A COMMON, OPEN SOURCE INTERFACE BETWEEN EARTH OBSERVATION DATA INFRASTRUCTURES AND FRONT-END APPLICATIONS

Deliverable 24

Version 1.0 from 2019/12/05

First Iteration of Use Case Chains

Change history

Issue	Date	Author(s)	Description
0.1	2019/11/28	Bernhard Gößwein, TU Wien Claudio Navacchi, TU Wien	First draft.
0.2	2019/11/29	Peter Zellner, EURAC	UC 4 - prototype with R client
0.3	2019/11/30	Bernhard Gößwein, TU Wien Claudio Navacchi, TU Wien	Preliminary version to be submitted to Wageningen.
0.4	2019/12/03	Milutin Milenković, WUR Andrei Mirt, WUR Dainius Masiliunas, WUR Jan Verbesselt, WUR	1st internal review.
0.5	2019/12/04	Bernhard Gößwein, TU Wien Claudio Navacchi, TU Wien	Fixed issues from the 1st internal review.
1.0	2019/12/05	Matthias Schramm, TU Wien	Final review for submission

For any clarifications please contact openEO@list.tuwien.ac.at.

Number of pages: **37**

Disclaimer

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 776242. Any dissemination of results reflects only the author's view and the European Commission is not responsible for any use that may be made of the information it contains.

Table of Contents

List of Listings

List of Figures

List of Acronyms

API Application Programming Interface

EO Earth Observation

JSON JavaScript Object Notation

NDVI Normalised Difference Vegetation Index

UDF User Defined Function

1 Executive summary

This deliverable focuses on describing the first iteration of the openEO use case chains. Some of these use cases need a complex setup of functionalities, which cannot be realised by the general processes provided by the openEO process specification[1]. To support users in developing algorithms and to not let their creativeness be hindered by a limited set of available processes, the concept of User Defined Functions (UDFs) was initiated during the first project year. The herein explained UDF API defines schemes, which enable a communication between the data and environment located at the back-end and the deployed algorithm of the user. Moreover, the UDF needs to follow an API framework specified for each programming language (currently available: Python and R). The API framework defines all relevant input and output data types for a UDF. Based on that, preliminary implementations of use cases show that UDFs can be easily combined with the clients, therefore diminishing the gaps between user requirements and Earth Observation (EO) data processing via openEO.

2 Introduction

During the draft phase of the openEO project, a number of use cases, which should cover different data sets, user interests and expertise of back-ends, were designed. The openEO API with its predefined processes should then allow to run these use cases independently from the back-end. However, after some process design iterations, it turned out, that only simple tasks can be accomplished by such a general interface. It is possible to exploit common data cube functionalities for EO data filtering (`filter_bands`, `filter_temporal`, ..), analysis (`mean`, `quantiles`, ...) and processing (`reduce`, `apply`, ...), but more complex tasks, e.g. time series analysis, pose a greater challenge. Such scientific algorithms often need a rather specific set of methods. A general definition of these functionalities would extend the scope of openEO and would affect its clarity and usability. Moreover, profit oriented users do not necessarily want to share their code with others and would thus only need a user-friendly data access at the back-end. Therefore, UDFs are introduced in openEO to enable a flexible interaction of common openEO processes and data with functions defined by the user.

The complexity and knowledge about how UDFs work and what needs to be considered strongly depends on the user itself, i.e. if one wants to create a UDF or wants to execute an existing UDF. The latter case demands knowledge about the input and output parameters of the UDF, whereas the former requires more expertise about the language-based UDF specification. But, on whatever the user decides, UDFs should allow to bring the user one step closer to the data not being too restrictively tied to predefined processes. To realise this and to deploy UDFs at each back-end, some effort is needed to offer an interface for each supported programming language, which are R and Python at the moment.

Revising the use case descriptions shows that only few can be implemented solely based on openEO processes and most have to be realised via a UDF. This emphasises the necessity of UDFs therefore being the main focus of this deliverable. It starts with an API description of UDFs including an elaboration of API schemas and processes being responsible for sending a

[1] http://api.openeo.org/processreference/

UDF to a back-end. Then, some UDF interface implementations in R and Python are shown in Chapter 4. This chapter should help to understand what tools are needed to create a UDF in a specific programming language. Finally, the deliverable concludes with some exemplary use case chains, where first realisations of the proposed openEO use cases are shown.

3 User Defined Functions - API

UDFs are customised code snippets that can be executed at the back-end where the input and output is embedded into a process graph. Figure 3.1 shows the workflow of this procedure. The *Process Graph Processing* component represents the part of the back-end where the processes of a process graph are executed. If there is a process that calls a UDF, it will send the code and the data to the back-ends *UDF service* component. The process chain is interrupted for the call of the UDF. It will then run the code and return the resulting data back to the process to follow. If the UDF is defined to be executed at an external defined container, the code and data will be forwarded to it and the resulting data returned.

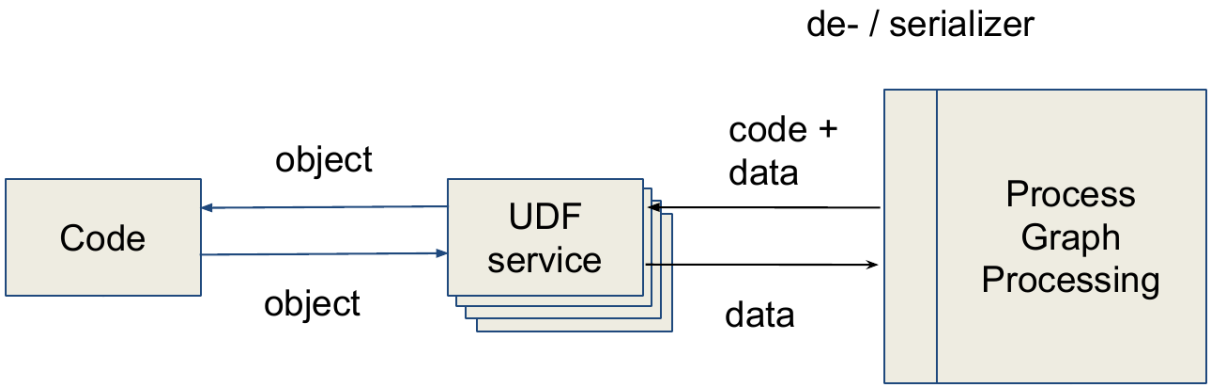

Figure 3.1: Overview of the UDF workflow.

There are different approaches to achieve the UDF workflow at a back-end. In the following, we will show different approaches to achieve parts of the UDF workflow:

UDF Service

There are three approaches to implement the *UDF service* component at the back-end:

1. **REST Service**
   The *UDF service* is a stand-alone micro RESTful service. The transfer of the data and the code is via HTTP calls. It has the advantage to be just loosely coupled to the back-end. Therefore, the *UDF service* implementation can be re-used by other back-end providers. The data needs to be serialised, so that it can be transferred via HTTP request. For this purpose, JavaScript Object Notation (JSON) serialisation can be used by the *UDF services*.
2. **Command Line Tool**
   The *UDF service* is a command line tool and called via command line or starting a container. Requires serialisation of the data into an image format. It is intended to run in a container, since it makes it independent on the system configuration of the back-end.
3. **In-Process Call**

The *UDF service* is implemented by an invocation of a script from the back-end with in-memory objects. This strategy is dependent on the host system configuration. Additionally, it is dependent on the programming language used for the script, but has the possibility of exchanging data in memory.

Code Execution

1. **Script**
   The code execution is implemented as a script, which provides high flexibility. The data injection can be defined as a programming language dependent data cube structure.
2. **Function**
   The code execution is implemented as a function call. The data can be injected as an array. This approach has the advantage, that back-end provider can easily manage and scale the execution. It may cause many requests in certain use cases, because the data is too small e.g. time series.

Data Re-import

Data re-import describes how the result of the UDF is returned to the process chain of the *Process Graph Processing* component. There are three approaches defined on that issue:

1. **In-memory**
   The in-memory import of the result has the advantage of being faster than the other approaches. The major disadvantage of this is that it is language dependent, since the back-end implementation needs to have the same programming language as the UDF.
2. **File based**
   The resulting data is stored in a file and the *Process Graph Processing* component fetches the data directly from that file. In this approach the metadata exchange has to be well defined e.g. by specifying dimensions.
3. **JSON response**
   This approach suggests to return the results of the UDF via a JSON response back to the *Process Graph Processing* component. This is an easy-to-implement solution for the standalone REST service approach. Nevertheless, it is potentially huge because of the plain text serialisation of JSON and may lack information.

The UDF API definition describes the endpoints necessary to execute the UDFs and get the resulting data using an openEO back-end (see Section 3.1). Furthermore, it defines serialiseable data schemes to have a standardised way of representing input and output data of the UDFs (see Section 3.2). There are openEO processes defined to call UDFs inside of a process graph (see Section 3.3).

3.1 UDF - Endpoints

This section describes the endpoints of the UDF service, if hosted as a REST service. The following endpoints describe how the UDF API defines the communication at the moment.

Execute UDF

- **POST /udf**
  Run a user defined function (UDF) on the provided data.
  - **Request type**: object
  - **Response**: The result of the UDF computation.
  - **Request example**

```
{
  "code": {
    "language": "python",
    "source": "import numpy as np \n \ndef udf(data): \n    pass\n"
  },
  "data": {
    "proj": "EPSG:4326",
    "hypercubes": [...],
    "feature_collection_tiles": [...],
    "structured_data": [...],
    "machine_learn_models": [...]
  }
}
```

Listing 1: Example request of the POST /udf_runtimes endpoint.

Machine learning

- **POST /storage**
  Store a machine learn model in the UDF machine learn database and return the corresponding md5 hash. The URL where the model is located must be provided as text in the HTTP request.
  - **Request type**: object
  - **Response**: The md5 hash of the stred model.
- **GET /storage**
  Return a list of all md5 hashes of the provided machine learn models
  - **Response**: A list of md5 hashes of all stored machine learn models.
- **DELETE /storage**
  Delete a machine learn model in the udf machine learn database that matches the provided md5 hash. The md5 hash of the model, which needs to be deleted, must be provided as text in the HTTP request.
  - **Request type**: object

3.2 UDF - Schemes

The UDF schemes define the format of the data objects used as in- and output data for UDFs. They are defined in a standalone openAPI document, other than the openEO API. There, the data objects are in JSON format following the schemes described in the following sections.

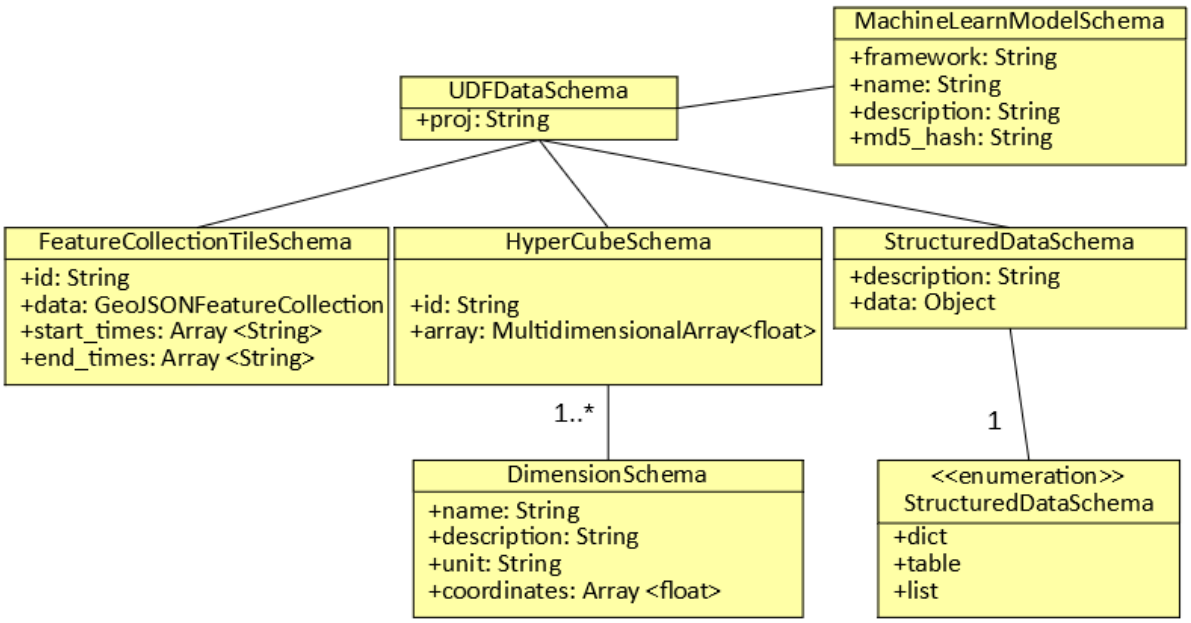

Figure 3.2: Overview of the UDF schemes.

3.2.1 UdfDataSchema

The UDF data object consists of the projection information, feature collection tiles, machine learn models, structured data and a list of hypercubes. The mentioned data structures are described in the following sections. The UDFDataSchema object is the argument for the UDF as well as their return value. If a function needs a specific input data type, the data has to be set in the corresponding part of the data object (e.g. hypercube).

Properties

- **proj (required)**
  The EPSG code or WKT projection string e.g.: EPSG:4326.
  - **Data type**: string
- **feature_collection_tiles**
  A list of feature collection tiles (see Section 3.2.3).
  - **Data type**: array
  - **Array items**: FeatureCollectionTileSchema
- **hypercubes**
  A list of hypercubes (see Section 3.2.2).
  - **Data type**: array
  - **Array items**: HyperCubeSchema
- **structured_data_list**
  A list of structured data objects that contain processing results that can not be represented by hypercubes or feature collection tiles. (see Section 3.2.4).
  - **Data type**: array
  - **Array items**: StructuredDataSchema
- **machine_learn_models**
  A list of machine learn models (see Section 3.2.5).
  - **Data type**: array
  - **Array items**: MachineLeanModelsSchema

3.2.2 HyperCubeSchema

A multi dimensional hypercube with configurable dimensions.

Properties

- **Id (required)**
  The identifier of this hypercube.
  - **Data type**: string
- **data**
  A multi-dimensional array of integer (8,16,32,64 bit) or float (16, 32, 64 bit) values. By default index dimension is as follows: [time][y][x]. Hence, the index data[0] returns the 2D slice for the first time stamp. The y-indexing is counted from top to bottom and represents the rows of the 2D array. The x-indexing is counted from left to right and represents the columns of the 2D array. The dimension options must be used to describe other dimension configurations.

- **Data type**: multi-dimensional array (float[][][])

– **dimensions**
The description of each dimension and the value as an ordered list. The order of the dimension in this array is the order of the dimension in the hypercube. The dimension with the name value describes the cell value.

- **Data type**: array
- **Array items**: object

Example

```
{
   "id":"test_data",
   "data":[[[0.0, 0.1],
            [0.2, 0.3]],
           [[0.0, 0.1],
            [0.2, 0.3]]
          ],
   "dimension":[
      {
         "name":"time",
         "unit":"ISO:8601",
         "coordinates":["2001-01-01", "2001-01-02"]
      },
      {
         "name":"X",
         "unit":"degree",
         "coordinates":[50, 60]
      },
      {
         "name":"Y",
         "unit":"degree",
         "coordinates":[40, 50]
      }
   ]
}
```

Listing 2: Example of a *HyperCubeSchema*.

3.2.3 FeatureCollectionTileSchema

The *FeatureCollectionTileSchema* is a tile of vector data that represents a spatio-temporal subset of a spatio-temporal vector dataset. It consists of a mandatory identifier string and a mandatory list of feature data objects and optional start and end dates. The feature data objects are defined as a GeoJSON[2] FeatureCollection.

[2]https://geojson.org/

Properties

- **Id (required)**
  The identifier of this vector tile.
  - **Data type**: string
- **data (required)**
  A GeoJSON FeatureCollection.
  - **Data type**: GeoJSON FeatureCollection
- **start_times**
  The array contains start time values for each vector feature as a date-time string format ISO 8601.
  - **Data type**: array
  - **Array items**: string (date-time ISO 8601 format)
- **end_times**
  The array contains the end time values for each vector feature, in case the time stamps for all or a subset of slices are intervals. For time instances the "from" and "to" time stamps must be equal or empty as date-time string format ISO 8601.
  - **Data type**: array
  - **Array items**: string (date-time ISO 8601 format)

Example


```
{
  "id": "test_data",
  "start_times": [ "2001-01-01T00:00:00", "2001-01-02T00:00:00"],
  "end_times": ["2001-01-02T00:00:00", "2001-01-03T00:00:00"],
  "data": {
    "features": [
      {
        "id": "0",
        "type": "Feature",
        "properties": {"a": 1, "b": "a"},
        "geometry": {
          "coordinates": [24, 50],
          "type": "Point"
        }
      },
      {
        "id": "1",
        "type": "Feature",
        "properties": {"a": 2, "b": "b"},
        "geometry": {
          "coordinates": [30, 53],
          "type": "Point"
        }
      }
    ],
    "type": "FeatureCollection"
```

```
  }
}
```

Listing 3: Example of a *FeatureCollectionTileSchema*.

3.2.4 StructuredDataSchema

The *StructuredDataSchema* represents structured data that can not be represented as a *HyperCube* or *FeatureCollectionTile*, for example, the result of a statistical computation. The data is self descriptive and supports the basic types dict/map, list and table. This data structure can also be used to provide contextual data from the user to the UDF e.g., kernel size, re-sampling pixel size, . . .

Properties

- **description (required)**
  A detailed description of the output format.
  - **Data type**: string
- **data (required)**
  The structured data. This field contains the UDF specific values (argument or return) as dictionary, list or table. A dictionary can be as complex as required by the UDF. A list must contain simple data types e.g. "list": [1,2,3,4]. A table is a list of lists with a header e.g. "table": [["id","value"], [1, 10], [2, 23], [3, 4]].
  - **Data type**: object (dictionary, list or table)
- **type (required)**
  The type of the structured data that may be of type dictionary, table or list. This is just a hint for the user how to interpret the provided data.
  - **Data type**: string enum [dict, table, list]

Example

```
{
  "description": "Output of a statistical analysis. The univariate analysis of multiple raster
                  collection tiles. Each entry in the output dict/map contains min, mean and
                  max of all pixels in a raster collection tile. The key is the id of the
                  raster collection tile.",
  "data": {
    "RED": {"min": 0, "max": 100, "mean": 50},
    "NIR": {"min": 0, "max": 100, "mean": 50}
  },
  "type": "dict"
}
```

Listing 4: Example of a *StructuredDataSchema*.

3.2.5 MachineLearnModelSchema

A machine learn model that should be applied to the UDF data.

Properties

- **framework (required)**
  The framework that was used to train the model.
  - **Data type**: string enum [sklearn, pytorch, tensorflow, R]
- **path (required)**
  The path to the machine learn model file to which the UDF must have read access.
  - **Data type**: string
- **name**
  The name of the machine learn model.
  - **Data type**: string
- **description**
  The description of the machine learn model.
  - **Data type**: string
- **md5_hash**
  The md5 checksum of the model that should be used to identify the machine learn model in the UDF storage system. The machine learn model must be uploaded to the UDF back-end.
  - **Data type**: string

Example

```
{
  "framework": "sklearn",
  "name": "random_forest",
  "description": "A random forest model",
  "path": "/tmp/model.pkl.xz"
}
```

Listing 5: Example of a *MachineLearnModelSchema*.

3.3 UDF - Processes

3.3.1 run_udf

Runs an UDF in one of the supported runtime environments. The process can either:

1. load and run a locally stored UDF from a file in the workspace of the authenticated user. The path to the UDF file must be relative to the root directory of the user's workspace, so without the user id in the path.

2. fetch and run a remotely stored and published UDF by absolute URI, for example from openEO Hub[3]).
3. run the source code specified inline as string.

The loaded UDF can be executed as a callback in several processes such as `aggregate_temporal`, `apply`, `apply_dimension`, `filter` and `reduce`. In this case an array is passed instead of a raster data cube. The user must ensure that the data is properly passed as an array so that the UDF can make sense of it.

Parameters

- **data (required)**
  The data to be passed to the UDF as array or raster data cube.
  - **Data types**:
    - **raster-cube (object)**
    - **array**:
      - **Min. number of items**: 1
      - **Array items**: Any data type.
- **urf (required)**
  Either source code, an absolute URL or a path to an UDF script.
  - **Data types**:
    - **uri (string)**: URI to an UDF
    - **string**: Source code as string
- **runtime (required)**
  An UDF runtime identifier available at the back-end.
  - **Data type**: string
- **version**
  An UDF runtime version. If set to `null`, the default runtime version specified for each runtime is used.
  - **Data type**: string / null
  - **Default value**: *null*
- **context**
  Additional data such as configuration options that should be passed to the UDF.
  - **Data type**: object
  - **Default value**: `{}` (Empty object)

[3] https://hub.openeo.org

Return Value The data processed by the UDF. Returns a raster data cube if a raster data cube was passed for `data`. If an array was passed for `data`, the returned value is defined by the context and is exactly what the UDF returned.

- **Data types**:
  - **raster-cube (object)**
  - **any**: Any data type.

3.3.2 run_udf_externally

Runs a compatible UDF container that is either externally hosted by a service provider or running on a local machine of the user. The UDF container must follow the openEO UDF specification[4].

The referenced UDF service can be executed as callback in several processes such as `aggregate_temporal`, `apply`, `apply_dimension`, `filter` and `reduce`. In this case an array is passed instead of a raster data cube. The user must ensure that the data is properly passed as an array so that the UDF can make sense of it.

Parameters

- **data (required)**
  The data to be passed to the UDF as array or raster data cube.
  - **Data types**:
    - **raster-cube (object)**
    - **array**:
      - **Min. number of items**: 1
      - **Array items**: Any data type.
- **url (required)**
  URL to a remote UDF service.
  - **Data type**: uri (string)
- **context**
  Additional data such as configuration options that should be passed to the UDF.
  - **Data type**: object
  - **Default value**: `{}` (Empty object)

Return Value The data processed by the UDF service. Returns a raster data cube if a raster data cube was passed for `data`. If an array was passed for `data`, the returned value is defined by the context and is exactly what the UDF returned.

[4] https://open-eo.github.io/openeo-udf/

- **Data types**:
  - **raster-cube (object)**
  - **any**: Any data type.

4 UDF Implementations

This chapter describes the UDF interface for both supported programming languages, Python and R, and has an emphasis on advanced users. Beside giving a rather theoretical view on the implementation, each section finishes with an example to show how a UDF could look like. How one could execute UDFs by using the Python client is explained in more detail for the different use case chains in Section 5.

4.1 Python

The current implementation of the Python 3 UDF framework can be found on the openEO GitHub site[5]. It makes use of many Python libraries and provides functionality to access raster and vector geo-data. Moreover, a command line tool is offered in addition to the API to test code and process data locally in a user-friendly way. Currently, the Python UDF API has the following components:

CollectionTile:	includes the spatial and temporal extent for raster and vector collection classes.
SpatialExtent:	stands for an axis aligned spatial extent of a collection tile.
RasterCollectionTile:	represents a three dimensional raster collection tile with time information and x/y slices with a single scalar value for each pixel. A tile represents a scalar field in space and time, for example a time series of a single Landsat 8 or Sentinel-2A band. A tile may be a spatio-temporal subset of a scalar time series or a whole time series.
FeatureCollectionTile:	is the vector complement of a *RasterCollectionTile*. It can be a subset or a whole feature collection where single vector features may have time stamps assigned.
StructuredData:	represents structured data that is produced by a UDF and can not be represented as a *RasterCollectionTile* or *FeatureCollectionTile*. For example the result of a statistical computation.

[5] https://github.com/Open-EO/openeo-udf

HyperCube:	is a hypercube representation of multi-dimensional data that stores an xarray, i.e. an *xarray.Dataset* or an *xarray.DataArray*, and provides methods to convert the xarray into the *HyperCube* JSON representation.
MachineLearnModel:	enables access and usage to machine learning tools. The model will be loaded at construction, based on the machine learning framework.
UdfData:	stores the arguments for a UDF, which includes lists of the above data types and information about the spatial reference system.

RasterCollectionTile will be deprecated in the future and will probably be replaced by *Hypercube*, since a *RasterCollectionTile* can be also represented by a *Hypercube*.

The string representation of UDFs and data objects are defined by the UDF schemes described in Section 3.2. In Python, three classes are used to store this information:

UdfRequestSchema:	is the UDF request JSON specification. It contains two properties, the code (*UdfCodeSchema*) and the data (*UdfDataSchema*).
UdfCodeSchema:	stores the UDF code and language specification.
UdfDataSchema:	data object that stores raster collection tiles, feature collection tiles, hypercubes, projection information and machine learn models. This object is argument for the UDF as well as its return value.

It has to be noted, that the above classes are not part of the API, but can be used for testing and debugging purposes.

4.1.1 Example

Building upon the functionalities offered by the Python API, one can now implement his/her own Python code being only limited by the aforementioned data type conventions. This means that the input and the output of a UDF are expected to be certain data types, which allow to translate data to JSON objects. In the following example a simple Normalised Difference Vegetation Index (NDVI) computation is realised as a UDF.

First, the data, i.e. an *xarray.Dataset* containing at least two data variables "red" and "nir", is retrieved from the hypercube. The NDVI formula is then applied to the selected data variables/bands in a next step and the result is returned by creating a new *HyperCube* object.

```python
# https://github.com/Open-EO/openeo-udf/blob/udf-api-sprint
#        /src/openeo_udf/functions/hypercube_ndvi.py
# -*- coding: utf-8 -*-


from openeo_udf.api.hypercube import HyperCube
from openeo_udf.api.udf_data import UdfData
from typing import Dict
import xarray

__license__ = "Apache License, Version 2.0"
__author__ = "Soeren Gebbert"
__copyright__ = "Copyright 2018, Soeren Gebbert"
__maintainer__ = "Soeren Gebbert"
__email__ = "soerengebbert@googlemail.com"


def apply_hypercube(cube: HyperCube,context:Dict) -> HyperCube:
    """Compute the NDVI based on a hypercube
    A hypercube with a 'band' dimension is required. A 'red' and 'nir' band should be available.
    The NDVI computation will be applied to all hypercube dimensions.
    Args:
        cube (HyperCube): The hypercube object containing an xarray DaraArray
    Returns:
        a HyperCube containing the computed NDVI, the band dimension will be dropped.
    """
    array:xarray.DataArray = cube.get_array()
    red = array.sel(band='red')
    nir = array.sel(band='nir')

    ndvi = (nir - red) / (nir + red)
    ndvi.name = "NDVI"

    hc = HyperCube(array=ndvi)
    return hc
```

Listing 6: Python UDF example calculating the NDVI.

In terms of a JSON request, one could call this code with some actual hypercube data being encoded as a JSON object. The Python code (e.g., content of the code snippet shown before) needs to be given as a string referring to the `"source"` attribute:

```json
{
  "code": {
    "source": "{...}",
    "language": "python"
  },
  "data": {
    "id": "hypercube_example",
    "hypercubes": [
      {
        "id": "multiband_hypercube",
        "dimensions": [
          {
            "name": "time",
```

```
            "coordinates": ["2017-05-01 08:16:11", "2017-05-11 08:20:11", "2017-05-21 08:16:11"]
          },
          {
            "name": "band",
            "coordinates": ["blue", "red", "nir", ...]
          },
          {
            "name": "x",
            "coordinates": [699965, 699975, 699985, 699995, ...]
          },
          {
            "name": "y",
            "coordinates": [7899995, 7899985, 7899975, 7899965, ...]
          }
        ],
        "data": [
          [[1017, 1017, 1017, 1017, ...], [...], ...], ...], ...
        ]
      }
    ],
    "proj": "EPSG:32734"
  }
}
```


Listing 7: Example of a Python request, including hypercube data. Note that the data just represents a snippet of the original example.

4.2 R

Other than the Python implementation, the R UDF implementation uses exclusively the *REST Service* approach for the *UDF Service* component. Therefore, it is a stand-alone micro service. The *in-process* approach is not implemented, since there is currently no back-end using R for processing. To enable this approach in the future, libraries that can run R code in Python and the other way around could be used, but that needs to be tested. The back-end communicates with the UDF service via RESTful HTTP requests. The R implementation is by the time of writing this document restricted to the data type of hypercubes. The input data of a UDF written in R has to be a hypercube or the deprecated *RasterCollectionTile* data format, which will not be further supported by openEO in the future. The output of a R UDF is always a hypercube. The R UDF service needs the R code and the hypercube data to start the execution of the UDF. Since this is decoupled from the back-end and other processes that can run in parallel, it has the advantage of running multiple UDFs in parallel. Additionally, the back-end has the possibility to scale the UDF executions via load balancing.

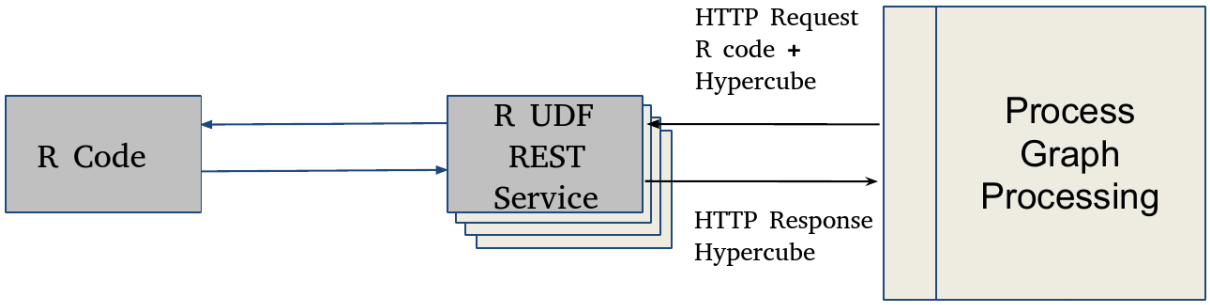

Figure 4.1: Overview of the R UDF implementation setup.

4.2.1 Example

Listing 8 shows an example of a UDF request to the R service. It contains the R code to run a minimum NDVI calculation over the hypercube data defined in the request. The hypercube has the dimensions `time`, `bands`, `x` and `y` in the `EPSG:3273` projection. The data element contains the values of the hypercube as well as an identifier. The source code in the example can access the input data via the `data` variable. First, it applies the NDVI formula on the input data resulting in the variable `ndvi_result`, whereas the eighth band is the near-infrared band and the fourth is the visible red band. Afterwards, it applies a minimum time reducer function on the ndvi. The last line in the code (the `min_ndvi` variable) defines the return value of the UDF. The complete JSON request can be viewed at the R UDF repository[6].

```
{
  "code": {
    "source": "{\n    all_dim = names(dim(data))\n
                    ndvi_result = st_apply(data, FUN = function(X, ...) {\n
                    (X[8] - X[4])/(X[8] + X[4])\n
                    }, MARGIN = all_dim[-which(all_dim == \"band\")])\n
                    all_dim = names(dim(ndvi_result))\n
                    min_ndvi = st_apply(ndvi_result, FUN = min, MARGIN = all_dim[-which(all_dim == \n
                    \"time\")])\n
                    min_ndvi\n}",
    "language": "R"
  },
  "data": {
    "id": "hypercube_example",
    "hypercubes": [
      {
        "dimensions": [
          {
            "name": "time",
            "coordinates": ["2017-05-01 08:16:11", "2017-05-11 08:20:11", "2017-05-21 08:16:11"]
          },
          {
```

[6] https://github.com/Open-EO/openeo-r-udf/blob/develop/examples/hypercube_post.json

```
            "name": "band",
            "coordinates": ["B01", "B02", "B03", "B04", ...]
          },
          {
            "name": "x",
            "coordinates": [699965, 699975, 699985, 699995, ...]
          },
          {
            "name": "y",
            "coordinates": [7899995, 7899985, 7899975, 7899965, ...]
          }
        ],
        "data": [
          [[[1017, 1017, 1017, 1017, ...], [...], ...], ...], ...
        ]
      }
    ],
    "proj": "EPSG:32734"
  }
}
```


Listing 8: Example of an R request, including the R code and the hypercube data. Note that the data just represents a snippet of the original example.

5 Use Case Chains

This chapter demonstrates first, exemplary implementations of all use cases, i.e. radar image compositing (cf. Sec. 5.1), multi-source phenology (cf. Sec. 5.2), forest monitoring (cf. Sec. 5.3) and snow monitoring (cf. Sec. 5.4). The first three use cases are realised by using the Python client, whereas the last one is based on R code.

5.1 UC1: Radar Image Compositing

Use case 1 aims to create RGB composites from aggregated backscatter data over time. The combination of the bands is either based on different points in time (e.g., different months) or polarisations. Depending on the used compositing method these images can be used for classification and crop monitoring [1]. The following example focuses on the latter case and combines polarisations from monthly aggregated data from March 2017. VV polarised, average backscatter is assigned to the red channel, VH polarised, average backscatter to the green channel and a cross-ratio (VH-VV polarised backscatter in dB) to the blue channel. This use case can be implemented by solely using the general processes in openEO.

```
import logging
import numpy as np

import openeo
from openeo import ImageCollection
from openeo.process.process import mean
```

```
logging.basicConfig(level=logging.DEBUG)

# connect with EODC back-end
session = openeo.session("nobody", "http://openeo.eodc.eu/openeo/0.4.0")

# define region of interest
min_x = 16.1
min_y = 16.6
max_x = 48.6
max_y = 47.2
# define start and end date of march
start_date = "2017-03-01"
end_date = "2017-04-01"
# define bands
bands = ["VV", "VH"]

# initialise/load data cubes
s1a_datacube = session.imagecollection("s1a_csar_grdh_iw")\
    .filter_temporal(start_date, end_date)\
    .filter_bbox(west=min_x, east=max_x, north=max_y, south=min_y, crs="EPSG:4326")\
    .filter_bands(bands)
s1b_datacube = session.imagecollection("s1b_csar_grdh_iw")\
    .filter_temporal(start_date, end_date)\
    .filter_bbox(west=min_x, east=max_x, north=max_y, south=min_y, crs="EPSG:4326")\
    .filter_bands(bands)

# merge data cubes
datacube = s1a_datacube.merge(s1b_datacube)

# compute mean
mean_datacube = datacube.reduce(mean, dimension="time")

# compute cross ratio
vv_band = mean_datacube.band('VV')
vh_band = mean_datacube.band('VH')
cr_band = vh_band - vv_band

# set label of spectral entry equal to "CR"
cr_band = cr_band.set_labels(['CR'], dimension='spectral')

# apply scaling to each band
vv_band = vv_band.linear_scale_range(inp_min=-2000, inp_max=-1000, out_min=0, out_max=254)
vh_band = vh_band.linear_scale_range(inp_min=-2000, inp_max=-1000, out_min=0, out_max=254)
cr_band = cr_band.linear_scale_range(inp_min=-500, inp_max=0, out_min=0, out_max=254)

# merge bands
composite_datacube = vv_band.merge(vh_band)
composite_datacube = composite_datacube.merge(cr_band)

# export RGB composite as GeoTIFF
composite_datacube.save_result(format="GeoTIFF", options={"nodatavalue": "255",
                                                          "R": "VV",
                                                          "G": "VH",
                                                          "B": "CR"})
```

Listing 9: Use case 1 using Python client and predefined processes.

5.2 UC2: Multi-Source Phenology Toolbox

This use case concentrates on data fusion tools, time-series generation and phenological metrics using Sentinel-2 data. It will be tested on several back-end platforms by pilot users from the Action against Hunger and the International Centre for Integrated Mountain Development. The here tested processes depend on the availability of orthorectified Sentinel-2 surface reflectance data including per pixel quality masks.

In this use case, the goal is to derive phenology information from Sentinel-2 time series data. In this case, phenology is defined by:

1. Start of season, a date and the corresponding value of the biophysical indicator
2. The maximum value of the growing curve for the indicator
3. End of season, a date and the corresponding value of the biophysical indicator

Multiple biophysical indicators exist, but in this use case, the enhanced vegitation index (EVI) is used.

We start by importing the necessary packages, and defining an area of interest. During the algorithm development phase, we work on a limited study field, so that we can use the direct execution capabilities of openEO to receive feedback on the implemented changes.

```
%matplotlib inline
import matplotlib.pyplot as plt
from rasterio.plot import show, show_hist
import rasterio

from shapely.geometry import Polygon

from openeo import ImageCollection

import openeo
import logging
import os
from pathlib import Path
import json

import numpy as np
import pandas as pd
import geopandas as gpd

import scipy.signal

#enable logging in requests library
from openeo.rest.imagecollectionclient import ImageCollectionClient

start = "2018-05-01"
end = "2018-10-01"

date = "2018-08-17"

parcels = gpd.read_file('potato_field.geojson')

polygon = parcels.geometry[0]
```

```
minx,miny,maxx,maxy = polygon.bounds


session = openeo.session("nobody", "http://openeo.vgt.vito.be/openeo/0.4.0")

#retrieve the list of available collections
collections = session.list_collections()
s2_radiometry = session.imagecollection("CGS_SENTINEL2_RADIOMETRY_V102_001") \
                    .filter_bbox(west=minx,east=maxx,north=maxy,south=miny,crs="EPSG:4326")
```

Listing 10: Use case 2: Connect to the openEO back-end, and create a Sentinel-2 datacube containing 10m reflectance bands using the Python client.

Preprocessing step 1: EVI computation

Create an EVI data cube, based on reflectance bands. The formula for the EVI index can be expressed using plain Python. The bands retrieved from the back-end are unscaled reflectance values with a valid range between 0 and 10000.

```
B02 = s2_radiometry.band('2')
B04 = s2_radiometry.band('4')
B08 = s2_radiometry.band('8')

evi_cube_nodate = (2.5 * (B08 - B04)) / ((B08 + 6.0 * B04 - 7.5 * B02) + 10000.0*1.0)

evi_cube = evi_cube_nodate.filter_temporal(start,end)

#write graph to json, as example
def write_graph(graph, filename):
    with open(filename, 'w') as outfile:
        json.dump(graph, outfile,indent=4)
write_graph(evi_cube.graph,"evi_cube.json")
```

Listing 11: Use case 2: Preprocessing step 1: EVI computation.

No actual processing has occurred until now, we have just been building a workflow consisting of multiple steps.

```
def show_image(cube,cmap='RdYlGn'):
    %time cube.filter_temporal(date,date).download("temp%s.tiff"%date,format='GTIFF')
    with rasterio.open("temp%s.tiff"%date) as src:
        band_temp = src.read(1)
    fig, (ax) = plt.subplots(1,1, figsize=(7,7))
    show(band_temp,ax=ax,cmap=cmap,vmin=0,vmax=1)
show_image(evi_cube_nodate)
```

Listing 12: Use case 2: Displaying the result of the `evi_cube`.

```
CPU times: user 7.86 ms, sys: 2.69 ms, total: 10.5 ms
Wall time: 4.06 s
```

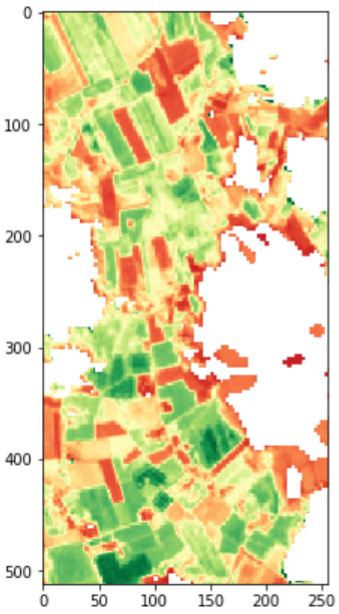

Figure 5.1: Visualisation of the `evi_cube` from the code example in Listing 11.

Preprocessing step 2: Cloud masking

In *Sen2cor sceneclassification* these values are relevant for phenology:

- 4: vegetated
- 5: not-vegetated: everything else is cloud, snow, water, shadow ...

In openEO, the mask function will mask every value that is set to `True`.

```
s2_sceneclassification = session.imagecollection("S2_FAPAR_SCENECLASSIFICATION_V102_PYRAMID") \
                    .filter_bbox(west=minx,east=maxx,north=maxy,south=miny,crs="EPSG:4326")

mask = s2_sceneclassification.band('classification')

mask = (mask != 4) & (mask !=5)
```

Listing 13: Use case 2: Cloud masking.

Masks produced by *sen2cor* still include a lot of unwanted clouds and shadow. This problem usually occurs in the proximity of detected clouds, so we try to extend our mask. To do that, we use a bit of fuzzy logic: blur the binary mask using a gaussian filter so that our mask gives us an indication of how close to a cloud we are.

By adjusting the window size, we can play around with how far from the detected clouds we want to extend our mask. A 30 pixel kernel applied to a 10m resolution image will cover a 300m area.

```
def makekernel(iwindowsize):
            kernel_vect = scipy.signal.windows.gaussian(iwindowsize, std = iwindowsize/4.0, sym=True)
            kernel = np.outer(kernel_vect, kernel_vect)
            kernel = kernel / kernel.sum()
            return kernel

fuzzy_mask = mask.apply_kernel(makekernel(29))
mask_extended = fuzzy_mask > 0.1

write_graph(mask_extended.graph,"mask.json")
mask_for_date = mask_extended.filter_temporal(date,date)

time fuzzy_mask.filter_temporal(date,date).download("mask%s.tiff"%date,format='GTIFF')
#s2_sceneclassification.filter_temporal(date,date).download("scf%s.tiff"%date,format='GTIFF')
time evi_cube_nodate.filter_temporal(date,date).download("unmasked%s.tiff"%date,format='GTIFF')
time evi_cube_nodate.filter_temporal(date,date).mask(rastermask=mask_for_date,replacement=np.nan)
                                               .download("masked%s.tiff"%date,format='GTIFF')

with rasterio.open("unmasked%s.tiff"%date) as src:
    band_unmasked = src.read(1)

with rasterio.open("masked%s.tiff"%date) as src:
    band_masked = src.read(1)

with rasterio.open("mask%s.tiff"%date) as src:
    band_mask = src.read(1)

fig, (axr, axg,axb) = plt.subplots(1,3, figsize=(14,14))

show(band_unmasked,ax=axr,cmap='RdYlGn',vmin=0,vmax=1)
show(band_masked,ax=axg,cmap='RdYlGn',vmin=0,vmax=1)
show(band_mask,ax=axb,cmap='coolwarm',vmin=0.0,vmax=0.8)
```

Listing 14: Use case 2: Cloud masking kernel.

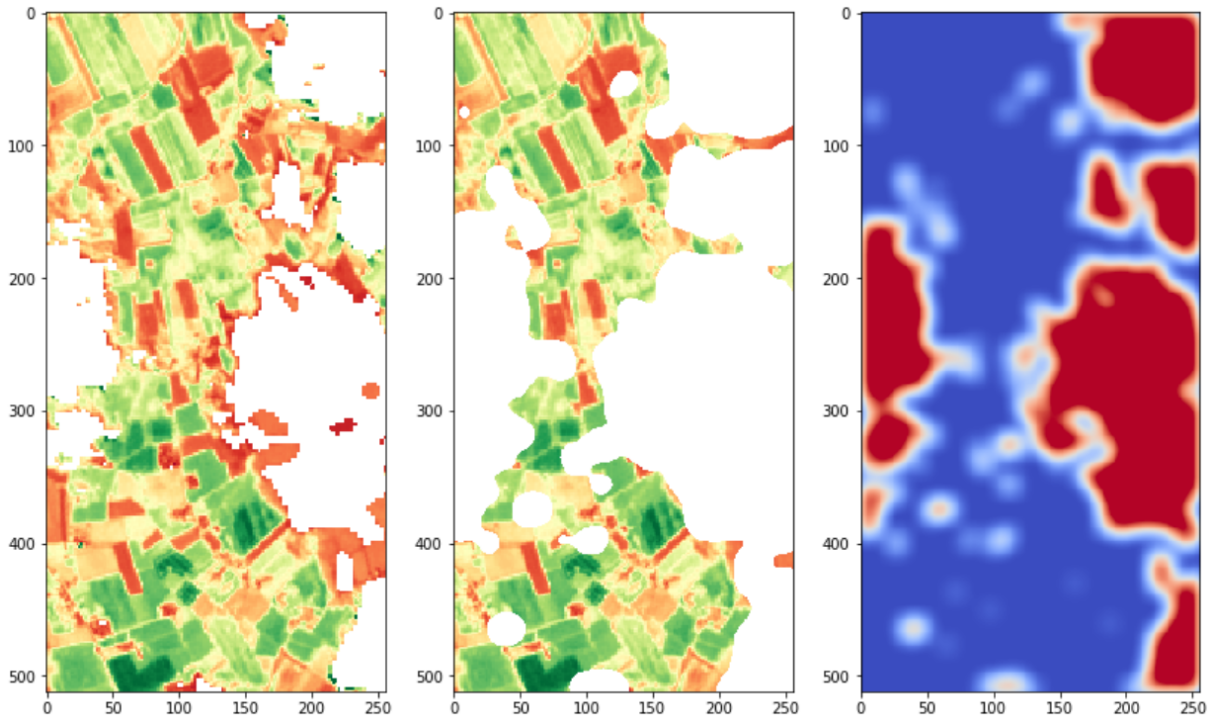

Figure 5.2: Visualisation of the `evi_cube` unmasked (l), masked (m) and the mask itself (r) from the code example in Listing 14.

Preprocessing step 3: Time series smoothing

Cloud masking has reduced the noise in our signal, but it is clearly not perfect. This is due to the limitations of the pixel based cloud masking algorithm, which still leaves a lot of undetected, bad pixels in our data.

A commonly used approach is to apply a smoothing on the timeseries. Here we suggest to use a 'Savitzky-Golay' filter.

In the end, the result should be a phenology map, so we need to apply our smoothing method on the pixel values. We use a UDF to apply custom Python code to a data cube containing time series per pixel.

The code for our UDF function is contained in a separate file, and shown below:

```
# -*- coding: utf-8 -*-
# Uncomment the import only for coding support
# from openeo_udf.api.base import SpatialExtent, RasterCollectionTile,
# FeatureCollectionTile, UdfData

__license__ = "Apache License, Version 2.0"


def rct_savitzky_golay(udf_data):
    from scipy.signal import savgol_filter
    import pandas as pd
```

```
    # Iterate over each tile
    for tile in udf_data.raster_collection_tiles:
        timeseries_array = tile.data

        # first we ensure that there are no nodata values in our input,
        # as this will cause everything to become nodata.
        array_2d = timeseries_array.reshape((timeseries_array.shape[0],
                   timeseries_array.shape[1] * timeseries_array.shape[2]))

        df = pd.DataFrame(array_2d)
        #df.fillna(method='ffill', axis=0, inplace=True)
        df.interpolate(inplace=True,axis=0)
        filled=df.as_matrix().reshape(timeseries_array.shape)

        #now apply savitzky golay on filled data
        smoothed_array = savgol_filter(filled, 5, 2,axis=0)
        #print(smoothed_array)
        tile.set_data(smoothed_array)


# This function call is the entry point for the UDF.
# The caller will provide all required data in the **data** object.
rct_savitzky_golay(data)
```

Listing 15: Use case 2: UDF for applying the 'Savitzky-Golay' filter.

Now we apply our UDF to the temporal dimension of the data cube. The code block below displays the API documentation:

```
smoothed_evi = evi_cube_masked.apply_dimension(smoothing_udf,runtime='Python', dimension='temporal')
timeseries_smooth = smoothed_evi.polygonal_mean_timeseries(polygon)

write_graph(timeseries_smooth.graph,"timeseries_udf.json")
ts_savgol = pd.Series(timeseries_smooth.execute()).apply(pd.Series)
ts_savgol.head(10)
```

Listing 16: Use case 2: Applying the UDF 'Savitzky-Golay' filter for the use case.

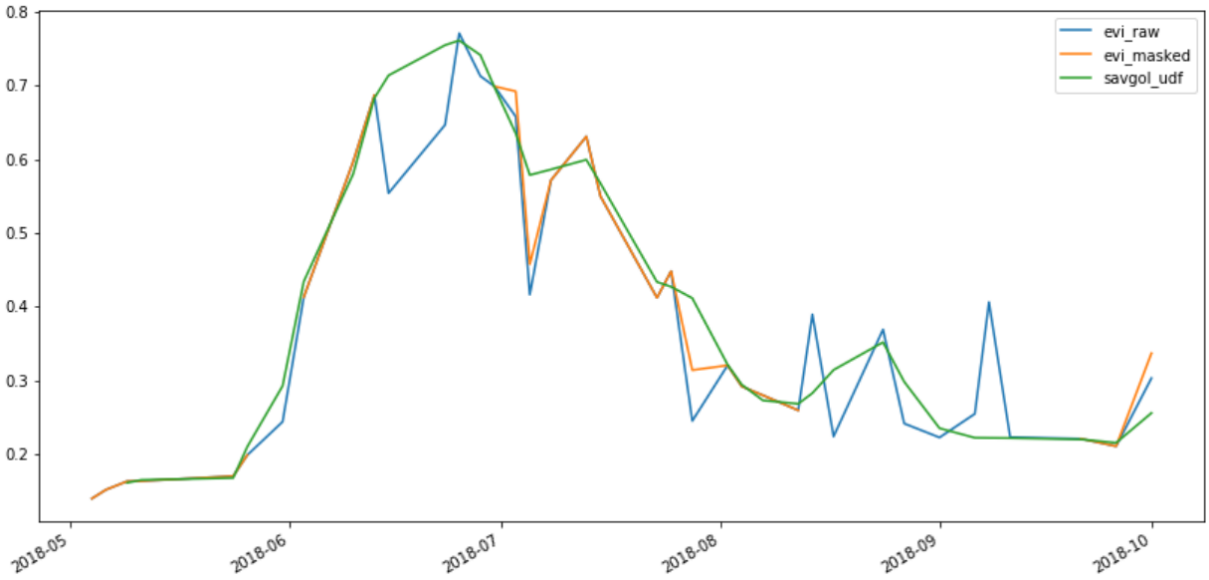

Figure 5.3: Visualisation of the timeseries `evi_raw`, `evi_masked` and using the smoothing 'Savitzky-Golay' filter UDF.

Figure 5.3 shows the result of applying a smoothing per pixel. The noise in the time series seems to be reduced.

5.3 UC3: Forest Monitoring

Use case 3 focus on a change detection method for tropical forest monitoring called Breaks For Additive Season and Trend (BFASTmonitor) [2]. The algorithm is here applied on Sentinel-1 data acquired from SentinelHub via the VITO back-end. As specified in Listing 17, the UDF functional approach and the openEO Python client are used with the functionalities of the BFASTmonitor python module[7] to detect breaks in time series for the year 2019. The time period from 2016 to 2019 is selected to infer the stable historical behaviour of the time series. The output of the UDF function (*bfast4openeo*, Listing 17) is a raster with integer values that represent a day of the year in 2019 when the algorithm detected a break in the time series. The pseudo-code example below illustrates the whole setup.

[7] https://bfast.readthedocs.io/en/latest/index.html

```
import openeo
import logging
import os
from datetime import datetime, date, time, timedelta
from shapely.geometry import Polygon
from pathlib import Path

logging.basicConfig(level=logging.INFO)

# initiate a session to the vito back-end:
VITO_DRIVER_URL = "https://openeo.vito.be/openeo/0.4.0/"
session = openeo.connect(VITO_DRIVER_URL)
# load the image collection:
s1 = session.imagecollection("SENTINEL1_GAMMA0_SENTINELHUB",bands=["VV_DB","VH_DB"])

# ---------------------------------------------------
# prepare a data cupe for the time and area of interest:
# ---------------------------------------------------
# define the dates and area of interest:
start_date = date(2016,1,1)
end_date = date(2019,11,26)
aoi_polygon = Polygon(
    shell=[[-55.8771, -6.7614], [-55.8771, -6.6503], [-55.7933, -6.6503], [-55.7933, -6.7614],
           [-55.8771, -6.7614]])
myCRS = "EPSG:4326"

# filter to a smaller datacube:
s1_vh = s1.filter_temporal([start_date.strftime("%Y-%m-%dT%H:%M:%S"),
                            end_date.strftime("%Y-%m-%dT%H:%M:%S")]) \
    .filter_bbox(west=aoi_polygon.bounds[0], east=aoi_polygon.bounds[2],
                 north=aoi_polygon.bounds[3], south=aoi_polygon.bounds[1], crs=myCRS) \
    .filter_bands(["VH_DB"])

# ---------------------------------------
# prepare functions to load the udf code:
# ---------------------------------------
def get_resource(relative_path):
    return str(Path(relative_path))

def load_udf(relative_path):
    with open(get_resource(relative_path), 'r+') as f:
        return f.read()

# -----------------------------------------
# load and apply the udf code on data cube:
# -----------------------------------------
BFASTMonitor_udf = load_udf('/local/path/to/BFASTMonitor_code.py')

# apply the udf code to reduce the data cube along the time
# dimension and get a raster which values shows
# the day of the year 2019 where the break was detected:
break_days = s1_vh.reduce(BFASTMonitor_udf, dimension='time')

# -------------------------------------------------------------
# download the deforestation probability map for a certain date:
```

```
# ----------------------------------------------------------------
# specify the raster output format
OUTFORMAT = "GTIFF"
# download the particular map:
break_days.download("breaks_detected_in_2019.tiff", format=OUTFORMAT)
# ----------------------------------------------------------------
```

Listing 17: Use case 3: Python client code using the BFASTMonitor_code.py UDF.

```
def bfast4openeo(udf_data):
    udf_data_xr = udf_data.get_array()
    from bfast import BFASTMonitor
    from bfast.utils import crop_data_dates
    from datetime import datetime
    import xarray as xr
    start_hist = datetime(2016, 1, 1)
    start_monitor = datetime(2019, 1, 1)
    end_monitor = datetime(2019, 11, 26)
    # --- revisit parsing of time/labels -----
    dates = udf_data_xr.coordinates['time']
    # ----------------------------------------
    data, dates = crop_data_dates(udf_data_xr.values, dates, start_hist, end_monitor)
    # --------------------------
    model = BFASTMonitor(
        start_monitor,
        freq=365,
        k=3,
        hfrac=0.25,
        trend=False,
        level=0.05,
        backend='Python',
        device_id=0,
    )
        # --------------------------
    model.fit(data, dates, n_chunks=1, nan_value=-32768)
    breaks_xr = xr.DataArray(model.breaks, coords=[udf_data_xr.coordinates['x'],
                             udf_data_xr.coordinates['y']], dim=['x', 'y'])
    udf_data.set_array(breaks_xr)

bfast4openeo(data)
```

Listing 18: Use case 3: The BFASTMonitor_code.py UDF code, called for use case 3.

5.4 UC4: Snow Monitoring with radar and optical EO data

The implementation of "UC4: Snow Monitoring with radar and optical EO data" creates a wet snow map using Sentinel-1 radar data in the first step. In a second step the resulting data cube is merged with the Modis Snow Cover data cube created by Eurac. These two layers of information allow to extract the four classes "no snow", "wet snow", "dry snow" and "snow" for every available time step. The crucial part of the calculation relies on more complex thresholding and nested if-else-statements which are expressed through user defined functions. The pseudo code example below describes the workflow using the openEO R-client and R-UDFs.

```
# libraries
library(openeo)
library(dplyr)

# connection to openEO Eurac back-end
driver_url = "https://openeo.eurac.edu"
user = "guest"
password = "guest_123"

conn = connect(host = driver_url,
               user = user,
               password = password,
               login_type = "basic")

# 1. Preprocessing Sentinel 1 A radar data for wet snow detection ==============
# define timespan
timespan = c("2014-09-01T00:00:00.000Z",
             "2015-09-01T00:00:00.000Z")
# start an empty process graph
graph = conn %>% process_graph_builder()
# load data cubes
s1a_vv = graph$load_collection(id = graph$data$`s1a_t117_epsg3035_20m_VV`,
                               temporal_extent = timespan)
s1a_vh = graph$load_collection(id = graph$data$`s1a_t117_epsg3035_20m_VH`,
                               temporal_extent = timespan)
s1a_lia = graph$load_collection(id = graph$data$`s1a_t117_epsg3035_20m_LIA`,
                                temporal_extent = timespan)
# create reference image VV
ref_vv = s1a_vv %>% graph$mean_time()
# create reference image VH
ref_vh = s1a_vh %>% graph$mean_time()
# calculate ratio vv to reference
rat_vv = s1a_vv %>% graph$calc_cubes(fun = "subtract", ref_vv)
# calculate ratio vh to reference
rat_vh = s1a_vh %>% graph$calc_cubes(fun = "subtract", ref_vh)
# prepare datacube as input for nagler wet snow algorithm
datacube = graph$merge_cubes(rat_vv, rat_vh, s1a_lia)

# 2. Wet snow detection using Nagler algorithm ================================
# nagler algorithm input description
# rat_vv, rat_vh  -> vv and vh backscatter ratio of the image versus the
#                    reference transformed to a logarithmic scale (dB).
# lia             -> local incidence angle in degrees
# k,theta1,theta2 -> Tuning parameters used for the computation of the
#                    combined vv-vh ratio image Rc. The values in (Nagler et
#                    al.,2016) are k=0.5, theta1=20, theta2=45
# thr             -> Threshold in dB. In (Nagler et. al, 2016) THR=-2

# 2.1 calculate weight (w) for the combined vv-vh ratio (rc) ------------------
# filter lia band
lia_band = datacube %>% filter_band("s1a_lia")
# define the udf r script
udf_script = quote({
  theta1 = 20
  theta2 = 45
```

```
  k = 0.5
  w = st_apply(data, MARGIN = "band", FUN = function(x){
    case_when(x > theta1 ~ 1,
              x < theta2 ~ k,
              TRUE ~ k*(1+(theta2-x)/(theta2-theta1)))
  })
  return(w)
})
# run the udf on the subsetted lia band
port = 5555
host = "http://euracrudfhost"
w = lia_band %>% graph$apply() %>%
  graph$run_udf(code = script_udf, host = host, port = port)

# 2.2 calculate the combined vv-vh ratio (rc): w*rat_vh+(1-w)*rat_vv -----------
rc = datacube %>%
  band_arithmetics(data = graph, formula = function(x){x[4]*x[2]+(1-x[4])*x[1]})

# 2.3 generation of the wet snow map ------------------------------------------
# keep only rc and lia bands in datacube
datacube = datacube %>% filter_bands(bands = c("rc", "lia"))
# classify based on bands rc and lia into:
# 1 = wet snow, 2 = no wet snow, 3 = shadow, 4 = overlay
udf_script = quote({
  thr = -2
  wet_snow = st_apply(data, MARGIN = "band", FUN = function(x){
    case_when(x["rc"] < thr ~ 1,
              x["rc"] >= thr ~ 2,
              x["lia"] < 25 ~ 3,
              x["lia"] >= 75 ~ 4)
  })
  return(wet_snow)
})
wet_snow = datacube %>% graph$reduce("band") %>%
  graph$run_udf(code = script_udf, host = host, port = port)

# 3. Combine the wet snow map with modis snow cover product ====================
# Identify the portion of snow which is wet snow
# modis_snow: 1 = snow, 0 = no_snow
# data_cube_wet_snow: 1 = wet_snow, 2 = no_wet_snow, 3 = shadow, 4 = overlay

# load modis snow cover product produced by Eurac Research
modis_snow = graph$load_collection(id = graph$data$`modis_snowcover_product`,
                                   temporal_extent = timespan)
# resample modis snow cover datacube to s1a wet snow datacube
modis_snow = graph$resample_cube_spatial(source = "modis_snow",
                                         target = "datacube_wet_snow",
                                         fun = "near")
modis_snow = graph$resample_cube_temporal(source = "modis_snow",
                                         target = "datacube_wet_snow",
                                         fun = "max")
# merge sentinel 1 wet snow and modis snow cover
datacube_snow = graph$merge_cubes(datacube_wet_snow, modis_snow)
# classify into no_snow = 0, wet_snow = 1, dry_snow = 2, snow = 3
udf_script = quote({
  snow_class = st_apply(data, MARGIN = "band", FUN = function(x){
    case_when(x["modis_snow"] == 0 ~ 0,
```

```
            x["modis_snow"] == 1 & x["wet_snow"] == 1 ~ 1,
            x["modis_snow"] == 1 & x["wet_snow"] == 2 ~ 2,
            x["modis_snow"] == 1 & x["wet_snow"] %in% c(3,4) ~ 3)
  })
  return(snow_class)
})
snow_class = datacube_snow %>% graph$reduce("band") %>%
  graph$run_udf(code = script_udf, host = host, port = port)
# set final node of the graph
graph$save_result(data = datacube_snow, format = "GeoTiff") %>%
  graph$setFinalNode()
# export snow class map as geotiff
job_id = conn %>% create_job(graph=graph,
                             title="wet_snow_map",
                             description="wet_snow_map",
                             format="GeoTiff")
conn %>% start_job(job_id)
result_obj = conn %>% list_results(job_id)
conn %>% download_results(job = job_id)
```

Listing 19: Use case 4: R client code using UDF.

6 References

[1] D. Nguyen, K. Clauss, S. Cao, V. Naeimi, C. Kuenzer, and W. Wagner, "Mapping rice seasonality in the mekong delta with multi-year envisat asar wsm data," *Remote Sensing*, vol. 7, no. 12, pp. 15 868–15 893, 2015.

[2] J. Verbesselt, R. Hyndman, G. Newnham, and D. Culvenor, "Detecting trend and seasonal changes in satellite image time series," *Remote Sensing of Environment*, vol. 114, no. 1, pp. 106 – 115, 2010. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S003442570900265X