USER DEVELOPMENTS IN ZSoil®

ZSoil®.PC 200101 report

by
A. Truty. K. Podles

GeoDev Sarl
PO Box CH-1001 Lausanne
Switzerland
https://zsoil.com

since 1985

Contents

Chapter 1

Organization of compilation/linking/debugging environment

During installation of ZSoil program its main components are placed in three directories as shown in the following window

Window 1-1: Location of ZSoil®files after installation procedure

ZSoil®.PC

```
ZSoil tree (Windows 7)
    ├── C:/Users/All Users/ZSoil v2018/Full (configuration and temporary files)
    │       ├── CFG
    │       └── ZTMP
    ├── C:/ProgramData/ZSoil v2018/Full (configuration and temporary files)
    │       ├── CFG
    │       └── ZTMP
    └── C:/Program Files/ZSoil/ZSoil v2018 v.18.00 x64 (binaries)
```

Window 1-1

There are two possible user developments in ZSoil®2023 ie. implementation of user supplied constitutive model for continuum, and user defined output foreseen for shell, continuum and beam elements. To develop constitutive models within ZSoil®environment Intel Fortran compiler must be used while user defined output requires Microsft Visual C++ one. Both compilers must be accessible from Microsoft Visual Studio 2012 or newer. Software development kit (SDK) for user development is not installed automatically during ZSoil®installation. SDK can be downloaded from https://www.zsoil.com/upgrades/v2018/. Local ZSoil®depository (including source files, library files and projects) required for the aforementioned developments can be located at any directory indicated by the user. It will be labeled as UserDevelopments.

Window 1-2: Local ZSoil®depository form user developments

ZSoil®.PC

The UserDevelopments directory is organized as follows

- UserDevelopments
  - exe-x64 (here is local exe directory for 64 bit version)
    - CFG
  - H (common headers *.h)
  - zutl (headers *.h for zutl module to handle units system)
  - zmate (headers *.h for zmate module to handle beam cross sections)
  - lib-x64 (libraries *.lib)
  - calc
    - prj
      - user_models (here project for user supplied constitutive model is kept)
    - src
      - src (headers *.inc files)
      - user_models (user sources *.for depository)
        - mod-x64 (precompiled 64 bit modules
  - z_post
    - user_output (user source *.cpp and project files)

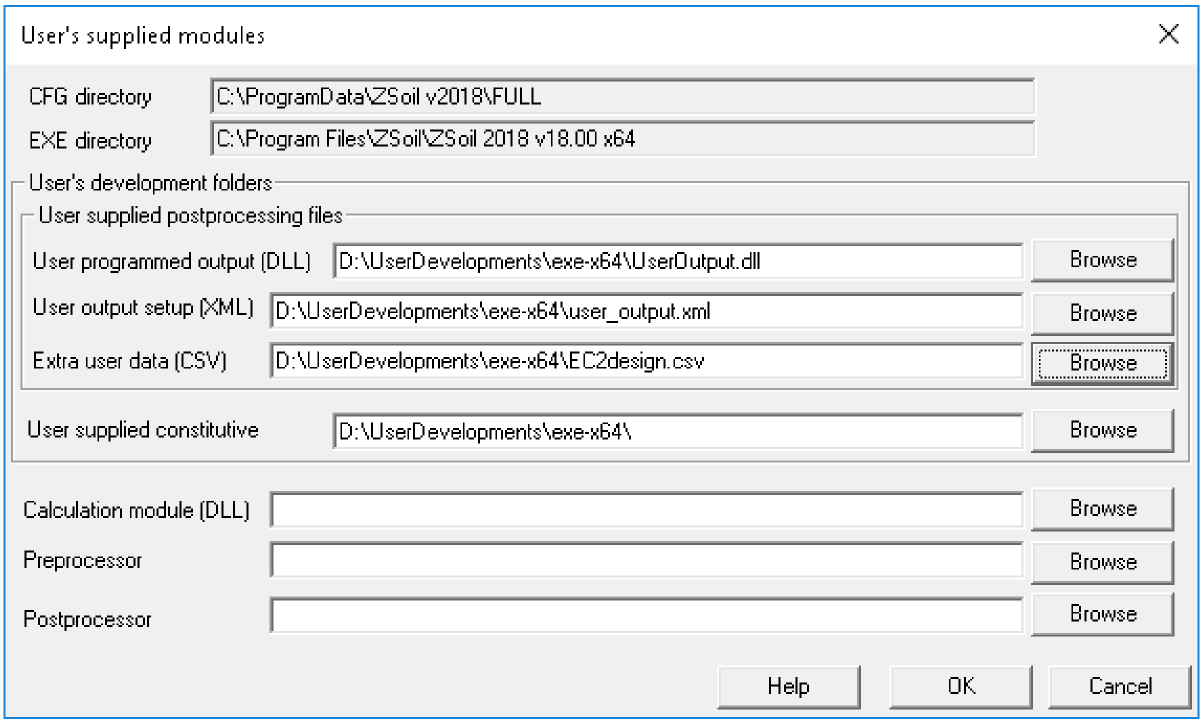
Window 1-2

The major part of the ZSoil calculation module has been written using Fortran 90/95/2003/2008 and C/C++ programming languages. Part of the development for user supplied constitutive model must be written using Fortran language. Intel Visual Fortran, compatible with Microsoft Visual Studio 2017, must be used. The user defined output must be written in C++ language. Projects for the two user developments are prepared and ready to be used.

Window 1-3: Pluging user prepared dlls in ZSoil®2020

ZSoil®.PC

Compiled dynamic linked libraries (usermodels.dll and/or useroutput.dll) must be pluged in ZSoil®2020 through the following dialog box (from main ZSoil®menu click on `System configuration/User's supplied modules`).


User's supplied modules

CFG directory: C:\ProgramData\ZSoil v2018\FULL

EXE directory: C:\Program Files\ZSoil\ZSoil 2018 v18.00 x64

User's development folders

User supplied postprocessing files

User programmed output (DLL): D:\UserDevelopments\exe-x64\UserOutput.dll

User output setup (XML): D:\UserDevelopments\exe-x64\user_output.xml

Extra user data (CSV): D:\UserDevelopments\exe-x64\EC2design.csv

User supplied constitutive: D:\UserDevelopments\exe-x64\

Calculation module (DLL):

Preprocessor:

Postprocessor:

Figure 1: Dialog box for plugging user developed modules

In case of user constitutive models we do declare the path to the resulting **usermodels.dll** dynamic link library and a script file **zsoil.usm** while for user programmed output we do indicate location of 3 files i.e. the resulting user output dll, the xml configuration file and extra auxiliary user data file.

Window 1-3

Chapter 2

User supplied constitutive models for continuum

2.1 List of modifications with respect to ZSoil®version 2013.

In order to be able to compute huge 3D models the implemented user models in previous ZSoil®versions will not directly work within 2018 environment. The following changes must be made in user supplied routines for constitutive models (see usr1.for routine as an example).

1. declaration of function → integer*4 :: IBUF_ElePtrGet is now obsolete

2. instead of aforementioned declaration the following statements must be added

```
!MS$IF   DEFINED (__X64)
integer*8 :: IBUF_EleHandleGet
!MS$ELSE
integer*4 :: IBUF_EleHandleGet
!MS$ENDIF
integer*4 :: this_BUF (*)
pointer (iptr,this_BUF)
```

   This modification allows to adress arrays with indices above 2^{31}-1 that is the limit for four byte integers (obviously under 64 bit systems).

3. before calling state update procedure (after keyword L_GP_NEW_STATE) the following modification must be made

```
c OBSOLETE since 2014 iptr = IBUF_ElePtrGet ( icurrNr_ELE,M )
c OBSOLETE since 2014 T = ELG_GetTempFromStorage (M(iptr),TisGiven)
iptr = IBUF_EleHandleGet ( icurrNr_ELE,M ) ! DON'T EDIT
T = ELG_GetTempFromStorage ( this_BUF,TisGiven) ! DON'T EDIT
```

2.2 Introduction

The ZSoil®2023 system allows the implementation of a user supplied constitutive model for continuum elements. This option is available to every user who has a legal license for 2023 (professional or academic).

Implementation and using user supplied constitutive models consists of the three major development steps:

1. creating a script file which defines model parameters (zsoil.usm file)
2. programming of a user supplied constitutive theory and then compiling it and linking to produce usermodels.dll
3. making modifications within the system to instanciate user model

In the following sections we will describe how to implement a Huber-Mises elasto-plastic model with plastic parameter dependent on temperature and an additional dependency of flow parameters on temperature.

2.3 Creating script files for user model interface

We assume that for the time being user can redefine material properties only in the two groups : "Elastic" and "Nonlinear" while the other ones can exclusively be extended (user can add new parameters which are not handled by the original system but can be handled by the user himself).

The script file **zsoil.usm** must be placed in the same folder as usermodels.dll (see 1).

In the following example this script file defines a Huber-Mises elasto-plastic model which inherits the basic parameters setup from standard elastic model and thus it requires to add group "Nonlinear" with 2 parameters: cohesion c and evolution function number (load time function) $LTFc(T)$. This evolution function modifies the cohesion c that may depend on the current temperature value (hence heat project should be attached to activate it). In addition fluid thermal dilatancy parameter is added to the "Flow" parameters group.

To avoid unexpected application errors when analyzing the script file the following rules must be fulfilled during its preparation.

- all keywords begin with @ character in the first column
- <Group> can be one of:
  - ⋆ @ MAIN (this group of parameters cannot be modified)
  - ⋆ @ DENS (this group of parameters cannot be modified)
  - ⋆ @ ELAS (elastic parameters can be replaced)
  - ⋆ @ GEOM (this group of parameters cannot be modified)
  - ⋆ @ FLOW (this group of parameters can be only extended)
  - ⋆ @CREEP (this group of parameters cannot be modified)

  - ⋆ @ NONL (nonlinear parameters can be replaced)
  - ⋆ @ HEAT (this group of parameters can be only extended)
  - ⋆ @HUMID (this group of parameters can be only extended)
  - ⋆ @ INIS (this group of parameters cannot be modified)
  - ⋆ @ STAB (this group of parameters cannot be modified)

- <Group> string is always 6 characters long with additional spaces added between character @ and the keyword
- <mode> can be one of
  - ⋆ REPLACE (means that standard parameters from template model will be replaced)
  - ⋆ ADD (means that a new set of parameters will be added to the standard ones)
- <string> is a string
- <par> is just a parameter string equivalent to <string>
- <count> is an integer value (usually nr of something)

The general structure of a script file for a single supplied material model is as follows:

@MODEL_NAME

<string> (model name up to 10 characters)

<string> (name to be used in user interface dialog box)

<string> (here always use -> ELASTIC_V as a template model name)

<Group><count><mode>

$< par >$

$< par >$

....

up to <count>

In the considered case we want to define a model which inherits its basic properties setup from the standard elasticity model (obligatory for time being (!)), labeled as USER1, with the 2 "Nonlinear" parameters, single additional parameter for "Flow" properties and additional single parameter for group "Heat".

In that case the script file must be defined as follows:

@MODEL_NAME

USER1

User model number 1

ELASTIC_V

@ NONL 2 REPLACE

LTF nr for c(T) c(T) = c LTF (T(t))

@ FLOW 1 ADD

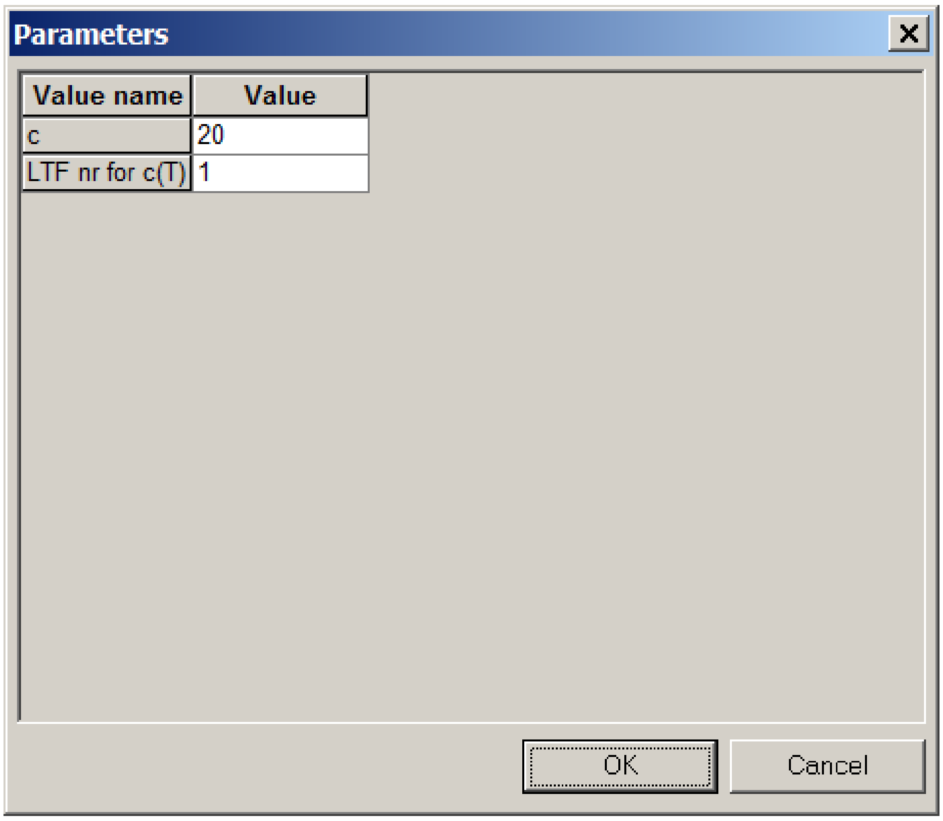
LTF nr for k(T) k(T) = k LTF (T(t))

@ HEAT 1 ADD

Fluid thermal dilatancy

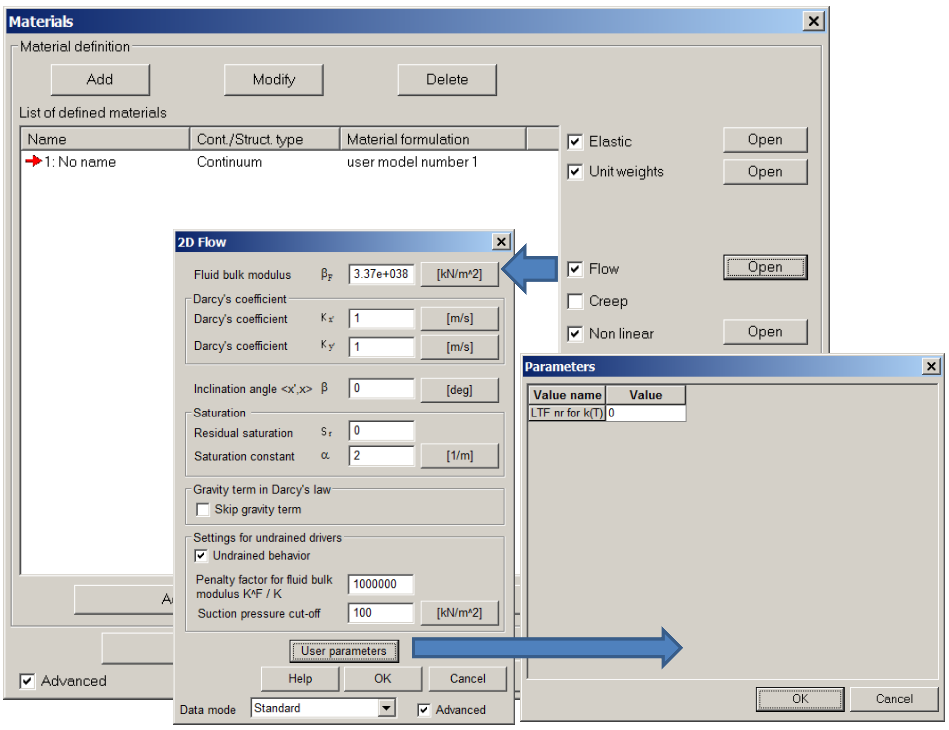
Remarks:

- it is possible to define up to 60 user supplied models in the script file zsoil.usm
- units system is not available for user defined parameters
- it is worth to add few (empty) parameters to the supplied model because each modification of number of parameters will result in loss of back compatibility
- the script file must be prepared with great care as any mistake may result in application error
- the ZSoil menu analyzes the script file and generates the user interface dialog box automatically (see Fig.(2.1))
- in the case of extension of the standard parameters groups like "Flow" or "Heat" an additional button "User parameters" is added in standard dialog box (see Fig.(2.2))



Value name	Value
c	20
LTF nr for c(T)	1

Figure 2.1: Dialog box for *Nonlinear* parameters



Materials

Material definition

List of defined materials

Name	Cont./Struct. type	Material formulation
1: No name	Continuum	user model number 1

- [✓] Elastic
- [✓] Unit weights
- [✓] Flow
- [ ] Creep
- [✓] Non linear
- [✓] Advanced

2D Flow

Fluid bulk modulus	β_F	3.37e+038	[kN/m^2]

Darcy's coefficient

Darcy's coefficient	$K_{x'}$	1	[m/s]
Darcy's coefficient	$K_{y'}$	1	[m/s]
Inclination angle <x',x>	β	0	[deg]

Saturation

Residual saturation	S_r	0	
Saturation constant	α	2	[1/m]

Gravity term in Darcy's law

- [ ] Skip gravity term

Settings for undrained drivers

- [✓] Undrained behavior

Penalty factor for fluid bulk modulus K^F / K	1000000	
Suction pressure cut-off	100	[kN/m^2]

Data mode: Standard

- [✓] Advanced

Parameters

Value name	Value
LTF nr for k(T)	0

Figure 2.2: Dialog box for *Flow* parameters

2.4 Compiling, linking and debugging

Project file to compile and link usermodels.dll file, prepared with aid of the MS Visual Studio 2012, can be found in directory *UserModels/calc/prj/user_models*.

In the *CFG* directory one may keep the int.dum file that contains information on current debugged data (content of this file may look as follows:
"C:\Program Files\ZSoil\ZSoil 2018 v18.00 x64\Z_Calc.exe" [#@STARTDIR C:\ProgramData\ZSoil v2018\FULL@#] [#@PRJ D:\UserDevelopments\inp\mydata@#])

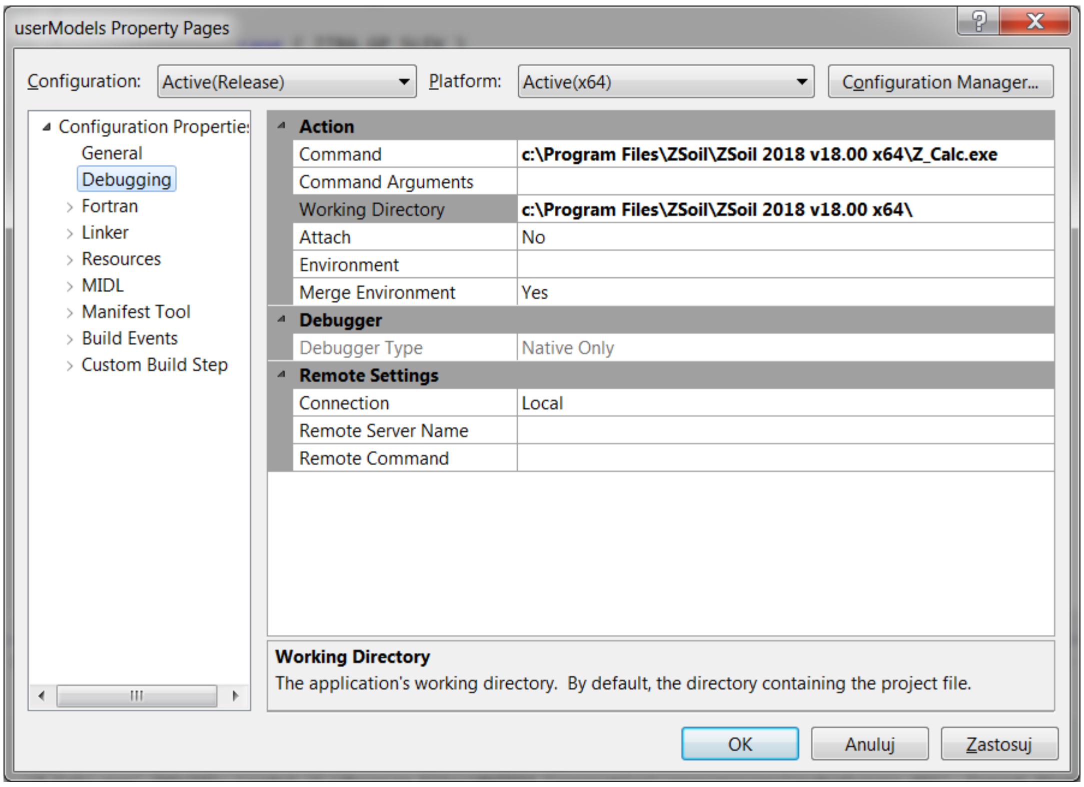
2.4.1 *Remarks on debugging the code*

It is a standard situation that content of common blocks which are used to communicate with ZSoil is not visible. This is a major drawback of Visual Studio debugger. To remedy this problem please make your own temporary arrays or variables and copy data from common block to this auxiliary objects (these object will be well visible under debugger).

2.4.2 *Settings for Microsoft Visual Studio*

Default settings for *Command* and *Working Directory* must be changed when default ZSoil®folder has been changed during installation (see 2.3).

Additionally, information on current debugged data can be copied from int.dum file and pasted to *Command Arguments* (see 2.4).



userModels Property Pages

Configuration: Active(Release) Platform: Active(x64) Configuration Manager...

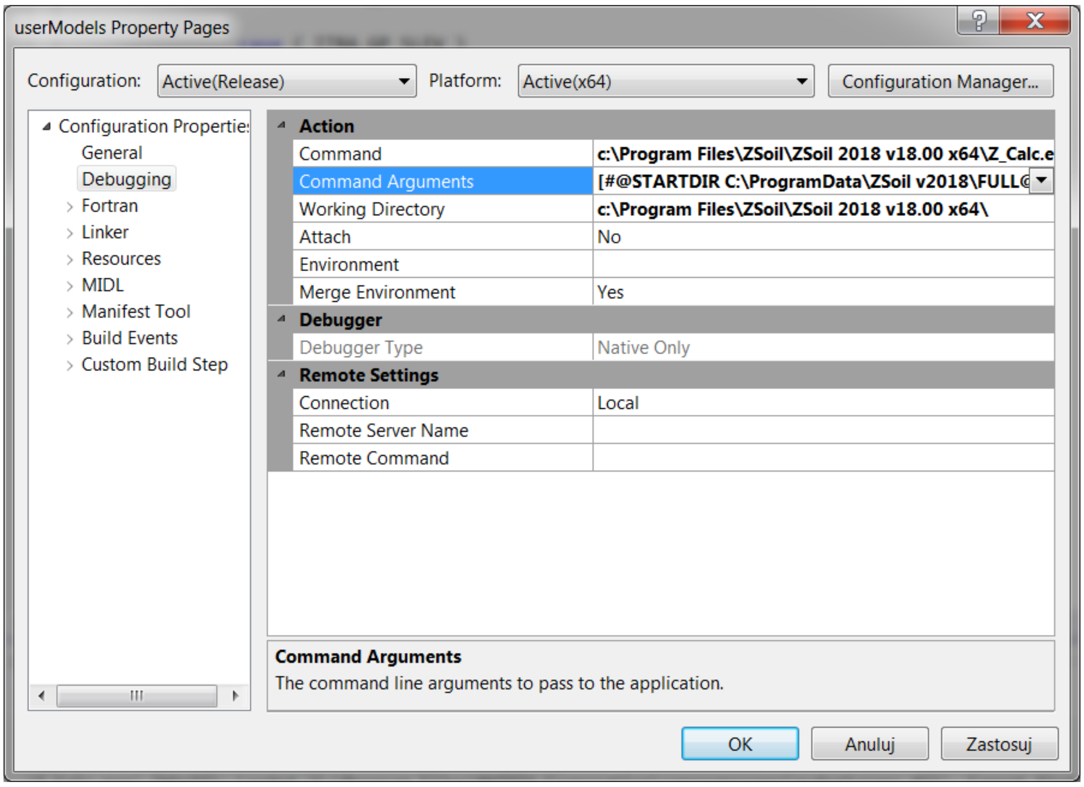
Configuration Properties
- General
- Debugging
- Fortran
- Linker
- Resources
- MIDL
- Manifest Tool
- Build Events
- Custom Build Step

Action	
Command	**c:\Program Files\ZSoil\ZSoil 2018 v18.00 x64\Z_Calc.exe**
Command Arguments	
Working Directory	**c:\Program Files\ZSoil\ZSoil 2018 v18.00 x64**
Attach	No
Environment	
Merge Environment	Yes
Debugger	
Debugger Type	Native Only
Remote Settings	
Connection	Local
Remote Server Name	
Remote Command	

Working Directory
The application's working directory. By default, the directory containing the project file.

OK Anuluj Zastosuj

Figure 2.3: Default *Command* and *Working Directory*



userModels Property Pages

Configuration: Active(Release) Platform: Active(x64) Configuration Manager...

Configuration Properties
- General
- Debugging
- Fortran
- Linker
- Resources
- MIDL
- Manifest Tool
- Build Events
- Custom Build Step

Action	
Command	**c:\Program Files\ZSoil\ZSoil 2018 v18.00 x64\Z_Calc.e**
Command Arguments	**[#@STARTDIR C:\ProgramData\ZSoil v2018\FULL@**
Working Directory	**c:\Program Files\ZSoil\ZSoil 2018 v18.00 x64**
Attach	No
Environment	
Merge Environment	Yes
Debugger	
Debugger Type	Native Only
Remote Settings	
Connection	Local
Remote Server Name	
Remote Command	

Command Arguments
The command line arguments to pass to the application.

OK Anuluj Zastosuj

Figure 2.4: *Command Arguments* settings

2.5 Programming user supplied constitutive model

2.5.1 Introduction

The main goal of each constitutive model is to find a new stress state for a given strain increment at a given time instance $N+1$ once the stress and other state variables are known for the time instance N. Usually the constitutive level is limited to a single integration point (so-called Gauss point in the case of standard finite elements) and thus the interface between integration point (local level) and the parent finite element is an important part of the code to be developed by the programmer. This part of the code has been prepared by ZSoil developers and presented in the template file usr1.for containing the full code for Huber-Mises model with cohesion value being dependent on temperature. The standard actions to be handled during computation at the integration point level are as follows:

- update of the state variables (like stresses for instance) (via case(L_GP_UPD_STATE)) and telling the system whether plasticity has occured (this information is returned through iplas_TRA variable kept in the common block defined in trans.inc file)
- computation of a new stress state for given effective strain increment(after subtraction of the initial and creep strain increments) (via case(L_GP_NEW_STATE)); note that for models with constant elastic parameters the elastic stiffnes matrix is given through Dev_TRA matrix (transferred via common block defined in trans.inc file); in case of nonlinear elastic models the content of that matrix must be set exactly at that place; the computed new stress state must be sent to the finite element level through array Sact_TRA (kept in common block defined in trans.inc file)
- initialization of all the state variables kept in the internal Gauss point storage
- setting the size of the internal gauss point storage size (expressed in integer*4 words); the storage size must be returned through InfoOut array
- setting symmetry status for tangent stiffness matrix; the information whether the tangent stiffness matrix is symmetric (1) or not (0) is returned through InfoOut array
- setting the current elastic stiffness matrix D; the current stiffness matrix must be returned through InfoOut array
- setting the current elastic compliance matrix C; the current compliance matrix must be returned through InfoOut array
- printing the material properties for groups specific for given model
- returning the information on
  - ⋆ current stress state (return through Sact_TRA array defined in common block in trans.inc file)
  - ⋆ Poisson ratio (return through v_TRA variable defined in common block in trans.inc file)
  - ⋆ current stress level (return through Slev_TRA variable defined in common block in trans.inc file)
  - ⋆ current plasticity index (return through Iplas_TRA variable defined in common block in trans.inc file)

  - ⋆ current Young modulus (return through E_TRA variable defined in common block in trans.inc file)

- modification of strength parameters during stability analysis

2.5.2 *Adding user model call to the list of custom model calls*

To activate models defined in the script file user must call them within the body of the subroutine SUM_Models which is listed below (see SuppliedModels.for file).

```
c=========================================
subroutine SUM_Models (model,iorder,Props,IGPbuff,InfoIn,InfoOut,M)
c=========================================
!MS$ IF DEFINED (__BUILD_DLL_ZCALC)
    !MS$ ATTRIBUTES DLLEXPORT :: SUM_Models
!MS$ END IF
include '..\\nodecl.inc'
include '..\\models.inc'
include '..\\prop.inc'
!MS$ IF DEFINED (__BUILD_DLL_ZCALC)
    !MS$ ATTRIBUTES DLLIMPORT :: /PRO_models_common/
!MS$ END IF
!MS$ IF DEFINED (__BUILD_DLL_ZCALC)
    !MS$ ATTRIBUTES DLLIMPORT :: /PRO_groups_common/
!MS$ END IF
!MS$ IF DEFINED (__BUILD_DLL_ZCALC)
     !MS$ ATTRIBUTES DLLIMPORT :: /PRO_common/
!MS$ END IF
integer*4 :: iorder          ! action to be activated for this gauss point
integer*4 :: model           ! model index
real*8 :: props (*)          ! array with properties
integer*4 :: IGPBuff (*)     ! gauss point internal storage
integer*4 :: InfoIn (*)      ! input buffer
integer*4 :: InfoOut (*)     ! output buffer
integer*4 :: M (*)           ! whole data space
character*10 xxx
xxx = PRO_TypeCharIdArray( model)
select case (PRO_TypeCharIdArray(model))
```

```
    case (' USER1')
        call USR1_Model (iorder,Props,IGPbuff,InfoIn,InfoOut,M)
case default
    stop 'USER_Models unknown'
end select
end subroutine SUM_Models
```

To activate a new model a new "case" statement must be added to the selection list. Any statement like $!MS$ is a compiler directive.

2.5.3 Programming new constitutive model

To program a new constitutive model user should start from the delivered template file usr1.for which contains an example of a Huber-Mises elasto-plastic model with strength parameter being dependent on temperature. Any line with comment DON'T EDIT should be left as it is. The detailed description of what the code should return to the element level and through which data channels, has been described in the previous sections. Hence the easiest way to program a new model is to copy usr1.for file onto new one, add it to the project (usermodels.dll).

2.5.4 Flow dependency on temperature

In this version the permeability coefficients are dependent on temperature and the source term related to fluid thermal dilatancy is added. This is included in the kOfT.for module. In the delivered code we assume that k = ko * f(T(t)) where ko is the one introduced as standard permeability parameter in Flow parameters group. Another possibility where f(T(t)) is governed by some constitutive function is allowed, but user must introduce necessary parameters via script file and to prepare a code within body of FSS_MakePearmTempDependent subroutine (see kofT.for module).

2.5.5 *Exported functions*

2.5.5.1 Functions to communicate with system

```
c==================================
integer*4 function IANA_IntInfo ( idata )
c==================================
```

- to use this function always include '..\\anal_enu.inc'
- idata is of type : integer*4
- idata can be one of the following

IANA_ITER_NR : for this enumerator the function returns actual iteration number
(at the global level)

```
c==================================
real*8 function ANA_RealInfo ( idata )
c==================================
```

- to use this function always include '..\\anal_enu.inc'
- idata is of type : integer*4
- idata can be one of the following

IANA_SAFETY_ACT : function will return actual global safety factor during stability analysis

IANA_SAFETY_BEG : function will return initial global safety factor during stability analysis

IANA_SAFETY_END : function will return final global safety factor during stability analysis

IANA_GRAV_ACT : function will return actual gravity multiplier for the initial state analysis

IANA_GRAV_END : function will return final gravity multiplier for the initial state analysis

IANA_GRAV_INC : function will return increment of gravity multiplier for the initial state analysis

IANA_DTIME_ACT : function will return actual time increment for time dependent drivers

IANA_DTIME_BEG : function will return initial time increment for time dependent drivers

IANA_TIME_FINAL : function will return final time value for time dependent drivers

IANA_DT_MULT : function will return time increment multiplier

IANA_TIME_ACT : function will return actual time

IANA_TIME_BEG : function will return initial time for time dependent drivers

2.5.5.2 Constitutive functions

```
c=====================================
subroutine ELA_EvaluateDmatr ( E,v,De,nstre )
c=====================================
real*8 :: E ! young modulus (IN)
real*8 :: v ! Poisson ratio (IN)
integer*4 :: nstre (IN) ! (equals to 3 for plane stress, 4 for plane strain and 6 for 3D)
real*8 :: De(nstre,nstre) ! elastic stiffness matrix (OUT)
c=====================================
subroutine ELA_EvaluateCmatr ( E,v,Ce,nstre )
c=====================================
real*8 :: E ! young modulus (IN)
real*8 :: v ! Poisson ratio (IN)
integer*4 :: nstre (IN) ! (equals to 3 for plane stress, 4 for plane strain and 6 for 3D)
real*8 :: De(nstre,nstre) ! elastic compliance matrix (OUT)
c=====================================
real*8 function get_I1 ( stress ,nstre)
c=====================================
integer*4 :: nstre (IN) ! (equals to 3 for plane stress, 4 for plane strain and 6 for 3D)
real*8 :: stress (nstre) (IN)
! returns first stress invariant I1

c=====================================
real*8 function get_p ( stress ,nstre)
c=====================================
integer*4 :: nstre (IN) ! (equals to 3 for plane stress, 4 for plane strain and 6 for 3D)
real*8 :: stress (nstre) (IN)
! returns p ( p= -I1/3)
c=====================================
real*8 function get_J2 ( stress,s,nstre)
c=====================================
integer*4 :: nstre (IN) ! (equals to 3 for plane stress, 4 for plane strain and 6 for 3D)
real*8 :: stress (nstre) (IN)
real*8 :: s (nstre) (OUT)
! returns J2 invariant and stress deviator s(nstre)
```

```
c======================================
real*8 function get_q ( stress,nstre)
c======================================
integer*4 :: nstre (IN) ! (equals to 3 for plane stress, 4 for plane strain and 6 for 3D)
real*8 :: stress (nstre) (IN)
! returns q invariant (q=sqrt(3*J2))
c======================================
real*8 function get_epsd ( strain,nstre)
c======================================
integer*4 :: nstre (IN) ! (equals to 3 for plane stress, 4 for plane strain and 6 for 3D)
real*8 :: strain (nstre) (IN) ! with gamma's for shear components
! returns sqrt(2/3 eij*eij) where eij is strain deviator
c======================================
real*8 function get_J3 ( stress,nstre)
c======================================
integer*4 :: nstre (IN) ! (equals to 3 for plane stress, 4 for plane strain and 6 for 3D)
real*8 :: stress (nstre) (IN)
! returns 3-rd stress invariant
c======================================
subroutine get_dI1dSig ( dI1dSig,nstre )
c======================================
integer*4 :: nstre (IN) ! (equals to 3 for plane stress, 4 for plane strain and 6 for 3D)
real*8 :: dI1dSig (nstre) (OUT)
! returns dI1/dSig derivative
c======================================
subroutine get_dJ2dSig ( dJ2dSig,s,nstre )
c======================================
integer*4 :: nstre (IN) ! (equals to 3 for plane stress, 4 for plane strain and 6 for 3D)
real*8 :: S(nstre) ! (IN) stress deviator
real*8 :: dJ2dSig (nstre) (OUT)
! returns dJ2/dSig derivative
c======================================
subroutine get_dJ3dSig ( dJ3_dSig,S,xJ2,nstre )
c======================================
integer*4 :: nstre (IN) ! (equals to 3 for plane stress, 4 for plane strain and 6 for 3D)
real*8 :: S(nstre) ! (IN) stress deviator
real*8 :: xJ2 ! (IN) J2 stress invariant
real*8 :: dJ3_dSig (nstre) (OUT)
! returns dJ3/dSig derivative
c======================================
subroutine get_dJ2dSig2 ( xM,nstre )
c======================================
integer*4 :: nstre (IN) ! (equals to 3 for plane stress, 4 for plane strain and 6 for 3D)
real*8 :: xM(nstre,nstre) ! (OUT)
!            2  2
! returns d J2  / dSig derivative


c======================================
subroutine GPL_StressInvariants (stress,sdev,nstre,xI1,xJ2,xJ3)
c======================================
integer*4 :: nstre (IN) ! (equals to 3 for plane stress, 4 for plane strain and 6 for 3D)
real*8 :: sDev (nstre) ! (OUT) stress deviator
real*8 :: xI1 ! (OUT) first stress invariant I1
real*8 :: xJ2 ! (OUT) second stress invariant J2
real*8 :: xJ3 ! (OUT) thrid stress invariant J3
c======================================
subroutine GPL_HaighWestergaard(xI1,xJ2,xJ3,Xsi,Ro,Theta)
c======================================
c converts I1,J2,J3 invariants to Haigh-Westergaard invariants xsi,ro, theta
integer*4 :: nstre (IN) ! (equals to 3 for plane stress, 4 for plane strain and 6 for 3D)
real*8 :: xI1 ! (IN) first stress invariant I1
real*8 :: xJ2 ! (IN) second stress invariant J2
real*8 :: xJ3 ! (IN) thrid stress invariant J3
real*8 :: xsi ! (OUT)
real*8 :: ro ! (OUT)
real*8 :: theta ! (OUT)
c======================================
subroutine Set_DepH ( De,Dep,a,b,H,nstre )
c======================================
```

integer*4 :: nstre (IN) ! (equals to 3 for plane stress, 4 for plane strain and 6 for 3D)

real*8 :: De (nstre,nstre) ! (IN) elastic constitutive matrix

real*8 :: a (nstre) ! (IN) dF/dSig gradient

real*8 :: b (nstre) ! (IN) dQ/dSig gradient

real*8 :: H ! (IN) hardening modulus

real*8 :: Dep (nstre,nstre) ! (OUT) elastic constitutive matrix

2.5.5.3 Handling load time functions

c==================================

real *8 function RLTF_Value(Time ,Iltf , M)

c==================================

integer*4 :: iltf ! (IN) load time function index

real*8 :: Time ! (IN) time value

integer*4 :: M(*) ! (IN) data space

! returns value of this ooad time function for given time

2.5.5.4 Mathematical utilities

c==================================

real *8 function RMAT_InvMatr123Ex (a,a_inv,n,detMIN)

c==================================

integer*4 :: n ! (IN) matrix size (up to 3)

real*8 :: a(n,n) !(IN) real matrix

real*8 :: a_inv(n,n) !(OUT) output inverse matrix

real*8 :: detMIN ! (IN) minimum determinat value under which inversion is not done

c==================================

real *8 function RMAT_InvMatr123 (a,a_inv,n)

c==================================

integer*4 :: n ! (IN) matrix size (up to 3)

real*8 :: a(n,n) !(IN) real matrix

real*8 :: a_inv(n,n) !(OUT) output inverse matrix

! returns determinant value

c==================================

logical*4 function MAT_Solve123 (a,n,rhs,sol)

```
c=====================================
integer*4 :: n ! (IN) matrix size ( up to 3)
real*8 :: a(n,n) !(IN) real matrix
real*8 :: rhs(n) !(IN) rhs vector
real*8 :: sol(n) !(OUT) solution of a * sol = rhs
! returns TRUE if everything ok and FALSE if matrix is singular

c=====================================
subroutine mat_vecnorm ( a,n )
c=====================================
integer*4 :: n ! (IN) vector size
real*8 :: a(n) !(IN) real vector
! a is normalized
c=====================================
subroutine MAT_axb ( a,m,n,b,l,c )
c=====================================
integer*4 :: n,m,l ! (IN)
real*8 :: a(m,n),b(n,l) !(IN ) real matrices
real*8 :: c(m,l) !(OUT) real matrix
! procedure evaluates c [m,l] = a [m,n] * b [n,l]

c=====================================
subroutine MAT_aTxb ( a,m,n,b,l,c )
c=====================================
integer*4 :: n,m,l ! (IN)
real*8 :: a(n,m),b(n,l) !(IN ) real matrices
real*8 :: c(m,l) !(OUT) real matrix
!                                              T
! procedure evaluates c [m,l] = a [n,m] * b [n,l]
c=====================================
subroutine MAT_axbT ( a,m,n,b,l,c )
c=====================================
integer*4 :: n,m,l ! (IN)
real*8 :: a(m,n),b(l,n) !(IN ) real matrices
real*8 :: c(m,l) !(OUT) real matrix T
! procedure computes c [m,l] = a [m,n] * b [l,n]
```

```
c=====================================
subroutine MAT_addvec ( a,b,c,n )
c=====================================
integer*4 :: n ! (IN)
real*8 :: a(n),b(n) !(IN ) real vectors
real*8 :: c(n) !(OUT) real vector
! procedure computes c = a + b

c=====================================
subroutine MAT_subtrvec ( a,b,c,n )
c=====================================
integer*4 :: n ! (IN)
real*8 :: a(n),b(n) !(IN ) real vectors
real*8 :: c(n) !(OUT) real vector
! procedure computes c = a - b
c=====================================
subroutine MAT_symm_mtrx ( a,n )
c=====================================
integer*4 :: n ! (IN) matrix size
real*8 :; a(n,n) ! (IN/OUT) matrix
! symmmetrization of the square matrix with assumption that the upper
! triangle is defined

c=====================================
subroutine MAT_symm_mtrxL ( a,n )
c=====================================
integer*4 :: n ! (IN) matrix size
real*8 :; a(n,n) ! (IN/OUT) matrix
! symmmetrization of the square matrix with assumption that the lower
! triangle is defined

c=====================================
subroutine MAT_axfactor ( a,n,factr,b )
c=====================================
integer*4 :: n ! (IN) vector size
real*8 :: a(n) ! (IN) vector
```

```
real*8 :: b(n) ! (OUT) vector
real*8 :: factor
! b = a * factor
! it is possible to call this routine like : call MAT_axfactor ( a,3,1.5d0,a )
c=====================================
real*8 function RMAT_dot (a,b,n)
c=====================================
integer*4 :: n ! (IN) matrix size
real*8 :: a(n) ! (IN) vector
real*8 :: b(n) ! (IN) vector
! returns scalar product a * b

c=====================================
subroutine MAT_Transpose ( a,m,n,at )
c=====================================
integer*4 :: n ! (IN) matrix size
real*8 :: a(m,n) ! (IN) matrix
real*8 :: at(n,m) ! (OUT) transposed matrix
c=====================================
subroutine MAT_IntTranspose ( ia,m,n,iat )
c=====================================
integer*4 :: n ! (IN) matrix size
integer*4 :: ia(m,n) ! (IN) matrix
integer*4 :: iat(n,m) ! (OUT) transposed matrix
c=====================================
subroutine MAT_Invert ( A,N,InvErr )
c=====================================
integer*4 :: N ! ( N <= 16 !!!! )
real*8 :: A(N,N) ! (IN/OUT) matrix given/inverted
logical*4 :: InvErr ! (OUT) inversion error flag (TRUE means failure)
c=====================================
integer*4 function MAT_GaussElim ( A,Max,N,B )
c=====================================
integer*4 :: Max,N
real*8 :: A(Max,Max), B(N)
```

```
! solves A * x = B
! B is replaced by solution x
! in case of success returns 0
! else returns 1
c====================================
subroutine mat_det ( a,n,det )
c====================================
integer*4 :: n ! (IN) n is up to 3 !!!
real*8 :: a(n,n) ! (IN)
real*8 :: det ! (OUT) returned determinant
c====================================
subroutine MAT_AddMultVec(X,factor,Y,N)
c====================================
integer*4 :: N ! (IN) vector size
real*8 :: X(N) ! (IN/OUT) vector
real*8 :: Y(N) ! (IN) vector
real*8 :: factor
! evaluates X = X + factor * Y

c====================================
subroutine MAT_NeqMultVec(X,factor,Y,N)
c====================================
integer*4 :: N ! (IN) vector size
real*8 :: X(N) ! (IN/OUT) vector
real*8 :: Y(N) ! (IN) vector
real*8 :: factor
! evaluates X = -X + factor * Y
c====================================
subroutine MAT_VecProd(a,b,c)
c====================================
real*8 :: a(3),b(3) ! (IN) vectors
real*8 :: c(3) ! (OUT) vector product c = a x b
c====================================
subroutine MAT_SetDiagMatrix ( A,n,value )
c====================================
real*8 :: A(n,n)
real*8 :: value
! sets a diagonal matrix with diagonal elements of value ->value
c====================================
subroutine MAT_ChangeSign ( A,n )
c====================================
real*8 :: A(n) ! (IN/OUT) vector
! changes sign of all elements of vector A
c====================================
integer*4 function MAT_Crout ( A,Max,N,Rhs )
c====================================
integer*4 :; Max,N ! (IN)
real*8 :: A(max,MaX) ! (IN) matrix
real*8 :: Rhs (N) ! (IN/OUT) vector
! solves A x b = Rhs using Crout decomposition
! solution is returned in array Rhs
! if matrix is nonsingular function returns 0
! else returns 1
c====================================
subroutine MAT_AtxBxCbuff(A,nRowA,nColA,B,nColB,C,nColC,AtBC,buff)
c====================================
integer*4 :: nRowA,nColA,nColB,nColC ! (IN)
real*8 :: A(nRowA,nColA) ! (IN) matrix
real*8 :: B(nRowA,nColB) ! (IN) matrix
real*8 :: C(nColB,nColC) ! (IN) matrix
real*8 :: AtBC(nColA,nColC) ! (OUT) matrix
real*8 :: buff (nRowA) ! (IN) work array
! T
! evaluates A B C
```

2.5.5.5 Functions to be used for addressing of subarrays in single integer*4 work space

```
c====================================
subroutine MEM_StartPos ( ipos )
```

c===================================

integer*4 :: ipos ! (IN) ! start position (usually 1)

c===================================

integer*4 function MEM_NextPos (size,iprec)

c===================================

integer*4 :: size ! (IN) subarray size

integer*4 :: iprec ! (IN) int*4 words per single subarray item (1 for int*4 arrays and 2 for real*8 arrays)

2.5.5.6 Functions for material data handling

c===================================

real *8 function PRO_GetFromDat(Props,igroup,item)

c===================================

integer*4 :: igroup ! (IN) group enumerator from include file 'prop.inc'

integer*4 :: item p ! (IN) parameter index

real*8 :: props (*) ! (IN) parameters array

! returns requested parameter

! igroup can be one of the following:

IPRO_GR_MAIN = 1,

IPRO_GR_DENS = 2, ! density group

IPRO_GR_ELAS = 3, ! elastic parameters

IPRO_GR_GEOM = 4, ! geometry

IPRO_GR_FLOW = 5, ! flow

IPRO_GR_CREEP = 6, ! creep

IPRO_GR_NONL = 7, ! nonlinear parameters

IPRO_GR_HEAT = 8, ! heat

IPRO_GR_HUMI = 9, ! humidity

IPRO_GR_INIS =10, ! initial state Ko

IPRO_GR_STAB =11, ! local stability

! item is just an index, for certain parameter groups these indices are enumerated and

! kept in prop.inc include file

like for flow parameters: (see prop.inc file)

parameter(

IPRO_FLOW_KX = 1,

IPRO_FLOW_KY = 2,

```
IPRO_FLOW_KZ = 3,
IPRO_FLOW_THETA = 4,
IPRO_FLOW_ANISO_X = 5,
IPRO_FLOW_ANISO_Y = 6,
IPRO_FLOW_ANISO_Z = 7,
IPRO_FLOW_BULK_MOD = 8,
IPRO_FLOW_Sr = 9,
IPRO_FLOW_ALPHA =10,
IPRO_FLOW_SKIP_GRAV =11,
IPRO_FLOW_UNDR_FLAG =12,
IPRO_FLOW_UNDR_EPS =13,
IPRO_FLOW_UNDR_PLIM =14
c=====================================
real *8 function PRO_GetFromAdd(Props,igroup,item)
c=====================================
integer*4 :: igroup ! (IN) group enumerator from include file 'prop.inc'
integer*4 :: item p ! (IN) parameter index
real*8 :: props (*) ! (IN) parameters array
! returns requested parameter from additional material model storage
c=====================================
subroutine PRO_PutToDat( Props,igroup, item, Value )
c=====================================
integer*4 :: igroup ! (IN) group enumerator from include file 'prop.inc'
integer*4 :: item p ! (IN) parameter index
real*8 :: props (*) ! (IN) parameters array
! sets certain parameter value ->Value
c=====================================
subroutine PRO_PutToAdd ( Props, igroup, item, Value )
c=====================================
integer*4 :: igroup ! (IN) group enumerator from include file 'prop.inc'
integer*4 :: item p ! (IN) parameter index
real*8 :: props (*) ! (IN) parameters array
! sets certain parameter value ->Value in additional storage
c=====================================
real *8 function PRO_GetParamNonl(props,item)
c====================================
integer*4 :: item p ! (IN) parameter index
real*8 :: props (*) ! (IN) parameters array
! returns parameter from group NONLINEAR for given item
c====================================
subroutine PRO_PutParamNonl(props,item,Value)
c====================================
integer*4 :: item p ! (IN) parameter index
real*8 :: props (*) ! (IN) parameters array
! sets parameter from group NONLINEAR and for given item ->value
c====================================
subroutine PRO_ModifAngleInNonl(props,item,Coeff)
c====================================
integer*4 :: item ! (IN) parameter index
real*8 :: Coeff ! (IN) given multiplier
real*8 :: props (*) ! (IN) parameters array
! modifies angle parameter (expressed in deg (!!!)) from group NONLINEAR
! parameter is kept in group NONLINEAR in -item- position
c====================================
subroutine PRO_ModifValueInNonl(props,item,Coeff)
c====================================
integer*4 :: item ! (IN) parameter index
real*8 :: Coeff ! (IN) given multiplier
real*8 :: props (*) ! (IN) parameters array
! modifies any parameter from group NONLINEAR (multiplies by Coeff)
! parameter is kept in group NONLINEAR in -item- position
```

2.5.5.7 Other utilities

```
c====================================
subroutine UTL_iclear(iarray,n)
c====================================
integer*4 :: n ! (IN) vector size
integer*4 :: iarray (n) ! (IN/OUT) vector
```

```
! clears an int*4 array

c====================================
subroutine UTL_clear(array,n)
c====================================
integer*4 :: n ! (IN) vector size
real*8 :: array (n) ! (IN/OUT) vector
! clears a real*8 array
c====================================
subroutine UTL_imove ( ia,ib,n )
c====================================
integer*4 :: n ! (IN) vector size
integer*4 :: ia(n) ! (OUT) vector
integer*4 :: ib(n) ! (IN ) vector
! copies ib ->ia () ( ia = ib )
c====================================
subroutine UTL_move ( a,b,n )
c====================================
integer*4 :: n ! (IN) vector size
integer*4 :: a(n) ! (OUT) vector
integer*4 :: b(n) ! (IN ) vector
! copies b ->a () ( a = b )
c====================================
subroutine UTL_PrincStress (nsd,s,p)
c====================================
integer*4 :: nsd ! (IN) space size (2 or 3)
real*8 :: s(n) ! (IN) stress vector ordered (xx,yy,xy,zz,xz,yz)
real*8 :: p(nsd) (OUT) ! principal values
! copies b ->a () ( a = b )
```

2.6 Sample data

In directory INP you have 3 examples in which user defined model is used:

- cutT.inp (heat transfer problem)
- cutLMS1.inp (transient flow problem based on temperature field generated in project cutT.inp)
- cutLMS2.inp (slope stability problem based on temperature field generated in project cutT.inp)

Chapter 3

User programmed output in ZSoil®postprocessor

3.1 Introduction

The ZSoil®2023 system allows the user to program his own output (in C++) and to plug it into the postprocessor. This option is available to every user who has a legal license of ZSoil®2023 (professional or academic).

The introduction of user programmed output consists of the two major steps:

1. creating necessary configuration files
2. programming of a user supplied C++ classes and then compiling them, and linking, to produce useroutput.dll file

3.2 Creating necessary configuration files

The first configuration file (prepared in xml format) must be labeled as user_output.xml. Structure of this file is as follows

```
<?xml version="1.0" encoding="ISO-8859-15"?>
<USER_OUTPUT>
<ELEMENT_CLASS label="SHELL">
<RESULT label="(User):As1-XX"></RESULT>
<RESULT label="(User):As1-YY"></RESULT>
<RESULT label="(User):As2-XX"></RESULT>
<RESULT label="(User):As2-YY"></RESULT>
</ELEMENT_CLASS>
<ELEMENT_CLASS label="CONTINUUM">
<RESULT label="(User):S*p"></RESULT>
</ELEMENT_CLASS>
<ELEMENT_CLASS label="BEAM">
<RESULT label="(User):As1"></RESULT>
<RESULT label="(User):As1-min"></RESULT>
<RESULT label="(User):As2-min"></RESULT>
```

```
</ELEMENT_CLASS>
</USER_OUTPUT>
```

As we can see the first line is just the header of the xml file. The word <USER_OUTPUT> is the main keyword that opens the definition of the extra output, while </USER_OUTPUT> closes it. Between these two keywords we can identify three output definitions for three classes of elements (only these classes are supported for the time being) ie. shells, continuum and beams. As an example a simple code is delivered that computes longitudinal reinforcement (based on Wood-Armer bending moments) in shells (without taking into account membrane efforts) and beams (without axial force either). For continuum as an example simple class that computes $S\ p$ nominal pore water pressure is delivered. In this xml file result labels can only be edited. Moreover it is important to remember in which order these results are declared, as later on, in the C++ code, we will be refering to them through enumerators 0, 1...etc... Maximum 20 extra results for each element class can be handled by ZSoil®.

The second file that can be created by the user may include extra data like concrete properties taken from the EC2 standard, steel properties etc... Content of the file (EC2design.csv) used in the delivered code is as follows (this is *.csv format that can easily be edited with aid of excel)

```
Concrete class C20/25;;
Ecm;30000;MPa
fck;20;MPa
fctm;2.2;MPa
fctk;1.5;MPa
epsilon_c3;0.00175;
epsilon_cu3;0.0035;
gamma_c;1.4;
eta;1;
lambda;0.8;
rho_min;0.0013;
Steel RB500;;
fyk;500;MPa
gamma_s;1.15;
Reinforcement position;;
a1;0.035;m
a2;0.035;m
```

3.3 Programming user supplied C++ code to handle extra output

The following C++ functions are delivered to start further developments

1. ExtraDataUserManipulation.cpp, ExtraDataUserManipulation.h
   This function allows the user to read the EC2design.csv configuration file and to serve this data if needed
2. GPRslBeamUser.cpp, GPRslBeamUser.h

This function allows the user to produce output for beam elements

3. GPRslContinuumUSER.cpp, GPRslContinuumUser.h This function allows the user to produce output for continuum elements

4. GPRslShellUSER.cpp, GPRslShellUser.h This function allows the user to produce output for shell elements

5. QuadReinforcement.cpp, QuadReinforcement.h
   This class is used to dimension reinforced concrete quadrilateral cross section

3.3.1 Example of programing user output for beam elements

In order to achieve full functionality of the user programmed output 3 basic functions in GPRslBeamUser.cpp must be defined i.e.

1. function give(...) that returns the result in form of double real value

2. function filter (..) that checks sign (or other verificaton) of the produced result (for instance reinforcement area must be positive)

3. function get_unit (..) that returns unit label for produced result (see header #include "..\..\zutl\zunits.h" for predefined unit labels)

Let us now analyze body of functions in GPRslBeamUser.cpp that produce extra results for beam elements.

```
#include "stdafx.h"
#include "QuadReinforcement.h"
#include "../GPRslBeamUser.h"
#include "../UserExtraData.h"
#include "../../zmate\CrossSection.h"
#include "../../zutl/zunits.h"



// ================================================================================
   double GPRslBeamUser_give (int        user_item,//USER_1=0,USER_2.....
                                         CString & item_string,
                                         int        user_material_number,
                                         UserExtraData* usr_data,
                                         double     *F, //NX,QY,QZ --> note that it can be NULL !
                                         double     *M, //MX,MY,MZ --> note that it can be NULL !
                                         std::vector <GPRslBeamLayersUser> & layers_core_info ,
                                         std::vector <GPRslBeamLayersUser> & layers_reinf_info ,
                                         int        *ret,
                                         CrossSection *sect)   //return 1 if ok
// ================================================================================
{
   double h = 0.0;
   double b = 0.0;

   switch (sect->type())
     {
     case CrossSection::CROSS_SECTION_RECT :
       {
       CrossSection_RECT *sectRect = (CrossSection_RECT*)sect;
       b = sectRect->b;
       h = sectRect->h;

       switch(user_item)
         {
         case 0://reinforcement XX-bottom
```

```
           {
           double MEd = M [2];
           *ret = 1;
           if ( MEd < 0.0  )
             return 0.0;
           else
             return QuadReinforcement::Quad1 (usr_data,b,h,MEd,QuadReinforcement::BOTTOM);
           }
           break;
         case 1://
           *ret = 1;
           return 0.0;
         case 2://
           *ret = 1;
           return 0.0;
         default:
           *ret = 0;
           return 0.0;
           break;
         }
       }
     break;

     case CrossSection::CROSS_SECTION_UNDEFINED :
     case CrossSection::CROSS_SECTION_RECT_TUBE :
     case CrossSection::CROSS_SECTION_CIRCLE :
     case CrossSection::CROSS_SECTION_CIRC_TUBE :
     case CrossSection::CROSS_SECTION_RECT_BOX :
     case CrossSection::CROSS_SECTION_I_SHAPE_S :
     case CrossSection::CROSS_SECTION_I_SHAPE_NS :
     case CrossSection::CROSS_SECTION_T_SHAPE :
     case CrossSection::CROSS_SECTION_RECT_AXI :
     default :
      *ret = 0;
       return 0.0;
     break;
     }

}


// ================================================================================
   double GPRslBeamUser_filter (int        user_item,//USER_1=0,USER_2.....
                                  CString &  item_string,
                                  double     rsl)
// ================================================================================
// this function filters user result after extrapolating it to the nodes (via superconvergent patch rec
   {
   switch (user_item)
     {
     case 0:
     case 1:
     case 2:
     case 3:
       return max(rsl,0.0); //reinforcement must be positive
     default:
       return rsl;
     }
   }

// ================================================================================
   int GPRslBeamUser_get_unit (int user_item,CString &  item_string,UNIT_manager* umngr)
// ================================================================================
   {
   switch(user_item)
     {
     case 0://reinforcement XX-bottom
       {
       return UNIT_AREA;
       }
       break;
     case 1://reinforcement YY-bottom
       {
       return UNIT_AREA;
      }
      break;
    case 2://reinforcement XX-top
      {
      return UNIT_AREA;
      }
      break;
    case 3://reinforcement YY-top
      {
      return UNIT_AREA;
      }
      break;
    default:
      return UNIT_DIMLESS;
      break;
    }
  }
```

As we can see user receives complete information for beam integration point that includes internal forces, but also (for layered beams) stresses, strains and other information for core material and all reinforcement layers (at a given integration point).

3.3.2 Example of programing user output for shell/continuum elements

Same scheme, as the one described for beams, is used for shell and for continuum elements.