Implementation of an Automated Benchmarking module for Kratos

Farshid Mossaiby, Dec. 2008

Contents

1. Introduction
2. Features
3. Implementation Details
4. Adding automatic benchmarking capability to existing test examples
5. References

Introduction

The size and complexity of software projects have increased considerably in the last years. Huge projects may involve hundreds or thousands of people working simultaneously on the same code base. This has increased the importance of Revision Control Systems. Revision Control Systems keep track of all changes done in a source code. More sophisticated systems may also be able to keep track of branches in the code. Without such tools, the development of huge softwares, e.g. operating systems would not be possible.

The Revision Control Systems help, among other things, minimizing the cost of source code maintenance and management. This can be very critical for large projects. Also, these systems help finding out bugs introduced in the code base while adding new features or fixing other bugs. For example, consider a scenario of adding a new feature in a program. This may involve changing the behavior of a previously used function. This may impact other parts of the code base which use it.

The Kratos project [1] at CIMNE is an open source Multi-physics software based on the Finite Element Method. Kratos due to its multi-disciplinary nature contains a large collection of numerical algorithms, Finite Elements, etc. It is being actively used and developed in CIMNE and also worldwide.

Kratos currently uses CVS (Concurrent Versions System) [2] as its Revision Control System. Users can fetch the most current version of Kratos via CVS, and use / modify / improve it. Also, users can commit modifications / improvements back to the code base, without much effort.

While this open architecture of Kratos has great advantages, it may cause some problems. For a simple example, consider a user commits a change which affects a wide range of users. As the users regularly update their local code base to get the newest updates, the committed change propagates, and as a result, compilation of a previously working program may fail or results of a program may change. In former case, the existence of a conflict can easy be detected, but in the latter, it may be hidden for a long time, and lead to confusing results.
Although there is no reliable way to prevent such conflicts, and also it is very hard to distinguish them from other improvements done in the code base which may also lead to change of the results, a practical technique is needed to minimize the impact of changes by a user on the others. There are a number of test examples in the Kratos, which can be regarded as a representative of the code base functionality. A major change in the behavior of these test examples may indicate a possible conflict, and needs to be reviewed carefully.

The steady and careful checking process of the test examples is almost impossible for developers and users of Kratos, and needs to be automated. This has been the inspiration of development of an *Automated Benchmarking* module. During the nightly builds of Kratos, which will be done on the upcoming Kratos server, this module automatically runs all test examples and verifies the results against a reference data set. It will send an email containing the status of each test example to a list of recipients. This can be useful in several ways. First, it will be clear when a conflict has occurred. This will help greatly to find the user committed the changes, ask him / her to correct / undo it. Also, it will let other users to be aware of a conflict in the latest code base, and avoid updating their own local code base, before it has been corrected. This can save them a lot of time and effort.

**Features**

The module was implemented in Python programming language. Kratos already uses Python for its interface to the end-users, and hence, the developers and users of Kratos are already familiar with it. It also eases the process of adding this automatic benchmarking capability to the existing test examples, as well as extension of the module itself. Development of the module in Python will help its platform independence, as Kratos itself is intended to be.

The automatic benchmarking module can also check the performance of various parts of the benchmarked program. It can be really helpful, as a committed change, still may lead to correct results can impact the performance of a program. The automatic benchmarking module will thus help identify major increase / decrease of the performance of Kratos.

The automatic benchmarking module detects the special mode used for benchmarking and will avoid interfering with normal execution of the test examples. This way, the users and developers do not need to enable / disable the benchmarking capability of the tests for benchmarking / normal

**Implementation details**

To verify the behavior of test examples in Kratos, some generated data by the test example must be verified, while letting small changes in the values because of improvements in the other parts of the code base. To achieve this, an interface between the test example and the automatic benchmarking module is needed. The provided interface must be easy to use, and general. Through this interface, the test example emits some values that can be stored as reference data set or checked against one to discover possible differences.
To the benchmarking module runs the test example in a special mode, namely benchmarking mode. This is simply achieved by adding --benchmarking switch to the command line of the test example. This switch is checked in the functions exposed by the automatic benchmarking module to avoid interfering with execution of a program in normal operation mode.

Table 1 lists the functions exposed by the automatic benchmarking module, along with a short description. These functions provide the interface between the module and the test examples. Also, some of these functions are used in scripts provided for automatic running of all test examples, automatic building of all reference data files or notifying the recipients from the status of the test examples. Here a brief usage example for these functions will be presented.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>InBenchmarkingMode()</td>
<td>Returns True if in benchmarking mode, False otherwise</td>
</tr>
<tr>
<td>Output(Var, Label = &quot;&quot;, AbsErr = None, RelErr = None)</td>
<td>Emits the value of Var, with a given label, absolute and relative error</td>
</tr>
<tr>
<td>StartTiming()</td>
<td>Returns the current time</td>
</tr>
<tr>
<td>StopTiming(t, AbsDiff, RelDiff)</td>
<td>Emits the difference between the current time and t (returned previously by StartTiming()), specifying absolute and relative difference allowed</td>
</tr>
<tr>
<td>BuildReferenceData(ExamplePath, ReferenceFile)</td>
<td>Runs the test example given in ExamplePath in benchmarking mode, and stores the values emitted in ReferenceFile</td>
</tr>
<tr>
<td>RunBenchmark(ExamplePath, ReferenceFile)</td>
<td>Runs the test example given in ExamplePath in benchmarking mode, and compares the values emitted with those in ReferenceFile, returns True if no difference encountered, or a message with some details on the difference found</td>
</tr>
<tr>
<td>NotifyViaEmail(Subject, Text, Recipients)</td>
<td>Sends an email to the given list of recipients, with given subject and text</td>
</tr>
</tbody>
</table>

**Table 1 Functions exposed by the automatic benchmarking module**

InBenchmarkingMode():

This function simply returns True to indicate that the test example is being run in benchmarking mode. This can be helpful, if you only need to calculate some values for use in benchmarking only:

```python
if (benchmarking.InBenchmarkingMode()):
    ...calculate some results...
    benchmarking.Output(...)  
```

Output():

Register for free at https://www.scipedia.com to download the version without the watermark
This function is the mostly used part of the interface. It emits the value of `var` to the module. This value can be an integer or floating number, as well as a string or a boolean. You may optionally specify a label, which will help you when something goes wrong. You can also specify absolute or relative tolerance for numbers (integer of floating). These tolerances default to `None`, which means they will not be checked. Please note that the relative tolerance is not in percents, and also you may not specify a relative tolerance for an exact value of zero. Here are some examples of usage of this function:

```python
benchmarking.Output(i)
benchmarking.Output(i + j > 7)
benchmarking.Output(dt, "Delta t")
benchmarking.Output(p, "Node 1 pressure", None, 0.05)
```

In the first example, the value of `i` has been emitted without any label or tolerances. In the second one, a boolean value has been emitted, again with no label. In the third one, the value of `dt` has been emitted, specifying a label for it. And in the last one, the value of `p` has been emitted, specifying a label, no absolute tolerance, but a relative tolerance of 5%.

**StartTiming(), StopTiming():**

These functions together help you control the running time of portions of your code. These portions can be nested. Here also you can specify absolute and relative tolerance permitted. The usage is very simple:

```python
t = benchmarking.StartTiming()
... the portion of the code to be checked ...
benchmarking.StopTiming(t, None, 0.05)
```

In the above example, no absolute tolerance specified, while a 5% relative tolerance allowed.

**BuildReferenceData(), RunBenchmark():**

These two functions are only used in the scripts provided for automatic running of all benchmarks or automatic building of all reference data files. The usage is simple; just give name of the test example script, and the reference data file. The former function will run the example in benchmarking mode, storing emitted values in given reference data file, while the latter checks the emitted values against the given reference data file. Both functions assume that `python` and `grep` are available. Due to sensitivity of Kratos examples to the path they are called from, it is strongly recommended to change your working directory to the where the test examples reside, and then call the above functions.

**NotifyViaEmail():**

This function is used to send an email with given subject and body text. It is used in the scripts provided for automatic running of all test examples, and reporting the status of each example. This function always sends the email from no-reply-kratos-benchmarking@cimne.upc.es, and hence you might need to add this
address to your address book / white list to avoid delivery of the emails to Bulk / Spam folder. The usage pattern is:

```
benchmarking.NotifyViaEmail("Subject here!", "Body here!", ["user1@somewhere.net","user2@nowhere.org"]
```

**Adding automatic benchmarking capability to existing test examples**

Integrating automatic benchmarking capability in existing test examples is very simple. One needs to add lines marked in **bold** to his / her code first:

```
#ATTENTION: here the order is important
#including kratos path
kratos_libs_path = '../../../../libs' ##kratos_root/libs
kratos_applications_path = '../../../../applications' ##kratos_root/applications
kratos_benchmarking_path = '../../../../benchmarking' ##kratos_root/benchmarking

import sys
sys.path.append(kratos_libs_path)
sys.path.append(kratos_applications_path)
sys.path.append(kratos_benchmarking_path)

#importing Kratos main library
from Kratos import *
kernel = Kernel()   #defining kernel

#importing applications
import applications_interface
applications_interface.Import_IncompressibleFluidApplication = True
applications_interface.Import_PFEMApplication = True
applications_interface.ImportApplications(kernel, kratos_applications_path)

import benchmarking

```

The next step would be selecting some key values in your problem which can almost reliably indicate the behavior of your code. This can be displacement or pressure of a given node, for example. To do this, you may first need to find the nodes using a helper function, like this:

```
def FindNode(node_list, x, y, z):
    for node in node_list:
        if ((node.X - x) ** 2 + (node.Y - y) ** 2 + (node.Z - z) ** 2 < 0.0000001):
            return node
```

Also, a helper function that emits the value can be helpful:

```
def BenchmarkCheck(time, node1, node2):
```
In the main program, you first find the nodes, for example:

```python
node_1 = FindNode(model_part.Nodes, 0.5, 0.0, 0.0)
node_2 = FindNode(model_part.Nodes, 0.24, 0.0, 0.0)
```

And in the main loop you call the former function as:

```python
BenchmarkCheck(time, node_1, node_2)
```

You may find more information about the usage of the automatic benchmarking module as well as other scripts provided / needs to be provided for automatic running of all benchmarks / building all reference data files in Kratos wiki [??].

References