
CREDO Documentation

Release CREDO 0.1.2 (For Underworld 1.5.0)

Patrick Sunter (editor)

August 04, 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	25
4.3	Scientific Benchmarking using CREDO	31
4.4	Different ways to launch CREDO scripts	34
5	CREDO Python Source API Documentation	37
5.1	Model API	38
5.2	Utils API	47
5.3	JobRunner API	47
5.4	IO (Input Output functions) API	48
5.5	Analysis API	55
5.6	SysTest API	60
6	Links to useful Python info, such as tutorials	77
6.1	General Python tutorials / skills	77
6.2	Tutorials more for the appropriate 'software engineering' in Python	77
6.3	Python libraries of interest for scientific analysis	77
7	Notes for CREDO developers / maintainers	79
7.1	Coding standards / style guides	79
7.2	Testing framework	79
8	What's new in CREDO	81
8.1	new in credo-0.1.2	81

8.2	new in credo-0.1.1	82
9	CREDO Appendix	83
9.1	How to build CREDO Documentation locally	83
10	CREDO FAQ	85
10.1	Problems running tests	85
11	Glossary of Terms	87
12	Indices and tables	89
	Bibliography	91
	Python Module Index	93
	Index	95

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:

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).

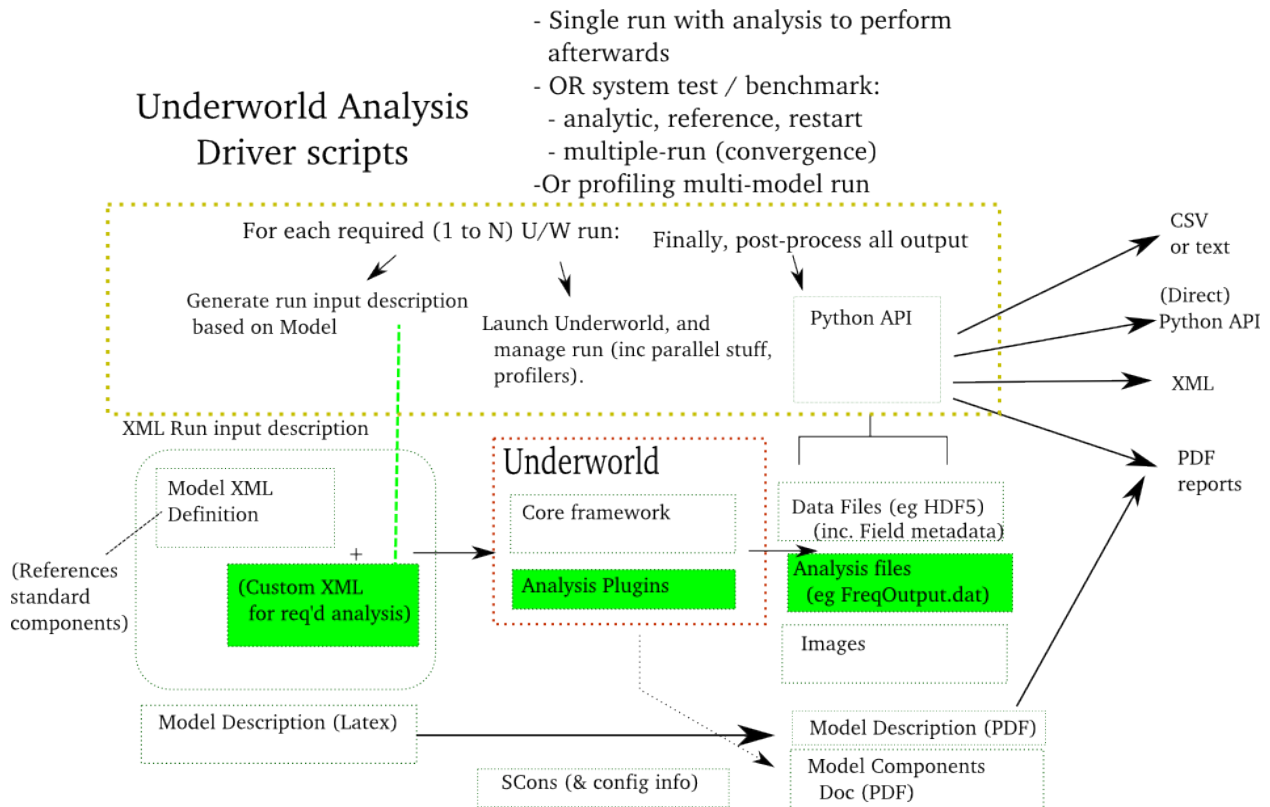
To meet these goals, we chose the *Python* scripting and programming language, explained more in the [Language choice - why Python?](#) section below.

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 explicitly 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 Stg-Underworld 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 (initially 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.

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 supercomputers), 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:

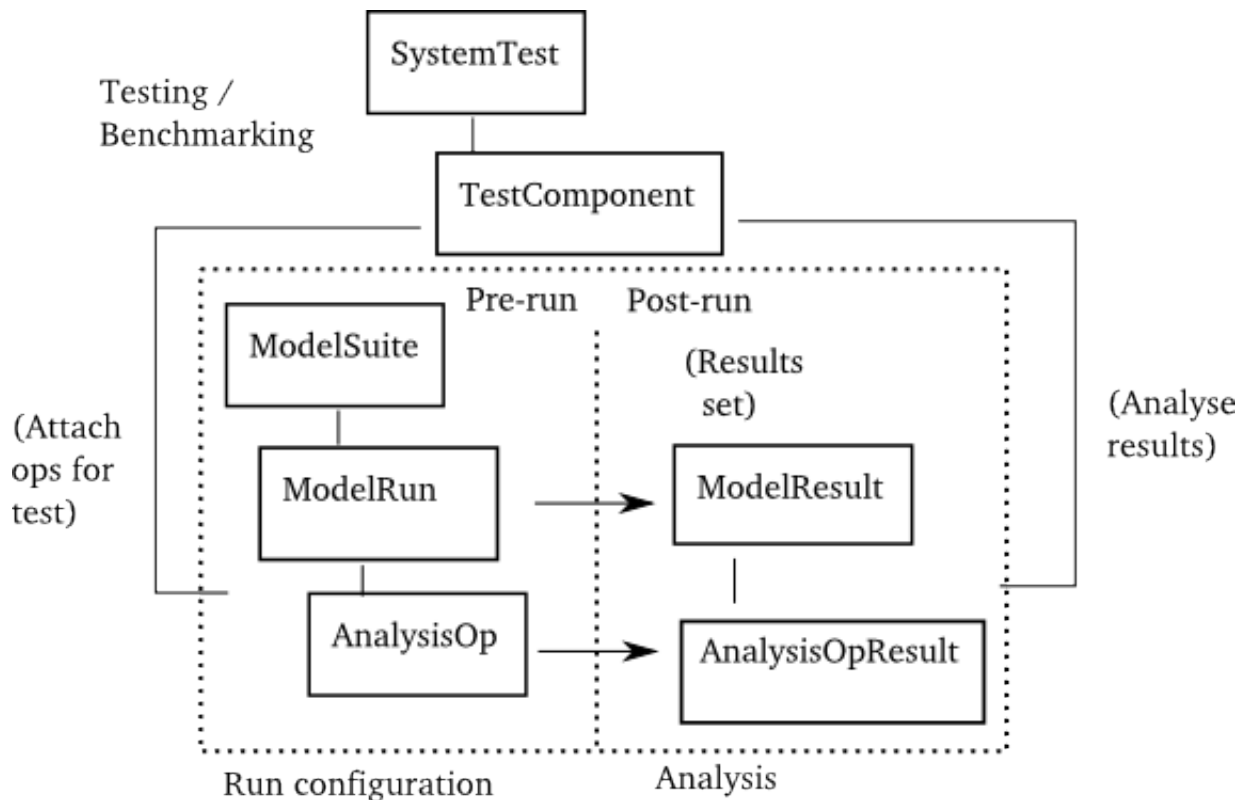
```
[GCC 4.4.1] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import credo
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ImportError: No module named credo
>>>
```

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

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.

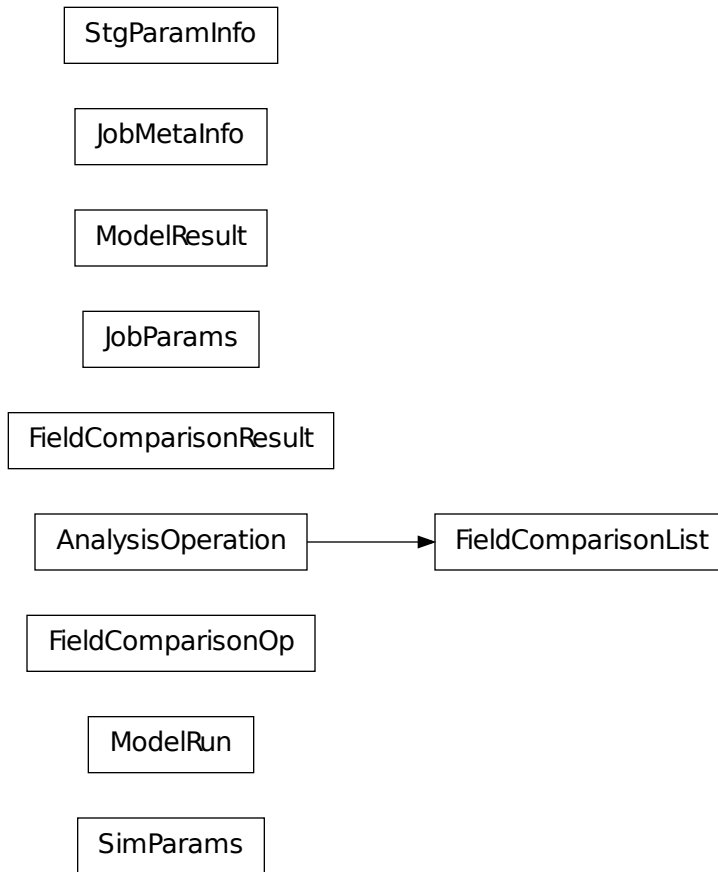
3.1 Organisation of Python Code and class hierarchy

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*.

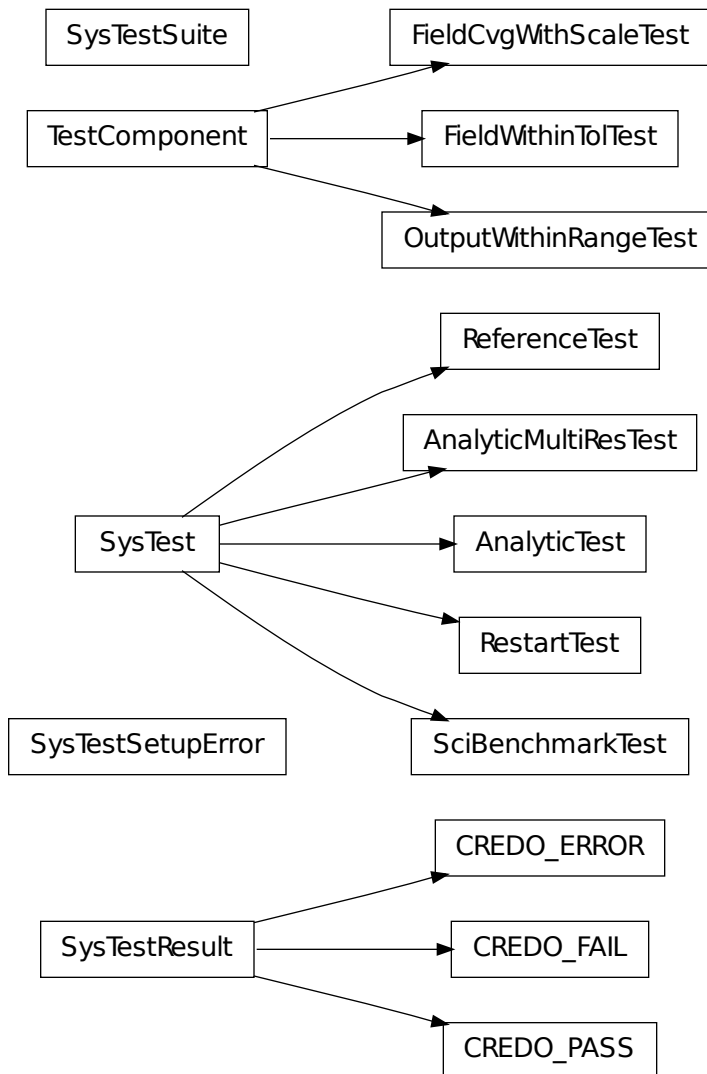
Note: CREDO uses Python's object-oriented capabilities to manage a set of Classes to implement this architecture. Python is designed to be easy to pick up, and from a user's perspective the object-oriented features shouldn't make scripting difficult. Thus if you only intend to write CREDO scripts to 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:



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 SCons build system, so for project such as Underworld it is now possible to run CREDO System tests via SCons.

For this to work, it requires test maintainers to follow the directions in *Requirements for importing test suites: Dual-mode, and the suite() function*, but you can follow the instructions below without knowing the internal contents of suites.

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.ReggressionTests.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 di
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 solut
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 'PICellator', 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 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_50pts = {"defaultDiffusivity":0.5, "A":0.1}  
for nproc in [1, 2, 4]:  
    testSuite.addStdTest(AnalyticTest, ["SteadyState1D-x.xml"],  
                          nproc=nproc, paramOverrides=ss0_50pts)  
  
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, ["TempDiffusion.xml"],  
                    fieldsToTest=["TemperatureField"], highResRatio=4)
```

```

testSuite.addStdTest(RestartTest, ["Multigrid.xml"])
testSuite.addStdTest(ReferenceTest, ["Multigrid.xml"])
testSuite.addStdTest(HighResReferenceTest, ["Multigrid.xml"], highResRatio=4,
    fieldTols={"VelocityField":0.2, "PressureField":0.5})

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.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.

4.2 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 mRun.writeInfoXML()
11
12 jobRunner = credo.jobrunner.defaultRunner()
13 mResult = jobRunner.runModel(mRun)
14 mResult.writeRecordXML()
15
16 #-----
17
18 mResult.readFrequentOutput()
19 mResult.freqOutput.plotOverTime('Vrms', depName='Time', show=True,
20         path=mResult.outputPath)
21
22 maxVal, maxTimeStep = mResult.freqOutput.getMax('Vrms')
23 maxTime = mResult.freqOutput.getValueAtStep('Time', maxTimeStep)
24 print "Maximum value of Vrms was %f, at time %d" % (maxVal, maxTime)

```

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:

```

    path=mResult.outputPath)

maxVal, maxTimeStep = mResult.freqOutput.getMax('Vrms')
maxTime = mResult.freqOutput.getValueAtStep('Time', maxTimeStep)
print "Maximum value of Vrms was %f, at time %d" % (maxVal, maxTime)

```

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:

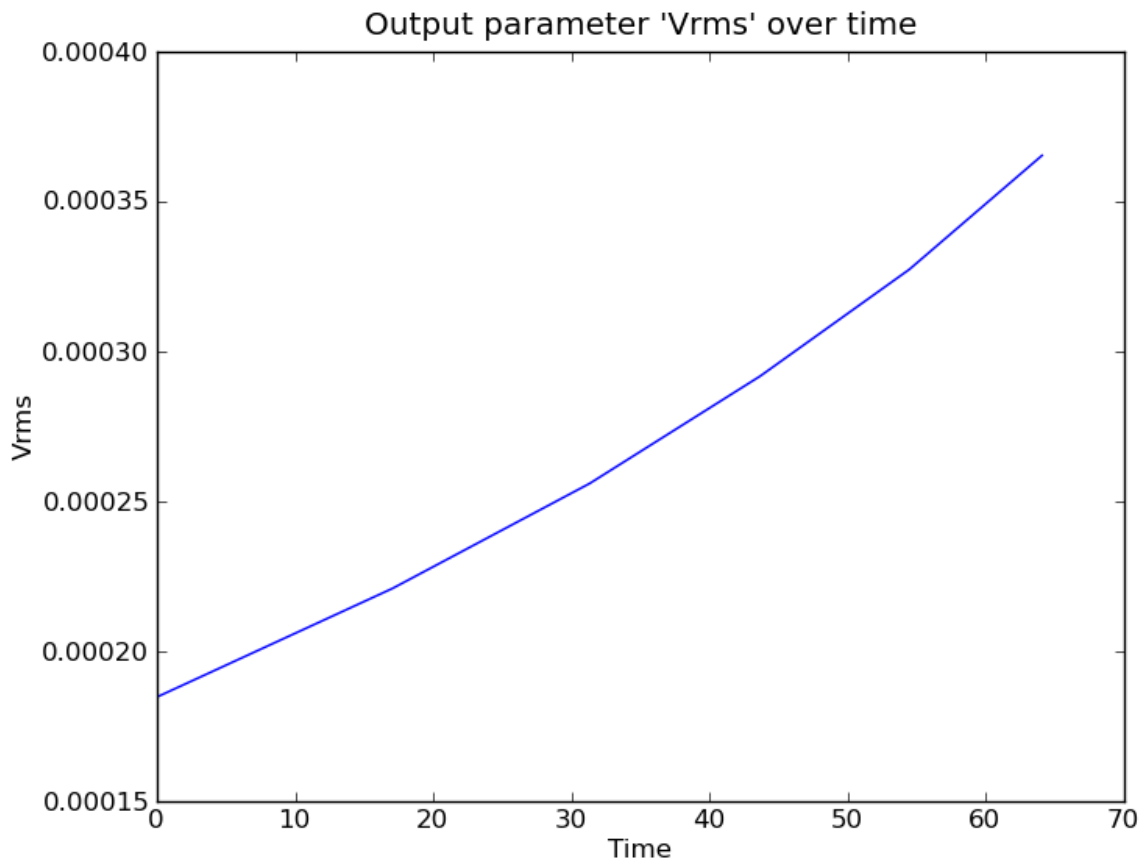
The `credo.io.stgfreq.FreqOutput` class, especially the `plotOverTime()` method.

Outputs

Running this script at the terminal produces:

```
Running model 'RayTay-basicBenchmark' with command 'mpirun -np 1 /home/psunter/AuScopeCodes/stgUnderworld/Underworld-1.5.0/bin/Underworld-1.5.0'
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:

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.

- 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.

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.

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 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:

```
1  #!/usr/bin/env python
2  from credo.modelrun import ModelRun, SimParams
3  from credo.modelsuite import ModelSuite, StgXMLVariant
4  import credo.jobrunner
5
6  mRun = ModelRun("RayTay-basic", "RayleighTaylorBenchmark.xml",
7                   "output/raytay-scibench-credo-basic",
8                   simParams=SimParams(stoptime=60.0))
9
10 mSuite = ModelSuite("output/raytay-suite", templateMRun=mRun)
11 gravRange = [0.7 + x * 0.1 for x in range(4)]
12 mSuite.addVariant("gravity", StgXMLVariant("gravity", gravRange))
13 ampRange = [0.02, 0.04, 0.07]
14 mSuite.addVariant("lightLayerAmplitude",
15                     StgXMLVariant("components.lightLayerShape.amplitude", ampRange))
16
17 mSuite.generateRuns()
18 jobRunner = credo.jobrunner.defaultRunner()
19 mResults = jobRunner.runSuite(mSuite)
20
21 #-----
22
23 for resI, mRes in enumerate(mResults):
24     print "Post-process result %d: with gravity=%g, init perturbation=%g" \
25         % (resI, mSuite.runs[resI].paramOverrides["gravity"], \
26                     mSuite.runs[resI].paramOverrides["components.lightLayerShape.amplitude"])
27     mRes.readFrequentOutput()
28     mRes.freqOutput.plotOverTime('Vrms', depName='Time', show=True,
29                                      path=mRes.outputPath)
30
31     maxVal, maxTimeStep = mRes.freqOutput.getMax('Vrms')
32     maxTime = mRes.freqOutput.getValueAtStep('Time', maxTimeStep)
33     print "\tMaximum value of Vrms was %f, at time %d" % (maxVal, maxTime)
```

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:

```
#-----
for resI, mRes in enumerate(mResults):
    print "Post-process result %d: with gravity=%g, init perturbation=%g" \
          % (resI, mSuite.runs[resI].paramOverrides["gravity"], \
            mSuite.runs[resI].paramOverrides["components.lightLayerShape.amplitude"])
    mRes.readFrequentOutput()
    mRes.freqOutput.plotOverTime('Vrms', depName='Time', show=True,
                                  path=mRes.outputPath)

    maxVal, maxTimeStep = mRes.freqOutput.getMax('Vrms')
    maxTime = mRes.freqOutput.getValueAtStep('Time', maxTimeStep)
    print "\tMaximum value of Vrms was %f, at time %d" % (maxVal, maxTime)
```

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.

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
```

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](#).

```
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
```

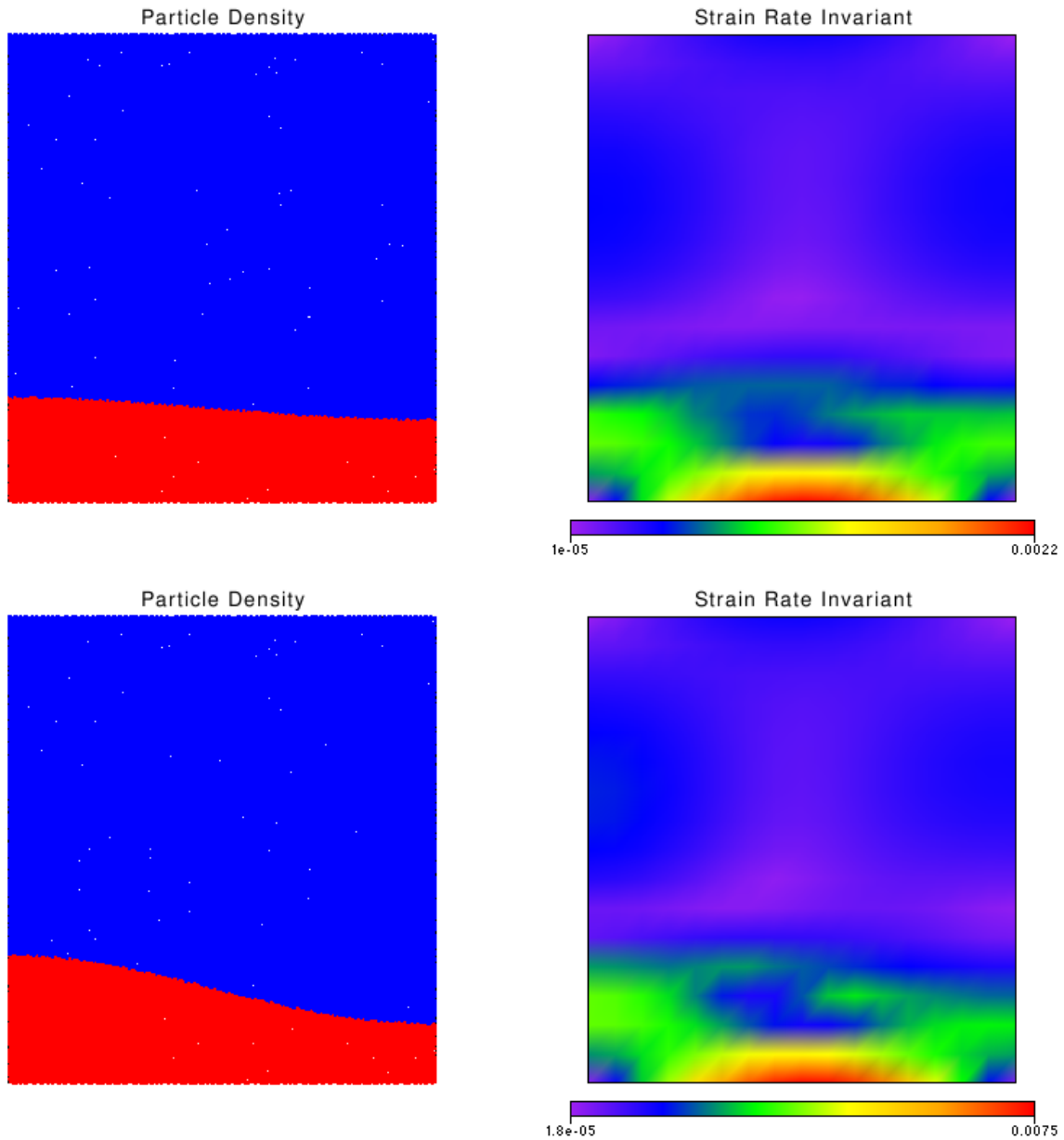
You can see the results of our post-processing for loop here, showing how the VRMS value varied in the different runs due to the different input parameters.

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*.

4.3 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.systest.outputWithinRangeTest import OutputWithinRangeTest

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

inputFiles = ["RayleighTaylorBenchmark.xml"]
modelName = "RayleighTaylor-VanKekenBenchmark"
outputPath = os.path.join('output', modelName)

sciBTest = SciBenchmarkTest(inputFiles, outputPath, nproc=1,
                             nameSuffix="128res",
                             paramOverrides={"stopTime":250, "elementResI":128, "elementResJ":128,
                                                "maxTimeSteps":-1})
vrmsCheck = OutputWithinRangeTest( "Vrms", max, (2.8e-3, 3.2e-3),
                                    tRange=(200,220) )

sciBTest.addTestComponent(vrmsCheck, "VRMS of first diapir")
#sciBTest.addOutputGraph("Vrms", "Time")
#sciBTest.addVisOutput("ParticleDensity", when=("Time", [0,500,1000,1500,2000] ))

testSuite.sysTests.append(sciBTest)

def suite():
    return testSuite

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

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/Au
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
```

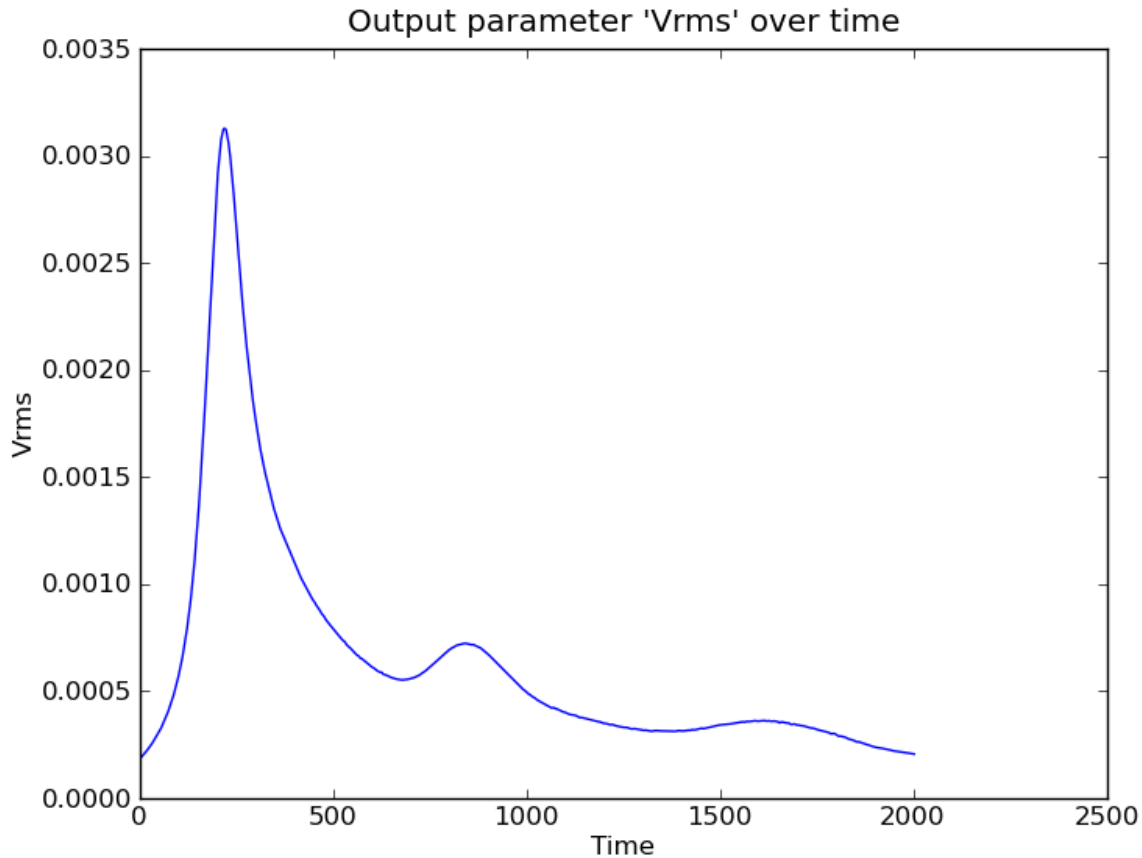
```
-----
CREDO System Tests results summary:
Ran 1 system tests, with 1 passes, 0 fails, and 0 errors
-----
```

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 familiarity 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.esequenece

# 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
```

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.StgParamInfo`

`credo.modelrun.SimParams`

`credo.modelrun.ModelRun`

`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` (*nproc*)

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

Note: Needs fleshing out, it's likely this could be expanded in future as the ability to run Models via PBS or over the grid is developed.

nproc

Number of processors to use in a parallel job.

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, basePath=None, logPath='log', cpReadPath=None, nproc=1, simParams=None, paramOverrides=None, solverOpts=None, xmlExtras=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

Initially *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='credo-analysis.xml'*)

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 ()

defaultModelRunFilename ()

Calculates and returns a default filename for the `ModelRun`’s XML record filename.

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

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.

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.

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`

`credo.modelresult.JobMetaInfo`

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.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.

writeInfoXML (*xmlNode*)

Writes information about this class into an existing, open XML doc node

class `credo.modelresult.ModelResult` (*modelName, outputPath, simtime*)

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 `JobMetaInfo`, recording information about the run such as time taken, Memory usage etc.

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.

readFrequentOutput ()

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

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.defaultModelResultFilename (modelName)`

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

`credo.modelresult.getSimInfoFromFreqOutput (outputPath)`

Get necessary information to create a `SimInfo` from the `FrequentOutput.dat`, given a particular output Path.

See Also:

`credo.io.stgfreq`.

`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`

`credo.modelsuite.ModelSuite`

`credo.modelsuite.StgXMLVariant`

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.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()`.

runDescripts

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

If set, this function will be used to customise the model sub-path based on each `modelRun`.

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.

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*)

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.

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 0x177f2a8>*)

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.zip` iterator generating function. See the Python `itertools` module for more.

getOutputSubPath (*paramIndices*)

Generate the standard output directory for a run with given `paramIndices` into the `modelVariants` to be applied.

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.

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.StgXMLVariant` (*paramPath, 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.

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.

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*)

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

applyToModelGenerator (*modelRun*)

Creates a Python Generator for applying parameters to Apply the ii-th value in the attr: *paramRange* to a particular *ModelRun*.

cmdLineStr (*ii*)

Return the command-line string to apply this value.

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.

`credo.modelsuite.getDefaultSubdir` (*modelVariants, paramIndices*)

`credo.modelsuite.getDefaultSubdirs` (*modelVariants, indicesIt*)

`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.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.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, iterGen, 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.

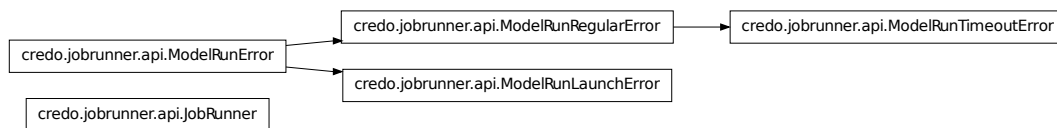
`credo.utils.productCalc(*args, **kwds)`

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.JobRunner`

constructStGermainRunCommand (*modelRun, extraCmdLineOpts=None*)

getStGermainExeStr ()

Get the string used to execute StGermain.

runModel (*modelRun, extraCmdLineOpts=None, dryRun=False, maxRunTime=None*)

Run the specified `modelRun`, and return a `ModelResult` recording the results of the run.

Parameters

- **modelRun** – the `ModelRun` to be run.
- **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.

runSuite (*modelSuite, extraCmdLineOpts=None, dryRun=False, maxRunTime=None*)

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()`.

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.

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*)

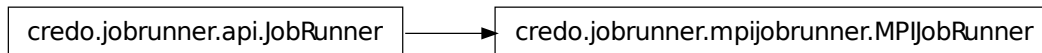
Bases: `credo.jobrunner.api.ModelRunRegularError`

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

maxRunTime

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

5.3.2 `credo.jobrunner.mpijobrunner`



class `credo.jobrunner.mpijobrunner.MPIJobRunner`

Bases: `credo.jobrunner.api.JobRunner`

getStGermainExeStr ()

runModel (*modelRun*, *extraCmdLineOpts=None*, *dryRun=False*, *maxRunTime=None*)

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

setup ()

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 <element> 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.writeMergeComponent` (*rootNode, compName, compType*)
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.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.)

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 occurred.

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 occurred.

getRecordAtStep (*tstep*)

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

getRecordDictAtStep (*tstep*)

For a given timestep *tstep*, looks up the records for that timestep and returns a dictionary of “header-name”: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*)

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.

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 timestep 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 need to be repeated in future.

printAllMinMax()

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

`credo.io.stgfreq.first` (*inList*, *key=None*)

A utility function designed to pass to `attr:~.FrequentOutput.getReductionOp` for getting the first value from a frequent output list.

`credo.io.stgfreq.last` (*inList*, *key=None*)

A utility function designed to pass to `attr:~.FrequentOutput.getReductionOp` for getting the last value from a frequent output list.

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.systemtest`).

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`.

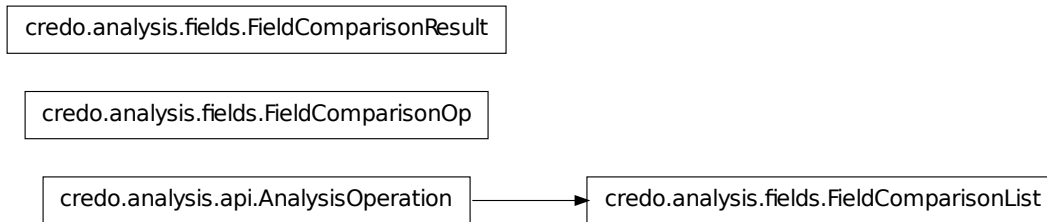
writeInfoXML (*parentNode*)

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` compo-

ment 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.stgcvlg`).

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.stgcvlg.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.stgcvlg.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 <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.colourDiff (img1, img2)`

Calculate image difference by colour histogram.

Parameters

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

Returns float representing euclidian distance between images in colour histogram space

`credo.analysis.images.compare (imgFilename1, imgFilename2, verbose=False)`

Compare two image files.

Returns A tuple containing the diffs for each component (colour space, 4 pixel subsample).

`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.pixelDiff2x2 (img1, img2)`

Compare two open images on a 2x2 basis.

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 StGermain-based codes, and communicate and record the results.

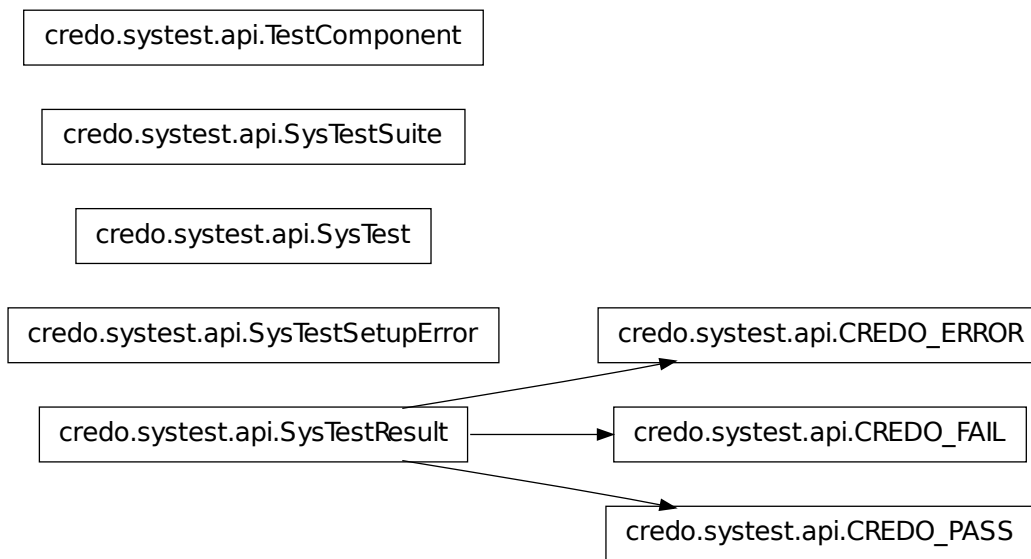
From a user perspective, doing an:

```
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*.

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.SysTest` (*inputFiles, outputPathBase, nproc, paramOverrides, solverOpts, testType, basePath=None, nameSuffix=None, 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.

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.

testType

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

inputFiles

StGermain XML files that define the Model to be tested.

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`.

testStatus

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

testComponents

A list of any `TestComponent` classes used as part of performing the test.

nproc

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

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`

resIndicesToTest

If this is set to other than *None*, then only these indices will be passed to `TestComponents` to check as part of the `getStatus` function.

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.

checkResultValid (*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.

defaultSysTestFilename ()

Return the default system test XML record filename, based on properties of the systest (such as *testName*).

genSuite ()

Must return a `credo.modelsuite.ModelSuite` containing all models that need to be run to perform the test.

Note: most standard system tests should override this function with their own suite generator, and save the suite as `mSuite` in the process.

getResultsSubset (*resultsSet*)

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.

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.

setResIndicesToTest (*resultIndices*)

Sets which result indices should be checked.

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.

setup ()

For the setup phase of tests. Since not all tests need a setup phase, the default behaviour is to do nothing.

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 specification 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.

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.SysTestSuite` (*projectName, suiteName, sysTests=None, subSuites=None, nproc=1*)

Class that aggregates a set of `SysTest`.

For examples of how to use, see the CREDO documentation, especially *Running CREDO system test suites directly, and how to modify them*.

projectName

Name of the “Project” the test suite is attached to, e.g. “Underworld” or “StgFEM”.

suiteName

A short descriptive name of the test suite. NB: doesn’t have to be unique, e.g. you may want to create different suites with the same suite names, but different project names.

sysTests

List of system tests that should be run and reported upon. Generally shouldn’t be accessed directly, recommend using `addStdTest()` to add to this list, and other methods to run and report on it.

subSuites

List of subSuites (defaults to none) associated with this suite. Associating sub-suites allows a nested hierarchy of system tests.

nproc

The default number of processors to use for tests associated with this suite (can be over-ridden per test case though).

addStdTest (*testClass, inputFiles, **testOpts*)

Instantiate and add a “standard” system test type to the list of System tests to be run. (The “standard” refers to the user needing to have access to the module containing the system test type to be added, usually from a *from credo.systest import ** statement.

Parameters

- **testClass** – Python class of the System test to be added. This needs to be a sub-class of `SysTest`.
- **inputFiles** – model input files to be passed through to the System test when instantiated.
- ****testOpts** – any other keyword arguments the user wishes to be passed through to the System test when it's instantiated. Can be used to customise a test.

Returns a reference to the newly created and added `SysTest`.

addSubSuite (*subSuite*)

Adds a single sub-suite to the list of sub-suites.

addSubSuites (*subSuites*)

Adds a set of sub-suites to the list of sub-suites.

newSubSuite (**subSuiteRegArgs, **subSuiteKWArgs*)

Shortcut to create a new sub-suite, add it to the existing suite, and return reference to the newly created sub-suite.

setAllTimeouts (*seconds=0, minutes=0, applyToSubSuites=True*)

Utility function to set all the timeouts for all system tests associated with the suite to a certain value.

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 word descriptive) string.

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 (*resultsSet*)

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

updateXMLWithResult (*tcNode, resultsSet*)

Updates a given XML node with the result of the test component, based on `resultsSet`.

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.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 Python 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.

runSuite (*suite, runSubSuites=True, subSuiteMode=False, outputSummaryDir='testLogs'*)

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.

runSuites (*testSuites, outputSummaryDir='testLogs'*)

Runs a list of suites, and prints a big summary at the end. :param testSuites: list of test suites to run. :keyword 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).

runTest (*sysTest*)

Run a given `sysTest`, and return the `credo.systest.api.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.

runTests (*sysTests, projName=None, suiteName=None, printSummary=True*)

Run all tests in the `sysTests` list. Will also save all appropriate XMLs (as discussed in `runTest()`) and print a summary of results.

`credo.systest.systestranner.getSuitesFromModules` (*suiteModNames*)

Gets a list of suites from the list of suites to import given in `suiteModNames`.

`credo.systest.systestranner.runSuitesFromModules` (*suiteModNames, outputSummary-Dir*)

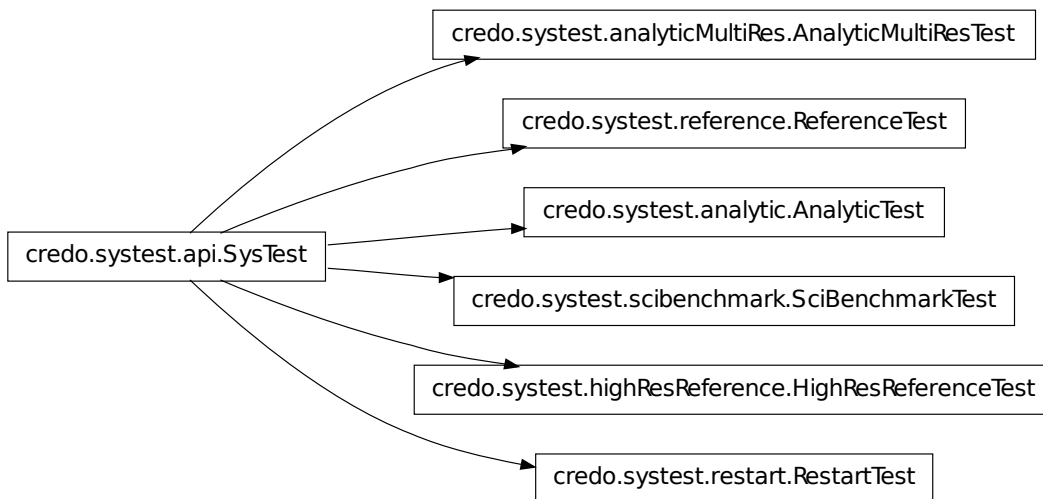
Runs a set of System test suites, where `suiteModNames` is a list of suites to import and run.

5.6.3 Core System Test class implementations

CREDO provides a set of core `SysTest` instantiations, 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.analytic

```
class credo.systest.analytic.AnalyticTest (inputFiles, outputPathBase, nproc=1, defFieldTol=0.02999999999999999, fieldTols=None, paramOverrides=None, solverOpts=None, basePath=None, nameSuffix=None, timeout=None)
```

Bases: `credo.systest.api.SysTest`

An Analytic System test. This case requires the user to have configured the XML correctly to load an analytic soln, and compare it to the correct fields. Will check that each field flagged to be analysed is within the expected tolerance. Uses a `FieldWithinTolTest` 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 `FieldWithinTolTest`'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.

checkResultValid (resultsSet)

See base class `checkResultValid()`.

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.reference

Provides a `ReferenceTest` for use in system testing.

credo.systest.reference.DEF_TEST_FIELDS

Default fields that will be tested, if not explicitly provided as a constructor keyword argument to `ReferenceTest` instantiations.

```
class credo.systest.reference.ReferenceTest (inputFiles, outputPathBase, nproc=1, fieldsToTest=None, runSteps=20, defFieldTol=0.01, fieldTols=None, paramOverrides=None, solverOpts=None, basePath=None, expPathPrefix='expected', nameSuffix=None, timeout=None)
```

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 `FieldWithinTolTest` 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 `FieldWithinTolTest`'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.

checkResultValid (*resultsSet*)

See base class `checkResultValid()`.

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.

setup ()

Do a run to create the reference solution to use.

credo.systest.restart

```
class credo.systest.restart.RestartTest (inputFiles, outputPathBase, nproc=1, field-
sToTest=['VelocityField', 'PressureField'], full-
RunSteps=20, defFieldTol=1.0000000000000001e-
05, fieldTols=None, paramOverrides=None,
solverOpts=None, basePath=None, nameSuf-
fix=None, timeout=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 `FieldWithinTolTest` 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 `FieldWithinTolTest`’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.

checkResultValid (*resultsSet*)

See base class `checkResultValid()`.

genSuite ()

See base class `genSuite()`.

For this test, will create a suite containing 2 model runs: one to initially 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.

credo.systest.analyticMultiRes

```
class credo.systest.analyticMultiRes.AnalyticMultiResTest (inputFiles,    outputPathBase, resSet, nproc=1,
                                                           paramOverrides=None,
                                                           solverOpts=None,
                                                           basePath=None, nameSuffix=None, timeout=None)
```

Bases: `credo.systest.api.SysTest`

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 `FieldCvgWithScaleTest` 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.

checkResultValid (resultsSet)

See base class `checkResultValid()`.

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.imageReference

```
class credo.systest.imageReference.ImageReferenceTest (inputFiles,    outputPathBase,
                                                         imagesToTest,    nproc=1,
                                                         runSteps=20,    deflImageTol=(0.01, 0.01),
                                                         imageTols=None,    paramOverrides=None,
                                                         solverOpts=None,
                                                         basePath=None,    expPathPrefix='expected',
                                                         nameSuffix=None, timeout=None)
```

Bases: `credo.systest.api.SysTest`

An image comparison against Reference System test. To do this, creates a set of several `ImageCompTest` Test Components for each image you wish to test.

Optional constructor keywords:

- **runSteps**: Number of steps the model should be run for before comparing images.
- **deflImageTol**: 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.

checkResultValid (resultsSet)

See base class `checkResultValid()`.

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.

setup ()

Do a run to create the reference images to use.

5.6.4 `credo.systest.scibenchmark`

class `credo.systest.scibenchmark.SciBenchmarkTest` (*inputFiles*, *outputPathBase*,
nproc=1, *paramOverrides=None*,
solverOpts=None, *basePath=None*,
nameSuffix=None, *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.

addTestComponent (*testComp*, *testCompName*)

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.

checkResultValid (*resultsSet*)

See base class `checkResultValid ()`.

genSuite ()

See base class `genSuite ()`.

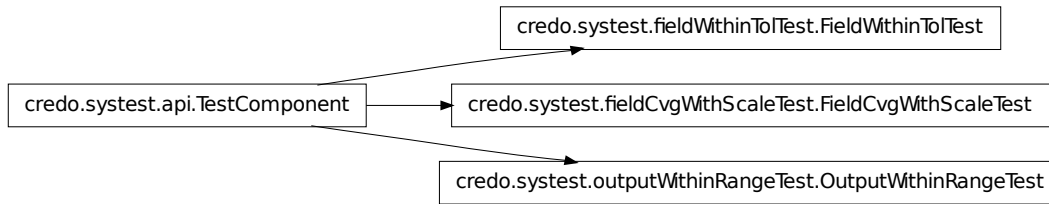
By default will create just a single model run.

Note: for Benchmarks that involve running a suite of models, this API may need re-thinking. Or else a separate `SciBenchmarkSuite` class could be added.

getStatus (*resultsSet*)

See base class `getStatus ()`.

5.6.5 Core TestComponent implementations



`credo.systest.fieldWithinTolTest`

```

class credo.systest.fieldWithinTolTest.FieldWithinTolTest (fieldsToTest=None,
                                                         defFieldTol=0.01, field-
                                                         Tols=None, useRefer-
                                                         ence=False, useHigh-
                                                         ResReference=False,
                                                         referencePath=None,
                                                         testTimestep=0)
    
```

Bases: `credo.systest.api.TestComponent`

Checks whether, for a particular set of fields, the error between each field and an (analytic or reference) solution is below a specified 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()`
- `referencePath`: 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.TestComponent.attachOps()`.

check (*resultsSet*)

Implements base class `credo.systest.api.TestComponent.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.fieldCvgWithScaleTest`

```
class credo.systest.fieldCvgWithScaleTest.FieldCvgWithScaleTest (fieldsToTest=None,
                                                                    calcCvg-
                                                                    Func=<function
                                                                    calcFieldCvg-
                                                                    WithScale      at
                                                                    0x1923320>,
                                                                    field-
                                                                    CvgCrits={ 'VelocityField':
                                                                    (1.6000000000000001,
                                                                    0.9899999999999999),
                                                                    'PressureField':
                                                                    (0.9000000000000002,
                                                                    0.9899999999999999),
                                                                    'recoveredEps-
                                                                    DotField':
                                                                    (1.6000000000000001,
                                                                    0.9899999999999999),
                                                                    'recovered-
                                                                    TauField':
                                                                    (1.6000000000000001,
                                                                    0.9899999999999999),
                                                                    'recovered-
                                                                    PressureField':
                                                                    (1.6000000000000001,
                                                                    0.9899999999999999),
                                                                    'recovered-
                                                                    SigmaField':
                                                                    (1.6000000000000001,
                                                                    0.9899999999999999),
                                                                    'StrainRateField':
                                                                    (0.8499999999999998,
                                                                    0.9899999999999999),
                                                                    'recovered-
                                                                    StrainRateField':
                                                                    (1.6000000000000001,
                                                                    0.9899999999999999)})
```

Bases: `credo.systest.api.TestComponent`

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 `FieldWithinTolTest`, except tests accuracy of solution with increasing 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 criteria to be used when checking the fields. Currently required to be in the form used by the convergence 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 (*modelRun*)

Implements base class `credo.systest.api.TestComponent.attachOps()`.

check (*resultsSet*)

Implements base class `credo.systest.api.TestComponent.check()`.

As well as performing check, will save relevant into to attributes `fErrorsByRun`, `fCvgMeetsReq`, `fCvgResults`.

`credo.systest.fieldCvgWithScaleTest.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.fieldCvgWithScaleTest.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.fieldCvgWithScaleTest.printCvgResult` (*fieldName*, *fieldConvResults*)

`credo.systest.fieldCvgWithScaleTest.testAllCvgWithScale` (*lenScales*, *fieldErrorData*, *fieldCvgCriteria*)

Given a lists of length scales, field error data (a dictionary mapping field names to dofError lists for that field), and field convergence criteria, returns a Bool specifying whether all the fields met their required convergence criteria.

The first two arguments can be created by running `getFieldScaleCvgData_SingleCvgFile()` on a path containing a single cvg file.

`credo.systest.fieldCvgWithScaleTest.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.outputWithinRangeTest`

```
class credo.systest.outputWithinRangeTest.OutputWithinRangeTest (outputName,
                                                                    reductionOp,
                                                                    allowedRange,
                                                                    tRange=None)
```

Bases: `credo.systest.api.TestComponent`

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.

attachOps (*modelRun*)

Implements base class `credo.systest.api.TestComponent.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 (*resultsSet*)

Implements base class `credo.systest.api.TestComponent.check()`.

`credo.systest.imageCompTest`

class `credo.systest.imageCompTest.ImageCompTest` (*imageFilename*, *tol=None*, *refPath=None*,
genPath=None)

Bases: `credo.systest.api.TestComponent`

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.TestComponent.attachOps()` .

check (*resultsSet*)

Implements base class `credo.systest.api.TestComponent.check()` .

LINKS TO USEFUL PYTHON INFO, SUCH AS TUTORIALS

6.1 General Python tutorials / skills

Good introductory 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.

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/>

NOTES FOR CREDO DEVELOPERS / MAINTAINERS

7.1 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.

7.2 Testing framework

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.

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.)

8.1 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 add-on, `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.

8.2 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.

CREDO APPENDIX

9.1 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.

CREDO FAQ

10.1 Problems running tests

10.1.1 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.

10.1.2 Can I get CREDO to submit PBS jobs, or run it via PBS?

The answer to the first of these is: no, not currently. The ability to get CREDO to create and submit PBS scripts is a possible future feature.

You can however submit a Python CREDO script in parallel on a HPC system running PBS by writing the appropriate PBS script.

See *Different ways to launch CREDO scripts* for an example.

10.1.3 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"`.)

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.

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, 55
- credo.analysis.api, 55
- credo.analysis.fields, 56
- credo.analysis.images, 59
- credo.analysis.stats, 59
- credo.io, 48
- credo.io.stgcmdline, 50
- credo.io.stgcvq, 53
- credo.io.stgfreq, 51
- credo.io.stgpath, 54
- credo.io.stgxml, 49
- credo.jobrunner, 47
- credo.jobrunner.api, 47
- credo.jobrunner.mpijobrunner, 48
- credo.modelresult, 42
- credo.modelrun, 38
- credo.modelsuite, 43
- credo.systest, 60
- credo.systest.analytic, 67
- credo.systest.analyticMultiRes, 69
- credo.systest.api, 60
- credo.systest.fieldCvgWithScaleTest, 73
- credo.systest.fieldWithinTolTest, 71
- credo.systest.imageCompTest, 76
- credo.systest.imageReference, 69
- credo.systest.outputWithinRangeTest, 75
- credo.systest.reference, 67
- credo.systest.restart, 68
- credo.systest.scibenchmark, 70
- credo.systest.systestrunner, 65
- credo.utils, 47

INDEX

Symbols

`_absRecordFile` (credo.systest.api.SysTestResult attribute), 63

A

`actualTime` (credo.systest.outputWithinRangeTest.OutputWithinRangeTest attribute), 75

`actualVal` (credo.systest.outputWithinRangeTest.OutputWithinRangeTest attribute), 75

`add()` (credo.analysis.fields.FieldComparisonList method), 57

`addNsPrefix()` (in module credo.io.stgxml), 49

`addRun()` (credo.modelsuite.ModelSuite method), 44

`addStdTest()` (credo.systest.api.SysTestSuite method), 63

`addSubSuite()` (credo.systest.api.SysTestSuite method), 64

`addSubSuites()` (credo.systest.api.SysTestSuite method), 64

`addTestComponent()` (credo.systest.scibenchmark.SciBenchmarkTest method), 70

`addVariant()` (credo.modelsuite.ModelSuite method), 44

`allowedRange` (credo.systest.outputWithinRangeTest.OutputWithinRangeTest attribute), 75

`AnalysisOperation` (class in credo.analysis.api), 55

`analysisOps` (credo.modelrun.ModelRun attribute), 40

`analysisXML` (credo.modelrun.ModelRun attribute), 40

`analysisXMLGen()` (credo.modelrun.ModelRun method), 40

`AnalyticMultiResTest` (class in credo.systest.analyticMultiRes), 69

`AnalyticTest` (class in credo.systest.analytic), 67

API, 87

`applyToModel()` (credo.modelsuite.StgXMLVariant method), 45

`applyToModelGenerator()` (credo.modelsuite.StgXMLVariant method), 45

`attachOps()` (credo.systest.api.TestComponent method), 64

`attachOps()` (credo.systest.fieldCvgWithScaleTest.FieldCvgWithScaleTest method), 69

`attachOps()` (credo.systest.fieldWithinTolTest.FieldWithinTolTest method), 74

method), 72

`attachOps()` (credo.systest.imageCompTest.ImageCompTest method), 76

`attachOps()` (credo.systest.outputWithinRangeTest.OutputWithinRangeTest method), 75

`attachOps()` (credo.systest.outputWithinRangeTest.OutputWithinRangeTest method), 75

`basePath` (credo.modelrun.ModelRun attribute), 39

`basePath` (credo.systest.api.SysTest attribute), 61

Benchmark, 87

C

`calcCvgFunc` (credo.systest.fieldCvgWithScaleTest.FieldCvgWithScaleTest attribute), 74

`calcFieldCvgWithScale()` (in module credo.analysis.fields), 58

`check()` (credo.systest.api.TestComponent method), 64

`check()` (credo.systest.fieldCvgWithScaleTest.FieldCvgWithScaleTest method), 72

`check()` (credo.systest.fieldWithinTolTest.FieldWithinTolTest method), 72

`check()` (credo.systest.imageCompTest.ImageCompTest method), 76

`check()` (credo.systest.outputWithinRangeTest.OutputWithinRangeTest method), 75

`checkAllXMLInputFilesExist()` (in module credo.io.stgpath), 54

`checkNoDuplicates()` (credo.modelrun.SimParams method), 41

`checkParamOverridesTypes()` (in module credo.modelrun), 41

`checkResultValid()` (credo.systest.analytic.AnalyticTest method), 67

`checkResultValid()` (credo.systest.analyticMultiRes.AnalyticMultiResTest method), 69

`checkResultValid()` (credo.systest.api.SysTest method), 62

`checkResultValid()` (credo.systest.imageReference.ImageReferenceTest method), 69

`checkResultValid()` (credo.systest.reference.ReferenceTest method), 68

- checkResultValid() (credo.systest.restart.RestartTest method), 68
- checkResultValid() (credo.systest.scibenchmark.SciBenchmarkTest method), 70
- checkSolverOptsFile() (credo.modelrun.ModelRun method), 40
- checkStgXMLResultsEnabled() (credo.analysis.fields.FieldComparisonList method), 57
- checkType() (credo.modelrun.StgParamInfo method), 41
- checkValidParams() (credo.modelrun.SimParams method), 41
- checkXMLInputFileExists() (in module credo.io.stgpath), 54
- cleanAllLogFiles() (credo.modelsuite.ModelSuite method), 44
- cleanAllOutputPaths() (credo.modelsuite.ModelSuite method), 44
- cmdLineStr() (credo.modelsuite.StgXMLVariant method), 46
- colourDiff() (in module credo.analysis.images), 59
- compare() (in module credo.analysis.images), 59
- constructStGermainRunCommand() (credo.jobrunner.api.JobRunner method), 47
- convertLocalXMLFilesToAbsPaths() (in module credo.io.stgpath), 55
- cpFields (credo.modelrun.ModelRun attribute), 40
- cpReadPath (credo.modelrun.ModelRun attribute), 39
- createFlattenedXML() (in module credo.io.stgxml), 49
- createNewStgDataDoc() (in module credo.io.stgxml), 49
- credo.analysis (module), 55
- credo.analysis.api (module), 55
- credo.analysis.fields (module), 56
- credo.analysis.images (module), 59
- credo.analysis.stats (module), 59
- credo.io (module), 48
- credo.io.stgcmdline (module), 50
- credo.io.stgcvg (module), 53
- credo.io.stgfreq (module), 51
- credo.io.stgpath (module), 54
- credo.io.stgxml (module), 49
- credo.jobrunner (module), 47
- credo.jobrunner.api (module), 47
- credo.jobrunner.mpijobrunner (module), 48
- credo.modelresult (module), 42
- credo.modelrun (module), 38
- credo.modelsuite (module), 43
- credo.systest (module), 60
- credo.systest.analytic (module), 67
- credo.systest.analyticMultiRes (module), 69
- credo.systest.api (module), 60
- credo.systest.fieldCvgWithScaleTest (module), 73
- credo.systest.fieldWithinTolTest (module), 71
- credo.systest.imageCompTest (module), 76
- credo.systest.imageReference (module), 69
- credo.systest.outputWithinRangeTest (module), 75
- credo.systest.reference (module), 67
- credo.systest.restart (module), 68
- credo.systest.scibenchmark (module), 70
- credo.systest.systestrunner (module), 65
- credo.utils (module), 47
- CREDO_ERROR (class in credo.systest.api), 60
- CREDO_FAIL (class in credo.systest.api), 60
- CREDO_PASS (class in credo.systest.api), 60
- CvgFileInfo (class in credo.io.stgcvg), 54
- cvgFileInfo (credo.analysis.fields.FieldComparisonResult attribute), 58
- CVGReadError, 53
- ## D
- DEF_TEST_FIELDS (in module credo.systest.reference), 67
- defaultModelResultFilename() (in module credo.modelresult), 43
- defaultModelRunFilename() (credo.modelrun.ModelRun method), 40
- defaultSysTestFilename() (credo.systest.api.SysTest method), 62
- defFieldTol (credo.systest.fieldWithinTolTest.FieldWithinTolTest attribute), 71
- defVal (credo.modelrun.StgParamInfo attribute), 41
- detailMsg (credo.systest.api.SysTestResult attribute), 63
- dofColMap (credo.io.stgcvg.CvgFileInfo attribute), 54
- dofErrors (credo.analysis.fields.FieldComparisonResult attribute), 58
- ## F
- fComps (credo.systest.fieldCvgWithScaleTest.FieldCvgWithScaleTest attribute), 74
- fComps (credo.systest.fieldWithinTolTest.FieldWithinTolTest attribute), 72
- fCvgMeetsReq (credo.systest.fieldCvgWithScaleTest.FieldCvgWithScaleTest attribute), 74
- fCvgResults (credo.systest.fieldCvgWithScaleTest.FieldCvgWithScaleTest attribute), 74
- fErrorsByRun (credo.systest.fieldCvgWithScaleTest.FieldCvgWithScaleTest attribute), 74
- Field, 87
- FieldComparisonList (class in credo.analysis.fields), 56
- FieldComparisonOp (class in credo.analysis.fields), 57
- FieldComparisonResult (class in credo.analysis.fields), 58
- fieldCvgCrits (credo.systest.fieldCvgWithScaleTest.FieldCvgWithScaleTest attribute), 73
- FieldCvgWithScaleTest (class in credo.systest.fieldCvgWithScaleTest), 73

fieldErrors (credo.systest.fieldWithinTolTest.FieldWithinTolTest attribute), 72

fieldName (credo.analysis.fields.FieldComparisonResult attribute), 58

fieldResults (credo.modelresult.ModelResult attribute), 42

fieldResults (credo.systest.fieldWithinTolTest.FieldWithinTolTest attribute), 72

fields (credo.analysis.fields.FieldComparisonList attribute), 57

fieldsToTest (credo.systest.fieldCvgWithScaleTest.FieldCvgWithScaleTest attribute), 73

fieldsToTest (credo.systest.fieldWithinTolTest.FieldWithinTolTest attribute), 71

fieldTols (credo.systest.fieldWithinTolTest.FieldWithinTolTest attribute), 71

FieldWithinTolTest (class in credo.systest.fieldWithinTolTest), 71

filename (credo.io.stgcvlg.CvgFileInfo attribute), 54

finalStep() (credo.io.stgfreq.FreqOutput method), 51

first() (in module credo.io.stgfreq), 53

FreqOutput (class in credo.io.stgfreq), 31

freqOutput (credo.modelresult.ModelResult attribute), 43

fromXML (credo.analysis.fields.FieldComparisonList attribute), 57

fTestName (credo.systest.analytic.AnalyticTest attribute), 67

fTestName (credo.systest.reference.ReferenceTest attribute), 68

fTestName (credo.systest.restart.RestartTest attribute), 68

G

genConvergenceFileIndex() (in module credo.io.stgcvlg), 54

generateResOpts() (in module credo.modelrun), 41

generateRuns() (credo.modelsuite.ModelSuite method), 44

genFlattenedXML() (credo.modelrun.ModelRun method), 40

genPath (credo.systest.imageCompTest.ImageCompTest attribute), 76

genSuite() (credo.systest.analytic.AnalyticTest method), 67

genSuite() (credo.systest.analyticMultiRes.AnalyticMultiResTest method), 69

genSuite() (credo.systest.api.SysTest method), 62

genSuite() (credo.systest.imageReference.ImageReferenceTest method), 70

genSuite() (credo.systest.reference.ReferenceTest method), 68

genSuite() (credo.systest.restart.RestartTest method), 68

genSuite() (credo.systest.scibenchmark.SciBenchmarkTest method), 70

getAllRecords() (credo.io.stgfreq.FreqOutput method), 51

getAllResults() (credo.analysis.fields.FieldComparisonList method), 57

getCheckStepsRange() (in module credo.io.stgcvlg), 54

getCmpSrcString() (credo.analysis.fields.FieldComparisonList method), 57

getColNum() (credo.io.stgfreq.FreqOutput method), 51

getComparisonOp() (credo.io.stgfreq.FreqOutput method), 51

getDir() (in module credo.modelsuite), 46

getDefaultSubdirs() (in module credo.modelsuite), 46

getDofErrors_ByDof() (in module credo.io.stgcvlg), 54

getDofErrors_ByStep() (in module credo.io.stgcvlg), 54

getDofErrorsByRun() (in module credo.systest.fieldCvgWithScaleTest), 74

getDofErrorsForStep() (in module credo.io.stgcvlg), 54

getElementType() (in module credo.io.stgxml), 49

getFieldScaleCvgData_SingleCvgFile() (in module credo.analysis.fields), 58

getHeaders() (credo.io.stgfreq.FreqOutput method), 51

getItemFromStrSpec_CurrentCtx() (in module credo.io.stgxml), 49

getListNode() (in module credo.io.stgxml), 49

getMax() (credo.io.stgfreq.FreqOutput method), 51

getMean() (credo.io.stgfreq.FreqOutput method), 51

getMin() (credo.io.stgfreq.FreqOutput method), 52

getModelResultsArray() (in module credo.modelsuite), 46

getNodeFromStrSpec() (in module credo.io.stgxml), 49

getNumDofs() (in module credo.systest.fieldCvgWithScaleTest), 74

getOutputSubPath() (credo.modelsuite.ModelSuite method), 45

getParam() (credo.modelrun.SimParams method), 41

getParamNode() (in module credo.io.stgxml), 49

getParamOverridesAsStr() (in module credo.modelrun), 41

getParamValue() (in module credo.io.stgxml), 49

getParamValues() (in module credo.modelsuite), 46

getParamValuesIter() (in module credo.modelsuite), 46

getRecordAtStep() (credo.io.stgfreq.FreqOutput method), 52

getRecordDictAtStep() (credo.io.stgfreq.FreqOutput method), 52

getRecordFile() (credo.systest.api.SysTestResult method), 63

getRecordNum() (credo.io.stgfreq.FreqOutput method), 52

getReductionOp() (credo.io.stgfreq.FreqOutput method), 52

getRes() (in module credo.io.stgcvlg), 54

getResult() (credo.analysis.fields.FieldComparisonOp method), 58

- getResultsSubset() (credo.systest.api.SysTest method), 62
 - getResultsTotals() (credo.systest.systestrunner.SysTestRunner method), 65
 - getRunByName() (credo.modelsuite.ModelSuite method), 45
 - getRunIndex() (credo.modelsuite.ModelSuite method), 45
 - getSimInfoFromFreqOutput() (in module credo.modelresult), 43
 - getStatus() (credo.systest.api.SysTest method), 62
 - getStatus() (credo.systest.scibenchmark.SciBenchmarkTest method), 70
 - getStdErrFilename() (credo.modelrun.ModelRun method), 40
 - getStdOutFilename() (credo.modelrun.ModelRun method), 40
 - getStdTestName() (in module credo.systest.api), 64
 - getStdTestNameBasic() (in module credo.systest.api), 64
 - getStgBinPath() (in module credo.io.stgpath), 55
 - getStGermainExeStr() (credo.jobrunner.api.JobRunner method), 47
 - getStGermainExeStr() (credo.jobrunner.mpijobrunner.MPIJobRunner method), 48
 - getStgStandardXMLPath() (in module credo.io.stgpath), 55
 - getStructNode() (in module credo.io.stgxml), 49
 - getSuiteResultsFilename() (credo.systest.systestrunner.SysTestRunner method), 65
 - getSuitesFromModules() (in module credo.systest.systestrunner), 66
 - getTimeStepMap() (credo.io.stgfreq.FreqOutput method), 52
 - getTimeStepsArray() (credo.io.stgfreq.FreqOutput method), 52
 - getTolForField() (credo.systest.fieldWithinTolTest.FieldWithinTolTest method), 72
 - getValueAtStep() (credo.io.stgfreq.FreqOutput method), 52
 - getValuesArray() (credo.io.stgfreq.FreqOutput method), 52
 - getVariantCmdLineOverrides() (in module credo.modelsuite), 46
 - getVariantIndicesIter() (in module credo.modelsuite), 46
 - getVariantNameDicts() (in module credo.modelsuite), 46
 - getVariantParamPathDicts() (in module credo.modelsuite), 46
 - getVerifyStgExePath() (in module credo.io.stgpath), 55
 - getVerifyStgMainExecutablePath() (in module credo.io.stgpath), 55
- ## H
- headerColMap (credo.io.stgfreq.FreqOutput attribute), 51
- ## I
- ImageCompTest (class in credo.systest.imageCompTest), 76
 - imageErrors (credo.systest.imageCompTest.ImageCompTest attribute), 76
 - imageFilename (credo.systest.imageCompTest.ImageCompTest attribute), 76
 - ImageReferenceTest (class in credo.systest.imageReference), 69
 - imageResults (credo.systest.imageCompTest.ImageCompTest attribute), 76
 - indentForPrettyPrint() (in module credo.io.stgxml), 50
 - Input File, 87
 - inputFiles (credo.systest.api.SysTest attribute), 61
 - insertNamedElementNode() (in module credo.io.stgxml), 50
- ## J
- JobMetaInfo (class in credo.modelresult), 42
 - jobMetaInfo (credo.modelresult.ModelResult attribute), 42
 - JobParams (class in credo.modelrun), 38
 - jobParams (credo.modelrun.ModelRun attribute), 39
 - JobRunner (class in credo.jobrunner.api), 47
- ## L
- last() (in module credo.io.stgfreq), 53
 - linreg() (in module credo.analysis.stats), 59
 - logPath (credo.modelrun.ModelRun attribute), 39
- ## M
- maxRunTime (credo.jobrunner.api.ModelRunTimeoutError attribute), 48
 - Mesh, 87
 - Model, 87
 - Model Result, 87
 - Model Run, 87
 - modelInputFiles (credo.modelrun.ModelRun attribute), 39
 - modelName (credo.modelresult.ModelResult attribute), 42
 - ModelResult (class in credo.modelresult), 42
 - ModelRun (class in credo.modelrun), 38
 - ModelRunError, 48
 - ModelRunLaunchError, 48
 - ModelRunRegularError, 48
 - ModelRunTimeoutError, 48
 - ModelSuite (class in credo.modelsuite), 43
 - modelVariants (credo.modelsuite.ModelSuite attribute), 44
 - moveAllToTargetPath() (in module credo.io.stgpath), 55
 - MPIJobRunner (class in credo.jobrunner.mpijobrunner), 48

mSuite (credo.systest.api.SysTest attribute), 61

N

name (credo.analysis.fields.FieldComparisonOp attribute), 58

name (credo.modelrun.ModelRun attribute), 39

navigateStrSpecHierarchy() (in module credo.io.stgxml), 50

newSubSuite() (credo.systest.api.SysTestSuite method), 64

normalise() (in module credo.analysis.images), 59

nproc (credo.modelrun.JobParams attribute), 38

nproc (credo.systest.api.SysTest attribute), 61

nproc (credo.systest.api.SysTestSuite attribute), 63

O

outputName (credo.systest.outputWithinRangeTest.OutputWithinRangeTest attribute), 75

outputPath (credo.modelresult.ModelResult attribute), 42

outputPath (credo.modelrun.ModelRun attribute), 39

outputPathBase (credo.modelsuite.ModelSuite attribute), 44

outputPathBase (credo.systest.api.SysTest attribute), 61

OutputWithinRangeTest (class in credo.systest.outputWithinRangeTest), 75

P

paramOverrides (credo.modelrun.ModelRun attribute), 39

paramOverrides (credo.systest.api.SysTest attribute), 61

paramPath (credo.modelsuite.StgXMLVariant attribute), 45

paramRange (credo.modelsuite.StgXMLVariant attribute), 45

paramStr() (in module credo.io.stgcmdline), 50

pixelDiff() (in module credo.analysis.images), 59

pixelDiff2x2() (in module credo.analysis.images), 59

plotOverTime() (credo.analysis.fields.FieldComparisonResult method), 58

plotOverTime() (credo.io.stgfreq.FreqOutput method), 52

plottedCvgFilename (credo.analysis.fields.FieldComparisonResult attribute), 58

populated (credo.io.stgfreq.FreqOutput attribute), 51

populateFromFile() (credo.io.stgfreq.FreqOutput method), 53

postRun() (credo.analysis.api.AnalysisOperation method), 56

postRun() (credo.analysis.fields.FieldComparisonList method), 57

postRunCleanup() (credo.modelrun.ModelRun method), 40

prepareOutputLogDirs() (credo.modelrun.ModelRun method), 40

printAllMinMax() (credo.io.stgfreq.FreqOutput method), 53

printCvgResult() (in module credo.systest.fieldCvgWithScaleTest), 74

printDetailMsg() (credo.systest.api.SysTestResult method), 63

printResultsDetails() (credo.systest.systestrunner.SysTestRunner method), 65

printResultsSummary() (credo.systest.systestrunner.SysTestRunner method), 65

printSuiteResultsByProject() (credo.systest.systestrunner.SysTestRunner method), 65

printSuiteResultsOrderFound() (credo.systest.systestrunner.SysTestRunner method), 65

printSuiteTotalsShortSummary() (credo.systest.systestrunner.SysTestRunner method), 65

productCalc() (in module credo.utils), 47

projectName (credo.systest.api.SysTestSuite attribute), 63

pType (credo.modelrun.StgParamInfo attribute), 41

R

readFrequentOutput() (credo.modelresult.ModelResult method), 43

readFromStgXML() (credo.analysis.fields.FieldComparisonList method), 57

readFromStgXML() (credo.modelrun.SimParams method), 41

readResultsFromPath() (credo.modelsuite.ModelSuite method), 45

recordFieldResult() (credo.modelresult.ModelResult method), 43

reductionOp (credo.systest.outputWithinRangeTest.OutputWithinRangeTest attribute), 75

referencePath (credo.analysis.fields.FieldComparisonList attribute), 57

ReferenceTest (class in credo.systest.reference), 67

refPath (credo.systest.imageCompTest.ImageCompTest attribute), 76

removeNsPrefix() (in module credo.io.stgxml), 50

resIndicesToTest (credo.systest.api.SysTest attribute), 61

resSet (credo.systest.analyticMultiRes.AnalyticMultiResTest attribute), 69

RestartTest (class in credo.systest.restart), 68

resultsList (credo.modelsuite.ModelSuite attribute), 44

runCustomOptSets (credo.modelsuite.ModelSuite attribute), 44

runDescripts (credo.modelsuite.ModelSuite attribute), 44

runExecCmd (credo.jobrunner.api.ModelRunLaunchError attribute), 48

runModel() (credo.jobrunner.api.JobRunner method), 47

runModel() (credo.jobrunner.mpijobrunner.MPIJobRunner method), 48
 runs (credo.modelsuite.ModelSuite attribute), 44
 runSuite() (credo.jobrunner.api.JobRunner method), 47
 runSuite() (credo.systest.systestrunner.SysTestRunner method), 65
 runSuites() (credo.systest.systestrunner.SysTestRunner method), 65
 runSuitesFromModules() (in module credo.systest.systestrunner), 66
 runTest() (credo.systest.systestrunner.SysTestRunner method), 65
 runTests() (credo.systest.systestrunner.SysTestRunner method), 66

S

SciBenchmarkTest (class in credo.systest.scibenchmark), 70
 setAllTimeouts() (credo.systest.api.SysTestSuite method), 64
 setErrorStatus() (credo.systest.api.SysTest method), 62
 setMergeType() (in module credo.io.stgxml), 50
 setParam() (credo.modelrun.SimParams method), 41
 setPath() (credo.modelrun.ModelRun method), 40
 setRecordFile() (credo.systest.api.SysTestResult method), 63
 setResIndicesToTest() (credo.systest.api.SysTest method), 62
 setTimeout() (credo.systest.api.SysTest method), 62
 setup() (credo.jobrunner.api.JobRunner method), 47
 setup() (credo.jobrunner.mpijobrunner.MPIJobRunner method), 48
 setup() (credo.systest.api.SysTest method), 62
 setup() (credo.systest.imageReference.ImageReferenceTest method), 70
 setup() (credo.systest.reference.ReferenceTest method), 68
 SimParams (class in credo.modelrun), 41
 simParams (credo.modelrun.ModelRun attribute), 39
 simtime (credo.modelresult.JobMetaInfo attribute), 42
 solverOpts (credo.modelrun.ModelRun attribute), 39
 solverOpts (credo.systest.api.SysTest attribute), 61
 solverOptsStr() (in module credo.io.stgcmdline), 50
 statusStr (credo.systest.api.SysTestResult attribute), 63
 StGermain, 87
 stgName (credo.modelrun.StgParamInfo attribute), 41
 StgParamInfo (class in credo.modelrun), 41
 stgParamInfos (credo.modelrun.SimParams attribute), 41
 StgXMLVariant (class in credo.modelsuite), 45
 strRes() (in module credo.modelrun), 41
 strToBool() (in module credo.io.stgxml), 50
 subOutputPathGenFunc (credo.modelsuite.ModelSuite attribute), 44
 subSuites (credo.systest.api.SysTestSuite attribute), 63

suiteName (credo.systest.api.SysTestSuite attribute), 63
 Swarm, 87
 System Test, 87
 SysTest (class in credo.systest.api), 61
 SysTestResult (class in credo.systest.api), 63
 SysTestRunner (class in credo.systest.systestrunner), 65
 sysTests (credo.systest.api.SysTestSuite attribute), 63
 SysTestSetupError, 63
 SysTestSuite (class in credo.systest.api), 63

T

tcStatus (credo.systest.api.TestComponent attribute), 64
 tcType (credo.systest.api.TestComponent attribute), 64
 templateMRun (credo.modelsuite.ModelSuite attribute), 44
 testAllCvgWithScale() (in module credo.systest.fieldCvgWithScaleTest), 74
 TestComponent (class in credo.systest.api), 64
 testComponents (credo.systest.api.SysTest attribute), 61
 testCvgWithScale() (in module credo.systest.fieldCvgWithScaleTest), 74
 testName (credo.systest.api.SysTest attribute), 61
 testStatus (credo.systest.api.SysTest attribute), 61
 testTimestep (credo.analysis.fields.FieldComparisonList attribute), 57
 testType (credo.systest.api.SysTest attribute), 61
 timeout (credo.systest.api.SysTest attribute), 61
 tol (credo.systest.imageCompTest.ImageCompTest attribute), 76
 tRange (credo.systest.outputWithinRangeTest.OutputWithinRangeTest attribute), 75

U

Underworld, 87
 updateModelResultsXMLFieldInfo() (in module credo.modelresult), 43
 updateXMLWithResult() (credo.systest.api.SysTest method), 62
 updateXMLWithResult() (credo.systest.api.TestComponent method), 64
 useHighResReference (credo.analysis.fields.FieldComparisonList attribute), 57
 useReference (credo.analysis.fields.FieldComparisonList attribute), 57

V

valLen() (credo.modelsuite.StgXMLVariant method), 46
 valStr() (credo.modelsuite.StgXMLVariant method), 46
 Virtual method, 87
 VRMS, 87

W

55

- withinRange (credo.systest.outputWithinRangeTest.OutputWithinRangeTest attribute), 75
- withinTol() (credo.analysis.fields.FieldComparisonResult method), 58
- writeAllModelResultXMLs()
(credo.modelsuite.ModelSuite method), 45
- writeAllModelRunXMLs()
(credo.modelsuite.ModelSuite method), 45
- writeInfoXML() (credo.analysis.api.AnalysisOperation method), 56
- writeInfoXML() (credo.analysis.fields.FieldComparisonList method), 57
- writeInfoXML() (credo.analysis.fields.FieldComparisonOp method), 58
- writeInfoXML() (credo.analysis.fields.FieldComparisonResult method), 58
- writeInfoXML() (credo.modelresult.JobMetaInfo method), 42
- writeInfoXML() (credo.modelrun.JobParams method), 38
- writeInfoXML() (credo.modelrun.ModelRun method), 40
- writeInfoXML() (credo.modelrun.SimParams method), 41
- writeInputsOutputsToCSV() (in module credo.modelsuite), 46
- writeMergeComponent() (in module credo.io.stgxml), 50
- writeParam() (in module credo.io.stgxml), 50
- writeParamList() (in module credo.io.stgxml), 50
- writeParamOverridesInfoXML() (in module credo.modelrun), 41
- writeParamSet() (in module credo.io.stgxml), 50
- writePreRunXML() (credo.systest.api.SysTest method), 62
- writePreRunXML() (credo.systest.api.TestComponent method), 64
- writeRecordXML() (credo.modelresult.ModelResult method), 43
- writeSolverOptsInfoXML() (in module credo.modelrun), 42
- writeStgDataDocToFile() (in module credo.io.stgxml), 50
- writeStgDataXML() (credo.analysis.api.AnalysisOperation method), 56
- writeStgDataXML() (credo.analysis.fields.FieldComparisonList method), 57
- writeStgDataXML() (credo.modelrun.SimParams method), 41
- writeValueUsingStrSpec() (in module credo.io.stgxml), 50
- writeXMLDoc() (in module credo.io.stgxml), 50

X

- xmlExistsInStdXMLPath() (in module credo.io.stgpath),