BBS end-to-end tests

This page describes the collection of python scripts that can be used to perform typical BBS runs on reference Measurementsets. These perform regular high-level BBS operations, e.g. PREDICT, SOLVE, CORRECT in parsets that mimic data reduction examples.

Due to size constraints in the test environment the sample MS have been DPPP'ed to single channel subbands, and also been cut to ca. 15 minutes, in order to keep file sizes and test execution times manageable.

The test scripts can be found on the CEP2 cluster in /globaldata/bbs/tests/scripts, while the sample Measurementsets reside in /globaldata/bbs/tests/MS.

Note
These scripts are part of the lofar package *BBSTools*.

Motivation

While BBS' internal functions are usually tested through unit tests of its individual classes, real-world data reduction requires precise interaction of all components which can reveal glitches that are not found in daily unit tests.
Therefore, data reduction parsets are applied to observational data, and the output (parameters and casa table columns) are compared to reference files. The inspection of the data is done automatically by value comparison with a user-definable numerical accuracy.

bbstest Python class

The bbstest python class which part of the lofar.bbs module, is a high-level interface to create typical test examples. It runs on unmodified parsets and handles test reference Measurementsets. It inherits from the lofar.bbs.testsip python class which handles common test procedures and is also the basis for NDPPP and Imager test cases.

The test environment is set up fully automatically by interpreting the parset that defines the data reduction. Depending on the parameters solved for and the output columns that are written to the test MS, checks against the reference MS are performed.

A typical run looks like this: ./testBBS_3C196_calibration.py

Execute test ./testBBS_3C196_calibration.py
Creating GDS file
Reading parms from parset
Reading columns from parset
Current BBS test settings
MS = ../MS/L24380_SB030_uv.MS.dppp.dppp.cut
Parset = uv-plane-cal.parset
Skymodel = 3C196-bbs_2sources.skymodel

```
test_MS = ../MS/test_L24380_SB030_uv.MS.dppp.dppp.cut
gds = ../MS/test_L24380_SB030_uv.MS.dppp.dppp.cut.gds
wd = .
host = Sven-Duschas-Macbook-Pro
clusterdesc = /Users/duscha/Cluster/Config/mbp.clusterdesc
dbserver = localhost
parms = ['Gain:0:0:*', 'Gain:1:1:*']
columns = ['CORRECTED_DATA']
acceptancelimit = 0.001

Comparing parmDB parameters in test MS ../MS/test_L24380_SB030_uv.MS.dppp.dppp.cut and reference MS ../MS/L24380_SB030_uv.MS.dppp.dppp.cut
Comparing parameters [########################################] 188/188
Comparing CORRECTED_DATA columns.
Forwarding reference column CORRECTED_DATA to ../MS/test_L24380_SB030_uv.MS.dppp.dppp.cut
Successful read/write open of default-locked table ../MS/test_L24380_SB030_uv.MS.dppp.dppp.cut: 27 columns, 510984 rows
Successful readonly open of default-locked table ../MS/L24380_SB030_uv.MS.dppp.dppp.cut: 27 columns, 510984 rows
Detailed test results:
Test CORRECTED_DATA passed.
Test ParmDB passed.
Test ./testBBS_3C196_calibration.py passed.
```

The output can be switched off (almost entirely - with the exception of pyrap.tables and TaQL messages) by setting verbose to False. The output is written to stdout, but can be redirected to a file in the shell, if a log of the test is meant to be kept.

Notes

The following additional test parameters can be set:

* verbose = True | False (default = *True*)
verbose test output
* taql= True | False (default = *True*)
use TaQL to compare columns, does not provide a progressbar and uses a relative error limit for comparison. Numpy uses an absolute error criterion (acceptancelimit)
* acceptancelimit = 10e-3 (= default value)
absolute difference error between reference and test parameter/column entry up to which the test is declared as passed.

Writing test case scripts

Having a reference MS, a parset that describes the data reduction and providing a skymodel the corresponding BBS run is easily tested automatically:

The python class **lofar.bbs.testbbs** provides a test constructor, individual test procedures and a high-level function executeTest() that performs a complete test with data evaluation.

Note: testbbs works up to now only on single subband Measurementsets, but creates the corresponding vds and gds files automatically. This allows to write a test case in the following way:

```
#!/usr/bin/env python
import lofar.bbs.testbbs as bbs

MS = "../MS/L24380_SB030_uv.MS.dppp.dppp.cut"
parset = "uv-plane-cal.parset"
skymodel = "3C196-bbs_2sources.skymodel"
verbose=True

test = bbs.testbbs(MS, parset, skymodel, verbose=verbose)
test.executeTest()
```

Available test scripts

Currently the following python test scripts are available in /globaldata/bbs/tests/

* calibration/testBBS_3C196_calibration.py

performs a simple electronic gain calibration on a 3C196 two-source-model, solves for the solution parameters and writes it to the CORRECTED_DATA column.

* simulation/testBBS_3C196_simulation.py

simulates 3C196 two-source-model UV data, using BBS PREDICT and writes it to the MODEL_DATA. Here no parameters are solved for.

* directional/testBBS_A-Team_directional.py

This is a somewhat lengthy test (takes ca. 40 minutes): It solves for CygA and CasA parameters, substracts these two sources and writes the substracted uv data to a SUBSTRACTED column in the MS. In a second step it corrects for 3C196.

bbstests.sh high-level test script

A collection of typcial bbstest scripts is bundled into a (bash) shell script which can be run to cover the most common test case scenarios.

These can be collectively run in the below mentioned **bbstests.sh** bash script through

bbstests.sh all

or indivdually through specifying *simulation*, *calibration*, *directional* as a list of command arguments.

–*help* will give usage information showing these options in detail, and –*verbose* turns verbose test information on.
In order to perform this test, log on to any of the CEP2 nodes. The script will copy the necessary reference Measurementsets to local storage on the node, run the test suite which internally creates a working copy of the MS (which after the test run is deleted). The reference MS is kept on local storage, so that on successive test runs no additional network copy has to be done.