partial p_s}{\partial d_n} & \frac{\partial p_s}{\partial d_s} \end{pmatrix}$$

be symmetrical, $\boldsymbol{D} = \boldsymbol{D}^{\mathrm{T}}$, which in turn implies $\frac{\partial p_n}{\partial d_s} = \frac{\partial p_s}{\partial d_n}$. If a model with an unsymmetrical stiffness matrix is used, it must be symmetrized internally in the `mortar_usrtbrk` routine.

In the `intfor` file, 1 – `dmg` (see Table 11) is output as the “tied at top face” / “tied at bottom face” tie indicator, which can be visualized in LS-PrePost, see Figure 38.

7.4 Subroutine example

In this section, a basic tiebreak condition depending on a history variable of the materials in contacts will be implemented. This example requires revision R13-1674-g3b7bda8165 or later of LS-DYNA. It shall be stressed that this example is not intended for use in any kind of production analysis, and it may very well contain errors or flaws.

The damage will be mapped linearly from 0 to 1 when the material history goes from a lower threshold value *thresh* to an upper limit *maxlim*. For this, three parameters would be required: in addition to the limits also an identifier for the (tracked side) history variable to scale the damage.

The keyword interface will be:

```
*CONTACT_AUTOMATIC_SURFACE_TO_SURFACE_TIEBREAK_USER_MORTAR_ID
CID       Title
$#1  SURFA     SURFB  SURFATYP SURFBSTYP                              SAPR       SBPR
Specify what should be in contact
$#2     FS
Specify friction
$#3

$#  OPTION       NHV      CT2CN        CN    OFFSET     NHMAT     NHWLD
       103         1                                 hisvarid
$#     UP1       UP2       UP3       UP4       UP5       UP6       UP7       UP8
thresh    maxlim    hisvarid
$#     UP9      UP10      UP11      UP12      UP13      UP14      UP15      UP16

$#    SOFT    SOFSCL    LCIDAB    MAXPAR     SBOPT     DEPTH     BSORT    FRCFRQ

$#  PENMAX    THKOPT    SHLTHK     SNLOG      ISYM     I2D3D    SLDTHK    SLDSTF
Optional distance for tying
```

where blue text indicates that the user should input sensical data, and red text indicates values that should not be changed. By setting *hisvarid* = 0, the maximum plastic strain from either tracked or reference side material will scale the damage.

The first part of the subroutine `mortar_usrtbrk` involving subroutine and variable declarations, follows:

```
      subroutine mortar_usrtbrk(tid,dmg,pn,ps,iconv,dpndn,dpnds,dpsdn,
```

```
     1     dpsds,istif,prm,hst,whst,selm,melm,sprt,mprt,styp,mtyp,
     2     dn,ds,en,es,ts,tm,seps,meps,shis,mhis,time,dt)
      implicit none
      include 'nlqparm'
      include 'iounits.inc'
      integer tid
      real dmg
      real pn,ps
      integer iconv
      real dpndn,dpnds,dpsdn,dpsds
      integer istif
      real prm(*),hst(*),whst(*)
      integer selm,melm
      integer sprt,mprt
      character*5 styp,mtyp
      real dn,ds
      real en,es
      real ts,tm
      real seps,meps
      real shis(*),mhis(*)
      real time,dt
c
      real pd,prmdiff,seval
      integer ihvar
```

In the original `dyn21cnt.f` Fortran file of the usermat package, the comments after the subroutine declaration provide some documentation of the user tiebreak subroutine, regarding for example parameters. These comments are omitted here. The variable *pd* will be used to store the previous damage value (*dmg*) when the subroutine is called. The variable *prmdiff* will hold the difference between *maxlim* and *thresh*, and *seval* is the quantity to compare with (plastic strain or history variable value). The variable *ihvar* simply is the number of the history variable.

The main part of the subroutine starts by writing a message to the `mes0*` files, to confirm what subroutine is active.

```
      if (tid.eq.103) then
C --- add some initial diagnose print out
        if(time.le.1.E-3)then
          write(iomsg, *) ' ---  mortar tiebreak 103'
          if(int(prm(3)).gt.0)then
            write(iomsg, *) '   using material history variable ',
     1                       int(prm(3))
          else
            write(iomsg, *) '   using plastic strain'
          endif
          write(iomsg, *) '   damage scaled from ',prm(1),' to ',prm(2)
          write(iomsg, *) ' ---  '
        endif
```

Then the linear, undamaged surface tractions are computed, and the reference quantity *seval* is evaluated:

```
        pn=en*dn
        ps=es*ds
```

```
      pd = dmg
      ihvar=int(prm(3))
      if(ihvar.eq.0)then
        seval=max(seps, meps)
      else
        seval=shis(ihvar)
      endif
```

Then follows the damage calculation, where it is ensured that the damage cannot decrease, nor exceed unity.

```
      prmdiff = max(1.E-5,abs(prm(2) - prm(1)))
      if(seval.gt.prm(2)) then
         dmg = 1.0
      elseif(seval.gt.prm(1)) then
         dmg = (seval-prm(1))/prmdiff
      endif
      dmg=max(pd, dmg)
      dmg=min(dmg,1.)
```

The damage value is stored in the 1st history variable, and the damaged surface tractions are computed.

```
      if(dmg.gt.0.)then
        hst(1)=dmg
        pn=(1.-dmg)*pn
        ps=(1.-dmg)*ps
      endif
```

A message is output if the tied contact is fully released.

```
      if (dmg.eq.1.and.pd.lt.1..and.iconv.eq.0) then
         write (iomsg,1) styp,selm,sprt,mtyp,melm,mprt,time
      endif
```

Finally, the tangential stiffness is computed. Since there are no couplings between the normal and tangential components, we get $\frac{\partial p_n}{\partial d_s} = \frac{\partial p_s}{\partial d_n} = 0$.

```
        if (istif.ne.0) then
c     compute normal stiffness, accounting for damage
            dpndn=en*(1.-dmg)
c     compute tanegntial stiffness, accounting for damage
            dpsds=es*(1.-dmg)
c     no couplings
            dpnds=0.
            dpsdn=0.
        endif
```

which concludes this subroutine example. A simulation example using this subroutine is presented in Section 7.5.

7.5 LS-DYNA simulation example

In this example, a cantilever beam (similar to the example of Section 4.6.1, 4.6.2) with a weld-on reinforcement plate, purple in Figure 35, is subjected to severe overloading. In the first stage, a

prescribed vertical displacement of -140 mm and then +140 mm is applied at the bolt holes of the end bracket. In the second stage, longitudinal compression of 500 mm is applied to the deformed beam.

Elastic-plastic material properties typical for aluminum (E = 70 GPa, ν = 0.31, σ_Y = 140 MPa) are used. Contact is considered between the square beam and the cylindrical rigid support, using `*CONTACT_AUTOMATIC_SURFACE_TO_SURFACE_MORTAR`, as well as self-contact within the C-section beam (red in Figure 35). The side-plate reinforcement is attached to the C-section using a tiebreak contact (`*CONTACT_AUTOMATIC_SURFACE_TO_SURFACE_TIEBREAK_USER_MORTAR`) with user-defined release condition based on plastic strain in the materials involved, according to the subroutine of Section 7.4. Deformation results from implicit analysis are shown in Figure 36 - Figure 37. The tied indicator is shown in a fringe plot in Figure 38 and as a history plot in Figure 39. Some of the nodes release already during the initial vertical loading (t < 1) while most damage to the tied contact occur during the final axial compression (t > 4).

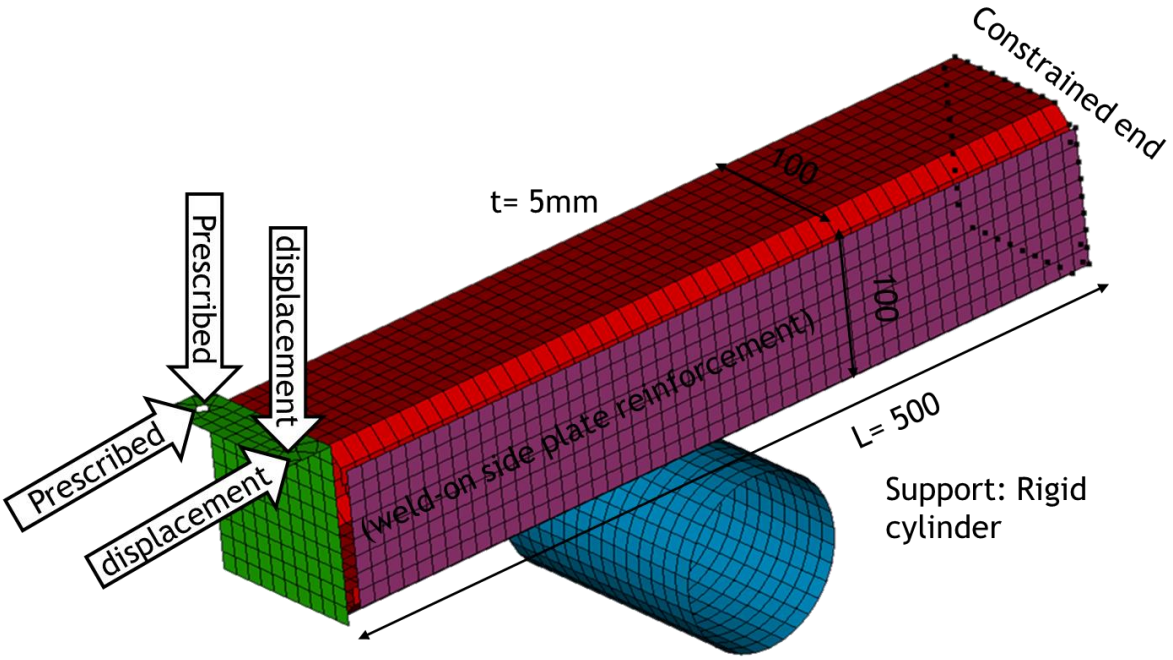


Figure 35. A C-profile (100 × 100 mm, t = 5 mm) cantilever beam with a weld-on reinforcement is subjected to prescribed displacement at the end bracket (green in the image) and contact with a rigid cylindrical support.

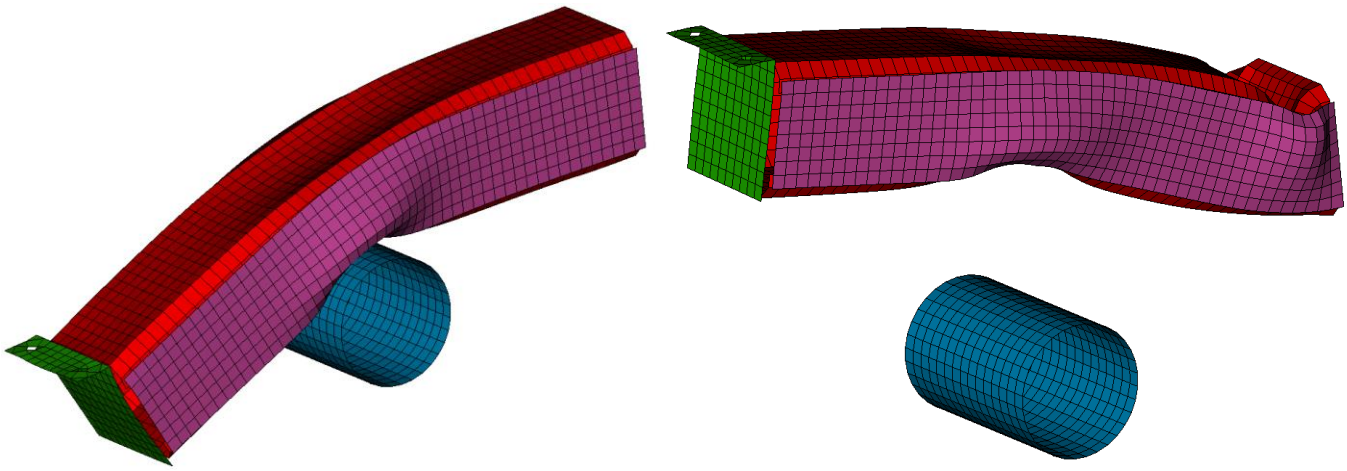


Figure 36. The left image shows the initial vertical displacement of -140 mm (t=1), and the right image shows the deformed configuration after the vertical displacement of +140 mm (t=3).

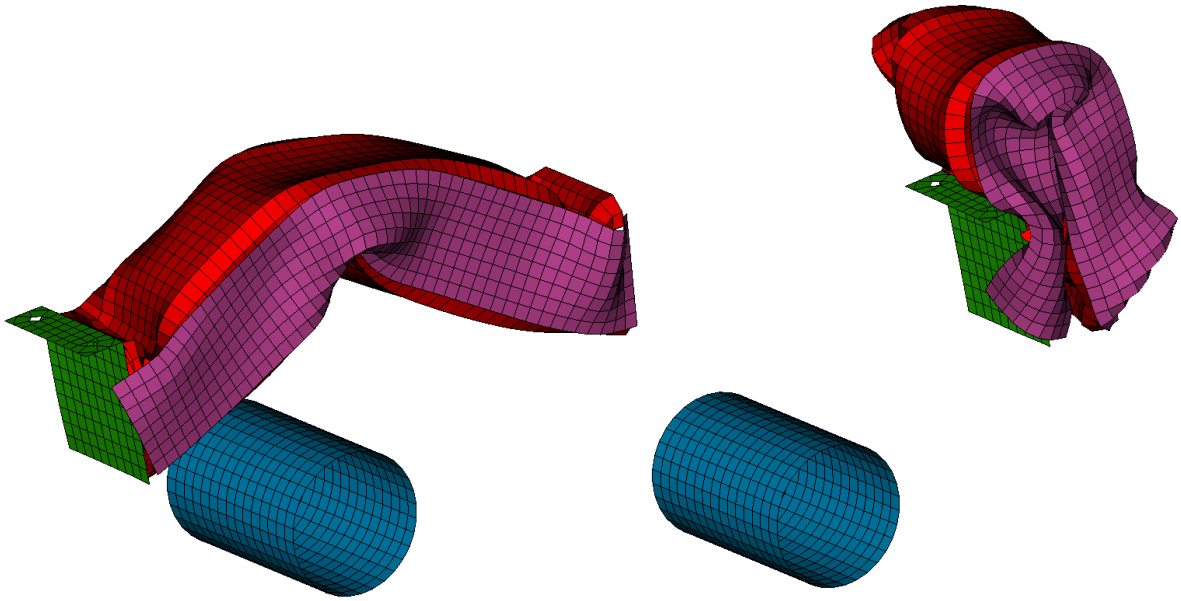


Figure 37. The left image shows an initial phase of the axial compression. The right image shows the final configuration.

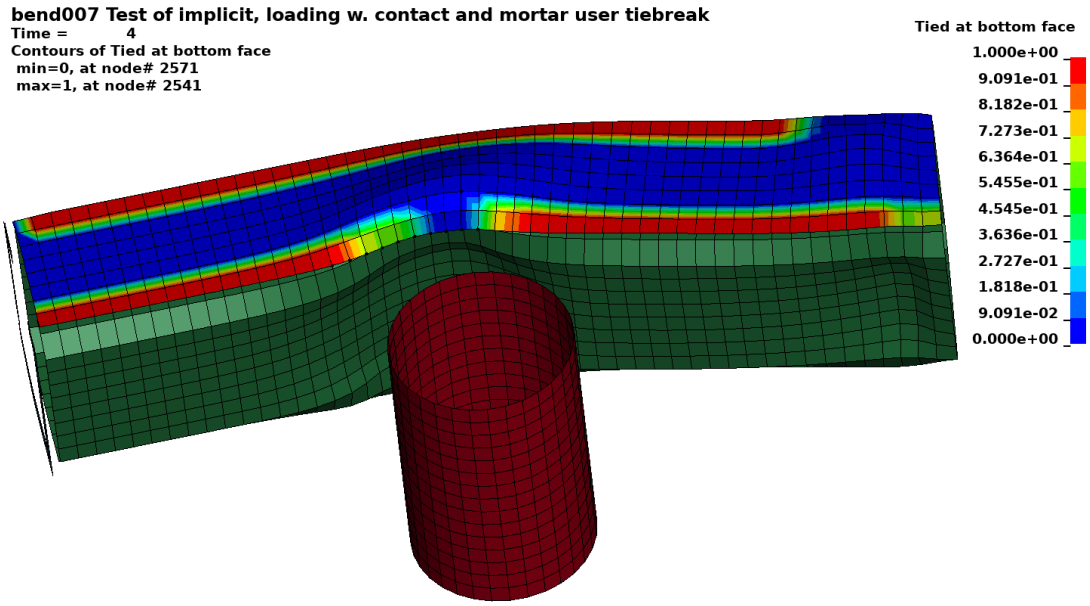


Figure 38. The fringe plot shows the tied indicator at the beginning of the axial compression stage from 0 (blue) to 1 (red).

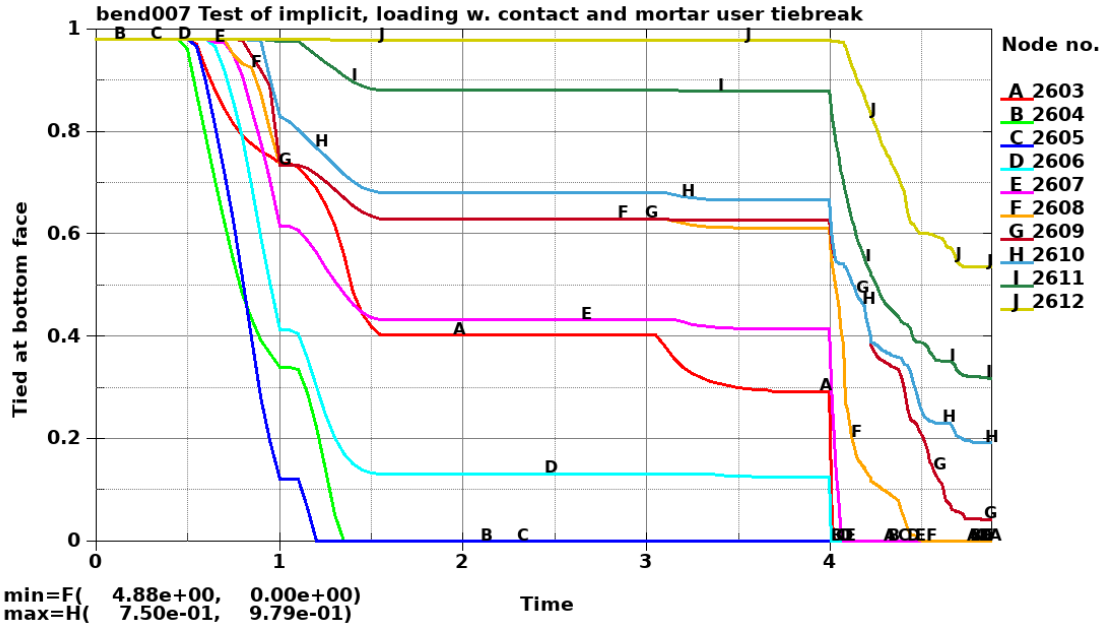


Figure 39. The evolution of the tied indicator in some of the nodes closes to the cylindrical support (marked by black dots in Figure 38).

8 Loads interface

There are many ways of defining customized loading using the built-in LS-DYNA keywords. For example, the `*LOAD_{OPTION}` keywords for applying loads to nodes or segments also accept definition of the loading scale factor not only via curves (`*DEFINE_CURVE`) but also via functions of time, initial coordinate and current coordinate using `*DEFINE_FUNCTION`, via a C-like programming language (see Example 2 under `*DEFINE_FUNCTION` in Ref. [1], where the definition of a hydrostatic pressure is

demonstrated). By the *LOAD_SEGMENT_{OPTION}_NONUNIFORM keyword, pressure loading acting at a specified direction to the surface (not necessarily the normal direction) can be applied.

In addition to the built-in keywords for loading definitions, it is possible to provide user defined loadings by the keywords *USER_LOADING and *USER_LOADING_SET and the corresponding Fortran subroutines loadud and loadsetud of the file dyn21.f in the usermat package. These subroutines also give access to nodal accelerations, velocities, and masses etc. These user defined loadings are supported by both the implicit and explicit mechanical solver of LS-DYNA. Under certain conditions, the user defined loading subroutines can also involve element deletion.

An overview (including both keyword and Fortran code examples) of the user defined loading options has previously been presented in Ref. [19]. A very brief description of the loadsetud subroutine can be found in Ref. [1] under the *USER_LOADING_SET keyword.

8.1 Keyword interface to the user defined loadings

The keyword interface to the user defined loadings is given by

- *USER_LOADING, mainly for applying nodal loads and
- *USER_LOADING_SET, for applying more general loadings, including temperatures.

The keyword input for the *USER_LOADING option is quite straight-forward:

```
*USER_LOADING
$     PARM1      PARM2      PARM3      PARM4      PARM5      PARM6      PARM7
         1.      2041.         2.       100.         5.
```

The variables of this keyword (*PARM1*, *PARM2* etc.) are simply parameter values to be read by the user subroutine loadud. The documentation of the subroutine should preferably describe what each parameter corresponds to. This is discussed further in Sections 8.3 and 8.4.

An example of keyword input for the *USER_LOADING_SET option follows:

```
*USER_LOADING_SET
$       SID      LTYPE       LCID        CID        SF1        SF2        SF3      IDULS
          2     PRESSS        100          0        0.0        0.0        0.0          1
```

where the variables are:

- SID: The ID of the set that the loading should be applied to. The set type is determined by the loading type (LTYPE).
- LTYPE: Loading type, for example PRESSS for pressure on segments.
- LCID: Curve ID for scaling the loading.
- CID: Coordinate system ID. Default is the global coordinate system.
- SF1, SF2, SF3: Scale factors with different meanings depending on the loading type.
- IDULS: An ID number that is passed to the subroutine loadsetud.

These keywords can also be combined. By this, additional parameters for *USER_LOADING_SET can be specified by the *USER_LOADING keyword input (see Section 8.5.3 for an example of this).

8.2 Post processing user defined loadings

Currently, there are no options to output data or results from the user defined loading subroutines, other than writing text in the mes0* and d3hsp – files. While for example forces applied using *LOAD_NODE_{*OPTION*} are output for post-processing in the bndout – file, the forces from the user defined loadings are not.

8.3 Interfaces to the user-defined loading subroutines

There are two different subroutines for defining the loading:

- loadud, corresponding to the *USER_LOADING keyword, for applying nodal loadings by direct modification of the global load vector. This is a *scalar* subroutine.
- loadsetud, corresponding to the *USER_LOADING_SET keyword, for providing a user defined load scale factor. This is a *vectorized* subroutine.

They are both found in the dyn21.f file. The parameter list for the user defined loading subroutine is:

```
      subroutine loadud(fnod,dt1,time,ires,x,d,v,a,ixs,
     . numels,ixb,numelb,idrflg,tfail,isf,p,npc,fval,iob,iadd64,numelh,
     . ixh,nhex_del,nbeam_del,nshell_del,hexarray,hextim,bemarray,
     . bemtim,shlarray,shltim,parm,numnp,fnodr,dr,vr,ndof,xmst,xmsr)
```

From R13 of LS-DYNA, some additional parameters related to thick shell elements were added:

```
      subroutine loadud(fnod,dt1,time,ires,x,d,v,a,ixs,
     . numels,ixb,numelb,idrflg,tfail,isf,p,npc,fval,iob,iadd64,numelh,
     . ixh,nhex_del,nbeam_del,nshell_del,hexarray,hextim,bemarray,
     . bemtim,shlarray,shltim,parm,numnp,fnodr,dr,vr,ndof,xmst,xmsr,
     . numelt,ixt,ntsh_del,tsharray,tshtim)
```

See also the attached Fortran file example for details. The objective of the loadud subroutine is to modify the force array fnod and/or the moment array fnodr, based on the input parameters of the array parm and other model data. An overview and brief description of the parameters to the subroutine is shown in

Table 12.

If the ires parameter has a negative value, it means that |ires| input parameters should be read in and stored in the parm array. NOTE! This must be done by explicit coding inside the loadud subroutine. A template for this is provided in the example code of the usermat package. See further Section 8.4.1 for an example.

In addition to allowing access to (almost) all nodal data of the current model (coordinates, velocities, accelerations, masses, etc.) the loadud subroutine may also trigger element deletion. This requires that the keyword *DEFINE_ELEMENT_DEATH is present in the main keyword deck, for one or more elements

of the type to be deleted. The deletion time (variable *TIME* of the *DEFINE_ELEMENT_DEATH keyword) can be set to a value that greatly exceeds the termination time for the run.

Table 12. The arguments to the subroutine loadud

Argument	Description	Input / Output
fnod	Global nodal forces	Input / Output
dt1	Current time step size	Input
time	Current problem time	Input
ires	Restart flag(1)	Input / Output
x	Original nodal coordinates	Input / Output
d	Nodal displacements	Input
v	Nodal velocities	Input
a	Nodal accelerations	Input
ixs	Shell element connectivities	Input
numels	Number of shell elements	Input
ixb	Beam element connectivities(2)	Input
numelb	Number of beam elements	Input
idrflg	Nonzero if dynamic relaxation phase	Input
tfail	Shell element failure time	Input / Output
isf	Shell element failure flag (=1 → On)	Input
p	Load curve data pairs (abscissa, ordinate)	Input
npc	Pointer into p	Input
fval	fval(lc) is the value of load curve ID lc at the current time	Input
iob	i/o buffer	
iadd64		
numelh	Number of solid elements	Input
ixh	Solid element connectivities	Input
nhex_del	if >0, element deletion option is active for solids	Input
nbeam_del	if >0, element deletion option is active for beam	Input
nshell_del	if >0, element deletion option is active for shells	Input
hexarray	Time to delete solid elements, the value should be > time	Output
hextim	Solid element deletion is checked when time is ≥ hextim	Input
beamarray	Time to delete beam elements, the value should be > time	Output
bemtim	Beam element deletion is checked when time is ≥ bemtim	Input
shlarray	Time to delete shell elements, the value should be > time	Output
shltim	Shell element deletion is checked when time is ≥ shltim	Input
parm	Array for storing input parameters	Input/Output(3)
numnp	Number of nodal points	Input
fnodr	Global nodal moments	Input / Output
dr	Nodal rotational displacements	Input
vr	Nodal rotational velocities	Input
ndof	Number of degrees of freedom per node in the solution phase (= 0 in the initialization phase)	Input
xmst	Reciprocal of nodal translational masses in solution phase	Input

xmsr	Reciprocal of nodal rotational masses in solution phase	Input
numelt	Number of thick shell elements	Input, **from R13**
ixt	Thick shell element connectivities	Input, **from R13**
ntsh_del	if >0, element deletion option is active for thick shells	Input, **from R13**
tsharray	Time to delete thick shell elements, the value should be > time	Input, **from R13**
tshtim	Thick shell element deletion is checked when time is ≥ tshltim	Input, **from R13**

Notes: (1) The ires parameter has special meanings, for example ires < 0 means that |ires| input parameters should be read. (2) To get also the third node defining beam orientation, set *NREFUP* = 1 on *CONTROL_OUTPUT. (3) The subroutine should populate the array during the initialization phase, no data passed by LS-DYNA.

The parameter list for the user subroutine loadsetud is:

```
subroutine loadsetud(time,lft,llt,crv,iduls,parm,nod,nnm1)
```

The parameter list for this subroutine is quite short, see Table 13 for an overview, since an approach involving extensive use of common declarations is used. The objective of the loadsetud subroutine is to provide a scale factor for the loading, to be stored in the udl array, based on the accessible data and the parameters of the array parm (which is read by the loadud subroutine). Note that the loading type defined by the ltype variable of the keyword *USER_LOADING_SET is not passed to the subroutine loadsetud, but the value of the variable iduls can be passed as a load model ID for providing different user-defined loading models for different sets.

Table 13. The arguments to the subroutine loadsetud

Argument	Description	Input / Output
time	Current problem time	Input
lft, llt	Start, stop indices of arrays for vectorized input/output	Input
crv	Value of LCURV(1) at the current problem time	Input
iduls	ID of user loading set (1)	Input
parm	Array for storing input parameters(2)	Input
nod	internal node numbers	Input
nnm1	offset for node block	Input

Notes: (1) From the *USER_LOADING_SET – keyword. (2) from the *USER_LOADING keyword.

In the subroutine loadsetud access to nodal coordinates, displacements, temperatures etc. is provided via a common block, see Table 14 for a brief overview.

Table 14. Some of the arrays accessible via `common` block declarations in the subroutine `loadsetud`. Note that the dimension of all arrays is `nlq`

Argument	Description	Input / Output
`x1,x2,x3`	Current coordinate of node or element center	Input
`d1,d2,d3`	Displacement of node or element center	Input
`v1,v2,v3`	Velocity of node or element center	Input
`temp`	temperature of node or element center	Input
`udl(nlq)`	Array for storing user defined load scale factor	Output

8.4 Subroutine examples

In this Section, some basic examples of user defined loadings via the subroutines `loadud` and `loadsetud` are given. It shall be stressed that these examples are not intended for use in any kind of production analysis, and there may very well be errors or flaws in them.

8.4.1 Example of subroutine loadud

In order to illustrate the definition of a user loading subroutine `loadud`, three different cases are considered:

1. Application of a nodal force controlled by a curve as a function of time. This is similar to the built-in `*LOAD_NODE` keyword.
2. Application of a nodal force in a fixed direction in space, proportional to the magnitude of the nodal displacement and scaled by a load curve as a function of time.
3. Application of nodal force, counteracting the displacement and scaled by a load curve as a function of time. This is similar to the built-in functionality of an `*ELEMENT_DISCRETE` spring.
4. Read in parameters for `*USER_LOADING_SET`.

The user will have to select which load model to use, to which node the loading should be applied, a scale factor and a curve ID for scaling the loading as a function of time. From the `*USER_LOADING` keyword, the following variables will be used for models 1 - 3:

- *P1*: load model
- *P2*: node ID
- *P3*: Translational degree of freedom (1 – 3)
- *P4*: curve ID
- *P5*: scale factor.

The first part of the (pre R13) subroutine `loadud` follows, with subroutine and variable declarations:

```
      subroutine loadud(fnod,dt1,time,ires,x,d,v,a,ixs,
     . numels,ixb,numelb,idrflg,tfail,isf,p,npc,fval,iob,iadd64,numelh,
     . ixh,nhex_del,nbeam_del,nshell_del,hexarray,hextim,bemarray,
     . bemtim,shlarray,shltim,parm,numnp,fnodr,dr,vr,ndof,xmst,xmsr)
c
      include 'iounits.inc'
      include 'bigprb.inc'
      include 'txtline.inc'
      include 'nlqparm'
```

```
c
      parameter (NPARM=1000)
c     common/usrldv/parm(NPARM)
c
      integer*8 iadd64
      real*8 x
      real*8 d,dr
      dimension a(3,*),v(3,*),d(3,*),fnod(3,*),ixs(5,*),ixb(4,*),
     . x(3,*),tfail(*),p(*),npc(*),fval(*),iob(*),ixh(9,*),
     . hexarray(*),bemarray(*),shlarray(*),parm(*),fnodr(3,*),
     . vr(3,*),dr(3,*),xmst(*),xmsr(*)
c
      integer nid, idcrv, nidof, kk, nodext
      real sclfac, dist, f1,f2,f3, dfv(3)
```

In the original `dyn21.f` Fortran file of the usermat package, the comments after the subroutine declaration provide some documentation of the user loading subroutine, regarding for example parameters and some details on element deletion. These comments are omitted here.

The next part of the subroutine is active in the initialization phase. It reads in the parameter values (*P1*, *P2*, etc.) of the `*USER_LOADING` keyword and stores them in the array `parm`. Also, some messages are written in the `mes0*` files to confirm what parameter values are read. This part was taken (more or less) from the original example subroutine provided in the usermat package.

```
      if (ires.lt.0) then
        n=abs(ires)
        write(iomsg,1030)
        call prludparm(0,parm,0,0)
        mssg='reading user loading subroutine'
        if (longs) then
          do 11 i=1,n,8
          call gttxsg (txts,lcount)
          read (txts,'(8e20.0)',err=400) (parm(j),j=i,min(i+3,n))
          write(iomsg,1040) (j,parm(j),j=i,min(i+3,n))
          call prludparm(1,parm,i,min(i+3,n))
   11     continue
        else
          do 10 i=1,n,8
          call gttxsg (txts,lcount)
          read (txts,1020,err=400) (parm(j),j=i,min(i+7,n))
          write(iomsg,1040) (j,parm(j),j=i,min(i+7,n))
          call prludparm(1,parm,i,min(i+7,n))
   10     continue
        endif
        write(iohsp,1050)
        call prludparm(2,parm,0,0)
        return
      endif
C
      if (ndof.eq.0) return
      if(parm(1).eq.4.) return
```

The last two rows of this part will return from the subroutine in case LS-DYNA is in the initialization phase (indicated by `ndof` = 0) or the load model 4 is chosen, in which case only the parameter reading

should be active. The final part of the subroutine applies the nodal forces, depending on what load model is selected. First, the internal[15] node ID to which the force should be applied, must be obtained.

```
      nodext = int(parm(2))
      nid =lqfe(nodext,1)
```

For going from the user-defined node ID given by the variable *P2* of the `*USER_LOADING` keyword to the node ID used by LS-DYNA internally, the `lqfe` function must be used in an mpp implementation. In case the particular node ID is not accessible by the current mpi thread, `lqfe` will return a value ≤ 0 (while use of `lqf` or `lqf8` will cause Error termination. However, `lqf8` is required for use with smp/LS-DYNA). Then follows some conversion of the entries of the `parm` array to more useful variables.

```
      if (nid.gt.0) then
        nidof = int(parm(3))
        idcrv = lcids(int(parm(4)))
        sclfac = parm(5)
        if(sclfac.eq.0.0) sclfac = 1.
```

By the last row, a similar behavior to the built-in `*LOAD_` … keywords is obtained, since the load scale factor is reset to a default value of 1 in case zero or no value is input by the user. Then follows the actual modifications of the force vector, corresponding to the different loading models:

```
        if(parm(1).eq.1.)then
          fnod(nidof, nid) = fnod(nidof, nid) + fval(idcrv)*sclfac
        elseif(parm(1).eq.2.)then
          dist = sqrt(sum(d(1:3,nid)**2))
          fnod(nidof, nid) = fnod(nidof, nid) -
     1         fval(idcrv)*abs(sclfac)*dist
        elseif(parm(1).eq.3.)then
          dfv(1:3)= -d(1:3,nid)*abs(sclfac)*fval(idcrv)
          fnod(1:3, nid) = fnod(1:3, nid) + dfv(1:3)
        else
          cerdat(1)='Unsuported user loading model'
          call lsmsg(3,MSG_SOL+1447,ioall,ierdat,rerdat,cerdat,0)
        endif
      endif
c
      return
```

In case another load model than 1, 2, 3 or 4 is requested, the subroutine will trigger an error termination. Some examples of simulations using this `loadud` subroutine are provided in Section 8.5.1 and 8.5.2.

[15] In Table 7 some other useful functions for converting between internal and keyword-input numbering are listed.

8.4.2 Example of subroutine loadsetud

As an example of the subroutine `loadsetud`, the application of a hydrostatic[16] (buoyancy) loading was implemented. This will require the user to input, via the `*USER_LOADING_SET`,

- *SID*: a segment set ID,
- *LTYPE*: `PRESSS` to specify pressure on a segment set,
- *LCID*: a curve ID for scaling the loading,

and via the `*USER_LOADING` keyword the parameters

- *P1*: 4.
- *P2*: the direction of the gravity, in the global coordinate system, (1 for X-direction, 2 for Y and 3 for Z which also is the default)
- *P3*: density of the fluid,
- *P4*: the gravitational acceleration,
- *P5*: the reference level for zero pressure.

The coding for the `loadud` subroutine, required for reading in the parameters *P1* – *P5* to the `parm` array, was already presented in Section 8.4.1. The first part of the subroutine `loasetdud` follows, with subroutine, common block and variable declarations:

```
      subroutine loadsetud(time,lft,llt,crv,iduls,parm,nod,nnm1)
c
c***********************************************************************
c|  Livermore Software Technology Corporation  (LSTC)                  |
c|  ---------------------------------------------------------------   |
c|  Copyright 1987-2008 Livermore Software Tech. Corp                  |
c|  All rights reserved                                                |
c***********************************************************************
c
c     Input (not modifiable)
c       time : analysis time
c       x    : coordinate of node or element center
c       d    : displacement of node or lement center
c       v    : velocity of node or lement center
c       temp : temperature of node or element center
c       crv  : value of LCURV at time=time
c       iduls: id of user_loading_set
c       parm : parameters from user_loading
c       nod  : internal node numbers
c       nnm1 : offset for node block
c     Output (defined by user)
c       udl : user-defined load curve value
      include 'nlqparm'
      include 'iounits.inc'
c
      common/aux8loc/
     & x1(nlq),x2(nlq),x3(nlq),v1(nlq),
```

[16] Hydrostatic loading in LS-DYNA in general does not require the coding of a user subroutine but can be achieved by the `*DEFINE_FUNCTION` keyword (see the example of Ref. [1]).

```
     & v2(nlq),v3(nlq),d1(nlq),d2(nlq),
     & d3(nlq),temp(nlq),udl(nlq),
     & xx11(nlq),xx21(nlq),xx31(nlq),
     & xx12(nlq),xx22(nlq),xx32(nlq),
     & xx13(nlq),xx23(nlq),xx33(nlq),
     & xx14(nlq),xx24(nlq),xx34(nlq),
     & xctr(nlq),yctr(nlq),zctr(nlq),
     & f(nlq), tr1(nlq), tr2(nlq), tr3(nlq)
c
      dimension parm(*),nod(*)
C
      integer dof, kk
      real grav, rho, refl, fact, cord(nlq)
```

In this case, the original comments of the `dyn21.f` file are kept. Then follows a translation of the values of the `parm` array to explicit variables:

```
      dof = int(parm(2))
      rho = parm(3)
      grav = parm(4)
      refl = parm(5)
```

In order to simplify the final calculations, an array `cord` is assigned the current coordinate corresponding to the direction of the gravity:

```
C --- z is the default direction
      cord(1:nlq) = x3(1:nlq)
      if(dof.eq.1)then
        cord(1:nlq) = x1(1:nlq)
      elseif(dof.eq.2)then
        cord(1:nlq) = x2(1:nlq)
      endif
```

And then the final calculation of the hydrostatic force and the corresponding load factor `udl` follows

```
     do kk=lft,llt
       fact = max(0., (refl-cord(kk))*grav*rho)
       udl(kk)= crv*fact
      enddo
```

which concludes the user subroutine.

8.5 LS-DYNA simulation examples

In this Section, some LS-DYNA simulation examples of the user defined loading options are presented.

8.5.1 Nodal force by load curve

An L-shaped beam, see Figure 40, is fully constrained at its base and subjected to transverse loading (5 kN). The tip deflection when the loading is applied by the built-in keyword `*LOAD_NODE` and the user defined loading is compared in Figure 41. For the explicit analyses, the Y-displacement of the tip of the beam is identical for the two different load application methods. The differences between the implicit static analysis and the explicit analyses are explained by the dynamic effects induced by the ramp-up

time of 20 ms. It is concluded that the coding of Section 8.4.1 and the related interfaces of the usermat package gives an equivalent result as the built-in load application functionality.

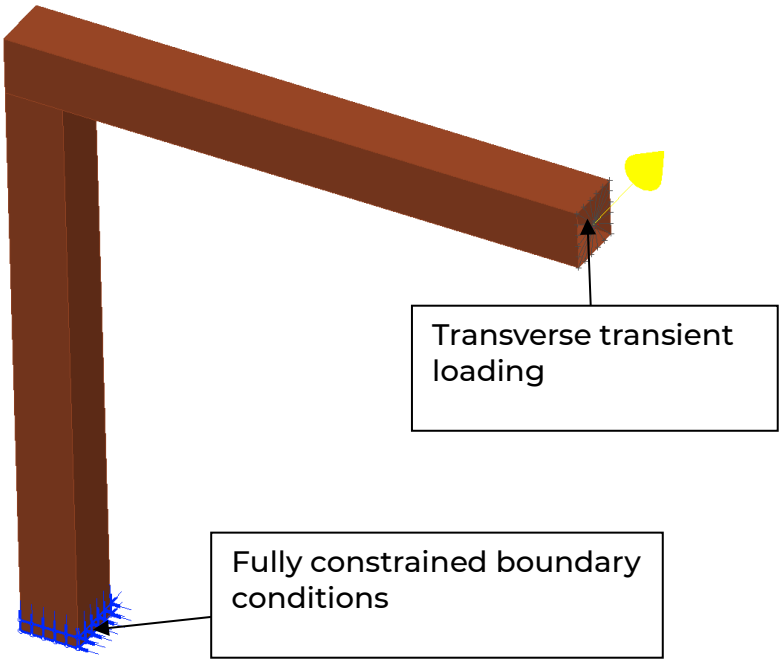


Figure 40. An L-shaped beam is fully constrained at its base and subjected to transverse loading at the center node of a `*CONSTRAINED_NODAL_RIGID_BODY`.

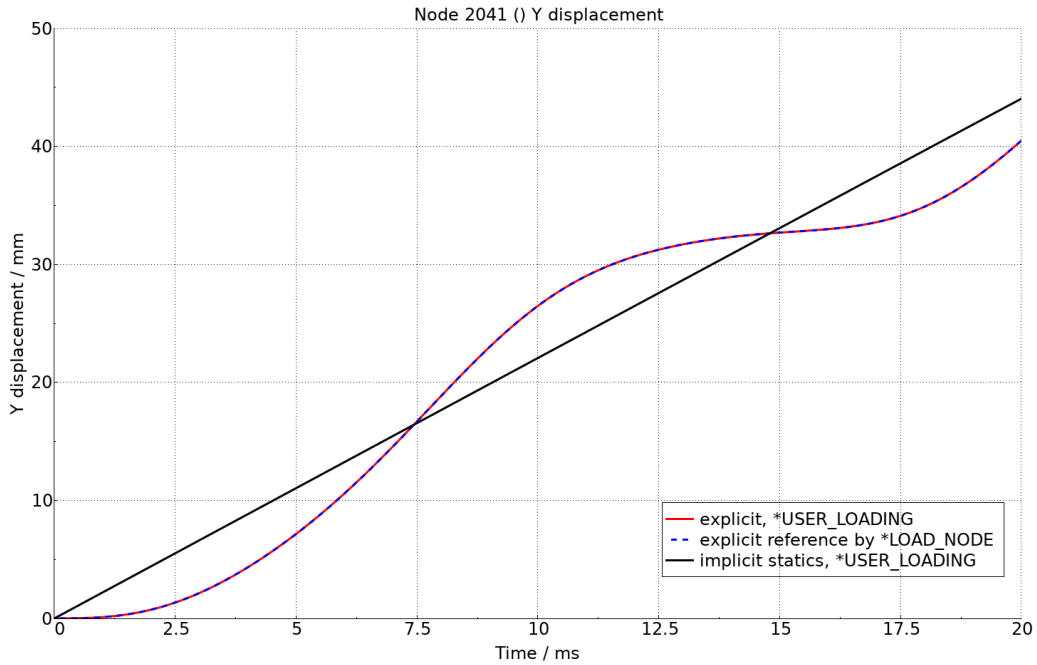


Figure 41. Results of LS-DYNA explicit and implicit simulations where the load is applied using `*LOAD_NODE` or `*USER_LOADING`.

8.5.2 Nodal force proportional to nodal displacement

In the second set of example analyses, the load model 3 as described in Section 8.4.1 is compared to a model version where a discrete spring element is used to apply a force counteracting the nodal displacement. The geometry of Figure 40 is used also in this case, and the loading (spring element) is applied at the center node of the constrained nodal rigid body at the end of the horizontal beam part. A transverse displacement of 40 mm is ramped-up during using `*BOUNDARY_PRESCRIBED_MOTION` with a death time of 20 ms, and the vibration motion of the L-beam is studied. If a constant curve is used, the user defined loading version gives the same tip deflection as the version using the built-in feature `*ELEMENT_DISCRETE`, see Figure 42. It is again concluded that the coding of Section 8.4.1 and the related interfaces of the usermat package gives an equivalent result as the built-in load functionality.

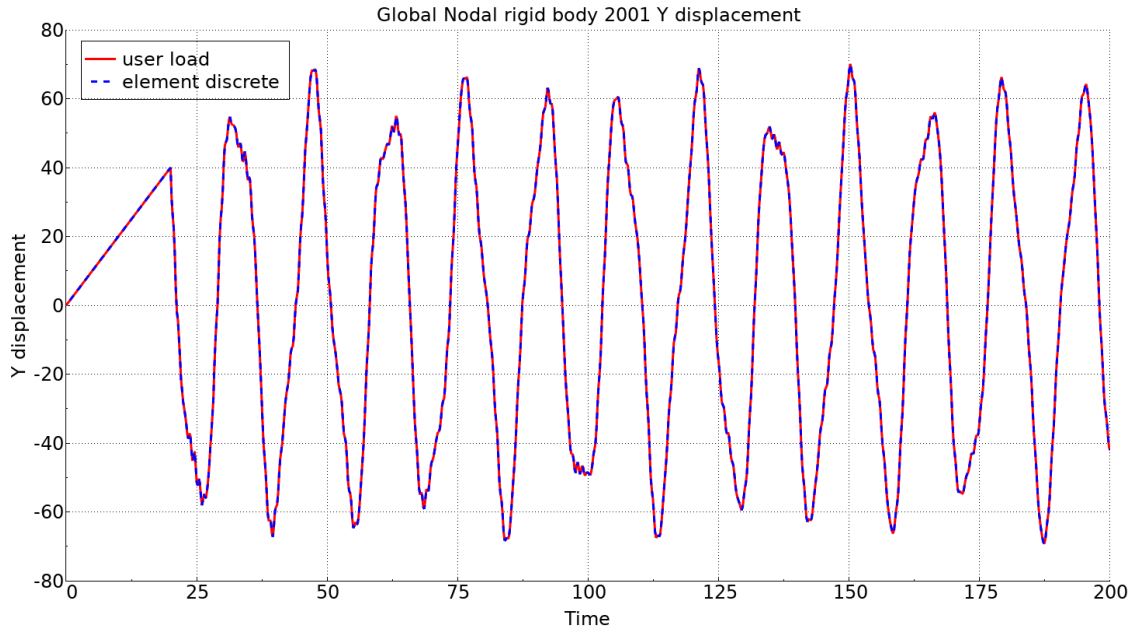


Figure 42. Comparison of tip displacement from the `*USER_LOADING` version (solid red curve) and the discrete element version (dashed blue curve).

By defining a non-constant curve for scaling the force, see for example the left image of Figure 43, corresponding to a discrete spring with time-varying stiffness, functionality that is not present by the built-in keywords is obtained. Clearly, the vibrational results are influenced by this time-varying stiffness, see the right image of Figure 43.

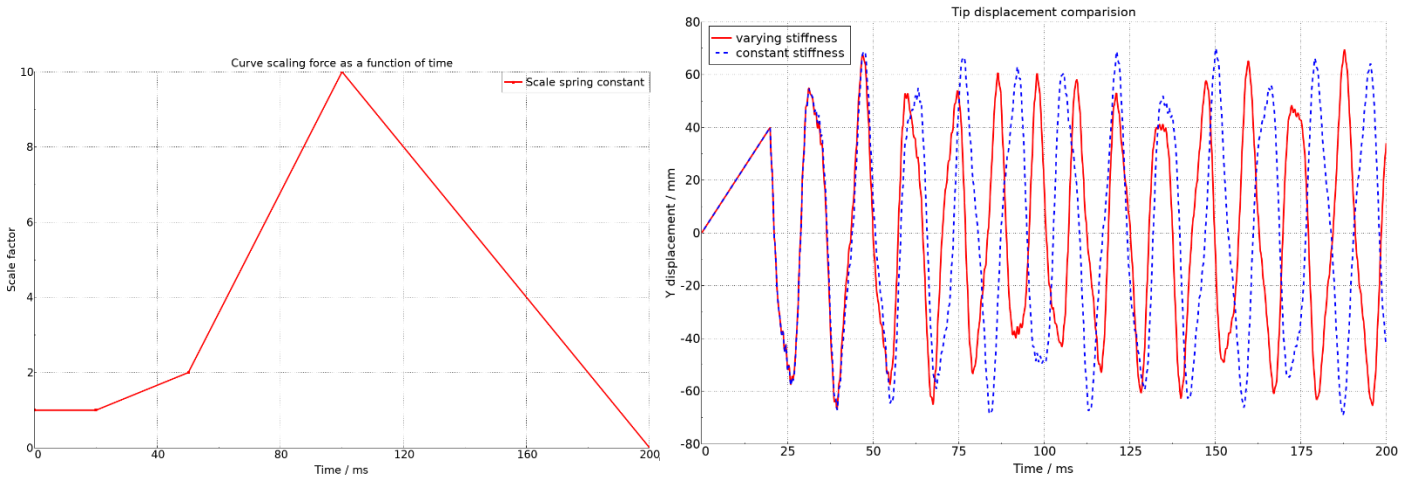


Figure 43. The left image: Curve for scaling the force as a function of time, corresponding to a varying spring stiffness. The right image: results comparison.

8.5.3 Hydrostatic pressure loading

In this example, the user defined hydrostatic loading as described in Section 8.4.2 is compared to the built-in keywords `*LOAD_SEGMENT_SET` and `*DEFINE_FUNCION` in an implicit, transient dynamic, analysis. The geometry of this example is shown in Figure 44, where the reference level is outlined as a semi-transparent plane. The Z-displacement of node ID 5375 is compared in Figure 45. It is concluded that the results of the different load application techniques are very close.

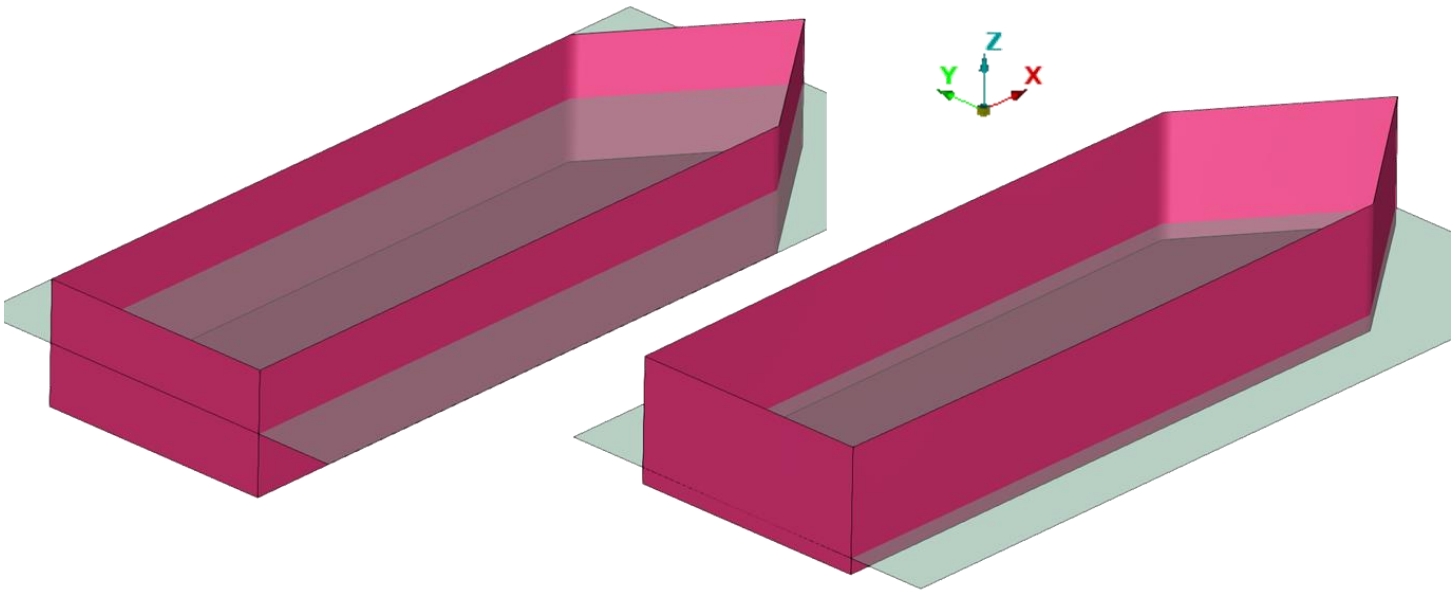
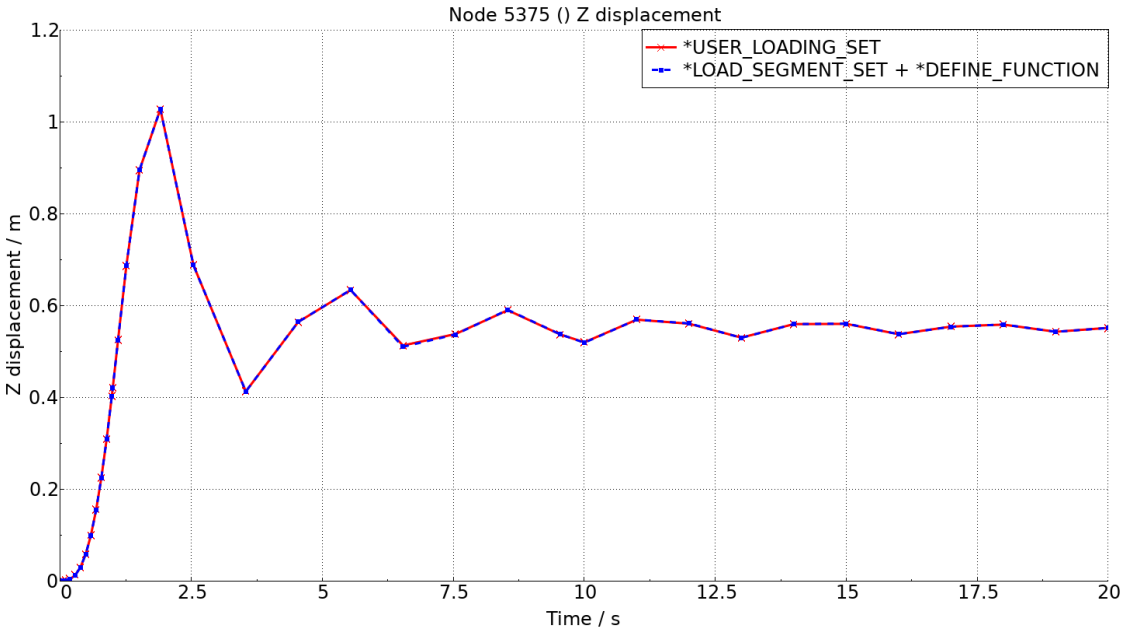


Figure 44. Geometry for the hydrostatic loading example. The left image shows the initial (guessed) configuration, and the right image shows the (equilibrium) position after 20 seconds. The reference level is indicated by a semi-transparent plane.



e

Figure 45. Comparison of the Z-displacement of the node ID 5375 from the simulation with user defined loading (solid red curve) and the simulation using built-in load application functionality (dashed blue curve).

9 Other user interfaces

In this section some other possibilities for user defined interaction with LS-DYNA are listed. Some of them will be described in more detail in coming revisions of this document.

- For wear analysis (`*CONTACT_ADD_WEAR`) a customized wear law can be defined by setting `WTYPE` < 0 and using the subroutine `usrwear` in `dyn21cnt.f`.
- For non-Mortar tiebreak contacts (`*CONTACT_AUTOMATIC_SURFACE_TO_SURFACE_TIEBREAK_USER`), custom tie failure conditions can be defined using the subroutines `utb101, …, utb105` in `dyn21cnt.f`.
- User defined thermal conductance for the 3D thermal contacts can be defined by the subroutine `usrhcon` in `dyn21.f`. It is invoked by the keyword `*USER_INTERFACE_CONDUCTIVITY`.
- Shell and solid elements can be defined by the user, either via the keywords `*DEFINE_ELEMENT_GENERALIZED_{SHELL/SOLID}` or via the user-defined interfaces of subroutines `ushl_b`*`YYY`* in `dyn21ushl.f` and `usld_b`*`ZZZ`* in `dyn21usld.f`, see Appendix C of Ref. [1], and Ref. [18]
- User defined damage / failure criteria can be defined for some materials by the subroutine `matusr_24` in `dyn21.f`. The user define failure criteria is then invoked by setting *`FAIL`* < 0 for the material types: 24, 36, 114, 123, 124, 133, 155, 182, 225, 238, 243, 251, 255. Also for MAT103, by the subroutine `matusr_103`.
- For interacting with other solvers in coupled analysis, the keyword `*COUPLE_TO_OTHER_CODES` can be used, which requires the family of user subroutines in the Fortran file `couple2other_user.f`.
- For implicit analysis, it is possible (from R11) to provide a user-defined linear equation solver by setting *`LSOLVR`* = 90 on `*CONTROL_IMPLICIT_SOLVER`.

References

[1] Ansys, LS-DYNA keyword user's manual Volume I, 2024 (see also http://lstc.com/download/manuals).
[2] Ansys, LS-DYNA keyword user's manual Volume II, 2024.
[3] Ansys, LS-DYNA theory manual, 2024.
[4] Karlsson, J., and Borrvall, T., Material modelling in LS-DYNA, Dynamore Course Notes, 2019.
[5] Karlsson, J., User defined material models in Ansys LS-DYNA, Ansys Course Notes, 2023.
[6] Erhart, T., User defined interface in LS-DYNA, Dynamore Course Notes, 2016.
[7] Erhart, T., User defined materials in LS-DYAN, Dynamore Course Notes, 2019.
[8] Forsberg, J., Contacts in LS-DYNA, Dynamore Course Notes, 2020.
[9] Contact types in LS-DYNA, Internet source: https://www.dynasupport.com/tutorial/contact-modeling-in-ls-dyna/contact-types
[10] Internet source: https://software.intel.com/en-us/fortran-compilers
[11] Saaby Ottosen, N., and Ristinmaa, M., The mechanics of constitutive modelling, Division of Solid Mechanics, Lund University, Lund (1999) Sweden
[12] Lemaitre J., and Chaboche, J. L., Mechanics of solid materials, Cambridge University Press, Cambridge 1998.
[13] Holzapfel, G. A., Nonlinear solid mechanics, John Wiley & Sons, New York 2000
[14] Andrade, F., Feucht, M. and Haufe, A., On the prediction of material failure in LS-DYNA: A comparison between GISSMO and DIEM, 13th International LS-DYNA Users Conference, Detroit 2014.
[15] Mattiasson, K., Jergeus, J., and Dubois, P., Models for strain path independent necking prediction in LS-DYNA, 9th European LS-DYNA Conference 2013.
[16] MG GenYld + CrachFem, internet source: https://www.matfem.de/data/Info-MFGenYld-CrachFEM.pdf
[17] Digimat – The material modelling platform, internet source: https://d2f709itdech1g.cloudfront.net/cdn/farfuture/Juq5VUV3VESVZlLYpVryCZmAdcjItn7uUzGFt9WWgRQ/mtime:1553075971/sites/default/files/general_software_brochure_2019.0.pdf (https://www.e-xstream.com/products/digimat/about-digimat)
[18] Borrvall, T., A user-defined element interface in LS-DYNA, 9th International LS-DYNA Users Conference, Detroit 2006.
[19] Adoum, M., and Pitot, H., Examples' manual for *USER_LOADING optoin, 4th European LS-DYNA Users Conference, Ulm 2003.
[20] Erhart, T., An overview of user-defined interfaces in LS-DYNA, 9th LS-DYNA Forum, Bamberg 2010, internet source: https://www.dynamore.de/de/download/papers/forum10/papers/L-I-01.pdf
[21] Benson, D. J., Radial return, internet source: https://www.dynasupport.com/tutorial/computational-plasticity/radial-return
[22] LS-PrePost Online Documentation, internet source: http://www.lstc.com/lspp/
[23] META Post Processor version 20.0.X. User Guide, BETA CAE Systems International AG, 2019.
[24] Material selector for LS-DYNA, internet source: http://lstc.com/dynamat/
[25] Anon. S-Rail Benchmark Problem (1996). Proceedings of the 3rd International Conference: Numerical Simulation of 3-D Sheet-Metal Forming Processes – Verification of Simulations with Experiments (NUMISHEET '96). Eds. Lee, Kinzel and Wagoner, September, 1996, pp. 612-799.
[26] NMAKE Reference, internet source: https://docs.microsoft.com/en-us/cpp/build/reference/nmake-reference?view=vs-2019
[27] Hidling, D., User-defined modules in LS-DYNA and anti-virus software, Dynamore technical Note, April 29, 2020.
[28] Sigvant, M., et *al.*, Friction modelling in sheet metal forming simulations: application and validation on an u-bend product, internet source: https://www.triboform.com/tribo/wp-content/uploads/2016/02/Technical-paper-U-bend-product_TriboForm-Engineering.pdf
[29] Borrvall, T., et al., Using MAT_ADD_INELASTICITY for Modelling of Polymeric Networks, 13th European LS-DYNA Conference 2021, Ulm, Germany.
[30] Svenning, E., Co-simulation with LS-DYNA through FMI, DYNAmore Nordic document (Webinar 2020-10-22) available from files.dynamore.se > Client Area
[31] Tong, X., and Yeh, I., Cross-platform co-simulation for vehicle safety analysis, 16th International LS-DYNA conference 2020 (Virtual Event)

[32] Andersson, H., A co-simulation tool applied to hydraulic percussion units, Ph.D. Thesis, Linköping University, Linköping 2022
[33] Wang, Z., Sha, Y., and Jakobsen, J.B., Coupled analysis of external dynamics and internal mechanics in ship-floating bridge collision studies, Proceedings of the OMEA, Singapore 2024.

10 Executing user-compiled LS-DYNA binaries under Windows

In some cases, the execution of a user-compiled LS-DYNA executable may be prevented by the different security measures of Windows 10. The text of this Appendix is based on Ref. [27], and describes the background and possible resolution to such situation.

To defend Windows 10 computers against unsafe or malicious software being executed or installed typically several approaches are used, including:

- Microsoft Defender SmartScreen (MDS)
- Microsoft anti-virus software (AV)
- Third party anti-virus software (AV)

MDS and AV will typically prevent execution of any executable that is not white-listed and/or code signed traceable back to a trusted root authority.

An independent software vendor will typically white-list and code sign all their executables and software installers with Microsoft and major third-party AV-software products to avoid issues when the end user is to install or execute the software from the ISV.

However, developing user-defined features in LS-DYNA in a Windows environment will result in a new, customized LS-DYNA binary (.exe – file) and therefore AV and/or MDS may prevent execution. In turn, this makes it impossible to make use of the user-defined features with LS-DYNA. A solution to this needs to be sought by cooperation between the engineers developing user-defined features and the department responsible for Windows IT security and management at each specific company (or university). Some of the following approaches are typically applied:

1. Trusted users are allowed to locally white-list the custom LS-DYNA executables at the company or on specified computers.
    - This method works well, though it can have some IT-security implications.
2. The policy to require white-listed/signed code is disabled on selected computers and/or for trusted users.
    - This method works well, though it can have some IT-security implications that are larger than method 1.
3. Allow trusted users to override warnings or blockings from the AV software or MDS for each execution.
    - This is rarely used with LS-DYNA because it prevents the use of queuing systems with LS-DYNA and that reduces work productivity – sometimes considerably.
4. The custom LS-DYNA executable is white-listed at the relevant AV software companies and with MDS (code signing).
    - This is usually not doable as the during development of user-defined features as these are rebuilt many times per day and the white-listing process is lengthy.

Keywords: Ansys LS-DYNA; user-defined interfaces; Fortran

ANSYS, Inc.
Southpointe
2600 Ansys Drive
Canonsburg, PA 15317
U.S.A.
www.ansys.com