Z-set development training

Transvalor | Armines | Onera

www.zset-software.com

Development in Z-set

- Generalities
  - C++ basics
  - Object factory
  - Plugins
- Mesher
- Post-processing
- Behaviors
  - material objects
  - ZebFront behaviors
  - finite strain
- Boundary conditions
- Z-program
- Elements
- Algorithms

C++ basics

- classes, derived classes
- member data, member functions
- overloading of member functions
- constructors, destructors
- polymorphism

Example: Surface calculation of simple geometric objects

Definition of user types: file Geometric_object.h (1/2)

```
class GEOMETRIC_OBJECT {                          // base class for geometric objects
  public :                                        // member functions
    GEOMETRIC_OBJECT() { }                        // constructor (defined inline)
    ~GEOMETRIC_OBJECT() { }                       // destructor
    virtual bool set_parameter(char* p,double val);  // virtual function: can be redefined
                                                     // (overloaded) in derived classes
    virtual double surface()=0;                   // pure virtual function: must be redefined
                                                  // in derived classes
};

class POLYGON : public GEOMETRIC_OBJECT {         // base class for polygon objects: derived from
                                                  // GEOMETRIC_OBJECT
  protected :                                     // can only be accessed by the class and its
                                                  // derived clases
    double width;                                 // member data inherited by derived classes
    double thickness;
  public :
    POLYGON();
   ~POLYGON();
    bool set_parameter(char* p,double val);
    virtual double surface()=0;                   // pure virtual function: must be defined in
                                                  // derived classes
};
```

Definition of user types: file Geometric_object.h (2/2)

```
class TRIANGLE : public POLYGON {           // class derived from POLYGON:
  public :                                  // inherits member data thickness and width
    double surface();
};

class RECTANGLE : public POLYGON {
  public :
    double surface();
};
```

Function implementation: file Geometric_object.c

```
#include <iostream.h>
#include <Geometric_object.h>

bool GEOMETRIC_OBJECT::set_parameter(char* par,double val)
{ cout<<"Unknown parameter: "<<par<<endl;
  return 0;
}

POLYGON::POLYGON()
{ width=thickness=1.; }

POLYGON::~POLYGON() { }

bool POLYGON::set_parameter(char* par,double val)
{ if(strcmp(par,"width")==0) width=val;
  else if(strcmp(par,"thickness")==0) thickness=val;
  else return GEOMETRIC_OBJECT::set_parameter(par,val);
  return 1;
}

double TRIANGLE::surface()
{ return (1./2.)*width*thickness; }

double RECTANGLE::surface()
{ return width*thickness; }
```

Main function: file Main.c

```
#include <iostream.h>
#include <string.h>
#include <Geometric_object.h>

int main()
{ char* str= new char[10];
  GEOMETRIC_OBJECT* geom;
  cout<<"What type? ";
  cin>>str;
  if(strcmp(str,"rectangle")==0)     geom = new RECTANGLE();
  else if(strcmp(str,"triangle")==0) geom = new TRIANGLE();
  else { cout<<"Unknown type: "<<str<<endl; exit(1); }

  double val;
  for(;;) {
    cout<<"Parameter name? ";
    cin>>str;
    if(strcmp(str,"end")==0) break;
    cout<<"Value? ";
    cin>>val;
    bool ok=geom->set_parameter(str,val);
    if(!ok) exit(1);
  }
  cout<<"Surface = "<<geom->surface()<<endl;
}
```

Compilation/Execution

- reset
- open terminal
- edit makefile
- compilation/link
- run

Add calculation for a circle

- define a new class deriving from GEOMETRIC_OBJECT in file Circle.h
  - Circle.h
  - do_it_for_me
- implement surface calculation in Circle.c
  - Circle.c
  - do_it_for_me
- modify Makefile
  - Makefile
  - do_it_for_me
- modify Main.c
  - Main.c
  - do_it_for_me

- reset
- Geometric_object.h
- compil
- open terminal
- Geometric_object.c
- run

Limitations of the C++ basic approach

Some polymorphism, but it is necessary to modify the main code each time a new geometric object is added:

```
#include <iostream.h>
#include <string.h>
#include <Geometric_object.h>
#include <Circle.h>

int main()
{
  char* str= new char[10];
  GEOMETRIC_OBJECT* geom;

  cout<<"What type? ";
  cin>>str;
  if(strcmp(str,"rectangle")==0)      geom = new RECTANGLE();
  else if(strcmp(str,"triangle")==0) geom = new TRIANGLE();
  else if(strcmp(str,"circle")==0)    geom = new CIRCLE();
  else {
    cout<<"Unknown type: "<<str<<endl; exit(1);
  }
  ...
  cout<<"Surface = "<<geom->surface()<<endl;
}
```

Object factory

Management of Z-set development projects

∗ reset project ∗ open terminal

- Configuration of the project: **Zsetup**
  - takes as input a "library_files" describing the project
  - generates "Makefile.dat" with architecture-independent commands

```
!DYNAMIC
!LIB_BASED
!USE_INC                                          # use standard includes $Z7PATH/include
!BFLAGS -L${Z7PATH}/PUBLIC/lib-${Z7MACHINE}       # path of standard Zebulon libraries
!INC src                                          # declares header files in src/
!SRC src src                                      # declares source files in src/
!DEBUG src
# Generates a program named Surf_$Z7MACHINE from source files in src/
# Links with standard libraries Zmat_base
!TARGET Surf src Zmat_base
!!RETURN
```

- Compilation/Link: **Zmake**
  - generates a real "makefile" for the architecture from "Makefile.dat"
  - compilation and link of the project

Add derived classes in the object factory

∗ Geometric_object.h ∗ Geometric_object.c

```
#include <Object_factory.h>
#include <Stringpp.h>
#include <Error_messager.h>
#include <Geometric_object.h>

bool GEOMETRIC_OBJECT::set_parameter(const STRING& par,double val)
{ ERROR("Unknown parameter: "+par);
  return 0;
}

POLYGON::POLYGON() { width=thickness=1.; }
POLYGON::~POLYGON() { }
bool POLYGON::set_parameter(const STRING& par,double val)
{ if(par=="width") width=val;
  else if(par=="thickness") thickness=val;
  else return GEOMETRIC_OBJECT::set_parameter(par,val);
  return 1;
}

DECLARE_OBJECT(GEOMETRIC_OBJECT,TRIANGLE,triangle)
double TRIANGLE::surface() { return (1./2.)*width*thickness; }
DECLARE_OBJECT(GEOMETRIC_OBJECT,RECTANGLE,rectangle)
double RECTANGLE::surface() { return width*thickness; }
```

Use of object factory to create objects

```
#include <Ziostream.h>
#include <Object_factory.h>
#include <Stringpp.h>
#include <Error_messager.h>
#include <Geometric_object.h>

int main()
{ STRING str;                                  // Zebulon STRING object: Stringpp.h
  GEOMETRIC_OBJECT* geom=NULL;
  Out<<"What type? "; Out.flush();             // Use Out,In instead of standard C++
  In>>str;                                     // cout, cin : Ziostream.h
  geom = Create_object(GEOMETRIC_OBJECT,str);
  if(!geom) ZERROR("Unknown type: "+str);      // Zebulon ERROR macro: Error_messager.h

  double val;
  for(;;) {
    Out<<"Parameter name? "; Out.flush();
    In>>str;
    if(str=="end") break;
    Out<<"Value? "; Out.flush();
    In>>val;
    geom->set_parameter(str,val);
  }
  Out<<"Surface = "<<geom->surface()<<endl;
}
```

Add calculation for a circle using OF

- reset project
- open terminal
- Zmake
- run

Add declaration of class CIRCLE in Object factory:
(note Circle.h is not necessary anymore)

- modify Circle.c
- do_it_for_me

Plugins

- add new depevelopments in shared (dynamic) libraries
- use the standard Zébulon executable
- objects defined in the library and installed in OF are loaded dynamically into Zébulon at run time

Plugin example

- Define a new class deriving from **BASE_PROBLEM** (Base_problem.h) to perform surface calculations

```
ZCLASS BASE_PROBLEM : public Z_OBJECT {
...
   public :
...
     virtual bool Execute() { NOT_IMPLEMENTED_ERROR("Base Execute"); return FALSE; }
     virtual void load(const STRING&, const STRING&);
..
};
```

- Overload virtual functions Execute() and load() in the new class
- Use an input file to declare **GEOMETRIC_OBJECT** types and parameters
- Enroll the new class in OF
- Generate a Plugin from the new source code

Plugin example

```
#include <Object_factory.h>
#include <Stringpp.h>
#include <List.h>
#include <Error_messager.h>
#include <Base_problem.h>
#include <Geometric_object.h>

class SURFACE : public BASE_PROBLEM
{
  private :
    PLIST<GEOMETRIC_OBJECT> to_calc;              // list of PTR type pointers:
  public :                                        // defined in Pointer.h
    SURFACE() {}
    ~SURFACE() {}
    bool Execute(void);                          // overloading BASE_PROBLEM methods
    void load(const STRING&,const STRING&);
};

DECLARE_OBJECT(BASE_PROBLEM,SURFACE,surface)     // enroll class SURFACE in OF
ADD_PB_TYPE(surf,surface)                        // add a problem switch -surf
```

Plugin example (input file)

```
void SURFACE::load(const STRING &inp, const STRING &o)
{ STRING fname=inp+".inp"; ASCII_FILE input(fname());
  input.locate_at_level(4,"surface");
  if(!input.ok) ERROR("Cannot find ****surface in file:"+fname);
  STRING str; GEOMETRIC_OBJECT* geom;
  for(;;) {
    str=input.getSTRING(); if(str=="****return") break;
    if(str.start_with("*")) {
      str.remove_all('*');
      geom=Create_object(GEOMETRIC_OBJECT,str);
      if(geom) to_calc.add(geom);
      else INPUT_ERROR("Cannot create a geometric object of type: "+str);
      for(;;) {
        str=input.getSTRING();
        if(str.start_with("*")) { input.back(); break; }
        STRING val=input.getSTRING();
        if(val.if_double()) geom->set_parameter(str,val.to_double());
        else INPUT_ERROR("Expected a value for parameter: "+str +" found: "+val);
      }
    } else INPUT_ERROR("Expected a geometric object, got: "+str);
  }
}
bool SURFACE::Execute(void)
{ for(int i=0;i<!to_calc;i++) Out<<"Surface="<<to_calc[i]->surface()<<endl;
  return TRUE;
}
```

Plugin example

- reset project
- edit source file:
  - Surface.c
  - Geometry.h
  - Geometry.c
  - Circle.c
- edit library_files
- Zmake
- edit input file input.inp
- run calculation: Zrun -surf input
- open terminal

Add printout of geometry type and parameters

- do_it_for_me in:
  - Geometry.h
  - Geometry.c
  - Circle.c
  - Surface.c

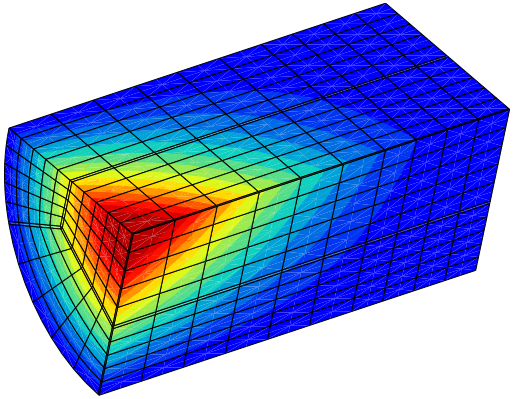
Meshers

- reads an input mesh defined in a .geof file
- apply transformations on the mesh
- write the output mesh in a .geof file

Mesher example

- create a binary file containing nodal temperatures to use as loading in a thermomecanical calculation
- temperatures calculated by means of an analytical function depending on the position of nodes

$$T(node) = T_{max} \left(1 - \frac{r(node)}{R}\right) \left(1 - K \frac{h(node)}{H}\right)^2$$



- generates nsets and elsets in the mesh for nodes/elements where temperature exceeds a critical value

Mesher example: How to connect ?

Base class for meshers and functions to overload

```
class TRANSFORMERS {
...
     virtual MODIFY_INFO_RECORD* get_modify_info_record(); // handles input
     virtual bool                verify_info();
     virtual void apply(UTILITY_MESH& mesh)=0;            // pass in a UTILITY_MESH
};                                                        // that can be modified
```

Object factory declaration

```
DECLARE_OBJECT(TRANSFORMERS,NEW_MESHER,new_mesher)
```

Input file

```
****mesher                          % loaded by Zrun -m
 ***mesh new                        % output meshfile: new.geof
  **open old.geof                   % input meshfile: old.geof
  **new_mesher                      % keyword of new class in OF
  ...
****return
```

Mesher example: Source code (1/3)

```
#include <Object_factory.h>
#include <Zstream.h>
#include <Modify_record.h>
#include <Error_messager.h>
#include <Vector.h>
#include <Transform_geometry.h>
#include <Utility_mesh.h>

Z_START_NAMESPACE;
class MESHER_CYL : public TRANSFORMERS {
  private :                    // member data initialized from the input file
    bool do_ut;
    double Tmax,K,Tcrit;
  public :
    MESHER_CYL();
    virtual ~MESHER_CYL() {}
    MODIFY_INFO_RECORD* get_modify_info_record();
    void apply(UTILITY_MESH&);
    bool verify_info();
};
Z_END_NAMESPACE;
Z_USE_NAMESPACE;

DECLARE_OBJECT(TRANSFORMERS,MESHER_CYL,cyl); // OF declaration
MESHER_CYL::MESHER_CYL() : do_ut(FALSE) {}   // constructor with default values

bool MESHER_CYL::verify_info()
{ if(K<=0.) ERROR("Parameter K invalid"); return TRUE; }
```

Mesher example: Source code (2/3)

```
// Automatic handling of input file syntax
MODIFY_INFO_RECORD* MESHER_CYL::get_modify_info_record()
{ MODIFY_INFO_RECORD* ret = new MODIFY_INFO_RECORD;
  ret->ptr = (void*)this;
  ret->info = "cyl";
  ADD_SINGLE_CMD_TO_MODIF_REC(max_temperature,Tmax);
  ADD_SINGLE_CMD_TO_MODIF_REC(critical_temperature,Tcrit);
  ADD_SINGLE_CMD_TO_MODIF_REC(K,K);
  ADD_SINGLE_BOOL_TO_MODIF_REC(do_ut,do_ut);
  return ret;
}

void MESHER_CYL::apply(UTILITY_MESH& mesh)
{ int i; double r,h,rmax=0.,hmax=0.;
//  Loop on nodes to get rmax and hmax
  for(i=0;i<!mesh.nodes;i++) {
    UTILITY_NODE* node =  mesh.nodes[i];
    VECTOR& posn = node->position;              // Use of a reference to avoid copy
    if(posn[2]>hmax) hmax = posn[2];           // Index goes from 0 to 2 (3D)
    double r = sqrt(posn[0]*posn[0]+posn[1]*posn[1]);
    if(r>rmax) rmax = r;
  }
  Zfstream temp;
  temp.open((mesh.pb_name+".node")(),ios::out|ios::trunc);
  UTILITY_NSET* crit_nodes = new UTILITY_NSET();
  crit_nodes->name="crit_nodes";               // Name of nset in .geof
```

Mesher example: Source code (3/3)

```
  for(i=0;i<!mesh.nodes;i++) {
    UTILITY_NODE* node =  mesh.nodes[i];
    VECTOR& posn = node->position;
    h = posn[2];
    r = sqrt(posn[0]*posn[0]+posn[1]*posn[1]);
    h /= hmax; r /= rmax;
    float Tf = Tmax*(1.-r)*pow(1.-K*h,2.);
    Out<<"Node "<<node->id<<" T="<<Tf<<endl;
    if(Tf>=Tcrit) crit_nodes->nodes.add(node);
    temp.write(&Tf,sizeof(float));
  }
  temp.close();
  if(!crit_nodes) mesh.nsets.add(crit_nodes);
  if(do_ut) {   // Generates a fake .ut to verify temperatures with Zmaster
    Zfstream ut; ut.open((mesh.pb_name+".ut")(),ios::out|ios::trunc);
    ut<<"**meshfile "<<mesh.pb_name<<".geof"<<endl;
    ut<<"**node temperature"<<endl;
    ut<<"**integ"<<endl;
    ut<<"**element"<<endl;
    ut<<"1 1 1 1 1.0"<<endl;
    ut.close();
  }
}
```

Mesher example

- reset project
- view original mesh: Zmaster cylinder
- edit source file: Mesher_cyl.c
- edit library_files
- Zmake
- edit input file input.inp
- run mesher: Zrun -m input
- view results with Zmaster: Zmaster new
- open terminal

Note: the same thing can be done with a **Z-program** : axi_average.z7p

Mesher example

TODO: Add an elset with elements where at-least one node has T>Tcrit
Some hints:

- use connection lists UTILITY_CONNECTION to select UTILITY_ELEMENT during the loop on nodes

```
UTILITY_CONNECTION mesh_connection(&mesh);
mesh_connection.build_node_to_element();
...
UTILITY_NODE* node;
int rk = node->give_rank();
for(i=0;i<mesh_connection.node_elem[rk].size();i++)
  UTILITY_ELEMENT* ele = mesh_connection.node_elem[rk][i];
```

- add element ranks in a SORTED_LIST (items added only once)
  Since an element can be connected to several nodes it is necessary to avoid adding several times the same element in the set.

```
UTILITY_ELEMENT* ele;
SORTED_LIST<int> ele_list;
...
ele_list.add(ele->give_rank());
```

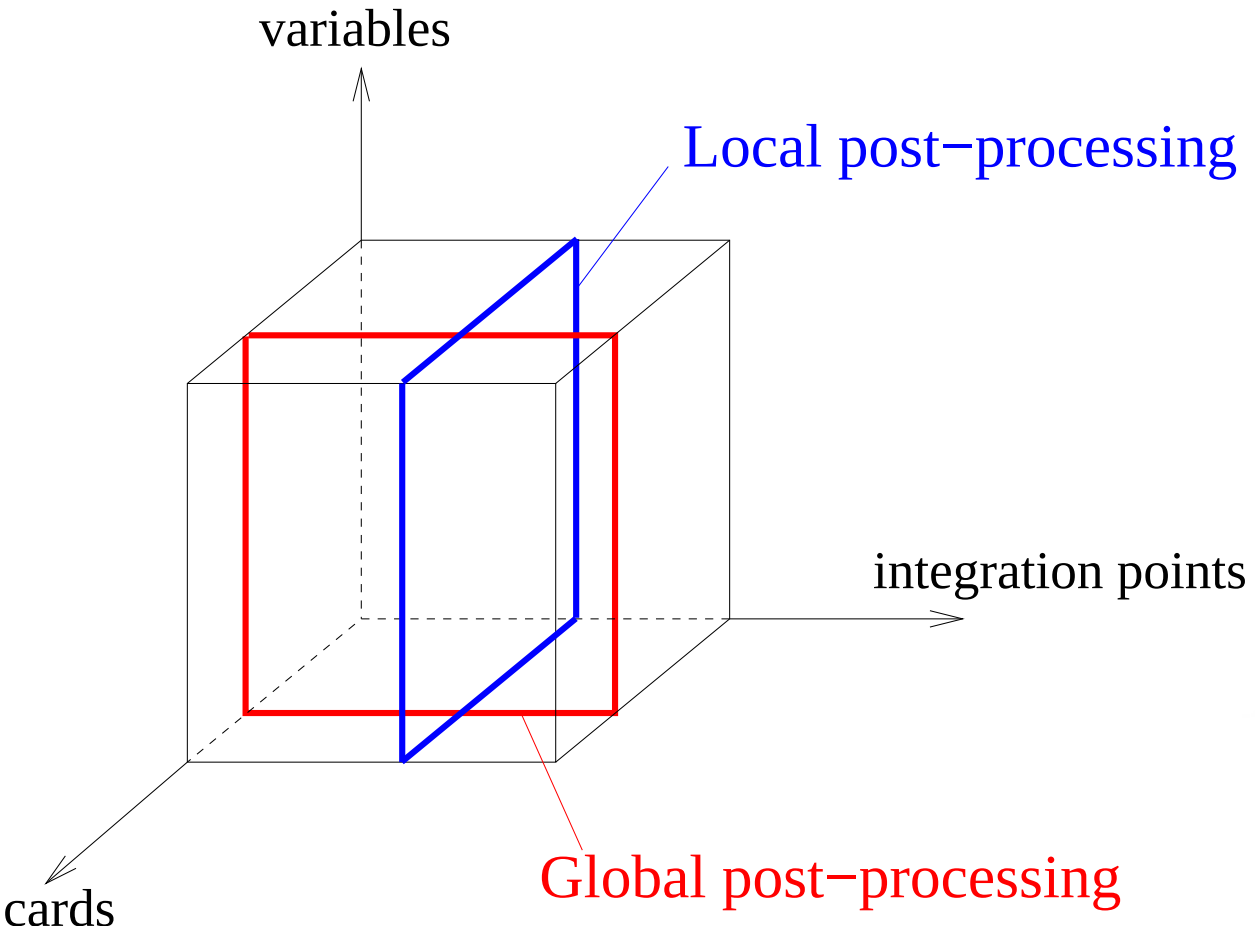
- create the UTILITY_ELSET from the SORTED_LIST after the loop on nodes

∗ do_it_for_me

Post-processing

- read Zébulon results files
- calculate additional results
- write post-processing results files

Local vs global post-processing



Post-processing: How to connect ? (1/2)

Base class for local post and functions to overload

```
class POST_COMPUTATION {
...
     virtual MODIFY_INFO_RECORD* get_modify_info_record();   // handles input
     virtual bool verify_info();
     virtual void input_i_need(int dim, ARRAY<STRING>& data_names)=0;
     virtual void output_i_give(bool& out_every_map, ARRAY<STRING>& data_names)=0;
     virtual bool need_material_file()const { return FALSE; }
     virtual bool compute_on_ipsets()const { return FALSE; }
};

class LOCAL_POST_COMPUTATION : public POST_COMPUTATION {
...
     virtual void compute(const ARRAY<VECTOR>& in,ARRAY<VECTOR>& out)=0;
     virtual bool compute_on_ipsets()const { return TRUE; }
};

class GLOBAL_POST_COMPUTATION : public POST_COMPUTATION {
...
     virtual void compute(ARRAY<POST_ELEMENT*>&,ARRAY<POST_NODE*>&,
       const ARRAY<int>&)=0;
};
```

Post-processing: How to connect ? (2/2)

Object factory declaration

```
DECLARE_OBJECT(LOCAL_POST_PROCESSING,NEW_L_POST,new_local)
DECLARE_OBJECT(GLOBAL_POST_PROCESSING,NEW_G_POST,new_global)
```

Input file

```
****post_processing                    % loaded by Zrun -pp
 ...
 ***local_post_processing
   **process new_local
    ...
 ***global_post_processing
   **process new_global
    ...
****return
```

Local post-processing example: source code (1/2)

Stress indicator: $sign(\sigma_{ii}) \sqrt{\frac{3}{2} s_{ij}\ s_{ij}}$

```
#include <Local_post_computation.h>
#include <Tensor2.h>

Z_START_NAMESPACE;
class POST_INDIC : public LOCAL_POST_COMPUTATION {
 protected :
  STRING var_name;
  int   tsz;
 public :
  POST_INDIC() {}
  virtual ~POST_INDIC() {}
  virtual MODIFY_INFO_RECORD* get_modify_info_record();
  virtual void input_i_need(int,ARRAY<STRING>&);
  virtual void output_i_give(bool&,ARRAY<STRING>&);
  virtual void compute(const ARRAY<VECTOR>&,ARRAY<VECTOR>&);
};
Z_END_NAMESPACE;
Z_USE_NAMESPACE;

DECLARE_OBJECT(LOCAL_POST_COMPUTATION,POST_INDIC,stress_indicator);
MODIFY_INFO_RECORD* POST_INDIC::get_modify_info_record()
{ MODIFY_INFO_RECORD* ret = new MODIFY_INFO_RECORD;
  ADD_SINGLE_CMD_TO_MODIF_REC(var,var_name);
  return ret;
}
```

Local post-processing example: source code (2/2)

```
void POST_INDIC::input_i_need(int dim,ARRAY<STRING>& ret)
{ tsz=TENSOR2::give_symmetric_size(dim);
  ret.resize(tsz);
  ret[0]=var_name+"11"; ret[1]=var_name+"22"; ret[2]=var_name+"33";
  if(dim>1) ret[3]=var_name+"12";
  if(dim>2) { ret[4]=var_name+"23"; ret[5]=var_name+"31"; }
}

void POST_INDIC::output_i_give(bool& every_card,ARRAY<STRING>& ret)
{ every_card=TRUE;
  ret.resize(1); ret[0]=var_name+"_indic";
}

void POST_INDIC::compute(const ARRAY<VECTOR>& in,ARRAY<VECTOR>& out)
{ for(int ic=0;ic<!in;ic++) {          // Loop on cards
    TENSOR2 stress(in[ic]);            // Create a tensor from a vector
    stress.add_sqrt2();                // For Voigt storage see note
    out[ic][0]=sign(stress.trace())*stress.mises();
  }
}
```

Post-processing example

- reset project
- edit source file: Post_indic.c
- edit library_files
- Zmake
- edit input file bending.inp
- run FE calculation: Zrun bending
- run post: Zrun -pp bending
- view results with Zmaster: Zmaster bending
- open terminal

Note the same thing can be done directly with **process function or in **Z-program**: stress_indic.z7p

Behaviors

- material objects
- complete behaviors (ZebFront)

Behavior generalities (1/6)

Definitions

- *External parameters* (`ep`) imposed as input
- *Integrated variables* (`vint`)
- *Auxiliary variables* (`vaux`), just for output
- *Coefficients* (`coef`), material parameters
  (can depend on ep, vint, vaux)
- *Primal and dual variables*, prescribed variables and associated fluxes

Behavior generalities (2/6)

Primal and dual variables in various fields

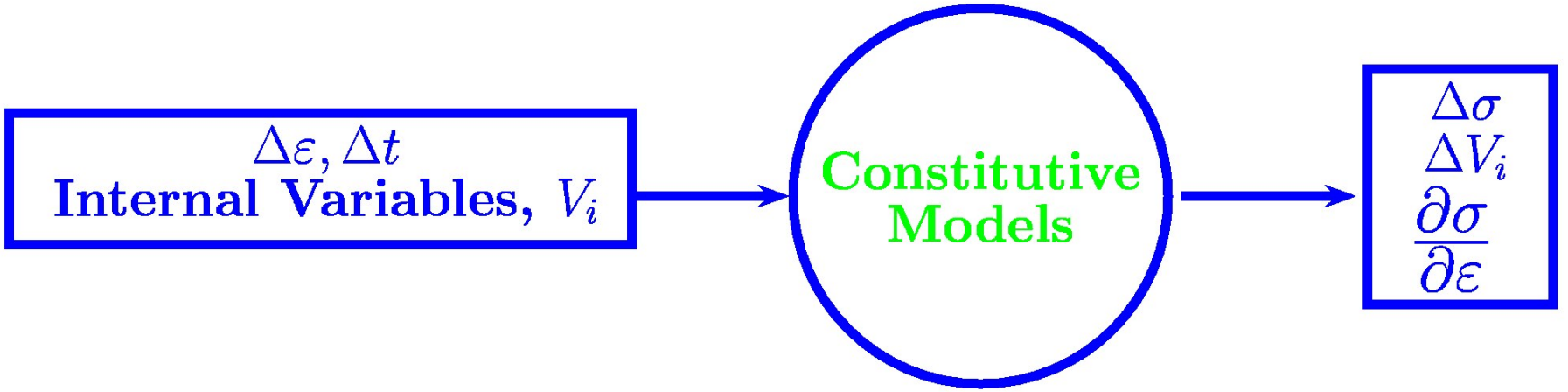
problem	primal	dual
mechanics, small perturbation	$\underset{\sim}{\varepsilon}$	$\underset{\sim}{\sigma}$
mechanics, large deformation	$\underset{\sim}{\boldsymbol{F}}$	$\underset{\sim}{\boldsymbol{S}}$
thermal pb	$(T, \underline{\mathbf{grad}}\, T)$	$(H, \underline{\boldsymbol{q}})$
diffusion	concentration	flux
electrostatics	$\underline{\mathbf{grad}}\, V$	$\underline{\boldsymbol{E}}$
magnetostatics	$\underline{rot}\ \underline{\boldsymbol{A}}$	$\underline{\boldsymbol{H}}$

$\underset{\sim}{\varepsilon}$ strain tensor, $\underset{\sim}{\boldsymbol{F}}$ deformation gradient, T temperature, V electric potential, $\underline{\boldsymbol{A}}$ potential vector, $\underset{\sim}{\sigma}$ Cauchy stress tensor, $\underset{\sim}{\boldsymbol{S}}$ second Piola–Kirchhoff stress tensor, H enthalpy, $\underline{\boldsymbol{q}}$ thermal flux , $\underline{\boldsymbol{E}}$ electric field , $\underline{\boldsymbol{H}}$ magnetic field.

Behavior generalities (3/6)

Generic interface for any constitutive equation

For each Gauss Point...



Behavior generalities (4/6)

Time discretization: $\Delta\alpha = \int_t^{t+\Delta t} \dot{\alpha} dt\tau$

Explicit integration: Runge-Kutta

Implicit integration:

- Generalized mid-point rule ($\theta - method\ A$):

$$\Delta\alpha = \dot{\alpha}(t + \theta\Delta t)\Delta t = \dot{\alpha}_\theta \Delta t$$

- Trapezoidal integration ($\theta - method\ B$):

$$\Delta\alpha = ((1-\theta)\dot{\alpha}_t + \theta\dot{\alpha}_{t+\Delta t})\,\Delta t = ((1-\theta)\dot{\alpha}_0 + \theta\dot{\alpha}_1)\,\Delta t$$

  - $0 < \theta < 1$, $\theta = 0$, explicit; $\theta = 1$, implicit
  - $0.5 < \theta < 1$, stable
  - $\theta = 0.5$, second order accurate

Behavior generalities (5/6)

Behavior object-oriented design

Material objects

- Flow
- Behavior
- Isotropic hardening
- Criterion
- Kinematic hardening
- Damage
- Elasticity
- Potential
- Thermal strain
- etc ...

Typical assembly for viscoplasticity

- Behavior
  - Elasticity
  - Thermal strain
  - Potential
    - Criterion
    - Flow
    - Isotropic hardening
    - Kinematic hardening
    - Kinematic hardening

Behavior generalities (6/6)

Typical viscoplastic model

behavior

- **elasticity isotropic**
- **thermal_strain isotropic** $\underset{\sim}{\epsilon}^{th} = \alpha(T - T_{ref})$
- **potential ev**
  - **criterion mises** $f = J\left(\underset{\sim}{\sigma}' - \sum_i \underset{\sim}{\mathbf{X}}_i\right) - R$
  - **flow norton** $\dot{p} = \left\langle \frac{f}{K} \right\rangle^n \quad , \quad \underset{\sim}{\dot{\epsilon}}^{ev} = \dot{p}\underset{\sim}{\mathbf{n}}$
  - **isotropic nonlinear** $R = R_0 + Q(1 - e^{-bp})$
  - **kinematic nonlinear** $\underset{\sim}{\mathbf{X}} = \frac{2}{3}C\underset{\sim}{\alpha} \quad , \quad \underset{\sim}{\dot{\alpha}} = \dot{p}\left[\underset{\sim}{\mathbf{n}} - \frac{3D}{2C}\underset{\sim}{\mathbf{X}}\right]$

Material object example: How to connect ?

Example for Sellars-tegart flow: $\dot{p} = A\left[sinh\left(\frac{f(\underset{\sim}{\sigma})}{K}\right)\right]^m$

Base class for FLOW and functions to overload

```
class FLOW : public MATERIAL_PIECE {
...
     virtual void initialize(ASCII_FILE&,MATERIAL_PIECE* boss);
     virtual double      flow_rate(double v, double crit);
     virtual double      dflow_dv();
     virtual double      dflow_dcrit();
...
};
```

Object factory declaration

```
DECLARE_OBJECT(FLOW,NEW_FLOW,new_flow)
```

Input file

```
***behavior
 **potential ...
  *flow new_flow
  ...
****return
```

Material object example: Source code (1/2)

```
#include <Object_factory.h>
#include <File.h>
#include <Flow.h>

Z_START_NAMESPACE;
class FLOW_SINH : public FLOW {
  protected :
    double _os;
    COEFF A,K,m;
  public :
    FLOW_SINH() {}
    virtual ~FLOW_SINH() {}
    void initialize(ASCII_FILE& file, MATERIAL_PIECE*);
    double flow_rate(double v, double crit);
    double dflow_dv();
    double dflow_dcrit();
};
Z_END_NAMESPACE;
Z_USE_NAMESPACE;

DECLARE_OBJECT(FLOW,FLOW_SINH,sinh)

double FLOW_SINH::flow_rate(double, double crit)
{  _os=crit;
   return A*pow(sinh(crit/K),m);
}

double FLOW_SINH::dflow_dcrit()
{ return (m*A/K)*pow( sinh(_os/K), m-1 )*cosh(_os/K); }
```

Material object example: Source code (2/2)

```
double FLOW_SINH::dflow_dv()
{ return 0.; }

void FLOW_SINH::initialize(ASCII_FILE& file, MATERIAL_PIECE* mp)
{ FLOW::initialize(file,mp);
  for(;;) {
    STRING str=file.getSTRING();
    if( str[0]=='*' ) break;
    else if(str=="A" ) A.read(str,file,mp);
    else if(str=="K" ) K.read(str,file,mp);
    else if(str=="m" ) m.read(str,file,mp);
    else INPUT_ERROR("Unknown coefficient:"+str);
  } file.back();
}
```

Material object example

- reset project
- edit source file: Flow_sinh.c
- edit library_files
- Zmake
- edit input file sinh.inp
- edit material file flow_sinh
- run simulation: Zrun -S sinh
- draw
- open terminal

ZebFront example

Norton viscoplastic Model

$$\underset{\sim}{\epsilon} = \underset{\sim}{\epsilon}_{el} + \underset{\sim}{\epsilon}_{v} \qquad \underset{\sim}{\sigma} = \underset{\sim}{\underset{\sim}{E}} : \underset{\sim}{\epsilon}_{el}$$

$$\dot{\underset{\sim}{\epsilon}}_{v} = \dot{p}\,\underset{\sim}{n} \qquad \dot{p} = \left\langle \frac{f(\underset{\sim}{\sigma})}{K} \right\rangle^{n}$$

$$f(\underset{\sim}{\sigma}) = J = \sqrt{\frac{3}{2}\,\underset{\sim}{s} : \underset{\sim}{s}} \qquad \underset{\sim}{n} = \frac{3}{2J}\,\underset{\sim}{s}$$

ZebFront code for explicit (runge-kutta) integration (1/2)

```
#include <Basic_nl_behavior.h>
#include <Basic_nl_simulation.h>
#include <Elasticity.h>
@Class NORTON : BASIC_NL_BEHAVIOR, BASIC_SIMULATOR {
   @Name     norton;
   @SubClass ELASTICITY elasticity;
   @Coefs    K, n;
   @tVarInt  eel;                          // tensorial integrated variable
   @sVarInt  evcum;                        // scalar integrated variable
};
```

ZebFront code for explicit integration (2/2)

```
@StrainPart {  // This block is called at the end of the integration
  sig = *elasticity*eel;
  m_tg_matrix=*elasticity;   // Note no consistent tg matrix !
}
@Derivative {  // Specify dvarint = ... for each integrated variable named varint
  sig = *elasticity*eel;
  TENSOR2 sprime = deviator(sig);
  double  J      = sqrt(1.5*(sprime|sprime));
  if (J<=0.0) { devcum = 0.0; deel = deto;
                resolve_flux_grad(*elasticity, deel, deto);
  }
  else  {
    devcum  = pow(J/K,n);
    TENSOR2 norm = sprime*(1.5/J); TENSOR2 dein = devcum*norm;
    deel = deto - dein;
    resolve_flux_grad(*elasticity, deel, deto, dein);
  }
}
```

Note resolve_flux_grad() is only needed in simulator mode for mixed loadings:

```
// input: mat, dein, d_grad (mixed strain/stress)
// output: deel, d_grad (strain)
void BASIC_SIMULATOR::resolve_flux_grad(const SMATRIX& mat, TENSOR2& deel,
  TENSOR2& dgrad, const TENSOR2& dein) {  ...  }
```

ZebFront example: run the simulation

- reset project
- edit source file: Norton.z
- edit library_files
- Zmake
- edit input file visco.inp
- edit material file norton.mat
- run simulation: Zrun -S visco
- draw
- open terminal

Add a threshold to the model:

$\dot{p} = \left\langle \frac{J-R0}{K} \right\rangle^n$ do_it_for_me

Add nonlinear isotropic hardening:

$\dot{p} = \left\langle \frac{J-R}{K} \right\rangle^n \qquad R = R0 + Q(1 - e^{-bp})$ do_it_for_me

ZebFront code with implicit integration (1/2)

```
@Class NORTON : BASIC_NL_BEHAVIOR, BASIC_SIMULATOR {
   @Name     norton;
   @SubClass ELASTICITY elasticity;
   @Coefs    K, n;
   @Coefs    R0, Q, b;
   @tVarInt  eel;
   @sVarInt  evcum;
   @Implicit
};
@StrainPart {
  evi  = eto - eel;
  sig  = *elasticity*eel;
  if (integration&LOCAL_INTEGRATION::THETA_ID) {  // calculation of consistent tgmat
     TENSOR4 tmp(psz,f_grad,0,0);                 // from jacobian see note
     if (Dtime>0.0) m_tg_matrix=*elasticity*tmp;
     else           m_tg_matrix=*elasticity;
  } else { if (m_flags&CALC_TG_MATRIX) m_tg_matrix=*elasticity; }
}

@Derivative {
// Same as before ...
}
```

ZebFront code with implicit integration (2/2)

```
@CalcGradF { // Residual and jacobian : f_vec_var is residual for int variable var
  ELASTICITY& E=*elasticity; sig = E*eel;
  double       R   = R0+Q*(1-exp(-b*p));
  TENSOR2 sprime = deviator(sig);
  double  J = sqrt(1.5*(sprime|sprime)); double f = J - R;
  if ( (f>0.0 && devcum>=0) || (devcum>0.0) ) {
     TENSOR2        norm            = sprime*(1.5/J);
                    f_vec_eel      += norm*devcum;   // residuals initialized to dvar
                    f_vec_evcum    -= dt*pow(f/K,n);
     SMATRIX dn_ds   = unit32;      // predefined to 1.5 * deviator operator
     dn_ds -= norm^norm; dn_ds *= theta*devcum/J;
     SMATRIX dn_deel = dn_ds*E;
     double  dv_df   = tdt*n*pow(f/K,n-1)/K;   // tdt predefined to theta*dt
     TENSOR2 df_fs   = dv_df*norm;
     deel_deel      += dn_deel;                // diagonal blocks init to 1
     devcum_devcum += dv_df*Q*b*exp(-b*evcum);
     deel_devcum     = norm;     devcum_deel   -= df_fs*E;
  }
}
```

At time $t+\theta\Delta t$: $\underset{\sim}{R}_{el} = \Delta\underset{\sim}{\epsilon}_{el} - \Delta\underset{\sim}{\epsilon} - \Delta p\underset{\sim}{n}$ $R_p = \Delta p - \Delta t \left(\frac{J-R}{K}\right)^n$

$$\frac{\partial \underset{\sim}{R}_{el}}{\partial \Delta\underset{\sim}{\epsilon}_{el}} = \underset{\sim}{1} + \Delta p \frac{\partial \underset{\sim}{n}}{\partial \underset{\sim}{\sigma}} : \frac{\partial \underset{\sim}{\sigma}}{\partial \underset{\sim}{\epsilon}_{el}} : \frac{\partial \underset{\sim}{\epsilon}_{el}}{\partial \Delta\underset{\sim}{\epsilon}_{el}} = \underset{\sim}{1} + \theta\Delta p \frac{\partial \underset{\sim}{n}}{\partial \underset{\sim}{\sigma}} : \underset{\sim}{E}$$

$$\frac{\partial \underset{\sim}{n}}{\partial \underset{\sim}{\sigma}} = \frac{1}{J}\left(\frac{3}{2}\underset{\sim}{I_s} - \underset{\sim}{n}\otimes\underset{\sim}{n}\right) \qquad \underset{\sim}{I_s} \text{ such that } \underset{\sim}{I_s} : \underset{\sim}{\sigma} = \underset{\sim}{s} \quad \ldots$$

ZebFront implicit example: run the simulation

- reset
- edit source file: Norton.z
- edit library_files
- Zmake
- edit material file norton.mat set_R0 set_iso
- simulation
  - edit sim input file visco.inp set rk set theta
  - run simulation: Zrun -S visco
  - draw
- FE with RVE element
  - edit FE input file lcf_rve.inp set rk set theta
  - run FE: Zrun lcf_rve
  - draw
- open terminal

ZebFront code with non-linear kinematic hardening (1/2)

$$f(\underset{\sim}{\sigma}) = J(\underset{\sim}{\sigma} - \underset{\sim}{X}) - R$$

$$\underset{\sim}{X} = \frac{2}{3} C \underset{\sim}{\alpha} \qquad \dot{\underset{\sim}{\alpha}} = \dot{p}\,(\underset{\sim}{n} - D\underset{\sim}{\alpha})$$

```
@Class NORTON : BASIC_NL_BEHAVIOR, BASIC_SIMULATOR {
// same as before ...
   @Coefs    C1, D1;
   @tVarInt  alpha1;
};
@StrainPart {
// same as before ...
}
@Derivative {
  double CC1=C1/1.5;
  sig = *elasticity*eel;
  TENSOR2 Xv1    = CC1*alpha1;
  TENSOR2 sigeff = sig - Xv1;
  TENSOR2 sprime = deviator(sigeff);
  double  J      = sqrt(1.5*(sprime|sprime));
// same as before
...
  if(CC1>0.) dalpha1  = devcum*(norm - D1*Xv1/CC1);
...
}
```

ZebFront code with non-linear kinematic hardening (2/2)

```
@CalcGradF {
  double CC1 = C1/1.5;
  TENSOR2 Xv1 = CC1*alpha1;
  TENSOR2 sigeff = sig - Xv1;
  TENSOR2 sprime = deviator(sigeff);
  double  J      = sqrt(1.5*(sprime|sprime));
// same as before
...
     if(CC1>0.) {
        SMATRIX dn_dal1 = dn_ds*CC1;
        TENSOR2 m1     = norm - D1*Xv1/CC1;
        f_vec_alpha1  -= devcum*m1;
        deel_dalpha1      -= dn_dal1;
        dalpha1_deel      -= dn_deel;
        dalpha1_dalpha1   += dn_dal1; dalpha1_dalpha1.add_to_diagonal(tdv*D1);
        dalpha1_devcum    -= m1;
        devcum_dalpha1     = df_fs*CC1;
     }
...
}
```

ZebFront with nl kinematic: run the simulation

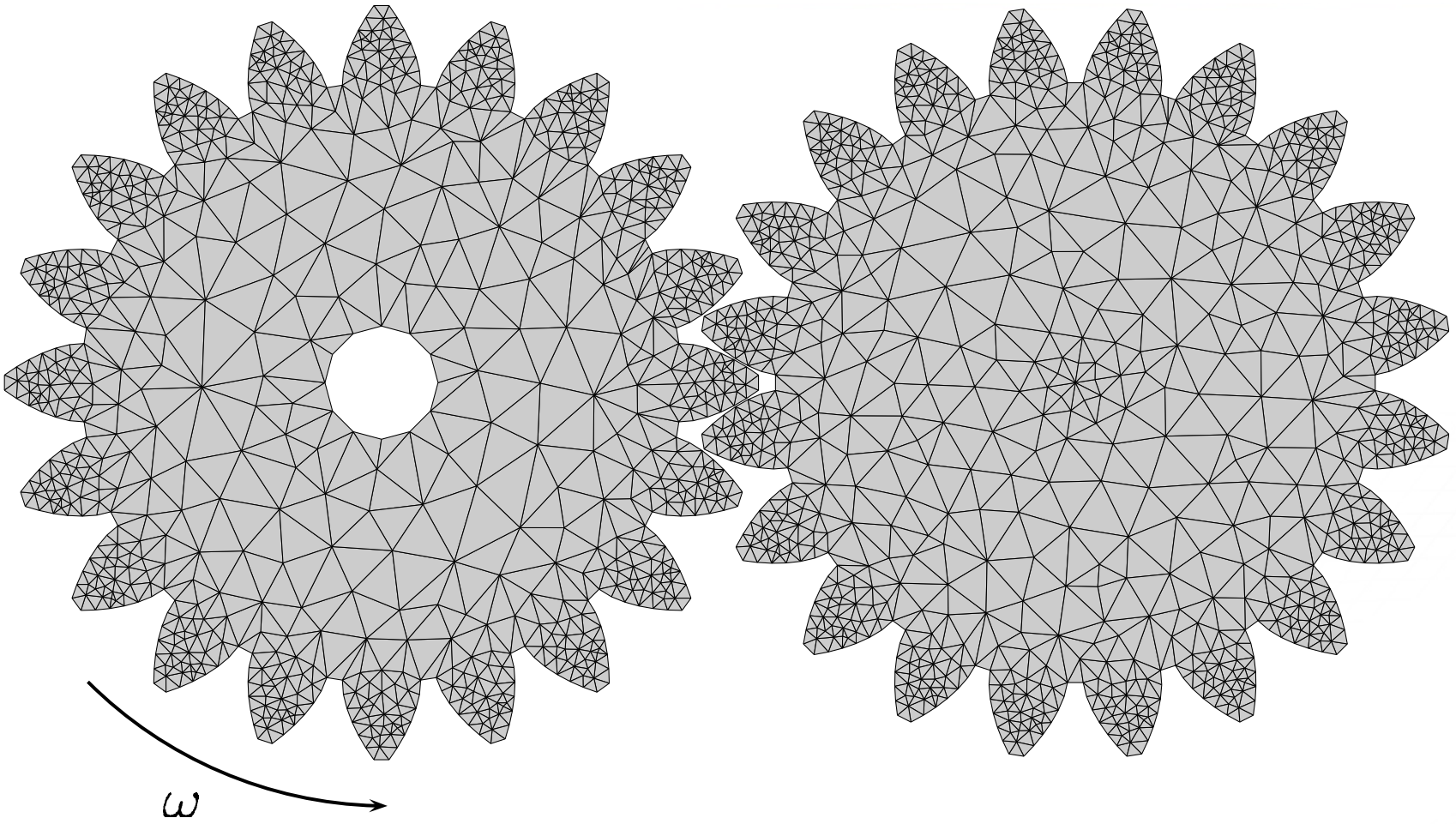
- reset
- edit source file: Norton.z
- Zmake
- edit material file norton.mat
- simulation
  - edit sim input file visco.inp set rk set theta
  - run simulation: Zrun -S visco draw
- FE with RVE element
  - edit FE input file lcf_rve.inp set rk set theta
  - run FE: Zrun lcf_rve draw
- open terminal

ToDo: Add a second kine hardening do it for me

Boundary conditions

- class BC

BC example: Impose an angular velocity ω



BC example: How to connect ?

Base class for BC and functions to overload

```
class BC : public BASE_VALUE_HANDLER {
...
  private :
    virtual void _update(MESH&)=0;
...
  public :
    virtual void initialize(ASCII_FILE&,MESH&,int ipc)=0;
...
};
```

Object factory declaration

```
DECLARE_OBJECT(BC,NEW_BC,new_bc)
```

Input file

```
****calcul
 ...
 ***bc
  **new_bc  ...
  ...
****return
```

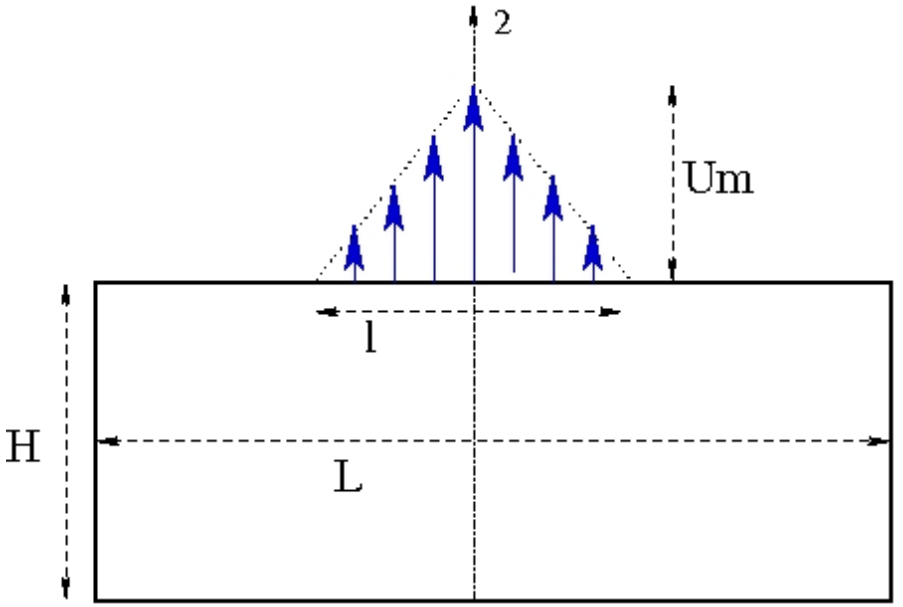
BC example: compile/run calculation

- reset
- edit source file: Angular_rotation.c
- Zmake
- edit input file all.inp
- run calculation Zrun all
- view results Zmaster all
- open terminal

Z-program

- scripting language (no compilation)
- C++ syntax
- re-use of Zébulon objects
- applications: meshing, boundary conditions, post-processing ...

Z-program example 1 (1/2)



- Create a mesh file with few parameters : dimenions (L, H) and the number of elements on the bottom line
- Apply specific boundary conditions on top line :

$$U_2(x) = \begin{cases} 0 & \text{if } |x| > l/2 \\ U_m(1 - |\frac{2x}{l}|) & \text{if } |x| \leq l/2 \end{cases}$$

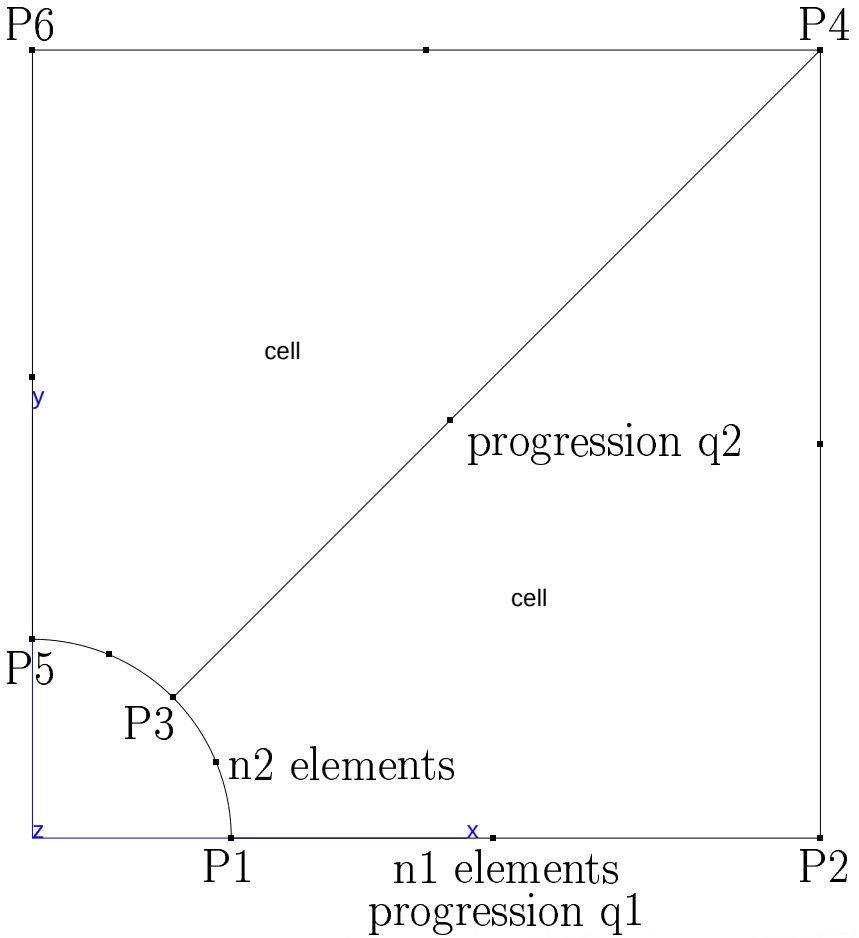
Z-program example 1 (2/2)

- Mesh build and data save :
  - reset
  - square.z7p : program source
  - Zrun -B square.z7p : create square.geof file
  - view geof
- Problem computation with "external" boundary conditions:
  - square.inp : a basic inp file with external bc after "**z7p"
  - bc.z7p : program source to apply a vertical displacement that depends on *x*.
  - Zrun square.inp : use standard command to launch the problem
  - view geof : show results

open terminal

Z-program example 2 (1/2)

Calculation of homogeneized elastic properties



- parameters: perforation radius, volume fraction, perforation angle
- takes as input a material file for the homegeneous material
- generates a material file for the equivalent homogeneized material

Z-program example 2 (2/2)

- reset
- edit Z-program source file: elastic_periodic.z7p
  - mesh one quarter of the 2D cell
  - apply symmetries to generate the whole 2D cell (Zrun -m)
  - apply extension to generate the 3D cell (Zrun -m)
  - modify coordinates according to the perforation angle (Zrun -m)
  - generate nsets to apply periodicity BCs (Zrun -m)
  - run 6 FE computations with the periodic element using components E_{ij} of the macroscopic strain tensor as loading
  - exploit results to generate the homegeneized material file
- edit input material file: homogeneous
- run Zrun -B elastic_periodic.z7p
- edit homogeneized material file: homogeneized
- open terminal

Geometrical Elements

- UTILITY_ELEMENT (geof file, Zmaster)
- geometrical ELEMENT
- INTERPOLATION
- INTEGRATION

Element geometry/integration architecture (1/3)

```
class ELEMENT : public Z_OBJECT {   // Geometrical element
  public :
...
    GEOMETRY*           geometry;
    ARRAY<GNODE*>       node;
    void get_elem_coord (MATRIX& elem_coord,bool use_coord=FALSE)const;
    virtual void start(const MATRIX& elem_coord);     // for loop on integ points
    virtual void next(const MATRIX& elem_coord);
    virtual int  ok()const;
    const VECTOR& shape() const;                      // at current integ point
    const VECTOR& shape_inv() const;
    void compute_dshape_dx(MATRIX& dshape_dx) const;
    void gradient_operator(const MATRIX& dshape_dx,MATRIX& op) const;
    void symmetric_gradient_operator(const MATRIX& dshape_dx,MATRIX& op) const;
    void integrate(const double& x, double& tot) const;
...
};
```

Example : Volume integration

```
ELEMENT* elem; double vol=0;
MATRIX elem_coord; get_elem_coord(elem_coord,TRUE);
for(elem->start(element_coord);elem->ok();elem->next(elem_coord)) {
  elem->integrate(1.,vol);
}
```

Element geometry/integration architecture (2/3)

```
class GEOMETRY : public Z_OBJECT {
  public :
    INTEGRATION* integration;
    GEOM_TYPE    geom_type;  // (NO_GEO,_1D,_SPH,_CYL,_2D,_AXI,_3D,_1DOT,_2DOT)
    SPACE_TYPE   space_type; // (ST_SPACE,ST_SURFACE,ST_LINE,ST_DOT,ST_SHELL,ST_BEAM)
    const INTERPOLATION_FUNCTION* interpolation;
    virtual void start(const MATRIX& elem_coord); ...
    virtual const VECTOR& shape() const;  ...
    virtual void integrate(const double& x, double& tot) const;
...
};
class SPACE : public GEOMETRY {
  protected :
    SMATRIX jacobian_matrix; MATRIX dshape_dx;
  public :
...
    void compute_dshape_dx(MATRIX& dshape_dx) const;
    virtual double jacob2(SMATRIX& jm, const MATRIX& elem_coord);
    virtual void gradient_operator(const MATRIX& dshape_dx, MATRIX& b)=0;
    virtual void symmetric_gradient_operator(const MATRIX& dshape_dx, MATRIX& b)=0;
...
};
class SPACE3D  : public SPACE { ... };
class SPACEAXI : public SPACE { ... };
```

Element geometry/integration architecture (3/3)

```
class INTEGRATION {
  public :
...
    ARRAY<REF_GAUSS_POINT*> rgp;  // list of reference Gauss Point
    REF_GAUSS_POINT*        crgp; // current reference Gauss Point
...
    virtual void start(const MATRIX& elem_coord);
    virtual void next(const MATRIX& elem_coord);
    virtual int  ok() const;
...
    virtual const VECTOR& shape() const     return crgp->shape;
    virtual const VECTOR& shape_inv() const return crgp->shape_inv;
    virtual const MATRIX& deriv() const     return crgp->deriv;
    virtual const double& weight() const    return crgp->weight;
...
};
class REF_GAUSS_POINT {
...
   VECTOR chi,shape,shape_inv;
   double weight;
   int    dimen, nb_node, nb_gp, gp_id;
   MATRIX deriv;
...
};
```

Elements: How to connect (1/4)

Base class for UTILITY_ELEMENT and functions to overload

```
class UTILITY_ELEMENT : public GRAPHICS_OBJECT {
...
     virtual void initialize(const STRING&);                // Set # nodes, integ points
     virtual ARRAY<UTILITY_BOUNDARY>& get_faces();
     virtual ARRAY<UTILITY_BOUNDARY>& get_edges();
     virtual void set_integ(VECTOR& values, int& index);  // Integ points iso-contours
};
```

Object factory declarations

```
DECLARE_OBJECT(UTILITY_ELEMENT,NEW_ELE,ele_name)
DECLARE_GEOMETRY(ele_name,#NNODE,#NDIM,#NGP,geom_type,"gauss",space_type)
where: enum GEOM_TYPE {NO_GEOMETRY,_1DOT,_2DOT,_3DOT,_1D,_SPH,_CYL,_2D,_AXI,_3D, ...}
       enum SPACE_TYPE {ST_SPACE,ST_SURFACE,ST_LINE,ST_DOT,ST_SHELL,ST_BEAM};
```

Input file: geof file

```
***geometry
...
 **element
1000
  1 c2d8  1 2 3 4 5 6 7 8
  1 ele_name 5 6 ...
  ...
***return
```

Elements: How to connect (2/4)

Base class for INTERPOLATION_FUNCTION and functions to overload:

```
class INTERPOLATION_FUNCTION {
...
  virtual VECTOR shape(const VECTOR& chi)const=0;
  virtual MATRIX deriv(const VECTOR& chi)const=0;
  virtual void   get_sides(ARRAY< ARRAY<int> >&)const=0;
  virtual void   local_position_of_node(int,VECTOR&)const=0;
};
```

Object factory declaration:

```
MAKE_INTERPOLATION_FUNCTION(#DIM,#NNODE)
```

- that expands as:

```
class INTERPOLATION_FUNCTION_#DIM_#NNODE : public INTERPOLATION_FUNCTION {
public :
    INTERPOLATION_FUNCTION_#DIM_#NNODE(int,int);
    virtual ~INTERPOLATION_FUNCTION_#DIM_#NNODE();
    virtual VECTOR shape(const VECTOR& chi)const;
    virtual MATRIX deriv(const VECTOR& chi)const;
    virtual void   get_sides(ARRAY< ARRAY<int> >&)const;
    virtual void   local_position_of_node(int,VECTOR&)const;
};
...
```

Elements: How to connect (3/4)

Base class for REF_GAUSS_POINT

```
class REF_GAUSS_POINT {
   int    dimen, nb_node, nb_gp, gp_id;
   double weight;
   VECTOR chi; // Coordinates of integ point gp_id in reference element
   VECTOR shape,shape_inv;
   MATRIX deriv;
};
```

Object factory declaration

```
DEFINE_REF_GAUSS_POINT(#DIM,#NGP,#NNODE)
```

Elements: How to connect (4/4)

Code needed for the new integration:

always the same

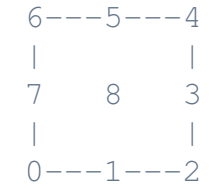
```
class RGP_#DIM_N#NNODE : public virtual REF_GAUSS_POINT {
  public :   RGP_#DIM_N#NNODE() { init(); }
             virtual ~RGP_#DIM_N#NNODE() {}
};
class RGP_#DIM_G#NGP : public virtual REF_GAUSS_POINT {
  public :   RGP_#DIM_G#NGP();
             virtual ~RGP_#DIM_G#NGP() {}
};
class RGP_#DIM_G#NGP_N#NNODE : public RGP_#DIM_#G#NGP , public RGP_#DIM_N#NNODE {
  public :   RGP_#DIM_G#NGP_N#NNODE(int gp_id);
             ~RGP_#DIM_G#NGP_N#NNODE() {}
};
```

depending on the particular integration

```
RGP_#DIM_G#NGP()  % coordinates/weight of integ point gp_id
{ chi[0]=...; chi[1]=...; weight=...;
}
RGP_#DIM_G#NGP_N#NNODE::RGP_#DIM_G#NGP_N#NNODE(int _gp_id) :
  REF_GAUSS_POINT(#DIM,#NGP,_gp_id,#NNODE)
{ shape_inv[0]=...;  // for each node at integ point _gp_id
}
```

Element example

Q9 element : 2D 9 nodes rectangular element with 9 (c2d9) or 4 (c2d9r) integration points



- reset project
- edit source files: Mesher_c2d9.c C2d9.c
- edit library_files
- Zmake
- edit input file T.inp
- run mesher to creates c2d9 from c2d8: Zrun -m T
- run FE calculation: Zrun T
- view results with Zmaster: Zmaster T
- open terminal

Element formulations

- geometrical element (ELEMENT)
- element with degrees of freedom (D_ELEMENT)
- physical elements (P_ELEMENT)
- element formulations
  (MCESD, MCE_FINITE_STRAIN, TCE, ...)

Basic element class hierachy

```
class ELEMENT : public Z_OBJECT {   // Geometrical element
  public :
    GEOMETRY*  geometry;
    ARRAY<GNODE*> node;
    int current_ip;
    virtual void start(const MATRIX& elem_coord);   // for loop on integ points
    const VECTOR& shape() const;                    // calls to GEOMETRY
    void gradient_operator(const MATRIX& dshape_dx,MATRIX& op) const;
    void integrate(const double& x, double& tot) const;
...
};
class D_ELEMENT : public ELEMENT {  // Element with dofs
public :
    CARRAY<DOF*>       dof;
    void get_elem_d_dof_tot (VECTOR& elem_d_dof_tot)const;
    void get_elem_d_dof_incr (VECTOR& elem_d_dof_incr)const;
    virtual INTEGRATION_RESULT* internal_reaction(bool,VECTOR&,SMATRIX&,bool);
...
};
class P_ELEMENT : public D_ELEMENT { // Element with behavior
  protected :
    ARRAY<MAT_DATA>  mat_data;
    ARRAY<LOCAL_FRAME*> rotation;
    ARRAY<BEHAVIOR*> behavior;
  public :
    MAT_DATA& pub_mat_data() { return mat_data[current_ip]; }
    BEHAVIOR* pub_behavior();                       // behavior at current ip
    virtual void start(const MATRIX&);              // overloaded to handle behavior & mat_data
    virtual bool cal_val_at_integration(const STRING& var_name, VECTOR& val)const;
...
};
```

MCESD: Mechanical continuum element small deformation (1/2)

```
class MECHANICAL_ELEMENT : public P_ELEMENT { ... };
class MECHANICAL_VOLUME_ELEMENT : public MECHANICAL_ELEMENT { ... };
class MCESD : public MECHANICAL_VOLUME_ELEMENT {
  public :
    virtual INTEGRATION_RESULT* internal_reaction(bool,VECTOR&,SMATRIX&,bool only_tg=FALSE);
    virtual void compute_B(MATRIX&);
    virtual void compute_Bu(TENSOR2& Ret,const MATRIX& B,const VECTOR& u,bool grad=FALSE);
    virtual void compute_Btu(VECTOR& Ret,const MATRIX& B,const TENSOR2& sig);
    virtual void compute_BtDB(MATRIX& Ret,const MATRIX& B, const MATRIX& D);
...
};

INTEGRATION_RESULT* MCESD::internal_reaction( bool if_compute_stiffness,
  VECTOR& resi, SMATRIX& stiff, bool only_get_tg_matrix)
{ INTEGRATION_RESULT* ret=NULL;
  int flags=0;
  // ----- get positions and displacements ------------------------
  MATRIX* tg_matrix=NULL;
  VECTOR elem_U(nb_dof()), elem_dU(nb_dof());
  TENSOR2 delta_grad(tsz);
  MATRIX B(tsz,nb_dof()); MATRIX elem_coord;
  SMATRIX dK; VECTOR dF(nb_dof());

  update();  // Set node coordinates to the beginnning of increment
             // if mcesd_flags = UPDATED

  get_elem_coord(elem_coord,mcesd_flags&UPDATED);  // MATRIX elem_coord(dim,nnode)
  get_elem_d_dof_tot(elem_U); get_elem_d_dof_incr(elem_dU);
  resi=0.;
  if(if_compute_stiffness) { stiff=0.; flags=BEHAVIOR::CALC_TG_MATRIX; dK.resize(nb_dof()); }
  if(!only_get_tg_matrix) flags=flags|BEHAVIOR::GIVE_FLUX;
  else { if(flags==0) flags=BEHAVIOR::CALC_TG_MATRIX; }

  double ddV; bool moddV;
```

MCESD: Mechanical continuum element small deformation (2/2)

```
  //integration of stiffness and internal forces
  for(start(elem_coord);ok();next(elem_coord)) {   // loop on integ points
     if(pub_behavior()->standard_grad[BSTD_GRAD_ETO] == NULL) {
         ERROR("Grad variable eto must be available for element id:"+itoa(id));
     }
     TENSOR2_GRAD& eto = *(TENSOR2_GRAD*)pub_behavior()->standard_grad[BSTD_GRAD_ETO];
     compute_B(B);
     compute_Bu(delta_grad, B, elem_dU,TRUE);
     compute_Bu(eto, B, elem_U,FALSE);
     rotate_to_material(delta_grad); rotate_to_material(eto);
     if((ret=pub_behavior()->integrate(pub_mat_data(), delta_grad, tg_matrix, flags)) )
         return ret;
     if(pub_behavior()->standard_flux[BSTD_FLUX_SIG] == NULL) {
         ERROR("FLux variable sig must be available for element id:"+itoa(id));
     }
     TENSOR2_FLUX& sig = *(TENSOR2_FLUX*)pub_behavior()->standard_flux[BSTD_FLUX_SIG];
     rotate_from_material(sig); rotate_from_material(eto);
     moddV=modif_dV(ddV,elem_U,elem_dU);
     if(if_compute_stiffness) {
       if((!rotation)==0) compute_BtDB(dK,B,*((SMATRIX*)tg_matrix));
       else {
         SMATRIX mat = *((SMATRIX*)tg_matrix);
         rotate_from_material(mat);
         compute_BtDB(dK, B, mat);
       }
       if(moddV) dK*=ddV;
       integrate(dK,stiff);
     }
     compute_Btu(dF,B,sig);
     if(moddV) dF*=ddV;
     integrate(dF,resi);
  }
  return NULL;
}
```

MCE_FINITE_STRAIN element (1/5)

```
class MCE_FINITE_STRAIN : public MECHANICAL_VOLUME_ELEMENT {
 public :
   enum FORMULATION_TYPE { UNDEFINED, UPDATED, TOTAL, FINAL };
 protected :
...
    FORMULATION_TYPE lagrange_type;
    virtual void compute_BF(const MATRIX& dshape_dx, MATRIX& BF);
    virtual void compute_Be(const MATRIX& dshape_dx, MATRIX& Be);
    virtual void compute_BE(const VECTOR& Ut, const MATRIX& BF, const MATRIX& Be,
      MATRIX& BE);
    virtual void compute_F(const MATRIX& BF0, const MATRIX& BF, const VECTOR& dof,
      const VECTOR& ddof, TENSOR2& F, TENSOR2& dF, TENSOR2& delta_F);
    virtual void compute_BEt_S(const MATRIX& BE, const TENSOR2& pk2, VECTOR& BEt_S);
    virtual void compute_BEt_D_BE(const MATRIX& BE, const MATRIX& D, MATRIX& BEt_D_BE);
    virtual void compute_NLSM(const MATRIX&, const TENSOR2&, MATRIX&);
    TENSOR4 transport_D(TENSOR4, const TENSOR2&);
    TENSOR2 transport_cauchy_to_PK(const TENSOR2&, const TENSOR2&);
    TENSOR2 transport_PK_to_cauchy(const TENSOR2&, const TENSOR2&);
...
 public :
    virtual INTEGRATION_RESULT* internal_reaction(bool,VECTOR&,SMATRIX&,bool);
...
};
```

MCE_FINITE_STRAIN element (2/5)

```
INTEGRATION_RESULT* MCE_FINITE_STRAIN::internal_reaction(bool if_compute_stiffness, VECTOR& resi,
   SMATRIX& stiff,bool only_get_tg_matrix)
{  INTEGRATION_RESULT* ret=NULL;
   MATERIAL_INTEGRATION_INFO mi_info(Time_ini,Dtime);
   mi_info.flags=0;
   int tsz=TENSOR2::give_symmetric_size(space_dimension()),
       utsz=TENSOR2::give_nonsymmetric_size(space_dimension());

   SMATRIX& tg_matrix=mi_info.tg_matrix;
   MATRIX elem_coord0(space_dimension(),nb_node());      // based on orig. positions
   MATRIX elem_coord(space_dimension(),nb_node());       // based on rearranged mesh
   VECTOR elem_displ(nb_dof());
   VECTOR elem_ddispl(nb_dof());

   get_elem_coord(elem_coord0,FALSE);   // for grad_f calculations
   get_elem_d_dof_tot(elem_displ);      //
   get_elem_d_dof_incr(elem_ddispl);    //

   MATRIX dshape_dx0(space_dimension(),nb_node());
   MATRIX dshape_dx(space_dimension(),nb_node());

   resi=0.0;
   if(if_compute_stiffness) { stiff=0.0; mi_info.flags|=BEHAVIOR::CALC_TG_MATRIX; }
   if(!only_get_tg_matrix)  mi_info.flags|=BEHAVIOR::GIVE_FLUX;

   MATRIX BE0, BF0(utsz,nb_dof()), BF0t(nb_dof(),utsz);
   MATRIX BE,  BF(utsz,nb_dof()),  BFt(nb_dof(),utsz);
   MATRIX Be(tsz,nb_dof());
   TENSOR2 delta_F, //   dXtp/dXt   nb. F = dXtp/dXo
           Fini;    //   dXt/dX0

   move_coord(elem_displ,elem_ddispl,elem_coord0,elem_coord);   // node coords according to lagrange_type
```

MCE_FINITE_STRAIN element (3/5)

```
for (start(elem_coord);ok();next(elem_coord)) {
    MAT_DATA& mdat = pub_mat_data();
    pub_behavior()->attach_all(mdat);
    TENSOR2_GRAD& F   = *(TENSOR2_GRAD*)pub_behavior()->get_grad_var("F");
    TENSOR2_FLUX& sig = *(TENSOR2_FLUX*)pub_behavior()->get_flux_var("sig");
    mi_info.delta_grad.resize(F.var_size);
    TENSOR2 dF(F.var_size,mi_info.delta_grad,0);

    compute_dshape_dx(elem_coord0,dshape_dx0);
    compute_BF(dshape_dx0,BF0);
    if(lagrange_type==TOTAL) {
        compute_Be(dshape_dx0,Be);
        compute_BE(elem_displ,BF0,Be,BE0);
    }
    compute_dshape_dx(elem_coord,dshape_dx);
    compute_BF(dshape_dx,BF);    // F = 1 + BF*U
    if(lagrange_type==UPDATED || lagrange_type==FINAL) {
        compute_Be(dshape_dx,Be);
        if(lagrange_type==UPDATED) compute_BE(elem_ddispl,BF,Be,BE);
        else {
              VECTOR d0(!elem_ddispl); d0=0.0; compute_BE(d0,BF,Be,BE);
        }
    }
    compute_F(BF0,BF,elem_displ,elem_ddispl,F,dF,delta_F);
    if (F.determin()<=0.0) {
        static INTEGRATION_RESULT  Fneg_err(INTEGRATION_RESULT::DIVERGENCE);
        Out << "det(F)<0 in Mce_finite_strain J="<<F.determin()
            << "   element: "<<itoa(id)<<endl;
        return &Fneg_err;
    }
    rotate_to_material(F);
    rotate_to_material(dF);

    if(ret=pub_behavior()->integrate(mi_info)) return ret;
```

MCE_FINITE_STRAIN element (4/5)

```
BEHAVIOR::STRESS_MEASURE stress_measure = pub_behavior()->get_stress_measure();
rotate_from_material(sig);
rotate_from_material(F);
rotate_from_material(dF);

TENSOR2 pk2;
if(lagrange_type==UPDATED) {
    if (stress_measure==BEHAVIOR::CAUCHY) {
        pk2 = transport_cauchy_to_PK(delta_F,sig);
    } else if(stress_measure==BEHAVIOR::PK2_t)  {
        ERROR("Mixed problem of Updated element/PK2 stress not implemented");
    } else if(stress_measure==BEHAVIOR::PK2_0)  {
        TENSOR2 pk0 = sig;
        sig = transport_PK_to_cauchy(F,pk0);
        pk2 = transport_PK_to_cauchy(F-dF,pk0);
        ERROR("Mixed problem of Updated element/PK2_0 stress not implemented");
    }
} else if(lagrange_type==TOTAL) {
    if (stress_measure==BEHAVIOR::CAUCHY)    pk2 = transport_cauchy_to_PK(F,sig);
    else if(stress_measure==BEHAVIOR::PK2_t) Assert(0);
    else if(stress_measure==BEHAVIOR::PK2_0)  {
        pk2 = sig;
        sig = transport_PK_to_cauchy(F,pk2);
    }
} else if(lagrange_type==FINAL) {
    if  (stress_measure==BEHAVIOR::CAUCHY)   pk2 = sig;
    else if(stress_measure==BEHAVIOR::PK2_t) Assert(0);
    else if(stress_measure==BEHAVIOR::PK2_0) Assert(0);
}
VECTOR BEt_S;
if(lagrange_type==UPDATED)    compute_BEt_S(BE,pk2,BEt_S);
else if(lagrange_type==TOTAL) compute_BEt_S(BE0,pk2,BEt_S);
else if(lagrange_type==FINAL) compute_BEt_S(Be,pk2,BEt_S);
```

MCE_FINITE_STRAIN element (5/5)

```
        double dv;
        bool if_modV = modif_dV(dv,elem_displ,elem_ddispl);
        if(if_modV) BEt_S*=dv;
        integrate(BEt_S,resi);

        if (if_compute_stiffness) {
             TENSOR4 D = tg_matrix;
             rotate_from_material(D);
             SMATRIX NLSM; SMATRIX BEt_D_BE;
             if (lagrange_type==UPDATED) {
               if(stress_measure==BEHAVIOR::CAUCHY)     transport_D(D,(delta_F));
               else if(stress_measure==BEHAVIOR::PK2_0) transport_D(D,F);
               compute_BEt_D_BE(BE,D,BEt_D_BE);
               compute_NLSM(BF,pk2,NLSM);
             }
             else if (lagrange_type==FINAL) {
                compute_BEt_D_BE(BE,D,BEt_D_BE);
                compute_NLSM(BF,pk2,NLSM);
             }
             else if (lagrange_type==TOTAL) {
               if(stress_measure==BEHAVIOR::CAUCHY)     transport_D(D,F);
               else if(stress_measure==BEHAVIOR::PK2_t) transport_D(D, inverse(F));
               compute_BEt_D_BE(BE0,D,BEt_D_BE);
               compute_NLSM(BF0,pk2,NLSM);
             }
             integrate(BEt_D_BE,stiff);
             integrate(NLSM,stiff);
        }
    }
    return ret;
}
```

Algorithms

- class PROBLEM (eg. static, dynamic, thermal...)
- class ALGORITHM (eg. newton)
- class GLOBAL_MATRIX (linear system solver)

Algorithm example: Dynamic regularization

Newmark implicit time discretization:

$$X^{t+\Delta t} = X^t + \Delta t\, \dot{X}^t + \Delta t^2 \left[\left(\tfrac{1}{2} - \alpha\right) \ddot{X}^t + \alpha\, \ddot{X}^{t+\Delta t} \right]$$

$$\dot{X}^{t+\Delta t} = \dot{X}^t + \Delta t \left[(1-\beta)\, \ddot{X}^t + \beta\, \ddot{X}^{t+\Delta t} \right]$$

stability for: $\beta \geq 0.5$ and $\alpha \geq 0.25(0.5+\beta)^2$

tangent matrix: $K_d = K + \frac{1}{\alpha \Delta t^2} M$ (without damping)

let $A = \frac{1}{\alpha \Delta t^2} M$ and $B = A^{-1} K$

$$K_d = A(I - B) \quad , \quad K_d^{-1} = (I-B)^{-1} A^{-1}$$

for small time steps Δt: $\rho(B) < 1$ and $(I-B)^{-1} \approx (I+B)$

$$K_d^{-1} \approx \alpha \Delta t^2 \left[M^{-1} - \alpha\, \Delta t^2\, M^{-1} K\, M^{-1} \right]$$

a direct calculation of a pseudo-inverse of K_d can be performed

Dynamic regularization

- reset
- edit source code:
  - Dynamic_regularization_problem.h
  - Dynamic_regularization_problem.c
  - Dynamic_regularization_algorithm.c
  - Dynamic_regularization.h Dynamic_regularization.c
  - Dynamic_regularization_auto_time.c
- Zmake
- edit input file ball.inp
- change algorithm: classic dynamic regularization
- run calculation: Zrun ball
- view results with Zmaster: Zmaster ball
- open terminal

Notes

syntax of ADD_SINGLE_CMD_TO_MODIF_REC macro:

`ADD_SINGLE_CMD_TO_MODIF_REC( keyword, member_data)`

- where:
  - *keyword* is the name of the command in the input file
  - *member_data* is the data that will be initialized from the value read after *keyword* in the input file
- function:
  - automatic handling of the correct * level
  - automatic verification of type for the values read compared to the one of the member data
  - automatic generation of graphics dialog in Zmaster

back

Notes

- note the difference between **UTILITY_NODE::id** and **UTILITY_NODE::rank**
  - *id* is the node number in the geof file (numbering can be discontinuous)
  - *rank* is the position of the node in the UTILITY_MESH list of nodes

back

Notes

- note that the **Zfstream** class automatically does a proper encoding for binary IO, depending on the type of data given in arguments of the **read()**, **write()** functions.

  This allows cross-platform use of the same input/output binary files by Zébulon or Zmaster.

  However, because of that, depending on the architecture those binary files cannot be managed by basic C/FORTRAN IO subroutines.

back

Notes: Local post compute() function

```
virtual void compute(const ARRAY<VECTOR>& in,ARRAY<VECTOR>& out)
{
// This function is called for each integration point/node
// * in is an array whose size is the number of cards selected in
// the input calculation
//     for each card in[ic] is a vector whose size is the number of data
//       specified with function input_i_need()
// * out is an array with size equal to in
//     for each card out[ic] is a vector whose size is the number of data
//       specified with function output_i_give()

   for(int ic=0;ic<!in;ic++) {  // Loop on input cards
     for(int iv=0;iv<!in[ic];iv++) { // Loop on input variables for card ic
        double val=in[ic][iv];        // To get input variables
        ...
     }
     for(int jv=0;jv<!out[ic];jv++) { // Loop on output variables for card ic
        out[ic][jv]= ...;              // To set output variables
        ...
     }
   }
}
```

back

Notes: Global post compute() function (1/3)

```
virtual void compute(ARRAY<POST_ELEMENT*>& ape,ARRAY<POST_NODE*>& apn,
  const ARRAY<int>& nip)
{     // This function is called for each results card selected
// 1. Post-processing on integ points : **file integ
   if(location==G_INTEG) {
     for(int ie=0;ie<!ape;ie++) { // Loop on elements selected with the **elset function
        POST_ELEMENT* pe=ape[ie]; int ip=0;
        for(pe->start();pe->ok();pe->next()) { // Loop on integ points

           INTEG_DATA& id=pe->idata[ip][0];    // Integ data for current ip
//         INTEG_DATA::data is a VECTOR whose size is the number of data
//           specified with the input_i_need() function
//         INTEG_DATA::out is a VECTOR whose size is the number of data
//           specified with the output_i_give() function

           for(int iv=0;iv<!id.data;iv++) { // Loop on input variables
             double val=id.data[iv]; // To get input variable
           }

           for(int jv=0;jv<!id.out;jv++) { // Loop on ouput variables
             id.out[jv]=...; // To set output variables
           }
        }
     }
   }
```

Notes: Global post compute() function (2/3)

```
// 2. Post-processing on ctmat or ctele files
   if(location==G_CTELE||location==G_CTMAT) {
     for(int ie=0;ie<!ape;ie++) { // Loop on nodes selected with the **elset function
        POST_ELEMENT* pe=ape[ie];
        for(int in=0;in<!pe->ndata;in++) { // Loop on element nodes

          INTEG_DATA& id=pe->ndata[in][0]; // Integ data for current node
//         INTEG_DATA::data is a VECTOR whose size is the number of data
//           specified with the input_i_need() function
//         INTEG_DATA::out is a VECTOR whose size is the number of data
//           specified with the output_i_give() function

           for(int iv=0;iv<!id.data;iv++) { // Loop on input variables
             double val=id.data[iv]; // To get input variable
           }
           for(int jv=0;jv<!id.out;jv++) { // Loop on ouput variables
             id.out[jv]=...; // To set output variables
           }
        }
     }
   }
```

Notes: Global post compute() function (3/3)

```
// 3. Post-processing on nodes: **file node or **file ctnod
   if(location==G_NODE||location==G_CTNOD) {
     for(int in=0;in<!apn;in++) { // Loop on nodes selected with the **nset function
        POST_NODE* pn=apn[in];

        NODE_DATA& nd=pn->data[0];
//      NODE_DATA::data is a VECTOR whose size is the number of data
//        specified with the input_i_need() function
//      NODE_DATA::out is a VECTOR whose size is the number of data
//        specified with the output_i_give() function

        for(int iv=0;iv<!nd.data;iv++) { // Loop on input variables
             double val=nd.data[iv]; // To get input variable
        }
        for(int jv=0;jv<!nd.out;jv++) { // Loop on ouput variables
          nd.out[jv]=...; // To set output variables
        }
     }
   }
}
```

back

Notes: Tensor 2 objects storage

Storage of tensor2 objects in a vector (Voigt notation)

- 2D symmetric: $\left\{ t_{11}, t_{22}, t_{33}, \sqrt{2} t_{12} \right\}$
- 2D non-symmetric: $\{ t_{11}, t_{22}, t_{33}, t_{12}, t_{21} \}$
- 3D symmetric: $\left\{ t_{11}, t_{22}, t_{33}, \sqrt{2} t_{12}, \sqrt{2} t_{23}, \sqrt{2} t_{31} \right\}$
- 3D symmetric: $\{ t_{11}, t_{22}, t_{33}, t_{12}, t_{23}, t_{31}, t_{21}, t_{32}, t_{13} \}$

back

Notes: Incremental consistent tangent matrix

After convergence,

$$\begin{pmatrix} d\Delta\underset{\sim}{\varepsilon} \\ 0 \end{pmatrix} = [\mathrm{J}] \begin{pmatrix} d\Delta\underset{\sim}{\varepsilon}^e \\ d\Delta\alpha_I \end{pmatrix} \ldots \text{then} \begin{pmatrix} d\Delta\underset{\sim}{\varepsilon}^e \\ d\Delta\alpha_I \end{pmatrix} = [\mathrm{J}]^{-1} \begin{pmatrix} d\Delta\underset{\sim}{\varepsilon} \\ 0 \end{pmatrix}$$

$$[\mathrm{J}]^{-1} = \left(\begin{array}{c|c} H & x \\ \hline x & x \end{array}\right), \text{ with } H = \frac{\partial \Delta\underset{\sim}{\varepsilon}^e}{\partial \Delta\underset{\sim}{\varepsilon}}$$

Consistent tangent matrix:

$$\underset{\approx}{L}_c = \frac{\partial \Delta\underset{\sim}{\sigma}}{\partial \Delta\underset{\sim}{\varepsilon}^e} : \frac{\partial \Delta\underset{\sim}{\varepsilon}^e}{\partial \Delta\underset{\sim}{\varepsilon}} = \underset{\approx}{\Lambda} : H$$

back