AN OBJECT-ORIENTED PROGRAMMING APPROACH TO THE LAGRANGIAN FEM ANALYSIS OF LARGE INELASTIC DEFORMATIONS AND METAL-FORMING PROCESSES

NICHOLAS ZABARAS * AND AKKARAM SRIKANTH

Sibley School of Mechanical and Aerospace Engineering, 188 Frank H. T. Rhodes Hall, Cornell University, Ithaca, NY 14853-3801, U.S.A.

SUMMARY

The general deformation problem with material and geometric non-linearities is typically divided into a number of subproblems including the kinematic, the constitutive, and the contact/friction subproblems. These problems are introduced for algorithmic purposes; however, each of them represents distinct physical aspects of the deformation process. For each of these subproblems, several well-established mathematical and numerical models based on the finite element method have been proposed for their solution.

Recent developments in software engineering and in the field of object-oriented C++ programming have made it possible to model physical processes and mechanisms more expressively than ever before. In particular, the various subproblems and computational models in a large inelastic deformation analysis can be implemented using appropriate hierarchies of classes that accurately represent their underlying physical, mathematical and/or geometric structures.

This paper addresses such issues and demonstrates that an approach to deformation processing using classes, inheritance and virtual functions allows a very fast and robust implementation and testing of various physical processes and computational algorithms. Here, specific ideas are provided for the development of an object-oriented C++ programming approach to the FEM analysis of large inelastic deformations. It is shown that the maintainability, generality, expandability, and code re-usability of such FEM codes are highly improved. Finally, the efficiency and accuracy of an object-oriented programming approach to the analysis of large inelastic deformations are investigated using a number of benchmark metal-forming examples. Copyright © 1999 John Wiley & Sons, Ltd.

KEY WORDS: object-oriented programming; metal-forming processes; large deformations; Lagrangian analysis; plasticity

1. INTRODUCTION

Analysis of large deformation inelastic problems is extremely important in metal-forming processes where the deformation results in a high amount of plastic flow. Lagrangian formulations have been used extensively for the solution of such non-linear deformation problems. The major limitation of Lagrangian large deformation formulations with a fixed mesh is that the finite elements may...
become severely distorted. However, Lagrangian formulations provide no difficulties in dealing with the calculation of residual stresses, with modelling of contact between deformable bodies or with the advection of material state histories. An extensive review of Lagrangian algorithms can be found in References 1–11.

This research field has reached a rather mature level and a number of computational algorithms has already been developed for the Lagrangian analysis of large deformation problems, namely the constitutive, kinematic and contact subproblems. Full Newton–Raphson linearizations of the principle of virtual work have been developed and implemented in a number of research and commercial codes. The various implementations lead to different tangent stiffness matrices based on the selected stress and strain measures and the resulting linearizations corresponding to the various constitutive models [2, 3, 5, 9, 11]. In an appropriate kinematic framework for inelastic analysis, the deformation gradient is decomposed by means of Lee’s multiplicative decomposition into plastic and elastic parts [12, 13].

Several inelastic constitutive models have been proposed including those in References 14–16. These models share a common state variable based structure and are integrated with a number of objective integration algorithms like radial return mappings or other projection methods [3, 4, 13, 17–21]. As part of the solution methodology of the incremental constitutive problem, the consistent linearized material moduli are also computed.

Finally, a number of algorithms exist for the analysis of multidimensional contact problems. Most of these algorithms are explicit in nature, but implicit contact algorithms have been considered as well. In particular, the augmented Lagrangian analysis of Simo and Laursen is considered appropriate for a fully implicit Lagrangian analysis of large deformation problems [22–26]. Radial return like mappings have been proposed for the integration of these contact models and the contact contributions to the tangent stiffness have been calculated as required in the Newton–Raphson iterations for the calculation of the non-linear kinematics.

In this work, the analysis of large hyperelastic–viscoplastic deformations is considered and an object-oriented framework is introduced for their implementation. The fundamentals of object-oriented C++ programming for scientific computations are reviewed in Reference 27. Object-oriented calculations have received extensive attention in computational mathematics and in particular in fluid dynamics [28–34]. A number of engineering applications have already been published in this and other computational Journals [35–37].

The main objective of this paper is to provide a number of specific ideas for the development of hierarchies of classes that are appropriate for the analysis of large deformation plasticity problems including forming processes. Such problems are amenable to an object-oriented environment in a non-trivial manner since the underlying material, kinematic and contact mechanisms are coupled in a highly implicit manner. To allow ourselves to concentrate on the development of deformation processing related classes, the present computations have been adapted to the widely used diffpack library of C++ classes [38]. However, most of the discussion here is general and can be implemented using other object-oriented libraries. The diffpack libraries include a variety of classes for linear algebra operations, a number of FEM-based classes for element definition, stiffness development, assembly and many other [39–41]. In addition, diffpack provides a secure way for pointer declarations as well as a framework for the development of fields to represent the various continuous variables [39]. Familiarity with the structure of the diffpack libraries is useful but not essential in understanding the present developments.

To the best of our knowledge this is the first time that a fully object-oriented Lagrangian non-linear deformation analysis has been implemented. Although examples presented here are for
particular numerical algorithms and material constitutive laws, the development of classes has been performed keeping in mind the general structure of the deformation problem. The objective is to develop a framework that allows code expandability (inclusion of other constitutive models, algorithms, etc.) without a significant programming effort. In addition, implementation of various applications (e.g. of an extrusion or a forging process) is performed at an object level that does not require the user to have a knowledge of the details of the various algorithms. The examples presented here are also obtained with a combination of the stiffness tangent and contact tangent operators that has not been reported earlier in the literature. These calculations include the implementation of the strain projection method presented in Reference 5 to avoid element locking in the fully plastic region.

The structure of this paper is as follows. At first, a review of the definition of an inelastic large deformation problem is presented. The particular constitutive structure proposed by Anand and colleagues is adopted as a reference framework for the following presentation and numerical examples [2, 3]. However, the general structure of the various parts of the large inelastic deformation analysis is also identified in order to motivate the definition of the various class hierarchies. Then, the basic elements of a Lagrangian FEM analysis are reviewed, namely the analysis of the kinematic, constitutive and contact subproblems. Following the problem definition, the general structure of the ‘Simulator’ class is introduced. The various class hierarchies for the analysis of the constitutive, kinematic and contact subproblems are presented. The paper concludes with the analysis of a number of benchmark test forming problems and a preliminary discussion on the accuracy and cpu time requirements of the proposed object-oriented simulations.

2. REVIEW OF THE BASIC ELEMENTS OF A NON-LINEAR DEFORMATION ANALYSIS

2.1. Kinematic and constitutive equations

Let us consider the motion of a body occupying the configuration $\mathbf{B}_0$ at time $t = 0$ under the action of external forces. The motion of the body is represented by a smooth mapping $\phi(\cdot, \cdot)$ such that, at any given time $t$, the location $\mathbf{x}$ of a material point $\mathbf{p}$, which at time $t = 0$ occupied the position $\mathbf{X}$, is given by

$$\mathbf{x} = \phi(\mathbf{X}, t)$$

The deformation gradient $\mathbf{F}$ with respect to the reference configuration $\mathbf{B}_0$ is given by

$$\mathbf{F} = \nabla \phi(\mathbf{X}, t) = \frac{\partial \phi(\mathbf{X}, t)}{\partial \mathbf{X}}$$

It is required that $\det \mathbf{F} > 0$ for the motion of the body to be meaningful. The total deformation gradients at times $t_n$ and $t_{n+1}$ are $\mathbf{F}_n$ and $\mathbf{F}_{n+1}$, respectively, as shown in Figure 1. In a Lagrangian analysis, the motion of each particle is followed as the particle occupies different configurations in the space at different times, and the deformation gradients are computed with respect to a reference configuration $\mathbf{B}_0$. In the case of the updated Lagrangian analysis, the reference configuration is
updated at each time step. Thus, the reference configuration at time \( t_{n+1} \), will be \( B_n \), whereas the current configuration becomes \( B_{n+1} \). The relative deformation gradient \( F_r \) is defined as

\[
F_r = F_{n+1}(F_n)^{-1}
\]  

(3)

In this work, the processes are quasi-static and the body is under equilibrium at all times. Let \( T \) be the Cauchy stress and \( b \), the body force field defined on the current configuration \( B_{n+1} \). Then, at a time \( t_{n+1} \), the equilibrium equations can be written as

\[
\nabla_{n+1} \cdot T + b = 0 \quad \forall x \in B_{n+1}
\]  

(4)

The subscript \((n + 1)\) indicates that the divergence of \( T \) is defined in the current configuration \( B_{n+1} \). The equilibrium equations can also be expressed in the reference configuration \( B_n \) as

\[
\nabla_n \cdot P_r + f_r = 0 \quad \forall x \in B_n
\]  

(5)

where the Piola–Kirchhoff I stress \( P_r \) and body force \( f_r \) are obtained as,

\[
P_r = \det F_r TF_r^{-T}
\]

\[
f_r = \det F_r b
\]  

(6)

In order to solve the equilibrium equations, certain boundary conditions have to be specified at all times on the boundaries of \( B \). In addition, the constitutive relationship between the Cauchy stress \( T \) and the deformation gradient \( F \) should be specified.

The total deformation gradient \( F \) at any time \( t \) is assumed to be decomposed as

\[
F = F^e \tilde{F}^p
\]  

(7)

where \( F^e \) is the elastic deformation gradient and \( \tilde{F}^p \) is the plastic deformation gradient (Figure 2). It is further assumed that the plastic deformation is isochoric, and hence,

\[
\det \tilde{F}^p = 1
\]

\[
\det F^e > 0
\]  

(8)
Following Anand [15], the hyperelastic constitutive equations are written as

$$\hat{\mathbf{T}} = \mathcal{L}^e[\hat{\mathbf{E}}^e]$$  \hspace{1cm} (9)

where the strain measure, $\mathbf{E}^e$, is defined with respect to the intermediate (unstressed) configuration as

$$\mathbf{E}^e = \ln \mathbf{U}^e$$ \hspace{1cm} (10)

The corresponding conjugate stress measure $\hat{\mathbf{T}}$ is the pullback of the Kirchhoff stress with respect to $\mathbf{R}^e$,

$$\hat{\mathbf{T}} = \det(\mathbf{U}^e)\mathbf{R}^e^T\mathbf{T}\mathbf{R}^e$$ \hspace{1cm} (11)

Here $\mathbf{U}^e$ and $\mathbf{R}^e$ are calculated from the polar decomposition of $\mathbf{F}^e$,

$$\mathbf{F}^e = \mathbf{R}^e\mathbf{U}^e$$ \hspace{1cm} (12)

For an isotropic material, the elastic moduli $\mathcal{L}^e$ are given by

$$\mathcal{L}^e = 2\mu\mathcal{J} + (\kappa - \frac{2}{3}\mu)\mathbf{I} \otimes \mathbf{I}$$ \hspace{1cm} (13)

where $\mu$ is the shear modulus, $\kappa$ is the bulk modulus and $\mathbf{I}$ and $\mathcal{J}$ denote the unit second- and fourth-order tensors, respectively.

Remark 1. Other hyperelastic models can also be considered (e.g. see Reference 5). The main differences between the various elasticity models are essentially in the selection of work-conjugate stress and elastic strain measures.

The most commonly used phenomenological model for elastic–plastic deformations is the classical $J_2$ flow theory with isotropic hardening. The flow rule for the evolution of $\tilde{\mathbf{F}}^p$ is here given as follows [3]:

$$\tilde{\mathbf{L}}^p = \tilde{\mathbf{F}}^p(\tilde{\mathbf{F}}^p)^{-1} = \sqrt{\frac{3}{2}}\tilde{\varepsilon}^p\tilde{\mathbf{N}}^p(\mathbf{T}, \tilde{\sigma})$$ \hspace{1cm} (14)
where
\[
\bar{N}^p(T', \tilde{\sigma}) = \sqrt{\frac{3}{2}} \frac{T'}{\tilde{\sigma}}
\]  
(15)
\[
\bar{T}' = \bar{T} - \frac{\text{tr} \bar{T}}{3} \mathbf{I}
\]  
(16)
and
\[
\tilde{\sigma} = \sqrt{\frac{3}{2}} \bar{T}' \cdot \bar{T}'
\]  
(17)

An assumption is made that an intermediate configuration can be chosen in case of isotropic materials such that the skew symmetric part of the velocity gradient (spin) vanishes identically. This results in
\[
\bar{D}^p = \bar{L}^p
\]
\[
\bar{W}^p = 0
\]  
(18)

The evolution of the equivalent plastic strain $\dot{\bar{\varepsilon}}^p$ is specified via uniaxial experiments as
\[
\dot{\bar{\varepsilon}}^p = f(\tilde{\sigma}, s)
\]  
(19)
and the evolution of the isotropic isothermal scalar resistance $s$ is also obtained from experiments in the form,
\[
\dot{s} = g(\tilde{\sigma}, s) = h(\tilde{\sigma}, s) \dot{\bar{\varepsilon}}^p - \dot{r}(s)
\]  
(20)
where $\dot{r}(s)$ is the static recovery function.

**Remark 2.** Alternative forms of the flow rule can also be considered (e.g. see References 5 and 8). However, most isotropic $J_2$ flow rules fit in a functional structure similar to that of the model above.

**Remark 3.** Rate-independent elasto-plasticity models can be described as well. The main difference is in the calculation of the equivalent plastic strain rate $\dot{\bar{\varepsilon}}^p$, which for rate-independent models is defined implicitly using the consistency condition (i.e. from $\tilde{\sigma} = \dot{s}$) [1, 2, 13].

### 2.2. The initial and the boundary value problem

The kinematic and the constitutive equations, together with the equilibrium equations and the initial and the boundary conditions, completely describe the motion of the body. In this section, a brief presentation is given of a numerical solution procedure to solve this boundary value problem. The problem is divided into three subproblems: (1) the constitutive problem, (2) the contact/friction problem and (3) the kinematic problem. In the constitutive incremental problem, one determines the triad $(\mathbf{T}, s, \bar{F}^p)$ at the end of the time step (time $t_{n+1}$) given the body configurations $\mathbf{B}_n$ and $\mathbf{B}_{n+1}$ as well as the triad $(\mathbf{T}, s, \bar{F}^p)$ at time $t_n$. In the contact/friction subproblem, given the configuration $\mathbf{B}_{n+1}$, the die location and shape at time $t_{n+1}$, as well as estimates of the contact tractions (e.g. from the previous time step or a previous Newton–Raphson iteration), one updates the regions of
contact as well as the applied contact tractions at $t_{n+1}$. The contact subproblem is directly coupled with the kinematic incremental problem where given the triad $(T_{n+1}, s_{n+1}, \tilde{F}_n^p)$, the configuration $B_n$ and the applied boundary conditions at $t_{n+1}$ (including the contact tractions), one calculates (updates) the configuration $B_{n+1}$.

2.2.1. Time integration of the constitutive model. In the following, a review is given of a radial return algorithm that calculates $(T_{n+1}, s_{n+1}, \tilde{F}_n^p)$, given the body configurations at time $t_n$ and $t_{n+1}$, and knowing $(T_n, s_n, \tilde{F}_n^p)$. The detailed algorithm is provided in References 3 and 9.

Employing the Euler backward integration to solve the differential equations (14) and (20), leads to the following:

$$\tilde{F}_n^p = \exp((\Delta t \tilde{D}_{n+1}^b(T_{n+1}, s_{n+1})))\tilde{F}_n^p$$

and

$$s_{n+1} - s_n - \Delta t g(\tilde{\sigma}_{n+1}, s_{n+1}) = 0$$

To solve these equations, a trial elastic deformation gradient $F^e_n$ is introduced as follows:

$$F^e_n = F_e F^e_n = R^e_n U^e_n$$

where $F_e$ is defined in equation (3). The trial stress is defined as follows:

$$\tilde{T}_n = \mathcal{L}^e([\tilde{F}^e_n]) = \mathcal{L}^e([\ln U^e_n])$$

The equivalent stress $\tilde{\sigma}_n$ is defined from the deviatoric part of the trial stress as,

$$\tilde{\sigma}_n = \sqrt{\frac{3}{2} \tilde{T}_n \cdot \tilde{T}_n}$$

and the pressure $\tilde{p}_n$ as

$$\tilde{p}_n = -\frac{1}{3} \text{tr}(\tilde{T}_n)$$

It can easily be shown that the tensors $\tilde{T}_n$ and $\tilde{T}_{n+1}$ are in the same direction, i.e [3].

$$\tilde{N}^b_{n+1} = \sqrt{\frac{3}{2} \tilde{T}_{n+1}}$$

It can be shown that the only scalar quantities $\tilde{\sigma}_{n+1}$, $\tilde{p}_{n+1}$ and $s_{n+1}$ need to be evaluated. One can show the following [3, 9]:

$$\tilde{p}_{n+1} = \tilde{p}_n$$

and

$$\tilde{\sigma}_{n+1} - \tilde{\sigma}_n + 3\mu \Delta tf(\tilde{\sigma}_{n+1}, s_{n+1}) = 0$$

Solving the non-linear, scalar, algebraic equations (29) and (22) together, one can obtain the values of $\tilde{\sigma}_{n+1}$ and $s_{n+1}$ (see the Appendix of Reference 2). From the values of $\tilde{\sigma}_{n+1}$, $\tilde{p}_{n+1}$ and $s_{n+1}$, one can calculate $T_{n+1}$ and $F^e_{n+1}$ as follows [3]:

$$T_{n+1} = \exp \left( \frac{\tilde{p}_n}{K} \right) R^e_{n+1} \tilde{T}_{n+1}$$

Copyright © 1999 John Wiley & Sons, Ltd.

and

\[ F_{n+1}^e = R_e \left( \sum \frac{\tilde{\lambda}_e^n}{\tilde{\lambda}_e^n \tilde{\lambda}_e^{n+1} \tilde{\lambda}_e^{n+1} \tilde{\lambda}_e^{n+1}} \epsilon^e \otimes \epsilon^e \right) \]  

(31)

where the radial return factor \( \eta_{n+1} = \tilde{\sigma}_{n+1}/\tilde{\sigma}_* \), and

\[ \tilde{T}_{n+1} = \eta_{n+1} \tilde{T}_* - \tilde{p}_* I \]  

(32)

Here, \( \lambda_e \) and \( \epsilon^e \) denote the eigenvalues and eigenvectors, respectively of \( U_e^* \).

**Remark 4.** Note that the developments above are quite general. For isotropic plasticity, in particular, the radial return mapping always leads to two non-linear algebraic equations similar to equations (22) and (29).

### 2.2.2 Boundary conditions.

In the following, only the contact/frictional boundary conditions are considered as other types of boundary conditions are explicit in nature and can be applied directly for the solution of the non-linear kinematics.

Contact and friction is modelled here following the scheme developed by Simo and Laursen [22–25]. The presentation here is limited to plane-strain and axisymmetric problems and dies are considered to be rigid. The die \( \mathcal{D} \) is parametrized in two dimensions using a parameter \( \xi \) and the functions \( y(\xi) = (y_1(\xi), y_2(\xi)) \), \( 0 \leq \xi \leq 1 \). A fixed right-handed basis \( (e_1, e_2, e_3) \) is defined, with \( e_2 \) going into the plane of the paper (as shown in Figure 3) and a convected basis \( (r, e_2, v) \) at each point defined by a particular value of \( \xi \). The tangent vector \( \tau_1 \) and the unit tangent vector \( r \) are given by

\[ \tau_1 = y_{\xi} = \frac{\partial y_1}{\partial \xi} e_1 + \frac{\partial y_2}{\partial \xi} e_3, \quad r = \frac{\tau_1}{\| \tau_1 \|} \]  

(33)

The unit normal vector \( \nu \) points towards the body (i.e. \( \nu = -n \), where \( n \) is the outward unit normal vector to the body) and is obtained as

\[ \nu = r \times e_2 \]  

(34)

where \( \times \) denotes vector cross product. Any vector \( a \) in the direction of \( \tau_1 \) is represented as

\[ a = a^1 \tau_1 = ar, \quad a^1 \| \tau_1 \| = a \]  

(35)
The die separates the space into ‘admissible’ and ‘inadmissible’ regions and the die is parametrized such that the normal vector \( \mathbf{v} \) is pointing into the admissible region. Then, one can define the gap function \( g \) of any point \( \mathbf{x} \) in space as the shortest distance of that point from the die (see Figure 3). Thus, one can write

\[
\mathbf{\tilde{y}} - \mathbf{x} = g(\mathbf{x})\mathbf{v}(\mathbf{\tilde{y}})
\]

(36)

where \( \mathbf{\tilde{y}} \in \mathcal{D} \) is the value of \( \mathbf{y} \) that minimizes the norm, \( \| \mathbf{x} - \mathbf{y} \| \). A unique value of the parameter \( \tilde{\xi} \) is associated to each \( \mathbf{\tilde{y}} \). With this definition of the gap function, the impenetrability constraints are as below: For all \( \mathbf{x}_n \in \mathcal{B}_n \), with \( \mathbf{x}_{n+1} = \mathbf{x}_n + \mathbf{u}(\mathbf{x}_n) \),

\[
g(\mathbf{x}_{n+1}) \leq 0
\]

(37)

\[
t_N = \mathbf{v} \cdot (\mathbf{p}_1 \mathbf{m})
\]

\[
t_N g(\mathbf{x}_{n+1}) = 0
\]

where \( t_N \) is the absolute value of the contact pressure. The vector \( \mathbf{p}_1 \mathbf{m} \) is the current traction per unit area in \( \partial \mathcal{B}_n \).

It is assumed that Coulomb friction exists in the contact region with a coefficient of friction \( \mu_T \). The tangential velocity of the workpiece relative to the die is written (using equation (33)) as

\[
\mathbf{v}_T = \frac{d}{dt} \{ \mathbf{\tilde{y}} \} = \mathbf{y}, (\tilde{\xi}) \tilde{\zeta} = \tilde{\xi} t_1
\]

(38)

This relative velocity \( \mathbf{v}_T \) makes sense only for points in contact with the die and moving along the die. The Coulomb friction law can be written as:

\[
\mathbf{t}_T = -\mathbf{p}_1 \mathbf{m} + t_N \mathbf{v}
\]

\[
\Phi := \| \mathbf{t}_T \| - \mu_T t_N \leq 0
\]

\[
\mathbf{v}_T = \zeta \frac{\mathbf{t}_T}{\| \mathbf{t}_T \|}
\]

\[
\zeta \Phi = 0
\]

(39)

Following Reference 22, the penalty parameters \( \varepsilon_N \) and \( \varepsilon_T \) are introduced as well as the Lagrangian multipliers \( \lambda_N \) and \( \lambda_T \). The normal traction is then expressed as follows:

\[
t_N = \langle \lambda_N + \varepsilon_N g \rangle
\]

(40)

where \( \langle \cdot \rangle \) is the Macauley bracket (i.e. \( \langle x \rangle = 1/2[|x| + x] \)). The frictional constraint is then written as follows:

\[
\mathbf{v}_T - \zeta \frac{\mathbf{t}_T}{\| \mathbf{t}_T \|} = \frac{1}{\varepsilon_T} \left( \frac{\nabla}{\| \mathbf{t}_T \|} \cdot \mathbf{t}_T - \lambda_T \right)
\]

(41)

where, for any vector \( \mathbf{a} \) in the direction \( \mathbf{\tau}_T \), \( \nabla \mathbf{a} \) is expressed as

\[
\nabla \mathbf{a} = \dot{\mathbf{a}}_1 \mathbf{\tau}_T, \quad \dot{\mathbf{a}}_1 = \frac{d}{dt} a_1
\]

Substituting the expression for \( \mathbf{v}_T \) from equation (38), equation (41) reduces to the following scalar equation:

\[
t_T^1 - \lambda_T^1 = \varepsilon_T \left( \zeta - \zeta \frac{t_T^1}{\| \mathbf{t}_T \|} \right)
\]

(42)
The above frictional constraint equation can be integrated over the time interval \([t_n, t_{n+1}]\) using the Euler backward integration scheme to obtain

\[
t_{T_{n+1}}^1 - t_{T_n}^1 = \lambda_{T_{n+1}}^1 - \lambda_{T_n}^1 + \varepsilon_T \left( \tilde{\zeta}_{n+1} - \tilde{\zeta}_n - (\tilde{\eta}_{n+1} - \tilde{\eta}_n) \frac{t_{T_{n+1}}^1}{\|t_{T_{n+1}}^1\|} \right) \tag{43}
\]

Further, the following constraint equation must be satisfied:

\[
\Phi_{n+1} := \|t_{T_{n+1}}^1\| - \mu f T_{N_{n+1}} \leq 0 \tag{44}
\]

A trial value for \(t_{T_{n+1}}^1\) is first introduced:

\[
t_{T_{n+1}}^{1(\text{trial})} := t_{T_n}^1 + \lambda_{T_{n+1}}^1 - \lambda_{T_n}^1 + \varepsilon_T (\tilde{\eta}_{n+1} - \tilde{\eta}_n) \tag{45}
\]

Then,

\[
\Phi_{n+1}^{(\text{trial})} := \|t_{T_{n+1}}^{1(\text{trial})}\| - \mu f T_{N_{n+1}} \tag{46}
\]

If \(\Phi_{n+1}^{(\text{trial})} < 0\), equation (39) would imply that the parameter \(\zeta = 0\), and therefore, the relative velocity of the point with respect to the die is zero. In other words, the point under consideration is ‘sticking’ to the die. In this case, \(t_{T_{n+1}}^1 = t_{T_{n+1}}^{1(\text{trial})}\). If, on the other hand, \(\Phi_{n+1}^{(\text{trial})} > 0\), the trial value violates the constraint equation (44). Therefore, the trial value must be modified so that the constraint equation is satisfied as an equality. This condition is called sliding. Thus, \(t_{T_{n+1}}^1\) is calculated as follows:

\[
t_{T_{n+1}}^1 = t_{T_{n+1}}^{1(\text{trial})} \quad \text{if} \quad \Phi_{n+1}^{(\text{trial})} \leq 0
\]

\[
= \mu f T_{N_{n+1}} \frac{t_{T_{n+1}}^{1(\text{trial})}}{\|t_{T_{n+1}}^{1(\text{trial})}\|} \quad \text{otherwise}
\tag{47}
\]

The contribution of the contact conditions to the tangent stiffness are briefly undertaken in the following section.

2.2.3. Principle of virtual work equation. Let us suppose that the configuration \(B_n\) at time \(t = t_n\) is known and is under equilibrium. Under the action of the external forces, the body deforms and occupies the configuration \(B_{n+1}\) at time \(t = t_{n+1}\). Equation (5) describes the equilibrium of the body at time \(t_{n+1}\) expressed in the updated reference configuration \(B_n\). Let \(\Gamma\) be the surface (\(\Gamma \subset \partial B_n\)) which corresponds to regions of the body that may potentially come into contact with the die.

The incremental quasi-static boundary value problem at time \(t = t_{n+1}\) is to find the incremental (with respect to the configuration \(B_n\)) displacement field \(u(x_n, t_{n+1}) \equiv u_{n+1}\) that will satisfy equation (5). The weak form of this equation is written as

\[
\tilde{G}(u_{n+1}, \eta) + G_c(u_{n+1}, \eta) = 0 \tag{48}
\]

for each test vector field \(\eta(x_n)\), which is compatible with the kinematic boundary conditions, where

\[
G(u_{n+1}, \eta) = \int_{B_n} \hat{P}_n \cdot \frac{\hat{\eta}}{\hat{x}_n} \, dV - \left( \int_{\partial B_{n+1}} \hat{\mathbf{n}} \eta \, dA + \int_{B_{n+1}} \hat{\mathbf{b}} \eta \, dV \right) \tag{49}
\]
and

$$G_c(u_{n+1}, \eta) = \int_{\Gamma} (-t_N v \bullet \eta + t_r \bullet \eta) \, dA \quad (50)$$

Equation (48) is a mixed form of the principle of virtual work. The internal work is expressed in the reference configuration $B_n$, whereas the external work is expressed in the current configuration where the applied surface tractions, $\hat{i}$, and body forces, $\hat{b}$, are given. The contact work is calculated in the reference configuration $B_n$.

To solve the non-linear equation (48) for $u(x_n, t_{n+1})$, a Newton–Raphson iterative scheme is employed. Let $u^{(k+1)}_{n+1}$ and $u^{(k)}_{n+1}$ be the displacement fields at the end of the $(k+1)$th step and the $k$th step, respectively, during the Newton–Raphson iterative process. Then, the linearized form of equation (48) is as follows:

$$\tilde{G}(u^{(k)}_{n+1}, \eta) + \frac{\partial \tilde{G}}{\partial u^{(k)}_{n+1}} (u^{(k+1)}_{n+1} - u^{(k)}_{n+1}) = 0 \quad (51)$$

$\tilde{G}$ consists of two terms, $G$ and $G_c$. In the following sections, the linearization of both terms is provided. The term $G$ contains the internal work and the external work done by the body forces and surface forces not including the contact work at the die/workpiece interface:

$$dG = \int_{B_n} dP_r \bullet \frac{\partial \eta}{\partial x_n} \, dV + d(\text{bodyforces + applied traction}) \quad (52)$$

The linearizations for the follower forces and body forces have been dealt with by Hibbit [42], and here only the linearizations of the internal work term are presented. The differential $dP_r$ is calculated as follows:

$$dP_r = d(\det F_r T_{n+1} F_r^{-T})$$

Complete linearization of this term leads to the following [3]:

$$dP_r = \det F_r \left\{ \text{tr}(dF_r F_r^{-1}) T_{n+1} - T_{n+1} (dF_r F_r^{-1})^T \right\}$$

$$- \text{tr} \left( \frac{1}{3\kappa} \mathbf{C} [d\mathbf{E}_e^s] \right) T_{n+1} + \exp \left( \frac{p_*}{\kappa} \right) \mathbf{C} [d\mathbf{E}_e^s] (\mathbf{E}_e^s)^T$$

$$+ (d\mathbf{R}_e^c (\mathbf{R}_e^c)^T) T_{n+1} - T_{n+1} (d\mathbf{R}_e^c (\mathbf{R}_e^c)^T)^T \} F_r^{-T} \quad (53)$$

$\mathbf{C}$ are the consistent linearized material moduli ($\mathbf{C} = \tilde{\mathbf{C}} \mathbf{E}_e^s \mathbf{T}$) given in Reference 3 in the following form:

$$\mathbf{C} = 2\mu' \mathbf{I} + (\kappa - \frac{2}{3} \mu') \mathbf{I} \otimes \mathbf{I} + 2\mu (c - \eta_{n+1}) \tilde{N}_{n+1}^p \otimes \tilde{N}_{n+1}^p \quad (54)$$

with $\tilde{N}_{n+1}^p$ given from equation (27),

$$\mu' = \eta_{n+1} \mu, \quad \eta_{n+1} = \frac{\tilde{\sigma}_{n+1}}{\tilde{\sigma}_s}$$

and

$$c = \frac{b_2}{a_1 b_2 + a_2 b_1}$$
\begin{align*}
a_1 & \equiv 1 + 3\mu \Delta t \frac{\partial f}{\partial \sigma_{n+1}}, \quad a_2 \equiv 3\mu \Delta t \frac{\partial f}{\partial \dot{\sigma}_{n+1}} \\
b_1 & \equiv \Delta t \frac{\partial g}{\partial \sigma_{n+1}}, \quad b_2 \equiv 1 - \Delta t \frac{\partial g}{\partial \dot{\sigma}_{n+1}}
\end{align*}

The linearization of the contact work is presented in Reference 22 and is summarized here for completeness. The term $G_c$ results in

\[ dG_c = \int_{\Gamma} (d(-t_N \mathbf{v}) \cdot \mathbf{e} + dt \mathbf{f} \cdot \mathbf{e}) \, dA \]

With differentiation of equations (40) and (47), one can show that

\[ \int_{\Gamma} d(-t_N \mathbf{v}) \cdot \mathbf{e} \, dA = \int_{\Gamma} \left( H(t_N) \varepsilon_{N}(\mathbf{v} \mathbf{d} \mathbf{u}) (\mathbf{v} \cdot \mathbf{e}) + \frac{t_N \rho}{1 + \rho} (\mathbf{r} \mathbf{d} \mathbf{u}) (\mathbf{r} \cdot \mathbf{e}) \right) \, dA \quad (55) \]

and

\[ \int_{\Gamma} dt \mathbf{f} \cdot \mathbf{e} \, dA = \int_{\Gamma} \frac{\Delta t}{1 + \frac{\rho}{g}} (\mathbf{t} \cdot \mathbf{e}) \, dA \\
+ \int_{\Gamma} \frac{t}{\| \tau \|^2 (1 + g\rho)^2} \left( (\mathbf{r} \cdot \mathbf{e}) (\mathbf{r} \mathbf{d} \mathbf{u}) (\mathbf{r} \cdot \mathbf{e}) (\mathbf{r} \cdot \mathbf{e}) + (\mathbf{r} \mathbf{d} \mathbf{u}) (\mathbf{r} \cdot \mathbf{e}) (\mathbf{r} \cdot \mathbf{e}) \right) \, dA \\
- \int_{\Gamma} \frac{t}{\| \tau \|^4 (1 + g\rho)^3} \left( (1 - 2g\rho) (\mathbf{r} \cdot \mathbf{e}) (\mathbf{r} \cdot \mathbf{e}) (\mathbf{r} \mathbf{d} \mathbf{u}) (\mathbf{r} \cdot \mathbf{e}) + g(\mathbf{r} \mathbf{d} \mathbf{u}) (\mathbf{r} \cdot \mathbf{e}) (\mathbf{r} \cdot \mathbf{e}) \right) \, dA \quad (56) \]

If the boundary is sticking (i.e. if $\Phi \leq 0$), then

\[ \Delta t^f = \frac{\varepsilon_f}{(1 + g\rho) \| \tau \|^2} (\mathbf{r} \mathbf{d} \mathbf{u}) \quad (57) \]

If sliding, then

\[ \Delta t^f = -\frac{\mu t_N \mathbf{e} (t^f)}{\| \tau \|^2} \left( \frac{\mathbf{r} \mathbf{d} \mathbf{u}}{\| \tau \|^2} (\mathbf{r} \cdot \mathbf{e}) \right) \quad (58) \]

where, in the above, $H(\cdot)$ is the heavyside function, equal to 1 when the argument is positive, and 0 otherwise. Also, the curvature $\rho$ is given by

\[ \rho = \frac{\mathbf{r} \cdot \mathbf{e}}{\| \tau \|^2} \quad (59) \]

A summary of the contact/friiction algorithm is provided in Section 2.5.

2.3. Convergence criteria and automatic time stepping

Two different criteria are currently employed to monitor the convergence of the Newton–Raphson iterations for the calculation of the incremental displacements. During each time increment, the incremental plastic work,

\[ \Delta W^p = \int_{t_n}^{t_{n+1}} \int_{\mathcal{B}_{n}} \frac{\partial \mathcal{G}^p}{\partial \dot{\mathbf{e}}} \, dV \, dt \]
is calculated. The Newton–Raphson iterations are assumed to have converged when the incremental plastic work in two successive iterations falls within a suitable tolerance (e.g. 0.001 per cent). Convergence is also monitored using the incremental displacement. Let \( d_i \) be the computed incremental displacements. Then, the displacement error \( R_d \) is defined using the \( L_\infty \) norm. A typical displacement-based convergence criterion is \( R_d < 10^{-5} \).

An automatic time-stepping algorithm as discussed in Reference 2 is employed here so that the incremental plastic strain in every time increment is kept close to a prescribed fixed value. Let \( \Delta \hat{\varepsilon}_p^{\text{max}} \) be the prescribed value and \( \Delta \hat{\varepsilon}_p^{\text{max}} \) the maximum attained effective plastic strain increment over the previous time step. Then the factor \( R \) is defined as follows:

\[
R \equiv \frac{\Delta \hat{\varepsilon}_p^{\text{max}}}{\Delta \hat{\varepsilon}_p^{\text{max}}}
\]

If \( R > 1.25 \), then the solution is rejected, a new time increment which is smaller by a factor of \((0.85/R)\) is taken and the whole step is repeated. If \( R \leq 1.25 \), then the solution is accepted and the time increment for the subsequent step is determined so that the ratio \( R \) is made close to 1.0. The following algorithm is used.

\[
\begin{align*}
\text{If } 0 \cdot 8 < R & \leq 1 \cdot 25 \text{ then } \Delta t_{n+1} = \Delta t_n/R \\
\text{If } 0 \cdot 5 < R & \leq 0 \cdot 80 \text{ then } \Delta t_{n+1} = 1 \cdot 25 \Delta t_n \\
\text{If } R & \leq 0 \cdot 50 \text{ then } \Delta t_{n+1} = 1 \cdot 50 \Delta t_n
\end{align*}
\]

A value of 0.05, is recommended in Reference 2 for \( \Delta \hat{\varepsilon}_p^{\text{max}} \), where the constitutive model of Reference 16 was used. However, different values may be more appropriate for other constitutive models.

2.4. The assumed strain method

In nearly incompressible conditions, the mesh is expected to lock [17]. To avoid locking, \( F_i \) is defined such that a predetermined number of volumetric degrees of freedom is obtained. For clarity of the presentation, \( F_i \) is here denoted as simply \( F \) and its components as \( F_{ij} \). The following decomposition of the kinematics is considered [5]:

\[
F = F_{\text{vol}} F_{\text{dev}} = F_{\text{dev}} F_{\text{vol}} \quad (60)
\]

The kinematics \( x_{hi} = \phi_{hi}(X, t) \) are usually approximated as follows:

\[
\phi_{hi} = \sum_e \sum_a \phi_{ia} N_a^e, \quad F_h = \nabla \phi_h, \quad J_h = \det(F_h) \quad (61)
\]

In the strain projection method, another measure of the discretized deformation field is used to model \( J_h \) or \( F_{\text{vol}} \equiv J_h^{1/3} I \). This leads to the following decomposition:

\[
F_h = J_h^{1/3} F_{\text{dev}} \quad (62)
\]
where \( \tilde{J}_h \) is defined using the so-called bar shape functions [43]. Interpolation with the bar shape functions is defined as follows:

\[
\tilde{J}_h(x) = \sum_{\epsilon} \sum_{Jh} \tilde{e}_a J_h(\tilde{e}_a) \tilde{N}^\epsilon_a(x)
\]

where \( \tilde{N}^\epsilon_a \) is the number of reduced integration Gauss points denoted here as \( \tilde{e}_a \). The bar shape functions are such that \( \tilde{N}^\epsilon_a(x) = \delta_{ab} \), i.e. \( J_h(\tilde{e}_a) \) is calculated at the reduced integration rule Gauss points [43]. In this section, all bar quantities indicate calculations using interpolation with the bar shape functions in the sense of equation (63).

In practice, an average deformation gradient is used (a-priori stabilization, see Reference 5) in order to eliminate possible pressure modes (\( \Pi \) between 0 and 1), i.e.

\[
F_{ave} = \Pi F + (1 - \Pi) \tilde{F}
\]

The internal work term \( G \) in the principle of virtual work (equation (49)) is now modified as follows:

\[
\langle DG_{ave}^d \rangle = \int_{B} P_d(F_{ave}) (\varepsilon \eta_{hi,l} + (1 - \varepsilon)(\tilde{\nabla} \eta_{hi,l})_{ik} \tilde{F}_{kl}) dV
\]

After performing the indicated linearizations, the above equation is modified as follows:

\[
\langle DG_{ave}^d \rangle = \int_{B} P_d(F_{ave}) (\varepsilon \eta_{hi,l} + (1 - \varepsilon)(\tilde{\nabla} \eta_{hi,l})_{ik} \tilde{F}_{kl}) dV
\]

where \( (\tilde{\nabla} \eta_{hi})_{ij} \) is defined as follows:

\[
(\tilde{\nabla} \eta_{hi})_{ij} = \eta_{hi,j}^{dev} + \frac{1}{3} \text{div} \eta_{hi,ij}
\]

The Newton–Rapshon equations requires taking a second derivative with respect to the unknown displacements, i.e.

\[
\langle D^2 G_{ave}^d, \eta_{hi,l} \rangle = \langle D(DG_{ave}^d), \eta_{hi,l} \rangle
\]

from which the following equation is derived [5]:

\[
\sum_{\epsilon} \int_{B} \left\{ \frac{\partial P_d(F_{ave})}{\partial F_{ave}} (\varepsilon u_{hi,ij} + (1 - \varepsilon)(\tilde{\nabla} u_{hi})_{im} \tilde{F}_{mj}) (\varepsilon \eta_{hi,l} + (1 - \varepsilon)(\tilde{\nabla} \eta_{hi,l})_{ik} \tilde{F}_{kl}) + P_d(F_{ave}) (\varepsilon \eta_{hi,l} + (1 - \varepsilon)(\tilde{\nabla} \eta_{hi,l})_{ik} \tilde{F}_{kl}), \eta_{hi,l} \right\} dV
\]

where the first term arises from linearizing the constitutive relations, whereas the second term is geometrical in nature. Let \( \tau = \tau_{ii}/3 \), where \( \tau \) is the Kirchhoff stress. After a number of lengthy but straightforward calculations, the linearized principle of virtual work takes the following final form:

\[
\sum_{\epsilon} \int_{B} D_{ij,l}(\varepsilon u_{hi,ij} + (1 - \varepsilon)(\tilde{\nabla} u_{hi})_{im} \tilde{F}_{mj}) (\varepsilon \eta_{hi,l} + (1 - \varepsilon)(\tilde{\nabla} \eta_{hi,l})_{ik} \tilde{F}_{kl}) dV
\]
Using the finite element discretization \( \eta_{hi} = \sum_{e} \sum_{a} \eta_{iak} N_{a}^{e} \) and the bar interpolation, one can write in terms of the vector of nodal displacements \( \{ u_{h}^{e} \} \) the following:

\[
\text{div } u_{h}^{e} = 3 \{ \mathbf{B}^{\text{vol}} \}^{T} \{ u_{h}^{e} \}, \quad \bar{\text{div }} u_{h}^{e} = 3 \{ \tilde{\mathbf{B}}^{\text{vol}} \}^{T} \{ u_{h}^{e} \}
\]

where \( \{ \mathbf{B}^{\text{vol}} \} \) and \( \{ \tilde{\mathbf{B}}^{\text{vol}} \} \) are well defined column vectors for each element. The vector form of the tensors \( \nabla u_{h}^{e}, \nabla_{a} u_{h}^{e} \) (reference gradient) and \( \bar{\nabla } u_{h}^{e} \) can now be written as follows:

\[
\nabla u_{h}^{e} = [\mathbf{B}] \{ u_{h}^{e} \}, \quad \nabla_{a} u_{h}^{e} = [\mathbf{B}_{a}] \{ u_{h}^{e} \}, \quad \bar{\nabla } u_{h}^{e} = [\tilde{\mathbf{B}}] \{ u_{h}^{e} \}
\]

where

\[
[\tilde{\mathbf{B}}(x)] = [\mathbf{B}(x)] - [\mathbf{B}^{\text{vol}}(x)] + [\tilde{\mathbf{B}}^{\text{vol}}(x)]
\]

Remark 5. The performance of the strain projection method is not the same for all types of finite elements. The issues of pressure modes and stabilization in Lagrangian analysis are not yet fully resolved and significant research is further needed in this area [45].
2.5. **Summary of the FEM algorithm**

A presentation is given here of the algorithm used to solve the principle of virtual work equations at time $t_{n+1}$ incorporating contact and frictional terms. The algorithm follows that given in Reference 23.

1. **Initialization:**
   
   Set $\lambda_N^{(0)} = \langle \lambda_N + \varepsilon_N g \rangle$ from the last time step
   
   \[ \Delta \lambda_T^{(0)} = 0 \]
   
   \[ k = 0 \]

2. **Using a Newton–Raphson iteration, solve for $u^{(k)}$ the following non-linear kinematic equations:**
   
   \[ \tilde{G}(u^{(k)}, \eta) = 0 \quad (73) \]

   where $t_N$ and $t_T$ are evaluated from equations (40) and (47), respectively.

3. **Check for constraint satisfaction.** If all the boundary points do remain in the ‘admissible region’ and the boundary tractions obey the Coulomb friction law, then the iterations have converged, i.e. IF $g(u^{(k)}) \leq TOLg$ AND $\|t_T\| \leq (1 + TOLf) \mu_f (\lambda_N^{(k)} + \varepsilon_N g)$ for all $x \in \Gamma$ THEN converge.

   EXIT (Here TOLg and TOLf are used to denote the selected tolerances for the gap and the frictional law).

   If any of the constraints are violated, augment the Lagrangian parameters, and solve the PVW equation once again, i.e.

   ELSE augment:

   \[ \lambda_N^{(k+1)} = \langle \lambda_N^{(k)} + \varepsilon_N g \rangle \]

   \[ t_T^{(trial)} = t_T^{(k)} + \Delta \lambda_T^{(k)} + \varepsilon_T \Delta \xi \]

   \[ \Phi^{(trial)} = \|t_T^{(trial)}\| - \mu_f \lambda_N^{(k+1)} \]

   IF $\Phi^{(trial)} \leq 0$, (sticking friction) THEN

   \[ \Delta \lambda_T^{(k+1)} = \Delta \lambda_T^{(k)} + \varepsilon_T \Delta \xi \]

   ELSE (sliding friction)

   \[ \Delta \lambda_T^{(k+1)} = \frac{t_T^{(trial)}}{\|t_T^{(trial)}\|} \mu_f \lambda_N^{(k+1)} - t_T^{(k)} \]

   ENDF

   \[ k = k + 1 \]

   GOTO 2

   ENDF

Note that at the first iteration at each time step, the $(n+1)$th configuration is initialized to that of the $n$th configuration. The normal and tangential tractions are also updated appropriately to account
for this change in the reference configuration. Also, after each successful calculation of the incremental displacements, the triad \((T_{n+1}, s_{n+1}, F_{n+1}^p)\) is also calculated as required for the calculations at the next time step.

Finally, it is noted that a symmetrized nested augmented Lagrangian algorithm has also been proposed in Reference 23. Its main difference from the above algorithm is that the return mapping (equation (47)) is eliminated from the solution phase in the principle of virtual work (equation (73)).

**Remark 6.** To accelerate the performance of the Newton–Raphson iterations, the tangent material moduli \(C\) at the first iteration of each time step are calculated using as \(N_{n+1}^p, \eta\) and \(\epsilon\) in equation (54), the values calculated at the end of the previous time step. This approach was found to provide a better performance than the one that calculates \(C\) using the solution of the constitutive problem for the trial deformation gradient \(F_e^* = F_e^n\) (i.e. with \(F_t = I\), see equation (23)). Also, some convergence problems of the Newton iterations may appear when a Gauss point comes in contact with the die. These problems have been attributed to the soft behaviour of the material moduli when abrupt changes occur in the direction of plastic flow. When it is needed, an artificial stiffening of the material moduli in the direction \(N_{n+1}^p\) was introduced by neglecting the last term in equation (54). Similar ideas are discussed in Reference 2.

### 3. THE GENERAL PICTURE OF A LAGRANGIAN CODE

An object-oriented framework for an updated Lagrangian analysis of large hyper-elastic viscoplastic deformations is presented here. Following the brief introduction of the various subproblems given in the earlier sections, we highlight the development of various classes and class hierarchies associated with the subproblems. Although not all classes or all details of each class can be presented in this paper, an attempt is made to provide sufficient ideas that make the general picture of a Lagrangian code clear. Various classes presented in *diffpack* are utilized here for basic linear algebra calculations as well as finite element pre- and post-processing. In addition, the stiffness/load calculations in the Newton–Rapshon iterations as well as the application of essential type of boundary conditions is performed using appropriate *diffpack* classes. The interested reader may consult References 39–41. In this section, emphasis is given to the newly developed classes for modeling non-linear kinematics, tangent stiffness calculations, non-linear material models and contact algorithms. The developed code is dimension independent and the selection of numerical techniques and algorithms, material models, die surfaces, etc., is performed at run time. Even though the developed Lagrangian simulator is general, numerical results are shown later only for particular selections of the numerical algorithms, constitutive models, etc.

#### 3.1. Review of *diffpack*

A brief review is given here of a few of the key *diffpack* classes that are used in the object oriented large inelastic analysis simulator.

Pointers are treated in *diffpack* using the class ‘HandleID’. The most important functionality of this class is that it allows the automatic deletion of an object only when all pointers that are binded to this object have been deleted. This is important for memory management when more than one pointer is binded to a given object.
The class ‘FiniteElement’ provides the necessary functionality for finite element calculations like, for example, evaluation of shape functions and their derivatives at Gauss points. The functionality that allows the stiffness and load calculations and the assembly of the linear system is performed using the class ‘FEM’. The various problem-dependent calculations are performed with the implementation in the user’s simulator of various virtual functions. For example, the main stiffness and load calculations at a particular Gauss point are performed using the pure virtual function ‘FEM:integrands()’. The class ‘ElmItgRules’ has also been implemented to allow the user to select full, uniformly reduced integration, or selected reduced integration rules.

Continuous fields are treated in *diffpack* using the class ‘FieldFE’. An object of this class is bound with a particular finite element grid (object of class ‘GridFE’). The functionality of the class ‘FieldFE’ allows the calculation of the nodal values of the field, the calculation of the field at any point within the domain, re-binding of the field with another grid, etc.

The classes ‘LinEqAdm’ and ‘NonLinEqSolver’ provide the necessary structure for handling various linear and non-linear algebraic equation solvers. The user via a menu is allowed to select from a list of available methods or implement his own solvers. The basic time management is performed using the class ‘TimePrm’.

Details on the above and many other *diffpack* classes can be found in References 39–41. It is important to note that for the implementation of the present large deformation simulator, a number of standard *diffpack* classes had to be modified. Such modifications were performed with the development of derived classes that provided the additional functionality. Most of these details are particular to the *diffpack* structure and they will not be further discussed. In the following sections, a presentation is given of the classes that are directly related to the large inelastic analysis. The following developments are general and can be easily implemented using other object-oriented libraries.

### 3.2. The matrix hierarchies

In this work, the deformation gradient $\mathbf{F}$ at a Gauss point is represented as a $3 \times 3$ $\mathbf{F}$ matrix in three dimensions and as a $3 \times 3$ $\mathbf{F}$ matrix with $F_{13} = F_{31} = F_{23} = F_{32} = 0$ in two dimensional and axisymmetric problems. Such particular matrices are represented as objects of a new class ‘MatDef’ that is derived from the class ‘Mat(real)’ which is a *diffpack* class for numerical matrix computations [40]. The hierarchies of the matrix class ‘MatDef’ are shown in Figure 4. In this and the following graphs, the solid arrow lines indicate an ‘is-a’ relation, whereas the dashed arrow lines represent a ‘has-a’ relation.

In addition to the functionality of the class ‘Mat(real)’, the matrix class ‘MatDef’ is also equipped with matrix operations that are commonly encountered in the finite element implementation of deformation problems. In particular, when the principle of virtual work equation is linearized by using the Newton–Raphson method, a linear system of algebraic equations is derived, with respect to the incremental relative deformation gradient $d\mathbf{F}$ (e.g. see equation (53)). However, the form of this equation is not amenable to solving by means of well-known linear system equation solvers. The equation obtained, though linear, involves a number of tensorial operations with known tensors. In order to automatically resolve this equation into a simplified problem (i.e. in the linear algebraic form $[\mathbf{K}][\{\mathbf{d}u\}] = \{\mathbf{f}\}$, where $\{\mathbf{d}u\}$ is the vector of the nodal incremental displacements), the following procedure is employed. Let us consider the weak form (equation (53)) in which all terms are known except the tensor $\mathbf{X} = d\mathbf{F}$. For simplicity of the presentation, a plane strain or an axisymmetric problem is assumed. One of the various tensor relations in the
Figure 4. A matrix hierarchy of particular importance to deformation problems. Mat(real) is a basic \textit{diffpack} class with its own hierarchies and functionality.

stiffness calculations in equation (53) is of the following form:

\[ AX = B \]

where \( A \) is a known tensor and \( B \) is the resulting (unknown) tensor from the operation shown on the left of the above equation. The above equation can be re-written in a discrete (matrix) form as follows:

\[ [A_l]\{X_c\} = \{B_c\} \]

where the tensor \( X \) is expressed with a column vector \( \{X_c\} \) as

\[ \{X_c\} = \begin{bmatrix} X_{11} \\ X_{12} \\ X_{21} \\ X_{22} \\ X_{33} \end{bmatrix} \]

Simple calculations show that \([A_l]\) is a 5 \times 5 matrix whose components are given by

\[ A_l = \begin{bmatrix} A_{11} & 0 & A_{12} & 0 & 0 \\ 0 & A_{11} & 0 & A_{12} & 0 \\ A_{21} & 0 & A_{22} & 0 & 0 \\ 0 & A_{21} & 0 & A_{22} & 0 \\ 0 & 0 & 0 & 0 & A_{33} \end{bmatrix} \]

Here, the subscript ‘\( l \)’ is used to denote that the original tensor operation is a multiplication of the unknown tensor \( X \) with the known tensor \( A \) from the left. The following additional tensor operations have been transformed to equivalent matrix/vector operations:

- \( XA \Rightarrow [A_{\text{right}}]\{X_c\} \)
- \( AX \Rightarrow [A_{\text{left}}]\{X_c\} \)
- \( \text{sym}(AX) \Rightarrow [A_{\text{sym}}]\{X_c\} \)
- \( \text{skew}(AX) \Rightarrow [A_{\text{skew}}]\{X_c\} \)
- \( \text{trace}(AX)B \Rightarrow [A_{\text{trace}}]\{X_c\} \), with \( B \) a given 2nd order tensor
- \( AX^T \Rightarrow [A_{\text{trans}}]\{X_c\} \)

Using the above transformations, it is now straightforward to transform the linearized tensorial form of the principle of virtual work in an algebraic system of linear equations.
In addition to the above transformations, a number of basic decompositions for ‘MatDef’ objects has also been implemented. These include the classical \( F = RU \) polar decomposition as well as the \( QDQ^T \) decomposition for symmetric matrices (where \( Q \) is a proper rotation and \( D \) a positive definite diagonal matrix). A summary of the basic functionality of class ‘MatDef’ is given in Box 1. The selection of appropriate virtual functions in the ‘MatDef’ hierarchy is performed at run time. Finally, it is noted that the class ‘MatDef’ is also used here for representing the discrete form of various other field variables that are not necessarily related with the kinematics. For such ‘MatDef’ objects, only the functionality of the `diffpack` class ‘Mat(real)’ is used.

Box 1. The class MatDef used to represent matrices

```cpp
class MatDef: public Mat(real)
{
    private:
        ...
    public:
        MatDef(): Mat(real) () {}
        MatDef(int n): Mat(real)(n) {}
        MatDef(int nrows, int ncolumns) : Mat(real) (nrows, ncolumns) {} 
        ~MatDef() {}

        virtual void right(MatDef& Aright) {}
        virtual void left(MatDef& Aleft) {}
        virtual void trace(MatDef& Atrace, const MatDef& B) {}
        virtual void symm() {}
        virtual void skew() {}
        virtual void transleft(MatDef& Atrans) {}
        virtual void RU(MatDef& R, MatDef& U) {}
        virtual void QDQt(MatDef& Q, MatDef& D) {}
        ...
};
```

3.3. The kinematic problem

The discrete kinematic problem (each iteration in the Newton–Raphson process) consists of the calculation of the incremental displacements assuming a known material state at time \( t_{n+1} \) (as calculated from the constitutive problem). The ‘Deformation’ class is defined to represent the discrete kinematics (e.g. the deformation gradient) at a given Gauss point in the current configuration (i.e. at a given ‘FiniteElement+ fc-’) with respect to its position in the reference configuration (in ‘GridFE+ grid’) (see Box 2).
Box 2. An object of the class ‘Deformation’ represents the non-linear discrete kinematics at an integration point

class Deformation: public HandleId
{
protected:
  GridFE* grid;
  FiniteElement* fe;
  ...
public:
  Deformation(GridFE* grid_, FiniteElement* fe_): grid(grid_),fe(fe_) {}
  ~Deformation() {}
  virtual void resetGrid(GridFE* grid_) {grid = grid_;}
  virtual void resetFe (FiniteElement* fe_) {fe = fe_;}
  virtual void calcF(MatDef& F);
  virtual void calcB(MatDef& B);
  virtual real detF(MatDef& F);
  virtual real calcFVol(MatDef& F);
  virtual void calcFDev(MatDef& F_dev);
  virtual void calcBVol(MatDef& B_vol);
  ...
};

Figure 5. The main class introduced to describe the non-linear kinematics

A discrete matrix ‘B’ is also defined to represent the relation between the velocity gradient tensor (as a vector) and the nodal velocities (as a vector). The calculations of the deviatoric tensor $F^{\text{dev}}$, volumetric tensor $F^{\text{vol}}$ as well as of the ‘$B^{\text{vol}}$’ portion of ‘$B$’ are also available through the ‘Deformation’ class. The ‘Deformation’ class hierarchy is shown in Figure 5.

3.4. The constitutive class hierarchies

As discussed earlier, the (isotropic) inelastic material behavior is defined via the evolution of the equivalent (scalar) plastic strain and the evolution of a number of state variables. Various constitutive models are represented as ‘functors’ (i.e. virtual functions in class hierarchies) in classes

derived from the classes ‘PStrainEvolveUDC’ and ‘StateEvolveUDC’. The definition of two separate classes for the evolution of the equivalent plastic strain and of the state variables has only been performed to simplify programming. A unified class for each constitutive model could have been defined instead that includes all evolution equations. The current implementation, however, allows an easy treatment of rate-independent models in which the plastic strain evolution is implicitly calculated via the consistency condition, whereas various hardening/relaxation models could be prescribed. A particular constitutive model is then defined as an object of the class ‘ConstitutiveBaseUDC’. This object is dynamically binded with objects of the classes ‘PStrainEvolveUDC’ and ‘StateEvolveUDC’ (see Box 3).

Box 3. The general class for the definition of an inelastic constitutive model

```cpp
class ConstitutiveBaseUDC: public HandleId {
    public:
        PStrainEvolveUDC& pstrdef;
        StateEvolveUDC& stadef;
        ConstitutiveBaseUDC(PStrainEvolveUDC& p, StateEvolveUDC& s);
        ~ConstitutiveBaseUDC();
        ..
};
```

The functionality of the classes ‘PStrainEvolveUDC’ and ‘StateEvolveUDC’ is shown in Boxes 4 and 5, respectively, where for simplicity one isotropic material state variable is considered.

Box 4. This base class provides the virtual functions for the definition of the equivalent plastic strain evolution

```cpp
class PStrainEvolveUDC: public HandleId {
    public:
        PStrainEvolveUDC();
        ~PStrainEvolveUDC();
        virtual real f(real sigma, real s) = 0;
        virtual real dfDsigma(real sigma, real s) = 0;
        virtual real dfDs(real sigma, real s) = 0;
        virtual real fInv(real eps, real s) = 0;
        virtual String name() = 0;
        ..
};
```
Box 5. This base class provides the virtual functions for the definition of the (isotropic) state variable evolution

```cpp
class StateEvolveUDC: public HandleId {
  protected:
    PStrainEvolveUDC& strain_type;
  public:
    StateEvolveUDC(PStrainEvolveUDC& strain_type_): strain_type(strain_type_)
        {} ~StateEvolveUDC() {}
    virtual real h(real sigma, real s)=0;
    virtual real dhDsigma(real sigma, real s)=0;
    virtual real dhDs(real sigma, real s)=0;
    virtual real r(real s)=0;
    virtual real drDs(real s)=0;
    real g(real sigma, real s);
    real dgDsigma(real sigma, real s);
    real dgDs(real sigma, real s);
    virtual String name()=0;
    ...
};
```

The functions $\dot{\bar{\boldsymbol{b}}} = f(\bar{\sigma}, s)$ and $\dot{s} = g(\bar{\sigma}, s) - r(s) = f(\bar{\sigma}, s)h(\bar{\sigma}, s) - r(s)$ as well as their various derivatives with respect to $\bar{\sigma}$ and $s$ are available.

The inverse function $\bar{\sigma} = f^{-1}(\dot{\bar{\boldsymbol{b}}}, s)$ is also provided. Note that most of these functions are pure virtual and must be specified in a derived class. A graphical representation of these hierarchies is given in Figure 6.

The base class ‘ConstitutiveIntegrationUDC’ is introduced to represent the discrete constitutive integration subproblem. An object of this class has direct access to the main Simulator data (‘LargeDef& data’) and thus to the values of the stresses, state variables and $\bar{\boldsymbol{F}}$ at the beginning of the time step (time $t_n$). The integration of the constitutive model is performed at each Gauss point in all finite elements. The class ‘ConstitutiveIntegrationUDC’ therefore has access to an object of the class ‘FiniteElement’ which can be set to a particular Gauss point. This class also has access to the constitutive model used (i.e. to an object of the class ‘ConstitutiveBaseUDC’). An object of class ‘MatDef’ which physically represents the deformation gradient and is used to drive the incremental constitutive problem is also a member of ‘ConstitutiveIntegrationUDC’.

The virtual function ‘update(Boolan post_process = dpFALSE)’ solves the discrete constitutive integration subproblem (i.e. calculates the stresses, state variables and $\bar{\boldsymbol{F}}$ at the end of the step, time $t_{n+1}$). The material state, stresses and $\bar{\boldsymbol{F}}$ are stored in two different formats. A discrete ‘MatDef’ format stores the values of these variables at every Gauss point of each element. This format is used in all calculations. A ‘FieldFE’ format of these variables is also computed and stored. This format is essential for post-processing purposes. The ‘Boolan’ argument in ‘update’ controls when the field format of the various variables is updated. The default argument ‘FALSE’ implies that only the discrete ‘MatDef’ format is updated. Such updates are performed during the Newton–Raphson iterations. The ‘FieldFE’ format is updated only after convergence has been achieved in a given time step. The argument ‘TRUE’ is used in this case.
Particular integration schemes are implemented as functors (virtual function ‘update’) in classes derived from the base class ‘ConstitutiveIntegrationUDC’. Figure 6 shows two such derived classes, ‘IsotropicRateDependent’ and ‘IsotropicRateIndependent’ that are used to define the constitutive integration schemes for isotropic rate-independent and rate-dependent plasticity models. Currently both of these classes have been developed using the radial return mapping as outlined earlier. Box 7 shows the functionality of the class ‘IsotropicRateDependent’. The names of the various functions are directly related with the various steps in the integration scheme and no further details of this functionality will be provided.

Finally, it is mentioned here that *diffpack* provides a convenient frame work for the use of ‘functors’. Let us for example consider the class ‘StateEvolveUDC’ and briefly outline how a
pointer of this base class can be rebinded appropriately in the derived hierarchy. At first, a parameter structure is defined that includes all arguments needed in the constructor of ‘StateEvolveUDC’ and in addition a ‘String’ parameter that can be used to identify the state model (see Box 5):

```cpp
struct prm(StateEvolveUDC) {
    String state_type;
    PStrainEvolveUDC* pstrain_type;
};
```

**Box 6.** This class is used to define the virtual function ‘update’ for the numerical integration of the discrete constitutive problem.

```cpp
class ConstitutiveIntegrationUDC: public HandleId {
    protected:
        LargeDef& data;
        FiniteElement* fe;
        ConstitutiveBaseUDC& model;
        MatDef* F;
    public:
        ConstitutiveIntegrationUDC(LargeDef& data_, FiniteElement* fe_,
                                   ConstitutiveBaseUDC& model_, MatDef* F_);
        ~ConstitutiveIntegrationUDC() {} 
        void resetFe(FiniteElement* _fe) fe = _fe;
        void resetF(MatDef* _F) {F = _F;}
        virtual void update(Bool post_process = dpFALSE) = 0;
        virtual real calcPlasticWorkRate() = 0;
        MatDef cauchy_stress;       // Cauchy stress
        MatDef tangent_moduli;      // Tangent material moduli
        real sigf, sf;              // Equivalent stress and scalar state variable
        real equiv_strain;          // Equivalent strain
        MatDef Fe;                  // Elastic deformation gradient
        Handle(MatDef) Fs;          // Trial deformation gradient
        real pres_trial_stress;     // Mean normal pressure of the trial stress
};
```

Here, ‘state_type’ is a ‘String’ object that the user provides at run time to define the specific state evolution model. Let us assume that an object ‘pm’ of ‘prm(StateEvolveUDC)’ has been defined. A pointer to the base class can now be appropriately rebinded to the specific state evolution model.
Box 7. This class defines a numerical integration object for isotropic viscoplastic constitutive models

class IsotropicRateDependent: public ConstitutiveIntegrationUDC {
    protected:
    friend class IsotropicRateIndependent;
    MatDef bar_cauchy_stress; // Rotation neutralized Cauchy stress
    real eq_trial_stress; // Equivalent trial stress
    MatDef trial_stress; // Trial stress tensor
    MatDef dev_trial_stress; // Deviatoric part of the trial stress
    real radial_return; // Radial return factor
    real trE; // Trace of the logarithmic strain
    MatDef trialR, trialU; // From the RU decomposition of the trial def. grad.
    MatDef logE; // Logarithmic strain
    void init();
    void defoFields2Matrix();
    void calcTrialStrain();
    void calcTrE(); // Calculates the trace of the trial strain
    void pressure(); // Calculates the pressure part of the trial stress
    void equivStress(); // Calculates the equivalent trial stress
    void calcTrialStress();
    void calcDevStress(); // Calculates the deviatoric part of trial stress
    void solve(); // Solves the two algebraic non-linear equs
    void calcBarStress(); // Calculates the rotation neutralized Cauchy stress
    void pushForward(); // Pushes forward to obtain the Cauchy stress
    void updateStress(); // Updates the stress in the main simulator
    void updateState(); // Updates the state variables
    void updateFe(); // Updates the elastic deformation gradient
    void calcEpsilon(); // Calculates the equivalent plastic strain
    void calcMaterialModuli(); // Calculates the material moduli
    ...

    public:
    IsotropicRateDependent(LargeDef& data, FiniteElement* fe, 
                            ConstitutiveBaseUDC& model, MatDef* F): 
        ConstitutiveIntegrationUDC(data, fe, model, F) {}
    ~IsotropicRateDependent() {}
    virtual void update(Boolen post_process = dpFALSE);
    virtual real calcPlasticWorkRate();
    ...
};
as follows:

StateEvolveUDC* ptr = NULL;

.. if (EQ(pm.state_type, PowLaw))
   ptr = new StatePowLaw(*pm.pstrain_type);
else if (EQ(pm.state_type, SimPowLaw))
   ptr = new StateSimPowLaw(*pm.pstrain_type);
else if (EQ(pm.state_type, Perzyna))
   ptr = new StatePerzyna(*pm.pstrain_type);
..
else if (EQ(pm.state_type, Sinh))
   ptr = new StateSinh(*pm.pstrain_type);
..

A similar re-binding process of a base class pointer is applied for all class hierarchies developed in this work.

3.5. The contact class hierarchies

As part of the augmented Lagrangian approach to contact that has been currently implemented, a class ‘Die’ is introduced to allow for an abstract representation of a generic die surface. The diffpack class ‘Ptv(real)’ provides an efficient means of defining ‘point’ objects. A parametric representation of the die surface is considered in terms of an object of the class ‘Ptv(real)’ representing the parameter \( \xi \) in two dimensions and the parameters \( (\xi, \eta) \) in three dimensions. The parameter range used is \([0,1]\). The various projection-related functions like finding the gap, curvature, etc., though dimension-dependent are generic enough to be implemented in the derived classes ‘Die2D’ and ‘Die3D’. The die-shape-related functions are however specific to each die and are implemented using ‘functors’ in the hierarchy. In particular, the die surfaces must be defined as virtual functions in a derived class e.g. the class ‘ExtrusionApplicationDie3D’ (see Figure 7). Such die-specific classes are usually introduced together with the particular Simulator so that re-compilation of the ‘Die’ class is not needed for each application. Also note that the various projection related functions can be re-implemented in a derived class if more efficient algorithms can be designed for the particular die. The basic functionality of the class ‘Die’ is given in Box 8 and a graphical representation of the various hierarchies is given in Figure 7. Note that other types of representations of die surfaces that are possibly needed for algorithms that are not based on the current augmented Lagrangian analysis can be introduced directly in the class ‘Die’ using either overloaded functions or additional virtual functions.

The base class for representing contact with a die is presented in Box 9. For each individual die, a contact indicator is assigned that is used to specify the potential regions of contact, while solving the kinematic problem. Each ‘Contact’ object is defined for a given die and has direct access to the data of the main Simulator. The most important virtual function in the class ‘Contact’ is the function ‘numItgOverContactSide()’ which implements the contribution of contact to the tangent stiffness matrix. These contributions are strongly dependent on the particular physical contact/friction model. The actual implementation of the function ‘numItgOverContactSide()’ is provided in classes derived from ‘Contact’. For Coulomb type of frictional models, the class ‘Coulomb’ implements both the symmetric and non-symmetric stiffness versions of the augmented
Lagrangian algorithm. Different frictional models that take into account micromechanical mechanisms within the contact interface, can also be implemented using the ‘Contact’ hierarchies. A graphical representation of these hierarchies is shown in Figure 8.

Box 8. Definition of the class ‘Die’. The various die shape-related functions are defined as virtual functions in a hierarchy derived from the class ‘Die’

```cpp
class Die: public HandleId {
  protected:
    virtual real distance(Ptv(real)& x, Ptv(real)& param) = 0;
  ...

class Die2D, Die3D, StraightForgingDie2D, FlatFaceDie, ExtrusionApplicationDie2D, ExtrusionApplicationDie3D;
```
Box 9. Objects of the class ‘Contact’ are here defined to represent geometric information related to contact as well as to solve the discrete contact problem

```cpp
class Contact: public HandleId
{
    protected:
        virtual void initMeshData()=0; // Find elements in contact, etc.
        int contact_indicator; // The contact boundary indicator
        LargeDef& data; // Mainly for information about the grids
        Die& die; // The die object against which contact occurs
    ...

    public:
        Contact(Die& die, LargeDef& data, int contact_indicator);
        ~Contact() {}
        real macauley(real x);
        real heavySide(real x);
        virtual void initDataStructures()=0;
        virtual void init4timeStep()=0;
        virtual void numItgOverContactSide(int side, int boind, ElmMatVec& elmat, FiniteElement& fe) = 0;
        virtual void symmetricStiffness(int side, int boind, ElmMatVec& elmat, FiniteElement& fe) = 0;
        virtual void nonSymmetricStiffness(int side, int boind, ElmMatVec& elmat, FiniteElement& fe) = 0;
        virtual Boollean isContactOK()=0; // Checks contact constraints
        virtual void augment()=0;
        virtual void calcTractions()=0; // Computes boundary tractions
        virtual void calcDieForce()=0; // Computes the total force on the die
        virtual String name()=0;
    ...
};
```

Figure 8. Class hierarchies introduced to describe contact with a rigid die in forming processes
3.6. The main Simulator class

The main Simulator, class ‘LargeDef’, is the most essential class that allows the interaction of all other developed classes. Some of the members of this class are shown in Box 10. Information is included about the initial \((t = 0)\), reference \((t = t_n)\) and current \((t = t_{n+1})\) grids in a ‘GridFE’ format. The main unknown field is the incremental displacement that is represented in a ‘FieldFE’ form. Scratch vectors for both the linear (i.e. for the differential incremental displacements) and non-linear solutions (i.e. for the incremental displacements) are also included. The various material histories (e.g. the Cauchy stress, the equivalent plastic strain, the state variables, \(\bar{F}^P\), etc.) are kept...
in both a ‘FieldFE’ format that allows the use of various plotting tools and in a ‘MatDef’ form that represents values at the Gauss points of each element. The handling of the non-linear and linear solvers is decided at run time via the appropriate binding of the relevant pointers. Finally, smart pointers are introduced for all subproblems discussed earlier (e.g. for the kinematic, constitutive
and contact subproblems). The binding of these initially ‘NULL’ pointers is performed at run time. At selected times, the essential fields are dumped in ‘SimResFile’ files that can be used either to restart a previously interrupted program or for plotting purposes. A specification of a ‘FieldFormat’ for each field that is being read is then essential to define in which form each field is read (e.g. from a file, function, etc.).

The most important functional aspects of the Simulator are presented in Box 11. The virtual functions ‘integrands’ and ‘integrands4side’ are inherited from the class ‘FEM’ and define the specific tangent stiffness/load calculations. All contributions of an element are accounted for in ‘calcElmMatVec()’ (see Box 11) that is also inherited from the class ‘FEM’. Indicators have been assigned to represent each type of boundary conditions. These indicators are assigned to element nodes and sides as part of the mesh generation process and are used in the virtual function ‘calcElmMatVec’ to account for the particular contributions of an element to the tangent stiffness and load. The treatment of essential boundary conditions is described in Reference 41.

The function ‘timeLoop’ contains the main time integration loop. Considering that different data have to be stored/plotted at various time steps for different processes (e.g. in ‘Forging’, one may be interested to calculate the die force at each time step), a number of virtual functions (e.g. ‘storeApplicationSpecificData’) declared in the main simulator can be defined by the user in each application (e.g. in the class ‘Forging:LargeDef’). However, it should be mentioned that the classes derived from ‘LargeDef’ have only a minor number of additional functionalities to that of ‘LargeDef’.

The functions ‘define’, ‘scan’ and ‘adm’ are used to define and read via a menu various input data, whereas the function ‘driver’ activates the problem solution process. A summary of the key members of the main simulator as well as of the various application classes is given in Figure 9.

4. NUMERICAL EXAMPLES

A number of numerical examples including some benchmark problems will be considered here to validate the developed simulator as well as to report the cpu requirements of the present object oriented implementation. All computations were performed on an IBM RS-6000 workstation at the Cornell Theory Center.

4.1. Example 1: Axisymmetric isothermal open-die forging of 1100-Al

As an example of a large deformation metal forming problem, the upset forging of a cylindrical 1100-Al billet is considered between parallel dies. This benchmark problem was examined earlier in References 2 and 3 using ABAQUS with the CAX4 special elements to address incompressibility. The radial return mapping for the integration of the constitutive model was the same as that given here, however, the tangent stiffness, the assumed strain model and the augmented Lagrangian algorithm for contact presented here were not implemented in the calculations of References 2 or 3.

The constitutive model for 1100-Al is presented in References 15 and 16 at 673 K. Under these isothermal conditions, the flow function \( f \) is given as follows:

\[
f (\bar{\sigma}, s) = A \left( \sinh \left( \frac{\bar{\sigma}}{s} \right) \right)^{1/m}
\]

Box 11. The basic functionality of class ‘LargeDef’

```cpp
class LargeDef: public FEM, public MenuUDC, public NonLinEqSolverUDC
{
    protected:
        virtual void timeLoop(); // Main time integration loop
        void solveAtThisTimeLevel(); // Solves the deformation probl. in a single time step
        void MakeAndSolveLinearSystem(int iter); // Solves a single Newton iteration
        void setIC(); // Sets initial conditions for the problem
        void fillEssBC(); // Treats essential boundary conditions
        virtual real ebcDisplacement() = 0; // Defines process specific essen. bound. cond.
        void calcElmMatVec(int e, ElmMatVec& elmat, FiniteElement& fe);
        void integrands(ElmMatVec& elmat, FiniteElement& fe);
        void integrands4side(int side, int boind, ElmMatVec& elmat, FiniteElement& fe);
        void calculateReduce(FiniteElement& fe, MatDef& F, MatDef& EF);
    public:
        LargeDef();
        ~LargeDef();
        virtual void define(MenuSystem& menu, int level = MAIN);
        virtual void scan(MenuSystem& menu);
        virtual void adm(MenuSystem& menu);
        virtual void driver();
};
```

and the hardening function $g$ is given by

$$g(\tilde{\sigma},s) = h(\tilde{\sigma},s) \ f(\tilde{\sigma},s)$$  \hspace{1cm} (75)

where the function $h$ is defined as follows:

$$h(\tilde{\sigma},s) = h_0 \left| 1 - \frac{s}{s^*} \right|^a$$  \hspace{1cm} (76)

with

$$s^* = \tilde{s} \left( \frac{\dot{\varepsilon}P}{A} \right)^n$$  \hspace{1cm} (77)

The specific values of the material parameters are given in Table I.
Table I. Material parameters for Al 1100-O at 673 K

<table>
<thead>
<tr>
<th>Material parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A$</td>
<td>$4.75 \times 10^{-7} \text{s}^{-1}$</td>
</tr>
<tr>
<td>$\zeta$</td>
<td>7.0</td>
</tr>
<tr>
<td>$m$</td>
<td>0.23348</td>
</tr>
<tr>
<td>$s_0$</td>
<td>29.7 MPa</td>
</tr>
<tr>
<td>$h_0$</td>
<td>1115.6 MPa</td>
</tr>
<tr>
<td>$a$</td>
<td>1.3</td>
</tr>
<tr>
<td>$\hat{s}$</td>
<td>18.92 MPa</td>
</tr>
<tr>
<td>$n$</td>
<td>0.07049</td>
</tr>
<tr>
<td>$\mu$</td>
<td>20.2 GPa</td>
</tr>
<tr>
<td>$\kappa$</td>
<td>66.0 GPa</td>
</tr>
</tbody>
</table>

Table II. Simulation parameters for the open die forging of a cylindrical 1100-Al billet (Example 1)

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Energy error norm</td>
<td>$1.0e-06$</td>
</tr>
<tr>
<td>Displacement $L_\infty$ error norm</td>
<td>$1.0e-06$</td>
</tr>
<tr>
<td>Normal penalty for contact</td>
<td>$1.0e+07$</td>
</tr>
<tr>
<td>Tolerance for gap</td>
<td>$1.0e-04$</td>
</tr>
<tr>
<td>Tangent penalty for contact</td>
<td>$1.0e+04$</td>
</tr>
<tr>
<td>Tolerance for friction condition</td>
<td>$1.0e-04$</td>
</tr>
</tbody>
</table>

In this forging example, the initial cylindrical billet is 2 mm in diameter and 3 mm high. The forging die is modeled as a rigid surface and to simulate sticking friction between the die and the workpiece, a coefficient of friction of 0.8 is applied. The deformation is highly non-homogeneous with variable rates of straining at material points and time-varying/unsteady contact. The symmetry of the problem allowed modeling one fourth of the geometry. A nominal strain rate of 0.01 was applied during the forging process. A rather coarse uniform grid of $10 \times 10$ four-noded quadrilateral finite elements was used to discretize the domain. A stabilizing parameter of $\varepsilon = 10^{-4}$ was used in the simulation. The initial billet was subject to a height reduction of 60 per cent in 90 s using a fixed time step $\Delta t = 1.0$ s.

The primary nested augmented Lagrangian algorithm was used to enforce the contact conditions. The main parameters used in the simulation are given in Table II. The deformed mesh after 60 per cent height reduction (90 time steps) is shown in Figure 10. Figures 11 and 12 show the contours of the equivalent plastic strain and of the internal state variable at the end of the upsetting process. The force necessary to forge the workpiece versus the stroke is shown with the continuous line in Figure 13. The above results match very well with the ABAQUS calculations reported in References 2 and 3. The computing statistics of this calculation is shown in Table III. Table IV shows the convergence characteristics that was obtained in a typical time step.

The above forging simulation was also performed with the automatic time stepping scheme by specifying the maximum allowable plastic strain increment in a time-step as 0.04. The corresponding force versus stroke curve is also shown in Figure 13 with discrete points and is indistinguishable.
Figure 10. Initial and final deformed mesh after 60 per cent height reduction in axisymmetric open die forging (Example 1)

Figure 11. Contours of the equivalent plastic strain in the final forged product (Example 1)

Figure 12. Contours of the internal state variable (in MPa) in the final forged product (Example 1)

from the fixed time step results. These calculations required 72 time increments. As expected, the time step is automatically reduced in the time domain intervals where ‘fold-over’ occurs.

4.2. Example 2: Axisymmetric isothermal closed die forging of an 1100-Al workpiece

We consider the closed die forging of a 1100-Al cylindrical workpiece with the same constitutive model as in Example 1. This problem demonstrates the effectiveness of the implicit contact algorithm in simulating the filling of a die cavity. The workpiece is a circular cylinder whose volume is chosen so that at the end of the forging process the material will fill exactly the die cavity without the formation of ‘flash’. The initial radius of the cylinder is 6.31 mm and the initial height is 10.0 mm. A friction coefficient of 0.1 is assumed at the die-workpiece interface. A nominal
strain rate of 0.01 was applied during the forging process. $13 \times 10$ four-noded quadrilateral finite elements were used to discretize the domain (Figure 14). A stabilizing parameter of $\epsilon = 10^{-4}$ was used. The automatic time stepping technique described in Section 2.3, with an allowable maximum plastic strain increment of 0.01, was used in the simulation. A total of 311 time increments was needed to maintain this strain control.

The main parameters used in the simulation are given in Table V. The deformed mesh at the final stage is shown together with the initial mesh in Figure 14. Figures 15 and 16 show the
Table V. Parameters for the closed die forging of the cylindrical 1100-Al billet (Example 2)

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Energy error norm</td>
<td>1·0e−04</td>
</tr>
<tr>
<td>Displacement $L_{\infty}$ error norm</td>
<td>1·0e−04</td>
</tr>
<tr>
<td>Normal penalty</td>
<td>1·0e+06</td>
</tr>
<tr>
<td>Tolerance for gap</td>
<td>1·0e−04</td>
</tr>
<tr>
<td>Tangent penalty</td>
<td>1·0e+03</td>
</tr>
<tr>
<td>Tolerance for friction condition</td>
<td>1·0e−04</td>
</tr>
</tbody>
</table>

Figure 15. Contours of the equivalent plastic strain in the final forged product (Example 2)

Contours of the equivalent plastic strain and of the internal state variable, respectively, at the end of the closed die forging process. The force necessary to forge the workpiece versus the stroke is shown in Figure 17. The computing statistics are summarized in Table VI.

4.3. Example 3: Axisymmetric extrusion of a cylindrical billet of a rate-independent material

Another benchmark test problem is considered here that is related with the axisymmetric extrusion of a workpiece through a rigid conical die.23 A rate-independent material is considered with linear hardening. The material properties and the simulation parameters are shown in Table VII. In this axisymmetric problem, the aluminum billet is of radius 5.08 cm and the initial length is 25.4 cm. The workpiece is pushed (using a displacement control method) to a total distance of...
Table VI. Computing statistics for the upset closed die forging of a cylindrical billet (Example 2)

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total cpu time</td>
<td>01 : 35 : 16</td>
</tr>
<tr>
<td>Augmentations per increment</td>
<td>1</td>
</tr>
<tr>
<td>Newton–Raphson iterations per increment</td>
<td>4.5</td>
</tr>
<tr>
<td>Time to assemble stiffness matrix every iteration</td>
<td>2.4 s</td>
</tr>
<tr>
<td>Time to solve every iteration</td>
<td>&lt;0.5 s</td>
</tr>
</tbody>
</table>

Table VII. Material and simulation parameters for the conical extrusion process (Example 3)

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\kappa$</td>
<td>63.84 GPa</td>
</tr>
<tr>
<td>$\mu$</td>
<td>26.12 GPa</td>
</tr>
<tr>
<td>Initial yield stress</td>
<td>31 MPa</td>
</tr>
<tr>
<td>Linear hardening rate</td>
<td>261.2 MPa</td>
</tr>
<tr>
<td>Time step</td>
<td>5 s</td>
</tr>
<tr>
<td>Total time</td>
<td>180.0 s</td>
</tr>
<tr>
<td>Friction coefficient</td>
<td>0·1</td>
</tr>
<tr>
<td>Displacement $L_{\infty}$ error norm</td>
<td>1·0e − 05</td>
</tr>
<tr>
<td>Energy error norm</td>
<td>1·0e − 05</td>
</tr>
<tr>
<td>Normal penalty</td>
<td>1·0e + 07</td>
</tr>
<tr>
<td>Tangent penalty for contact</td>
<td>1·0e + 04</td>
</tr>
</tbody>
</table>

18 cm into a conical die with semi-angle of 5°. The discretization of the billet was performed using a uniform $4 \times 20$ grid of four-noded quadrilateral elements. Due to the nature of the contact conditions, the stiffened material moduli suggested in Remark 6 are used here at all time increments. Using such inconsistent material moduli and a constant time step of 5 s resulted in a total cpu time of about 10 min.

The effect of friction is clearly seen in the deformed meshes which depict the shearing near the die walls (Figure 18). These results are in excellent agreement with results given in Reference 23. The contours of the equivalent plastic strain and of the state variable are shown in Figure 19. The extrusion force versus displacement is shown in Figure 20. The extrusion force results as well as the deformed configuration reported in Reference 23 are in excellent agreement with the results given here. The computing statistics of this calculation performed using 36 time increments is reported in Table VIII.

Finally, it was observed that the average number of contact augmentations required per time increment (here $\sim 1.6$) was significantly smaller for simulations with smaller time step (e.g. $\sim 1$ augmentation for a time step of 2.0 s).

4.4. Example 4: Plane strain frictionless closed die forging of an 1100-Al workpiece

A plane strain forging problem, where a cylindrical aluminum billet is converted from a circular cross section to a crucible shape, presented in Reference 2 is analysed here. The initial billet
was sized so that the die would be filled completely and in addition some amount of flash would result. This problem presents several interesting features such as a high degree of non-homogeneous deformation, variable regions of contact between the billet and the die, time-varying deformation rates at material points and a rapid rise of the total die force as the die becomes filled.

The die–workpiece interface is considered frictionless and the material constitutive model is the same as that of Example 1. The symmetry of the workpiece is used to model only one quarter of the workpiece. 448 four-noded quadrilateral elements are used to discretize the geometry. The initial geometry and finite element mesh are shown in Figure 21. A total of 280 time increments were used to simulate the process. The various simulation parameters are summarized in Table IX. Figure 22 shows the finite element mesh at various stages of deformation. The material flows predominantly horizontally until approximately 4.5 mm of stroke. Figures 23 and 24 show the contours of the equivalent plastic strain and the internal state variable (in MPa), respectively. The die force versus the stroke is shown in Figure 25. It is noted that there is a sudden increase in the required forging force as the workpiece is squeezed to fill the die.

The results above are in very good agreement with the results in Reference 2 where ABAQUS was used for the simulations [46]. The computing statistics are summarized in Table X. For the given discretization, a total cpu time of about 6 hrs was required for the complete simulation. A cpu time of 19.0 h was reported in the simulation of Reference 2, where a coarser mesh than the one used here was considered.

5. CONCLUSIONS

An object-oriented simulator based on ddiffpack was developed for the analysis of large inelastic deformations and metal forming processes. Several benchmark test problems were examined to demonstrate the accuracy of the developed algorithms.

The present implementation provides a powerful and efficient tool for the analysis of large inelastic deformations and metal-forming processes. Some of the key elements of the developed simulator are the following:

Figure 19. Contours of equivalent plastic strain and isotropic deformation resistance (in MPa) for the extruded material (Example 3)

Table VIII. Computing statistics for the conical extrusion problem (Example 3)

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total cpu time</td>
<td>00:10:15</td>
</tr>
<tr>
<td>Augmentations per increment</td>
<td>1.55</td>
</tr>
<tr>
<td>Newton–Raphson iterations per augmentation</td>
<td>6.5</td>
</tr>
<tr>
<td>Formation of the linear system of equations</td>
<td>1.4 s</td>
</tr>
<tr>
<td>Time to solve the linear system</td>
<td>&lt;0.5 s</td>
</tr>
</tbody>
</table>
Figure 20. Extrusion force versus displacement curve for the conical extrusion problem (Example 3)

Figure 21. Initial geometry and finite element mesh for the plane strain frictionless closed die forging problem (Example 4)

Table IX. Simulation parameters for the plane strain closed die forging process (Example 4)

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time step</td>
<td>0.2 s</td>
</tr>
<tr>
<td>Total time</td>
<td>56.0 s</td>
</tr>
<tr>
<td>Displacement $L_\infty$ error norm</td>
<td>1.0e − 04</td>
</tr>
<tr>
<td>Energy error norm</td>
<td>1.0e − 04</td>
</tr>
<tr>
<td>Normal penalty</td>
<td>1.0e + 06</td>
</tr>
</tbody>
</table>

1. The same code can be used for two- or three-dimensional problems.
2. The class hierarchies developed allow the implementation of additional constitutive models, integration algorithms, die surfaces, contact algorithms, etc., without any changes to the program structure. The key idea is that new algorithms or materials have to be simply programmed as ‘functors’ in the already existing class hierarchies.
Figure 22. The finite element mesh at various stages of deformation: after 1, 2, 3, 4, 5 mm of stroke and at the final forged product, respectively (Example 4)

Figure 23. Contours of the effective plastic strain $\varepsilon^P$ in the final forged product (Example 4)
Figure 24. Contours of the internal state variable $\varepsilon$ (in MPa) in the final forged product (Example 4)

Figure 25. Die force versus die stroke for the plane strain closed die forging process (Example 4)

3. Various forming process simulators can be derived from the developed ‘LargeDef’ simulator with only a minor programming effort, for example by simply implementing (as a ‘functor’) a particular process dependent post-processing functionality. The most commonly needed pre- and post-processing operations have already been introduced as pure virtual functions in the main simulator.

4. The use of smart pointers (‘Handles’) for memory management and fields to represent the various continuous variables (‘FieldFE’ objects) allows a straightforward coupling of the
present simulator with other object-oriented simulators, e.g. a thermal or a fluid flow simulator 
(see References 32–34 for examples of how such coupling can be implemented). Similarly, 
the consideration of deformable dies can easily be implemented by representing the die itself 
as an object of the class ‘LargeDef’.

Various time statistics were presented for the examples examined in order to allow the interested 
reader to compare the efficiency of the present *diffpack* implementation with other non-object orien-
ted simulators of large inelastic analysis. In comparison to our earlier simulator [44], the present 
object-oriented approach was found to be only slightly more expensive. A precise comparison was 
not possible since the present simulator includes many features that are not available in Refer-
ence 44 (for e.g. the additional stiffness terms required for the implementation of the assumed 
strain method). We believe that the extra cost required for object-oriented implementations is only 
a small penalty that someone has to pay for receiving the rewards of a robust code that can 
rung without modifications in most computer platforms and that allows generality, expandability, 
maintainability and FEM code re-usability [30].

ACKNOWLEDGEMENTS

This work was funded by NSF grant DMII-9522613 to Cornell University. The computing was 
supported by the Cornell Theory Center, which receives major funding by the NSF and IBM cor-
poration, with additional support from the New York State. The academic license for using the 
various libraries of *diffpack* is acknowledged. Finally, the help of Mr. Yangang Bao in program-
ming various parts of the present Simulator is appreciated.

REFERENCES

3. Weber G, Anand L. Finite deformation constitutive equations and a time integration procedure for isotropic, hyperelastic-
4. Eterović AL, Bathe K-J. A hyperelastic-based large strain elasto-plastic constitutive formulation with combined isotropic-
kinematic hardening using logarithmic stress and strain measures. *International Journal for Numerical Methods in 
5. Moran B, Ortiz M, Shih CF. Formulation of implicit finite element methods for multiplicative finite deformation 
6. Zabaras N, Arif AFM. A family of integration algorithms for constitutive equations in finite deformation elasto-


46. *ABAQUS, Reference Manuals*, Hibbitt, Karlsson and Sorensen Inc., 100 Medway Street, Providence, RI 02906–4402, U.S.A.