SDQC

Release 0.5.0

Eneko Martin Martinez

Nov 09, 2022

CONTENTS

System Dynamics Quality Check (SDQC): run quality checks over Vensim models input data files.

System Dynamics models are generally data intensive. Large datasets are difficult to maintain and are commonly plagued with issues that affect their quality. When low quality data is fed to SD models, the issues propagate to their outputs.

This library facilitates the task of loading input data to System Dynamic models from external files (spreadsheets), running common data quality checks on it (e.g. outlier detection, missing values, etc.) and generating data quality reports.

This project was originally thought as an extension to the PySD library, and in fact, it uses it to load the model input data by parsing either the Vensim model file (.mdl) or the Python translation (using PySD). Additionally, the library also allows to load data from external data files by reading the location of each dataset from a user defined dictionary.

Vensim is a well known platform to build SD models. However, its flexibility when it comes to loading external data also comes at a price. Indeed, when defining calls to external data from Vensim, it is surprisingly easy to introduce errors that generally go unnoticed by the software and that may lead to unexpected model results. If addition, the root of such unexpected results is generally painful to track down. Accordingly, **this library should be of particular interest to Vensim users who work on large models** and want to make sure that what they feed to their model is what they expect.

The code for SDQC is available here. If you find a bug, or if you wish to requuest a new feature, please use GitLab's issue tracker. For contributions see the *Development* section.

CHAPTER

ONE

CONTENTS:

1.1 Installation

1.1.1 Installing using pip

To install the SDQC package from the Python package index use the following command:

```
pip install sdqc
```

1.1.2 Installing from source

To install from source, clone the project with git:

```
git clone https://gitlab.com/eneko.martin.martinez/sdqc
```

or download the latest version from GitLab.

From the project's main directory, use the following command to install it:

```
python setup.py install
```

1.1.3 Required Dependencies

SDQC requires Python 3.7+, PySD 3.8+, pypandoc (and pandoc 2.17+), netCDF4 (1.5 for Python 3.7 in Windows and 1.6+ for all other cases), and importlib-metadata 2.0 (only for Python 3.7).

If not installed, PySD should be built automatically if you are installing via *pip* or from source.

1.2 Usage

The SDQC library can load external model input data directly from Vensim's *.mdl* files, PySD translated *.py* files and user-defined dictionaries containing the required information on the location of each dataset in the data file. The `check()` function is used to launch the quality check for the three file types.

By default, a `pandas.DataFrame` will be returned, including information on the data quality issues identified. If *verbose=True* is passed as argument to the `check()` function, the information of all performed cheks will be returned, regardless of whether any issue was identified or not.

Warning: Using the default parameters (see section *Default configuration*), the `check()` will only identify missing values and check for the monotonicity of series data. To modify the default configurations see *Overriding defaults*.

Note: For more information on the arguments accepted by the `check()` function, check the *API section* or use the `help()` as follows:

```
import sdqc

help(sdqc.check)
```

1.2.1 Running the data quality checks

Using a Vensim model file

To run the **default** data quality checks directly from a Vensim model '.mdl' file:

```
import sdqc

model = sdqc.check('my_model.mdl')
```

Using a PySD model file

To run the **default** data quality checks from a PySD model file '.py' file:

```
import sdqc

model = sdqc.check('my_model.py')
```

Note: The previous two cases require PySD to translate the *.mdl* file. Using the already translated *.py* file will naturally be faster.

Note: The PySD library is under constant development and does not yet support the complete list of functions available in Vensim. Therefore, the use of not yet implemented Vensim functions may cause the PySD translator to stop unexpectedly. If that is your case, please use PySD issue tracker on GitHub to report the issue.

Using either a Vensim or PySD model file and a netCDF file containing the external data

The only difference between this and the previous two cases, is that in this particular case the external data will be loaded from a netCDF file (*.nc*). This has the advantage of being much faster:

```
import sdqc

model = sdqc.check('my_model.py', externals="externals.nc")
```

Note: For instructions on how to export the external objects into a netCDF file, please check the PySD documentation.

Using a list of initialized PySD External objects

To be able to run the checks to a set of all the model external data elements, the user can opt to load and initialize those elements beforehand (using the `load_data()` function), and then pass them to the `check()` function as follows:

```
import sdqc
from sdqc.loading import load_data

elements = ['Var1_data', 'Var2_data', 'Var3_data']

external_data = load_data('my_model.mdl', elements_subset=elements)

sdqc.check(external_data, config_file='checks_config.ini')
```

The path passed to the `load_data()` function can either point to a Vensim model file or a PySD model file. Similarly, the names of the variables passed as the *elements_subset* argument can use the original name of the variable in Vensim (original_name), or the names used in Python for the function that calls the external object (py_short_name), or the name of the external object itself (py_name). If the *elements_subset* is not passed, all the external data elements of the model will be loaded.

Alternatively to the *elements_subset* argument, the user can pass the *files_subset* argument, which is a list of the names of the files from which to loaded external data. If the *files_subset* argument is not passed, variables from all external files in the model will be loaded. See example below:

```
import sdqc
from sdqc.loading import load_data

files = ['model_parameters/file1.xlsx', 'model_parameters/file2.xlsx']

external_data = load_data('my_model.mdl', files_subset=files)

sdqc.check(external_data, config_file='checks_config.ini')
```

Note: File paths provided in the *files* list must be relative to the location of the model file.

Loading and initializing the external data objects beforehand, allows to run the `check()` function with any combinations of check configurations, report configurations and report formats, with a single load of the required model external data (which is computationally expensive). This is particularly convenient for large models, to separate the checks configurations in multiple separate files. See example below:

```
import sdqc
from sdqc.loading import load_data

# separate quality check configuration files for outliers, monotonicity and missing␣
↪values
check_config_files = ['outliers.ini', 'monotonicity.ini', 'missing_values.ini']

# paths of the reports to be generated (note the different report formats)
report_files = ['outliers.html', 'monotonicity.md', 'missing_values.docx']

# configuration of the thresholds that define the severity of the issue
# (e.g. "WARNING", "ERROR", "CRITICAL") depending on the number of issues for
# each type of check
report_config_file = 'report_config.ini'

# load and initialize all model external data objects
external_data = load_data('my_model.py')

# run the data quality checks configured in each configuration file, and
# generate the reports defined in the report_files
for chck, rep in zip(check_config_files, report_files):
    sdqc.check(external_data, config_file=chck, output='report',
               report_config=report_config_file, report_filename=rep,
               verbose=True)
```

The previous code will load all model external data first, and then run the data quality checks configured in each configuration file, and generate the reports defined in the report_files (in html, markdown and docx, respectively).

Using a user-defined dictionary

Using a dictionary to load the external data is also possible. The key-value pairs required to read LOOKUP, DATA and CONSTANT elements are shown hereafter:

```
{
  'element_lookup':{
    'type': 'EXTERNAL'
    'excel': [
        {'filename': file_name,
         'sheet': sheet_name,
         'cell': cell_value,
         'x_row_or_cols': x_row_or_cols,
         'subs': None,
         'root': root_to_file}
    ]
  },
  'element_data':{
    'type': 'EXTERNAL'
    'excel': [
        {'filename': file_name,
         'sheet': sheet_name,
         'cell': cell_value,
         'x_row_or_cols': time_row_or_cols,
```

(continues on next page)

(continued from previous page)

```
         'subs': None,
         'root': root_to_file}
    ]
  }
  'element_constant':{
    'type': 'EXTERNAL'
    'excel': [
        {'filename': file_name,
         'sheet': sheet_name,
         'cell': cell_value,
         'x_row_or_cols': None,
         'subs': None,
         'root': root_to_file}
    ]
  }
}
```

Note: Though an unlimited number of elements may be loaded, the element name given as a key to the dictionary must be unique.

For matrices defined in several lines, the subscript information must be provided in the dictionary as follows:

```
{
  'element': {
    'type': 'EXTERNAL'
    'excel': [
        {
        'filename': file_name,
        'sheet': sheet_name,
        'cell': cell_value,
        'x_row_or_cols': time_row_or_cols,
        'subs': ['A', 'Dim2'],
        'root': root_to_file
        },
        {
        'filename': file_name,
        'sheet': sheet_nam3,
        'cell': cell_value2,
        'x_row_or_cols': time_row_or_cols,
        'subs': ['B', 'Dim2'],
        'root': root_to_file
        }
      ]
    },
  'Dim1': {
    'type': 'SUBSCRIPT',
    'values': ['A', 'B']
    },
  'Dim2': {
    'type': 'SUBSCRIPT',
    'values': ['C', 'D', 'E']
```

(continues on next page)

(continued from previous page)

```
    }
}
```

Note: Subscripts must be defined using the subscript range as the key and their values inside a list.

The example below demonstrates how to load data located in *Sheet1* of the *inputs.xlsx* file (located in the current working directory), based on the information available in the dictionary:

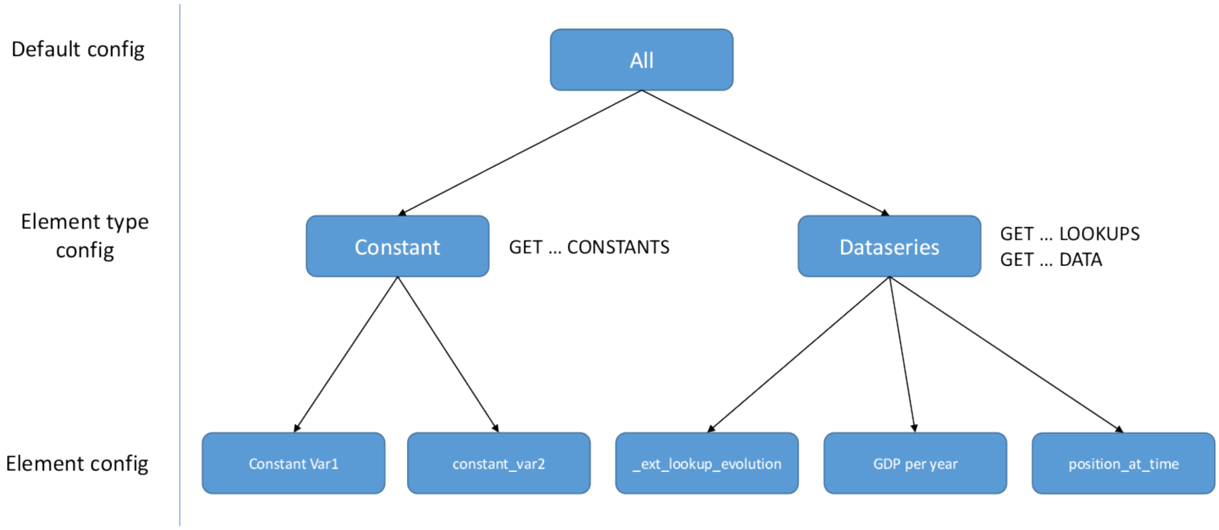
```
 import os
 import sdqc

 _root = os.getcwd()  # ful path to current working directory

 element_dict = {
  'element': {
    'type': 'EXTERNAL'
    'excel': [
        {
        'filename': 'inputs.xlsx',
        'sheet': 'Sheet1',
        'cell': 'B4',
        'x_row_or_cols': '3',
        'subs': ['A', 'Dim2'],
        'root': _root
        },
        {
        'filename': 'inputs.xlsx',
        'sheet': 'Sheet1',
        'cell': 'B7',
        'x_row_or_cols': '3',
        'subs': ['B', 'Dim2'],
        'root': _root
        }
      ]
    },
  'Dim1': {
    'type': 'SUBSCRIPT',
    'values': ['A', 'B']
    },
  'Dim2': {
    'type': 'SUBSCRIPT',
    'values': ['C', 'D', 'E']
    }
}

model = sdqc.check(element_dict)
```

Note: This example shows how to transform a json with a random structure to a dictionary that uses the structure required by the SDQC library. Several test json files may be found here.

1.2.2 Configuring the data quality checks

Structure of the configuration files

The configuration files used to parametrise the quality checks in SDQC follow the hierarchy shown in the figure below:



The types of checks to perform (see section *Currently implemented data quality checks* for a list of the available checks) may be defined at any of the 3 levels, under the sections with the same names (*All, Constants/Dataseries, individual element name*) in the configuration file.

Note: Specific elements can be defined using Vensim's variable names (*e.g. Constant Var 1*), PySD element names (*e.g. constant_var_1*) or PySD object names (*e.g. _ext_constant_constant_var_1*).

Configurations defined for child element/s override the configuration of the parent, for that/those particular element/s. See the *Overriding defaults* and the *Examples* sections for more details on the hierarchy.

Default configuration

The default configuration file (*default-conf.ini*), located in the project's main folder, contains the following parameters:

```
[All]
# MISSING VALUES
# check for missing values
missing = yes
# completness of missing values (see documentation of missing_values_data)
completeness = any

# OUTLIERS
# check for possible outliers
outliers = no
```

(continues on next page)

(continued from previous page)

```
# method to use (see documentation of outlier_values)
outliers_method = std
# number of standard deviations to detect outliers (std method)
outliers_nstd = 2
# number of interquartile ranges to detect outliers (iqr method)
outliers_niqr = 1.5

# MONOTONY
# check series monotony
series_monotony = yes

# RANGE
# series range
series_range = false
# min, max values
series_range_values = -1e350 1e350

# INCREMENT
# series increment type
series_increment = no
series_increment_type = linear
```

Note: The only checks activated in the default configuration are the detection of **missing values** and the **monotonicity** for ALL elements (note that all configurations are defined under the [*All*] section).

Note: To see the list of data quality checks that you can add to the configuration file, see the *Currently implemented data quality checks* section.

Overriding defaults

To override any (or all) of the default configurations set in the *default-conf.ini*, users are encouraged to set the desired values in a new *.ini* file (following the structure defined in *Structure of the configuration file* section).

Note: If for one or more elements, any of the potential configuration parameters are not set, then the value of that parameter in the default configuration file will be used.

For the new configurations to take effect, the path to the new *.ini* file must be passed as an argument to the `check()` function:

```
import sdqc

model = sdqc.check('my_model.mdl', 'my_cofig.ini')
```

Examples

Note: This section assumes that the user has created a new *.ini* configuration file, whose path will be passed to the `check()` function to override the default configurations from the *default-conf.ini* file.

To assign specific data quality checks to a certain individual element, users may add the following section:

```
[Constant Var 1]
missing = true
outliers = false
```

In the example below, *Constant Var 1* does not declare the *outliers_method*, and so it will inherit the value of this parameter from the *Constants* section (parent). Also, and even if the parent section (*Constants*) says otherwise, *Constant Var 1* will not be checked for missing values (*missing=false*). Similarly, the outlier detection method (*outlier_method*) for all *Constants* will be *iqr* (inter quartile range), even if the parent section (*All*) defines it as *std* (Standard deviation).:

```
[All]
outliers_method = std

[Dataseries]
outliers = true

[Constants]
outliers_method = iqr
missing = true

[Constant Var 1]
missing = false
outliers = true
```

1.3 Currently implemented data quality checks

A list of available checks is shown below:

Check	Description	Flag	Arguments	Check function	Target
Missing values	Checks for missing values on the data.	missing	completeness*	missing_values missing_values_data*	Constant, series** and dataseries*
Outlier values	Checks for outlier values on the data.	outliers	outliers_method, outliers_nstd, outliers_niqr	outlier_values	Constant and data
Series range	Checks if series is inside a range.	series_range	series_range_values	series_range	Series
Series monotony	Checks if series is monotonically increasing.	series_monotony		series_monotony	Series
Series increment type	Checks if series series increment type	series_increment	series_increment_type	series_increment_type	Series

* *completness* argument is only used for *dataseries* calling *missing_values_data*

** the check for missing values is always passed over series values as the missing values in the series dimesion have to be removed before passing other tests.

Information about each check argument is shown in the table below:

Argument	Check	Description	Possible values	Default
completeness	Missing values	If set to 'any' the check will fail if there is any missing value for any series value. If set to 'all' the check will fail if all the data values are missing for a given series value (column). It only has an effect when data is a matrix (2 or more dimensions).	'any' or 'all	'any'
outliers_method	Outlier values	The method to be used. Can be 'std' for standard deviation method or 'iqr' for interquartile range method.	'std' or 'iqr'	'std'
outliers_nstd	Outlier values	For 'std' method, the number of standard deviations to define outliers.	float > 0	2
outliers_niqr	Outlier values	For 'iqr' method, the number of interquartile ranges to define outliers.	float > 0	1.5
series_range	Series values	The minimum and maximum value of the series.	[float, float]	[-inf, inf]
series_increment_type	Series increment type	The series distribution. If 'linear' will check if the series increment linearly.	'linear'	'linear'

1.4 API

1.5 Report Generation

SDQC reports allow to classify the identified issues by severity (*WARNING*, *ERROR* and *CRITICAL*), to generate some statistics about their number and type and to export the result to a file.

Currently supported report file formats are **html** (default), **markdown** (github), **docx** and **PDF**.

Warning: Generating markdown and pdf reports requires the *pandoc* package. Additionally, to build pdf reports, the *Tex Live* and *XeTeX* packages are required.

Note: SDQC has been tested with Pandoc >= 2.17. Pandoc versions below 2.14 do not support the rowspan attribute in html <th> tags and will not produce adequate results for report formats other than html. Note that when installing pypandoc with conda, it will already install Pandoc's latest version. When installing with pip, Pandoc will not be installed automatically, but it's latest version may be installed using `pypandoc.pandoc_download()`. For more details, please check pypandoc documentation.

To generate reports with default configurations and format, use the following command:

```
sdqc.check('my_model.mdl', output='report', verbose=True)
```

If the argument *verbose* is set to *True*, a summary with more detailed data quality statistics will be added to the report.

1.5.1 Overriding default report configuration

The default configuration for the reports is given in the *default-report-conf.json* file.

In order to override the defaults, the user can copy and rename the default JSON file and set new values. Then, the path to the new configuration file can be passed as an argument as follows:

```
sdqc.check('my_model.mdl', output='report', report_config='my-report-conf.json',␣
↪verbose=True)
```

Configuration files must follow the same structure than the default one, and only include the fields that the user wants to override. The available configuration fields and their values in the default configuration file are:

- **missing_values_series**: {“WARNING”: 20, “ERROR”: 30, “CRITICAL”: 50},
- **outlier_values**: {“WARNING”: 20, “ERROR”: 30, “CRITICAL”: 50},
- **missing_values**: {“WARNING”: 20, “ERROR”: 30, “CRITICAL”: 50},
- **missing_values_data**: {“WARNING”: 20, “ERROR”: 30, “CRITICAL”: 50},
- **series_monotony**: “CRITICAL”,
- **series_range**: “ERROR”,
- **series_increment_type**: “ERROR”

Note: *missing_values_series*, *outlier_values*, *missing_values* and *missing_values_data* only accept objects of three integer values between 0 and 100, corresponding to the tolerance associated to WARNING, ERROR and CRITICAL. For instance, in the case of *missing_values*, if the value is set to `{'WARNING': 20, 'ERROR': 30, 'CRITICAL': 50}`, it means that if the data contain less than 20% of missing values, the issue will be graded as a WARNING, while a grading of ERROR and CRITICAL will be given for more than 30 or 50% of missing values, respectively.

Note: *series_monotony*, *series_range* and *series_increment_type* only accept strings corresponding to the severity given to the issues of each of these types.

1.5.2 Generating multiple reports at once

The methods *build_report* and *report_to_file* of the *Reports* class allow to pass alternative report configurations and output files (and formats) for the same quality check results. This way, users can run the quality checks just once, and generate any number of reports in any of the suported formats.

To do that, the user must first create a *Reports* object, by passing the argument *output='report'* to the *check* function as described in the first section. Then call the *build_report* and *report_to_file* as many times as different report configurations (json files or dictionaries) are defined:

```
# build a report using the report_conf.json report configuration file and save it to the␣
↪file report.html
report_obj = sdqc.check('my_model.mdl', output='report', report_config='report_conf.json
↪', report_filename='report.html', verbose=True)

# updating report configuration for missing values:
updated_report_config = {"missing_values_series": {"WARNING": 20, "ERROR": 40, "CRITICAL
↪": 50}}

# update report using the new configuration (updated_report_config dictionary)
report_obj.build_report(updated_report_config)

# save the new report to a markdown file (report.md)
report_obj.report_to_file(report_filename='report.md')

# save the new report to a docx file (report.docx)
report_obj.report_to_file(report_filename='report.docx')
```

The previous code will generate a report in *html*, and then two additional reports, using a different report configuration, in *md* and *docx*.

1.6 Development

Available soon…

CHAPTER

TWO

ADITIONAL RESOURCES

2.1 PySD docs

PySD documentation is available at: http://pysd.readthedocs.org/

CHAPTER

THREE

ACKNOWLEDGMENTES

This library was originally developed for H2020 LOCOMOTION project by @eneko.martin.martinez at Centre de Recerca Ecològica i Aplicacions Forestals (CREAF).

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No. 821105.