Lazy Evaluation of PDE Coefficients in the EScript System

Joel Fenwick **Lutz Gross**

Earth Systems Science Computational Centre (ESSCC),
The University of Queensland,
St Lucia, QLD 4072, Australia.
Email: JoelFenwick@uq.edu.au

Abstract

EScript is an extension to Python for solving partial differential equations on parallel computers. It is parallelised for both MPI and shared memory, multi-core systems using OpenMP. In this paper, we discuss *lazy evaluation* as a strategy to reduce the cost of evaluating the coefficients of PDEs prior to solving. We show that our implementation provides significant memory and time savings for a problem involving complex expressions.

1 Introduction

EScript is a Python extension for solving general linear, steady state, second order partial differential equations (PDEs) (Gross et al. 2007). It supports parallel execution for OpenMP, MPI or both (source code is available on launchpad (lpe 2009)). The goal of *escript* is to provide users, who are primarily modellers, with a means to construct and run simulations on parallel computers without needing expertise in parallel programming or lower level languages such as *C++*. With this in mind, the work described in this paper had three competing objectives:

1. reduced run time.
2. reduced peak memory usage.
3. minimal disruption to the existing interface and minimal work on the part of users to apply new features.

EScript and the simulations built upon it, contain a significant amount of Python code. Any changes we make, must be able to work in an interactive (and interpreted) Python session if required. This interpreted aspect, while making experimentation and scripting easier for non-programmers, does impose some constraints on the methods used to improve performance. In particular, this limits techniques which require preprocessing or precompilation of scripts.

To describe a PDE in *escript*, functions must be constructed (or loaded) to form its coefficients. The coefficients may depend on a PDE solution from a previous iteration or timestep. They may also depend on the solutions to a different PDE in the case of a coupled problem. In this paper, we discuss the representation and evaluation of these coefficient functions.

By default, functions are stored explicitly. That is, the representation stores a number of floating point values proportional to the complexity of the domain. The more operations required to construct a function and the more complex the objects involved, the greater the amount of memory required to store intermediate functions. As a very simple example, consider the expression

$$y = ax + b.$$

In order to evaluate y, an intermediate result

$$\tau = ax$$

is required. If a and x are simple values then storing τ won't present much of a problem. If they are more complex objects though, storing their product could require non-trivial amounts of memory. Such intermediate functions are typically not required once the final function is computed. For large domains and intermediate functions involving tensors, this memory cost can be significant. This can limit the problem sizes which can be handled by the system. We investigate the use of *lazy evaluation* to reduce the burden on the system. Lazy evaluation means that "arguments to a function are evaluated only when needed for computation"(Pandey 2008).

After a brief comment on OpenMP, we will discuss data representation followed by threading. Section 5 contrasts evaluation in functional environments to *escript*. Section 6 describes two implementations of the resolve operation. Performance experiments follow in Section 7.

2 OpenMP

The OpenMP (omp 2009*a*) model of shared-memory parallelism uses a team of threads. Code executes on a single master thread until it reaches a parallel region. All threads in the team execute the section. At the end of the section, execution continues on the master thread (omp 2009*b*). In the case of *escript*, OpenMP is used primarily to parallelise *for* loops.

3 Representation

EScript can be compiled to use MPI, OpenMP or both (hybrid mode). In the case of MPI, function information is distributed among the MPI processes at creation time and processed independently. There may be an aggregation step at the end for operations such as integration. Since MPI does not present threading issues, we will ignore it for the purposes of this discussion.

This work is supported by the AuScope National Collaborative Research Infrastructure Strategy by the Australian Commonwealth, the Queensland State Government and The University of Queensland.

Copyright ©2010, Australian Computer Society, Inc. This paper appeared at the 8th Australasian Symposium on Parallel and Distributed Computing (AusPDC 2010), Brisbane, Australia.. Conferences in Research and Practice in Information Technology (CRPIT), Vol. 107, Jinjun Chen and Rajiv Ranjan, Ed. Reproduction for academic, not-for profit purposes permitted provided this text is included.

Functions are represented in two parts (input and output). The input to the function is determined by a Domain object (a mesh) and a point selection strategy (called a FunctionSpace in *escript* terminology). For example, the values of the function could be derived from the nodes bounding an element or from the points in the interior. Storing this information is the responsibility of the Domain object. Note that *escript* itself does not impose any meaning on particular FunctionSpace IDs; these are determined by the Domain in use.

Interpolation between different FunctionSpaces (where possible), is performed by the Domain object. Regardless of the FunctionSpace in use, the collection of points representing an element is termed a *sample*.

It should be noted that while the ordering of samples and points must be consistent, *escript* does not assume a global ordering of points.[1]

Data objects represent the values (outputs) of functions and are linked to a particular Domain and FunctionSpace. Values can be scalars (rank[2] 0) through to 4-Tensors (rank 4) of various shapes. Mathematical operations including basic arithmetic, matrix operations and tensor products are defined on Data objects. The majority of computation in *escript* itself acts on Data objects and evaluating expressions involving Data is the focus of this paper.

The final component is the solver. In the *escript* distribution, the PDE is converted to a sparse matrix representation and passed to the *Paso* sparse matrix solver. This paper will not directly discuss the performance of Paso either.

Operations on Data objects could be thought of as batch operations in that the same operation is performed on each value of the collection. Since the values in the Data object (function output) must correspond to points in the Domain, it might be tempting to view Data as an array. But this analogy breaks down when the Domain is distributed across multiple processes.

Internally, Data objects act as proxies for instances of the DataAbstract class which actually store the values. This allows flexibility in switching representations without disturbing the rest of the system. For the rest of the paper, we will use “node” to refer to instances of DataAbstract rather than to parts of a mesh.

For the purposes of this paper, Data objects reference one of two types of nodes: Ready and Lazy. *Ready* (or *non-lazy*) nodes store their values explicitly. *Lazy* nodes represent maths operations and link to other lazy nodes in a *directed acyclic graph* (DAG) to represent expressions. Special *identity* nodes are used to wrap ready nodes so they can be added to the graph. The acyclic property of the graph is guaranteed because a lazy node can only be formed using existing nodes. For example, if three Data variables P, Q, R which store their values in nodes $V1$, $V2$, $V3$ (Figure 1(a)), then the expressions:

$$S = P + Q$$

$$T = S/R$$

result in the DAG shown in Figure 1(b).

Once in the DAG, a node is never modified except when a sub-expression is replaced by a ready node. The *root* of an expression is the node directly referenced by the Data object (which might not be a root in the underlying DAG). The roots in Figure 1 are the nodes pointed to by dotted lines.

[1]This is particularly true when MPI is involved and the same sampleID may represent different entities on different machines.

[2]We follow the terminology from Python where the rank of an object is the number of indices required to identify a component. The shape of an object is the range of values for each index.

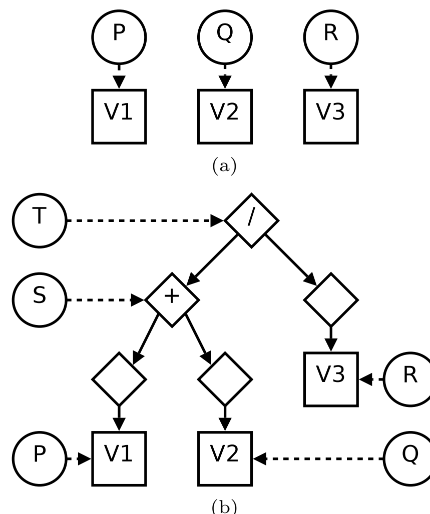


Figure 1: A DAG representing an expression. Circles represent Data objects, squares represent Ready nodes and diamonds represent lazy nodes. Empty diamonds represent identity operations. Solid lines are DAG edges, dotted lines show ownership of nodes.

The lifetimes of interrelated Domain, FunctionSpace and DataAbstract nodes are managed using shared pointers from *Boost* (boo 2009). So for example, Data have a shared pointer to their DataAbstract node while FunctionSpaces hold a shared pointer to their Domain. (*Boost* also allows us to make these wrappers transparent to Python.)

Sharing DataAbstract nodes between Data objects makes for efficient copy and return operations but it introduces complications.[3] When nodes are shared or incorporated into a lazy expression, they must keep their values from that point in time. Users should not need to be aware that a particular Data object uses shared values, so *escript* implements transparent copy on write (COW)(Glass & Ables 2003). Any Data method which modifies values, checks to see if it is the sole owner of the node. If not, it makes a copy of the node, transfers ownership to a it and modifies the copy instead.

4 Threading

EScript employs the following threading model. The Python layer (both user scripts and *escript* modules) is assumed to be single threaded. Some objects and methods are implemented in C++/C and these are parallelised using OpenMP and MPI.

No multithreading apart from OpenMP is used. Hence, there are no threading issues to consider apart from those within individual parallel regions. Because some of the memory allocation is based on the number of threads in use, we assume that the number of threads available to OpenMP does not change. For this reason we also do not use nested OpenMP parallel regions.

In theory, the sole owner requirement for COW could be checked using the `.unique()` method on *Boost* pointers but threading issues prevented this. Specifically, it proved difficult to differentiate between a node being passed from one owner to another and a node with two owners. Instead, we maintain a list of which Data objects own which nodes. To simplify

[3]“Copies” here refers to objects created at the C++ level (possibly indirectly by calling Python methods), not the Python assignment statement (which leads to two references to the same object).

matters involving large DAGs which encode a number of expressions, once a node becomes part of a lazy expression it is assumed to always be shared.

5 Lazy Evaluation

Lazy evaluation is "popular in functional programming languages (those with no effects) and rarely found elsewhere"(Friedman & Wand 2008). Hudak's 1989 survey (Hudak 1989) lists lazy evaluation as one of the distinguishing features of modern functional languages. His definition includes the idea that a particular expression be evaluated at most once.

The Haskell functional language (Thompson 1999) uses this form of laziness (outermost function application evaluated first and expressions only evaluated once) for its expression processing. For functions defined by conditionals, Haskell only performs enough evaluation to determine which branch of the condition to take. That is, it performs short circuit evaluation.

In the case of *escript*, there is a single conditional type operation (a masked copy) and it does not currently permit short circuit or lazy evaluation (although it could be simply modified to do so).

The main benefits of lazy evaluation for functional languages are (Thompson 1999)(Hudak 1989)(Pandey 2008):

1. Expressions need not be evaluated at all if they are not required.
2. Since lists do not always *need* to be represented explicitly, programs can be written to deal with conceptually infinite lists.

Our requirements differ from the functional setting. We do not need to process infinite objects. We do allow large objects to be examined a chunk at a time though, and our operations make use of this capability. However, user scripts must request this operation explicitly for their own use.

Our expressions only include values and operations from a predefined set. As such, they cannot contain arbitrary user defined functions. Our implementation links directly to values, so there are no names (functions or variables) which require forming a closure. We also differ from the functional version in that expressions may (depending on implementation strategy) be evaluated more than once. This is because caching the complete result is something we wish to avoid.

6 Resolution

Values of a function represented as lazy data can be resolved in two ways. Either the function can be queried one sample at a time or it can be *completely resolved*, that is all samples are evaluated and the result stored as a new function. Because of the extra memory required, we wish to avoid complete resolution as much as possible. Note that the process for assembling a sparse matrix to solve, only requires one sample at a time.

In the current build, Lazy data will be completely resolved when one of the following occurs:

- `.resolve()` is called on the expression.
- An attempt is made to set the value of the function at a point.[4]
- A masked copy is attempted.
- An operation is performed which depends on all points for its value, such as `integrate()`.
- The expression becomes too large.

[4] This operation is permitted in *escript* but not encouraged.

Apart from the first point, these may change in the future. The size constraint refers to either the total number of descendants or the height of the expression root (or both).

Some care must be taken when acting on lazy or shared data because *escript* may need to resolve or duplicate the data. In both cases memory may be allocated, deallocated and ownership may change. For this reason, we require that issues of resolution and node creation be settled before entering parallel sections. This restriction is only of concern to implementors.

We have implemented two strategies for evaluating samples in Lazy expressions. In both cases, the expression is evaluated using a recursive post-order traversal.

6.1 Temporary Buffer (Method 1)

A buffer is created and passed into the evaluation to function somewhat like a stack. To evaluate each node, space is reserved for the result and the remaining space is available to evaluate the node's children. Evaluation returns a pointer to the buffer (and an offset within the buffer) containing the result. This is to avoid copying data from non-lazy nodes under the Identity operation. The size required for the buffer is computed using a modified Sethi-Ullman register labelling algorithm (Appel & Palsberg 2002). Once the result has been retrieved, the buffer can be disposed of.

6.2 Per-Node Cache (Method 2)

Each node has a buffer big enough to store a single sample for each OpenMP thread. When the value of a sample for a lazy node is computed, it is stored in this buffer. Again we make an exception for identity nodes which wrap non-lazy values. The final value of the sample can be retrieved from the root node of the expression. The size limits on expressions described above were introduced primarily for Method 1 and should be relaxed or removed for this method.

7 Performance

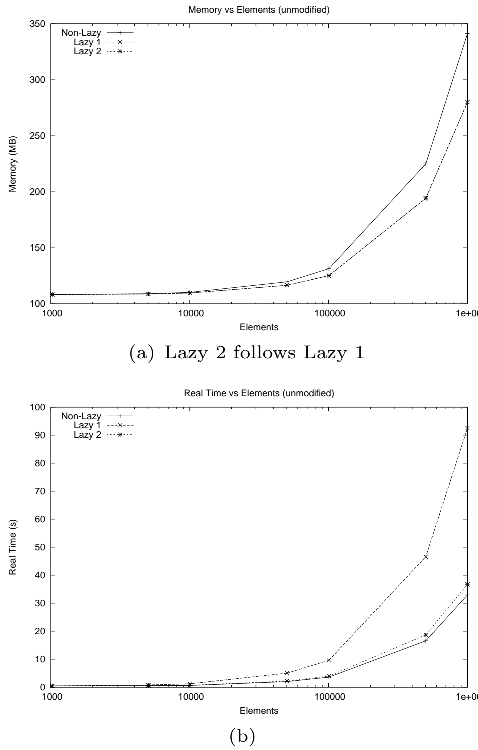
The experiments for this paper were carried out on a single compute node[5] of an SGI ICE 8200 EX. The code was compiled with support for OpenMP threading but not MPI. Please note that understanding the applications from which these tests are derived is not necessary to understand the performance changes.

7.1 Experiment 1 — Power Law

Here a script to compute power law values(Muhlhaus & Regenauer-Lieb 2005) was run for a single OpenMP thread[6]. See Appendix A for details of the script. Both lazy resolution methods were tested. Figure 2 shows the peak memory and real time use for runs with various numbers of elements. The values plotted are averaged over ten runs. The memory use for the two lazy methods is reasonably similar, although Method 1 (temporary buffer) is slightly smaller. The run time for Method 1 was significantly higher than non-lazy while Method 2 (node cache) was only slightly higher.

[5] 32 GB RAM and two quad-core 2.8Ghz Xeon processors. The version of *escript* was repository revision 2532.

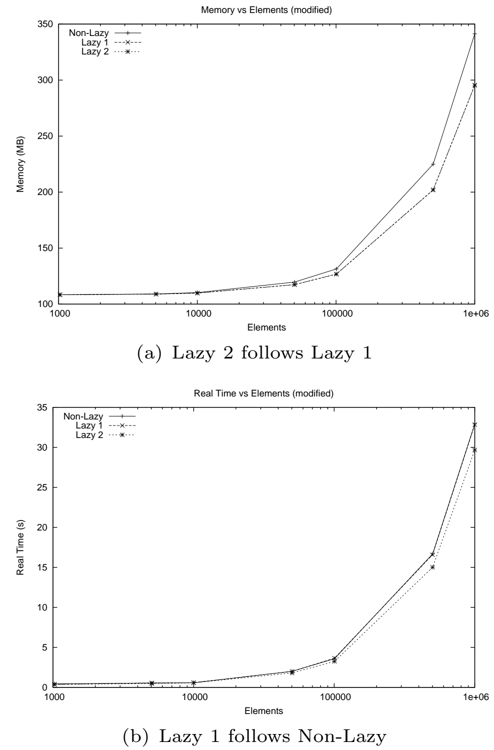
[6] That is, OMP_NUM_THREADS=1.



(a) Lazy 2 follows Lazy 1

(b)

Figure 2: Unmodified Powerlaw



(a) Lazy 2 follows Lazy 1

(b) Lazy 1 follows Non-Lazy

Figure 3: Modified Powerlaw

7.2 Repeated Sub-Expressions

The temporary buffer method uses less memory than the second method because the same storage can be used to evaluate multiple nodes. In memory intensive problems though, both methods use less memory than the non-lazy method.

The run time for Method 1 can be reduced by forcing nodes which appear multiple times to be resolved first before resolving the main expression. This means that the values can be retrieved directly instead of being recomputed. This has two drawbacks:

1. More memory is required to hold the extra resolved expressions. This is particularly significant if the results are of high rank. For example the Drucker-Prager tests in Section 7.3 contain rank 4 tensors.

2. The user must examine their expressions to determine suitable variables for early resolution and the order in which to do so. For example if f is a function of g (and both are repeated expressions), then g should be resolved first. If not, the work to evaluate g will be done twice.

Next we make two changes to the computation. Computing the power law values builds an expression $\eta(\theta+\omega)/(\eta\theta^2+\omega)$. We resolve η, θ, ω before building the main expression.

Secondly, we rearrange calls to the L^∞-norm so that sub-expressions are evaluated first. The performance for this modified version can be seen in Figure 3. Memory use for lazy resolution is still below non-lazy resolution. The time spent for Method 1 now approximates non-lazy time, while Method 2 is lower. Note that the memory usage in Figure 3(a) is worse than in Figure 2(a).

7.3 Experiment 2 — Drucker-Prager

A script (see Appendix B) to compute coefficients for Drucker-Prager flow (Gross et al. 2008) was run for a single OpenMP thread. Only Method 2 (node cache) was used for lazy testing. See Figures 4, 5 for results (values shown are averaged over ten runs). In both 2D and 3D domains, Method 2 shows significantly lower costs in both time and space. In Figure 5(a) the non-lazy version could not complete the 512, 000 element test (presumably due to exhausting available memory).

8 Discussion

When the two lazy evaluation methods are assessed against our goals (memory, time and ease of use), we have some success. For less complex expressions such as those in the power law tests, there is some reduction in memory use. There is a time penalty involved however, so employing lazy evaluation for this problem would only be of benefit in cases where memory consumption is close to system limits.

The presence of repeated subexpressions seems to severely limit the usefulness of Method 1 (temporary buffer). As we showed in Section 7.2, these problems can be reduced be strategic calls to `resolve()` and reordering of expressions but this comes at the price of ease of use. Adding additional resolves also increases memory useage (contrast Figure 2(a) with Figure 3(a)). Deciding on the size limits for expressions also requires some care.

On the other hand when used on Drucker-Prager, Method 2 (node cache) showed significant reductions in both memory and time for large instances. In fact it rendered larger instances accessible.

Currently, lazy evaluation in *escript* can be switched on or off at runtime. Doing this with Method 2 seems to be the best policy.

Further work in this area should be directed to making automatic resolution more intelligent and providing more specific guidance for users as to when a problem is “large enough” for lazy evaluation to provide significant benefits.

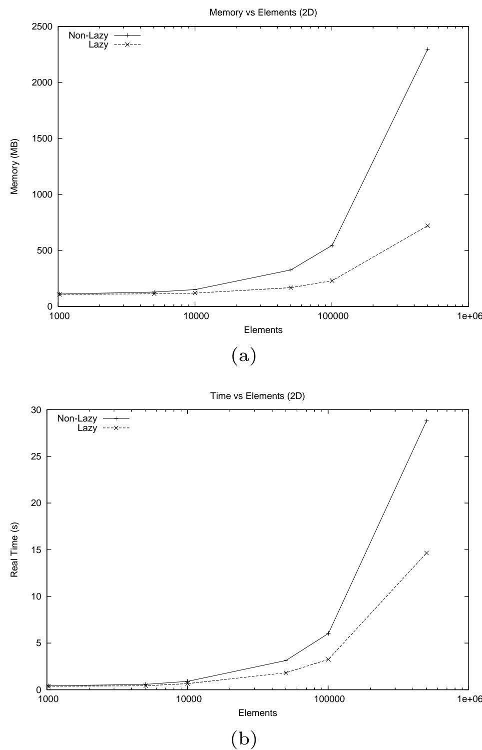


Figure 4: Drucker-Prager 2D

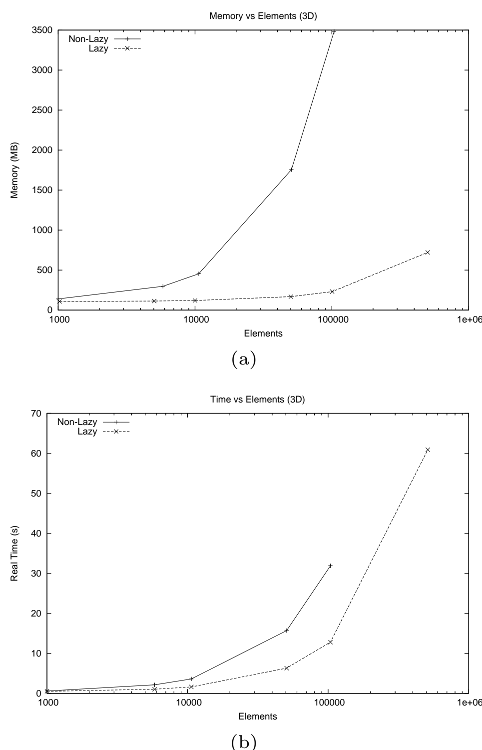


Figure 5: Drucker-Prager 3D

References

Appel, A. W. & Palsberg, J. (2002), *Modern compiler implementation in Java*, second edn, Cambridge University Press.

boo (2009), 'Boost C++ libraries', `http://www.boost.org/`.
URL: *http://www.boost.org*

Friedman, D. P. & Wand, M. (2008), *Essentials of Programming Languages*, third edn, MIT Press.

Glass, G. & Ables, K. (2003), *Unix for Programmers and Users*, third edn, Pearson Education.

Gross, L., Cumming, B., Steube, K. & Weatherley, D. (2007), 'A python module for pde-based numerical modelling', *PARA* **4699**, 270–279.

Gross, L., Muhlhaus, H., Thorne, E. & Steube, K. (2008), *Earthquakes : Simulations, Sources and Tsunamis*, Pageoph Topical Volumes, Birkhäuser Basel, chapter A New Design of Scientific Software Using Python and XML, pp. 653–670.

Hudak, P. (1989), 'Conception, evolution, and application of functional programming languages', *ACM Comput. Surv.* **21**(3), 359–411.

lpe (2009), 'escript-finley', `https://launchpad.net/escript-finley`.

Muhlhaus, H.-B. & Regenauer-Lieb, K. (2005), 'Towards a self-consistent plate mantle model that includes elasticity: simple benchmarks and application to basic modes of convection', *Geophysical Journal International* **163**(2), 788–800.

omp (2009*a*), 'OpenMP specification', `http://openmp.org/`.

omp (2009*b*), 'OpenMP tutorial', `https://computing.llnl.gov/tutorials/openMP/`.

Pandey, A. K. (2008), *Programming Languages, Principles and Paradigms*, Alpha Science.

Thompson, S. (1999), *Haskell, The craft of Functional Programming*, second edn, Addison-Wesley.

A Power Law script

This is the script used in the power law tests (some boilerplate and irrelevant lines removed). The `NE` variable should be set to the number of elements required.

```
SIDE=int(math.ceil(float(NE)**(1./2)))
setEscriptParamInt("TOO_MANY_LEVELS",15)
setEscriptParamInt("TOO_MANY_NODES",500)

d=Rectangle(SIDE,SIDE).getX()+(1,1)
pl=PowerLaw(numMaterials=3, verbose=False)
pl.setDruckerPragerLaw(tau_Y=100.)
pl.setPowerLaws(eta_N=[2.,0.01,25./4.], \
   tau_t=[1, 25.,64.], power=[1,2,3])
pl.setEtaTolerance(1.e-8)
z=pl.getEtaEff(length(d))
z.resolve()
```

B Drucker Prager script

This is the script used in the drucker prager tests (some boilerplate and irrelevant lines removed). The `domain` variable should be a `Rectangle` (or `Brick` for 3D) with the required number of elements.

```
setEscriptParamInt("TOO_MANY_NODES",10000)
setEscriptParamInt("TOO_MANY_LEVELS",70)

G=10.
K=12
alpha=0.2
beta=0.03
h=1.
deps_th=0.1

stress=Tensor(1.,Function(domain))
tau_Y_safe=Scalar(13.,Function(domain))
tau_Y=Scalar(13.,Function(domain))
du=domain.getX()
plastic_stress=Scalar(0.,Function(domain))

d=domain.getDim()
abs_tol=1.e-15
SAFTY_FACTOR=1.e-8
k3=kronecker(Function(domain))
# elastic trial stress:
g=grad(du)
D=symmetric(g)
W=nonsymmetric(g)
s_e=stress+K*deps_th*k3+2*G*D+(K-2./3    \
       *G)*trace(D)*k3+2*symmetric(      \
       matrix_mult(W,stress))
p_e=-1./d*trace(s_e)
s_e_dev=s_e+p_e*k3
tau_e=sqrt(1./2*inner(s_e_dev,s_e_dev))

F=tau_e-alpha*p_e-tau_Y
chi=whereNonNegative(F+SAFTY_FACTOR*tau_Y)
l=chi*F/(h+G+beta*K)
tau=tau_e-G*l
stress=tau/(tau_e+abs_tol*               \
         whereZero(tau_e,abs_tol))       \
         *s_e_dev-(p_e+l*beta*K)*k3
plastic_stress=plastic_stress+l

hardening=(tau_Y-tau_Y_safe)/(l+abs_tol* \
         whereZero(l))
sXk3=outer(stress,k3)
k3Xk3=outer(k3,k3)
s_dev=stress-trace(stress)*(k3/d)
tmp=G*s_dev/(tau+abs_tol*                \
         whereZero(tau,abs_tol))

S=G*(swap_axes(k3Xk3,0,3)+               \
     swap_axes(k3Xk3,1,3)) + (K-2./3*G)  \
      *k3Xk3 + (sXk3-swap_axes(          \
      swap_axes(sXk3,1,2),2,3))          \
      + 1./2*(swap_axes(swap_axes(sXk3,  \
        0,2),2,3)                        \
      -swap_axes(swap_axes(sXk3,0,3),2,3)\
      -swap_axes(sXk3,1,2)               \
      +swap_axes(sXk3,1,3))              \
     - outer(chi/(hardening+G+alpha*beta
      *K)*(tmp+beta*K*k3),tmp+alpha*K*k3)

S.resolve()
tau.resolve()
stress.resolve()
plastic_stress.resolve()
```