Dynamic Gravity Processor Documentation

Release 0.1a

Daniel Aliod, Chris Bertinato, Zachery Brady

Sep 25, 2018

Getting Started

CHAPTER 1

What is DGP?

DGP is a library as well a graphical desktop application for processing gravity data collected with dynamic gravity systems, such as those run on ships and aircraft.

The library can be used to automate the processing workflow and experiment with new techniques. The application was written to fulfill the needs of of gravity processing in production environments.

The project aims to bring all gravity data processing under a single umbrella by:

- accommodating various sensor types, data formats, and processing techniques
- providing a flexible framework to allow for experimentation with the workflow
- providing a robust and efficient system for production-level processing

1.1 Core Dependencies

(Subject to change)

- Python >= 3.6
- numpy >= 1.13.1
- pandas == 0.20.3
- scipy == 1.1.0
- pyqtgraph >= 0.10.0
- PyQt5 >= 5.10
- PyTables >= 3.4.2

1.1.1 Installation

TODO: All

1.1.2 User Guide

Todo: Write documentation/tutorial on how to use the application, targeted at actual users, not developers.

Creating a new project

Project Structure (Airborne)

An Airborne gravity project in DGP is centered primarily around the `Flight` construct as a representation of an actual survey flight. A flight has at least one `DataSet` containing Trajectory (GPS) and Gravity data files, and at least one associated `Gravimeter`.

A `Flight` may potentially have more than one `DataSet` associated with it, and more than one `Gravimeter`.

Each DataSet has exactly one Trajectory and one Gravity DataFile contained within it, and the `DataSet` may define `DataSegments` which are directly associated with the encapsulated files.

DataSegments are used to select areas of data which are of interest for processing, typically this means they are used to select the individual Flight Lines out of a continuous data file, i.e. the segments between course changes of the aircraft.

Creating Flights/Survey's

Importing Gravimeter (Sensor) Configurations

Importing Gravity/Trajectory (GPS) Data

Data Processing Workflow

Selecting Survey Lines

Selecting/Applying Transformation Graphs

Viewing Line Repeats

1.1.3 dgp.core package

Core modules and packages defining the project data-layer and controllers for interfacing with the data-layer via the user interface.

dgp.core.models package

The models package contains and defines the various data classes that define the logical structure of a 'Gravity Project'

Currently we are focused exclusively on providing functionality for representing and processing an Airborne gravity survey/campaign. In future support will be added for processing and managing Marine gravity survey's/campaigns.

The following generally describes the class hierarchy of a typical Airborne project:

`AirborneProject`

```
├── Flight
│ ├── DataSet
│ │ ├── DataFile – Gravity
│ │ ├── DataFile – Trajectory
│ │ └── DataSegment – Container (Multiple)
│ └── Gravimeter – Link
└── Gravimeter
```

The project can have multiple `Flight`, and each Flight can have 0 or more `FlightLine`, `DataFile`, and linked `Gravimeter`. The project can also define multiple Gravimeters, of varying type with specific configuration files assigned to each.

Model Development Principles

- Classes in the core models should be kept as simple as possible.
- `@properties` (getter/setter) are encouraged where state updates must accompany a value change
- Otherwise, simple attributes/fields are preferred
- Models may contain back-references (upwards in the hierarchy) only to their parent (using the 'magic' parent attribute) - otherwise the JSON serializer will complain.
- Any complex functions/transformations should be handled by the model's controller
- Data validation should be handled by the controller, not the model.
- A limited set of complex objects can be used and serialized in the model, support may be added as the need arises in the JSON serializer.
- Any field defined in a model's `__dict__` or `__slots__` is serialization by the ProjectEncoder, and consequently must be accepted by name (keyword argument) in the model constructor for de-serialization

Supported Complex Types

- `pathlib.Path`
- `datetime.datetime`
- `datetime.date`
- `dgp.core.oid.OID`
- All classes in *`dgp.core.models`*

See `ProjectDecoder` and `ProjectEncoder` for implementation details.

Contents

dgp.core.models.project module

Project Serialization/De-Serialization Classes

dgp.core.models.meter module

New in version 0.1.0.

dgp.core.models.flight module

dgp.core.models.datafile module

dgp.core.models.dataset module

New in version 0.1.0.

dgp.core.controllers package

The Controllers package contains the various controller classes which are layered on top of the core 'data models' (see the *dgp.core.models package*) which themselves store the raw project data.

The function of the controller classes is to provide an interaction layer on top of the data layer - without complicating the underlying data classes, especially as the data classes must undergo serialization and de-serialization.

The controllers provide various functionality related to creating, traversing, and mutating the project tree-hierarchy. The controllers also interact in minor ways with the UI, and more importantly, are the layer by which the UI interacts with the underlying project data.

TODO: Add Controller Hierarchy like in models.rst

Controller Development Principles

Controllers typically should match 1:1 a model class, though there are cases for creating controllers such as the `ProjectFolder` which is a utility class for grouping items visually in the project's tree view.

Controllers should at minimum subclass `VirtualBaseController` which configures inheritance for `QStandardItem` and `AttributeProxy`. For more complex and widely used controllers, a dedicated interface should be created following the same naming scheme - particularly where circular dependencies may be introduced.

Context Menu Declarations

Due to the nature of `QMenu`, the menu cannot be instantiated directly ahead of time as it requires a parent `QWidget` to bind to. This has led to the current solution which lets each controller declaratively define their context menu items and actions (with some common actions mixed in by the view at runtime). The declaration syntax at present is simply a list of tuples which is queried by the view when a context menu is requested.

Following is an example declaring a single menu item to be displayed when right-clicking on the controller's representation in the UI

```
bindings = [
    ('addAction', ('Properties', lambda: self._show_properties())),
]
```

The menu is built by iterating through the bindings list, each 2-tuple is a tuple of the QMenu function to call ('addAction'), and the positional arguments supplied to the function - in this case the name 'Properties', and the lambda functor to call when activated.

Contents

Interfaces

The following interfaces provide interface definitions for the various controllers used within the overall project model.

The interfaces, while perhaps not exactly Pythonic, provide great utility in terms of type safety in the interaction of the various controllers. In most cases the concrete subclasses of these interfaces cannot be directly imported into other controllers as this would cause circular import loops

e.g. the `FlightController` is a child of an `AirborneProjectController`, but the FlightController also stores a typed reference to its parent (creating a circular reference), the interfaces are designed to allow proper type hinting within the development environment in such cases.

Controllers

Concrete controller implementations

Containers

Utility/Helper Modules

dgp.core.types package

Stuff about types

Sub Modules

dgp.core.file_loader module

dgp.core.oid module

1.1.4 dgp.lib package

This package contains library functions and utilities for loading, processing, and transforming gravity and trajectory data.

dgp.lib.gravity_ingestor module

dgp.lib.time_utils module

dgp.lib.trajectory_ingestor module

1.1.5 dgp.gui package

This package contains modules and sub-packages related to the Graphical User Interface (GUI) presentation layer of DGP.

DGP's User Interface is built on the Qt 5 C++ library, using the PyQt Python bindings.

Custom Qt Views, Widgets, and Dialogs are contained here, as well as plotting interfaces.

Qt Interfaces and Widgets created with Qt Creator generate .ui XML files, which are then compiled into a Python source files which define individual UI components. The .ui source files are contained within the ui directory.

dgp.gui.plotting package

The plotting package contains the backend wrappers and classes used by the DGP application to interactively plot data within the GUI.

The interactive plotting framework that we utilize here is based on the PyQtGraph python package, which itself utilizes the Qt Graphics View Framework to provide a highly performant interactive plotting interface.

The modules within the plotting package are separated into the *Bases*, *Plotters* and *Helpers* modules, which provide the base plot wrappers, task/application specific plot widgets, and plot utility functions/ classes respectively.

The *Bases* module defines the base plot wrappers which wrap some of PyQtGraph's plotting functionality to ease the plotting and management of Pandas Series data within a plot surface.

The *Plotters* module provides task specific plot widgets that can be directly incorporated into a QtWidget application's layout. These classes add specific functionality to the base 'backend' plots, for example to enable graphical click-drag selection of data segments by the user.

Types/Consts/Enums

backends.**MaybePlot = Union[DgpPlotItem, None]**
Typedef for a function which returns a DgpPlotItem or None

backends.**MaybeSeries = Union[pandas.Series, None]**
Typedef for a function which returns a pandas.Series or None

backends.**SeriesIndex = Tuple[str, int, int, Axis]**
Typedef for a tuple representing the unique index of a series on a plot within a GridPlotWidget

Bases

Plotters

Helpers

dgp.gui.workspaces package

The Workspaces sub-package defines GUI widgets for various controller contexts in the DGP application. The idea being that there are naturally different standard ways in which the user will interact with different project objects/controllers, depending on the type of the object.

The workspaces are intended to be displayed within a QTabWidget within the application so that the user may easily navigate between multiple open workspaces.

Each workspace defines its own custom widget(s) for interacting & manipulating data associated with its underlying controller (VirtualBaseController).

Workspaces may also contain sub-tabs, for example the DataSetTab defines sub-tabs for viewing raw-data and selecting segments, and a tab for executing transform graphs on the data.

Contents

Base Interfaces

New in version 0.1.0.

Workspaces

Project Workspace

> **Warning:** Not yet implemented

Note: Future Planning: Project Workspace may display a map interface which can overlay each flight's trajectory path from the flights within the project. Some interface to allow comparison of flight data may also be integrated into this workspace.

Flight Workspace

> **Warning:** Not yet implemented

Note: Future Planning: Similar to the project workspace, the flight workspace may be used to display a map of the selected flight. A dashboard type widget may be implemented to show details of the flight, and to allow users to view/configure flight specific parameters.

DataSet Workspace

New in version 0.1.0.

DataFile Workspace

> **Warning:** Not yet implemented

Note: Future Planning: The DataFile workspace may be used to allow users to view and possibly edit raw data within the interface in a spreadsheet style view/control.

See also:

Qt 5 Documentation

PyQt5 Documentation

1.1.6 Data Management in DGP

DGP manages and interacts with a variety of forms of Data. Imported raw data (GPS or Gravity) is ingested and maintained internally as a `pandas.DataFrame` or `pandas.Series` from their raw representation in comma

separated value (CSV) files. The ingestion process performs type-casts, filling/interpolation of missing values, and time index creation/conversion functions to result in a ready-to-process DataFrame.

These DataFrames are then stored in the project's HDF5 data-file, which natively supports (with PyTables and Pandas) the storage and retrieval of DataFrames and Series.

To facilitate storage and retrieval of data within the project, the `HDF5Manager` class provides an easy to use wrapper around the `pandas.HDFStore` and provides utility methods for getting/setting meta-data attributes on nodes.

1.1.7 Contributing

Creating a branch

This project uses the GitFlow branching model. The `master` branch reflects the current production-ready state. The `develop` branch is a perpetual development branch.

New development is done in feature branches which are merged back into the `develop` branch once development is completed. Prior to a release, a release branch is created off of `develop`. When the release is ready, the release branch is merged into `master` and `develop`.

Development branches are named with a prefix according to their purpose:

- `feature/`: An added feature or improved functionality.
- `bug/`: Bug fix.
- `doc/`: Addition or cleaning of documentation.
- `clean/`: Code clean-up.

When starting a new branch, be sure to branch from develop:

```
$ git checkout -b my_feature develop
```

Keep any changes in this branch specific to one bug or feature. If the develop branch has advanced since your branch was first created, then you can update your branch by retrieving those changes from the develop branch:

```
$ git fetch origin
$ git rebase origin/develop
```

This will replay your commits on top of the latest version of the develop branch. If there are merge conflicts, then you must resolve them.

Committing your code

When committing to your changes, we recommend structuring the commit message in the following way:

- subject line with less than < 80 chars
- one blank line
- optionally, a commit message body

Please reference the relevant GitHub issues in your commit message using GH1234 or #1234.

For the subject line, this project uses the same convention for commit message prefix and layout as the Pandas project. Here are some common prefixes and guidelines for when to use them:

- ENH: Enhancement, new functionality
- BUG: Bug fix

- DOC: Additions/updates to documentation
- TST: Additions/updates to tests
- BLD: Updates to the build process/scripts
- PERF: Performance improvement
- CLN: Code cleanup

Combining commits

When you're ready to make a pull request and if you have made multiple commits, then you may want to combine, or "squash", those commits. Squashing commits helps to maintain a compact commit history, especially if a number of commits were made to fix errors or bugs along the way. To squash your commits:

```
git rebase -i HEAD~#
```

where # is the number of commits you want to combine. If you want to squash all commits on the branch:

```
git rebase -i --root
```

Then you will need to push the branch forcefully to replace the current commits with the new ones:

```
git push origin new-feature -f
```

Incorporating a finished feature on develop

Finished features should be added to the develop branch to be included in the next release:

```
$ git checkout develop
Switched to branch 'develop'
$ git merge --no-ff myfeature
Updating ea1b82a..05e9557
(summary of changes)
$ git branch -d myfeature
Deleted branch myfeature (was 05e9557).
$ git push origin develop
```

The `--no-ff` flag causes the merge to always create a commit, even if it can be done with a fast-forward. This way we record the existence of the feature branch even after it has been deleted, and it groups all of the relevant commits for this feature.

Note that pull-requests into develop require passing Continuous Integration (CI) builds on Travis.ci and AppVeyor, and at least one approved review.

Code standards

DGP uses the PEP8 standard. In particular, that means:

- we restrict line-length to 79 characters to promote readability
- passing arguments should have spaces after commas, *e.g.*, `foo(arg1, arg2, kw1='bar')`

Continuous integration will run the flake8 tool to check for conformance with PEP8. Therefore, it is beneficial to run the check yourself before submitting a pull request:

```
git diff master --name-only -- '*.py' | flake8 --diff
```

Test-driven development

All new features and added functionality will require new tests or amendments to existing tests, so we highly recommend that all contributors embrace test-driven development (TDD).

All tests should go to the `tests` subdirectory. We suggest looking to any of the examples in that directory to get ideas on how to write tests for the code that you are adding or modifying.

DGP uses the pytest framework for unit testing and coverage.py to gauge the effectiveness of tests by showing which parts of the code are being executed by tests, and which are not. The pytest-cov extension is used in conjunction with Py.Test and coverage.py to generate coverage reports after executing the test suite.

Continuous integration will also run the test-suite with coverage, and report the coverage statistics to Coveralls

Running the test suite

The test suite can be run from the repository root:

```
pytest --cov=dgp tests
# or
coverage run --source=dgp -m unittest discover
```

Add the following parameter to display lines missing coverage when using the pytest-cov extension:

```
--cov-report term-missing
```

Use `coverage report` to report the results on test coverage:

```
$ coverage report -m
Name                                Stmts   Miss  Cover   Missing
-----------------------------------------------------------------
dgp/__init__.py                         0      0   100%
dgp/lib/__init__.py                     0      0   100%
dgp/lib/etc.py                          6      0   100%
dgp/lib/gravity_ingestor.py            94      0   100%
dgp/lib/time_utils.py                  52      3    94%   131-136
dgp/lib/trajectory_ingestor.py         50      8    84%   62-65, 93-94, 100-101, 106
-----------------------------------------------------------------
TOTAL                                 202     11    95%
```

Documentation

The documentation is written in reStructuredText and built using Sphinx. Some other things to know about the docs:

- It consists of two parts: the docstrings in the code and the docs in this folder.

  Docstrings provide a clear explanation of the usage of the individual functions, while the documentation in this folder consists of tutorials, planning, and technical documents related data formats, sensors, and processing techniques.

- The docstrings in this project follow the NumPydoc docstring standard. This standard specifies the format of the different sections of the docstring. See this document for a detailed explanation and examples.

- See Quick reStructuredText for a quick-reference on reStructuredText syntax and markup.
- Documentation can also contain cross-references to other classes/objects/modules using the Sphinx Domain Reference Syntax
- Documentation is automatically built on push for designated branches (typically master and develop) and hosted on Read the Docs

Building the documentation

Navigate to the `dgp/docs` directory in the console. On Linux and MacOS X run:

```
make html
```

or on Windows run:

```
make.bat
```

If the build completes without errors, then you will find the HTML output in `dgp/docs/build/html`.

Alternately, documentation can be built by calling the sphinx python module e.g.:

```
python -m sphinx -M html source build
```

1.1.8 Software Requirements Specification

Overall Description

User Classes and Characteristics

There are three types of users that interact with the system differentiated by the subset of product functions used. The user classes are:

- Operator
- Scientist
- Engineer

The Operator uses the software to assess data set quality to ensure that the systems are functioning nominally.

The Scientist uses the software to produce a gravity anomaly. They will also seek to compare data sets across flights, projects, and sensor systems.

The Engineer uses the software to evaluate hardware and software and to troubleshoot issues.

Functional Requirements

1. FR1

- Description: The user shall be able to import gravity data. The user shall be able to choose file type and define the format.
- Priority: High
- Rationale: Required to process gravity data. Allowing the user to define the type and format reduces future work to incorporate other sensors or changes to file types and formats.

- Dependencies: None

2. FR2

- Description: The user shall be able to import position and attitude data. The user shall be able to choose file type and define the format.
- Priority: High
- Rationale: Required to process gravity data. Allowing the user to define the type and format reduces future work to incorporate other sensors or changes to file types and formats.
- Dependencies: None

4. FR4

- Description: The user shall be able to organize data by project and flight.
- Priority: High
- Rationale: This is a standard organizing principle.
- Dependencies: None

5. FR5

- Description: The user shall be able to import and compare multiple trajectories for a flight.
- Priority: Medium
- Rationale: For comparison of INS hardware and post-processing methods.
- Dependencies: None

6. FR6

- Description: The user shall be able to combine and analyze data across projects and flights.
- Priority: Medium
- Rationale: For comparison of line reflown, or to produce a grid of lines flown for a survey, for example.
- Dependencies: None

7. FR7

- Description: The user shall be able to select sections of a flight for processing.
- Priority: High
- Rationale: Necessary to properly process gravity.
- Dependencies: None

8. FR8

- Description: The user shall be able to plot all corrections.
- Priority: High
- Rationale: For troubleshooting, for example.
- Dependencies: None

9. FR9

- Description: The user shall be able to choose to plot any channel.
- Priority: High
- Rationale: For quality control of data, diagnostics, and performance assessment.

- Dependencies: None

10. FR10
    - Description: The user shall be able to compare with lines and grids processed externally.
    - Priority: Medium
    - Rationale: For quality control of data, diagnostics, and performance assessment.
    - Dependencies: None
11. FR11
    - Description: The user shall be able to export data. The user shall be able to choose file type and define the format.
    - Priority: High
    - Rationale: For further processing or use in another system.
    - Dependencies: None
12. FR12
    - Description: The user shall be able to specify sensor-specific parameters.
    - Priority: High
    - Rationale: Required to process gravity data.
    - Dependencies: None
13. FR13
    - Description: The user shall be able to plot flight track on a map.
    - Priority: High
    - Rationale: To facilitate selection of sections for processing.
    - Dependencies: FR2
14. FR14
    - Description: The user shall be able to import a background image or data set as the background for the map.
    - Priority: Low
    - Rationale: To facilitate selection of sections for processing.
    - Dependencies: FR13
15. FR15
    - Description: The user shall be able to choose the method used to filter data and any associated parameters.
    - Priority: High
    - Rationale: To facilitate comparison of processing methods.
    - Dependencies: None
16. FR16
    - Description: The user shall be able to compute statistics for any channel.
    - Priority: High
    - Rationale: For quality control of data, diagnostics, and performance assessment.

- Dependencies:

17. FR17
    - Description: The user shall be able to perform cross-over analysis.
    - Priority: Medium
    - Rationale: For quality control at the level of a whole survey.
    - Dependencies:
18. FR18
    - Description: The user shall be able to perform upward continuation.
    - Priority: Low
    - Rationale: For quality control at the level of a whole survey.
    - Dependencies:
19. FR19
    - Description: The user shall ble able to flag bad data within lines and choose whether to exclude from processing.
    - Priority: High
    - Rationale: For quality control of data, diagnostics, and performance assessment.
    - Dependencies:
20. FR20
    - Description: The user shall be able to import outside data sets (e.g., SRTM, geoid) for comparison with flown gravity.
    - Priority: High
    - Rationale: For quality control of data, diagnostics, and performance assessment.
    - Dependencies

1.1.9 Documentation ToDo's

Todo: Write documentation/tutorial on how to use the application, targeted at actual users, not developers.

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/dgp/checkouts/develop/docs/source/userguide.rst, line 4.)

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

Python Module Index

d

Index

B

D