CREDO Documentation

Release 0.1.3

Patrick Sunter (editor)

October 28, 2011

CONTENTS

1 Introduction to CREDO **3**
1.1 Core design goals . . . 3
1.2 How CREDO fits into the workflow of using the Underworld modelling code . . . 3
1.3 CREDO's Benchmarking Goals / Motivation . . . 4
1.4 Scientific analysis of computational codes - core capabilities . . . 5
1.5 Language choice - why Python? . . . 5

2 Installation & setup quickstart instructions **7**
2.1 Installing Dependencies and Options . . . 7
2.2 Required- Python: 2.5 or 2.6 . . . 7
2.3 Setting up CREDO - Temporary instructions for beta users . . . 8

3 Core CREDO Architecture **11**
3.1 Organisation of Python Code and class hierarchy . . . 12

4 Examples of how to use CREDO **15**
4.1 Using CREDO for System Testing of StGermain-based codes such as Underworld . . . 15
4.2 Doing Model analysis with CREDO . . . 28
4.3 Scientific Benchmarking using CREDO . . . 38
4.4 Different ways to launch CREDO scripts . . . 42

5 CREDO Python Source API Documentation **45**
5.1 Model API . . . 46
5.2 Utils API . . . 56
5.3 JobRunner API . . . 56
5.4 IO (Input Output functions) API . . . 61
5.5 Analysis API . . . 68
5.6 SysTest API . . . 74

6 Links to useful Python info, such as tutorials **91**
6.1 General Python tutorials / skills . . . 91
6.2 Tutorials more for the appropriate 'software engineering' in Python . . . 91
6.3 Python libraries of interest for scientific analysis . . . 92

7 What's new in CREDO **93**
7.1 new in credo-0.1.3 . . . 93
7.2 new in credo-0.1.2 . . . 94
7.3 new in credo-0.1.1 . . . 94

8 CREDO Appendix (inc Developer notes) **97**

8.1 Notes for CREDO developers / maintainers . . . 97
8.2 How to build CREDO Documentation locally . . . 98
8.3 Process to follow for creating new CREDO releases . . . 98

9 CREDO FAQ **101**
9.1 CREDO's capabilities . . . 101
9.2 Problems running tests . . . 101

10 Glossary of Terms **103**

11 Indices and tables **105**

Bibliography **107**

Python Module Index **109**

Index **111**

This documentation describes the CREDO scientific benchmarking, testing, profiling and analysis toolkit. It comprises several sections, outlined below, for different purposes.

For users of CREDO: we suggest you start by reading the *Introduction to CREDO* section, then the *Installation & setup quickstart instructions* section to get up and running, followed by running through several of the *Examples of how to use CREDO*. The examples should be self-explanatory, but if you're not familiar with the Python programming and scripting language, you might like to check out the *Links to useful Python info, such as tutorials* section.

CREDO is currently being used as the Testing and workflow support tool of the Underworld Geodynamics modelling project:- the manual of which is available online.

For developers of CREDO:, you'll likely want to jump straight to the *CREDO Python Source API Documentation*, informed by the overall description in *Core CREDO Architecture*.

Contents:

CHAPTER

ONE

INTRODUCTION TO CREDO

CREDO is a Python toolkit for system testing, benchmarking and analysis of Modelling tools based on the StGermain framework, such as Underworld.

It is distributed with the *stgUnderworld* application bundle, but in future it's planned that it will also be possible to be obtained and installed independently.

1.1 Core design goals

CREDO's core design goals are to:

- Support the effective development, inquiry and maintenance of a suite of benchmarks that test the scientific features, numerical accuracy, and computational performance of StGermain-based codes.
- Significantly enhance the ease and effectiveness of performing scientific analysis of the results of StGermain modelling applications.

Both these tasks are possible using a combination of command lines and custom 3rd party tools, both proprietary and open source - but the goal of CREDO is to provide an integrated system that makes doing both of them to a high level much easier and clearer - and also in a readily repeatable manner so that the status of a code as it evolves can be readily checked (e.g. as part of a continuous integration system).

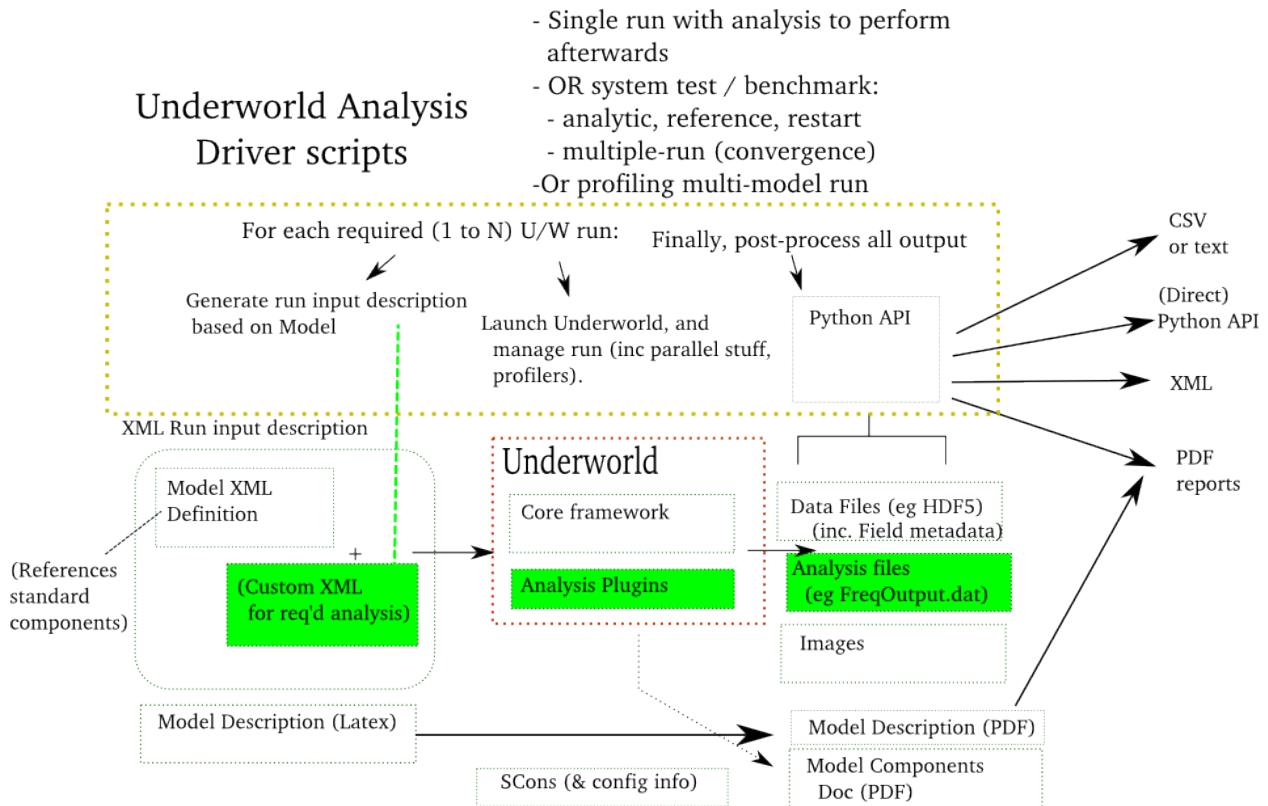
To meet these goals, we chose the *Python* scripting and programming language, explained more in the *Language choice - why Python?* section below.

1.2 How CREDO fits into the workflow of using the Underworld modelling code

CREDO still runs StGermain codes such as Underworld "under the hood", but it's role is to:

- Construct XML data files based on the analysis/testing the user asks for in a CREDO script;
- Launch the necessary StGermain jobs required to perform the analysis;
- ..and finally provide access to the results created by the model at a high-level, facilitate post-processing of these results, and perform any post-run analysis required by tests/benchmarks.

This is summarised in the diagram below.



As such, it currently explicity doesn't provide run-time access to the StGermain objects (written in a custom framework implemented in the C language). Rather, it works in detail with the StGermain XML format for defining models, and the defined StGermain data output formats.

1.3 CREDO's Benchmarking Goals / Motivation

There has long been an interest in a more systematic way of testing both the software performance, and scientific/numerical reliability of the StgUnderworld code – for example the issue was explored in a paper presented at the APAC05 conference [FarringtonEtAl2005].

The benefits of such a system are:

- A clear and up-to-date record of the performance and reliability of the code to present to scientists interested in using it or comparing it to similar codes. The chosen benchmarks would be of a "scope" of much more interest to modellers and scientists than the detailed unit and integration tests. This ties in to the concept of 'reproducible research' ([FomelHennenfent2007]).
- A definitive record to examine the changing performance of the code over time, which has always been a big concern among the existing research community using the code;
- As a corollary to the above, the system would give the development team a clear and timely warning if a design change accidentally negatively affected performance or accuracy – which has been a problem in the past.
- The ability to use the science benchmark suite and framework to quickly evaluate the performance of StgUnderworld on new hardware systems, to assist purchasing decisions and new collaborators.
- The goal would be to automate as much as possible the regular collection of these benchmarks (initally proposed as weekly), assessment of them, and publishing of them on-line in a succinct and meaningful form (IE with key metrics emphasised, and linked to clear definitions of the benchmark problem).

1.4 Scientific analysis of computational codes - core capabilities

To achieve the above goals, the features below are either provided by CREDO, or under active development:

- Ability to quickly extract and compare observables that are produced during runs of the code (such as *VRMS*). This includes testing that an observable is within a given range at a given time.
- Ability to extract information about the 2D or 3D *mesh*, *field* or *swarm* data produced by a StGermain application.
  - For testing purposes, this includes functions like checking that a field produced by the model run compares as expected with a reference field, loaded in from a separate data file.
- Ability to record, and compare performance metadata about the code, such as time that simulations ran for, memory usage, etc.

Examples of doing this sort of analysis and testing is provided in the *Examples of how to use CREDO* section.

1.5 Language choice - why Python?

The CREDO code is written in the dynamic scripting and programming language **Python**. Python was chosen as the implementation language because of:

- It's ability to run in either interactive or scripted mode:- and thus facilitate either scripted, repeatable workflows, or interactive exploration;
- The fact that it's clear, concise syntax and high level of abstraction is recommended for human developer productivity - while computationally intensive tasks can be performed in compiled languages and libraries (such as Underworld itself).
- The fact that it's a highly portable language between operating systems and architectures.
- The increasingly stable, feature-rich and wide-ranging set of open source packages for mathematical and scientific analysis in Python, such as SciPy, NumPy, Matplotlib, SAGE, Paraview, and MayaVI.

See Also:

Links to useful Python info, such as tutorials

The fact that CREDO is written in Python doesn't prevent you from using a favourite tool or language for your final analysis work - in this case, CREDO is being written to allow you to extract the needed observables from a set of model results in common formats such as CSV or XML files.

CHAPTER

TWO

INSTALLATION & SETUP QUICKSTART INSTRUCTIONS

As discussed in the introduction, the main way of distributing and obtaining CREDO is as a core part of the *stgUnderworld* geophysics framework.

So setting up to use CREDO is very similar to setting up stgUnderworld, except for some extra Python configuration at the conclusion.

2.1 Installing Dependencies and Options

CREDO has been designed to operate with just Python as a core dependency - other libraries for more advanced functionality are optional, although are recommended to use the full functionality.

2.2 Required- Python: 2.5 or 2.6

CREDO is written to work with Python 2.5 or 2.6. One of these versions should come pre-installed on most Linux systems (including supercomputees), Mac OS machines, and clusters. For Windows, you may need to download and install Python yourself - check out the Python website for instructions and documentation.

2.2.1 NB: Python and scientific library "super set installs"

For users with a Mac machine or intending to run analysis on a cluster, you may wish to consider one of the "scientific Python" distributions, which will install Python, plus a large set of useful scientific libraries including those mentioned below, as a group in one large install package. The idea is this takes away some of the pain of individually installing these libraries, and making sure you have compatible versions installed. Some of the leading "install sets" of this type are:

- SAGE Math: SAGE is a collection of open source Python libraries, aiming to provide a free open source alternative to MATLAB etc. As well as the libraries, it provides a customised Python interpreter, that includes a more mathematical syntax for defining math functions etc.
- Enthought Python Distribution: Enthought are a commercial company and provide their Python distribution with a per-user licence model, but also have an academic licence for trials of the software.

Note: we don't expressly recommend either of these packages at this time, but provide them as alternative options to the regular install process. As mentioned above, they may be especially relevant to Mac machines, where installing several of the optional packages used by CREDO is non-trivial.

2.2.2 Optional Packages

These packages are optional, they provide extra capabilities to CREDO such as plotting and visualisation of output, but are not essential for running system tests and doing basic analysis.

- **Matplotlib**: http://matplotlib.sourceforge.net: A good plotting Library available in Python. CREDO has several functions/options to auto-create plots of things of interest using Matplotlib. On Mac Os X, this is available via Macports for different Python versions, such as *py26-matplotlib*.
- **NumPy** and **SciPy**: http://numpy.scipy.org/ and http://www.scipy.org/. These libraries are mature and provide efficient and effective interfaces for operating on Numerical data. CREDO doesn't use either explicitly currently, but they may be useful for doing custom analysis on StGermain data made available by CREDO.
- Visualisation: **ParaView** or **MayaVI**: http://www.paraview.org/, http://mayavi.sourceforge.net/: these two open-source visualisation packages, both based on the VTK toolkit, provide considerable capabilities for operating on the sorts of 2D and 3D datasets StGermain-based applications produce, and interact well with Python. We are considering providing explicit integration support for one or both of these packages in future.

2.3 Setting up CREDO - Temporary instructions for beta users

Currently, CREDO is primarily designed to be distributed with the stgUnderworld application bundle. It is possible to install it on its own, but currently the instructions below focus on this main use case.

2.3.1 obtaining CREDO as part of stgUnderworld

To obtain CREDO as part of stgUnderworld:

1. Check out the stgUnderworld codebase, from https://www.mcc.monash.edu.au/hg/stgUnderworld. (Check out instructions at the Underworld download page if unsure how to do this.

   e.g. use the following command:

   ```
   hg clone https://www.mcc.monash.edu.au/hg/stgUnderworld
   ```

2. run obtainRepositories.py to download all the sub-packages, including CREDO:

   ```
   ./obtainRepositories.py
   ```

- (You may have to submit your repository authentication details while cloning both repositories such as Experimental above, if you have specified to download them.)

3. Configure and build the codebase as normal using scons, as detailed on the Underworld website.

You should now have a working version of the stgUnderworld codebase installed, including CREDO.

2.3.2 Setting up your environment to use CREDO

Note: If you only intend to run CREDO System tests via SCons commands like *./scons.py check* (see *Running a test target, or test suite, via the SCons build system*), then you don't need to read the section below, as CREDO is now integrated with SCons. However, the environment variables are needed if you want to run CREDO tests directly.

To run any CREDO scripts directly, you need to modify a couple of shell environment variables.

These variables are:

Variable	Value to set to
PATH	needs to be extended with a reference to the credo/scripts directory in your checkout.
PYTHON-PATH	needs to be extended to reference the main tree of CREDO python code (credo/credo)
STG_BASEDIR	specifies the base directory that StGermain has been checked out to. Optional, can individually specify the variables below instead if necessary.
STG_BINDIR	needs to specify the path that StGermain executables have been compiled and installed to. For a default installation, you can just use STG_BASEDIR instead and CREDO will work out the binaries location within that.
STG_XMLDIR	needs to specify the path that StGermain standard XMLs are stored in when the code is compiled. For a default installation, you can just use STG_BASEDIR instead and CREDO will work out the XMLs location within that.

There are also several environment variables specific to different 'job launchers' that you can use with CREDO, starting with the default MPI one:

MPI Jobrunner (default)

Variable	Value to set to
MPI_RUN_COMMAND	(Optional) this will set the command used to run parallel jobs using MPI. Otherwise the default of "mpirun" will be used.

The sections below will advise you how to set these up correctly.

Modifying the shell variables directly

If you would like to manually set up these environment variables, just first work out the correct values, and set them in your shell. E.g. if your stgUnderworld checkout with CREDO included was located at ~/AuScopeCodes/stgUnderworldE, then in Bash you would type:

```
export PATH=$PATH:~/AuScopeCodes/stgUnderworldE/credo/scripts/
export PYTHONPATH=$PYTHONPATH:~/AuScopeCodes/stgUnderworldE/credo/credo/
export STG_BINDIR=~/AuScopeCodes/stgUnderworldE/build/bin/
```

You might like to then save these lines to a config file for when you log in.

Updating and sourcing the provided bash config file in stgUnderworld

Alternatively, a Bash script that does all the necessary exports once you specify one single path, has been included as *updatePathsCREDO.sh* in the base directory of the stgUnderworld repository.

So you can just source this file into your environment each time you want to start a session and use CREDO:

```
source updatePathsCREDO.sh
```

you will then be ready to use CREDO.

2.3.3 Testing you're set up correctly to use CREDO

It's easy to test if these environment variables have been set up correctly - just open a Python script and test that you can import CREDO:

```
psunter@auscope-02:~/AuScopeCodes/stgUnderworldE$ python
Python 2.6.4 (r264:75706, Dec  7 2009, 18:43:55)
[GCC 4.4.1] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import credo
>>>
```

No message is the expected result, it means the credo package was successfully loaded.

If there's an error, you will see something like:

```
[GCC 4.4.1] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import credo
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ImportError: No module named credo
>>>
```

...which means you need to go back through the steps - most likely it's a problem with the setup of the environment variables above.

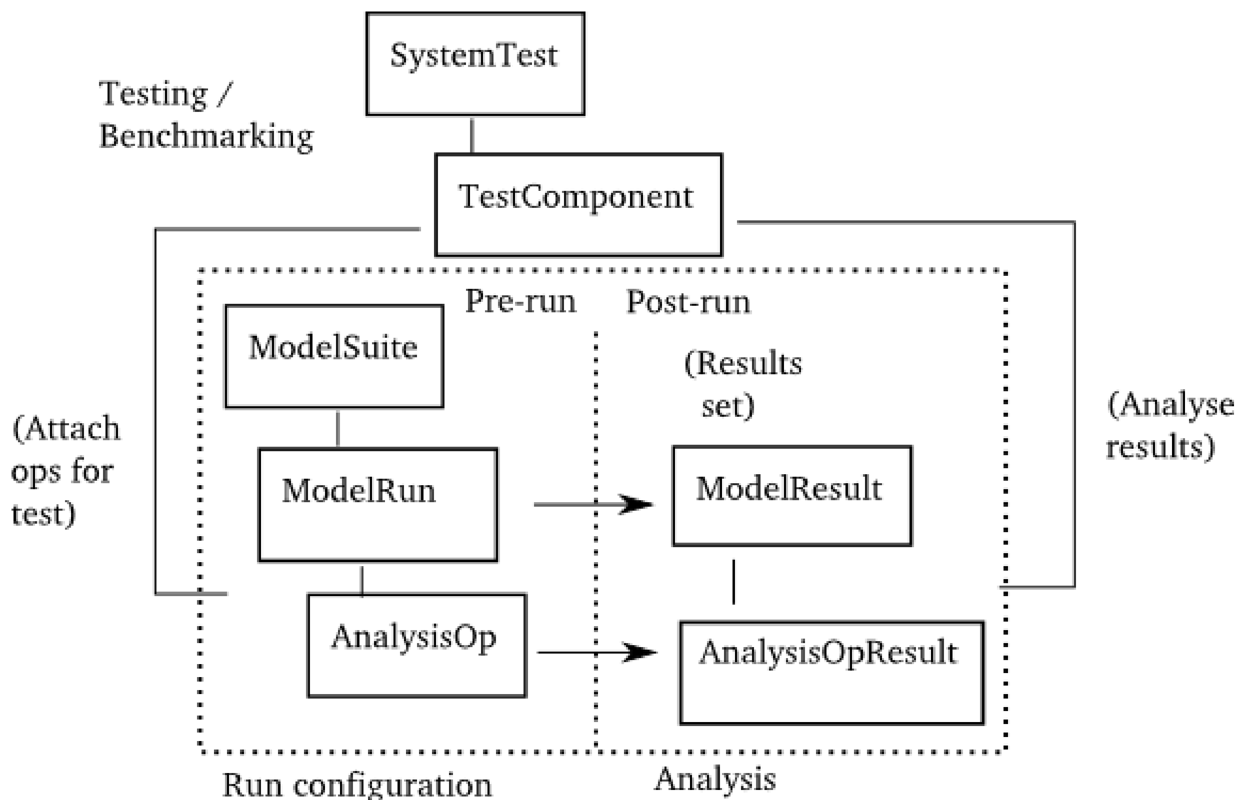
CHAPTER

THREE

CORE CREDO ARCHITECTURE

This section is aimed to give you a quick overview of the architecture of the CREDO toolkit. It's strongly suggested you read at least the *How CREDO fits into the workflow of using the Underworld modelling code* section from the introductory section first, to put this in perspective. This architecture description may help illustrate the *Examples of how to use CREDO*, and help explain the details in the *CREDO Python Source API Documentation*.

This diagram gives an idea of the essential architecture of the CREDO toolkit's classes:



The diagram shows that the CREDO framework works at two main levels:

- Running and analysis of models (inside the dotted box): ability to construct a *Model Run*, run it, and analyse the results.
- Testing and benchmarking: run and analyse models in defined ways, to allow system testing. Thus, if CREDO is being used purely for testing, it can operate at a high-level of abstraction and manage the details of constructing and analysing models [1].

[1] Although for complex benchmarks, the user will likely need to work with and customise the models at a more detailed level, which is supported.

Inside the dotted lines are shown the core Analysis functionality and classes. They implement the concept discussed in the *How CREDO fits into the workflow of using the Underworld modelling code*, of the ability to abstract and manage a StGermain-based code simulation through the concepts of a *Model Run* and *Model Result*. For each ModelRun, one or more analysis operations (AnalysisOps) can be applied to it, which will also produce data that can be referenced through an AnalysisResult.

Thus what essentially happens for an analysis or testing run is generally:

- A script constructs a ModelRun (or suite of runs as a ModelSuite), and customises it, possibly including specifying AnalysisOps.
- The model is ran, producing a ModelResult class.
- This class can then be used as a basis for post-processing, with a defined set of capabilities that allow the user to explore the results of the model, and perform analysis on those results.

Note: In future, we may allow the ability of running a set of AnalysisOps as a post-processing operation on the results of a ModelRun, based on checkpointed data. However, this feature is not currently implemented.

The examples in the *Examples of how to use CREDO* section help illustrate how these capabilities work in practice.

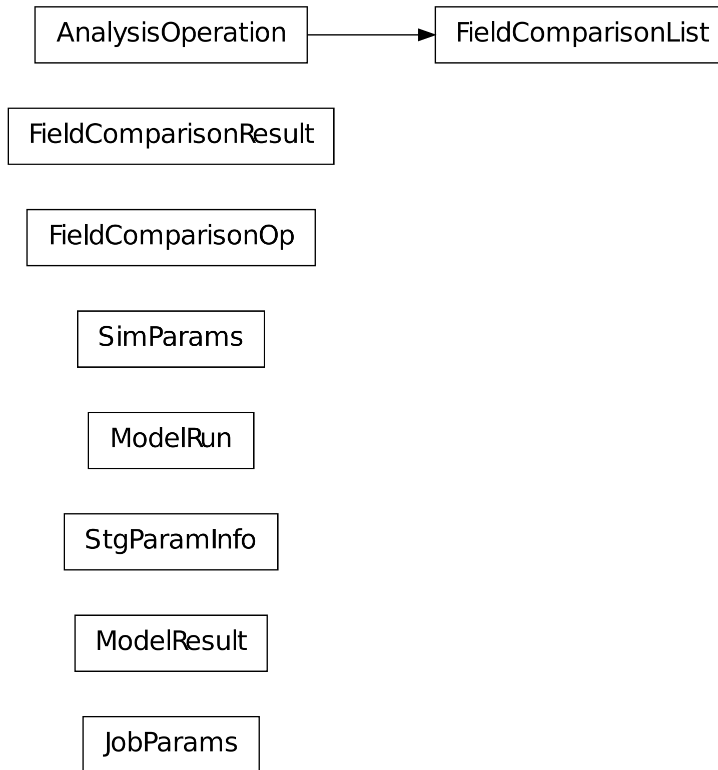
3.1 Organisation of Python Code and class hierarchy

As usual for Python projects, the code is separated into a set of directories of related functionality called *packages*, which are then further divided into individual *modules*.

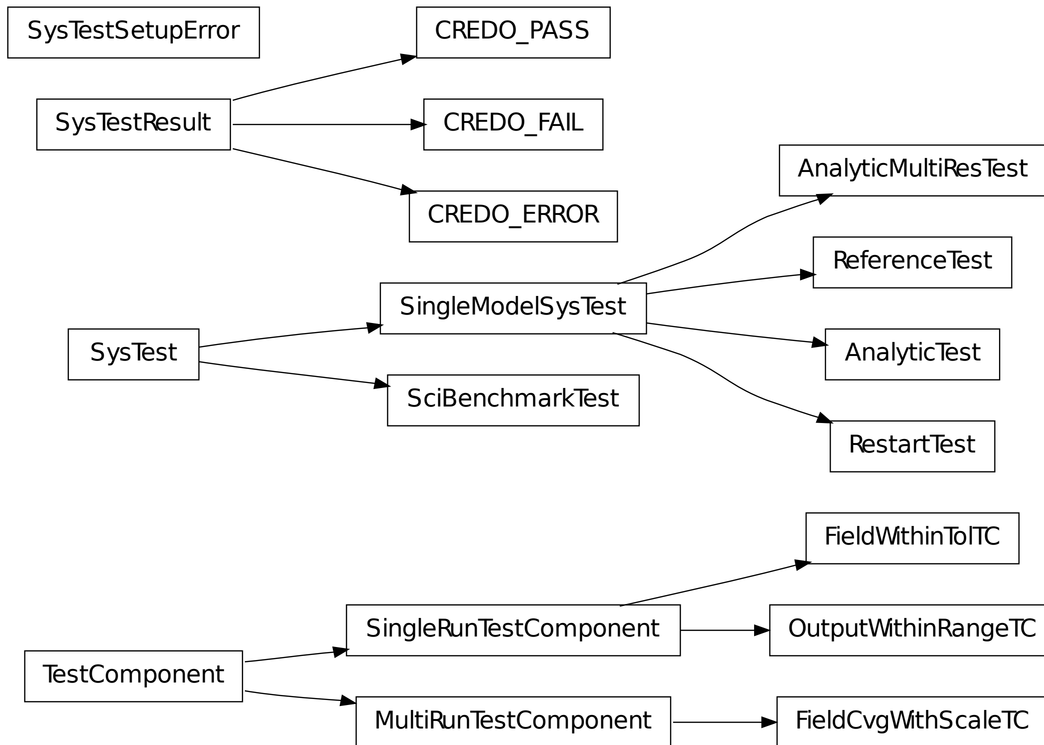
Note: CREDO uses Python's object-oriented capabilities to manage a set of Classes to implement this architecture. Python is designed to be easy to pick up, and from a user's perspective the object-oriented features shouldn't make scripting difficult. Thus if you only intend to write CREDO scripts to perform analysis, you should be able to move straight to the *Examples of how to use CREDO* without worrying about how Python classes work in detail. However if you intend to add and develop new Analysis capabilities, you may wish to run through the Python tutorial's section on Classes as a primer to get you started.

These key models and packages are described and documented in the *CREDO Python Source API Documentation* section. The linked diagrams below show several of the key classes that implement the architecture discussed above:

The core ModelRun, ModelResult and Analysis hierarchy:



And several key System testing classes:


SysTestSetupError
CREDO_PASS
SysTestResult
CREDO_FAIL
AnalyticMultiResTest
CREDO_ERROR
ReferenceTest
SingleModelSysTest
SysTest
AnalyticTest
SciBenchmarkTest
RestartTest
FieldWithinTolTC
SingleRunTestComponent
OutputWithinRangeTC
TestComponent
MultiRunTestComponent
FieldCvgWithScaleTC

CHAPTER

FOUR

EXAMPLES OF HOW TO USE CREDO

The sections below should help given an overview of how to use CREDO, through worked examples.

The *Using CREDO for System Testing of StGermain-based codes such as Underworld* section explains how to use CREDO to run and set-up the basic system tests of a StGermain code that supercede the previous system testing scripts.

The *Doing Model analysis with CREDO* section shows examples of how to configure and run Underworld runs using CREDO, and analyse/post-process the results. This is for custom analysis, rather than for addition to the automated testing system.

The *Scientific Benchmarking using CREDO* section shows the more complex use-case of the code, where scientific benchmarks are set up: generally requiring both analysis-style set-up of models to run, while also using the system testing features to allow automated regular running of this benchmark.

The *Different ways to launch CREDO scripts* section gives examples of how to run CREDO scripts in different ways, including via PBS.

Note: CREDO is designed in such a manner that it should be possible to readily convert analysis scripts into repeatable system tests, and after reading examples of all the sections above you should have a handle on how to go about this.

4.1 Using CREDO for System Testing of StGermain-based codes such as Underworld

4.1.1 Running a test target, or test suite, via the SCons build system

CREDO is now integrated with the Python-based SCons build system, so for projects such as Underworld which use SCons it is now possible to fairly easily register with and run CREDO System tests via SCons.

This section will start with a guide to running CREDO tests via SCons where they are already registered. Then the following section *Registering CREDO Test suites with SCons* explains how these test suites are actually registered with SCons so they can be used in this form.

Running a project-defined multi-suite test target

If you invoke the SCons help for a project using CREDO in it's base directory, e.g. stgUnderworld, by typing:

```
./scons.py --help
```

the full list of check 'targets' for a project will be printed. CREDO has been set up to also work with the PCU (Parallel C-Unit) unit testing suite also used by StGermain codes, so that some testing targets can run both unit and system tests.

For example, the "check-lowres" target has been set up to run low-resolution system test suites, so you can invoke this target by running from the stgUnderworld base directory:

```
./scons.py check-lowres
```

which will invoke a series of system tests [1].

The output printed after running this command should start with something like the following:

```
scons: Reading SConscript files ...
Mkdir("./testLogs")
Copy("build/config.log", "config.log")
Copy("build/config.cfg", "config.cfg")
scons: done reading SConscript files.
scons: Building targets ...
runSuites(["testLogs/lowresSuiteSummary.xml"], ["Underworld/SysTest/RegressionTests/testAll_lowres.py
Importing suite for Underworld.SysTest.RegressionTests.testAll_lowres:
Running the following system test suites:
 Project 'Underworld', suite 'RegressionTests-lowres'
--------------------------------------------------------------------------------
Running System Test Suite for Project 'Underworld', named 'RegressionTests-lowres', containing 30 dir
Running System test 1/30, with name 'Anisotropic-referenceTest-np1-lowres':
Writing pre-test info to XML
Running the 1 modelRuns specified in the suite
Doing run 1/1 (index 0), of name 'Anisotropic-referenceTest-np1-lowres':
ModelRun description: "Run the model, and check results against previously generated reference soluti
Generating analysis XML:
Running the Model (saving results in output/Anisotropic-referenceTest-np1-lowres):
Running model 'Anisotropic-referenceTest-np1-lowres' with command 'mpirun -np 1 /home/psunter/AuScope
Model ran successfully (output saved to path output/Anisotropic-referenceTest-np1-lowres, std out & s
Doing post-run tidyup:
Checking test result:
Field comp 'VelocityField' error within tol 0.01 of reference solution for all runs.
Field comp 'PressureField' error within tol 0.01 of reference solution for all runs.

Test result was Pass
```

..and so on through a series of test suites, concluding with:

```
--------------------------------------------------------------------------------
CREDO System Tests results summary, project 'Underworld', suite 'RegressionTests-lowres':
Ran 30 system tests, with 29 passes, 1 fails, and 0 errors
Failures were:
 CylinderRiseThermal-referenceTest-np2-lowres: A Field was not within tolerance of reference soln.
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
CREDO System Tests summary for all project suites ran:
------
Project 'Underworld':
 Suite 'RegressionTests-lowres': 30 tests, 29/1/0 passes/fails/errors
30 tests, 29/1/0 passes/fails/errors
------
ALL Projects Total:  30 tests, 29/1/0 passes/fails/errors
```

[1] While you no longer need to run a custom scons executable from the local directory ("./scons.py") to get system test summary results, as CREDO is now fully integrated with stgUnderworld's scons config files and is able to print a summary using the standard scons executable ... since we still distribute a patched scons with stgUnderworld, we recommend continuing to run scons via the local script. All of the examples throughout this documentation will thus be in this form.

```
--------------------------------------------------------------------------------
scons: done building targets.
```

This gives you a summary of the results of the system test suite run. For check targets that run across multiple projects, the final summary will show the totals sorted by project, for example:

```
--------------------------------------------------------------------------------
CREDO System Tests summary for all project suites ran:
------
Project 'StgFEM':
 Suite 'PerformanceTests': 3 tests, 3/0/0 passes/fails/errors
3 tests, 3/0/0 passes/fails/errors
------
Project 'PICellerator':
 Suite 'PerformanceTests': 3 tests, 3/0/0 passes/fails/errors
3 tests, 3/0/0 passes/fails/errors
------
Project 'Underworld':
 Suite 'PerformanceTests': 16 tests, 16/0/0 passes/fails/errors
16 tests, 16/0/0 passes/fails/errors
------
ALL Projects Total:  22 tests, 22/0/0 passes/fails/errors
--------------------------------------------------------------------------------
```

The test commands will also save an XML file, for parsing by the likes of Bitten, summarising the results of the test suite run. These are saved in the "testLogs" subdirectory, in sub-directories based on the name of the target you ran. For example, after running the lowres suite above, the testLogs directory will contain a directory *testLogs/check-lowres*, which would contain a suite record file *Underworld-RegressionTests-lowres.xml*. The format of these files is designed to be compatible with parsers that can read results of the Python unittest XML reporting extensions.

Running a single test suite via SCons

Instead of running a whole set of system test suites, you may wish to run just a single suite. Of course this can be done directly as described in *Running CREDO system test suites directly, and how to modify them*, but we also have the facility to do this at the base directory of a project via SCons.

The syntax for running individual suites is based on the Python import name of the suite. This is based on it's position in the file tree, but with "." replacing directory separators.

For example on a Linux system, if you wanted to run the PICellerator convergence test suite as part of the stgUnderworld project, the file is located in the subdirectory *PICellerator/SysTest/PerformanceTests/testAll.py*. In Python import syntax, this becomes *PICellerator.SysTest.PerformanceTests.testAll*.

So invoking the following at the command line:

```
./scons.py PICellerator.SysTest.PerformanceTests.testAll
```

will cause that suite to be run, printing output as follows:

```
scons: Reading SConscript files ...
Mkdir("./testLogs")
Copy("build/config.log", "config.log")
Copy("build/config.cfg", "config.cfg")
scons: done reading SConscript files.
scons: Building targets ...
runSuites(["testLogs/PICellerator.SysTest.PerformanceTests.testAll.xml"], ["PICellerator/SysTest/Per
Importing suite for PICellerator.SysTest.PerformanceTests.testAll:
Running the following system test suites:
 Project 'PICellerator', suite 'PerformanceTests'
```

```
--------------------------------------------------------------------------------
Running System Test Suite for Project 'PICellerator', named 'PerformanceTests', containing 3 direct t
Running System test 1/3, with name 'BuoyancyExample-analyticMultiResConvergenceTest-np1':
Writing pre-test info to XML
Running the 4 modelRuns specified in the suite
Doing run 1/4 (index 0), of name 'BuoyancyExample-analyticMultiResConvergenceTest-np1-10x10':
ModelRun description: "Run the model at res 10x10"
Generating analysis XML:
Running the Model (saving results in output/BuoyancyExample-analyticMultiResTest-np1/10x10):
Running model 'BuoyancyExample-analyticMultiResConvergenceTest-np1-10x10' with command 'mpirun -np 1
Model ran successfully (output saved to path output/BuoyancyExample-analyticMultiResTest-np1/10x10, s
Doing post-run tidyup:
Doing run 2/4 (index 1), of name 'BuoyancyExample-analyticMultiResConvergenceTest-np1-20x20':
ModelRun description: "Run the model at res 20x20"
Generating analysis XML:
Running the Model (saving results in output/BuoyancyExample-analyticMultiResTest-np1/20x20):
Running model 'BuoyancyExample-analyticMultiResConvergenceTest-np1-20x20' with command 'mpirun -np 1
Model ran successfully (output saved to path output/BuoyancyExample-analyticMultiResTest-np1/20x20, s
Doing post-run tidyup:
Doing run 3/4 (index 2), of name 'BuoyancyExample-analyticMultiResConvergenceTest-np1-30x30':
ModelRun description: "Run the model at res 30x30"
Generating analysis XML:
Running the Model (saving results in output/BuoyancyExample-analyticMultiResTest-np1/30x30):
Running model 'BuoyancyExample-analyticMultiResConvergenceTest-np1-30x30' with command 'mpirun -np 1
Model ran successfully (output saved to path output/BuoyancyExample-analyticMultiResTest-np1/30x30, s
Doing post-run tidyup:
Doing run 4/4 (index 3), of name 'BuoyancyExample-analyticMultiResConvergenceTest-np1-40x40':
ModelRun description: "Run the model at res 40x40"
Generating analysis XML:
Running the Model (saving results in output/BuoyancyExample-analyticMultiResTest-np1/40x40):
Running model 'BuoyancyExample-analyticMultiResConvergenceTest-np1-40x40' with command 'mpirun -np 1
Model ran successfully (output saved to path output/BuoyancyExample-analyticMultiResTest-np1/40x40, s
Doing post-run tidyup:
Checking test result:
Testing convergence for field 'PressureField'
Field PressureField, dof 0 - cvg rate      1, corr 1.000000
  -Good
Test result was Pass
Saved test result to output/BuoyancyExample-analyticMultiResTest-np1/SysTest-BuoyancyExample-analytic
Running System test 2/3, with name 'BuoyancyExample-analyticMultiResConvergenceTest-np2':
Writing pre-test info to XML
Running the 4 modelRuns specified in the suite
Doing run 1/4 (index 0), of name 'BuoyancyExample-analyticMultiResConvergenceTest-np2-10x10':
ModelRun description: "Run the model at res 10x10"
Generating analysis XML:
Running the Model (saving results in output/BuoyancyExample-analyticMultiResTest-np2/10x10):
Running model 'BuoyancyExample-analyticMultiResConvergenceTest-np2-10x10' with command 'mpirun -np 2
Model ran successfully (output saved to path output/BuoyancyExample-analyticMultiResTest-np2/10x10, s
Doing post-run tidyup:
Doing run 2/4 (index 1), of name 'BuoyancyExample-analyticMultiResConvergenceTest-np2-20x20':
ModelRun description: "Run the model at res 20x20"
Generating analysis XML:
Running the Model (saving results in output/BuoyancyExample-analyticMultiResTest-np2/20x20):
Running model 'BuoyancyExample-analyticMultiResConvergenceTest-np2-20x20' with command 'mpirun -np 2
Model ran successfully (output saved to path output/BuoyancyExample-analyticMultiResTest-np2/20x20, s
Doing post-run tidyup:
Doing run 3/4 (index 2), of name 'BuoyancyExample-analyticMultiResConvergenceTest-np2-30x30':
ModelRun description: "Run the model at res 30x30"
```

```
Generating analysis XML:
Running the Model (saving results in output/BuoyancyExample-analyticMultiResTest-np2/30x30):
Running model 'BuoyancyExample-analyticMultiResConvergenceTest-np2-30x30' with command 'mpirun -np 2
Model ran successfully (output saved to path output/BuoyancyExample-analyticMultiResTest-np2/30x30, s
Doing post-run tidyup:
Doing run 4/4 (index 3), of name 'BuoyancyExample-analyticMultiResConvergenceTest-np2-40x40':
ModelRun description: "Run the model at res 40x40"
Generating analysis XML:
Running the Model (saving results in output/BuoyancyExample-analyticMultiResTest-np2/40x40):
Running model 'BuoyancyExample-analyticMultiResConvergenceTest-np2-40x40' with command 'mpirun -np 2
Model ran successfully (output saved to path output/BuoyancyExample-analyticMultiResTest-np2/40x40, s
Doing post-run tidyup:
Checking test result:
Testing convergence for field 'PressureField'
Field PressureField, dof 0 - cvg rate      1, corr 1.000000
  -Good
Test result was Pass
Saved test result to output/BuoyancyExample-analyticMultiResTest-np2/SysTest-BuoyancyExample-analytic
Running System test 3/3, with name 'BuoyancyExample-analyticMultiResConvergenceTest-np4':
Writing pre-test info to XML
Running the 4 modelRuns specified in the suite
Doing run 1/4 (index 0), of name 'BuoyancyExample-analyticMultiResConvergenceTest-np4-10x10':
ModelRun description: "Run the model at res 10x10"
Generating analysis XML:
Running the Model (saving results in output/BuoyancyExample-analyticMultiResTest-np4/10x10):
Running model 'BuoyancyExample-analyticMultiResConvergenceTest-np4-10x10' with command 'mpirun -np 4
Model ran successfully (output saved to path output/BuoyancyExample-analyticMultiResTest-np4/10x10, s
Doing post-run tidyup:
Doing run 2/4 (index 1), of name 'BuoyancyExample-analyticMultiResConvergenceTest-np4-20x20':
ModelRun description: "Run the model at res 20x20"
Generating analysis XML:
Running the Model (saving results in output/BuoyancyExample-analyticMultiResTest-np4/20x20):
Running model 'BuoyancyExample-analyticMultiResConvergenceTest-np4-20x20' with command 'mpirun -np 4
Model ran successfully (output saved to path output/BuoyancyExample-analyticMultiResTest-np4/20x20, s
Doing post-run tidyup:
Doing run 3/4 (index 2), of name 'BuoyancyExample-analyticMultiResConvergenceTest-np4-30x30':
ModelRun description: "Run the model at res 30x30"
Generating analysis XML:
Running the Model (saving results in output/BuoyancyExample-analyticMultiResTest-np4/30x30):
Running model 'BuoyancyExample-analyticMultiResConvergenceTest-np4-30x30' with command 'mpirun -np 4
Model ran successfully (output saved to path output/BuoyancyExample-analyticMultiResTest-np4/30x30, s
Doing post-run tidyup:
Doing run 4/4 (index 3), of name 'BuoyancyExample-analyticMultiResConvergenceTest-np4-40x40':
ModelRun description: "Run the model at res 40x40"
Generating analysis XML:
Running the Model (saving results in output/BuoyancyExample-analyticMultiResTest-np4/40x40):
Running model 'BuoyancyExample-analyticMultiResConvergenceTest-np4-40x40' with command 'mpirun -np 4
Model ran successfully (output saved to path output/BuoyancyExample-analyticMultiResTest-np4/40x40, s
Doing post-run tidyup:
Checking test result:
Testing convergence for field 'PressureField'
Field PressureField, dof 0 - cvg rate      1, corr 1.000000
  -Good
Test result was Pass
Saved test result to output/BuoyancyExample-analyticMultiResTest-np4/SysTest-BuoyancyExample-analytic
--------------------------------------------------------------------------------
CREDO System Tests results summary, project 'PICellerator', suite 'PerformanceTests':
Ran 3 system tests, with 3 passes, 0 fails, and 0 errors
```

```
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
CREDO System Tests summary for all project suites ran:
------
Project 'PICellerator':
 Suite 'PerformanceTests': 3 tests, 3/0/0 passes/fails/errors
3 tests, 3/0/0 passes/fails/errors
------
ALL Projects Total:  3 tests, 3/0/0 passes/fails/errors
--------------------------------------------------------------------------------
scons: done building targets.
```

..and an XML log of the suite results will also be created in the *testLogs* directory: in this case in the subdirectory *PICellerator.SysTest.PerformanceTests*, containing a record of the one suite ran *PICellerator-PerformanceTests.xml*.

4.1.2 Registering CREDO Test suites with SCons

For CREDO tests to be able to be registered with and work with SCons, it currently requires several things to be done.

To set up your initial links between the SCons configuration for your project and CREDO:

- Use of the CREDO SCons-related functions distributed with CREDO in the *scons* folder;
- Addition of several targets to your project's main SCons config file, e.g. *SConstruct*.

Then to register particular CREDO test suites with SCons just requires:

- Calling the CREDO functions on your environment in your project configuration files, e.g. *Underworld/SConscript*.

These sections will be explained in turn - if you are working on an existing project that already has CREDO system testing integrated into the project's SCons build system, you can safely jump ahead to *Registering CREDO system tests suites for your project with SCons*.

Note: for the instructions in this section to work, it requires the CREDO system test suite files to follow the conventions in *Requirements for importing test suites: Dual-mode, and the suite() function*, as this convention allows suites to be easily imported from other files.

Setting up the links between SCons and CREDO for a project

This setup just needs to be done once for a project, but requires several steps.

If you wish to understand the rationale in this section, a quick read through of SCons primer material may be helpful. Though written in Python, SCons extends the language with several "magic" features like variable lists that can be imported and exported between scripts, and a few of these are used in integrating CREDO with SCons. Some useful links for a primer may be:

- An intro slideset to SCons ;
- The SCons user guide, especially the sections on:
  - 'Builder' objects, that use Python functions;
  - Where to put custom builders and tools.
- The online SCons manual - especially the Action objects section.
- The SCons wiki page http://www.scons.org/wiki/UnitTests, especially the section "unit tests integration with an SCons tool".

CREDO integration into SCons is based on the above strategies - essentially we provide a CREDO SCons "toolkit" that defines several 'Builders' and 'Actions' relevant to running CREDO tests, based on test lists.

The process of using this in a project is as follows:

1. Add the CREDO toolkit in your project's Environment in your SConstruct.

   This would involve a line such as the following as soon as you create your main SCons environment object:

   ```
   # Load CREDO, the system testing tool
   env.Tool('credosystest', toolpath=['credo/scons'])
   ```

2. Add SCons targets to run system tests linked to the relevant CREDO SCons builders and variables.

   This part needs to occur at the end of your main SConstruct file, after all project-related configuration has been read and processed. The first part is to 'magic import' the special SCons variables CREDO uses to keep track of test lists, e.g.:

   ```
   Import('LOWRES_SUITES')
   Import('INTEGRATION_SUITES')
   Import('CONVERGENCE_SUITES')
   Import('SCIBENCH_SUITES')
   ```

   This is immediately followed by the targets needed to execute the suites, so that for example running *scons check-lowres* will actually run all the low resolution tests you've registered. Each of these targets is set in "always run" mode, so SCons knows to re-run the tests whenever you invoke the commands.

   This requires SCons code such as the following: repeated for all the test suite variable lists defined above:

   ```
   lowresSuiteRun = env.RunSuites(
     Dir(os.path.join(env['TEST_OUTPUT_PATH'], env["CHECK_LOWRES_TARGET"])),
     LOWRES_SUITES)
   env.AlwaysBuild(lowresSuiteRun)
   env.Alias(env["CHECK_LOWRES_TARGET"], lowresSuiteRun)
   ```

   ... finally you need to set up any SCons aliases so that one test command can run multiple other targets, such as the following:

   ```
   # Run the lowres checks as part of default and complete
   env.Alias(env['CHECK_DEFAULT_TARGET'], env['CHECK_LOWRES_TARGET'])
   env.Alias(env['CHECK_COMPLETE_TARGET'], env['CHECK_LOWRES_TARGET'])
   # For the others, just add to the complete target
   env.Alias(env['CHECK_COMPLETE_TARGET'], env['CHECK_INTEGRATION_TARGET'])
   env.Alias(env['CHECK_COMPLETE_TARGET'], env['CHECK_CONVERGENCE_TARGET'])
   env.Alias(env['CHECK_COMPLETE_TARGET'], env['CHECK_SCIBENCH_TARGET'])`
   ```

... and that's it (phew)! You then need to actually define which CREDO test suite files are registered with each target on a per-project basis, explained in *Registering CREDO system tests suites for your project with SCons*.

Registering CREDO system tests suites for your project with SCons

Registering test suites to belong to SCons-visible testing targets is part of functionality provided by the CREDO SCons toolkit. So provided you (or whoever has setup your project) has followed the instructions in *Setting up the links between SCons and CREDO for a project*, you just need to:

1. Right after you set up the cloned environment for a sub-project, define a CURR_PROJECT env variable recording the name of the project.

   CREDO's SCons toolkit can then use this to record and report on which project a test suite is registered to.

   For example, for the Underworld the following lines are near the top of the project's SConscript file:

```
Import('env')

env = env.Clone()
env['CURR_PROJECT'] = "Underworld"
```

2. use functions such as the following to add a test suite to an SCons target:
   - env.AddLowResTestSuite
   - env.AddIntegrationTestSuite
   - env.AddConvergenceTestSuite
   - env.AddSciBenchTestSuite

   ... where the only input to each function is the relative path to the CREDO Suite to register. For example, in the Underworld project this section looks like the following towards the bottom of the project's SConscript file:

```
env.AddLowResTestSuite('SysTest/RegressionTests/testAll_lowres.py')
env.AddIntegrationTestSuite('SysTest/RegressionTests/testAll.py')
env.AddConvergenceTestSuite('SysTest/PerformanceTests/testAll.py')
env.AddSciBenchTestSuite('SysTest/ScienceBenchmarks/testAll.py')
```

That's it! This will then allow you to run SCons command-line testing targets and get reporting on a per-project basis as shown in *Running a test target, or test suite, via the SCons build system*.

4.1.3 Running CREDO system test suites directly, and how to modify them

CREDO provides the ability to run suites of system tests directly via the command line, similar to the interface of the previous SYS scripts system.

To start off with, make sure the necessary environment variables necessary to run CREDO have been set up, as detailed in *Setting up your environment to use CREDO*.

Then check out one of the test suite files in a SysTest directory within the code. We'll show in this example StgFEM/SysTest/RegressionTests/testAll-new.py:

```
#!/usr/bin/env python

from credo.systest import *

testSuite = SysTestSuite("StgFEM", "RegressionTests")

analyticModels = ["CosineHillRotateBC.xml", "CosineHillRotateBC-DualMesh.xml",
    "HomogeneousNaturalBCs.xml", "HomogeneousNaturalBCs-DualMesh.xml",
    "SteadyState1D-x.xml", "SteadyState1D-y.xml",
    "AnalyticSimpleShear.xml"]

for modelXML in analyticModels:
    for nproc in [1, 2, 4]:
        testSuite.addStdTest(AnalyticTest, modelXML, nproc=nproc)

ss0_5Opts = {"defaultDiffusivity":0.5, "A":0.1}
for nproc in [1, 2, 4]:
    testSuite.addStdTest(AnalyticTest, ["SteadyState1D-x.xml"],
        nproc=nproc, paramOverrides=ss0_5Opts)

asOpts = {"sinusoidalLidWavenumber":1}
for nproc in [1, 2, 4]:
    testSuite.addStdTest(AnalyticTest, ["AnalyticSinusoid.xml"],
```

```
        nproc=nproc, paramOverrides=asOpts)

for nproc in [1, 2, 4, 8]:
    testSuite.addStdTest(AnalyticTest, ["TempDiffusion.xml"],
        nproc=nproc)

testSuite.addStdTest(HighResReferenceTest, ["TempDiffusionHR.xml"],
    fieldsToTest=["TemperatureField"], highResRatio=4)

for nproc in [1,2]:
    testSuite.addStdTest(RestartTest, ["Multigrid.xml"],
        solverOpts='options-uzawa-mg.opt', nproc=nproc)
    testSuite.addStdTest(ReferenceTest, ["Multigrid.xml"],
        solverOpts='options-uzawa-mg.opt', nproc=nproc)
    testSuite.addStdTest(HighResReferenceTest, ["Multigrid.xml"],
        highResRatio=2, fieldTols={"VelocityField":0.2, "PressureField":0.5},
        solverOpts='options-uzawa-mg.opt', nproc=nproc)

testSuite.setAllTimeouts(minutes=10)

def suite():
    return testSuite

if __name__ == "__main__":
    testRunner = SysTestRunner()
    testRunner.runSuite(testSuite)
```

This Python script uses CREDO to run several system tests, and process their results. To see it in practice, cd to that directory and then run the script (since it's set as executable, you don't need to invoke Python explicitly). You should see output starting with:

```
Running System Test Suite for Project 'StgFEM', named 'RegressionTests', containing 33 direct tests a
Running System test 1/33, with name 'CosineHillRotateBC-analyticTest-np1':
Writing pre-test info to XML
Running the 1 modelRuns specified in the suite
Doing run 1/1 (index 0), of name 'CosineHillRotateBC-analyticTest-np1':
ModelRun description: "Run the model and generate analytic soln."
Generating analysis XML:
```

and finishing with:

```
--------------------------------------------------------------------------------
CREDO System Tests results summary, project 'StgFEM', suite 'RegressionTests':
Ran 33 system tests, with 33 passes, 0 fails, and 0 errors
--------------------------------------------------------------------------------
```

The following sections will explain how the file is set up, and show what the different sections do.

Importing CREDO and creating a testRunner object

To explain the first few lines of the script, as shown below:

```
#!/usr/bin/env python

from credo.systest import *

testSuite = SysTestSuite("StgFEM", "RegressionTests")
```

The first denotes the file as an executable script, using Python.

The next imports everything directly under the credo.systest package for use in the rest of the Python script - this is a convenience since all the objects we'll need for the rest of the script, such as various types of System test classes, are contained here.

The next line creates a SysTestSuite object to use for managing a test suite, and assigns it to the name 'testSuite'. The 2 arguments when creating the SysTestSuite are for recording the project the suite belongs to, and a textual name of the suite.

Note that the SysTestSuite object definition was one of those we imported with the preceding *import* statement.

See Also:

The `credo.systest.api.SysTestSuite` class in the API documentation.

Adding system tests to the suite, and configuring them

Let's look at the next few lines, which declare a set of test models to run, and add them to the test suite:

```
analyticModels = ["CosineHillRotateBC.xml", "CosineHillRotateBC-DualMesh.xml",
    "HomogeneousNaturalBCs.xml", "HomogeneousNaturalBCs-DualMesh.xml",
    "SteadyState1D-x.xml", "SteadyState1D-y.xml",
    "AnalyticSimpleShear.xml"]

for modelXML in analyticModels:
    for nproc in [1, 2, 4]:
        testSuite.addStdTest(AnalyticTest, modelXML, nproc=nproc)
```

The analyticModels has been created as a Python List. This list can then be used inside the *for* loop below it to add each particular model as an AnalyticTest using the *addStdTest* method of the testSuite object we created above.

The addStdTest method uses a special shorthand to add tests to the suite, to save some of the work from the users in this high-level interface. Its arguments are:

- Test class: the class of test you're adding. Generally this should be one of the standard set imported above, i.e. RestartTest, ReferenceTest, AnalyticTest, or AnalyticMultiResTest.
- XML file name(s): list of XML files that make up the model to be used in the test. If in the common case this is just one top-level XML file, a single string instead of a list is ok.
- Any over-riding options you wish to pass to the test.

In the case of over-riding parameters to the tests, there are 2 main categories:

1. Passing parameters that customise the model itself (paramOverrides)
2. Passing parameters that modify the nature of the test.

We can see an example of the first of these in the next section of the test script:

```
ss0_5Opts = {"defaultDiffusivity":0.5, "A":0.1}
for nproc in [1, 2, 4]:
    testSuite.addStdTest(AnalyticTest, ["SteadyState1D-x.xml"],
        nproc=nproc, paramOverrides=ss0_5Opts)
```

... this section means that an AnalyticTest on the *SteadyState1D-x.xml* model is being added to the suite, with the model being modified by over-riding the "defaultDiffusivity" and "A" parameters from the values specified in the Model XML file, to 0.5 and 0.1 respectively. The *paramOverrides* option is a Python Dictionary.

Here, the paramOverrides option is a dictionary of overrides to perform exactly the same as described in the section for running a single analysis model through CREDO below.

With regard to the second option type, there are several options you can use to over-ride the default behaviour of system tests. Please refer to the API section on System Tests for more. The principle is the same for all of these regarding SysTestRunner suites though: just specify the options as you would to the constructor of an individual SystemTest, and they will be passed through by the SysTestRunner.

See Also:

The `AnalyticTest`, `RestartTest`, `ReferenceTest`, and `AnalyticMultiResTest` classes.

Creating a SysTestRunner to run all tests

To actually run tests, in the test script we need to call a SysTestRunner to run a group of system tests, usually packaged together in one or more Suites. The basic code to do this is:

```
testRunner = SysTestRunner()
testRunner.runSuite(testSuite)
```

This will actually trigger the running of the tests, and produce a report of what happened both to the terminal, and in XML files.

See Also:

The `SysTestRunner` class documentation.

Requirements for importing test suites: Dual-mode, and the suite() function

You'll notice in the example test script from StgFEM, we don't simply create a testRunner at the end and run the tests. As a reminder, the actual code is:

```
def suite():
    return testSuite

if __name__ == "__main__":
    testRunner = SysTestRunner()
    testRunner.runSuite(testSuite)
```

The reason for this approach is so that the test can be run directly from the command line to run the suite and report the results, *or* imported from another Python file when running and analysing a whole set of suites. This is known as a "dual-mode" script in Python, and is needed so that CREDO test scripts can be run via SCons, as discussed in *Running a test target, or test suite, via the SCons build system*.

Generally those working on these scripts can just follow this pattern. The 2 key aspects to keep in mind are:

1. The *suite()* function must be defined if you expect to be able to use this script in SCons from other directories.
2. The *__name__ == "__main__"* section will only be executed if the script is running directly from the terminal - meaning that the testRunner will only be created and run in this case. This means that when importing this script, you can control how and when to run the suite from the importing program.

See Also:

http://docs.python.org/tutorial/modules.html#executing-modules-as-scripts

Example: modifying an existing test suite to test with Multigrid options

Suppose you want to run a pre-defined test suite, but with some non-standard options applied to test a particular feature or algorithm, such as Multigrid.

While it'd be fine to make a copy of the original script and just modify each system test case as it's added to the suite, to reduce the amount of typing and also possibly ease long-term challenge of keeping the scripts in Synch, with CREDO you can modify the tests in a system test-suite *after* they are defined but *before* they are run.

For example, suppose for each test in a suite we want to do several things to test how they perform with Multigrid:

- Add an extra XML file to be applied
- Define a PETSc solver options file
- Get the test directory record and log file to append "-mg" to the end.

This can be done by modifying the `SysTest` objects that are kept in the `sysTests` attribute of test suites, in a *for* loop:

```
# Customise to run each test with Multigrid options
mgSetup = "MultigridEXPERI.xml"
mgOpts = "options-uzawa-mg.opt"
for sysTest in testSuite.sysTests:
    sysTest.testName += "-mg"
    sysTest.outputPathBase += "-mg"
    sysTest.inputFiles.append(mgSetup)
    sysTest.solverOpts = mgOpts
```

Perhaps an even better way to achieve this would be to run our modified script as a totally different file that *imports* the existing test suite.

For example if the existing test suite is in a file *testAll.py*, and we want to run the Multigrid-extended version as *testAll-mg.py*, then the contents of the latter file would be:

```
#!/usr/bin/env python
import os
import copy
from credo.systest import *
from testAll import testSuite

mgSuite = copy.deepcopy(testSuite)
# Customise to run each test with Multigrid options
mgSuite.suiteName += "-mg"
mgSetup = "MultigridEXPERI.xml"
mgOpts = "options-uzawa-mg.opt"
for sysTest in mgSuite.sysTests:
    sysTest.testName += "-mg"
    sysTest.updateOutputPaths(sysTest.outputPathBase + "-mg")
    sysTest.inputFiles.append(mgSetup)
    sysTest.solverOpts = mgOpts

if __name__ == "__main__":
    testRunner = SysTestRunner()
    testRunner.runSuite(mgSuite)
```

The only significant change in this imported script from the lines of Python earlier is we've used the "deep copy" functionality of Python to create a new TestSuite that is a copy of the one we imported. In this case it's not essential as our changes here wouldn't permanently affect the old suite, but it's good practice to get in to for this sort of activity.

Alternative: Running a single test from the command-line

Alternatively to the above, scripts have been provided to allow you to run a single system test from the command line prompt, just as you could with the SYS scripts.

To do this, the relevant scripts are as follows:

Script Name	Sys Test class it invokes
credo-referenceTest.py	ReferenceTest
credo-restartTest.py	RestartTest
credo-analyticTest.py	AnalyticTest
credo-analyticTestMultiResCvg.py	AnalyticMultiResTest

So for example, to run a RestartTest on the Multigrid.xml model, type:

```
credo-restartTest.py Multigrid.xml
```

... which will run the test.

4.1.4 Writing new CREDO Testing (and analysis) components

Note: This is currently a very bare-bones description, needs to be expanded upon. In future refactors we also hope to simplify this process.

The recommended steps are essentially as follows:

1. Test that your new code works, e.g. just write a basic Python script (Use Docstrings, and ideally list arguments of your functions).
2. Write a `credo.analysis` component to perform your operation

   Note: Remember to add the CREDO header/licence text (just copy it from one of the other files).

3. Write a new `credo.systest.api.TestComponent`, in the credo/systest directory.
   - specify members needed by the class in the "init" function;
   - Fill out the "check" function;
   - Save specification of the test in the _writeXMLCustomSpec() function;
   - Save results of the test in the _writeXMLCustomResult() function.
4. Write a Python unittest that your new test component works

   This should live in the 'tests' sub-directory.
5. (Optional) Write a new SysTest component that will use your new TestComponent.

   E.g. in the case of the Image testing, the new component that simply creates a SysTest, attaches an image testing TestComponent, and passes relevant images through.
6. Add the new SysTest and TestComponent to the *credo.systest.__init__.py* systest module file's import lists, so they can be easily imported by user system testing scripts.
7. add your new modules to the Sphinx doc-generator, to be auto generated. E.g. for images:

   To file credo/doc/api/analysis.rst, added section:

   ```
   :mod:`credo.analysis.images`
   ============================

   .. automodule:: credo.analysis.images
      :members:
   ```

```
:undoc-members:
:show-inheritance:
```

TODO: also add the new component to the documentation list to generate main inheritance diagrams for SysTests and TestComponents.

4.2 Doing Model analysis with CREDO

4.2.1 Using CREDO to run and analyse a Rayleigh-Taylor problem in Underworld

This examples shows how to use CREDO to run a simple single Underworld job, and perform some basic post-processing on the results, such as getting relevant values from the FrequentOutput.dat, and plotting observables of interest.

Setup

The script to run a Rayleigh Taylor model is as included below, currently in the Underworld/InputFiles directory:

```
1  #! /usr/bin/env python
2  from credo.modelrun import ModelRun, SimParams
3  import credo.jobrunner
4
5  mRun = ModelRun("RayTay-basicBenchmark", "RayleighTaylorBenchmark.xml",
6      "output/raytay-scibench-credo-basic",
7      simParams=SimParams(stoptime=20.0),
8      paramOverrides={"gravity":1})
9      #solverOpts="myOpts.opt")
10
11 def postProc(mRun, mResult):
12     mResult.readFrequentOutput()
13     mResult.freqOutput.plotOverTime('Vrms', depName='Time', show=True,
14         path=mResult.outputPath)
15
16     maxVal, maxTimeStep = mResult.freqOutput.getMax('Vrms')
17     maxTime = mResult.freqOutput.getValueAtStep('Time', maxTimeStep)
18     print "Maximum value of Vrms was %f, at time %d" % (maxVal, maxTime)
19
20 if __name__ == "__main__":
21     jobRunner = credo.jobrunner.defaultRunner()
22     mResult = jobRunner.runModel(mRun)
23     postProc(mRun, mResult)
```

The script above the #———— comment line is setting up and running the model, and that below it is for doing some simple post-processing and analysis of the result.

Essentially, what we are doing is setting up a ModelRun to run the "RayleighTaylorBenchmark.xml" model, with some small customisations to some of the parameters. We also specify to save information about the model run via the `writeInfoXML()` method and `writeModelResultsXML()` function.

See Also:

Modules `credo.modelrun` and `credo.modelresult`

Looking at the post-processing in more detail:

```
if __name__ == "__main__":
    jobRunner = credo.jobrunner.defaultRunner()
    mResult = jobRunner.runModel(mRun)
    postProc(mRun, mResult)
```

We first use the `readFrequentOutput()` method to read the FrequentOutput.dat results into memory and make them accessible through CREDO, bound to a *freqOutput* attribute of the mRes object. We are then able to use various methods of this `credo.io.stgfreq.FreqOutput` class to query the Frequent Output for properties of interest - in this case the maximum value of the "Vrms" property, the time this occurred. We also use the plotOverTime() method to plot and save a graph of the value of Vrms over time in the model.

See Also:

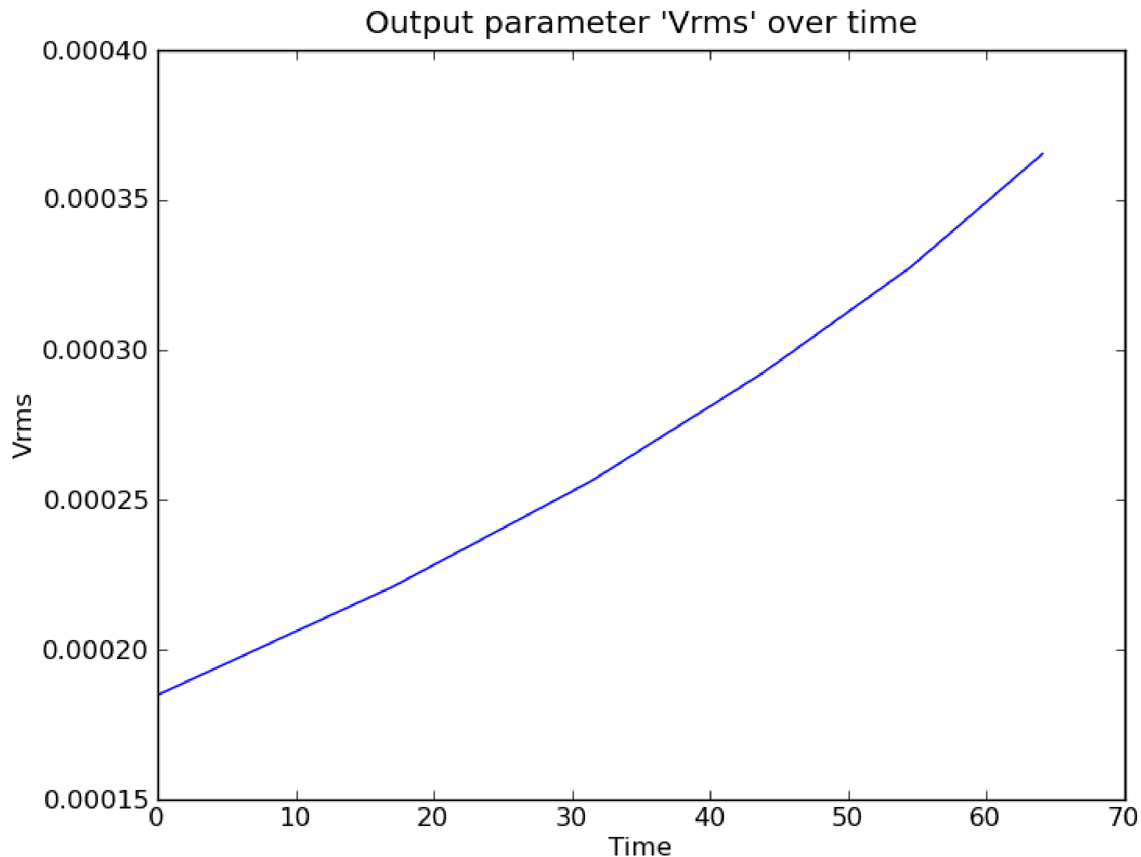
The `credo.io.stgfreq.FreqOutput` class, especially the `plotOverTime()` method.

Outputs

Running this script at the terminal produces:

```
Running model 'RayTay-basicBenchmark' with command 'mpirun -np 1 /home/psunter/AuScopeCodes/stgUnderw
Model ran successfully.
Maximum value of Vrms was 0.000365, at time 64
```

Where the last line is the result of our post-processing query. If you have Matplotlib installed, it will also produce a pop-up window showing the graph of VRMS against time, something like that shown below:



In the script above you can see the output path requested for the model was *./output/raytay-scibench-credo-basic*. If you have a look at the contents of the directory, as well all of the standard output that an Underworld run saves [2], you'll see several things specific to CREDO:

- An *credo-analysis.xml* file, recording a summary StGermain format XML of over-rides or new components created to complete the required analysis specified in the CREDO script;
- *ModelRun-RayTay-basicBenchmark.xml* and *ModelResult-RayTay-basicBenchmark.xml* files, which keep a record of what CREDO was asked to run and the result it produced
- *Vrms-timeSeries.png*, a saved copy of the image shown above [3].

The contents of the ModelRun and ModelResult XML files should look something like the below:

```
<StgModelRun>
  <name>RayTay-basicBenchmark</name>
  <modelInputFiles>
    <inputFile>RayleighTaylorBenchmark.xml</inputFile>
  </modelInputFiles>
  <outputPath>output/raytay-scibench-credo-basic</outputPath>
  <jobParams>
    <nproc>1</nproc>
  </jobParams>
  <simParams>
    <restartstep>None</restartstep>
    <stoptime>60.0</stoptime>
    <nsteps>None</nsteps>
    <dumpevery>1</dumpevery>
    <cpevery>1</cpevery>
  </simParams>
  <paramOverrides>
    <param modelPath="gravity" paramVal="1" />
  </paramOverrides>
  <analysis />
</StgModelRun>
```

... and:

```
<StgModelResult>
  <modelName>RayTay-basicBenchmark</modelName>
  <outputPath>output/raytay-scibench-credo-basic</outputPath>
  <jobMetaInfo>
    <simtime>64.0557</simtime>
  </jobMetaInfo>
</StgModelResult>
```

You will see that they save essential quantities about the run requested and the result [4].

4.2.2 Using CREDO to run and analyse a Suite of Rayleigh-Taylor problems

This examples shows how to use CREDO to run a suite of Underworld runs, based on the same Model XML, but varying different parameters.

[2] such as FrequentOutput.dat, and a record of the flattened XML produced by the run as *input.xml*. For more on these, see the Underworld manual.

[3] Note it's possible not to save these images, by passing *save=False* as a keyword argument to the plotOverTime method (see `plotOverTime()`).

[4] Note: in future, we plan to provide the capability to read in a ModelResult.xml file into CREDO, which will create a ModelResult object for post-processing. However as yet this capability isn't included.

Note: This example script is based on the same Rayleigh Taylor model as described in the *Using CREDO to run and analyse a Rayleigh-Taylor problem in Underworld* section, so please read that section first as it goes into more detail about set-up and results.

Note: This capability of the CREDO toolkit is under active development, so check back here regularly for new features and updates.

Setup

The script to run a suite of Rayleigh Taylor model is as included below, currently in the Underworld/InputFiles directory:

```
#! /usr/bin/env python
import os
from credo.modelrun import ModelRun, SimParams
from credo.modelsuite import ModelSuite, StgXMLVariant
import credo.modelsuite as msuite
import credo.jobrunner
from credo.analysis import modelplots
import credo.reporting.standardReports as sReps
from credo.reporting import getGenerators

elRes=16
stoptime=600.0
mRun = ModelRun("RayTay-basic", "RayleighTaylorBenchmark.xml",
    simParams=SimParams(stoptime=stoptime, nsteps=-1, dumpevery=3))
mRun.paramOverrides={"elementResI":elRes, "elementResJ":elRes}

mSuite = ModelSuite("output/raytay-suite-%dx%d-%d_t" % (elRes, elRes, stoptime),
    templateMRun=mRun)
gravRange = [0.7 + x * 0.1 for x in range(4)]
mSuite.addVariant("gravity", StgXMLVariant("gravity", gravRange))
ampRange = [0.02, 0.04, 0.07]
mSuite.addVariant("initPerturbation",
    StgXMLVariant("components.lightLayerShape.amplitude", ampRange))

mSuite.generateRuns()

def reportResults(mSuite, mResults):
    indicesIter = msuite.getVariantIndicesIter(mSuite.modelVariants,
        mSuite.iterGen)
    varNameDicts = msuite.getVariantNameDicts(mSuite.modelVariants, indicesIter)
    for resI, mRes in enumerate(mResults):
        print "Post-process result %d: with" % resI,
        print ", ".join(["%s=%g" % item for item in varNameDicts[resI].iteritems()])
        mRes.readFrequentOutput()
        mRes.freqOutput.plotOverTime('Vrms', depName='Time',
            path=mRes.outputPath)
        maxVal, maxTimeStep = mRes.freqOutput.getMax('Vrms')
        maxTime = mRes.freqOutput.getValueAtStep('Time', maxTimeStep)
        print "\tMaximum value of Vrms was %f, at time %.2f" % (maxVal, maxTime)
    modelplots.plotOverAllRuns(mResults, 'Vrms', depName='Time',
        path=mSuite.outputPathBase)
    mSuite.analysisImages = [
        'Vrms-multiRunTimeSeries.png']
```

```
44      mSuite.modelImagesToDisplay = [[] for runI in range(len(mSuite.runs))]
45      for runI, mRun in enumerate(mSuite.runs):
46          mRes = mResults[runI]
47          fStep = mRes.freqOutput.finalStep()
48          fTime = mRes.freqOutput.getValueAtStep('Time', fStep)
49          dEvery = mRun.simParams.dumpevery
50          lastImgStep = fStep / dEvery * dEvery
51          vrmsMax, vrmsPeakStep = mRes.freqOutput.getMax('Vrms')
52          vrmsMaxTime = mRes.freqOutput.getValueAtStep('Time', vrmsPeakStep)
53          vrmsPeakImgStep = mRun.simParams.nearestDumpStep(fStep, vrmsPeakStep)
54          vrmsPeakImgTime = mRes.freqOutput.getValueAtStep('Time',
55              vrmsPeakImgStep)
56          mSuite.modelImagesToDisplay[runI] = [
57              (0, "initial"),
58              (vrmsPeakImgStep, "t=%f, near VRMS peak %g (at t=%g)" %\
59                  (vrmsPeakImgTime, vrmsMax, vrmsMaxTime)),
60              (mRun.simParams.nearestDumpStep(fStep, fStep*.5), "1/2"),
61              (mRun.simParams.nearestDumpStep(fStep, fStep*.75), "3/4"),
62              (lastImgStep, "final state (t=%f)" % fTime)]
63
64      for rGen in getGenerators(["RST", "ReportLab"], mSuite.outputPathBase):
65          sReps.makeSuiteReport(mSuite, mResults, rGen,
66              os.path.join(mSuite.outputPathBase, "%s-report.%s" %\
67                  ("RayTaySuite-examples", rGen.stdExt)), imgPerRow=3)
68
69  if __name__ == "__main__":
70      postProcFromExisting = True
71      if postProcFromExisting == False:
72          jobRunner = credo.jobrunner.defaultRunner()
73          mResults = jobRunner.runSuite(mSuite)
74      else:
75          mResults = mSuite.readResultsFromPath(mSuite.runs[0].basePath)
76      reportResults(mSuite, mResults)
```

As with the script described in the *Basic Ray-Tay example*, above the #———— comment line is setting up and running the model, and that below it is for doing some simple post-processing and analysis of the result.

In this case, as well as setting up a ModelRun object as we did in the basic Ray-Tay example, we also want to create a `ModelSuite` where the RayTay ModelRun is used as a template (line 11). We then set up the suite to vary the model parameters Gravity, and the amplitude of the Light layer shape (lines 12-17). Finally, on line 18 we request CREDO to run the entire suite. Unlike in the *Basic Ray-Tay example*, we don't have to specifically request XMLs records to be saved, the ModelSuite will do this for us automatically.

See Also:

Modules `credo.modelsuite`, `credo.modelrun`, and `credo.modelresult`.

Looking at the post-processing in more detail:

```
mSuite.addVariant("gravity", StgXMLVariant("gravity", gravRange))
ampRange = [0.02, 0.04, 0.07]
mSuite.addVariant("initPerturbation",
    StgXMLVariant("components.lightLayerShape.amplitude", ampRange))

mSuite.generateRuns()

def reportResults(mSuite, mResults):
    indicesIter = msuite.getVariantIndicesIter(mSuite.modelVariants,
        mSuite.iterGen)
```

```
    varNameDicts = msuite.getVariantNameDicts(mSuite.modelVariants, indicesIter)
    for resI, mRes in enumerate(mResults):
        print "Post-process result %d: with" % resI,
        print ", ".join(["%s=%g" % item for item in varNameDicts[resI].iteritems()])
        mRes.readFrequentOutput()
        mRes.freqOutput.plotOverTime('Vrms', depName='Time',
            path=mRes.outputPath)
        maxVal, maxTimeStep = mRes.freqOutput.getMax('Vrms')
        maxTime = mRes.freqOutput.getValueAtStep('Time', maxTimeStep)
        print "\tMaximum value of Vrms was %f, at time %.2f" % (maxVal, maxTime)
    modelplots.plotOverAllRuns(mResults, 'Vrms', depName='Time',
        path=mSuite.outputPathBase)
    mSuite.analysisImages = [
        'Vrms-multiRunTimeSeries.png']
    mSuite.modelImagesToDisplay = [[] for runI in range(len(mSuite.runs))]
    for runI, mRun in enumerate(mSuite.runs):
        mRes = mResults[runI]
        fStep = mRes.freqOutput.finalStep()
        fTime = mRes.freqOutput.getValueAtStep('Time', fStep)
        dEvery = mRun.simParams.dumpevery
        lastImgStep = fStep / dEvery * dEvery
        vrmsMax, vrmsPeakStep = mRes.freqOutput.getMax('Vrms')
        vrmsMaxTime = mRes.freqOutput.getValueAtStep('Time', vrmsPeakStep)
        vrmsPeakImgStep = mRun.simParams.nearestDumpStep(fStep, vrmsPeakStep)
        vrmsPeakImgTime = mRes.freqOutput.getValueAtStep('Time',
            vrmsPeakImgStep)
        mSuite.modelImagesToDisplay[runI] = [
            (0, "initial"),
            (vrmsPeakImgStep, "t=%f, near VRMS peak %g (at t=%g)" %\
                (vrmsPeakImgTime, vrmsMax, vrmsMaxTime)),
            (mRun.simParams.nearestDumpStep(fStep, fStep*.5), "1/2"),
            (mRun.simParams.nearestDumpStep(fStep, fStep*.75), "3/4"),
            (lastImgStep, "final state (t=%f)" % fTime)]

    for rGen in getGenerators(["RST", "ReportLab"], mSuite.outputPathBase):
        sReps.makeSuiteReport(mSuite, mResults, rGen,
            os.path.join(mSuite.outputPathBase, "%s-report.%s" %\
                ("RayTaySuite-examples", rGen.stdExt)), imgPerRow=3)

if __name__ == "__main__":
    postProcFromExisting = True
    if postProcFromExisting == False:
        jobRunner = credo.jobrunner.defaultRunner()
        mResults = jobRunner.runSuite(mSuite)
    else:
        mResults = mSuite.readResultsFromPath(mSuite.runs[0].basePath)
    reportResults(mSuite, mResults)
```

We are using the Python for control flow statement to loop over each ModelResult in the group of them produced by the ModelSuite [5]. We are also using the saved set of ModelRun objects in the ModelSuite (in the *mSuite.runs* list) to look up the relevant values for gravity and the shape's amplitude that were generated from the ranges we specified earlier.

Apart from this, the actual analysis done for each run is the same as in the *Basic Ray-Tay example*.

See Also:

The `credo.io.stgfreq.FreqOutput` class, especially the `plotOverTime()` method.

[5] The 'enumerate' is a useful little Python function to also produce an index of each element in a for loop - see its Python documentation.

Outputs

Since we asked for a suite of ModelRuns, we expect CREDO to run each of the individual runs in turn. The "base" output path requested was *output/raytay-suite*: this means that CREDO will save the results in here, with individual ModelRuns saved in sub-directories with names based on the parameters involved.

Let's start with the output produced by running the script at the terminal, which should produce results starting with:

```
> python credo-rayTaySuite.py
Running the 12 modelRuns specified in the suite
Doing run 1/12 (index 0), of name 'RayTay-basic':
ModelRun description: "lightLayerAmplitude_0.02-gravity_0.7"
Generating analysis XML:
Running the Model (saving results in output/raytay-suite/lightLayerAmplitude_0.02-gravity_0.7):
Running model 'RayTay-basic' with command 'mpirun -np 1 /home/psunter/AuScopeCodes/stgUnderworldE-cre
Model ran successfully.
Doing post-run tidyup:
```

and ending with:

```
Doing run 12/12 (index 11), of name 'RayTay-basic':
ModelRun description: "lightLayerAmplitude_0.07-gravity_1.0"
Generating analysis XML:
Running the Model (saving results in output/raytay-suite/lightLayerAmplitude_0.07-gravity_1.0):
Running model 'RayTay-basic' with command 'mpirun -np 1 /home/psunter/AuScopeCodes/stgUnderworldE-cre
Model ran successfully.
Doing post-run tidyup:
Post-process result 0: with gravity=0.7, init perturbation=0.02
        Maximum value of Vrms was 0.000204, at time 62
Post-process result 1: with gravity=0.8, init perturbation=0.02
        Maximum value of Vrms was 0.000262, at time 68
Post-process result 2: with gravity=0.9, init perturbation=0.02
        Maximum value of Vrms was 0.000295, at time 60
Post-process result 3: with gravity=1, init perturbation=0.02
        Maximum value of Vrms was 0.000365, at time 64
Post-process result 4: with gravity=0.7, init perturbation=0.04
        Maximum value of Vrms was 0.000450, at time 66
Post-process result 5: with gravity=0.8, init perturbation=0.04
        Maximum value of Vrms was 0.000548, at time 64
Post-process result 6: with gravity=0.9, init perturbation=0.04
        Maximum value of Vrms was 0.000658, at time 62
Post-process result 7: with gravity=1, init perturbation=0.04
        Maximum value of Vrms was 0.000778, at time 60
Post-process result 8: with gravity=0.7, init perturbation=0.07
        Maximum value of Vrms was 0.000837, at time 63
Post-process result 9: with gravity=0.8, init perturbation=0.07
        Maximum value of Vrms was 0.001038, at time 61
Post-process result 10: with gravity=0.9, init perturbation=0.07
        Maximum value of Vrms was 0.001306, at time 61
Post-process result 11: with gravity=1, init perturbation=0.07
        Maximum value of Vrms was 0.001603, at time 60
```

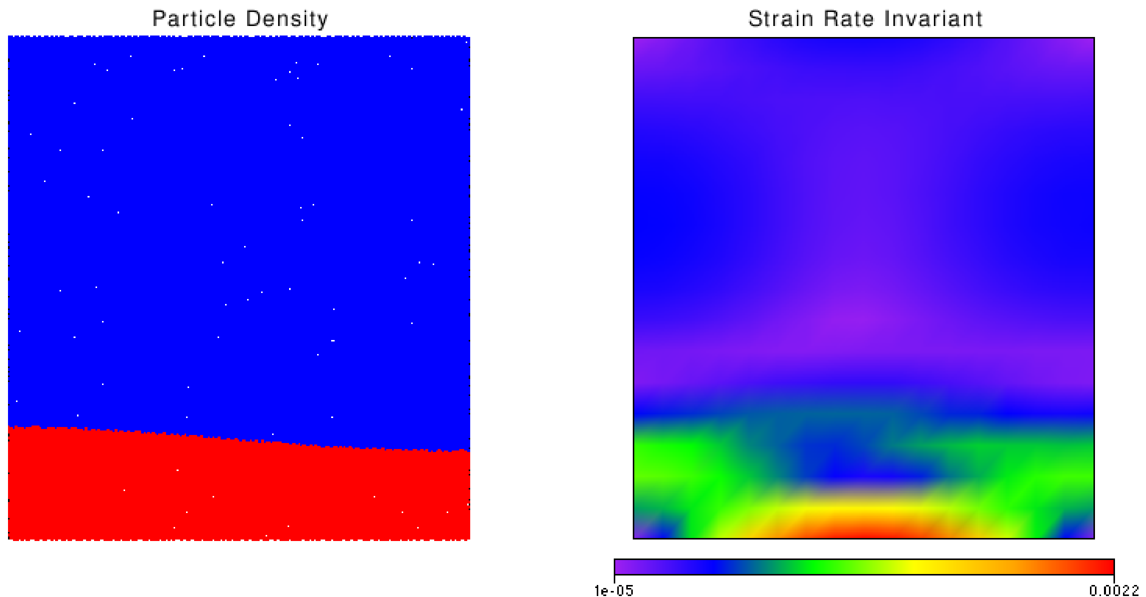
You can see the results of our post-processing for loop here, showing how the VRMS value varied in the different runs due to the different input parameters.

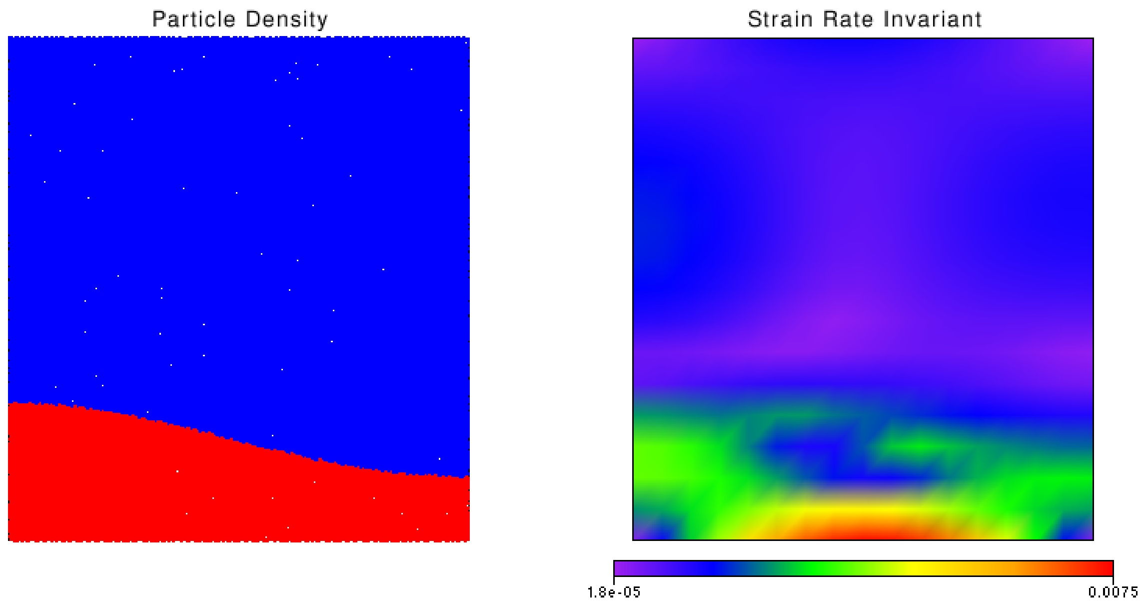
Note: In future, this kind of run-comparison capability will be explicitly supported by CREDO, including ability to plot properties of the different runs in the same graph, and save tables of information comparing runs in a text or CSV file.

If you look at the *output/raytay-suite* directory, you should see a set of sub-directories named:

```
lightLayerAmplitude_0.02-gravity_0.7
lightLayerAmplitude_0.02-gravity_0.8
lightLayerAmplitude_0.02-gravity_0.9
lightLayerAmplitude_0.02-gravity_1.0
lightLayerAmplitude_0.04-gravity_0.7
lightLayerAmplitude_0.04-gravity_0.8
lightLayerAmplitude_0.04-gravity_0.9
lightLayerAmplitude_0.04-gravity_1.0
lightLayerAmplitude_0.07-gravity_0.7
lightLayerAmplitude_0.07-gravity_0.8
lightLayerAmplitude_0.07-gravity_0.9
lightLayerAmplitude_0.07-gravity_1.0
```

where each of them contains the results of the ModelRun with the parameters set to the given values. The results are as usual for a single ModelRun, as described in the *Basic Ray-Tay example*. For example, you might like to compare how varying the light layer shape's amplitude parameter changes the initial conditions of the models by using a file or image browser to examine some of the window.png files. For example, here are two of the images at timestep 1, for a perturbation of 0.02 versus 0.07 respectively:





Each directory will also contain a saved VRMS over time plot, as discussed in *Basic Ray-Tay example*.

4.2.3 Running 2 suites using different numerical features and comparing results

This examples shows how to use CREDO to run two suites of Underworld runs, with varying numerical solver approaches, and compare the performance of the two.

Thus, it is a good baseline to use as an example for more complex performance analysis/accuracy testing.

Note: The actual script is available in the *examples* sub-directory of the CREDO distribution.

Setup

The script is shown below:

```
#! /usr/bin/env python
import os, copy
import csv
from credo.modelrun import ModelRun, SimParams
from credo.modelsuite import ModelSuite
import credo.jobrunner

jobRunner = credo.jobrunner.defaultRunner()

outPathBase = os.path.join('output','PPC_Compare')
if not os.path.exists(outPathBase):
    os.makedirs(outPathBase)

defParams = SimParams(nsteps=2)
stdRun = ModelRun("Arrhenius-normal",
    os.path.join('..','..', 'Underworld', 'InputFiles', 'Arrhenius.xml'),
    simParams=defParams)
ppcRun = ModelRun("Arrhenius-ppc", "Arrhenius.xml",
```

```
    basePath=os.path.join("Ppc_Testing","udw_inputfiles"),
    simParams=defParams)

stdSuite = ModelSuite(os.path.join(outPathBase, "arrBasic"))
ppcSuite = ModelSuite(os.path.join(os.getcwd(), outPathBase, "arrPIC"))

for ii in range(10):
    stdRun.outputPath = os.path.join(stdSuite.outputPathBase, "%.5d" % ii)
    ppcRun.outputPath = os.path.join(ppcSuite.outputPathBase, "%.5d" % ii)
    stdSuite.addRun(copy.deepcopy(stdRun))
    ppcSuite.addRun(copy.deepcopy(ppcRun))

stdResults = jobRunner.runSuite(stdSuite)
ppcResults = jobRunner.runSuite(ppcSuite)

#-----------------------------

cpuRegs = []
cpuPPCs = []
for stdRes, ppcRes in zip(stdResults, ppcResults):
    stdRes.readFrequentOutput()
    ppcRes.readFrequentOutput()
    fStep = stdRes.freqOutput.finalStep()
    cpuReg = stdRes.freqOutput.getValueAtStep('CPU_Time', fStep)
    cpuPPC = ppcRes.freqOutput.getValueAtStep('CPU_Time', fStep)
    print "CPU time regular was %g, PPC was %g" % (cpuReg, cpuPPC)
    cpuRegs.append(cpuReg)
    cpuPPCs.append(cpuPPC)

avgReg = sum(cpuRegs) / len(cpuRegs)
avgPPC = sum(cpuPPCs) / len(cpuPPCs)

print "Avg over 10 runs: regular=%f, PPC=%f" % (avgReg, avgPPC)
sName = os.path.join(outPathBase, "comparePPC.txt")
csvName = os.path.join(outPathBase, "comparePPC-runs.csv")
avgInfo = open(sName, "w")
avgInfo.write("Avg regular = %f\n" % avgReg)
avgInfo.write("Avg PPC = %f\n" % avgPPC)
avgInfo.close()
csvFile = open(csvName, "wb")
wtr = csv.writer(csvFile)
wtr.writerow(["Run", "Reg t(sec)", "PPC t(sec)"])
for runI, (cpuReg, cpuPPC) in enumerate(zip(cpuRegs, cpuPPCs)):
    wtr.writerow([runI, cpuReg, cpuPPC])
csvFile.close()

print "Wrote summary to %s, run results to %s" % (sName, csvName)
```

Similar to the *Using CREDO to run and analyse a Suite of Rayleigh-Taylor problems*, the script first sets up CREDO suites to run, runs them using a JobRunner, then performs some analysis on the results.

Unlike the RayTaySuite example though, in this case we're not varying a parameter across the suites, but are attaching the same model run a fixed number of times to both suites, in order to be able to average results.

See Also:

Modules `credo.modelsuite`, `credo.modelrun`, and `credo.modelresult`.

Note that in this example, we're using the *basePath* option to one of the suites, because the XML Model files must be run in a sub-directory of the current path the script is located in. This is an example of how CREDO can work in with

any arbitrary directory structure that best suits you.

Other things of note about this script as an example are:

- Use of the *os.path.join()* Python standard library function to construct paths, and re-using the `credo.modelsuite.ModelSuite.outputPathBase` attribute to help with constructing these. This is a good practice to keep outputs from an analysis run all in the same directory.
- Use of the Python *csv* library to write custom results of interest to a CSV file as a useful record. A good tutorial on writing CSV files is available on Steven Lott's Python pages.

Expected Results

Running the script should produce a report at the end similar to the following:

```
Doing post-run tidyup:
Restoring initial path '/home/pds/AuScopeCodes/stgUnderworldEGM-packages-devel/Experimental/InputFile
CPU time regular was 0.826317, PPC was 0.830904
CPU time regular was 0.840123, PPC was 0.814179
CPU time regular was 0.809768, PPC was 0.809136
CPU time regular was 0.813736, PPC was 0.827999
CPU time regular was 0.826321, PPC was 0.81942
CPU time regular was 0.815167, PPC was 0.844881
CPU time regular was 0.808789, PPC was 0.849388
CPU time regular was 0.799198, PPC was 0.826902
CPU time regular was 0.849262, PPC was 0.831013
CPU time regular was 0.800944, PPC was 0.810086
Avg over 10 runs: regular=0.818962, PPC=0.826391
```

And also save a text file and CSV file with contents such as:

```
Avg regular = 0.817589
Avg PPC = 0.932345

Run,Reg t(sec),PPC t(sec)
0,0.794358,0.889719
1,0.803794,0.951473
2,0.792373,0.935146
3,0.806235,0.952792
4,0.798883,0.796491
5,0.797865,1.00772
6,0.843122,0.923531
7,0.867709,0.986717
8,0.801994,0.849298
9,0.869554,1.03056
```

4.3 Scientific Benchmarking using CREDO

4.3.1 Running and configuring Scientific Benchmark Tests

The Sci Benchmark testing interface for CREDO is still being developed, but essentially requires the user to write a Python script to configure and run a particular benchmark. This interface was chosen since benchmarks generally require more detailed specification and configuration than standard system tests.

Setup

An example science benchmark test is that a Rayleigh Taylor model can perform as required by the Van Keken benchmark:

```
#! /usr/bin/env python
# Rayleigh-taylor benchmark

import os
from credo.systest import *
from credo.modelrun import ModelRun, SimParams
import credo.jobrunner
from credo.analysis import modelplots
import credo.io.stgfreq as stgfreq
import credo.reporting.standardReports as sReps
from credo.reporting import getGenerators

testSuite = SysTestSuite("Underworld", "SciBench-RayTay")

# Test Configuration shorthand vars
elRes = 128

sciBTest = SciBenchmarkTest("RayleighTaylor-VanKekenBenchmark")
sciBTest.description = """This models the evolution of a sinusoidal
    Rayleigh-Taylor instability. It has a lower density fluid on the bottom
    of the box, and a higher density fluid above. The results are
    benchmarked against P.E. van Keken et al, 'A comparison of methods for
    the modelling of thermochemical convection', <i>JGR</i>, 1997."""

testSuite.sysTests.append(sciBTest)
#Configure suite
mRun = ModelRun("RayleighTaylorOverturn",
    "RayleighTaylorBenchmark.xml",
    simParams=SimParams(dumpevery=10, cpevery=50, stoptime=300.0, nsteps=-2),
    paramOverrides={"elementResI":elRes, "elementResJ":elRes})
sciBTest.mSuite.addRun(mRun, "Ray tay benchmark single run at res %s^3" % elRes)
#Configure test components
sciBTest.setupEmptyTestCompsList()
sciBTest.addTestComp(0, "VRMS of first diapir",
    OutputWithinRangeTC("Vrms", stgfreq.maxOp,
        allowedRange=(2.8e-3, 3.2e-3), tRange=(200,220)))

def customReporting(sciBTest, mResults):
    # This gets called immediately post the models being run.
    # First set up overall analysis images
    modelplots.plotOverAllRuns(mResults, 'Vrms', depName='Time',
        path=sciBTest.outputPathBase)
    import plotCpuTimesAllRuns as plotCpus
    plotCpus.plotAllRuns(sciBTest.outputPathBase)
    sciBTest.mSuite.analysisImages = ["Vrms-multiRunTimeSeries.png",
        'cpuTimePerStep.png']
    # Now specific per-run images
    fStep = mResults[0].freqOutput.finalStep()
    dEvery = sciBTest.mSuite.runs[0].simParams.dumpevery
    lastImgStep = fStep / dEvery * dEvery
    vrmsPeakTime = sciBTest.testComps[0]['VRMS of first diapir'].actualTime
    vrmsPeakStep = mResults[0].freqOutput.getClosest('Time', vrmsPeakTime)[1]
    vrmsPeakImgStep = int(round(vrmsPeakStep / float(dEvery))) * dEvery
    vrmsPeakImgStep = min([vrmsPeakImgStep, lastImgStep])
```

```
    # Create an empty list of images to display
    sciBTest.mSuite.modelImagesToDisplay = [[] for runI in \
        range(len(sciBTest.mSuite.runs))]
    # Choose which model timestep images to display:- note that here we're
    #  programmatically choosing to show the Peak VRMS timestep.
    sciBTest.mSuite.modelImagesToDisplay[0] = [
        (10, "initial state"),
        (120, ""),
        (vrmsPeakImgStep, "near first VRMS peak at t=%f" % vrmsPeakTime),
        (lastImgStep, "")]
    # Here we just ask the CREDO reporting API to get Report Generators for
    # PDF (ReportLab) and RST (Restructured Text) output, and create a
    # standard science benchmark report.
    for rGen in getGenerators(["RST", "ReportLab"], sciBTest.outputPathBase):
        sReps.makeSciBenchReport(sciBTest, mResults, rGen,
            os.path.join(sciBTest.outputPathBase, "%s-report.%s" %\
                (sciBTest.testName, rGen.stdExt)), imgPerRow=2)

sciBTest.setCustomReporting(customReporting)

def suite():
    return testSuite

if __name__ == "__main__":
    jobRunner = credo.jobrunner.defaultRunner()
    testResult, mResults = sciBTest.runTest(jobRunner,
        postProcFromExisting=True, createReports=True)
```

As the code shows, once you set up a SciBenchmarkTest, you need to then add TestComponents that check that the model to be run actually passes some benchmark conditions. In this case, we're checking that the Vrms output into the FrequentOutput.txt each timestep has a maximum value within a specified range, within a specified time range.

Outputs

Running the above script will take some time on most PCs, as we've asked for a significant number of steps at a reasonable resolution.

It should show something like the following at the terminal:

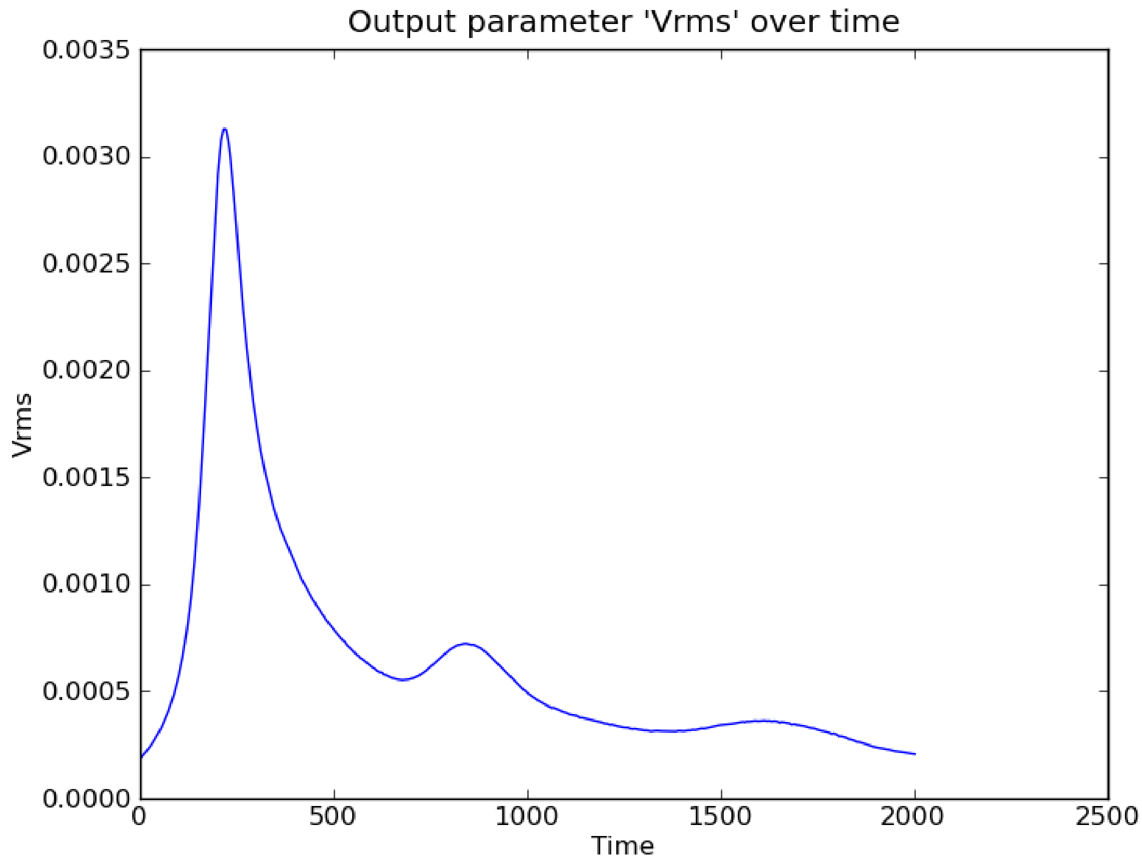
```
Running System test 0, with name 'RayleighTaylorBenchmark-sciBenchmarkTest':
Writing pre-test info to XML
Running the 1 modelRuns specified in the suite
Doing run 1/1 (index 0), of name 'RayleighTaylorBenchmark-sciBenchmarkTest':
ModelRun description: "Run the model needed for the benchmark."
Generating analysis XML:
Running the Model (saving results in output/RayleighTaylor-VanKekenBenchmark):
Running model 'RayleighTaylorBenchmark-sciBenchmarkTest' with command 'mpirun -np 1 /home/psunter/AuS
Model ran successfully.
Doing post-run tidyup:
Checking test result:
Model output 'Vrms' value 0.00312004 within required range (0.0028,0.0032) for all runs.
Test result was Pass
Saved test result to output/RayleighTaylor-VanKekenBenchmark/SysTest-RayleighTaylorBenchmark-sciBench
--------------------------------------------------------------------------------
CREDO System Tests results summary:
Ran 1 system tests, with 1 passes, 0 fails, and 0 errors
--------------------------------------------------------------------------------
```

and will save outputs from the run in the directory *output/RayleighTaylor-VanKekenBenchmark/*.

These include the following XML description of the SysTest as a record of the result:

```
<StgSysTest name="RayleighTaylorBenchmark-sciBenchmarkTest" status="Pass" type="SciBenchmark">
  <description>Runs a user-defined science benchmark.</description>
  <testSpecification>
    <inputFiles>
      <inputFile>RayleighTaylorBenchmark.xml</inputFile>
    </inputFiles>
    <outputPathBase>output/RayleighTaylor-VanKekenBenchmark</outputPathBase>
    <nproc>1</nproc>
    <paramOverrides>
      <param modelPath="maxTimeSteps" paramVal="-1" />
      <param modelPath="stopTime" paramVal="250" />
      <param modelPath="elementResI" paramVal="128" />
      <param modelPath="elementResJ" paramVal="128" />
    </paramOverrides>
  </testSpecification>
  <testComponents>
    <testComponent name="VRMS of first diapir" status="Pass" type="outputWithinRange">
      <specification>
        <outputName value="Vrms" />
        <reductionOp value="&lt;built-in function max&gt;" />
        <allowedRange-min value="0.0028" />
        <allowedRange-max value="0.0032" />
        <tRange-min value="200" />
        <tRange-max value="220" />
      </specification>
      <result status="Pass">
        <statusMsg>Model output 'Vrms' value  within required range
        (0.0028,0.0032) for all runs.</statusMsg>
        <actualValue>0.00312004</actualValue>
        <actualTime>213.728</actualTime>
        <withinRange>True</withinRange>
      </result>
    </testComponent>
  </testComponents>
  <testResult status="Pass">
    <statusMsg>All aspects of the benchmark passed.</statusMsg>
  </testResult>
</StgSysTest>
```

In future we will add capability to automatically save images of important aspects of the System test, such as a graph of VRMS against time like the one below, and compare them to expected results:



4.4 Different ways to launch CREDO scripts

4.4.1 Running a CREDO script (either System testing or analysis) using PBS

As well as launching CREDO scripts directly, you can also use CREDO to run suites of test model runs or analysis model runs on systems that use the PBS scheduler to manage resources and access.

You will need basic famiarity with how PBS works, e.g. see the VPAC page on these.

Essentially, to run a CREDO script using PBS you need to use a PBS script that:

- Loads appropriate modules to run CREDO (Python) and the underlying code (eg PETSc, MPI in the case of Underworld)
- Sets the MPI command for CREDO to use correctly (generally *mpiexec* on HPC systems).
- Launches the CREDO script command as normal.

Here is an example script that launches a CREDO test script:

```
#!/bin/bash
# Usage: qsub pbs-script

# NOTE: To activate a PBS option, remove the whitespace between the '#' and 'PBS'
# As a bare minimum you must set number of cpus and application name.
```

```
# Declares the shell that interprets the job script.
# PBS -S /bin/sh

# To give your job a name, replace "MyJob" with an appropriate name
# PBS -N MyJob

# Select the number of CPUs, if using mpirun.ch_gm need to adjust that line too.

# For Serial Jobs:

# For Parallel Jobs: ie. To reserve 8 nodes with 1 processors each, ie 8cpus.
#PBS -l nodes=2

# For Parallel Jobs: ie. To reserve 4 nodes with 2 processors each, ie 8cpus.
# PBS -l nodes=4:ppn=2

# For Parallel jobs with an odd number of processes; eg. 5 CPUs:
# Request 2 full nodes 'nodes=2', with 2 CPUs each 'ppn=2' and 1 extra CPU '+1'
# PBS -l nodes=2:ppn=2+1

# set your minimum acceptable walltime=hours:minutes:seconds
# PBS -l walltime=xx:xx:xx
#PBS -l walltime=00:10:00

## Specify your email address to be notified of progress.
# PBS -M yourname@domain
#PBS -M patdevelop@gmail.com

# To receive an email:
#         - job is abored: 'a'
#        - job begins execution: 'b'
#         - job terminates: 'e'
#         Note: Please ensure that the PBS -M option above is set.
#
#PBS -m abe

# To capture to your root dir (rather than working dir ):
#        - output stream: 'o'
#          The output stream will be saved as job_name.osequence
#        - error stream:         'e'
#          The error stream will be saved as job_name.esequence

# PBS -k oe

# Changes directory to your execution directory (Leave as is)
cd $PBS_O_WORKDIR

# Command to run a job, either mpi or serial :
# For mpi its a choice of mpiexec or mpirun.ch_gm. We recomend
# mpiexec, its smarter and cleans up after itself.
# Serial or single cpu app need only the app name and any options.

# Usage: mpirun.ch_gm -machinefile $PBS_NODEFILE -np 4 ./program
# Usage: mpiexec ./program
# Usage: ./program


#/usr/local/bin/mpiexec ./myapp
```

```
source /usr/local/Modules/default/init/bash

module load petsc
module load python
module load hdf5
#/usr/local/Modules/default/bin/modulecmd sh petsc

source ../../../updatePathsCREDO.sh
export MPI_RUN_COMMAND=mpiexec
./testAll_lowres.py
```

CHAPTER

FIVE

CREDO PYTHON SOURCE API DOCUMENTATION

These are the API (Application Programming Interface) documents for CREDO, describing the Python classes and functions provided by the library and how they function. They are automatically generated from documentation strings "docstrings" in the CREDO Python source code - so in an interactive Python session you can access this same information using the help() function on objects in your session.

For users: you may wish to start with the *Examples of how to use CREDO* section, and then refer back to this API as you become more familiar with CREDO and need to look up details.

For developers: the *Core CREDO Architecture* section gives an initial overview of how the various sub-packages in CREDO relate and are integrated.

Note: As well as navigating via the contents below, you can also use the name-sorted *genindex* of all functions and members, or the module-sorted *modindex*.

API Documentation contents:

5.1 Model API

5.1.1 `credo.modelrun`

credo.modelrun.SimParams

credo.modelrun.ModelRun

credo.modelrun.StgParamInfo

credo.modelrun.JobParams

A core module for CREDO, since it defines and manages running models of a StGermain-based code such as Underworld.

Primary interface is via the `ModelRun`, which enables you to specify, configure and run a Model, and save records of this as an XML. This process will produce a `credo.modelresult.ModelResult` class.

class `credo.modelrun.`**`JobParams`**(***kwargs*)

Bases: `dict`

Small class, to record parameters that specify job control of a ModelRun, such as numbers of processors used.

All attributes are stored as regular dictionary parameters, to facilitate easy updating.

`writeInfoXML`(*parentNode*)

Writes information about this class into an existing, open XML doc node, in a child list

class `credo.modelrun.`**`ModelRun`**(*name, modelInputFiles, outputPath=None, basePath=None, logPath=None, cpReadPath=None, nproc=1, simParams=None, paramOverrides=None, solverOpts=None, xmlExtras=None, inputFilePath=None*)

A class to keep records about a StgDomain/Underworld Model Run, including access to the underlying XML of the actual model.

This is one of the key core classes in CREDO, useful on it's own for managing analysis, but also underlying the `credo.systest` module.

The basic usage pattern of this class is that a ModelRun needs to be constructed and configured to specify the basic XML files defining a StGermain Model, but also any customisations and `credo.analysis.api.AnalysisOperation` classes attached to be performed.

After the model is run (see `credo.jobrunner`), a `ModelResult` will be produced as a record of the run and for further analysis.

Examples of using the ModelRun are documented in CREDO, see *Doing Model analysis with CREDO*.

Key attributes:

name
name of the modelRun.

basePath
The path from which all paths to model input files (on the local machine) are specified relative to, and which the job will run in (if running on local machine).

modelInputFiles
'Input files' that comprise the XML model that will be run.

outputPath
Output path that all model results will be saved to (is passed through to StGermain).

cpReadPath
Path that checkpoints are read from (is passed through to StGermain).

logPath
Path that log files of the run will be saved to.

jobParams
A `JobParams` class, to record options needed to define how the model should be actually run (eg number of procs to use).

simParams
A `SimParams` object, recording key parameters to control the model. This defaults to *None*, unless the user specifically instantiates one either in the constructor or subsequently sets this directly later. This idiom assumes that if simParams is *None*, all the necessary parameters are defined in the StGermain XML directly.

Note: You shouldn't provide a particular parameter in both a SimParams object, and as an override of the ModelRun's `paramOverrides` list. At both construction-time and just before the model is run, a check is performed that this has not occurred.

---

solverOpts
The name of the file storing options passed through to the PETSc numerical solver framework. Depending on the Model being solver, these can have an important role determining the performance and numerical approach taken. See the 'System Routines' section of PETSc 2.0.16 Changes log. This option file is separate to the `paramOverrides` attribute, although the options passed through to PETSc may be used to further customise matrices specified as part of StGermain components.

A file recording the options used for a modelRun will be saved in the output path, with name specified in `credo.modelrun.SOLVER_OPTS_RECORD_FILENAME`.

Note: the interface here has been kept as specifying a filename with the options rather than using a Python list, as the solver options can run into the hundreds, and generally several of these files are available and maintained by the developers for different solvers.

---

paramOverrides
A list of StGermain XML parameter overrides that should be applied, in the form (XML_param_path, override_value). These will then be passed through at the command line.

E.g. if paramOverrides is set to [("gravity",0.8)], it means that the "gravity" model parameter will be set to the value 0.8.

Note: Currently this is a conceptual "catch-all" for overriding model parameters, for things not specific to the `simParams` list. In future this approach may be refactored.

---

analysisOps
A list of `credo.analysis.api.AnalysisOperation` that are associated with this ModelRun, and will be applied when the model is actually run (which involves writing and submitting additional StGermain XML).

analysisXML
Initally *None*, this will be populated with the filename of the additional XML document written containing parameter overrides, and requested analysis operations.

cpFields
A list of fields that the user wishes to checkpoint in the run. Defaults to [], in which case the list (if any) in the model run's XML will be left as-is.

analysisXMLGen(*filename=None*)
Generates an XML file, in StGermainData XML format, to over-ride necessary parameters of the model as specified on this ModelRun instance. Returns the name of the just-written XML file.

Overrides can have the following main sources:

- Over-ridden simulation parameters that have been specified as members of the ModelRun itself, such as cpReadPath, and cpFields;
- Over-ridden simulation parameters on this ModelRun's SimParams attribute (if it exists);
- Requested analysis operations that've been added to the ModelRun, as specified in the self.analysisOps member list.

Note: Remember that as well as those overrides written to this XML, the user can over-ride particular parameters in the ModelRun via the command line by setting the self.paramOverrides member dictionary.

---

checkSolverOptsFile()

checkValidRunConfig()
Check the given modelRun is valid and ready to be run.

defaultModelRunFilename()
Calculates and returns a default filename for the ModelRun's XML record filename.

genFlattenedXML(*cmdLineOverrides=None*, *flatFilename=None*)

getModelRunAppExeCommand()
Return the full path of the executable of the modelling program. (e.g. "/usr/local/bin/StGermain")

getModelRunCommand(*extraCmdLineOpts=None*, *absXMLPaths=False*)
Given a model run, construct the command needed to run that model, and return as a string.

Parameters

- **extraCmdLineOpts** – any extra command line options, to be passed straight through to the model.
- **absXMLPaths** – if True, converts any Model XML input files to absolute paths first in cmd line.

getSimParams()
Utility function to get SimParams - since in the current design the self.simParams parameter may be 'None', and we need to read from the model XML.

getStdErrFilename()
Get the name of the file this Model's stderr needs to/has been saved to.

getStdOutFilename()
Get the name of the file this Model's stdout needs to/has been saved to.

postRunCleanup()
function designed to be run after a modelRun has completed, and will do any post-run cleanup to get ready for analysis - e.g. moving files into the output directory that were created to configure the run and need to be kept.

preRunPreparation()
Do any preparation necessary before the run itself proceeds.

prepareOutputLogDirs()
Prepare the output and log dirs - usually in preparation for running a `credo.modelrun.ModelRun`.

setPath(*inPath*)
Do any needed manipulations when setting paths.

writeInfoXML(*writePath=''*, *filename=''*, *update=False*, *prettyPrint=True*)
Writes an XML recording the key details of this ModelRun, in CREDO format - useful for benchmarking etc.

writePath and *filename* can be specified, if not they will use default values (the outputPath of the model, and the value returned by `defaultModelRunFilename()`, respectively).

class credo.modelrun.**SimParams**(***kwargs*)
A class to keep records about the simulation parameters used for a StgDomain/Underworld Model Run, such as number of timesteps to run for. Has functionality to save this list to an XML record, and also read them from a StGermain XML.

After construction, it will make all these parameters directly available as attributes of the SimParams object.

stgParamInfos
A dictionary of `StgParamInfo`, specifying which parameters are actually controlled by this class. The keys are the short-hand names which can be used to refer to them, as well as Stgermain names.

checkNoDuplicates(*paramOverridesList*)
Function to check there are no duplicates between sim param overrides set, and cmd-line parameter overrides.

checkValidParams()
Checks that the parameter set is valid to run a StGermain model (e.g. that either the stop time, or total number of steps, is set).

getParam(*paramName*)
Get the value of a parameter with given paramName.

nearestDumpStep(*finalStep*, *inputStep*)
Utility method to get the nearest step at which a dump result was created.

readFromStgXML(*inputFilesList*, *basePath*, *cmdLineOverrides*)
Reads all the parameters of this class from a given StGermain set of input files

setParam(*paramName*, *val*)

writeInfoXML(*parentNode*)
Writes information about this class into an existing, open XML doc node, in a child list

writeStgDataXML(*xmlNode*)
Writes the parameters of this class as parameters in a StGermain XML file

class credo.modelrun.**StgParamInfo**(*stgName*, *pType*, *defVal*)
A simple Class that keeps track of the type of a StgParam, and it's full name.

stgName
The name of this parameter used in the StGermain dictionary and Model XML files.

pType
Type of this parameter (will be used in casting etc).

defVal
Default value of the parameter.

checkType (*value*)
Checks that the value is of the correct type of this parameter.

credo.modelrun.**checkParamOverridesTypes** (*paramOverrides*)
Checks that, for a given list of paramOverrides, each is of a valid type that can actually be successfully used in a StGermain dictionary.

credo.modelrun.**generateResOpts** (*resTuple*)
Given a tuple of desired resolutions for a model, convert this to options to pass to StG on the command line

credo.modelrun.**getParamOverridesAsStr** (*paramOverrides*)
Given a list of parameter overrides, return these as a string ready for passing to StGermain on the command line.

credo.modelrun.**strRes** (*resTuple*)
Turn a given tuple of resolutions into a string, suitable for using as an output dir

credo.modelrun.**writeParamOverridesInfoXML** (*paramOverrides*, *parentNode*)
Writes a record, under the given parentNode, of all the parameter overrides specified in the list paramOverrides.

credo.modelrun.**writeSolverOptsInfoXML** (*solverOpts*, *parentNode*)
Writes a record, under the given parentNode, of the solver options file used.

5.1.2 credo.modelresult

credo.modelresult.ModelResult

This module allows recording and post-processing of the results of running a StGermain-based application.

The primary interface is via the ModelRun class.

See Also:

credo.modelrun.

class credo.modelresult.**ModelResult** (*modelName*, *outputPath*)
A class to keep records about the results of a StgDomain/Underworld model run. These are normally produced as a result of running a ModelRun.

Note: In future, we intend to add the ability to create a ModelResult class by reading in an XML file specifying output directory, etc.

modelName
Name of the Model that was run.

outputPath
Path to the output results the ModelRun produced.

jobMetaInfo
A `jobrunner.api.JobMetaInfo`, recording information about the run such as time taken, Memory usage etc (generally attached by a `credo.jobrunner.api.JobRunner` soon after the ModelResult created).

fieldResults
A list of FieldComparisonResult objects.

Note: is a legacy of early design of CREDO to allow construction of XML files from pre-existing sys test scripts, may be removed soon.

---

freqOutput
Initially *None*, if `readFrequentOutput()` is called, this will be populated with a reference to a `credo.io.stgfreq.FreqOutput` class, to allow post-processing of info in the Frequent Output file saved as part of the model run.

defaultRecordFilename()
Get the default filename to use, based on the model name of a particular model.

readFrequentOutput()
Opens and reads in info from the Frequent Output file produced as part of the run, and saves to the attribute `freqOutput`.

readFromRecordXML(*xmlFilename*)

recordFieldResult(*fieldName*, *tol*, *errors*)
Records the info required for a FieldResult in the array of stored FieldResults kept by the ModelResult. Returns a reference to the just-added FieldResult.

writeRecordXML(*outputDir=''*, *filename=''*, *prettyPrint=True*)
Write an XML record of a `ModelResult`.

credo.modelresult.**getSimInfoFromFreqOutput**(*outputPath*)
utility function to get basic information about the simulation from the FrequentOutput.dat, given a particular output Path.

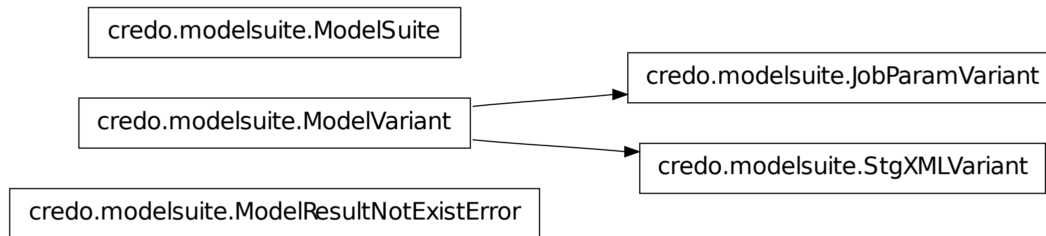
See Also:

`credo.io.stgfreq`.

credo.modelresult.**readModelResultFromPath**(*path*)

credo.modelresult.**updateModelResultsXMLFieldInfo**(*filename*, *newFieldResult*, *prettyPrint=True*)
Update an existing XML record of a `ModelResult` with info about a particular fieldResult.

5.1.3 credo.modelsuite



This module allows the running of a whole suite of related `ModelRun` s, and managing and analysing their results as a consistent set.

The key class in this module is the `ModelSuite`.

For example usage, see the CREDO documentation Examples section, *Examples of how to use CREDO*, e.g. *Using CREDO to run and analyse a Suite of Rayleigh-Taylor problems*.

This class also performs a key role in the System tests provided in the `credo.systest` module.

class `credo.modelsuite.`**`JobParamVariant`**(*jobParam*, *paramRange*)

Bases: `credo.modelsuite.ModelVariant`

A `ModelVariant` designed to modify job parameters.

`applyToModel`(*modelRun*, *ii*)

Apply the ii-th value in the attr:.*paramRange* to a particular `ModelRun`.

`cmdLineStr`(*ii*)

Return the command-line string to apply this value.

exception `credo.modelsuite.`**`ModelResultNotExistError`**

Bases: `exceptions.Exception`

Exception for specifying that a Model Result that CREDO was asked to read in, doesn't exist.

class `credo.modelsuite.`**`ModelSuite`**(*outputPathBase*, *templateMRun=None*)

A class for running a suite of Models (e.g. a group for profiling, or a System Test that requires multiple runs).

The two main ways of using this class are:

- Creating a `ModelSuite`, and then adding `ModelRun` s to the suite using the `addRun()` method.
- Creating a `ModelSuite`, and providing a `ModelRun` as a template, then adding `StgXMLVariant` s to define what sort of parameter sweep should be performed. In this case, `generateRuns()` needs to be called after all variants have been added.

`outputPathBase`

The base path to use for saving model results under.

`runs`

A list of `ModelRun` s to be run as part of the suite. See `generateRuns()` and `addRun()`.

`runDescrips`

Short (eg 1 line) textual description for each ModelRun stored in the `runs`.

runCustomOptSets
Custom sets of options (to be used at the command line) associated with each run in `runs` (strings).

resultsList
Initially *None*, after the suite has been run (using `runAll()`), saves a reference to all `ModelResult` s generated.

subOutputPathGenFunc
This function will can be used to customise the model sub-path based on each modelRun. Override it if you wish to use other than the default.

templateMRun
(Optional) setting this to an `ModelRun` means this run can be used as a "template" to add variants to, and create a parameter sweep over this run.

iterGen
(Related to auto-generation): A generator function to create an iterator to use when auto-generating a suite based on modelVariants. See Python module `itertools` module for more.

modelVariants
Set of `StgXMLVariant` s to apply to the template run in order to auto-generate a suite to vary certain parameters. See `templateMRun` for more.

addRun(*modelRun*, *runDescrip=None*, *runCustomOpts=None*, *forceOutputPathBaseSubdir=True*)
Add a model run to the list of those to be run.

Parameters

- **modelRun** – A `ModelRun` to be added.
- **runDescrip** – An (optional) string describing the run.
- **runCustomOpts** – (optional) string of any custom options that should be passed through to StGermain, only for this run.
- **forceOutputPathBaseSubdir** – if True (default), will update the model run's output dir to enforce it's a subdir of `outputPathBase`

Returns the index of the newly added run in the modelRun list.

addVariant(*name*, *modelVariant*)
Add a `StgXMLVariant` to the list to be applied to a template run. See `modelVariants`.

cleanAllLogFiles()
Remove all stdout and stderr files from each ModelRun's designated output and log paths.

cleanAllOutputPaths()
Remove all files in each model's output path. Useful to get rid of results still there from previous jobs. Doesn't delete sub-directories, in case they are other model runs' results that should be ignored.

generateRuns(*iterGen=<function productCalc at 0x2cf3230>*)
When using a template modelRun, will generate runs for the suite based on it. The generated runs are saved to the `runs` attribute ready to be run using `runAll()`.

This requires that there are one or more `StgXMLVariant` recorded on the class already.

Parameters **iterGen** – this determines what iterator strategy should be used to generate the runs. Defaults to a product, but a simple "zip" style can be achieved using the itertools.izip iterator generating function. See the Python `itertools` module for more.

getCustomOpts(*runI*, *extraCmdLineOpts*)
Get the custom opts (as a string) to apply for modelRun runI.

getRunByName (*runName*)
Get a modelRun instance from the suite with a particular name.

getRunIndex (*runName*)
Get the index within the suite of a run with the given name.

preRunCleanup ()
Convenience function to call all sub-methods for tasks to do before running to clean up directories.

readResultsFromPath (*basePath*, *overrideOutputPath=None*, *checkAllPresent=True*)
Read the results generated for a given ModelSuite located off the given basePath where the suite was run, and return the list of results.

This will ignore results in the directory not related to this suite.

Parameters

- **overrideOutputPath** – if specified, this path overrides the default outputPath of the suite itself to search for the results. (I.e. useful if you are reading from a previous suite with different output path.)
- **checkAllPresent** – if True this will check that all runs expected for the suite were found in the list of results.

writeAllModelResultXMLs ()
Save an XML record of each ModelResult currently in `resultsList`.

writeAllModelRunXMLs ()
Save an XML record of each ModelRun currently in `runs`.

class credo.modelsuite.**ModelVariant** (*paramRange*)
A class that can be added to a `ModelSuite` to help auto-generate a suite of ModelRuns to perform, where a particular parameter is being varied over a certain range.

This is an abstract base class, you should select an actual ModelVariant.

paramRange
A list, containing the values that the parameter should be varied over. E.g. [0,1,2], or [5.6, 7.8, 9.9]. Needs to be of the correct type for the particular parameter. The Python range() function can be useful in generating such a list.

applyToModel (*modelRun*, *ii*)
Function to apply the ii-th value of paramRange to a model.

valLen ()
Returns the length of the list of parameter values to vary specified in `paramRange`.

valStr (*ii*)
Return a string version of the ii-th parameter value.

class credo.modelsuite.**StgXMLVariant** (*paramPath*, *paramRange*)
Bases: `credo.modelsuite.ModelVariant`

A `ModelVariant` designed to modify StGermain XML model input parameters.

paramPath
The value to use when over-riding the parameter in a StGermain dictionary, using the StGermain command line format.

E.g. Setting “gravity” would override the gravity parameter in the dictionary, whereas setting to “components.initialConditionsShape.startX” would override the startX parameter, within the initialConditionsShape component.

applyToModel (*modelRun*, *ii*)
Apply the ii-th value in the attr:.*paramRange* to a particular `ModelRun`.

cmdLineStr (*ii*)
Return the command-line string to apply this value.

credo.modelsuite.**getModelResultsArray** (*baseName*, *baseDir*)
Post-processing: given a base model name and base output directory, search this directory for model results, and read into a list of `ModelResult` .

Note: Needs more checking added, and ability to recover metadata about the ModelRuns.

credo.modelsuite.**getOtherParamValsByVarRunIs** (*varRunIsMap*, *varDicts*, *otherParam*)
Given a varRunIsMaps generated by `varRunIsMap()`, a varDict and the name of another variant param in the dict, returns a mapping from the variant values to the values of the other param at the same indices.

credo.modelsuite.**getParamValues** (*modelVariants*, *iterGen*)
Shortcut to create a list of param values using `getParamValuesIter()`

credo.modelsuite.**getParamValuesIter** (*modelVariants*, *iterGen*)
Given a list of model variants and an iterator generator function (eg itertools.izip or itertools.product) to use, generates a specific iterator that can be used on the modelVariants to obtain the actual param values.

credo.modelsuite.**getResultsByVarRunIs** (*varRunIsMap*, *results*)
Given a varRunIsMap generated by `getVarRunIs()` and an array of results, gives a mapping directly from variant values to corresponding result.

credo.modelsuite.**getSubdirTextParamVals** (*modelVariants*, *paramIndices*)
Creates a subdirectory text based on the names and values of each variant.

credo.modelsuite.**getSubdir_RunIndex** (*modelRun*, *modelVariants*, *paramIndices*, *runIndex*)
Simply prints the index of the run as a subdirectory.

credo.modelsuite.**getSubdir_RunIndexAndText** (*modelRun*, *modelVariants*, *paramIndices*, *runIndex*)
Subdir is based on both the run index, and the textual variant names.

credo.modelsuite.**getSubdir_TextParamVals** (*modelRun*, *modelVariants*, *paramIndices*, *runIndex*)
Generate an output sub-directory name for a run with a printed version of `ModelSuite.modelVariants` names, and vales for this run. (Good in the sense of being fairly self-describing, but can be long if you have many model variants).

credo.modelsuite.**getTextParamValsSubdirs** (*modelVariants*, *indicesIt*)
Given a list of `ModelVariants` and an index iterator, returns a list of all subDirs to use.

credo.modelsuite.**getVarRunIs** (*varName*, *modelVariants*, *runDicts*)
Given a variant name, modelVariants dict and iterGen function, returns a mapping of values of the named modelVariant to run indices

credo.modelsuite.**getVariantCmdLineOverrides** (*modelVariants*, *indicesIt*)
Generates a list of strings to use at cmd line for each model run, given a list of `StgXMLVariant` and an iterator into them (e.g. generated by `getVariantIndicesIter()`.

credo.modelsuite.**getVariantIndicesIter** (*modelVariants*, *iterGen*)
Given a list of model variants and iterator generator function to use, generates an iterator of indices into the modelVariants list.

credo.modelsuite.**getVariantNameDicts** (*modelVariants*, *indicesIt*)
Generates a list of dictionaries of parameters to be modified for each model run, given a list of `StgXMLVariant` and an iterator into them (e.g. generated by `getVariantIndicesIter()`.

`credo.modelsuite.`**`getVariantParamPathDicts`**`(`*modelVariants*, *indicesIt*`)`
Generates a list of dictionaries of parameters to be modified for each model run, given a list of `StgXMLVariant` and an iterator into them (e.g. generated by `getVariantIndicesIter()`.

`credo.modelsuite.`**`writeInputsOutputsToCSV`**`(`*mSuite*, *observablesDict*, *fname*`)`
Write a CSV file, containing all the ModelVariants defined for a ModelSuite, and also all the observables in the observablesDict.

Parameters

- **observablesDict** – a dictionary of 'observables', each entry in the form 'obsName':[obsVals for each run], e.g. "vrms":[0.6, 0.8, 0.9].
- **fname** – file name of the CSV file to create, inside the model suite's base output path.

Note: Could be a function on the ModelSuite?

5.2 Utils API

5.2.1 `credo.utils`

A module for general utility functions in CREDO, that don't clearly fit into other modules.

`credo.utils.`**`dictAsPrettyStr`**`(`*inDict*`)`
A small function to create a string representation of a dictionary, by getting the str() of each item in a dict, not the repr(). Useful for floating points for example to be 'prettier' (less zeros after the number).

Note: No effort has been made to ensure this is super-efficient for large dictionaries, it's suited to small lists of parameters

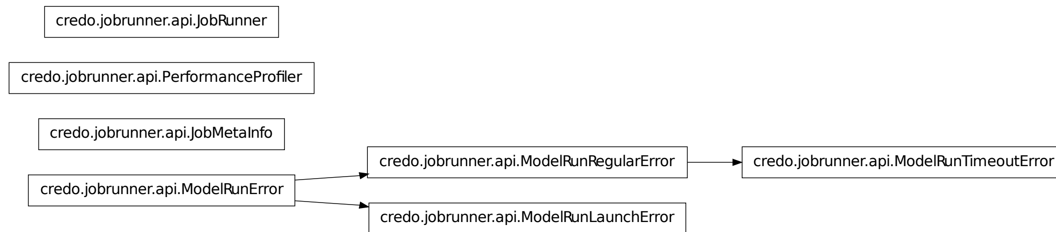
`credo.utils.`**`getCallingPath`**`(`*stackNum*`)`
Get the path of the calling stack at stackNum levels higher.

`credo.utils.`**`productCalc`**`(`**args*, ***kwds*`)`
Basic implementation of itertools.product from Python 2.6: For Python 2.5 backward compatibility. productCalc('ABCD', 'xy') –> Ax Ay Bx By Cx Cy Dx Dy productCalc(range(2), repeat=3) –> 000 001 010 011 100 101 110 111

5.3 JobRunner API

This module allows running CREDO jobs using various approaches - e.g. via MPI locally, via PBS scripts in a queueing system, or via grid submission.

5.3.1 `credo.jobrunner.api`



class `credo.jobrunner.api.`**`JobMetaInfo`**(*simtime*)
A simple class for recording meta info about a job, such as walltime, memory usage, etc.

`simtime`
Simulated time the model ran for.

`readFromXMLNode`(*xmlNode*)

`verbPlatformString`()
Returns a useful string about the platform, for printing.

`writeInfoXML`(*xmlNode*)
Writes information about this class into an existing, open XML doc node

class `credo.jobrunner.api.`**`JobRunner`**
Class used for running ModelRun instances. This is an abstract base class, user code will need to choose a concrete implementation. Is designed to allow both serial, and parallel non-blocking job submission and reporting.

`runSuiteNonBlockingDefault`
Determines that for a given job runner, whether suites should be run in non-blocking mode by default, or

`profilers`
List of `PerformanceProfiler` that will be applied to report on any ModelRuns that this JobRunner is used for.

`defaultProfiler`
Profiler that will be used to provide "default" results in the JobMetaInfo.

`attachPerformanceInfo`(*jobMI*)
Attach relevant performance information to the jobMI (`JobMetaInfo`), such as time use, memory use, etc

`attachPlatformInfo`(*jobMI*)
Attach provenance info relevant to the platform used to run the job to the JobMetaInfo object.

`blockResult`(*modelRun*, *jobMetaInfo*)
Block on a modelRun until the result is completed ... requires appropriate info to be passed in the job-MetaInfo object.

`blockSuite`(*modelSuite*, *jobMetaInfos*)
Blocks on each ModelRun in a Suite, given a list of JobMetaInfos for each run.

`runModel`(*modelRun*, *prefixStr=None*, *extraCmdLineOpts=None*, *dryRun=False*, *maxRunTime=None*)
Run the specified modelRun, and return the ModelResult.

Parameters

- **modelRun** – the `ModelRun` to be run.
- **prefixStr** – optional precursor string for running the model, e.g. for timing.
- **extraCmdLineOpts** – if specified, these extra cmd line opts will be passed through on the command line to the run, extra to any `ModelRun.simParams` or `ModelRun.paramOverrides`.
- **dryRun** – If set to True, just print out what *would* be run, but don't actually run anything.
- **maxRunTime** – maximum time (in seconds) a model should be allowed to run for before killing the job.

Returns A `ModelResult` recording the results of the run.

runSuite (*modelSuite*, *prefixStr=None*, *extraCmdLineOpts=None*, *dryRun=False*, *maxRunTime=None*, *runSuiteNonBlocking=None*, *writeRecords=True*)

Run each ModelRun in the suite - with optional extra cmd line opts. Will also write XML records of each ModelRun and ModelResult in the suite.

Input arguments same as for `runModel()`, except those listed below:

Parameters

- **runSuiteNonBlocking** – controls whether the suite will be run "non-blocking", i.e. all modelRuns submitted initially, then a separate phase to block until they're all completed.
- **writeRecords** – sets whether you want each ModelRun in the suite, and each ModelResult generated, to automatically write an XML record of itself in default location as it is run/produced.

Returns a reference to the `resultsList` containing all the ModelResults generated.

setup ()

Does any necessary setup checks to run models.

By default, does nothing - sub-classes need to override.

submitRun (*modelRun*, *prefixStr=None*, *extraCmdLineOpts=None*, *dryRun=False*, *maxRunTime=None*)

Submit the job to be run. TODO: comment on parameters ... Returns: a jobMetaInfo (that can later be attached ...)

submitSuite (*modelSuite*, *prefixStr=None*, *extraCmdLineOpts=None*, *dryRun=False*, *maxRunTime=None*, *writeRecords=True*)

Submits each modelRun in a suite to be run, and returns a list of all jobMetaInfos for the submitted jobs.

exception `credo.jobrunner.api.`**ModelRunError** (*modelName*)

Bases: `exceptions.Exception`

Base class of ModelRunError exception hierarchy.

exception `credo.jobrunner.api.`**ModelRunLaunchError** (*modelName*, *runExecCmd*, *launchHint=None*)

Bases: `credo.jobrunner.api.ModelRunError`

An Exception for when Models fail to run due to being unable to launch the run process in some way.

runExecCmd

command used to launch the job.

exception `credo.jobrunner.api.`**ModelRunRegularError** (*modelName*, *retCode*, *stdOutFilename*, *stdErrFilename*)

Bases: `credo.jobrunner.api.ModelRunError`

An Exception for when Models fail to run.

exception credo.jobrunner.api.**ModelRunTimeoutError**(*modelName*, *stdOutFilename*, *stdErrFilename*, *maxRunTime*)

Bases: credo.jobrunner.api.ModelRunRegularError

An Exception for when Models fail to run due to timing out.

maxRunTime

maximum time to run that the model exceeded, in seconds.

class credo.jobrunner.api.**PerformanceProfiler**(*typeStr*)

Class to use to attach to JobRunner instances, which will then profile performance of each ModelRun ran by given JobRunner.

This is an abstract base class, user code will have to select a concrete instantiation.

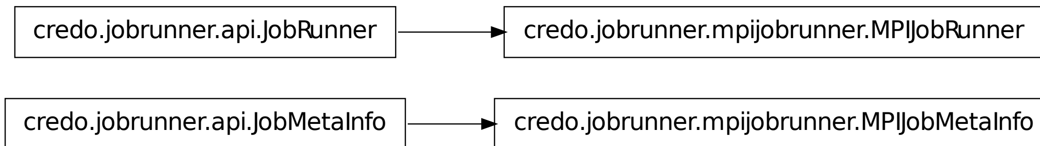
attachPerformanceInfo(*jobMetaInfo*, *modelResult*)

modifyRun(*modelRun*, *modelRunCommand*, *jobMetaInfo*)

setup(*modelName*, *modelBasePath*, *modelOutputPath*, *jobMetaInfo*)

Do any necessary setup functions.

5.3.2 credo.jobrunner.mpijobrunner



class credo.jobrunner.mpijobrunner.**MPIJobMetaInfo**

Bases: credo.jobrunner.api.JobMetaInfo

writeInfoXML(*xmlNode*)

class credo.jobrunner.mpijobrunner.**MPIJobRunner**

Bases: credo.jobrunner.api.JobRunner

archiveRunCommand(*modelRun*, *runCommand*)

Save the given runCommand to a file in output directory.

attachPlatformInfo(*jobMI*)

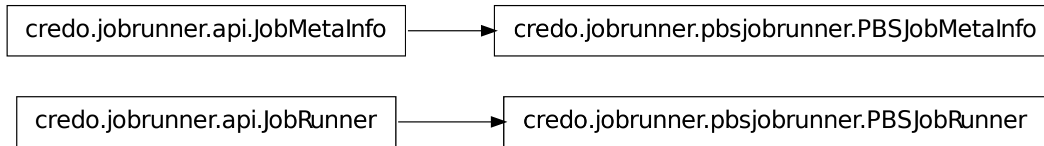
blockResult(*modelRun*, *jobMI*)

setup()

submitRun(*modelRun*, *prefixStr=None*, *extraCmdLineOpts=None*, *dryRun=False*, *maxRunTime=None*)

See credo.jobrunner.api.JobRunner.submit().

5.3.3 `credo.jobrunner.pbsjobrunner`



class `credo.jobrunner.pbsjobrunner.`**`PBSJobMetaInfo`**

Bases: `credo.jobrunner.api.JobMetaInfo`

`writeInfoXML`(*xmlNode*)

class `credo.jobrunner.pbsjobrunner.`**`PBSJobRunner`**

Bases: `credo.jobrunner.api.JobRunner`

A JobRunner to submit CREDO jobs via creating PBS script files, and submitting these via command-line utils like qsub.

Note: this module is currently still in development, and needs tuning for different HPC machines.

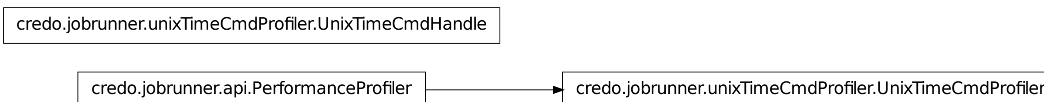
`blockResult`(*modelRun*, *jobMetaInfo*)

`setup`()

`submitRun`(*modelRun*, *prefixStr=None*, *extraCmdLineOpts=None*, *dryRun=False*, *maxRunTime=None*)

See `credo.jobrunner.api.JobRunner.submit()`.

5.3.4 `credo.jobrunner.unixTimeCmdProfiler`



class `credo.jobrunner.unixTimeCmdProfiler.`**`UnixTimeCmdHandle`**(*resFName*)

class `credo.jobrunner.unixTimeCmdProfiler.`**`UnixTimeCmdProfiler`**(*fmtEls=None*)

Bases: `credo.jobrunner.api.PerformanceProfiler`

A performance profiler that uses the Unix 'time' command.

`fmtEls`

Format elements to be used in determining what gets profiled.

`attachPerformanceInfo`(*jobMetaInfo*, *modelResult*)

modifyRun (*modelRun*, *oldModelRunCommand*, *jobMetaInfo*)

setup (*modelName*, *modelBasePath*, *modelOutputPath*, *jobMetaInfo*)

credo.jobrunner.unixTimeCmdProfiler.**getFmtString** (*fmtEls*, *fmtSpec=', '*, *fmtSep='\n'*)

Return the format string to use, given a list of format elements, using standard separators, so it can be parsed later on.

Parameters **fmtEls** – a list of Tuples, of form (nameStr, fmtItem) - where nameStr is the name of the entry to return, and fmtItem is the 'resource specifier' character as specified in 'man time'.

e.g. [("runTime", "E")] - where "E" is the specifier for Elapsed time

credo.jobrunner.unixTimeCmdProfiler.**getResDict** (*resFName*, *fmtSpec=', '*, *fmtSep='\n'*)

Get a dictionary of results, contained in the given resFName, formatted in the standard way as done by getFmtString.

credo.jobrunner.unixTimeCmdProfiler.**getRunPrefix** (*resFName*, *fmtStr*)

Get the prefix for a run that will apply the time command.

Do "man time" for more info.

Parameters

- **resFName** – filename that 'time' profiling results should be saved to.
- **fmtStr** – format string to use.

credo.jobrunner.unixTimeCmdProfiler.**parseUnixTimeElapsed** (*timeElapsedStr*)

5.4 IO (Input Output functions) API

The IO Module in CREDO is for Input/Output to various forms of data important for analysis and testing of StGermain codes.

These are primarily related to the core data formats that StG-based codes such as Underworld:

- Accept data in to configure and run models, such as the StGermainData XML format.
- Produce data in as a result of running models, such as the FrequentOutput.txt files specifying observables produced on a regular basis during running of a timestepped model.

5.4.1 `credo.io.stgxml`

This module is for accessing, manipulating and writing out XML files stored in the StGermain data format.

It currently does not create a new Object-oriented representation of XML docs themselves, but most functions operate on an ElementTree object representative of an element in an open Stg XML document.

credo.io.stgxml.**addNsPrefix** (*tagName*)

Simple utility func to add the Namespace prefix that LXML adds to element tag names when it parses in from a file

credo.io.stgxml.**createFlattenedXML** (*inputFiles*, *cmdLineOverrides=''*, *flatFilename='output.xml'*)

Flatten a list of provided XML files and optionally also cmdLineOverrides (string), using the StGermain FlattenXML tool.

Returns the file name of the newly created flattened file.

credo.io.stgxml.**createNewStgDataDoc**()
Create a new empty StGermain model XML file (can be merged with other model files).

Returns a tuple of the new xml doc (Element Tree), and the root node of the new doc.

credo.io.stgxml.**getElementType**(*elementNode*)
Checks the "type" of a StGermain data node element. Here, we deal with 3 possibilities:

- The <param>, <list>, <struct> node tag format.
- The <elmement> tag format, where type is an attribute.
- The special elements <import>, <plugins> -> list, <components> -> dict.

credo.io.stgxml.**getItemFromStrSpec_CurrentCtx**(*currCtxNode*, *nodeSpecStr*)
Gets the node at a current context. If context is a struct, treats nodeSpecStr as a named element. If context is a list, nodeSpecStr must be a list specifier.

credo.io.stgxml.**getListNode**(*elNode*, *listName*)
Returns the element node (in etree form) of a particular list element that's a child of the given elNode with given listName. If a node with the given name not found, returns none.

credo.io.stgxml.**getNodeFromStrSpec**(*parentNode*, *strSpec*)
From a given specification of a node in a StGermain model file (eg plugins[0].Context), return the element to operate on.

credo.io.stgxml.**getParamNode**(*elNode*, *paramName*)
Returns the element node (in etree form) of a particular Param parameter that's a child of the given elNode with given paramName. If a node with the given name not found, returns none.

credo.io.stgxml.**getParamValue**(*elNode*, *paramName*, *castFunc*)
Gets the value of a parameter from a StGermain XML model file that's a child of the given elNode, with the given paramName. The value is cast according to the castFunc, which should be a standard Python casting function, e.g. 'int', 'double' or 'bool'.

credo.io.stgxml.**getStructNode**(*elNode*, *structName*)
Returns the element node (in etree form) of a particular struct element that's a child of the given elNode with given structName. If a node with the given name not found, returns none.

credo.io.stgxml.**indentForPrettyPrint**(*elem*, *level=0*, *spacer=' '*)
Indent an XML file in xml.etree format, useful for pretty printing.

Credit to http://infix.se/2007/02/06/gentlemen-indent-your-xml

credo.io.stgxml.**insertNamedElementNode**(*parentNode*, *elementName*, *createType*)

credo.io.stgxml.**navigateStrSpecHierarchy**(*currNode*, *strSpec*, *insertMode=False*)
Navigate a document, based on a remaining StGermain command-line style element specification. Returns currNode, nodeName of the final entry in the hierarchy

credo.io.stgxml.**removeNsPrefix**(*tagName*)
Simple utility func to remove the Namespace prefix that LXML adds to element tag names when it parses in from a file

credo.io.stgxml.**setMergeType**(*xmlNode*, *mergeType*)
Set the "MergeType" of an XML node: usually for writing new nodes in an XML designed to over-ride existing XML of a model.

credo.io.stgxml.**strToBool**(*boolStr*)
Converts a string (eg from a param in XML) to a Python Bool and returns this, using same idiom as in StGermain.

(See Dictionary_Entry_Value_AsBool() in Dictionary_Entry_Value.c in StGermain/Base/IO).

credo.io.stgxml.**writeComponent**(*parentNode*, *compName*, *compType*, *mt=None*)
Write XML to merge a given component to the components list - and return new comp elt

credo.io.stgxml.**writeIncludeLine**(*parantNode*, *includeValue*, *mt=None*)
Write a Stg XML include line for referencing other input xml files

credo.io.stgxml.**writeMergeComponent**(*rootNode*, *compName*, *compType*)
Write XML to merge a given component to the components list - and return new comp elt

credo.io.stgxml.**writeMergeComponentStruct**(*rootNode*)
Write XML to merge a given component to the components list - and return new comp elt

credo.io.stgxml.**writeParam**(*parentNode*, *paramName*, *paramVal*, *mt=None*)
Writes a particular parameter, with name of paramName, val of paramVal, to the open XML file at position specified by parentNode.

credo.io.stgxml.**writeParamList**(*parentNode*, *listName*, *paramVals*, *mt=None*)
Write a Stg XML List structure, made up purely of (unnamed) parameters

credo.io.stgxml.**writeParamSet**(*parentNode*, *paramsDict*, *mt=None*)
Writes a set of parameters, with name:value mappings as specified by paramsDict, to the open XML file at position specified by parentNode.

credo.io.stgxml.**writeStgDataDocToFile**(*xmlDoc*, *filename*)
Write a given StGermain xmlDoc to the file given by filename

credo.io.stgxml.**writeStruct**(*parentNode*, *mt=None*)
Write a Stg XML List structure, made up purely of (unnamed) parameters

credo.io.stgxml.**writeStructList**(*parentNode*, *listName*, *mt=None*)
Write a Stg XML List structure, made up purely of (unnamed) parameters

credo.io.stgxml.**writeValueUsingStrSpec**(*rootNode*, *strSpec*, *value*)

credo.io.stgxml.**writeXMLDoc**(*xmlDoc*, *outFile*, *prettyPrint=True*)
Necessary because the xml.etree doesn't include a pretty_print functionality by default (unlike lxml)

5.4.2 credo.io.stgcmdline

This module is for properly formatting StGermain command line operations.

credo.io.stgcmdline.**paramStr**(*paramPath*, *paramVal*)
Return the command line string to override a certain parameter path and value.

credo.io.stgcmdline.**solverOptsStr**(*optsFilename*)
Return the cmd line string to apply a given PETSc solver options file.

5.4.3 credo.io.stgfreq

credo.io.stgfreq.FreqOutput

A Module for convenient access to StGermain FrequentOutput files.

Primary construct is the FreqOutput class, which once constructed has numerous methods to access data from a FrequentOutput file.

class `credo.io.stgfreq.`**`FreqOutput`**(*path*, *filename='FrequentOutput.dat'*)

A simple class to store information about a frequent output file, and make it accessible. Once passed a filename of a FrequentOutput file in it's constructor, has methods to get information and values from that named file.

The FrequentOutput file can be either "cached" into memory using the `populateFromFile()` function so subsequent access is quick, or else calling an access function directly such as `getRecordDictAtStep()` will automatically populate the data cache for you behind the scenes.

Key attributes:

`populated`

Bool, states whether the class's data structures have been populated from file, shouldn't be modified externally.

`headerColMap`

dict, mapping header names in the FrequentOutput to the column number they occupy in the file.

`finalStep`()

Returns the highest timestep number that has information recorded for it in the associated FrequentOutput file.

`getAllRecords`()

Get all records of timestep information from the associated FrequentOutput file.

Records are stored in an array, where each entry is of the form of a list of floats of the records at that timestep. Thus likely needs to be used in conjunction with other functions to access the data itself by header name, ie the self.headerColMap.

Saves this as self.records, and returns a reference to it. (Also populates the self._finalTimeStep attribute.)

`getClosest`(*headerName*, *targVal*)

Gets the closest value and timestep to the given value.

`getColNum`(*headerName*)

Get the column number in the record data structure/FrequentOutput file itself - associated with a particular header name.

`getComparisonOp`(*headerName*, *cmpFunc*)

Utility function for doing comparison operations on the records list, e.g. the max or minimum - where cmpFunc is a single operator

`getHeaders`()

Read the headers from the associated FrequentOutput file, populate attr:*headerColMap*, and return the names of the headers as a list.

`getMax`(*headerName*)

get the Maximum of the records for a given header, including the timestep at which that minimum occured.

`getMean`(*headerName*)

gets the Mean of the records for a given header.

Note: this is provided for convenience. If user wants to do more complex statistical operations, use the getValuesArray, then process this directly using stats functions/libraries.

`getMin`(*headerName*)

get the Minimum of the records for a given header, including the timestep at which that minimum occured.

getRecordAtStep(*tstep*)
Gets the record (in raw form, see getAllRecords) at a given timestep, and returns.

getRecordDictAtFinalStep()
Utility wrapper function to get a dictionary of records in the FreqOutput at the final timestep - see `getRecordDictAtStep`.

getRecordDictAtStep(*tstep*)
For a given timestep tstep, looks up the records for that timestep and returns a dictionary of "headername":recordVal mappings, where headername is the name of each header in the FrequentOutput file, and recordVal is the value of that property at the requested timestep.

getRecordNum(*tstep*)
Gets the record number in the FrequentOutput file of a given timestep. E.g. in a FrequentOutput file where values were saved every 5 timesteps, then the 15th timestep will map to the 3rd record.

getReductionOp(*headerName*, *reduceFunc*, ***kwargs*)
Utility function for doing comparison operations on the records list, e.g. the max or minimum - where reduceFunc can operate on the whole records list at once, and support the "key" syntax to pick correct field out of tuples for comparison.

Note: This has been written to allow both standard Python 'reduction ops' like *max()* and *min()*, and also more complex operators defined in this module, or by the user.

---

getTimeStepsArray(*range='all'*)
Returns an array of all timestep numbers that have records saved in the associated FrequentOutput file.

Note: the "range" parameter is not yet operational and should be ignored for now.

---

getTimestepMap()
Calculate a map of timestep number of model, to record number in the FrequentOutput file - stores this, and returns a reference to it.

This is important especially if the FrequentOutput has been sampled from the model at a timstep frequency less than 1.

getValueAtStep(*headerName*, *tstep*)
Gets the values of a property given by 'headerName', at a specified timestep 'tstep'.

getValuesArray(*headerName*, *range='all'*)
Returns an array of all values over time for the property defined by "headerName" in the associated FrequentOutput file.

Note: the "range" parameter is not yet operational and should be ignored for now.

---

plotOverTime(*headerName*, *depName='Timestep'*, *show=False*, *save=True*, *path='.'*)
Plot the value of property given by 'headerName', against parameter 'depName', which defaults to 'Timestep'.

Note: Use of this function requires the Python library matplotlib to be installed.

The argument "show" enables whether the graph is to be immediately shown interactively, and "save" if it should be saved. If "save" is true, the "path" parameter determines the path the resulting plot file will be saved under.

populateFromFile()
This function will read all essential data from the FrequentOutput file associated with the class into data structures in memory, for fast subsequent access. Saves the fact that this has occurred so it doesn't neeed to be repeated in future.

printAllMinMax()
Print the maximum and minimum values of all fields in the frequent output.

credo.io.stgfreq.**closestToSimTime**(*inList*, *key*, *stgFreq*, *targTime*)
Utilitiy function for use with attr:~.*FreqOutput.getReductionOp*: Gets the value at a given timestep.

Parameters **target** – the target simulation time.

credo.io.stgfreq.**closestToStep**(*inList*, *key*, *stgFreq*, *targStep*)
Utilitiy function for use with attr:~.*FreqOutput.getReductionOp*: Gets the value at a given timestep.

Parameters **target** – the target timestep.

credo.io.stgfreq.**closestToVal**(*inList*, *key*, *stgFreq*, *targVal*, *targObsName=None*)
Utilitiy function for use with attr:~.*FreqOutput.getReductionOp*: Gets the entry where the value of chosen observable (eg VRMS) is closest to a target value.

Parameters

- **targVal** – the target observable value.
- **targObsName** – the target header to check value at. If *None*, then use the same as for the observable we'll return.

credo.io.stgfreq.**firstOp**(*inList*, *key*, *stgFreq*)
A utility function designed to pass to attr:~.*FreqOutput.getReductionOp* for getting the first value from a frequent output list.

credo.io.stgfreq.**lastOp**(*inList*, *key*, *stgFreq*)
A utility function designed to pass to attr:~.*FreqOutput.getReductionOp* for getting the last value from a frequent output list.

credo.io.stgfreq.**maxOp**(*inList*, *key*, *stgFreq*)

credo.io.stgfreq.**minOp**(*inList*, *key*, *stgFreq*)

5.4.4 credo.io.stgcvg

credo.io.stgcvg.CvgFileInfo

credo.io.stgcvg.CVGReadError

A Module for dealing with StGermain "Convergence" files, i.e. records of comparison between a set of Fields and either reference or Analytic solutions, produced by the FieldTester Component.

The module is not fully object-oriented, and in the current design allows for the possibility of a set of cvg files in one path that should all be referenced. (e.g. the see function `genConvergenceFileIndex()`).

It provides a similar, though not identical, interface to the `credo.io.stgfreq` module.

The format of CVG files is space-separated, of the form:

- Header lines, which may be repeated throughout the file:

```
#Res        FieldName1      FieldName2 ...
```

- Values lines, which are of the form:

```
<len scale> <field value 1> <field value 2>
```

For example:

```
#Res TemperatureField1 TemperatureField2
1.000000e-01 6.16e-03 5.5e-2
#Res TemperatureField1 TemperatureField2
1.000000e-01 6.14e-03 5.4e-2
#Res TemperatureField1 TemperatureField2
1.000000e-01 6.12235812e-03 5.3e-2
```

exception `credo.io.stgcvg.`**`CVGReadError`**

Bases: `exceptions.IOError`

An exception for specifying problems reading an Underworld convergence file.

class `credo.io.stgcvg.`**`CvgFileInfo`**(*filename*)

A simple class to store info about what fields map to what convergence files. Currently implicit is the name of the Field, this is usually handled by storing CvgFileInfos in a dictionary, with the keys being Field names.

`filename`

The filename (as a string) that the cvg info is stored in.

`dofColMap`

A dictionary mapping for each degree of freedom of a field, the column number it is stored in in the cvg file.

`credo.io.stgcvg.`**`genConvergenceFileIndex`**(*path*)

Returns a dictionary relating field names to `CvgFileInfo` classes, after reading all .cvg files in the given path.

`credo.io.stgcvg.`**`getCheckStepsRange`**(*cvgFile*, *steps*)

Checks that "steps" specified is valid for a given cvgFile (Python File), and if so converts it into a list of step numbers within the range specified. into a tuple of start and end values. If steps isn't valid, raises assertion.

"steps" can be of the format:

- The string "all", meaning a list from 0 to the last step number will be returned;
- The string "last", meaning that a list containing only the last step number in the cvgFile will be returned;
- A tuple of two step numbers (integers), in which case a list will be returned containing a range between the two steps (using Python's range() function).

`credo.io.stgcvg.`**`getDofErrorsForStep`**(*cvgFileInfo*, *stepNum*)

For the given `CvgFileInfo` and step number, returns a list indexed by dof number of the error of each dof in that step.

`credo.io.stgcvg.`**`getDofErrors_ByDof`**(*cvgFileInfo*, *steps='all'*)

For a given cvgFileInfo, get the errors in the specified dof from the specified file, indexed primarily by Dof. The 'steps' arg can be either 'all' (for all timesteps), 'last', or a tuple specifying range (see `getCheckStepsRange()` for more). If only one step result is asked for, the dofs are returned as a simple 1D array, otherwise they're returned as a list.

`credo.io.stgcvg.`**`getDofErrors_ByStep`**`(`*cvgFileInfo*, *steps='all'*`)`
For a given `CvgFileInfo`, return the errors in the cvgFileInfo's specified file - for the timestep range specified by *steps*, indexed primarily by Timestep. The *steps* arg can be either 'all' (for all timesteps), 'last', or a tuple specifying range (see `getCheckStepsRange()` for more). If only one step result is asked for, the dofs are returned as a simple 1D array.

`credo.io.stgcvg.`**`getRes`**`(`*cvgFilename*, *steps='all'*`)`
For a given cvg Filename, return the 'resolutions' (length scale) for the given set of steps, where "steps" is of the form documented in `getCheckStepsRange()`.

5.4.5 `credo.io.stgpath`

This module is for accessing and working with StGermain-related paths.

`credo.io.stgpath.`**`checkAllXMLInputFilesExist`**`(`*inputFilesList*`)`
Checks a whole set of XML input files exist, and raises an IOError if one of them doesn't. See `checkXMLInputFileExists()`.

`credo.io.stgpath.`**`checkXMLInputFileExists`**`(`*inputFile*`)`
Check if an XML input file exists in either the standard XML path, or relative to the current directory. Raises an IOError if not.

`credo.io.stgpath.`**`convertLocalXMLFilesToAbsPaths`**`(`*inputFilesList*, *callingPath*`)`
Check through the given input file list, and for any that aren't found relative to either the local directory or the StGermain standard path, convert them to be relative to the given *callingPath*

Returns new list of adjusted XML files

`credo.io.stgpath.`**`getStgBinPath`**`()`
Get the path of StGermain binaries (given by env variable STG_BINDIR).

`credo.io.stgpath.`**`getStgStandardXMLPath`**`()`
Returns the path that StGermain standard XML files are stored in once installed (and is automatically searched within by StGermain when input files are specified on either the command line, or in include statements).

`credo.io.stgpath.`**`getVerifyStgExePath`**`(`*exeName*`)`
For a given executable name (eg "StGermain"), return the full path name of that executable (in the path given by the STG_BINDIR env variable).

`credo.io.stgpath.`**`getVerifyStgMainExecutablePath`**`()`
return the full path to the main StGermain executable (in the path given by the STG_BINDIR env variable)

`credo.io.stgpath.`**`moveAllToTargetPath`**`(`*startPath*, *targetPath*, *fileExt*`)`
Move all files with extension *fileExt* from *startPath* to *targetPath* - automatically over-writing any existing files with identical names.

`credo.io.stgpath.`**`xmlExistsInStdXMLPath`**`(`*inputFile*`)`

5.5 Analysis API

The CREDO Analysis module is to group together tools for analysis of the results of Underworld models: for use in both individual user scripts, and as part of system testing (see `credo.systest`).

For analysis operations that are designed to be added to `ModelRun` instances, and possibly used as part of system testing, these should inherit from `AnalysisOperation`. There are also several functions that can be used to access and post-process Underworld results in a more direct manner, using the `credo.io` interface.

For examples of using the analysis operations with models, see the *Doing Model analysis with CREDO* section of the documentation.

5.5.1 `credo.analysis.api`

credo.analysis.api.AnalysisOperation

This is the core interface for analysis operations in CREDO.

class `credo.analysis.api.`**`AnalysisOperation`**

Abstract base class for Analysis Operations in CREDO: i.e. that require some analysis to be done during a `ModelRun`. All instances should provide at least this standard interface so that records of analysis can be stored.

`postRun` (*modelRun*, *runPath*)

Does any required post-run actions for this analysis op, e.g. moving generated files into the correct output directory. Is passed a reference to a `ModelRun`, so can use it's attributes such as `outputPath`.

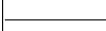
`writeInfoXML` (*parentNode*)

Virtual method for writing Information XML about an analysis op, that will be saved as a record of the analysis applied.

`writeStgDataXML` (*rootNode*)

Writes the necessary StGermain XML to require the analysis to take place. This is likely to include creating and configuring Components, and adding them to the Components dictionary. See `credo.io.stgxml` for the interface for setting these up.

5.5.2 `credo.analysis.fields`



CREDO functions and classes for doing analysis of Fields in StGermain-based codes.

Currently this is primarily based on the FieldTest component/plugin within StgFEM, which allows comparison between one or more Fields in a model run (eg "VelocityField"), and either saved reference fields, or analytic solutions.

The CREDO Field functions allow either single-model comparisons, e.g. those defined by `FieldComparisonOp`, or analysis on fields to be performed across multiple runs, e.g. `calcFieldCvgWithScale()`.

Note: In future it's planned to add functions that load a checkpointed field result into Python for further analysis, but this feature is not yet implemented.

---

class `credo.analysis.fields.`**`FieldComparisonList`**(*fieldsList=None*)

Bases: `credo.analysis.api.AnalysisOperation`

Class for maintaining and managing a list of field comparisons (managed as a list of `FieldComparisonOp` objects), including IO from StGermain XML files.

Currently maps to the "FieldTest" component's functionality in StgFEM.

Note: Currently the whole _list_ of Field comparisons is a single `credo.analysis.api.AnalysisOperation`, because this is the design of the FieldTest component in StgFEM. In future we may look at modularising this functionality further so that single comparisons can be managed as operators.

`fields`

A dictionary mapping field names that need to be compared, to `FieldComparisonOp` to perform the comparison.

`fromXML`

If True, means the list of fields to compare (ie `fields`) should be read from the Model XML files of the model to attach to. If False, the user has to manually specify the fields to compare.

`useReference`

Determines whether fields are compared against a reference solution (if True), or analytic (if False). If useReference is true, user must also specify `referencePath` so that the appropriate StGermain XML for the operation can be written.

`useHighResReference`

Determines whether fields are compared against a high-res reference solution (if True), or analytic (if False). If useHighResReference is true, user must also specify `referencePath` so that the appropriate StGermain XML for the operation can be written.

Note: Don't also specify `useReference`, choose one or the other.

`referencePath`

(Relative or absolute) path to the reference solutions for the specified fields.

`testTimestep`

Integer, the timestep of the model that the comparison will occur at. If 0, means the final timestep. Based on the capability of the StGermain FieldTest component.

`add`(*fieldComparisonOp*)

Add another `FieldComparisonOp` to the list to compare.

`checkStgXMLResultsEnabled`(*inputFilesList, basePath*)

Checks that the field comparison has the writing of comparison info to file enabled (returning Bool).

`getAllResults`(*modelResult*)

Return a list of `FieldComparisonResult` based on all the `FieldComparisonOps` specified to be done during a run, from the given modelResult (`ModelResult`).

getCmpSrcString()
Returns an appropriate string to document the comparison source of the fields being compared - i.e. either reference or analytic.

postRun(*modelRun*, *runPath*)
Implements `AnalysisOperation.postRun()`. In this case, moves all CVG files created to output path.

readFromStgXML(*inputFilesList*, *basePath*)
Read in the list of fields that have already been specified to be tested from a set of StGermain input files. Useful when e.g. working with an Analytic Solution plugin.

writeInfoXML(*parentNode*)
Writes information about this class into an existing, open XML doc node, in a child element.

writeStgDataXML(*rootNode*)
Writes the necessary StGermain XML to enable these specified fields to be compared.

class `credo.analysis.fields.`**FieldComparisonOp**(*fieldName*)
Class for setting up and performing a comparison between two Fields. Currently uses the functionality of the FieldTest component in StgFEM, and requires using a `FieldComparisonList` to run a group of FieldComparisons at once (this is as a result of the structure of the FieldTest component).

name
name of the field that is being compared (to an analytic or ref soln).

getResult(*modelResult*)
Gets the result of the operator on the given fields (as a `FieldComparisonResult`), given a modelResult (`ModelResult`) which refers to a directory containing field comparisons (i.e. cvg files, see `credo.io.stgcvg`).

writeInfoXML(*parentNode*)
Writes info about a comparison op: currently assumes will be called by `FieldComparisonList.writeInfoXML()`.

class `credo.analysis.fields.`**FieldComparisonResult**(*fieldName*, *dofErrors*)
Simple class for storing CREDO FieldComparisonOp Results, so they can be analysed and saved.

By default only contains the difference between the field DOFs at the final timestep - but recording a reference to the `credo.io.stgcvg.CvgFileInfo` for this field allows more complex analysis.

fieldName
Name of the field that has been compared.

dofErrors
Comparison errors for each DOF of the field, at the final timestep that was run.

cvgFileInfo
A `credo.io.stgcvg.CvgFileInfo` allowing detailed access to the CVG result for this field. Required for plotting etc. Is optional, needs to be recorded after the class has been constructed.

plottedCvgFilename
If the `plotOverTime()` method has been called, this attribute will record the filename the plot was saved to.

plotOverTime(*save=True*, *show=False*, *dofIndex=None*, *path='.'*)
Plot the result of a FieldComparison over all timesteps of a model. Requires the cvgFileInfo paramater to have been set to give access to the cvg info of this field.

Note: Requires you to have the Matplotlib library installed.

'show', 'save' and 'path' parameters are the same as for `credo.io.stgfreq.FreqOutput.plotOverTime()`. The optional 'dofIndex' parameter allows you to only plot a particular DOF of the field, otherwise all dofs will be plotted on separate graphs.

withinTol (*tol*)
Checks that the difference between the fields is within a given tolerance, at the final timestep.

writeInfoXML (*fieldResultsNode*)
Writes information about a FieldComparisonResult into an existing, open XML doc node

credo.analysis.fields.**calcFieldCvgWithScale** (*fieldName*, *lenScales*, *dofErrors*)
Gets the convergence and correlation of a field with resolution (taking the log10 of both).

lenScales argument should simply be an array of length scales for the different runs. dofErrors must be a list, for each dof of the field, of the error vs some expected solution at the corresponding length scale.

returns a list of tuples, one per dof, where each tuple contains: (convergence rate, pearson correlation) over the set of scales.

credo.analysis.fields.**getFieldScaleCvgData_SingleCvgFile** (*cvgFilePath*)
Given a path that CVG files reside in, returns the length scales of each run (as a list), and a list of field error data for each field/cvg info in the given path. Thus is a utility function for generating necessary fieldErrorData for a multi-res convergence analysis.

Note: This assumes all cvg info is stored in the same convergence file (the default approach of the legacy SYS tests)

5.5.3 credo.analysis.stats

A library of useful stats functions for analysis operations.

The aim is for simple functions to be able to run without further dependencies ... with more advanced stats libraries from the likes of SciPy being able to be loaded at the user's discretion.

credo.analysis.stats.**linreg** (*X*, *Y*)

Summary Linear regression of y = ax + b

Usage real, real, real = linreg(list, list)

Returns coefficients to the regression line "y=ax+b" from x[] and y[], and R^2 Value

(Obtained from From http://www.answermysearches.com/how-to-do-a-simple-linear-regression-in-python/124/) Useful for field analysis, e.g. when applied to a list of length scales & field errors, to calculate convergence info, such as `credo.analysis.fields.calcFieldCvgWithScale()`.

5.5.4 credo.analysis.images

Utilities for basic image analysis and testing in relation to CREDO. Original image test comparison scripts contributed by Owen Kaluza. You will need the Python Imaging Library (PIL) installed to use.

credo.analysis.images.**channelDiff** (*channel*, *hist1*, *hist2*, *pixels*, *bins*)
Calculates histogram difference in one colour channel

credo.analysis.images.**colourDiff** (*img1*, *img2*)
Calculate image difference by colour histogram.

Parameters

- **img1** – open PIL image
- **img2** – open PIL image

Returns float representing difference between images by colour histogram

`credo.analysis.images.`**`compare`**(*imgFilename1*, *imgFilename2*, *verbose=False*)
Compare two image files.

Returns A tuple containing the diffs for each component (colour space, 400 pixel subsample).

`credo.analysis.images.`**`luminanceDiff`**(*img1*, *img2*)
Calculate image difference by luminance histogram.

Parameters

- **img1** – open PIL image
- **img2** – open PIL image

Returns float representing difference between images in luminance histogram space

`credo.analysis.images.`**`normalise`**(*array*, *maxval*)

`credo.analysis.images.`**`pixelDiff`**(*img1*, *img2*)
Compare two open images on a pixel by pixel basis.

`credo.analysis.images.`**`pixelDiff20x20`**(*img1*, *img2*)
Compare two open images on a 20x20 basis.

5.5.5 `credo.analysis.modelplots`

Collection of utility functions for plotting interesting aspects of models

`credo.analysis.modelplots.`**`getNumEls`**(*mRun*)
Calculate the number of elements used by a model run.

`credo.analysis.modelplots.`**`getSpeedups`**(*mRuns*, *mResults*, *profilerName=None*)

`credo.analysis.modelplots.`**`getTimePerEls`**(*mRuns*, *mResults*, *profilerName=None*)

`credo.analysis.modelplots.`**`getValsFromAllRuns`**(*mResults*, *outputName*)

`credo.analysis.modelplots.`**`plotOverAllRuns`**(*mResults*, *outputName*, *depName='Timestep'*, *show=False*, *save=True*, *path='.'*, *labelNames=True*)
Create a plot of values over multiple runs.

`credo.analysis.modelplots.`**`plotSpeedups`**(*mRuns*, *mResults*, *profilerName=None*, *show=False*, *save=True*, *path='.'*, *showIdeal=True*)
Plot the speedup of a set of mResults, by processor.

`credo.analysis.modelplots.`**`plotTimePerEls`**(*mRuns*, *mResults*, *profilerName=None*, *show=False*, *save=True*, *path='.'*, *showIdeal=True*)
Plot the speedup of a set of mResults, by processor.

`credo.analysis.modelplots.`**`plotWalltimesByRuns`**(*mRuns*, *mResults*, *profilerName=None*, *show=False*, *save=True*, *path='.'*, *showIdeal=True*)

5.6 SysTest API

This module contains the System Testing functionality of CREDO.

Working at a higher-level than the `credo.modelrun` and `credo.analysis` modules, it is able to use their capabilities to run system tests of scientific applications, and communicate and record the results.

From a user perspective, doing an:

```
from credo.systest import *
```

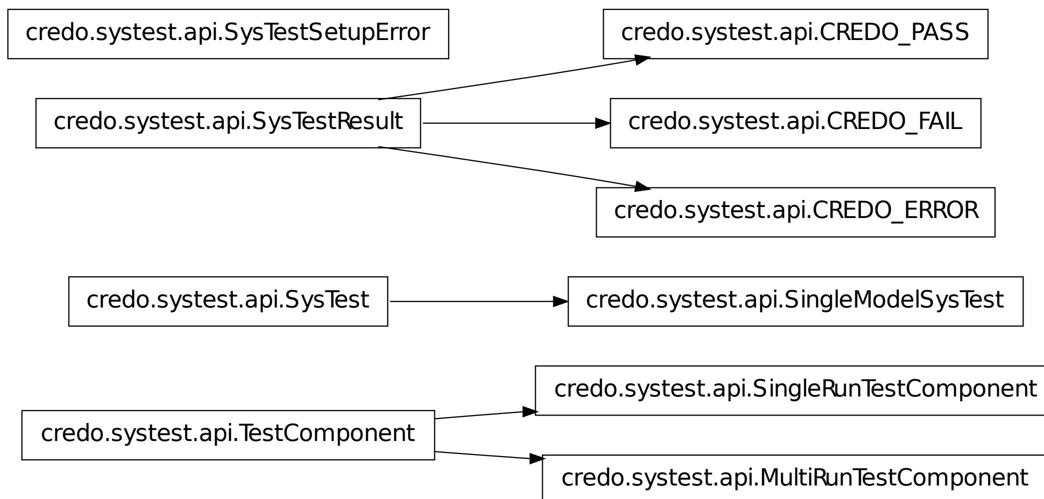
Should allow key functionality to be accessed: e.g. the `credo.systest.systestrunner.SysTestRunner` class, and standard system tests such as Analytic, Restart and Reference.

Examples of how to use this module are provided in the CREDO documentation, see *Using CREDO for System Testing of StGermain-based codes such as Underworld*.

Note: in early CREDO versions, importing credo.systest would have imported into it the entire namespace of the credo.systest.api module. But Python 2.5 disallows this for relative imports (although the feature was added back in in Python 2.6) ... so the import of the credo.systest.api namespace into credo.systest was removed, and we only import the 'public interface' of implementations designed to be used from the API in testing.

As a result, if you want to define new `api.SysTest` or `api.TestComponent` implementations, you should *import credo.systest.api* to access the base classes of these.

5.6.1 `credo.systest.api`



Core API of the `credo.systest` module.

This defines the two key classes of the model, `SysTest` and `TestComponent`, from which actual examples of System tests or Test components need to inherit.

class `credo.systest.api.`**`CREDO_ERROR`**(*errorMsg*)
Bases: `credo.systest.api.SysTestResult`

Simple class to represent an CREDO error

class `credo.systest.api.`**`CREDO_FAIL`**(*failMsg*)
Bases: `credo.systest.api.SysTestResult`

Simple class to represent an CREDO failure

class `credo.systest.api.`**`CREDO_PASS`**(*passMsg*)
Bases: `credo.systest.api.SysTestResult`

Simple class to represent an CREDO pass

class `credo.systest.api.`**`MultiRunTestComponent`**(*tcType*)
Bases: `credo.systest.api.TestComponent`

A type of component designed to operate and report on multiple modelRuns (e.g., analysing they converge or overall meet some requirement.

Unlike the SingleRunTestComponent, this class's `attachOps()` and `check()` methods operate on a list of modelRuns and modelResults, not just a single one.

`attachOps`(*modelRuns*)
Provided a list of modelRuns (`credo.modelrun.ModelRun`) attaches any necessary analysis operations to each run needed for the test. (see `credo.modelrun.ModelRun.analysis`).

`check`(*mResults*)
A function to check a set of results - returns True if the Test passes, False if not. Also updates the `tcStatus` attribute.

class `credo.systest.api.`**`SingleModelSysTest`**(*testType, inputFiles, outputPathBase, basePath=None, nproc=1, timeout=None, paramOverrides=None, solverOpts=None, nameSuffix=None*)
Bases: `credo.systest.api.SysTest`

A subclass of `SysTest` for common system test types that are based on variations on a single Model (defined by a group of XML model files, see `inputFiles`. Includes utility functions for easily creating new model runs based on these standard parameters.

Constructor keywords not in member attribute list:

- nameSuffix: if specified, this defines the suffix that will be added to the test's name, output path, and log path (where the test's result and stderr/out will be saved respectively) - Overriding the default one based on params used.

`inputFiles`
StGermain XML files that define the Model to be tested.

`paramOverrides`
Any model parameter overrides to be passed to ModelRuns performed as part of running the test - see `credo.modelrun.ModelRun.paramOverrides`. Thus allow customisation of the test properties.

`solverOpts`
Solver options to be used for any models making up this test. See `credo.modelrun.ModelRun.solverOpts`

`updateOutputPaths`(*newOutputPathBase*)
Function useful for when modifying suites, and you wish to change the output Path the suite reports are saved in. Necessary because suites with multiple runs will use different sub-dirs for each run.

Note: in current design, _don't_ also update expected/reference soln paths, just output-related paths.

class credo.systest.api.**SingleRunTestComponent**(*tcType*)

Bases: credo.systest.api.TestComponent

attachOps(*modelRun*)

Provided a modelRun (credo.modelrun.ModelRun) attaches any necessary analysis operations to that run in order to produce the results needed for the test. (see credo.modelrun.ModelRun.analysis).

check(*mResult*)

A function to check a set of results - returns True if the Test passes, False if not. Also updates the tcStatus attribute.

class credo.systest.api.**SysTest**(*testType*, *testName*, *basePath*, *outputPathBase*, *nproc=1*, *timeout=None*)

A class for managing SysTests in CREDO. This is an abstract base class: you must sub-class it to create actual system test types.

The SysTest is designed to interact with the SysTestRunner class, primarily by creating a ModelSuite on demand to run a set of ModelRuns required to do the test. It will then do a check that the results pass an expected metric:- generally by applying one or more TestComponent classes.

testType

records the "type" of the system test, as a string (e.g. "Analytic", or "SciBenchmark") - for printing purposes.

testName

The name of this test, generally auto-generated from other properties.

basePath

The base path of the System test, that runs will be done relative to.

outputPathBase

The "base" output path to store results of the test in. Results from individual Model Runs performed as part of running the test may also be stored in that directory, or in subdirectories of it.

mSuite

The suite of Models that will be run as part of the test. Initially None, must be filled in as part of calling genSuite.

nproc

Number of processors to be used for the test. See credo.modelrun.ModelRun.nproc.

timeout

if set to a positive integer, this will be used as a maximum time (in seconds) the test is allowed to run for - if it runs over this the result of the test will be set to an Error. If timeout is None, 0 or negative, no timeout will be applied.

testStatus

Status of the test. Initially *None*, once the test has been run the SysTestResult generated will be saved here.

testComps

A list of dictionaries of TestComponent (Single run) classes used as part of performing this system test. The primary list is indexed by run number of the model run in the systest's mSuite.

multiRunTestComps

A dictionaries of MultiRunTestComponent classes used as part of performing this system test.

generatedReports
A list of the file-names of all generated reports created based on this benchmark.

attachAllTestCompOps()
Useful in `configureTestComps()`. but default is to call 'attachOps' method of all testComps (requires all testComps to have been already set up and declared in `configureTestComps()`)

checkModelResultsValid(*resultsSet*)
Check that the given result set is "valid", i.e. exists, has the right number of model results, and model results have necessary analysis ops associated with them to allow aspects of test to evaluate properly.

configureSuite()
Function for configuring the `credo.modelsuite.ModelSuite` to be used for testing on. Must be saved to `mSuite` by the end of this function. By Default, calls method `genSuite()` which the user should override.

configureTestComps()

createReports(*mResults*)
Create any custom reports, then update record XML

defaultSysTestFilename()
Return the default system test XML record filename, based on properties of the systest (such as `testName`).

genSuite()
Must update the `mSuite` attribute so it contains all models that need to be run to perform the test.

getStatus(*resultsSet*)
After a suite of runs created by :meth:".genSuite" has been run, when this method is passed the results of the suite (as a list of `credo.modelresult.ModelResult`), it must decide and return the status of the test (as a `SysTestResult`).

It also needs to save this status to `testStatus`.

By default, this simply gets each `TestComponent` registered for the system test do check its status, all must pass for a total pass.

Note: if using this default method, then sub-classes need to have defined *failMsg* and *passMsg* attributes to use.

---

getTCRes(*tcName*, *allowMissing=True*)
Utility function for single run test components to get lists of test components, and tc results, for each run of a given testComp (This can be done by list manipulation in Python, this function just makes it easier).

Parameters allowMissing – if True, runs that don't have a TC of givne name applied to them will have *None* placed in output array. if False, KeyError exception will be propagated.

Returns a tuple of 2 lists: all test components of a given name ordered by model run in the test, and a list of corresponding test component results.

regenerateFixture(*jobRunner*)
Function to do any setup of tests for the first time they're run, or e.g. when updating to a new Underworld version. Since not all tests need this functionality, the default behaviour is to do nothing.

runTest(*jobRunner*, *postProcFromExisting=False*, *createReports=True*)
Run this sysTest, and return the `SysTestResult` it produces. Will also write an XML record of the System test, and each ModelRun and ModelResult in the suite that made up the test.

Parameters

- **postProcFromExisting** – if True, will not actually run the test, but will read the result from existing modelResults.
- **createReports** – if True, will create external reports (additional to the XML record) this test specifies.

Returns SysTestResult, and list of ModelResults (since latter may be useful for further post-processing)

setCustomReporting(*customReportingFunc*)
Method to use to set the value of the customReporting method.

setErrorStatus(*errorMsg*)
Utility function for if a model run fails as part of the test, this function can be called to automatically set the test status.

setTimeout(*seconds=0, minutes=0, hours=0, days=0*)
Sets the `timeout` parameter, used to determine how long the test is allowed to run for.

setupEmptyTestCompsList()

setupTest()

updateXMLWithReports(*outputPath='', filename='', prettyPrint=True*)
Updates a Sys Test XML with record of any report files generated as a result of the test. If the XML file has the standard name, as defined by `defaultSysTestFilename()`, then it should be found automatically.

Other arguments and return value same as for `writePreRunXML()`.

updateXMLWithResult(*resultsSet, outputPath='', filename='', prettyPrint=True*)
Given resultsSet, a set of model results (list of `ModelResult`), updates a Sys Test XML with the results of the test. If the XML file has the standard name, as defined by `defaultSysTestFilename()`, then it should be found automatically.

Other arguments and return value same as for `writePreRunXML()`.

writePreRunXML(*outputPath='', filename='', prettyPrint=True*)
Write the SysTest XML with as much information before the run as is possible. This includes general info about the test, and detailed specificiation of appropriate parameters and test components.

Parameters

- **outputPath** – (opt) path the XML should be saved to.
- **filename** – (opt) filename within that path that should be used.
- **prettyPrint** – whether to indent the XML for better human-readability (pretty-printing).

Returns the name of the file written to.

writeRecordXML(*mResults, outputPath='', filename='', prettyPrint=True*)
Convenience function to call all other XML writing funcs (pre-run and post-run) in one go.

class credo.systest.api.**SysTestResult**
Class to represent an CREDO system test result.

detailMsg
detailed message of what happened during the test.

statusStr
One-word status string summarising test result (eg 'Pass').

_absRecordFile
The absolute path of where the system test was saved to.

getRecordFile()

printDetailMsg()

setRecordFile(*recordFile*)
Save the record file: as an absolute path.

exception credo.systest.api.**SysTestSetupError**
Bases: exceptions.Exception

An exception for when a System test fails to set up correctly.

class credo.systest.api.**TestComponent**(*tcType*)
A class for TestComponents that make up an CREDO System test/benchmark. Generally they will form part a list contained by a SysTest.

This is an abstract base class, individual test components must subclass from this interface.

tcStatus
Status of this test component. Initially None, will be updated after the test component is evaluated to a SysTestResult.

tcType
Type of the test component, as a (single world descriptive) string.

updateXMLWithResult(*tcNode*, *resultInfo*)
Updates a given XML node with the result of the test component. the 'resultInfo' will be passed through to sub-functions, and varies according to the type of test being performed.

writePreRunXML(*parentNode*, *name*)
Function to write out info about the test component to an XML file, as a sub-tree of parentNode.

credo.systest.api.**getStdOutputPath**(*testClass*, *inputFiles*, *testOpts*)
Get the standard name for the test's output path. Attempts to avoid naming collisions where reasonable.

credo.systest.api.**getStdTestName**(*testTypeStr*, *inputFiles*, *nproc*, *paramOverrides*, *solverOpts*, *nameSuffix*)
Utility function, to get a standard name for system tests given key parameters of the tests. If nameSuffix is given a string, it will be used as the suffix after processor number, instead of one based on any parameter over-rides used.

credo.systest.api.**getStdTestNameBasic**(*testTypeStr*, *inputFiles*)
Basic part of the test name. Useful for restart runs etc.

5.6.2 credo.systest.systestrunner

credo.systest.systestrunner.SysTestRunner

Package for manipulation of a suite of system tests. Analogous to the role of the Pythun unittest TestRunner.

class credo.systest.systestrunner.**SysTestRunner**
Class that runs a set of SysTest, usually collected into SysTestSuite collections.

For examples of how to use, see the CREDO documentation, especially *Running CREDO system test suites directly, and how to modify them*.

getResultsTotals (*results*)
Gets the totals of a set of results, and returns, including indices of which results failed, and which were errors.

getSuiteResultsFilename (*suite*)
Get a standard name for a suite record file, from given suite and its attributes.

printResultsDetails (*sysTests*, *results*)
Prints details of which tests failed in a sub-suite.

printResultsSummary (*sysTests*, *results*, *projName=None*, *suiteName=None*)
Print a textual summary of the results of running a set of sys tests.

printSuiteResultsByProject (*testSuites*, *resultsLists*)
Utility function to print a set of suite results out, categorised by project, in the order that the projects first appear in the results.

printSuiteResultsOrderFound (*testSuites*, *resultsLists*)
Utility function to print a set of results in the order they were entered (not sub-categorised by project).

printSuiteTotalsShortSummary (*results*, *projName*, *suiteName*)
Prints a short summary, useful for suites with sub-suites.

runSingleTest (*sysTest*, ***kwargs*)
Convenience function to setup and run a single SysTest. .. note:: all keywords appopriate to

`credo.systest.api.SysTest.runTest()` are passed through directly in the kwargs parameter.

runSuite (*suite*, *runSubSuites=True*, *subSuiteMode=False*, *outputSummaryDir='testLogs'*, ***kwargs*)
Runs a suite of system tests, and prints results. The suite may contain sub-suites, which will also be run by default.

Returns a list of all results of the suite, and its sub-suites

Note: Currently, just returns a flat list of results, containing results of all tests and all sub-suites. Won't change this into a hierarchy of results by sub-suite, unless really necessary.

Note: all keywords appopriate to `credo.systest.api.SysTest.runTest()` are passed through directly in the kwargs parameter.

runSuites (*testSuites*, *outputSummaryDir='testLogs'*, ***kwargs*)
Runs a list of suites, and prints a big summary at the end.

Parameters

- **testSuites** – list of test suites to run.
- **outputSummaryDir** – name of directory to save a summary of tests to.

Returns a list containing lists of results for each suite (results list in the same order as testSuites input argument).

Note: all keywords appopriate to `credo.systest.api.SysTest.runTest()` are passed through directly in the kwargs parameter.

runTests (*sysTests*, *projName=None*, *suiteName=None*, *printSummary=True*, ***kwargs*)
Run all tests in the sysTests list. Will also save all appropriate XMLs and print a summary of results.

Parameters

- **projName** – the name of the 'project' to report these tests as belonging to.
- **suiteName** – the name of the suite these tests should be reported as belonging to.

Note: all keywords appopriate to `credo.systest.api.SysTest.runTest()` are passed through directly in the kwargs parameter.

`credo.systest.systestrunner.`**getSuitesFromModules** (*suiteModNames*)
Gets a list of suites from the list of suites to import given in suiteModNames.

`credo.systest.systestrunner.`**runSuitesFromModules** (*suiteModNames*, ***kwargs*)
Runs a set of System test suites, where suiteModNames is a list of suites to import and run.

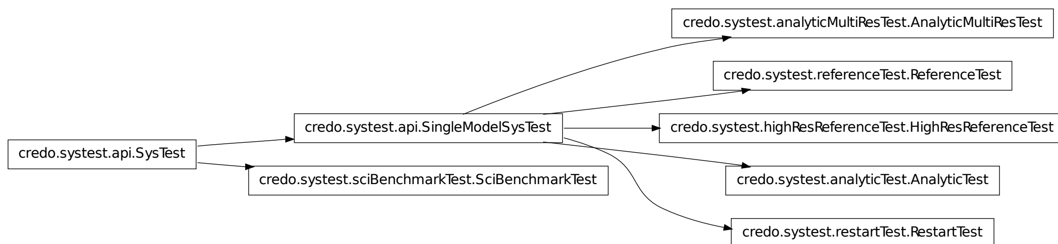
Note: all keywords appopriate to `credo.systest.api.SysTest.runTest()` are passed through directly in the kwargs parameter.

5.6.3 Core System Test class implementations

CREDO provides a set of core `SysTest` instantations, which supercede the functionality of the pre-existing test scripts system, which are documented below.

The user can always add to this list, by defining new SysTest classes to use.

The most flexible of the set is the `SciBenchmarkTest`, but this requires the most customisation (i.e. generally can't be created in the short-hand form of the other tests using the sysTestRunner's `addStdTest()` method).



depending on whether you have the Python Imaging Library (PIL) installed, you can also use the `credo.systest.imageReference.ImageReferenceTest` system test.

credo.systest.analyticTest

class `credo.systest.analyticTest.`**`AnalyticTest`**(*inputFiles*, *outputPathBase*, *basePath=None*, *nproc=1*, *timeout=None*, *paramOverrides=None*, *solverOpts=None*, *nameSuffix=None*, *defFieldTol=0.029999999999999999*, *fieldTols=None*)

Bases: `credo.systest.api.SingleModelSysTest`

An Analytic System test. This case requires the user to have configured the XML correctly to load an anlytic soln, and compare it to the correct fields. Will check that each field flagged to be analysed is within the expected tolerance. Uses a `FieldWithinTolTC` test component to perform the check.

Optional constructor keywords:

- defFieldTol: The default tolerance to be applied when comparing fields of interest to the analytic solution. See also the FieldWithinTolTC's `defFieldTol`.
- fieldTols: a dictionary of tolerances to use when testing particular fields, rather than the default tolerance defined by the defFieldTol argument.

`fTestName`

Standard name to use for this test's field comparison TestComponent in the `testComponents` list.

`checkModelResultsValid`(*resultsSet*)

See base class `checkModelResultsValid()`.

`configureTestComps`()

`genSuite`()

See base class `genSuite()`.

For this test, just a single model run is needed, to run the model and compare against the analytic solution.

credo.systest.referenceTest

Provides a `ReferenceTest` for use in system testing.

`credo.systest.referenceTest.`**`DEF_TEST_FIELDS`**

Default fields that will be tested, if not explicitly provided as a constructor keyword argument to `ReferenceTest` instantiations.

class `credo.systest.referenceTest.`**`ReferenceTest`**(*inputFiles*, *outputPathBase*, *basePath=None*, *nproc=1*, *timeout=None*, *paramOverrides=None*, *solverOpts=None*, *nameSuffix=None*, *fieldsToTest=None*, *runSteps=20*, *defFieldTol=0.01*, *fieldTols=None*, *expPathPrefix='expected'*)

A Reference System test. This case simply runs a given model for a set number of steps, then checks the resultant solution matches within a tolerance of a previously-generated reference solution. Uses a `FieldWithinTolTC` test component to perform the check.

Optional constructor keywords:

- runSteps: number of steps the reference solution should run for.
- fieldsToTest: Which fields in the model should be compared with the reference solution, as a list. If not provided, will default to `DEF_TEST_FIELDS`.

- defFieldTol: The default tolerance to be applied when comparing fields of interest to the reference solution. See also the FieldWithinTolTC's `defFieldTol`.
- fieldTols: a dictionary of tolerances to use when testing particular fields, rather than the default tolerance as set in the defFieldTol argument.

fTestName
Standard name to use for this test's field comparison TestComponent in the `testComponents` list.

checkModelResultsValid(*resultsSet*)
See base class `checkModelResultsValid()`.

configureTestComps()

genSuite()
See base class `genSuite()`. For this test, just a single model run is needed, to run the model and compare against the reference solution.

regenerateFixture(*jobRunner*)
Do a run to create the reference solution to use.

Note: by default, this will save checkpoint for the entire step, not just fields to be checkpointed against.

credo.systest.restartTest

class credo.systest.restartTest.**RestartTest**(*inputFiles, outputPathBase, basePath=None, nproc=1, timeout=None, paramOverrides=None, solverOpts=None, nameSuffix=None, fieldsToTest=['VelocityField', 'PressureField'], fullRunSteps=20, defFieldTol=1.0000000000000001e-05, fieldTols=None*)

A Restart System test. This case simply runs a given model for set number of steps, then restarts half-way through, and checks the same result is obtained. (Thus it's largely a regression test to ensure checkpoint-restarting works for various types of models). Uses a `FieldWithinTolTC` test component to perform the check.

Optional constructor keywords:

- fullRunSteps: number of steps to do the initial "full" run for. Must be a multiple of 2, so it can be restarted half-way through.
- fieldsToTest: Which fields in the model should be compared with the reference solution.
- defFieldTol: The default tolerance to be applied when comparing fields of interest between the restarted, and original solution. See also the FieldWithinTolTC's `defFieldTol`.
- fieldTols: a dictionary of tolerances to use when testing particular fields, rather than the default tolerance defined by the defFieldTol argument.

fTestName
Standard name to use for this test's field comparison TestComponent in the `testComponents` list.

checkModelResultsValid(*resultsSet*)
See base class `checkModelResultsValid()`.

configureTestComps()

genSuite()
See base class `genSuite()`.

For this test, will create a suite containing 2 model runs: one to initally run the requested Model and save the results, and a 2nd to restart mid-way through, so that the results can be compared at the end.

updateOutputPaths(*newOutputPathBase*)
See base class `updateOutputPaths()`.

credo.systest.analyticMultiResTest

class `credo.systest.analyticMultiResTest.`**AnalyticMultiResTest**(*inputFiles, outputPathBase, resSet, basePath=None, nproc=1, timeout=None, paramOverrides=None, solverOpts=None, nameSuffix=None*)

Bases: `credo.systest.api.SingleModelSysTest`

A Multiple Resolution system test. This test can be used to convert any existing system test that analyses fields, to check that the error between the analytic solution fields and the actual results improves at the required rate as the model resolution is increased. Uses a `FieldCvgWithScaleTC` test component to perform the check.

Optional constructor keywords:

- resSet: a list of resolutions to use for the test, as tuples. E.g. to specify testing at 10x10 res then 20x20, resSet would be [(10,10), (20,20)]

resSet
Set of resolutions to use, as described for the resSet keyword to the constructor.

checkModelResultsValid(*resultsSet*)
See base class `checkModelResultsValid()`.

configureTestComps()

genSuite()
See base class `genSuite()`.

The generated suite will contain model runs all with the same model XML files, but with increasing resolution as specified by the `resSet` attribute.

credo.systest.imageReferenceTest

class `credo.systest.imageReferenceTest.`**ImageReferenceTest**(*inputFiles, outputPathBase, imagesToTest, basePath=None, nproc=1, timeout=None, paramOverrides=None, solverOpts=None, nameSuffix=None, imageTols=None, expPathPrefix='expected', runSteps=20, compareEvery=1, defImageTol=(0.10000000000000001, 0.050000000000000003)*)

Bases: `credo.systest.api.SingleModelSysTest`

An image comparison against Reference System test. To do this, creates a set of several ImageCompTC Test Components for each image you wish to test.

Optional contructor keywords:

- runSteps: Number of steps the model should be run for.
- **compareEvery: Number of steps the model should be run for before** comparing images.
- **defImageTol: Default tolerance to use when comparing images (as a tuple** as required by credo.analysis.images.compare()).
- **imageTols: If provided, must be a dictionary mapping image names to** tolerances to use when checking.
- expPathPrefix: Directory to look for expected image file runs.

checkModelResultsValid(*resultsSet*)
See base class checkModelResultsValid().

configureTestComps()

genSuite()
See base class genSuite().

For this test, just a single model run is needed, to run the model and compare against the reference solution.

regenerateFixture(*jobRunner*)
Do a run to create the reference images to use.

5.6.4 credo.systest.sciBenchmarkTest

class credo.systest.sciBenchmarkTest.**SciBenchmarkTest**(*testName*, *outputPathBase=None*, *basePath=None*, *nproc=1*, *timeout=None*)

Bases: credo.systest.api.SysTest

A Science benchmark test. This is an open-ended system test, designed for the user to add multiple TestComponent s to, which test the conditions of the benchmark. Contains extra capabilities to report more fully on the test result than a standard system test.

See the examples section of the CREDO documentation, *Scientific Benchmarking using CREDO*, for examples of sci benchmarking in practice.

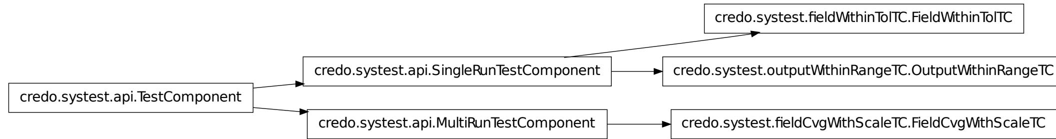
addTestComp(*runI*, *testCompName*, *testComp*)
Add a testComponent (TestComponent) with name testCompName to the list of test components to be applied as part of determining if the benchmark has passed. Does basic error-checking.

configureSuite()

configureTestComps()

setupTest()
Overriding default SysTest.setupTest() method, as for SciBenchmarks we want to allow the user to manage test setup explicitly in their benchmark script. Thus assume suite runs and test components have been setup correctly already.

5.6.5 Core TestComponent implementations



credo.systest.fieldWithinTolTC

class `credo.systest.fieldWithinTolTC.`**`FieldWithinTolTC`**(*fieldsToTest=None*, *defFieldTol=0.01*, *fieldTols=None*, *useReference=False*, *useHighResReference=False*, *referencePath=None*, *testTimestep=0*)

Bases: `credo.systest.api.SingleRunTestComponent`

Checks whether, for a particular set of fields, the error between each field and an (analytic or reference) solution is below a specificed tolerance.

This relies largely on functionality of:

- `credo.analysis.fields` to specify the comparison operations
- `credo.io.stgcvg` to analyse the "convergence" files containing comparison information produced by these operations.

Other than those that are directly saved as attributes documented below, the constructor arguments of interest are:

- useReference: determines whether the fields are compared against a reference, or analytic solution. See `credo.analysis.fields.FieldComparisonList.useReference()`
- useHighResReference: determines whether the fields are compared against a high resolution reference field, or analytic solution. See `credo.analysis.fields.FieldComparisonList.useHighResReference()`
- referencePath: See `credo.analysis.fields.FieldComparisonList.referencePath()`
- testTimestep: See `credo.analysis.fields.FieldComparisonList.testTimestep()`

`fieldsToTest`
A list of strings containing the names of fields that should be tested- i.e. those that will be compared with an expected solution. If left as *None* in constructor, this means the fieldsToTest list will be expected to be defined in the StGermain model XML files themselves.

`defFieldTol`
The default allowed tolerance for global normalised error when comparing Fields with their expected values.

`fieldTols`
A dictionary, mapping particular field names to particular tolerances to use, overriding the default. E.g. {"VelocityField":1e-4} means the tolerance used for the VelocityField will be 1e-4.

fComps
A `credo.analysis.fields.FieldComparisonList` used as an operator to attach to ModelRuns to be tested, and do the actual comparison between fields.

fieldResults
Initially {}, after the test is completed will store a dictionary mapping each field name to a Bool saying whether or not it was within the required tolerance.

fieldErrors
Initially {}, after the test is completed will store a dictionary mapping each field name to a float representing the global normalised error in the comparison.

attachOps (*modelRun*)
Implements base class `credo.systest.api.SingleRunTestComponent.attachOps()`.

check (*mResult*)
Implements base class `credo.systest.api.SingleRunTestComponent.check()`.

getTolForField (*fieldName*)
Utility func: given fieldName, returns the tolerance to use for testing that field (may be given by `defFieldTol`, or been over-ridden in `fieldTols`).

credo.systest.fieldCvgWithScaleTC

class credo.systest.fieldCvgWithScaleTC.**FieldCvgWithScaleTC** (*fieldsToTest=None, calcCvgFunc=<function calcFieldCvgWithScale at 0x2e32d70>, fieldCvgCrits={'VelocityField': (1.6000000000000001, 0.98999999999999999), 'PressureField': (0.90000000000000002, 0.98999999999999999), 'recoveredEpsDotField': (1.6000000000000001, 0.98999999999999999), 'recoveredTauField': (1.6000000000000001, 0.98999999999999999), 'recoveredPressureField': (1.6000000000000001, 0.98999999999999999), 'recoveredSigmaField': (1.6000000000000001, 0.98999999999999999), 'StrainRateField': (0.84999999999999998, 0.98999999999999999), 'recoveredStrainRateField': (1.6000000000000001, 0.98999999999999999)}*)
Bases: `credo.systest.api.MultiRunTestComponent`

Checks whether, for a particular set of fields, the error between each field and an (analytic or reference) solution reduces with increasing resolution at a required rate. Thus similar to `FieldWithinTolTC`, except tests accuracy of solution with increasing model resolution.

This relies largely on functionality of:

- `credo.analysis.fields` to specify the comparison operations
- `credo.io.stgcvg` to analyse the "convergence" files containing comparison information produced by these operations.

fieldsToTest
A list of strings containing the names of fields that should be tested- i.e. those that will be compared with an expected solution. If left as *None* in constructor, this means the fieldsToTest list will be expected to be defined in the StGermain model XML files themselves.

fieldCvgCrits
List of Convergence criterions to be used when checking the fields. Currently required to be in the form used by the convernce checking `credo.analysis.fields.calcFieldCvgWithScale()`, which requires tuples of the form (cvg_rate, correlation).

Note: if this list doesn't contain a cvg criterion for a field that's tested, the behaviour is to skip the formal test of this field, but print a warning (based on previous SYS test behaviour).

---

calcCvgFunc
Function to use to calculate convergence of errors of a group of runs - currently uses `credo.analysis.fields.calcFieldCvgWithScale()` by default.

fComps
A `credo.analysis.fields.FieldComparisonList` used as an operator to attach to ModelRuns to be tested, and do the actual comparison between fields.

fErrorsByRun
Initially {}, after the test is completed will store a dictionary mapping each field name to a list of floats representing the global normalised error in the comparison, for each ModelRun, indexed by ModelRun.

fCvgMeetsReq
Initially {}, after the test is completed will store a dictionary mapping each field name to a Bool recording whether the field error converged acceptably as resolution increased, according to the convergence algorithm used.

fCvgResults
Initially {}, after the test is completed will store a dictionary mapping each field name to a tuple containing information on actual convergence rate. See the return value of `credo.analysis.fields.calcFieldCvgWithScale()` for more.

attachOps(*modelRuns*)
Implements base class `credo.systest.api.SingleRunTestComponent.attachOps()`.

check(*resultsSet*)
Implements base class `credo.systest.api.MultiRunTestComponent.check()`.

As well as performing check, will save relevant into to attributes `fErrorsByRun`, `fCvgMeetsReq`, `fCvgResults`.

`credo.systest.fieldCvgWithScaleTC.`**getDofErrorsByRun**(*fComp*, *resultsSet*)
For a given field comparison op, get all the dof errors from a set of runs, indexed primarily by run index.

credo.systest.fieldCvgWithScaleTC.**getNumDofs**(*fComp*, *mResult*)

Hacky utility function to get the number of dofs of an fComp, by checking the result. Need to do this smarter/neater.

credo.systest.fieldCvgWithScaleTC.**printCvgResult**(*fieldName*, *fieldConvResults*)

credo.systest.fieldCvgWithScaleTC.**testAllCvgWithScale**(*lenScales*, *fieldErrorData*, *fieldCvgCriterions*)

Given a lists of length scales, field error data (a dictionary mapping field names to dofError lists for that field), and field convergence criterions, returns a Bool specifying whether all the fields met their required convergence criterions.

The first two arguments can be created by running getFieldScaleCvgData_SingleCvgFile() on a path containing a single cvg file.

credo.systest.fieldCvgWithScaleTC.**testCvgWithScale**(*fieldName*, *fieldConvResults*, *fieldCvgCriterion*)

Tests that for a given field, given a list of fieldConvResults (See credo.analysis.fields.calcFieldCvgWithScale()) - that they converge according to the given fieldCvgCriterion.

Returns result of test (Bool)

credo.systest.outputWithinRangeTC

class credo.systest.outputWithinRangeTC.**OutputWithinRangeTC**(*outputName*, *reductionOp*, *allowedRange*, *tRange=None*, *opDict=None*)

Bases: credo.systest.api.SingleRunTestComponent

Test component to check that a given output parameter (found in the frequent output) is within a given range, and optionally also that this occurs within a given set of model times.

See Also:

credo.io.stgfreq.

outputName

The name of the model observable to check, as it's recorded in the Frequent Output file. E.g. "Vrms".

reductionOp

The reduction operation to perform in choosing where the value should be checked. Simple examples using Python built-ins could be:

- max() - the Maximum value
- min() - the Minimum value

See Also:

credo.io.stgfreq.FreqOutput.getReductionOp()

allowedRange

The allowed range for the paramter to fall into for the test to pass. A tuple of (min,max) form.

tRange

(Optional) determines if a secondary check should be performed, that the parameters value checked (eg max) also fell within a given range of model simulation times as a (min,max) tuple. If *None*, this secondary check won't be performed.

actualVal
After the check is performed, the actual value of the parameter is recorded here.

actualTime
After the check is performed, records the model sim time at which the parameters chosen property (eg max or min) occurred.

withinRange
After the check is performed, records a Bool saying whether the test component passed.

opDict
(Optional) - will be later passed as keyword arguments to the `reductionOp` function - so use if the reduction op function requires these.

attachOps (*modelRun*)
Implements base class `credo.systest.api.SingleRunTestComponent.attachOps()`.

Note: Currently does nothing. Intend to make it ensure the correct plugin is set to be loaded (to make sure observable is generated in FrequentOutput.dat).

check (*mResult*)
Implements base class `credo.systest.api.SingleRunTestComponent.check()`.

credo.systest.imageCompTC

class `credo.systest.imageCompTC.`**ImageCompTC** (*imageFilename*, *tol=None*, *refPath=None*, *genPath=None*)
Bases: `credo.systest.api.SingleRunTestComponent`

Checks whether an image produced by the run (eg by gLucifer) is within a given "tolerance" of an expected image, using functionality of `credo.analysis.images` module.

imageFilename
Filename of the image to be tested.

tol
Tolerance tuple that the resultant image compared to the reference image must be within. In form required by `credo.analysis.images.compare()` .

refPath
Path to look for reference images.

genPath
Path to look for generated images (if left as *None*, will default to ModelRun's specified output path.)

imageResults
List, indexed by run number, of result of comparison test for each run.

imageErrors
List, indexed by run number, containing errors between reference and generated images after comparison.

attachOps (*modelRun*)
Implements base class `credo.systest.api.SingleRunTestComponent.attachOps()`.

check (*mResult*)
Implements base class `credo.systest.api.SingleRunTestComponent.check()`.

`credo.systest.imageCompTC.`**floatsToStr** (*floatList*)
Convert a list/tuple of tolerance to a nice string to print

CHAPTER

SIX

LINKS TO USEFUL PYTHON INFO, SUCH AS TUTORIALS

6.1 General Python tutorials / skills

Good intoductory tutorials for learning Python.

- Basic tutorial: http://docs.python.org/tutorial/
- The archived Python mailing list
- General (computational) programming tutes, focused around Python:
  - Rur-Ple: - An interactive tutorial about Python, based around programming a simple robot - looks fun!
  - Alan Gauld's quick programming tutorial - http://www.freenetpages.co.uk/hp/alan.gauld/
- Software Carpentry: http://software-carpentry.org/ (A good all-round intro to Software for scientists & engineers, but with a particular focus on Python)
- S Lott's Building Skills in Python - looks really good.
  - Includes a section on writing CSV, XML and other file formats.
- Michael Foord's Python Articles page - includes good articles on eg Testing, and Duck Typing in Python.
- The book Python for Software Design - how to think like a Computer Scientist, (direct html link) has some good examples couched in general programmer skill training.

Tutorials particularly from the perspective of those with experience in Perl scripting, and Bash scripting:

- Python and Perl tutorial: http://xahlee.org/perl-python/index.html - showing examples of how to do common tasks in both languages.
- Red Hat Magazine's "Python for Bash Scripters - a well-kept secret"

6.2 Tutorials more for the appropriate 'software engineering' in Python

- Tutorial on making & managing distributable packages: http://guide.python-distribute.org/

6.3 Python libraries of interest for scientific analysis

Regarding these libraries, there is potential for their future integration into the standard CREDO toolbox - but for now users are free to download and experiment with them.

- HDF5 analysis in Python: http://h5py.alfven.org/

CHAPTER

SEVEN

WHAT'S NEW IN CREDO

This page summarises what's new in each CREDO version.

(See the files in the "changelogs" sub-directory of the CREDO distribution for full ChangeLogs based on Mercurial commits.)

7.1 new in credo-0.1.3

An incremental release, the main changes of interest were:

- Improvements to management of timeouts in ModelRuns and Tests
- Added the `credo.jobrunner` package to handle the mechanics of running Models, and refactored the appropiate parts out of ModelRun class.
  - This includes the capability to run models in directories other than where the CREDO script is invoked using the "basePath" parameter.
  - And a small hierarchy of `credo.jobrunner.api.ModelRunError` exceptions.
  - Default MPI command used by the MPIJobRunner is *mpiexec*.
- Improving capabilities of ModelSuite (`credo.modelsuite`):
  - to generate new sorts of suites based on iterators and `credo.modelsuite.StgXMLVariant` classes.
  - to post-process existing ModelSuite results, using new funcs `readResultsFromPath`
- Added a `credo.io.stgcmdline` module to handle simple command-line issues.
- Improved the reporting of model run errors from the command line.
- General improvements to ModelRun class:
  - Added a hook to do post-run cleanup
  - Added functions to do pre-run validation in various ways
- `credo.systest` package improvements:
  - added function to import and run a given set of tests based on test names- useful for integration into scons test suites.
  - Added a new System test, HighResReference, based on code Wendy Sharples sent through.
  - New system test and test components to compare images based on Owen Kaluza's work.
- Bugfixes:

- Fixed an issue in _createDefaultModelRun so paramOverride lists are not accidentally modified.
- In the CREDO SCons module, changing to avoid problems compiling code on systems with old Python.

- Documentation:
  - More examples/howtos, such as *Running a CREDO script (either System testing or analysis) using PBS* and *Example: modifying an existing test suite to test with Multigrid options*
  - Several notes in the FAQ about errors that come up running tests occasionally.

7.2 new in credo-0.1.2

The main changes in this version were:

- Applied an LGPLv2.1 license to the codebase, and added appropriate Copyright statements.
- Can now over-ride the MPI command used for running models by setting the MPI_RUN_COMMAND env variable.
- Updated the `SysTestRunner` class's XML output to be more like that of the Python Unittest XML suite addon, unittest-xml-reporting (which helps Bitten integration). The XML suite results are now written to separate sub-files, in a "testLogs" sub directory by default.
- Refactored the `SysTest` class hierarchy to simplify it, it's now easier to write sub-classes as they all use a default *check* function that checks all TestComponents.
- Added capability to specify a Timeout for system tests, after which time the test is deemed to have failed if still running.
- New exception classes:
  - Added a custom exception, `ModelRunError`, to record if a model failed to run.
  - Added a `ModelRunTimeoutError`, (see comments on Timeout above.)
- Added new reduction operators for `stgfreq` module, *first* and *last*, that help with setting up system tests based on this.
- Bug/version fixes:
  - Updated the code used to generate model suites so that the itertools.product function, which is Python 2.6 onwards, is replaced by similar functionality if using Python 2.5
  - Fixed SCons integration stuff to make sure paths are set correctly on all machines.
  - Fixed the ability to save plots in non-standard directories from frequent output data.

7.3 new in credo-0.1.1

The main changes in this version were:

- Change from all tests being directly attached to a `SysTestRunner` class, to ability to define `SysTestSuite` classes that could then be run by the SysTestRunner. This makes it possible to better control running of multiple suites, and also makes the interface closer to Python's Unittest module.
- Setting up of conventions so CREDO Suites can be both imported and run as part of a collection, or run directly. See *Requirements for importing test suites: Dual-mode, and the suite() function*.

- Much better integration with SCons, so the user doesn't have to set up special environment variables if you just wish to run tests via SCons, and also are able to generate proper reports on the different suites that were run. See *Running a test target, or test suite, via the SCons build system*.
- Several small but useful interface improvements when defining suites and system tests, including:
  - the ability to define a suffix to append to the output directory of model runs and sys test suites.
  - The ability to define a solverOpts file to use for each run, that contains options to customise the PETSc solver.
  - Making sure the stdout and stderr logs of system tests and models are saved properly.
  - Checking that the input files specified for a model are defined correctly.
- Improved the `credo.systest.fieldCvgWithScaleTest` module considerably, to be more modular, and also better handle fields that don't have a convergence criterion specified. Default criterion were also added for common recovered fields (eg recovered Pressure).
- Created a new `credo.io.stgpath` module, which contains several useful path-manipulation utilities.
- Testing of CREDO: improved the unit tests of CREDO itself, so the majority of testing dirs now have a "testAll.py" script that runs all the other tests.

CHAPTER

EIGHT

CREDO APPENDIX (INC DEVELOPER NOTES)

This is for appendix topics, such as notes for CREDO developers.

8.1 Notes for CREDO developers / maintainers

8.1.1 Coding standards / style guides

On recommendation of the majority of Python tutorials out there, we are following the standard Python style guide, at http://www.python.org/dev/peps/pep-0008/.

Some of the notable things here are:

- Use *4 spaces* for indentation, instead of tabs (obviously crucial to be consistent about this in Python, since indenting whitespace is used to control blocks.
- Spaces around assignment symbols, =, but not between brackets and arguments.

Where we 'break' any of these rules in the package, we'll document this here.

8.1.2 Testing framework of CREDO itself

Yes, a system testing and benchmarking framework needs itself to be tested to minimise the chance of bugs occuring when CREDO is put to use!

Currently, we use the Python Unittest framework. Tests are mainly organised per file (module), rather than per class.

It's important to write unit tests for all new classes and to maintain and improve the suite for existing ones. Unittest is pretty easy to use and is well documented.

The current test structure is:

- Each CREDO package has a *tests* subdirectory (e.g. credo/systests has a subdirectory credo/systests/tests) where test suites are stored;
  - In this subdirectory each CREDO module and/or Class from the above package should have a test suite associated with it, named *xxsuite.py*, where *xx* is the name of the class/module being tested (e.g. imageCompTestsuite.py). The consistent naming pattern aids auto-discovery of tests.

  Note: we leave it up to developer judgement whether to write tests focused on modules or classes within them. It varies depending on the nature of the package.

- The test suites themselves are structured as normal for unittest suites, including a dual-mode __main__ section at the end that runs the current suite if the test is run directly from the command line.
- Each testing package also should have a "testAll.py", that automatically runs all of the suites in that directory. These can be pretty standard code to auto-discover the other suites - look at any of the examples of these that already exist.

Note: Currently, there is no automated way to test the entire suite of CREDO Tests in one batch. This will be coming soon in the next release.

8.2 How to build CREDO Documentation locally

CREDO uses the Sphinx Python documentation package to manage and build its documentation, see http://sphinx.pocoo.org/.

It uses several Sphinx plugins to auto-generate various documentation sections such as class diagrams, requiring Graphviz to be installed. If you want it to generate Latex output as well as HTML, you will need to install a suitable Latex compiler as well.

On Linux systems such as Ubuntu, you should be able to install the Sphinx package.

On Mac Os X, we've tested using the *py26-sphinx* and *graphviz* MacPorts packages and it's working. You will need to set the variable *SPHINXBUILD=sphinx-build-2.6* as Macports installs the Sphinx binaries with a non-standard name.

8.3 Process to follow for creating new CREDO releases

8.3.1 General release configuration description

CREDO follows a fairly standard open source project release naming convention, where releases are named MAJOR.MINOR.MICRO, e.g. 0.1.3, where MAJOR is the major release, MINOR is the micro release, and MICRO are small releases with incremental updates/bugfixes.

CREDO follows a typical 'named branches' and tags strategy for managing these versions and releases in the Mercurial repository:

- There are 'named branches' for major release lines, e.g. *credo-0.1*
- Actual micro releases are tags along these branches of when the code was released, e.g. *credo-0.1.1*
- That way, patches/fixes can be ported across to the release branch lines from the 'default' trunk branch if necessary.

Note: The following websites are useful references for how named branches, especially w.r.t. releases, can be managed in Mercurial:

- http://hgbook.red-bean.com/read/managing-releases-and-branchy-development.html
- http://mercurial.selenic.com/wiki/NamedBranches
- http://stackoverflow.com/questions/890723/mercurial-named-branches-vs-multiple-repositories (especially Geoffrey Zheng's answer).

8.3.2 Release Process Steps

Follow these steps when creating a CREDO release:

1. Review the README.txt file, and if necessary make any updates (eg updating the list of contributors, or web links).
2. Create/switch to the release branch:
   - In the cases where the release branch doesn't exist, you need to create it by using a "hg branch BRANCH-NAME" command.
   - In cases where it does exist, just switch to the branch using "hg update -C BRANCHNAME".
3. Create a CHANGELOG file for the release:
   - use Mercurial command to select all changes since last release tag made, and save to a file in the changelogs subdirectory. E.g. for getting changes between the 0.1.2 and the tip (upcoming 0.1.3 release):

     ```
     hg log -v -r credo-0.1.2:tip > changelogs/CHANGELOG-0_1_3
     ```

     Note: Save this to file in the "changelogs" subdirectory named *CHANGELOG-X-Y-Z*, where X, Y, and Z are the major, minor and micro revisions.

   - Edit the file you just created, and add an appropriate header, e.g.

     ```
     =================================================================
     Mercurial commits made during CREDO development from 0.1.2 to 0.1.3
     =================================================================
     ```

   - Remember to *hg add* the file, you will commit it later.

   Note: We would be interested in any better method for assembling changelogs in Python mercurial code that helps select changes of real interest.

4. Update the *doc/credo-whatsnew.rst* file:
   - Add a new section header for the new release following on the pattern of previous ones, and add a short summary of key new features in bullet point form.

     The CHANGELOG file you just created in the previous step should be helpful in doing this, as well as trac tickets and roadmaps.

     Where appropriate use Sphinx links in this summary to help users find new features.
5. Update the release number in the documentation conf.py file:

   This will look something like the following, just edit as appropriate:

   ```
   # The short X.Y version.
   version = '0.1'
   # The full version, including alpha/beta/rc tags.
   release = '0.1.2'
   ```

6. At this stage you should commit all the changes made above to the release branch, and also tag the release code:

   ```
   hg commit
   hg tag TAGNAME
   hg push
   ```

7. Create a source tarball of this release:

   Make sure you are in the root directory of your CREDO checkout and type:

   ```
   hg archive -t tgz ../credo-X.Y.Z.tar.gz
   ```

   ... where *X.Y.Z* is the release number, e.g. *0.1.3*. This will create the tarball in the parent directory.

8. Update links and upload the just-created tarball in the release table at https://www.mcc.monash.edu.au/trac/AuScopeEngineering/wiki/CREDO

9. Now make sure you commit all the changes you just made on the release branch back to the default trunk development branch:

   ```
   hg update default
   hg merge RELEASE_BRANCH
   hg commit
   hg push
   ```

   Note: This will mark your release branch as (inactive), but this is fine: if it's needed you can hg update to it and reactivate it later.

   ---

10. TODO: in future we should perhaps have a process to create a copy of the documentation of each release (PDF/html) and put online.

CHAPTER

NINE

CREDO FAQ

9.1 CREDO's capabilities

9.1.1 Can I get CREDO to submit PBS jobs, or run it via PBS?

The answer to the first of these is: yes, but this is an experimental feature still in beta. Check out `credo.jobrunner.pbsjobrunner` if you are interested in this, and if you have an Underworld checkout, the script in *Underworld/InputFiles/credo_rayTaySuitePBS.py*.

You can also submit a Python CREDO script in parallel on a HPC system running PBS by writing the appropriate PBS script yourself, and embedding the CREDO call within it - see *Different ways to launch CREDO scripts* for an example.

9.2 Problems running tests

9.2.1 When I try to run a CREDO system test script get an "ImportError" message

This problem is usually because you haven't add the directory containing the CREDO Python source to your PYTHONPATH. See *Setting up your environment to use CREDO* for the various ways to do this.

9.2.2 Errors trying to run one of the Underworld Science benchmarks from command line

Currently (11/4/2011), these tests have been written so that by default when run from the command line, they expect to post-process the benchmark tests and reporting from an existing set of results.

If you want to modify this behaviour so that you *do* first run and generate the models required by the benchmark, then set the *postProcessFromExisting* flag to *False* in the __main__ section at the bottom of a benchmark, e.g.

```
if __name__ == "__main__":
    postProcFromExisting = False
    jobRunner = credo.jobrunner.defaultRunner()
    testResult, mResults = sciBTest.runTest(jobRunner, postProcFromExisting,
        createReports=True)
```

9.2.3 Problem with parallel system tests, a CVGReadError occurs trying to read CVGs

Q: I have a problem running parallel CREDO system tests, along the lines of being unable to parse Field convergence results, where it brings up an exception message along the lines of: "credo.io.stgcvg.CVGReadError: Error, couldn't read expected error" ...

A: This problem is often caused by using an "mpirun" or "mpiexec" not corresponding to the MPI library you compiled the code with. Not doing this (eg running the code with OpenMPI that you compiled with MPICH2) can result in annoying parallel bugs e.g. corruptions when writing output files, which among other things throw off the CREDO system testing now included with the code.

What are the ways of dealing with this?

1. On clusters using the "module" system, just make sure you load the same modules when running the code (eg in PBS scripts) as when you compiled it.
2. Make sure your PATH system environment variable is set up correctly so that "mpirun" will launch the correct version you compiled with. (You can test this by running 'which mpirun'). It's a good idea to set this up in your login scripts, e.g. .bashrc.
3. For CREDO specifically, you can set the environment variable MPI_RUN_COMMAND to tell it to use a custom mpirun/mpiexec rather than the default one in your PATH. (E.g. setting MPI_RUN_COMMAND="/usr/local/packages/mpich2-1.2.1p1-debug/bin/".)

CHAPTER

TEN

GLOSSARY OF TERMS

API API - Application Programming Interface. This means a defined, documented interface for using any software tool or framework, that others can use in their programs to complete a task.

Benchmark In the context of CREDO, this means a test of a code's results/performance according to some metric, which has generally been published in the literature.

Field A 3D domain - in StGermain codes, the FieldVariable class provides an interface for accessing data from Fields, which are usually discretised as either a *Mesh* or *Swarm*. For reference to the FieldVariable class, see http://www.auscope.monash.edu.au/codex/StgDomain.html#FieldVariable

Input File In the StGermain context, see *Model*.

Mesh A mesh of nodes or data in a discretised domain. For the reference to the Mesh class provided by StGermain and Underworld, see http://www.auscope.monash.edu.au/codex/StGermain.html#Mesh.

Model In the context of StGermain-based codes, a "Model" refers to a complete, consistent scientific application: encapsulated by a set of StGermain components. These are normally represented by a set of XML files specifying these components. Sometimes referred to as simply an "input file" to Underworld, and stored in an InputFiles sub-directory.

Model Result In the context of CREDO, this refers to the "result" of a *Model Run*. This includes all the data the model run produced, usually stored in an output directory (such as checkpoint files, and the FrequentOutput.dat summary), plus any 'meta-data' about how long the model took to run, how much data was used, etc. See the `credo.modelresult` module.

Model Run In the context of CREDO, this means the specification of a *Model* to run, PLUS the metadata required for the run (e.g. over-rides of simulation parameters, processors to use, whether to run locally or over the grid, etc). See the `credo.modelrun` module.

StGermain An object-oriented framework, written in C, to enable the development of Scientific applications to run on parallel computers. See http://www.stgermainproject.org

Swarm In the context of StGermain-based codes, a "Swarm" refers to a set of particles. In Underworld a swarm with a large number of particles is used to discretise a *Field* via individual material points. See http://www.auscope.monash.edu.au/codex/StgDomain.html#Swarm.

System Test In the context of software, a test that the entire software system works as expected, for some sort of non-trivial problem.

Underworld A geophysics modelling framework, implemented using *StGermain* - see http://www.underworldproject.org

Virtual method A method of a class that is not actually implemented, and thus requires sub-classes to implement.

VRMS Velocity Root Mean Squared. An observable commonly calculated for convection runs of the Underworld code.

CHAPTER

ELEVEN

INDICES AND TABLES

- (CREDO Python framework) *genindex*
- (CREDO Python framework) *modindex*
- *search*

BIBLIOGRAPHY

[FarringtonEtAl2005] Farrington, R, Moresi, L, Quenette, S, Turnbull, R, & Sunter, P, 2005, 'Geodynamic benchmarking tests in HPC', Presented at the 2005 APAC Conference, Gold Coast, Australia.

[FomelHennenfent2007] S. Fomel and G. Hennenfent, 2007, 'Reproducible computational experiments using SCons,' in *Proc. IEEE Int. Conf. on Acoustics, Speech and Signal Processing, vol. 4, Apr. 2007, pp. 1257–1260*. [Online]. Available: http://slim.eos.ubc.ca/Publications/Public/Conferences/ICASSP/2007/fomel07icassp.pdf

PYTHON MODULE INDEX

C

credo.analysis, 68
credo.analysis.api, 69
credo.analysis.fields, 69
credo.analysis.images, 72
credo.analysis.modelplots, 73
credo.analysis.stats, 72
credo.io, 61
credo.io.stgcmdline, 63
credo.io.stgcvg, 66
credo.io.stgfreq, 63
credo.io.stgpath, 68
credo.io.stgxml, 61
credo.jobrunner, 56
credo.jobrunner.api, 57
credo.jobrunner.mpijobrunner, 59
credo.jobrunner.pbsjobrunner, 60
credo.jobrunner.unixTimeCmdProfiler, 60
credo.modelresult, 50
credo.modelrun, 46
credo.modelsuite, 52
credo.systest, 74
credo.systest.analyticMultiResTest, 84
credo.systest.analyticTest, 82
credo.systest.api, 74
credo.systest.fieldCvgWithScaleTC, 87
credo.systest.fieldWithinTolTC, 86
credo.systest.imageCompTC, 90
credo.systest.imageReferenceTest, 84
credo.systest.outputWithinRangeTC, 89
credo.systest.referenceTest, 82
credo.systest.restartTest, 83
credo.systest.sciBenchmarkTest, 85
credo.systest.systestrunner, 79
credo.utils, 56

INDEX

Symbols

_absRecordFile (credo.systest.api.SysTestResult attribute), 78

A

actualTime (credo.systest.outputWithinRangeTC.OutputWithinRangeTC attribute), 90
actualVal (credo.systest.outputWithinRangeTC.OutputWithinRangeTC attribute), 89
add() (credo.analysis.fields.FieldComparisonList method), 70
addNsPrefix() (in module credo.io.stgxml), 61
addRun() (credo.modelsuite.ModelSuite method), 53
addTestComp() (credo.systest.sciBenchmarkTest.SciBenchmarkTest method), 85
addVariant() (credo.modelsuite.ModelSuite method), 53
allowedRange (credo.systest.outputWithinRangeTC.OutputWithinRangeTC attribute), 89
AnalysisOperation (class in credo.analysis.api), 69
analysisOps (credo.modelrun.ModelRun attribute), 47
analysisXML (credo.modelrun.ModelRun attribute), 48
analysisXMLGen() (credo.modelrun.ModelRun method), 48
AnalyticMultiResTest (class in credo.systest.analyticMultiResTest), 84
AnalyticTest (class in credo.systest.analyticTest), 82
API, 103
applyToModel() (credo.modelsuite.JobParamVariant method), 52
applyToModel() (credo.modelsuite.ModelVariant method), 54
applyToModel() (credo.modelsuite.StgXMLVariant method), 54
archiveRunCommand() (credo.jobrunner.mpijobrunner.MPIJobRunner method), 59
attachAllTestCompOps() (credo.systest.api.SysTest method), 77
attachOps() (credo.systest.api.MultiRunTestComponent method), 75
attachOps() (credo.systest.api.SingleRunTestComponent method), 76
attachOps() (credo.systest.fieldCvgWithScaleTC.FieldCvgWithScaleTC method), 88
attachOps() (credo.systest.fieldWithinTolTC.FieldWithinTolTC method), 87
attachOps() (credo.systest.imageCompTC.ImageCompTC method), 90
attachOps() (credo.systest.outputWithinRangeTC.OutputWithinRangeTC method), 90
attachPerformanceInfo() (credo.jobrunner.api.JobRunner method), 57
attachPerformanceInfo() (credo.jobrunner.api.PerformanceProfiler method), 59
attachPerformanceInfo() (credo.jobrunner.unixTimeCmdProfiler.UnixTimeC method), 60
attachPlatformInfo() (credo.jobrunner.api.JobRunner method), 57
attachPlatformInfo() (credo.jobrunner.mpijobrunner.MPIJobRunner method), 59

B

basePath (credo.modelrun.ModelRun attribute), 47
basePath (credo.systest.api.SysTest attribute), 76
Benchmark, 103
blockResult() (credo.jobrunner.api.JobRunner method), 57
blockResult() (credo.jobrunner.mpijobrunner.MPIJobRunner method), 59
blockResult() (credo.jobrunner.pbsjobrunner.PBSJobRunner method), 60
blockSuite() (credo.jobrunner.api.JobRunner method), 57

C

calcCvgFunc (credo.systest.fieldCvgWithScaleTC.FieldCvgWithScaleTC attribute), 88
calcFieldCvgWithScale() (in module credo.analysis.fields), 72
channelDiff() (in module credo.analysis.images), 72
check() (credo.systest.api.MultiRunTestComponent method), 75
check() (credo.systest.api.SingleRunTestComponent method), 76

check() (credo.systest.fieldCvgWithScaleTC.FieldCvgWithScaleTC method), 88
check() (credo.systest.fieldWithinTolTC.FieldWithinTolTC method), 87
check() (credo.systest.imageCompTC.ImageCompTC method), 90
check() (credo.systest.outputWithinRangeTC.OutputWithinRangeTC method), 90
checkAllXMLInputFilesExist() (in module credo.io.stgpath), 68
checkModelResultsValid() (credo.systest.analyticMultiResTest.AnalyticMultiResTest method), 84
checkModelResultsValid() (credo.systest.analyticTest.AnalyticTest method), 82
checkModelResultsValid() (credo.systest.api.SysTest method), 77
checkModelResultsValid() (credo.systest.imageReferenceTest.ImageReferenceTest method), 85
checkModelResultsValid() (credo.systest.referenceTest.ReferenceTest method), 83
checkModelResultsValid() (credo.systest.restartTest.RestartTest method), 83
checkNoDuplicates() (credo.modelrun.SimParams method), 49
checkParamOverridesTypes() (in module credo.modelrun), 50
checkSolverOptsFile() (credo.modelrun.ModelRun method), 48
checkStgXMLResultsEnabled() (credo.analysis.fields.FieldComparisonList method), 70
checkType() (credo.modelrun.StgParamInfo method), 50
checkValidParams() (credo.modelrun.SimParams method), 49
checkValidRunConfig() (credo.modelrun.ModelRun method), 48
checkXMLInputFileExists() (in module credo.io.stgpath), 68
cleanAllLogFiles() (credo.modelsuite.ModelSuite method), 53
cleanAllOutputPaths() (credo.modelsuite.ModelSuite method), 53
closestToSimTime() (in module credo.io.stgfreq), 66
closestToStep() (in module credo.io.stgfreq), 66
closestToVal() (in module credo.io.stgfreq), 66
cmdLineStr() (credo.modelsuite.JobParamVariant method), 52
cmdLineStr() (credo.modelsuite.StgXMLVariant method), 55
colourDiff() (in module credo.analysis.images), 72
compare() (in module credo.analysis.images), 73
configureSuite() (credo.systest.api.SysTest method), 77
configureSuite() (credo.systest.sciBenchmarkTest.SciBenchmarkTest method), 85
configureTestComps() (credo.systest.analyticMultiResTest.AnalyticMultiRe method), 84
configureTestComps() (credo.systest.analyticTest.AnalyticTest method), 82
configureTestComps() (credo.systest.api.SysTest method), 77
configureTestComps() (credo.systest.imageReferenceTest.ImageReferenceT method), 85
configureTestComps() (credo.systest.referenceTest.ReferenceTest method), 83
configureTestComps() (credo.systest.restartTest.RestartTest method), 83
configureTestComps() (credo.systest.sciBenchmarkTest.SciBenchmarkTest method), 85
convertLocalXMLFilesToAbsPaths() (in module credo.io.stgpath), 68
cpFields (credo.modelrun.ModelRun attribute), 48
cpReadPath (credo.modelrun.ModelRun attribute), 47
createFlattenedXML() (in module credo.io.stgxml), 61
createNewStgDataDoc() (in module credo.io.stgxml), 61
createReports() (credo.systest.api.SysTest method), 77
credo.analysis (module), 68
credo.analysis.api (module), 69
credo.analysis.fields (module), 69
credo.analysis.images (module), 72
credo.analysis.modelplots (module), 73
credo.analysis.stats (module), 72
credo.io (module), 61
credo.io.stgcmdline (module), 63
credo.io.stgcvg (module), 66
credo.io.stgfreq (module), 63
credo.io.stgpath (module), 68
credo.io.stgxml (module), 61
credo.jobrunner (module), 56
credo.jobrunner.api (module), 57
credo.jobrunner.mpijobrunner (module), 59
credo.jobrunner.pbsjobrunner (module), 60
credo.jobrunner.unixTimeCmdProfiler (module), 60
credo.modelresult (module), 50
credo.modelrun (module), 46
credo.modelsuite (module), 52
credo.systest (module), 74
credo.systest.analyticMultiResTest (module), 84
credo.systest.analyticTest (module), 82
credo.systest.api (module), 74
credo.systest.fieldCvgWithScaleTC (module), 87
credo.systest.fieldWithinTolTC (module), 86
credo.systest.imageCompTC (module), 90
credo.systest.imageReferenceTest (module), 84

credo.systest.outputWithinRangeTC (module), 89
credo.systest.referenceTest (module), 82
credo.systest.restartTest (module), 83
credo.systest.sciBenchmarkTest (module), 85
credo.systest.systestrunner (module), 79
credo.utils (module), 56
CREDO_ERROR (class in credo.systest.api), 74
CREDO_FAIL (class in credo.systest.api), 75
CREDO_PASS (class in credo.systest.api), 75
CvgFileInfo (class in credo.io.stgcvg), 67
cvgFileInfo (credo.analysis.fields.FieldComparisonResult attribute), 71
CVGReadError, 67

D

DEF_TEST_FIELDS (in module credo.systest.referenceTest), 82
defaultModelRunFilename() (credo.modelrun.ModelRun method), 48
defaultProfiler (credo.jobrunner.api.JobRunner attribute), 57
defaultRecordFilename() (credo.modelresult.ModelResult method), 51
defaultSysTestFilename() (credo.systest.api.SysTest method), 77
defFieldTol (credo.systest.fieldWithinTolTC.FieldWithinTolTC attribute), 86
defVal (credo.modelrun.StgParamInfo attribute), 50
detailMsg (credo.systest.api.SysTestResult attribute), 78
dictAsPrettyStr() (in module credo.utils), 56
dofColMap (credo.io.stgcvg.CvgFileInfo attribute), 67
dofErrors (credo.analysis.fields.FieldComparisonResult attribute), 71

F

fComps (credo.systest.fieldCvgWithScaleTC.FieldCvgWithScaleTC attribute), 88
fComps (credo.systest.fieldWithinTolTC.FieldWithinTolTC attribute), 86
fCvgMeetsReq (credo.systest.fieldCvgWithScaleTC.FieldCvgWithScaleTC attribute), 88
fCvgResults (credo.systest.fieldCvgWithScaleTC.FieldCvgWithScaleTC attribute), 88
fErrorsByRun (credo.systest.fieldCvgWithScaleTC.FieldCvgWithScaleTC attribute), 88
Field, 103
FieldComparisonList (class in credo.analysis.fields), 70
FieldComparisonOp (class in credo.analysis.fields), 71
FieldComparisonResult (class in credo.analysis.fields), 71
fieldCvgCrits (credo.systest.fieldCvgWithScaleTC.FieldCvgWithScaleTC attribute), 88
FieldCvgWithScaleTC (class in credo.systest.fieldCvgWithScaleTC), 87
fieldErrors (credo.systest.fieldWithinTolTC.FieldWithinTolTC attribute), 87
fieldName (credo.analysis.fields.FieldComparisonResult attribute), 71
fieldResults (credo.modelresult.ModelResult attribute), 51
fieldResults (credo.systest.fieldWithinTolTC.FieldWithinTolTC attribute), 87
fields (credo.analysis.fields.FieldComparisonList attribute), 70
fieldsToTest (credo.systest.fieldCvgWithScaleTC.FieldCvgWithScaleTC attribute), 88
fieldsToTest (credo.systest.fieldWithinTolTC.FieldWithinTolTC attribute), 86
fieldTols (credo.systest.fieldWithinTolTC.FieldWithinTolTC attribute), 86
FieldWithinTolTC (class in credo.systest.fieldWithinTolTC), 86
filename (credo.io.stgcvg.CvgFileInfo attribute), 67
finalStep() (credo.io.stgfreq.FreqOutput method), 64
firstOp() (in module credo.io.stgfreq), 66
floatsToStr() (in module credo.systest.imageCompTC), 90
fmtEls (credo.jobrunner.unixTimeCmdProfiler.UnixTimeCmdProfiler attribute), 60
FreqOutput (class in credo.io.stgfreq), 64
freqOutput (credo.modelresult.ModelResult attribute), 51
fromXML (credo.analysis.fields.FieldComparisonList attribute), 70
fTestName (credo.systest.analyticTest.AnalyticTest attribute), 82
fTestName (credo.systest.referenceTest.ReferenceTest attribute), 83
fTestName (credo.systest.restartTest.RestartTest attribute), 83

G

genConvergenceFileIndex() (in module credo.io.stgcvg), 67
generatedReports (credo.systest.api.SysTest attribute), 76
generateRECOpts() (in module credo.modelrun), 50
generateRuns() (credo.modelsuite.ModelSuite method), 53
genFlattenedXML() (credo.modelrun.ModelRun method), 48
genPath (credo.systest.imageCompTC.ImageCompTC attribute), 90
genSuite() (credo.systest.analyticMultiResTest.AnalyticMultiResTest method), 84
genSuite() (credo.systest.analyticTest.AnalyticTest method), 82
genSuite() (credo.systest.api.SysTest method), 77

genSuite() (credo.systest.imageReferenceTest.ImageReferenceTest method), 85
genSuite() (credo.systest.referenceTest.ReferenceTest method), 83
genSuite() (credo.systest.restartTest.RestartTest method), 83
getAllRecords() (credo.io.stgfreq.FreqOutput method), 64
getAllResults() (credo.analysis.fields.FieldComparisonList method), 70
getCallingPath() (in module credo.utils), 56
getCheckStepsRange() (in module credo.io.stgcvg), 67
getClosest() (credo.io.stgfreq.FreqOutput method), 64
getCmpSrcString() (credo.analysis.fields.FieldComparisonList method), 70
getColNum() (credo.io.stgfreq.FreqOutput method), 64
getComparisonOp() (credo.io.stgfreq.FreqOutput method), 64
getCustomOpts() (credo.modelsuite.ModelSuite method), 53
getDofErrors_ByDof() (in module credo.io.stgcvg), 67
getDofErrors_ByStep() (in module credo.io.stgcvg), 67
getDofErrorsByRun() (in module credo.systest.fieldCvgWithScaleTC), 88
getDofErrorsForStep() (in module credo.io.stgcvg), 67
getElementType() (in module credo.io.stgxml), 62
getFieldScaleCvgData_SingleCvgFile() (in module credo.analysis.fields), 72
getFmtString() (in module credo.jobrunner.unixTimeCmdProfiler), 61
getHeaders() (credo.io.stgfreq.FreqOutput method), 64
getItemFromStrSpec_CurrentCtx() (in module credo.io.stgxml), 62
getListNode() (in module credo.io.stgxml), 62
getMax() (credo.io.stgfreq.FreqOutput method), 64
getMean() (credo.io.stgfreq.FreqOutput method), 64
getMin() (credo.io.stgfreq.FreqOutput method), 64
getModelResultsArray() (in module credo.modelsuite), 55
getModelRunAppExeCommand() (credo.modelrun.ModelRun method), 48
getModelRunCommand() (credo.modelrun.ModelRun method), 48
getNodeFromStrSpec() (in module credo.io.stgxml), 62
getNumDofs() (in module credo.systest.fieldCvgWithScaleTC), 88
getNumEls() (in module credo.analysis.modelplots), 73
getOtherParamValsByVarRunIs() (in module credo.modelsuite), 55
getParam() (credo.modelrun.SimParams method), 49
getParamNode() (in module credo.io.stgxml), 62
getParamOverridesAsStr() (in module credo.modelrun), 50
getParamValue() (in module credo.io.stgxml), 62
getParamValues() (in module credo.modelsuite), 55
getParamValuesIter() (in module credo.modelsuite), 55
getRecordAtStep() (credo.io.stgfreq.FreqOutput method), 64
getRecordDictAtFinalStep() (credo.io.stgfreq.FreqOutput method), 65
getRecordDictAtStep() (credo.io.stgfreq.FreqOutput method), 65
getRecordFile() (credo.systest.api.SysTestResult method), 78
getRecordNum() (credo.io.stgfreq.FreqOutput method), 65
getReductionOp() (credo.io.stgfreq.FreqOutput method), 65
getRes() (in module credo.io.stgcvg), 68
getResDict() (in module credo.jobrunner.unixTimeCmdProfiler), 61
getResult() (credo.analysis.fields.FieldComparisonOp method), 71
getResultsByVarRunIs() (in module credo.modelsuite), 55
getResultsTotals() (credo.systest.systestrunner.SysTestRunner method), 80
getRunByName() (credo.modelsuite.ModelSuite method), 53
getRunIndex() (credo.modelsuite.ModelSuite method), 54
getRunPrefix() (in module credo.jobrunner.unixTimeCmdProfiler), 61
getSimInfoFromFreqOutput() (in module credo.modelresult), 51
getSimParams() (credo.modelrun.ModelRun method), 48
getSpeedups() (in module credo.analysis.modelplots), 73
getStatus() (credo.systest.api.SysTest method), 77
getStdErrFilename() (credo.modelrun.ModelRun method), 48
getStdOutFilename() (credo.modelrun.ModelRun method), 48
getStdOutputPath() (in module credo.systest.api), 79
getStdTestName() (in module credo.systest.api), 79
getStdTestNameBasic() (in module credo.systest.api), 79
getStgBinPath() (in module credo.io.stgpath), 68
getStgStandardXMLPath() (in module credo.io.stgpath), 68
getStructNode() (in module credo.io.stgxml), 62
getSubdir_RunIndex() (in module credo.modelsuite), 55
getSubdir_RunIndexAndText() (in module credo.modelsuite), 55
getSubdir_TextParamVals() (in module credo.modelsuite), 55
getSubdirTextParamVals() (in module credo.modelsuite), 55
getSuiteResultsFilename() (credo.systest.systestrunner.SysTestRunner

method), 80
getSuitesFromModules() (in module credo.systest.systestrunner), 81
getTCRes() (credo.systest.api.SysTest method), 77
getTextParamValsSubdirs() (in module credo.modelsuite), 55
getTimePerEls() (in module credo.analysis.modelplots), 73
getTimestepMap() (credo.io.stgfreq.FreqOutput method), 65
getTimeStepsArray() (credo.io.stgfreq.FreqOutput method), 65
getTolForField() (credo.systest.fieldWithinTolTC.FieldWithinTolTC method), 87
getValsFromAllRuns() (in module credo.analysis.modelplots), 73
getValueAtStep() (credo.io.stgfreq.FreqOutput method), 65
getValuesArray() (credo.io.stgfreq.FreqOutput method), 65
getVariantCmdLineOverrides() (in module credo.modelsuite), 55
getVariantIndicesIter() (in module credo.modelsuite), 55
getVariantNameDicts() (in module credo.modelsuite), 55
getVariantParamPathDicts() (in module credo.modelsuite), 55
getVarRunIs() (in module credo.modelsuite), 55
getVerifyStgExePath() (in module credo.io.stgpath), 68
getVerifyStgMainExecutablePath() (in module credo.io.stgpath), 68

H

headerColMap (credo.io.stgfreq.FreqOutput attribute), 64

I

ImageCompTC (class in credo.systest.imageCompTC), 90
imageErrors (credo.systest.imageCompTC.ImageCompTC attribute), 90
imageFilename (credo.systest.imageCompTC.ImageCompTC attribute), 90
ImageReferenceTest (class in credo.systest.imageReferenceTest), 84
imageResults (credo.systest.imageCompTC.ImageCompTC attribute), 90
indentForPrettyPrint() (in module credo.io.stgxml), 62
Input File, 103
inputFiles (credo.systest.api.SingleModelSysTest attribute), 75
insertNamedElementNode() (in module credo.io.stgxml), 62
iterGen (credo.modelsuite.ModelSuite attribute), 53

J

JobMetaInfo (class in credo.jobrunner.api), 57
jobMetaInfo (credo.modelresult.ModelResult attribute), 51
JobParams (class in credo.modelrun), 46
jobParams (credo.modelrun.ModelRun attribute), 47
JobParamVariant (class in credo.modelsuite), 52
JobRunner (class in credo.jobrunner.api), 57

L

lastOp() (in module credo.io.stgfreq), 66
linreg() (in module credo.analysis.stats), 72
logPath (credo.modelrun.ModelRun attribute), 47
luminanceDiff() (in module credo.analysis.images), 73

M

maxOp() (in module credo.io.stgfreq), 66
maxRunTime (credo.jobrunner.api.ModelRunTimeoutError attribute), 59
Mesh, 103
minOp() (in module credo.io.stgfreq), 66
Model, 103
Model Result, 103
Model Run, 103
modelInputFiles (credo.modelrun.ModelRun attribute), 47
modelName (credo.modelresult.ModelResult attribute), 50
ModelResult (class in credo.modelresult), 50
ModelResultNotExistError, 52
ModelRun (class in credo.modelrun), 46
ModelRunError, 58
ModelRunLaunchError, 58
ModelRunRegularError, 58
ModelRunTimeoutError, 59
ModelSuite (class in credo.modelsuite), 52
ModelVariant (class in credo.modelsuite), 54
modelVariants (credo.modelsuite.ModelSuite attribute), 53
modifyRun() (credo.jobrunner.api.PerformanceProfiler method), 59
modifyRun() (credo.jobrunner.unixTimeCmdProfiler.UnixTimeCmdProfiler method), 60
moveAllToTargetPath() (in module credo.io.stgpath), 68
MPIJobMetaInfo (class in credo.jobrunner.mpijobrunner), 59
MPIJobRunner (class in credo.jobrunner.mpijobrunner), 59
mSuite (credo.systest.api.SysTest attribute), 76
MultiRunTestComponent (class in credo.systest.api), 75
multiRunTestComps (credo.systest.api.SysTest attribute), 76

N

name (credo.analysis.fields.FieldComparisonOp attribute), 71
name (credo.modelrun.ModelRun attribute), 46
navigateStrSpecHierarchy() (in module credo.io.stgxml), 62
nearestDumpStep() (credo.modelrun.SimParams method), 49
normalise() (in module credo.analysis.images), 73
nproc (credo.systest.api.SysTest attribute), 76

O

opDict (credo.systest.outputWithinRangeTC.OutputWithinRangeTC attribute), 90
outputName (credo.systest.outputWithinRangeTC.OutputWithinRangeTC attribute), 89
outputPath (credo.modelresult.ModelResult attribute), 50
outputPath (credo.modelrun.ModelRun attribute), 47
outputPathBase (credo.modelsuite.ModelSuite attribute), 52
outputPathBase (credo.systest.api.SysTest attribute), 76
OutputWithinRangeTC (class in credo.systest.outputWithinRangeTC), 89

P

paramOverrides (credo.modelrun.ModelRun attribute), 47
paramOverrides (credo.systest.api.SingleModelSysTest attribute), 75
paramPath (credo.modelsuite.StgXMLVariant attribute), 54
paramRange (credo.modelsuite.ModelVariant attribute), 54
paramStr() (in module credo.io.stgcmdline), 63
parseUnixTimeElapsed() (in module credo.jobrunner.unixTimeCmdProfiler), 61
PBSJobMetaInfo (class in credo.jobrunner.pbsjobrunner), 60
PBSJobRunner (class in credo.jobrunner.pbsjobrunner), 60
PerformanceProfiler (class in credo.jobrunner.api), 59
pixelDiff() (in module credo.analysis.images), 73
pixelDiff20x20() (in module credo.analysis.images), 73
plotOverAllRuns() (in module credo.analysis.modelplots), 73
plotOverTime() (credo.analysis.fields.FieldComparisonResult method), 71
plotOverTime() (credo.io.stgfreq.FreqOutput method), 65
plotSpeedups() (in module credo.analysis.modelplots), 73
plottedCvgFilename (credo.analysis.fields.FieldComparisonResult attribute), 71
plotTimePerEls() (in module credo.analysis.modelplots), 73
plotWalltimesByRuns() (in module credo.analysis.modelplots), 73
populated (credo.io.stgfreq.FreqOutput attribute), 64
populateFromFile() (credo.io.stgfreq.FreqOutput method), 65
postRun() (credo.analysis.api.AnalysisOperation method), 69
postRun() (credo.analysis.fields.FieldComparisonList method), 71
postRunCleanup() (credo.modelrun.ModelRun method), 49
prepareOutputLogDirs() (credo.modelrun.ModelRun method), 49
preRunCleanup() (credo.modelsuite.ModelSuite method), 54
preRunPreparation() (credo.modelrun.ModelRun method), 49
printAllMinMax() (credo.io.stgfreq.FreqOutput method), 66
printCvgResult() (in module credo.systest.fieldCvgWithScaleTC), 89
printDetailMsg() (credo.systest.api.SysTestResult method), 79
printResultsDetails() (credo.systest.systestrunner.SysTestRunner method), 80
printResultsSummary() (credo.systest.systestrunner.SysTestRunner method), 80
printSuiteResultsByProject() (credo.systest.systestrunner.SysTestRunner method), 80
printSuiteResultsOrderFound() (credo.systest.systestrunner.SysTestRunner method), 80
printSuiteTotalsShortSummary() (credo.systest.systestrunner.SysTestRunner method), 80
productCalc() (in module credo.utils), 56
profilers (credo.jobrunner.api.JobRunner attribute), 57
pType (credo.modelrun.StgParamInfo attribute), 50

R

readFrequentOutput() (credo.modelresult.ModelResult method), 51
readFromRecordXML() (credo.modelresult.ModelResult method), 51
readFromStgXML() (credo.analysis.fields.FieldComparisonList method), 71
readFromStgXML() (credo.modelrun.SimParams method), 49
readFromXMLNode() (credo.jobrunner.api.JobMetaInfo method), 57
readModelResultFromPath() (in module credo.modelresult), 51

readResultsFromPath() (credo.modelsuite.ModelSuite method), 54
recordFieldResult() (credo.modelresult.ModelResult method), 51
reductionOp (credo.systest.outputWithinRangeTC.OutputWithinRangeTC attribute), 89
referencePath (credo.analysis.fields.FieldComparisonList attribute), 70
ReferenceTest (class in credo.systest.referenceTest), 82
refPath (credo.systest.imageCompTC.ImageCompTC attribute), 90
regenerateFixture() (credo.systest.api.SysTest method), 77
regenerateFixture() (credo.systest.imageReferenceTest.ImageReferenceTest method), 85
regenerateFixture() (credo.systest.referenceTest.ReferenceTest method), 83
removeNsPrefix() (in module credo.io.stgxml), 62
resSet (credo.systest.analyticMultiResTest.AnalyticMultiResTest attribute), 84
RestartTest (class in credo.systest.restartTest), 83
resultsList (credo.modelsuite.ModelSuite attribute), 53
runCustomOptSets (credo.modelsuite.ModelSuite attribute), 52
runDescrips (credo.modelsuite.ModelSuite attribute), 52
runExecCmd (credo.jobrunner.api.ModelRunLaunchError attribute), 58
runModel() (credo.jobrunner.api.JobRunner method), 57
runs (credo.modelsuite.ModelSuite attribute), 52
runSingleTest() (credo.systest.systestrunner.SysTestRunner method), 80
runSuite() (credo.jobrunner.api.JobRunner method), 58
runSuite() (credo.systest.systestrunner.SysTestRunner method), 80
runSuiteNonBlockingDefault (credo.jobrunner.api.JobRunner attribute), 57
runSuites() (credo.systest.systestrunner.SysTestRunner method), 80
runSuitesFromModules() (in module credo.systest.systestrunner), 81
runTest() (credo.systest.api.SysTest method), 77
runTests() (credo.systest.systestrunner.SysTestRunner method), 80

S

SciBenchmarkTest (class in credo.systest.sciBenchmarkTest), 85
setCustomReporting() (credo.systest.api.SysTest method), 78
setErrorStatus() (credo.systest.api.SysTest method), 78
setMergeType() (in module credo.io.stgxml), 62
setParam() (credo.modelrun.SimParams method), 49
setPath() (credo.modelrun.ModelRun method), 49
setRecordFile() (credo.systest.api.SysTestResult method), 79
setTimeout() (credo.systest.api.SysTest method), 78
setup() (credo.jobrunner.api.JobRunner method), 58
setup() (credo.jobrunner.api.PerformanceProfiler method), 59
setup() (credo.jobrunner.mpijobrunner.MPIJobRunner method), 59
setup() (credo.jobrunner.pbsjobrunner.PBSJobRunner method), 60
setup() (credo.jobrunner.unixTimeCmdProfiler.UnixTimeCmdProfiler method), 61
setupEmptyTestCompsList() (credo.systest.api.SysTest method), 78
setupTest() (credo.systest.api.SysTest method), 78
setupTest() (credo.systest.sciBenchmarkTest.SciBenchmarkTest method), 85
SimParams (class in credo.modelrun), 49
simParams (credo.modelrun.ModelRun attribute), 47
simtime (credo.jobrunner.api.JobMetaInfo attribute), 57
SingleModelSysTest (class in credo.systest.api), 75
SingleRunTestComponent (class in credo.systest.api), 76
solverOpts (credo.modelrun.ModelRun attribute), 47
solverOpts (credo.systest.api.SingleModelSysTest attribute), 75
solverOptsStr() (in module credo.io.stgcmdline), 63
statusStr (credo.systest.api.SysTestResult attribute), 78
StGermain, 103
stgName (credo.modelrun.StgParamInfo attribute), 49
StgParamInfo (class in credo.modelrun), 49
stgParamInfos (credo.modelrun.SimParams attribute), 49
StgXMLVariant (class in credo.modelsuite), 54
strRes() (in module credo.modelrun), 50
strToBool() (in module credo.io.stgxml), 62
submitRun() (credo.jobrunner.api.JobRunner method), 58
submitRun() (credo.jobrunner.mpijobrunner.MPIJobRunner method), 59
submitRun() (credo.jobrunner.pbsjobrunner.PBSJobRunner method), 60
submitSuite() (credo.jobrunner.api.JobRunner method), 58
subOutputPathGenFunc (credo.modelsuite.ModelSuite attribute), 53
Swarm, 103
System Test, 103
SysTest (class in credo.systest.api), 76
SysTestResult (class in credo.systest.api), 78
SysTestRunner (class in credo.systest.systestrunner), 79
SysTestSetupError, 79

T

tcStatus (credo.systest.api.TestComponent attribute), 79
tcType (credo.systest.api.TestComponent attribute), 79

templateMRun (credo.modelsuite.ModelSuite attribute), 53
testAllCvgWithScale() (in module credo.systest.fieldCvgWithScaleTC), 89
TestComponent (class in credo.systest.api), 79
testComps (credo.systest.api.SysTest attribute), 76
testCvgWithScale() (in module credo.systest.fieldCvgWithScaleTC), 89
testName (credo.systest.api.SysTest attribute), 76
testStatus (credo.systest.api.SysTest attribute), 76
testTimestep (credo.analysis.fields.FieldComparisonList attribute), 70
testType (credo.systest.api.SysTest attribute), 76
timeout (credo.systest.api.SysTest attribute), 76
tol (credo.systest.imageCompTC.ImageCompTC attribute), 90
tRange (credo.systest.outputWithinRangeTC.OutputWithinRangeTC attribute), 89

U

Underworld, 103
UnixTimeCmdHandle (class in credo.jobrunner.unixTimeCmdProfiler), 60
UnixTimeCmdProfiler (class in credo.jobrunner.unixTimeCmdProfiler), 60
updateModelResultsXMLFieldInfo() (in module credo.modelresult), 51
updateOutputPaths() (credo.systest.api.SingleModelSysTest method), 75
updateOutputPaths() (credo.systest.restartTest.RestartTest method), 84
updateXMLWithReports() (credo.systest.api.SysTest method), 78
updateXMLWithResult() (credo.systest.api.SysTest method), 78
updateXMLWithResult() (credo.systest.api.TestComponent method), 79
useHighResReference (credo.analysis.fields.FieldComparisonList attribute), 70
useReference (credo.analysis.fields.FieldComparisonList attribute), 70

V

valLen() (credo.modelsuite.ModelVariant method), 54
valStr() (credo.modelsuite.ModelVariant method), 54
verbPlatformString() (credo.jobrunner.api.JobMetaInfo method), 57
Virtual method, 103
VRMS, 103

W

withinRange (credo.systest.outputWithinRangeTC.OutputWithinRangeTC attribute), 90
withinTol() (credo.analysis.fields.FieldComparisonResult method), 72
writeAllModelResultXMLs() (credo.modelsuite.ModelSuite method), 54
writeAllModelRunXMLs() (credo.modelsuite.ModelSuite method), 54
writeComponent() (in module credo.io.stgxml), 62
writeIncludeLine() (in module credo.io.stgxml), 63
writeInfoXML() (credo.analysis.api.AnalysisOperation method), 69
writeInfoXML() (credo.analysis.fields.FieldComparisonList method), 71
writeInfoXML() (credo.analysis.fields.FieldComparisonOp method), 71
writeInfoXML() (credo.analysis.fields.FieldComparisonResult method), 72
writeInfoXML() (credo.jobrunner.api.JobMetaInfo method), 57
writeInfoXML() (credo.jobrunner.mpijobrunner.MPIJobMetaInfo method), 59
writeInfoXML() (credo.jobrunner.pbsjobrunner.PBSJobMetaInfo method), 60
writeInfoXML() (credo.modelrun.JobParams method), 46
writeInfoXML() (credo.modelrun.ModelRun method), 49
writeInfoXML() (credo.modelrun.SimParams method), 49
writeInputsOutputsToCSV() (in module credo.modelsuite), 56
writeMergeComponent() (in module credo.io.stgxml), 63
writeMergeComponentStruct() (in module credo.io.stgxml), 63
writeParam() (in module credo.io.stgxml), 63
writeParamList() (in module credo.io.stgxml), 63
writeParamOverridesInfoXML() (in module credo.modelrun), 50
writeParamSet() (in module credo.io.stgxml), 63
writePreRunXML() (credo.systest.api.SysTest method), 78
writePreRunXML() (credo.systest.api.TestComponent method), 79
writeRecordXML() (credo.modelresult.ModelResult method), 51
writeRecordXML() (credo.systest.api.SysTest method), 78
writeSolverOptsInfoXML() (in module credo.modelrun), 50
writeStgDataDocToFile() (in module credo.io.stgxml), 63
writeStgDataXML() (credo.analysis.api.AnalysisOperation method), 69
writeStgDataXML() (credo.analysis.fields.FieldComparisonList method), 71
writeStgDataXML() (credo.modelrun.SimParams

method), 49
writeStruct() (in module credo.io.stgxml), 63
writeStructList() (in module credo.io.stgxml), 63
writeValueUsingStrSpec() (in module credo.io.stgxml), 63
writeXMLDoc() (in module credo.io.stgxml), 63

X

xmlExistsInStdXMLPath() (in module credo.io.stgpath), 68