
cbm

European Commission, Joint Research Centre

Apr 17, 2023

INTRODUCTION TO JRC CBM

1	Main elements of the JRC CbM	3
2	DIAS and Copernicus Sentinel	5
3	CbM in the context of the CAP	7
4	JRC solutions for CbM	9
5	Contributing to JRC CbM	11
6	System architecture	13
6.1	Scope of the CbM system	13
6.2	Cloud resources: DIAS infrastructure	15
6.2.1	What are DIAS	15
6.2.2	DIAS Virtual Machines	16
6.3	Sentinel data access: DIAS object storage	16
6.3.1	What is DIAS object storage	17
6.3.2	Copernicus ARD generation	17
6.4	Data repository - PostgreSQL/PostGIS	17
6.5	Extraction of statistics - Python processing	18
6.6	Data access - RESTful API	19
6.7	Interfaces for data use - Jupiter Notebooks	20
6.8	JRC CbM roles	20
7	Backend	21
7.1	Goals	21
7.2	General requisites to set up the CbM system	21
7.3	How to create a CbM infrastructure	22
7.3.1	Links to the technical documentation	22
7.3.2	Set up a DIAS virtual machine	22
7.3.3	CbM database	22
7.3.4	CARD	22
7.3.5	Extract signatures	23
7.3.6	Create a RESTful API	23
7.4	Deployment of the CbM system	23
7.5	Expertise required	24
7.6	Enhancement Proposals	24
7.6.1	Terrain Correction	24
7.6.2	S2 Multi-band	24
7.6.3	Dask tests	24
7.6.4	ML Revision	25

7.6.5	RESTful meteo	25
8	Frontend	27
8.1	Goals	27
8.2	Links to the technical documentation	27
8.3	RESTful API	28
8.4	Python CbM package	28
8.5	Expertise required	29
9	Data analytics	31
9.1	Goals	31
9.2	Links to the technical documentation	31
9.3	Marker detection	32
9.4	Applications	32
9.5	Expertise required	33
10	Setup DIAS VMs	35
10.1	Connecting to the ‘tenant host’ vm via SSH	36
11	Software prerequisites	39
11.1	Docker (Ubuntu 20.04)	39
11.2	PostGIS	39
11.2.1	Optimizing	41
11.2.2	Essential CbM tables	41
11.3	Jupyter server	42
11.3.1	Build Jupyter image from source	43
12	Database for JRC CbM	45
12.1	DB in the CbM architecture	45
12.2	Spatial database in a nutshell	45
12.2.1	Main database elements	46
12.2.2	The spatial bit	47
12.2.3	PostgreSQL and PostGIS	47
12.2.4	SQL	48
12.3	JRC CbM DB structure	49
12.4	Database access	51
12.4.1	Server/client structure	51
12.4.2	Access parameters	51
12.4.3	User access policy	52
12.5	Data retrieval	52
12.5.1	PgAdmin	52
12.5.2	phpPgAdmin	54
12.5.3	Psql	54
12.5.4	QGIS	54
12.5.5	Jupiter Notebook	55
12.5.6	RESTful API	56
12.5.7	R	56
12.6	Export and import data	56
12.7	Performance optimization	57
12.7.1	Basic optimization	57
12.7.2	Advanced optimization	58
13	Data preparation	59
13.1	Adding shapefiles (.shp) to PostGIS database	59
13.2	Transfer metadata from the DIAS catalog	59

13.2.1	OpenSearch compliant catalogs (CREODIAS, MUNDI)	60
13.2.2	Database catalog (SOBLOO)	60
13.3	Benchmarking data formats for fast image processing	61
14	Parcel extraction	63
14.1	Overview	63
14.2	Implementation details	63
14.3	Configuration files	64
14.4	Run with docker	65
14.5	Parallelization with docker swarm	65
14.5.1	Create VMs	66
14.5.2	Install docker and set up docker swarm	67
14.5.3	Configure and run	68
14.6	Caveats	69
15	Backing up the database	71
15.1	Overview	71
15.2	Use pg_dump	71
16	Build a RESTful API	73
16.1	Prerequisites	73
16.2	Create RESTful API users	73
16.3	Database connection	74
16.4	Dataset configuration	74
16.5	Deploy the RESTful API docker container	75
16.5.1	Build from source	75
16.6	Provide available options (Optional)	76
16.7	Adding orthophotos (Optional)	76
17	Overview	79
18	Parcel information	81
19	Parcel Time Series	83
19.1	parcelTimeSeries	83
19.2	weatherTimeSeries	84
19.3	Simple python client to plot parcel Time Series	84
20	Sentinel image chips	89
20.1	chipByLocation	89
20.2	rawChipByLocation	90
20.3	Example client code.	91
21	Parcel orthophotos	97
22	Parcel peers	99
22.1	parcelPeers	99
22.2	parcelStatsPeers	99
22.3	parcelsByPolygon	100
23	Data analytics	101
23.1	parcel selection	101
23.2	time series statistics	101
23.3	histogram analysis, clustering	102
23.4	...	102

24 POST requests	103
24.1 rawChipsBatch	103
24.2 Example test script	104
24.3 rawS1ChipsBatch	105
25 SALMS R-package	109
25.1 Introduction	109
25.2 Descriptors	110
25.3 Input data	110
25.3.1 FOIinfo	111
25.3.2 ts-files	111
25.4 plotTimeSeries	113
25.5 createSignals	114
25.6 What is week0	115
25.7 Produce and plot means, standard deviations and t-tests	115
25.8 Analysing signal probabilities	116
25.9 Summarizing the results	118
25.10 Correlations	118
25.11 References	119
26 Calendar view	121
26.1 1. Introduction	121
26.2 2. Dependencies	121
26.2.1 2.1 Calendar view Modules	121
26.3 3 Structure of the code	123
26.4 3.1 Initialization	124
26.5 3.2 Main Processing Loop	125
26.6 3.3 Output examples	125
27 FOI Assessment	137
27.1 Concept	137
27.2 Methods and algorithms	138
27.3 Test results	139
27.4 Cardinality	141
27.5 FOI Clustering	142
28 Machine learning	143
28.1 Introduction	143
28.2 Data preparation	143
28.2.1 Determine which classes to include	144
28.2.2 Composing the features for machine learning	144
28.2.3 Splitting in training and testing sets	144
28.3 Running the training and testing	145
28.4 Checking the results	145
28.4.1 Confusion matrix for testing sets	145
28.4.2 Combining different tensorflow runs	145
28.4.3 Identifying outliers	145
28.5 Next steps	145
29 Overview of the marker detection tool	147
30 Modules description	149
31 Option file	151

32 Graphical user interface	153
33 Examples of marker detection	155
34 Overview	157
34.1 Notebook widgets	158
34.2 Data structure	158
34.3 Get widget	158
34.4 View widget	159
35 Installation	161
35.1 Dependencies	161
35.2 Installing from PyPI	161
35.3 Installing for development	161
35.4 Installing from source:	162
35.5 Uninstallation	162
35.6 Install GDAL	162
35.7 Troubleshooting	163
36 Configuration	165
36.1 Configuration widget	165
37 DIAS catalog	167
37.1 Notebook widget	167
38 Extraction	169
38.1 Extract widget	170
39 FOI examples	171
39.1 Version 1	171
39.2 Version 2	174
40 Data access	177
40.1 Parcel information	177
40.2 Time series	178
40.2.1 Show time series	178
40.2.2 Download time series	179
40.3 Background images	180
40.3.1 Show background images	180
40.3.2 Download background images	181
40.4 Sentinel chips	181
41 Contributing to CbM	183
41.1 Using the issue tracker	183
41.2 Pull requests	183
41.3 Bug reports	184
41.4 License	184
42 Pull requests	185

The Joint Research Centre (JRC) Checks by Monitoring (CbM) is as a proof of concept that demonstrates the feasibility of agricultural monitoring systems for the common agricultural policy (CAP) of the European Union (EU) based on Copernicus [Sentinel](#) data on cloud-based infrastructures, particularly the [Data and Information Access Systems \(DIAS\)](#). Its goal is to help Paying Agencies (PA) of Member States (MS) and their technological partners to get started with CbM and to define a set of best practices in context of the [Integrated Administration and Control Systems \(IACS\)](#). The same approach can potentially be applied to the area monitoring system (AMS) of the future CAP (2023-2027). In this page we introduce DIAS and Sentinel in the context of the CAP and we give an overview of the goals, the architecture and the main elements of the solutions developed by GTCAP (JRC CbM). The detailed description of methods and tools is provided in the [JRC CbM technical documentation](#). The code is made available as open source in the [JRC CbM GitHub repository](#).

MAIN ELEMENTS OF THE JRC CBM

The CbM solution proposed by JRC is based on two architecture layers, the backend and the frontend, that provide users with data and tools for their analysis.

- The **backend** server provides the end-points to retrieve data: it includes the physical infrastructure based on DIAS and the routines that process Sentinel data and generates the information needed for the data analytics. It is developed, managed and maintained by a system administrator with expertise in cloud computing and big data analytics. More information on this layer is reported in the [backend documentation](#).
- The **frontend** is the intermediate layer that provides access to the data generated by the backend through standard Application Programming Interface (API). Users can retrieve and process data, both statistics and satellite images, ready to be analysed. They can get data using simple commands without backend expertise. More information is available in the [frontend documentation](#).
- The **data analytics** use data generated by and stored in the backend and made available through the frontend to perform CbM tasks, particularly to confirm/reject the farmer declaration on specific parcels in the context of a CAP scheme. A typical example is the use of time series of Sentinel data statistics extracted per declared parcel to detect markers that can be associated to agricultural activities. More information is provided in the [data analytics documentation](#).

In the JRC CbM system the boundary between the backend, the frontend and the data analytics is fluid. Once new procedures are defined and tested by analysts, they can be integrated into the backend and automatically available to users as starting point for additional processing (e.g. analysis, reporting, classification). JRC CbM software modules are all open source and make use of open standards. The code is available in the [JRC CbM GitHub repository](#). A complete overview of the system is illustrated in the [JRC CbM architecture](#) page.

DIAS AND COPERNICUS SENTINEL

The **DIAS** are cloud-based platforms providing centralised access to Copernicus data and information, as well as to processing tools (processing as a service) closely coupled with the data store. The European Commission has funded the deployment of five **DIAS instances** (CREODIAS, MUNDI, ONDA, SOBLOO, WEKEO) managed under contract with the European Space Agency (ESA) that facilitate and standardise access to the **full archive of Sentinel-1 and Sentinel-2 data**. The Copernicus **Sentinels** are a family of satellite missions for the operational needs of the EU Copernicus earth observation programme. Sentinel-1 provides all-weather, day and night radar imagery, nominally acquired every 6 days with 5 meters spatial resolution. Sentinel-2 provides multispectral high-resolution optical imagery with approximately 5 days revisit time and 10 meters spatial resolution. Their image archive provides free images of agricultural areas since June 2015 for all Europe. Each of the five DIAS competitive platforms also provides access to additional commercial satellite or non-space data sets as well as premium offers in terms of support or priority. Thanks to a single access point for the entire Copernicus data and information, DIAS allows the users to develop and host their own applications in the cloud, while removing the need to download bulky files from several access points and process them locally. A key service offered by the DIAS and essential for all Member States/Paying Agencies is the generation of **Copernicus Application (or Analysis) Ready Data (CARD)**, which are data over the area of interest prepared for a user to analyse without having to pre-process the imagery themselves. It includes georeferenced, calibrated sensor data (Level 2). Sentinel 1 and 2 data are, by default, delivered by ESA through the Copernicus Hub as Level 1 data. Since March 2018, Level 2 (atmospherically corrected) Sentinel-2 data is also made available. For Sentinel-1, this is not the case (yet).

CBM IN THE CONTEXT OF THE CAP

In 2018 **Checks by Monitoring (CbM)** were introduced for area-based support schemes as an alternative to classical on-the-spot-checks ([Article 40a of implementation regulation \(EU\) 746/2018 of 18 May 2018](#)) for the IACS managed by the European Union Member States. The CbM approach is based on the automated analysis on the whole population of aid recipients verifying the consistency of farmers' aid applications with satellite-based evidence. The DIAS provide the relevant functionalities and an optimized use of Sentinel-1 and Sentinel-2 imagery for the continuous detection of agricultural conditions and activities on all the individual land parcels retrieved from the Land Parcel Identification System (LPIS) and the Geospatial Aid Application (GSAA). The information extracted from the Sentinel data (signatures) per individual parcel and date (time series) can be used to observe specific events (e.g. mowing, ploughing) or sequence of events (scenarios) when certain conditions are met (markers). This process results in a simplification of the huge Sentinel data set with a more efficient and largely automated check of all aid applications by Paying Agencies and, potentially, in the assessment and monitoring of the impact of the CAP measures.

JRC SOLUTIONS FOR CBM

The ongoing implementation of the CAP and the development of a CAP post-2020 that comply with new EU commitments on climate change and sustainable development are priorities for the [Joint Research Centre \(JRC\)](#). In this framework, the JRC Directorate D5 and the [Directorate General Agriculture and Rural Development - DG Agri](#) signed an Administrative Arrangement for CAP Implementation Support Activities **CAPISA 2** that covers the period from October 2020 to September 2022. One of the work packages (WP) is on supporting (onboarding) Member States/Paying Agencies on the deployment and use of DIAS cloud solutions for CAP area aid schemes. The GTCAP group in liaison with other JRC teams and in close consultation with DG AGRI:

- validates the specific functional requirements of the CAP on cloud instances and addresses deployment issues including portability;
- develops and operates a repository of shared technical solutions, best practices and algorithms, particularly on the implementation of methods based on earth observation (EO) signals;
- provides training sessions to Member States/Paying Agencies on the use of DIAS for CbM;
- implements and monitors contract for purchase of DIAS services;
- advice on long term cloud-based solutions for CAP monitoring to ensure their future/continued availability.

The JRC solutions for the CbM data analytics are developed in the framework of the the GTCAP institutional project **Outreach**, that builds on the infrastructure development under CAPISA/CAPISA2 to benchmark CbM and, in extension, AMS application contexts. Outreach aims at producing good practices for Member States for the creation of a CbM system offering tools, support and a platform to explore Sentinel data and experiment algorithms to detect agricultural practices. It bundles the processing needs of a large selection of Member States on a JRC managed DIAS platform, in order to lower the technological deployment hurdles. JRC CbM has [substantial differences](#) compared to the ESA project [SEN4CAP](#).

CONTRIBUTING TO JRC CBM

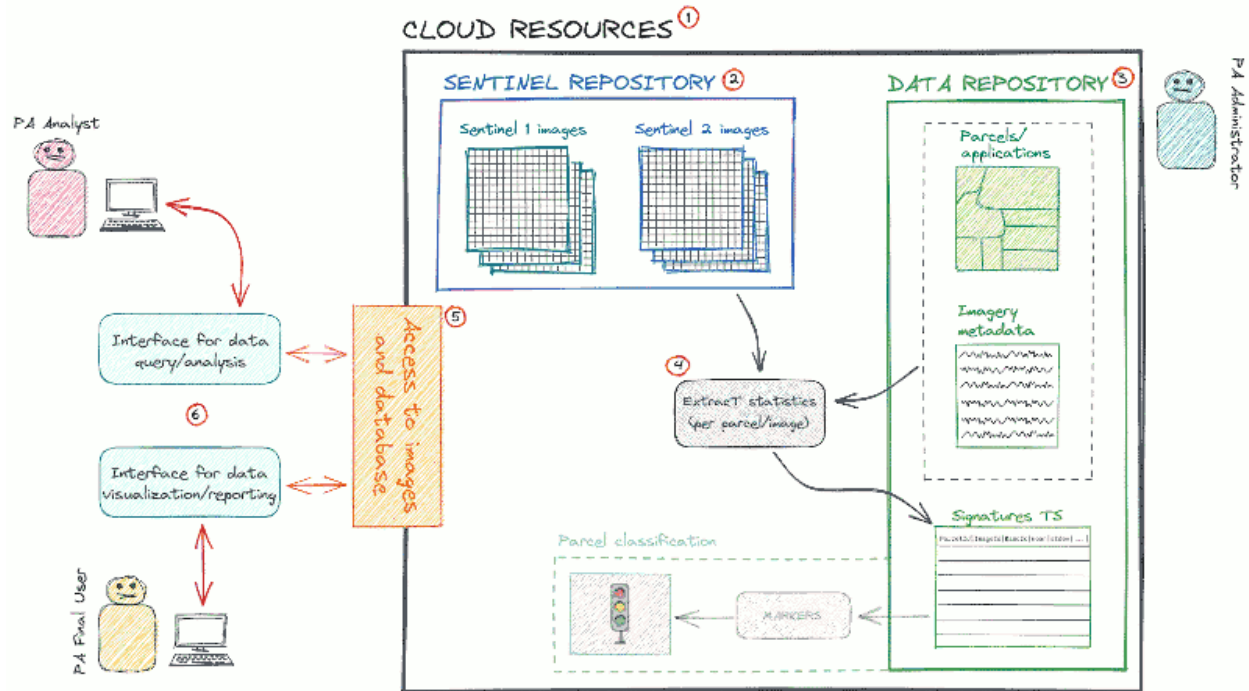
JRC CbM is the result of a collaborative effort between the JRC and many Member States. The code stored at <https://github.com/ec-jrc/cbm> is open to contribution. In the [HOW TO CONTRIBUTE documentation page](#) you can find all the information to raise an issue, report a bug and create a pull request to integrate new improvements. Contact us for if you need more details.

SYSTEM ARCHITECTURE

The page provides an overview on the **logic, concepts and key elements of the JRC CbM architecture** for all the experts of the Paying Agencies (system administrators, analysts, decision makers, project managers and consultants). The aim is to help understanding the approach and the software solutions proposed and assess and plan its operational implementation. The technical details of JRC CbM are reported in the [TECHNICAL DOCUMENTATION](#) and the code is made available in the [JRC CbM GitHub space](#). Here we analyse the requirements of a CbM system and illustrate the solution proposed by the JRC CbM. The goal is to demonstrate the feasibility of the technical implementation of a CbM and that can be used and adapted by MS to create their own system tuned on their specific needs. In the last section (JRC CbM roles), we provide additional information on the different roles in the system.

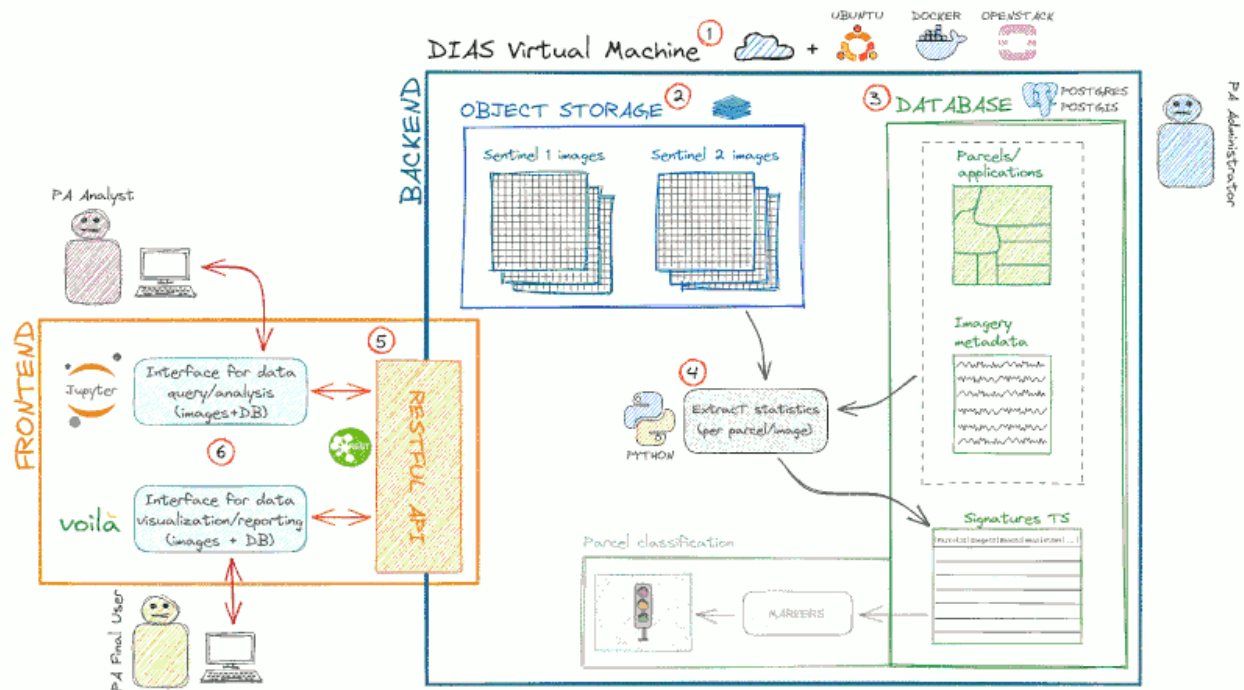
6.1 Scope of the CbM system

The scope of a CbM system is to exploit the time series of Sentinel data to continuously monitoring all the agricultural parcels, i.e. the polygons from the Land Parcel Identification System (LPIS) and the Geospatial Application (GSA) that are associated with a CAP scheme. It generates the minimum required information to confirm/reject compliance with the declared practice and to communicate discrepancies to the farmers in real time, so that they can be amended in due time, if needed. This process is optimized through automation and **reduction of information** to handle the data load (i.e. summarize the spatio-temporal image stack of Sentinel into time series of bands statistics per agricultural parcel). The results feed into a traffic light system for follow up: most of the cases will have a definite classification (green: confirmed, red: rejected), while some of them will require further investigation (yellow: inconclusive). This reduces considerably the situations to be verified with other tools (e.g. geotagged photos, higher resolution satellite images, orthophotos, field visits). The same approach can also be used to set up an area monitoring system (AMS), as requested by the new post 2020 CAP regulation (2023-2027). The main elements of a CbM system are illustrated in *Figure 1*. This architecture is modular and reflects the general requirements of any CbM system.



General architecture of a CbM system.

In Figure 2 we show the proposed solution (JRC CbM) for the technical implementation of the general architecture described by Figure 1.



JRC CbM software platform. Satellite data are made available in the Object Storage of the DIAS and processed in that environment by Python modules. The base layers are stored in a spatial database based on PostgreSQL/PostGIS

installed in the DIAS server, inside the same environment of the satellite image archive. The server is set up and managed using Ubuntu, Docker and OpenStack. Data access is filtered by a RESTful API that feed the user's interface, based on Jupiter Notebook and Voilà.

The JRC CbM system is based on two architecture layers: the **backend** and the **frontend** (in Figure 2, backend and frontend are identified by the blue and the orange boxes). The backend and the frontend support data analytics. The backend server provides the end-points to retrieve data: it includes the physical infrastructure and the routines that generates the information used by the analysts and the decision makers. It is developed, managed and maintained by a system administrator with expertise in cloud computing and big data analytics. An extended introduction to this layer is illustrated in the [BACKEND SYSTEM DEVELOPMENT](#) documentation page. The frontend is the component manipulated by the user: it provides access to the data generated by the backend through standard Application Programming Interface (API). It includes the interfaces and the code for using the data to support typical Paying Agencies functions. Frontend makes data available to analysts without backend expertise. More information are available in the documentation of the [FRONTEND](#). The data generated by and stored in the backend and made available through the frontend is then used by a third layer, data analytics, where the information is used to perform typical CbM tasks (see [DATA ANALYTICS](#) documentation page). In the JRC CbM system the boundary between the backend and the frontend is fluid. Once new procedures are defined and tested by analysts, they can be integrated into the backend and automatically available to users as starting point for additional processing (e.g. analysis, reporting, classification). JRC CbM software modules are all open source and make use of open standards. The code is available in the [JRC CbM GitHub repository](#). The same structure can be operationalized using other software with similar functionalities if the system has to be integrated in an already existing infrastructure. JRC CbM is designed on a cloud-centric basis, but can also be adapted to run stand-alone. The elements of a CbM system and its implementation in the JRC CbM are described below.

6.2 Cloud resources: DIAS infrastructure

The huge **amount of data** (Sentinel-1 and Sentinel-2, parcels) to be stored and the **resources needed to process them** require an hardware infrastructure that can be hardly set up on a local system. Cloud platforms offer the possibility to optimize the hardware resources according to the specific needs and scale to large data volume without the burden of extensive hardware and software maintenance. In addition, direct access to Sentinel data is needed to avoid downloading the images on a local system. Keeping all the data in the same (cloud-based) container (e.g. images, parcels, statistics) is a further advantage for the efficiency of the CbM process and enforce data consistency and accessibility. Using cloud resources, it is possible to create modular processing pipelines that are tailored to the specific needs of the CbM and the technical capacities of the end users. In the JRC CbM, DIAS are used as cloud infrastructure.

6.2.1 What are DIAS

DIAS are five European core computing and storage infrastructure (CREODIAS, MUNDI, ONDA, SOBLOO, Wekeo) provided as Infrastructures as a Service (IaaS). The actual DIAS use by end users, such as MS Paying Agencies, or any other public or private user, is managed through accounts, i.e. subscription services that are costed on the basis of actual use of computing resources (e.g. CPU time, data transfer, additional storage requirements). DIAS fit current and future CbM system needs. In addition to the typical features of cloud infrastructures (e.g. configurable compute resources that can scale to needs and possibility to run parallel tasks when tasks can be parallelized, which is the case for CbM), they grant access to a consistent, complete Sentinel data archive and provision of on-demand standard CARD processing. They are based on open industry standards and built up with core open source modules. DIAS may be used at increasing levels of complexity, e.g. (1) simple CARD downloads, (2) extracting reductions to (3) full data analytics. More information on DIAS setup and configuration are reported in the [technical documentation pages](#).

6.2.2 DIAS Virtual Machines

The DIAS Virtual Machines (VMs) are managed using Linux as operating system ([Ubuntu](#)). Linux bash scripting is used for orchestration and parsing. Reprojection and other basic spatial processing are based on the [GDAL library](#). The DIAS tenant can select and configure VMs for specific functions: permanent VMs (e.g. data base server, Jupyter Hub, RESTful) and transient VMs (use on demand, run large tasks in parallel, tear down). As a first step, we use Openstack resources “marshalling”. [OpenStack](#) is a free, open standard cloud computing platform. It is deployed as infrastructure as a service where virtual servers and other resources are made available to users. It scales horizontally and is designed to scale on hardware without specific requirements. In a second step, we “orchestrate” the resources to execute parallel tasks (for example parcel and chip extraction). We use Docker containerization to ease cross-VM installation. [Docker](#) is an open platform for developing, shipping, and running applications. It uses OS-level virtualization to deliver software in packages called containers. Containers are isolated from one another and bundle their own software, libraries and configuration files. They can communicate with each other through well-defined channels. Docker enables to separate applications from the infrastructure so that the infrastructure can be managed in the same ways you manage the applications. Docker containers behave like specialized VMs. We use Docker Swarm as container orchestration tool, meaning that it allows us to manage multiple containers deployed across multiple host machines. Figure 3 shows the Copernicus DIAS IaaS architecture.

Copernicus DIAS single-tenant cloud IaaS

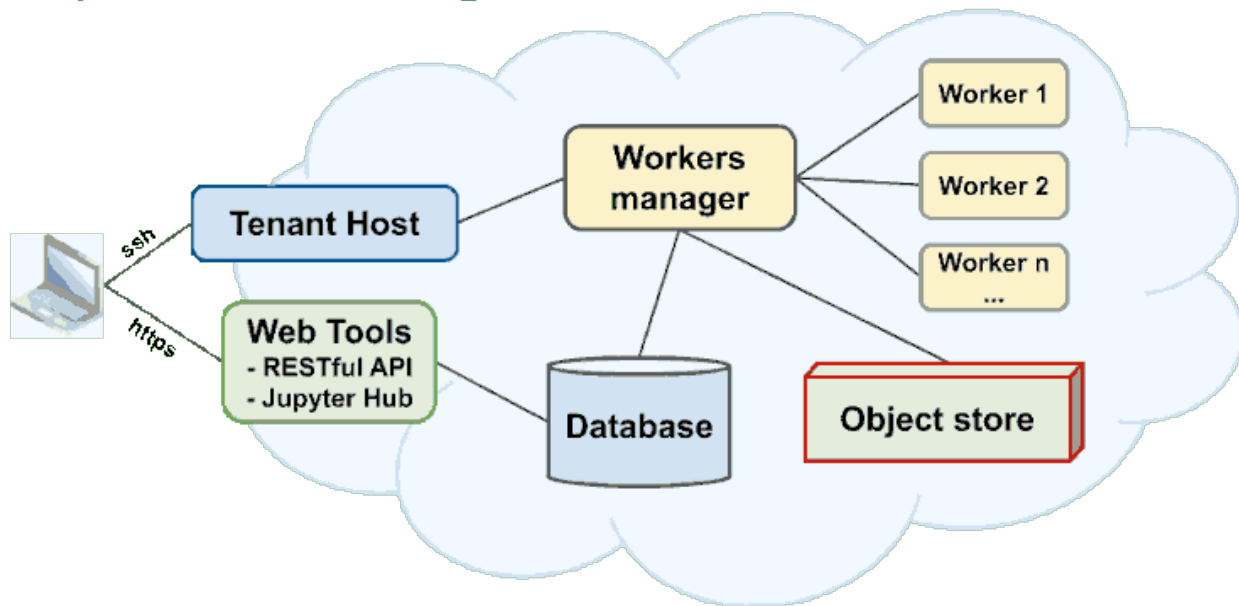


Figure 3. Copernicus DIAS Infrastructures as a Service.

6.3 Sentinel data access: DIAS object storage

Sentinel-1 and Sentinel-2 data are at the heart of the system. The information must be accessible in near real time and for all the agricultural areas under assessment. The huge amount of data, tens of petabytes (i.e. ten million gigabytes), requires **dedicated and performant tools for storage and access**, being a old-fashion flat-file based approach inefficient and not practical in this context. Sentinel imageries should be available to end users as Application (or Analysis) Ready Data (ARD), i.e. in a format that is ready for analysis so that users can work with the images without the burden of complex and time consuming pre-processing steps. Copernicus ARD (CARD) should, as a minimum, include

georeferenced, calibrated sensor data (Level 2). The DIAS object storage made available the Copernicus data in an efficient way.

6.3.1 What is DIAS object storage

The best tool to store and manage immutable “Big Data” blobs that are write once and then read often (e.g. YouTube video, DIAS Sentinel data) is object storage. Object storage (also known as object-based storage) is a data storage architecture that manages data as objects, as opposed to other storage architectures like file systems which manages data as a file hierarchy, and block storage which manages data as blocks within sectors and tracks. Each object typically includes the data itself, a variable amount of metadata, and a globally unique identifier. It is simpler to manage and extend than file or block storage, and it is much cheaper. The access is generally slower, especially since it is not optimized for partial reads. It requires API to transfer to classical file system to be handle as normal file. CREODIAS, MUNDI, SOBLOO and Wekeo all use [S3](#) (AWS, GCS standard) object storage. ONDA uses [ENS](#) (OpenStack Swift). Object storages can manage tens of Petabytes data (e.g. CREODIAS ~ 20 PB, Goggle Earth Engine ~ 85 PB). In JRC CbM, DIAS object storage is accessed through the Python library [Boto3](#).

6.3.2 Copernicus ARD generation

DIAS instances offer a Processing as a Service (PaaS) solution for generating Copernicus Application Ready Data (ARD). ARD is stored in the S3 object store of the DIAS, in public or private buckets. Sentinel-1 and Sentinel-2 data are, by default, delivered by ESA through the Copernicus Hub as Level 1 data. Level 2 (atmospherically corrected) Sentinel-2 data is also made available. For Sentinel-1, this is not the case (an anomaly that may be resolved in the future). Sentinel-1 ARD processing is done by JRC CbM backend to generate calibrated geocoded backscattering coefficients from Level 1 GRD (CARD-BS) and geocoded coherence from Level 1 SLC S1A and S1B pairs (CARD-COH6). If ARD does not already exist in DIAS archive for a given country, it has to be generated by the system. After a bulk generation of ARD is done until the last available date, a backend batch process will ingest (with a typical delay of 1-2 days) the new images that are made available in the ESA hub and transferred to the DIAS. ARD generation process is not yet part of JRC CbM documentation, but it is available on request. More information on JRC CbM CARD data are reported in the [technical documentation pages](#).

6.4 Data repository - PostgreSQL/PostGIS

In addition to Sentinel images, the system has to manage and process other control data sets in order to check farmers' declaration. The first one is the **agricultural parcels**: polygons from the Land Parcel Identification System (LPIS) and Geospatial Application (GSA) that are associated with a CAP scheme, managed by the PA. It is a spatial layer with associated non-spatial attributes. The number of polygons can be in the order of hundreds of thousands. The second layer is the **metadata of Sentinel images**. This data set is derived from the Sentinel archive and includes the key information of each image (e.g. id, date, extent) and make images retrieval and identification faster. Finally, the system has to store the **signatures** (or statistics) of Sentinel bands extracted per polygon (also called *reduction* in the context of JRC CbM). The spatial nature of these data sets, their well defined relations, their size and the need for scalability (signature data set increase quickly over time), the importance of data consistency and the need to make the data available to multiple tools and users point to spatial relational databases (SRDBMS) as the state of art technical solution. In an operational national CbM system, a relational database can be used not only to support extraction and management of signatures data, but is also a good candidate as central repository for all the information relevant for the CAP process. In such a context, the technical solution depends on the specific goals and constraints of each Paying Agency (parcel classification according to the traffic light system is included in *Figure 1* with transparency as it is just an option for an extension of the CbM). In the JRC CbM solution, the proposed software for data repository is the open-source Relational DataBase Management System (RDBMS) [PostgreSQL](#) with its spatial extension [PostGIS](#). They represent the state of art SRDBMS, which can efficiently manage very large spatial (and temporal) and non spatial datasets with a complex structure in a multi-user and secure environment. In JRC CbM, Sentinel metadata are parsed into PostgreSQL/PostGIS dias_catalogue table, while data on parcels are ingested by the users. Signature data are

generated by the JRC CbM backend processing. JRC CbM architecture integrates the database into the DIAS server, in the same environment of the object storage that manage Sentinel data, but it can also be deployed on a local server. The information about the database content, structure, description, connection, management and how to retrieve data is provided in the [technical documentation pages](#).

6.5 Extraction of statistics - Python processing

The key information to assess compliance of a declared practice is the **time series of statistics** (mean, max, min, median, 25 quantile, 75 quantile, standard deviation, number of pixels), also called signatures, of Sentinel bands for each polygon. It is generated by a **spatial intersection** between the Sentinel-1 and Sentinel-2 bands and the polygons from the Land Parcel Identification System (LPIS) and Geospatial Application (GSA). The most relevant bands for CbM are Sentinel-2 B02, B03, B04, B08, B5, B11 and Sentinel-1 VV, VH. Sentinel-2 scene classification map (SCL) is also important to identify (and ignore) images where the signal is affected by clouds. The image is selected using the image metadata of the Sentinel archive. The availability of Sentinel CARD ensures that the spectral and temporal information (i.e. time series) can be extracted consistently. Although it is technically feasible to generate a CARD time series on demand, it is often convenient to extract large sets of parcels, for pre-selected bands, in a batch process. Cloud infrastructures, and particularly DIAS, are specifically tailored to these kind of tasks, since extraction is typically a massively parallel processing step that can benefit from the use of multiple machines. Since CbM covers 100% of the territory, it needs to be supported by smart analytics that filter out the anomalies quickly and consistently. The signal statistics can be used in such analytics, e.g. to support outlier identification, detection of heterogeneity, marker generation. The extraction has to be an automated and performant process given the amount of elements to be calculated. To give an order of magnitude, if the parcels are 500,000, the Sentinel-2 images are 73 (an image every 5 days in a single year) and the bands are 8, the number of signature records generated is about 300 millions. Other ancillary data may be required to explain signal artefacts (DEM, weather parameters, etc) and the system should be able to manage them as well. In the JRC CbM, the backend processing of Sentinel data extraction to reduce the spatio-temporal image stacks of ARD to parcel time series is based on [Python](#), a general-purpose programming language with simple syntax and code readability that make coding easy and efficient. It has a large set of specific libraries, particularly for data science. Extraction is set up as an automated process that:

- finds the oldest image that is not yet processed
- transfers the image bands from the S3 store onto local disk
- queries the database for all parcels within the image bounds
- extracts the statistics (, , min, max, p25, p50, p75) for the bands of the image
- stores the results in the time series database table
- clears the local disk

In the JRC CbM, Python is used not only for data extraction but is the syntactic glue of the whole system, especially in the frontend. The [numpy](#), [pandas](#), [geopandas](#), [rasterio](#), and [osgeo](#) packages are widely used in many of the system modules. The [psycopg2](#) library is used to connect with the database and retrieve and manipulate the data. In a complete CbM system, the backend can also use Python to detect agricultural practices linked to farmer's declaration processing. The output then should feed into traffic light system for follow up (e.g. the need for complementary information). In principle, there is no reason why parcel extraction could not be offered as a PaaS, and this might be an option for the future, simplifying CbM systems.

6.6 Data access - RESTful API

Analysts need to **access the data generated by the backend** (i.e. CARD data, if not already in DIAS archive, and parcel time series of Sentinel spatio-temporal image stacks) to feed into post-processing. While direct access to tables, spatial features and images stored and managed in the cloud infrastructure is possible with a DIAS account and recommended for developers/analysts that can retrieve and visualize data with specific tools (e.g. [Python Jupiter Notebooks](#) and database clients like [PgAdmin](#) or [QGIS](#)), it exposes the system to multiple issues. First of all, it requires advanced technical skills that analysts and decision makers do not always have in their background (for example SQL, the database language). Then, directly open the system to many users exposes it to potential security risks. Finally, inexperienced users could try to run processing that is not optimized and thus consuming abnormal resources. An intermediate layer that offers pre-defined functionalities and formats to consume data from the backend, for example the time series of statistics and images for a user-defined parcel, can help to address these problems. The final users can use this intermediate layer that provide predefined functionalities to extract data based on a limited and controlled set of parameters. This ensures performance and security by preventing poorly designed resource-intensive image retrieval and database queries and facilitates access to basic users with no knowledge of tools for data extraction (e.g. SQL) as it can serve standard queries and abstracts more complex tasks. Analytical algorithms and markers are designed and tested in the analytical layer that gets data through the frontend, but when methods are consolidated the system should allow their implementation in the backend. In the JRC CbM system, analysts access the backend data sets through a RESTful API. A RESTful API is an application programming interface (API) that conforms to the constraints of REST architectural style. An [API](#) is a set of definitions and protocols for communicating what you want to that system so it can understand and fulfil the request (see Figure 4). [REST](#) stands for representational state transfer. It is an architectural style for distributed hypermedia systems: it defines a set of constraints for how the architecture of an Internet-scale distributed hypermedia system should behave. An essential character of the RESTful services is that they follow standard calling conventions, which allow their use both in interactive testing and scriptable machine access. RESTful services are further described in the [technical documentation pages](#).

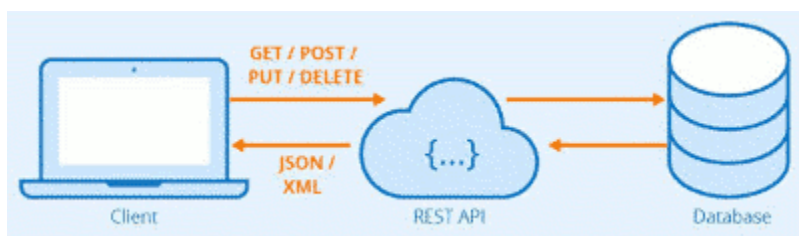


Figure 4. Schema of the interaction between users and the database through the RESTful API.

In the JRC CbM system, the RESTful API provides abstracted access to database tables, to sub-selections of S3 stored CARD data and to advanced server-side processing routines. Examples of RESTful requests in JRC CbM are:

- basic information retrieval on parcels (`parcelByLocation`)
- parcel time series statistics (`parcelTimeSeries`, `parcelPeers`) [fast]
- image chip selection, for visualization (`chipByLocation`, `backgroundByLocation` (Google, Bing and orthophotos via WMTS)) [slow]
- image chip selection, full resolution GeoTIFFs (`rawChipByLocation`, `rawChipsBatch`, `rawS1ChipsBatch`)

RESTful can be integrated in scripts, automated reports, and Jupyter notebooks via Python requests. Another option for accessing data is the use of a [Jupyter Hub](#) server configured to hide the intrinsic access protocols and supporting interactive scripting and data visualization. A mixture of direct access and RESTful services is also possible, for example with a Jupyter notebook running on the Jupyter Hub that uses both. Finally, an alternative use pattern would be to transfer the time series database to a dedicated server set up outside the DIAS cloud infrastructure, for instance, into the PA's ICT infrastructure. RESTful services running on DIAS can then be used to maintain access to the DIAS CARD data store to support on demand generation of image fragments, for instance.

6.7 Interfaces for data use - Jupiter Notebooks

In addition to the intermediate layer to access the CbM data, the data analytics layer requires **automated tools and interactive interfaces** to classify, visualize and post-process the information generated by the system. These interfaces must be coherent with their technical skills and goals. Analysts should be able to interact with data using the most common geospatial processing languages and libraries, while final users need visual output that can be interpreted (e.g. graphs, images, reports) and tools to support the compliance verification procedure. Users can access data directly or through the RESTful API. In both cases, they need an interface to visualize (as graphs, maps, tables) and analyse (applying for example statistical algorithms, machine learning) data according to their needs. In The JRC CbM system, the main tool used for this task is Jupiter Notebooks. [Jupyter Notebooks](#) are documents that contain live code (particularly Python), equations, graphics and narrative text. The open-source interface offers a web-based interactive computational environment for, among the others, data cleaning and transformation, numerical simulation, statistical modelling, data visualization, and machine learning. Python is used to get, process and display the data. In particular, analysts are interested in the identification of markers. Markers are sequence of signatures (statistics derived from Sentinel bands for each parcels) that can be associated to the agricultural practices declared by the farmer according to CAP schemes (when Sentinel data can pick up the signature of the events and patterns that relate to the practice). Jupiter Notebooks offer an ideal environment to explore and test new markers and analytical approaches. Once established by analysts and consolidated, markers can be integrated in the backend and used to confirm/reject compliance with the declared practice, feeding into traffic light system for follow up (e.g. the need for complementary information) by decision makers. In the JRC CbM a flexible marker detection framework has been developed to facilitate the data analytics.

6.8 JRC CbM roles

JRC CbM considers roles. Not all roles need to work with all modules. There are three main classes of roles:

- ICT experts create and maintain the infrastructures with its required server components, e.g. monitoring CARD production, run extractions. This backend role can be managed by one person per MS (or even DIAS).
- Data analytics experts program and run core analytics (e.g. extraction, machine learning, development of markers). They can be both internal and external to the PA.
- Data consumers extract, cross-check, verify, combine, decide and report.

BACKEND

In this page, we explain how to deploy the backend component (software and hardware infrastructure) of the JRC CbM system. An introduction to the system is described in the [INTRODUCTION TO THE JRC CbM ARCHITECTURE](#). Here you find the reference to the technical material, to the instruction and to the code needed to set up a DIAS virtual machine, create a database to store parcel information and image metadata, access/generate Sentinel Copernicus Analysis Ready Data (CARD) for your area, process it to populate the signature table (satellite bands statistics per parcel unit) and provide interfaces for CbM users to visualize and analyse the relevant information. The use of the information generated by the backend is documented in the [FRONTEND PAGES](#) and the [DATA ANALYTICS PAGES](#).

7.1 Goals

The main functions of the JRC CbM backend are:

- to set up the hardware needed to run the JRC CbM system
- to generate Application Ready Data (ARD), if not already in DIAS archive
- to reduce the spatio-temporal image stacks of ARD to parcel time series
- to provide server components and APIs for data access

This component of the JRC CbM is the basis for the other tasks and is preliminary for the development of CAP-related data analytics applications. Direct assistance is also provided by JRC to support Member States/Paying Agencies in the deployment and customization of the system that is the final expected output of this activity.

7.2 General requisites to set up the CbM system

In the framework of the Outreach project, a cloud infrastructure (based on CreoDIAS) to experiment the functionalities of the system has been created by GTCAP with the backend component developed and managed by JRC. MS can use dedicated API to explore and analyse Sentinel data extracted for they declared parcels. In this project, users do not have to install anything and can run the example code for data retrieval and manipulation stored in this repository, particularly using Python and Jupiter Notebooks. Instead, to start a dedicated CbM system for a PA is essential to have:

1. Computing resources
2. Copernicus Analysis Ready Data (CARD) (Sentinel 1 and 2 data for the study area)
3. Agricultural parcel data (typically, declared parcels from the Land Parcel Identification System (LPIS) and the Geospatial Aid Application (GSAA))

The first two requisites can be achieved using one of the five Copernicus Data and Information Access Services (DIAS) available (CREODIAS, WEKEO, SOBLOO, MUNDI, ONDA).

7.3 How to create a CbM infrastructure

7.3.1 Links to the technical documentation

1. [CONFIGURE DIAS VIRTUAL MACHINES](#)
2. [INSTALL REQUIRED SOFTWARE](#)
3. [SETUP DATABASE FOR CbM](#)
4. [POPULATE THE DATABASE](#)
5. [EXTRACT SIGNATURE TIME SERIES](#)
6. [BACK UP THE DATABASE](#)
7. [BUILD THE RESTful API](#)

7.3.2 Set up a DIAS virtual machine

The virtual machines (VMs) of DIAS Infrastructure as a Service (IaaS) are emulations of fully functional computational instances. Users obtain VMs with full root access and can define different parameters and characteristics, including machine type (physical or virtual), RAM, CPU, storage quantity and type, operating system, middleware components and virtual networks connected to the machine. In the documentation we describe how to connect to the machine and to install the software required to run the JRC CbM.

7.3.3 CbM database

The CbM database stores the base layers (particularly, parcels and image metadata, the latter generated by the DIAS) and the signatures (i.e. number of pixels, mean, std, min, max, 25th percentile, 50th percentile, 75th percentile) that result from the intersection of satellite image bands and parcels. [PostgreSQL](#) with its spatial extension [PostGIS](#) are used as reference database software. In the documentation page, we introduce SRDBMS, we describe the database that has been set up for the JRC CbM. We describe how to create the database structure and populate it, and how access, retrieve, export and backup data stored in the DB.

7.3.4 CARD

Copernicus Analysis Ready Data (CARD) in CbM are Sentinel Images available in a format that is ready for analysis so that users can work with them without the burden of complex and time consuming images pre-processing steps, which include, as a minimum, georeferenced, calibrated sensor data (Level 2) over the whole area of interest. Although it is technically feasible to generate a CARD time series on demand, it is often convenient to extract large sets of parcels, for pre-selected bands, in a batch process. In the documentation we explain how to generate CARD (if not already available in the DIAS), how to make it available in the S3 storage and how to access it.

7.3.5 Extract signatures

The generation of bands signal statistics (signature time series) for each agricultural parcel is used to support CbM analytics, reducing the complexity and huge amount of satellite data covering 100% of the territory to a simpler and manageable data set (this process in the context of JRC CbM is also called *reduction*). Examples of typical tasks based on signature time series are marker generation, outlier identification, detection of heterogeneity. At all times, the direct link to the CARD inputs is maintained, which means that the source of the particular indicator can be (automatically) retrieved, e.g. for more detailed inspection. In the documentation we illustrate how to configure the system and run the Python procedure that extracts signature statistics and populate the database.

7.3.6 Create a RESTful API

In order to facilitate the access to parcel time series and satellite images, for analysts and final users who do not have a DIAS account, and particularly for they with limited technical background and no knowledge of SQL, a RESTful API with [Flask](#) can be build. It acts as an intermediate layer that provides predefined functionalities to extract data based on a limited and controlled set of parameters. This ensures performance and security. In the documentation we explain how to deploy and configure the RESTful API docker container.

7.4 Deployment of the CbM system

There are several steps to set up the core components for CbM that require different types of technical expertise.

1. Setup server applications
 - Docker (containerization system)
 - Postgres database with PostGIS extension
 - Jupyter (interactive analysis and visualization environment)
 - Restful API (intermediate layer to access and use Copernicus data and the database)
2. Adding data to the database
 - Parcels data
 - CARD Metadata and other setting data
3. Process Sentinel data to derive relevant information
 - Parcel stats extraction routines
 - Machine learning algorithms
 - Analytical routines (e.g. markers detection)
4. Analyzing and reporting
 - Using Jupyter Notebooks to explore and analyze data
 - Generate reports and other outputs to classify aid applications

7.5 Expertise required

To complete the tasks and set up a running system, Member States/Paying Agencies need the support of experts with

- strong skills in Linux Virtual Machine (VM) configuration and administration
- strong skills in the use of relevant open source components for EO image processing and geospatial data analysis (as a minimum: GDAL, web map services - WMS)
- strong skills in programming and scripting (preferably python and relevant data processing libraries) for geospatial analysis
- good working knowledge of PostgreSQL/Postgis and SQL
- working knowledge of cloud or cluster computing solutions (Openstack, Docker, Docker Swarm) for VM orchestration for parallel computing
- working knowledge of server interfaces for data access and analytics (Jupyter Hub, RESTful)
- working knowledge of Sentinel data specifications, processing toolkits, and use case requirements in agriculture
- good English communication skills if direct support from JRC is needed

In the future, the backend development may be impacted by Copernicus programme decisions (e.g. ARD production) and adoption of novel approaches (Kubernetes, Dask, GPU).

7.6 Enhancement Proposals

JRC CbM backend evolves very quickly with improved or new features. In this section we list some improvements that we want to integrate into the system in the future. If you want to suggest a new enhancement or you want to contribute with new modules, in the next section you find the links with the instructions to do so.

7.6.1 Terrain Correction

Introduce Radiometric Terrain Correction in CARD-BS processing: RTC normalizes for SAR viewing configuration, allowing better comparison of ASC and DESC and multiple orbit data.

7.6.2 S2 Multi-band

Implemented multi-band and index extraction for Sentinel-2 L2A: significant for marker generation. Database size may become an issue, thus, this is strictly done for cloud free parcels only.

7.6.3 Dask tests

Test Dask and decide whether it can replace (some of) the Docker based parallelisation: Dask allows for parallel processing of large and deep image stacks, e.g. for extraction and local image processing tasks (e.g. segmentation).

7.6.4 ML Revision

Revisit the 2018 machine learning code (DNN in tensorflow) and update with latest best-practice: the old code needs to be revisited and implemented as a “crop” marker. Documentation to be completed. Best to start with S1 signatures.

7.6.5 RESTful meteo

Create a RESTful service that accesses open meteorological “now cast” data (e.g. ERA5, GFS): Interpretation of time series often requires temperature and precipitation data to understand spurious trends.

FRONTEND

This page lists the frontend functionalities available to analysts, organized per tasks with the links to the technical documentation. These tools are designed to provide an interface to the CbM backend to explore Sentinel data. They help to develop new methods to detect agricultural practice. An introduction to the JRC CbM system based on a DIAS cloud infrastructure is provided in the [OVERVIEW OF THE SYSTEM ARCHITECTURE PAGES](#). The frontend is based on the data generated by the backend, documented in the [BACKEND PAGES](#), and contributes to the tools and methods to process data and take decisions, documented in the [DATA ANALYTICS PAGES](#).

8.1 Goals

The main functions of the JRC CbM frontend for data analysis are:

- to integrate backend results into the Paying Agency (PA) workflow
- to support analytics and development of methods
- to facilitate the design and application of marker analysis for decision support
- to provide access to relevant ancillary reference data

While the CbM frontend can potentially access backend data directly, all the examples provided make use of the RESTful API set up as intermediate layer between users and data. The examples are based on Jupyter Notebooks and Python. Some mature frontend functionality of generic interest to many users may be integrated as backend logic, once they are created and tested.

8.2 Links to the technical documentation

RESTful API functionalities

1. [RESTful API: ACCESS TIME SERIES](#)
2. [RESTful API: ACCESS IMAGE CHIPS](#)
3. [RESTful API: SUPPORT TO DATA ANALYTICS](#)
4. [RESTful API: USE OF POST FOR ADVANCED SELECTIONS](#)

CbM Python package

1. [Python CbM: OVERVIEW OF THE PACKAGE](#)
2. [Python CbM: INSTALLATION OF THE PACKAGE](#)
3. [Python CbM: CONFIGURATION OF THE PACKAGE](#)
4. [Python CbM: TRANSFERT OF DIAS CATALOG METADATA](#)

5. Python CbM: EXTRACTION OF STATISTICS
6. Python CbM: ASSESSMENT OF FOI HETEROGENEITY AND CARDINALITY
7. Python CbM: GET TIME SERIES, CHIPS AND BACKGROUND IMAGES

8.3 RESTful API

JRC RESTful (see the [introduction to JRC CbM architecture](#) for more info) service requests have predefined logical query names that need to be configured with a set parameters. The response is a JSON formatted dictionary with the data requested by the user, extracted from the [JRC CbM database](#) or from the Sentinel images stored in the DIAS object storage. In the documentation pages you find the available queries with use examples.

In the [Time series documentation page](#), you can see an example of the use of the *parcelByLocation* command to find a parcel ID for a given geographical location, *parcelTimeSeries* to get the signature time series for a parcel ID, and *parcelPeers* to retrieve parcels with the same crop type as the reference parcel within a certain distance. The documentation shows how to integrate these RESTful queries in a Python script to generate products such as graphs.

In the [Image chips documentation page](#), we describe the use of the *chipByLocation* command to generate a series of extracted Sentinel-2 LEVEL2A segments of 128x128 pixels as a composite of 3 bands, *backgroundByLocation* to extract a high resolution overview of the parcel situation based on Google and Bing (or Virtual Earth) background image sets (does not depend on DIAS S3 store), *rawChipByLocation* to generate a series of extracted Sentinel-2 LEVEL2A segments of 128x128 (10m resolution bands) or 64x64 (20 m) pixels as list of full resolution GeoTIFFs. A more structured example shows how to integrate these RESTful services with some more advanced processing concepts that help build up to more complex logic in the next steps. https://jrc-cbm.readthedocs.io/en/latest/dias4cbm_use.html In the [Data analytics documentation page](#), we work out a case study for CbM using the RESTful services. For a limited area of interest, we illustrate how to select all parcel IDs, then for each of them how to check the extracted signature statistics. Given specific criteria, we show how to sort on expected heterogeneity of the parcels to check if it is meaningful with respect to the crop stage. For the top 10 heterogeneous parcels, we explain how to extract chips for further testing. This documentation page is still **under development**.

In the [POST documentation page](#), we show how to use [POST methods](#) with RESTful services, instead of GET methods, to perform tailored made selections. Using POST has the additional advantage that it is much easier to provide request parameters as more complex structures, e.g. including lists. In the documentation we present to script that are based on POST methods. In the example based on *rawChipsBatch*, you can see how to extract a set of image chips. In this case the parameters are lists of detailed references to individual chip selections that you want to collect from the server and that are then extracted on parallel Virtual Machines on the server. The script screens for cloud-free bands of interest. It generates an NDVI map and scales the result to a byte image with a colour palette of choice. The example based on *rawSIChipsBatch* is analogue to the *rawChipBatch* query. This script retrieves Sentinel-1 chips setting the date range selection exploiting the convenience of POSTing JSON structures to pass on the request parameters.

8.4 Python CbM package

We created the **CbM Python library** to provide an easy and organized way to run a variety of different tasks for CbM using [Python programming language](#). In the [Overview page](#) we introduce the functions that part of the CbM package and we describe its sub-package *ipycbm*, focussed on interactive graphical configuration panels and data visualization tools for [Jupyter Notebooks](#).

In the [Installation page](#) and in the [Setup page](#), we document how to install the package and its dependencies, and how to configure it, while in the [DIAS catalogue page](#) we describe how we implemented a metadata transfer step, which makes metadata available in a single, consistent, database table *dias_catalogue* across the various DIAS instances in order to minimize portability issues. This page is still **under development**.

In the [Extraction of statistics page](#) we explain how to configure the connection parameters for the database and the S3 storage and then how to use the `cbm.extract.s2` command to extract statistics in a given time range. We show how to use `ipycbm.extract()` to activate a graphical interface to run the extraction.

In the [Feature Of Interest \(FOI\) assessment page](#) we document how to verify that the declared parcels correspond to homogeneous areas on the ground. In JRC CbM a FOI is the area that represents an homogeneous surface, which means that in FOI polygons there should be only one type of land use on the same type of land cover, and consequently that it is observable by the Sentinel as a meaningful individual entity (i.e. the signal of the pixels that are part of the FOI is similar). To verify that this condition is met, you can use the result of an image classification algorithm and verify that each unit does not contains clusters of heterogeneous pixels. We illustrate two methods provided by the CbM Python package: one based on area calculation (version 1) and one based on cluster size calculation (version 2). We show how to use this tool with a graphical user interface through the `ipycbm.foi()` command.

In the [Get module page](#), we present the code to use GET to retrieve a set of information. With `cbm.get.time_series.by_pid` you can get the signal time series of a band for a given parcel. With `cbm.get.chip_images.by_pid` you can download the chip image that corresponds to a given parcel. With `cbm.get.background.by_location` and `cbm.get.background.by_pid` you can download a background image (orthophoto) by selected location or parcel id. You can also learn how to use `ipycbm.get()` function to active an interactive Jupyter Notebook widget to get data from different sources, with variety of different methods (coordinates, parcels ids, map marker, polygon).

8.5 Expertise required

Analysts in Member States/Paying Agencies that work on the CbM using the frontend tools must have the following expertise:

- strong skills in Python coding
- working knowledge of geospatial data processing
- working knowledge of remote sensing
- working knowledge of Sentinel data specifications, processing toolkits, and use case requirements in agriculture

Developers that creates the frontend services must have the following expertise:

- strong skills in server interfaces for data access and analytics (Jupyter Hub, RESTful)

DATA ANALYTICS

The JRC CbM data analytics is the environment where the information generated by and stored in the backend and made available through the frontend is used to perform typical CbM tasks with the goal to confirm/reject the farmer declaration on specific parcels in the context of a CAP scheme. In this page we describe the main JRC CbM data analytics tools developed so far, particularly for the identification of agricultural activity based on marker detection and for advanced data reporting. Here you can find the links to the relevant documentation of each tool with examples of the use of the data generated by the JRC CbM system. An introduction to the JRC CbM system is provided in the [OVERVIEW OF THE JRC CbM SYSTEM ARCHITECTURE PAGES](#). Data analytics is based on the data stored in the JRC CbM backend, described in the [BACKEND PAGES](#), and accessed through the JRC CbM frontend, documented in the [FRONTEND PAGES](#).

9.1 Goals

The main functions of the JRC CbM data analytics are:

- to provide a set of tools and a simple interface for decision makers with limited technical background in coding and remote sensing for the analysis of CbM data
- to enable analysts to develop and apply algorithms for the detection of agricultural practices
- to ensure full reproducibility of decisions
- to build out an augmented application logic

9.2 Links to the technical documentation

1. [MARKER DETECTION](#)
2. [CALENDAR VIEW](#)
3. [FOI ASSESSMENT](#)
4. [MACHINE LEARNING](#)

9.3 Marker detection

One of the key tasks in CbM is the identification of agricultural activities at parcel level. We created a flexible and extensible computational framework based on Python that analyse Sentinel-1 and Sentinel-2 time series generated by the backend for each parcel. This tool detects markers (observations of a relevant bio-physical change of state, made on the Sentinel signal, on a given date) that can be associated to agricultural practices. The processing architecture is built on four main modules: 1) import of band statistics, 2) data preprocessing, 3) identification of relevant changes in the signal time series (marker), 4) aggregation of markers from different sensors and association with land management practice, through the relevant bio-physical stages. The code, stored in the [JRC CbM GitHub repository](#) is constantly under development and new features are frequently added.

In the [Overview page](#), we illustrate the goals of the tool, the target users, the technical skills needed to use it and how it fits into the CbM logic. We also introduce the main modules and the structure of the code.

In the [Modules page](#), we provide a detailed description of the 4 main modules and how they interact to produce the final output. Links to the Python code helps to explore the logic of the framework.

In the [Option file page](#), we describe in details what option files are, their structure and how to create them, with links to practical examples of settings, illustrated by diagrams.

In the [GUI page](#), we present the graphic user interface that we created to facilitate the interaction with the tool for user with limited knowledge in Python coding.

In the [Examples page](#), we list the Jupiter notebooks that we created with practical examples of marker detection.

9.4 Applications

In this section we show some complex workflows that implement use cases. Here the frontend tools for analysts are used to generate examples of typical outputs or carry out necessary tasks for the CbM process.

In the [Calendar view page](#), you learn how to use the `run_calendar_view.py` script, based on the `calendar_view` Python package, to download, process and display Sentinel-1 and Sentinel-2 data products for specific parcels. The graphical outputs has been designed and optimized to provide an immediate and intuitive access to both temporal and spatial dimensions of Sentinel-derived data, including time series of image statistics (per parcel) and structured sequences of Sentinel chips.

The [FOI assessment page](#) describes the concept of Feature of Interest in the context of the JRC CbM and its characteristics. It describes the tests that can be run to verify that a FOI (area of declared parcel) corresponds to an homogeneous area and that it does not contain non-agriculture features.

In the [Machine learning \(ML\) page](#) we discuss the use of ML TO check the agricultural parcels information given in the farmers' declarations by generating relevant markers that confirm compliance with the scheme under which the declaration is made. ML can exploit the information provided by the continuous, territory-wide Copernicus Sentinel. We show examples that are complementary to classification, applying machine learning to the parcel averages. The documentation explains how to prepare the data set, running and testing ML and checking results.

9.5 Expertise required

Analysts in Member States/Paying Agencies that work on CbM data analytics:

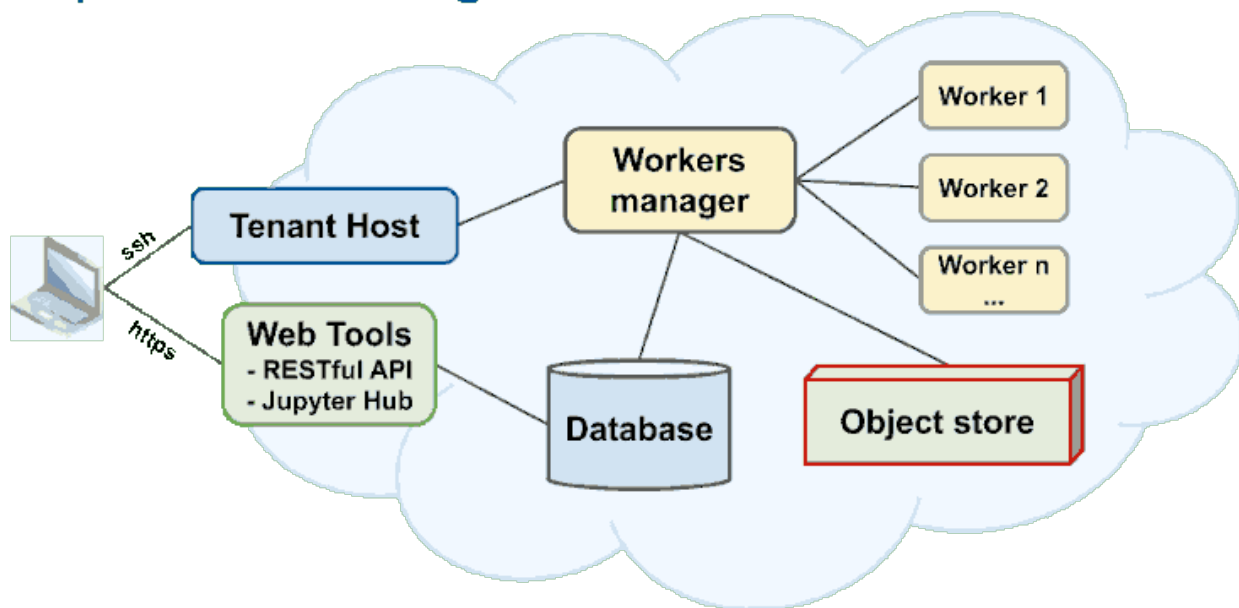
- working knowledge of Python coding
- knowledge of CAP schemes
- knowledge of basic principles of agriculture monitoring with remote sensing

SETUP DIAS VMS

Access and configuration for DIAS virtual machine instances.

The most common Virtual Machines (VMs) structure of DIAS IaaS is as shown in the below figure. VMs are emulations of fully functional computational instances. They are based on computer architectures and provide functionality of a real physical computer. Users obtain VMs with full root access and can implement specialized hardware, software, or a combination based on their needs.

Copernicus DIAS single-tenant cloud IaaS



Simplified structure of DIAS IaaS

In a DIAS infrastructure users can define different parameters and characteristics of the VMs, including machine type (physical or virtual), RAM, CPU (vCores), Storage quantity and type, Operating System, middleware components and Virtual Networks connected to the machine. The parameters of the VMs depends on the processing needs of the specific project and the designed structure. An example of configured parameters of each VM can be found in the following table.

VMs	vCores	RAM	Storage	Notes
Database	4	16 GB	500 GB	Postgres database with PostGIS extension.
Worker(s)	8	16 GB	60 GB	The number of workers depends on the processing needs.
Workers manager	4	8 GB	60 GB	Also a worker VM can be configured to be a manager VM.
RESTful API	4	8 GB	40 GB	Can be combined with Jupyter server.
Jupyter	2	4 GB	40 GB	For a single user jupyter environment.
Jupyter Hub	8	32 GB	100 GB	For up to 10 active jupyter Hub users.
Tenant host	1	4 GB	8 GB	Can be combined with workers manager.

Recommended OS for the VMs is **Ubuntu 18.04 LTS**.

Some VMs can be combined e.g. the RESTful API VM with Jupyter VM or Worker Manager VM with one of the workers VM, but it is recommended to keep the database VM separated and secured with regular backups.

10.1 Connecting to the ‘tenant host’ vm via SSH

From Linux (recommended):

To be able to remotely connect to an SSH server an SSH client program is needed. It can be installed on Ubuntu with the following command:

```
sudo apt install openssh-client
```

To connect to the remote computer, the hostname/domain name or IP address of the remote computer (provided by the service provider) is needed.

To connect with SSH using Ubuntu, in the Home directory make a hidden folder if it doesn't exist “.ssh”, with the command:

```
mkdir ~/.ssh
```

In .ssh make a text file “config” and a folder “keys”

```
touch ~/.ssh/config
mkdir ~/.ssh/keys
```

Move the key file to the keys folder

```
mv ~/YOURKEYLOCATION/NAMEOFTHEKEY.key ~/.ssh/keys/NAMEOFTHEKEY.key
```

and change the permission of the file to 600

```
chmod 0600 ~/.ssh/keys/NAMEOFTHEKEY.pem
```

Open the “config” file with a text editor, e.g.

```
nano ~/.ssh/config
```

and add the following lines. Configure with the server information (HostName and User) and change the NAMEOFTHEKEY with the filename of the key.

```
Host bastion_vm
  HostName 111.111.111.111 # or MYHOST.com
  User USERNAME
  IdentityFile ~/.ssh/keys/NAMEOFTHEKEY.key
  ServerAliveInterval 10 # number in seconds to wait if the connection is lost
  VisualHostKey yes # For visual security check
# ProxyCommand nc -X connect -x 000.000.000.000:0000 %h %p # only if needed (replace
↪with your proxy)
```

If everything is configured correctly it will be able to connect to the virtual machine remotely and manage and control the machine using the terminal. To connect to the VM run in the terminal:

```
ssh my_vm01
```

From Windows:

There are many different tools for window e.g.: PuTTY(<https://www.putty.org>), Bitvise(<https://www.bitvise.com>), Solar-PuTTY, KiTTY, MobaXterm, mRemoteNG, Xshell 6 Client, PuttyTray and SuperPutty. Documentation can be found in there website.

SOFTWARE PREREQUISITES

11.1 Docker (Ubuntu 20.04)

Note these instructions are for Ubuntu 20.04 and may not work for other platforms. Installation instructions for other platforms can be found at docs.docker.com.

The open virtualization software Docker was used to deploy all the applications required for CbM development. The Docker is a set of platform as a service products that use OS-level virtualization to deliver software in packages called containers. To run the extraction routines it is recommended to install the latest version of docker with the below steps:

```
sudo snap remove docker
rm -R /var/lib/docker
sudo apt-get remove docker docker-engine docker.io
sudo apt-get update
sudo apt-get install -y ca-certificates curl gnupg lsb-release
curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo gpg --dearmor -o /usr/
↪share/keyrings/docker-archive-keyring.gpg
echo \
  "deb [arch=$(dpkg --print-architecture) signed-by=/usr/share/keyrings/docker-archive-
↪keyring.gpg] https://download.docker.com/linux/ubuntu \
  $(lsb_release -cs) stable" | sudo tee /etc/apt/sources.list.d/docker.list > /dev/null
sudo apt-get update
sudo apt-get install -y docker-ce docker-ce-cli containerd.io
```

To add the user to the docker group (the user will be able to run docker commands without sudo). Restart may be required.

```
sudo usermod -aG docker $USER
```

11.2 PostGIS

For this project we use PostgreSQL database with the PostGIS extension. Postgis extends the open source PostgreSQL database server with spatial data constructs (e.g. geometries) and spatial querying capacities, allowing storage and query of information about location and mapping.

To run a postgres database with postgis extension run:

```
docker run --name cbm_db --restart always -v pgdata:/var/lib/postgresql -p 5432:5432 -e
↪POSTGRES_PASSWORD=MYPASSWORD -d postgis/postgis
```

Change the POSTGRES_PASS=mydiaspassword to a secure password for the database.

This will return with a long docker container ID. Check if all is well:

```
docker ps -a
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS              PORTS               NAMES
85fc1f296000       postgis/postgis    "docker-entrypoint.s..." 9 seconds ago       Up 7 seconds        0.0.0.0:5432->5432/tcp   cbm_db
```

You need postgresql client tools to access the database. Make sure these match the database version (which is 10 in the example case). For example, for the command line interface on ubuntu:

```
sudo apt-get install postgresql-client-common postgresql-client-10
```

You can now connect to the database:

```
psql -h localhost -d postgres -U postgres

psql (10.12 (Ubuntu 10.12-0ubuntu0.18.04.1), server 10.7 (Debian 10.7-1.pgdg90+1))
Type "help" for help.

postgres=#
```

To be sure the required postgres extensions are enable run:

```
CREATE EXTENSION postgis;
CREATE EXTENSION postgis_raster;
```

The postgis image may contain the TIGER data base per default (this is a often used in postgis training). We don't need it, so remove with:

```
postgres=# DROP schema tiger, tiger_data cascade;
```

and exit and reconnect (you are now in schema *public*). List the default tables in that schema:

```
postgres=# \q
psql -h localhost -d postgres -U postgres
postgres=# \d

          List of relations
Schema |          Name          | Type  | Owner
-----+-----+-----+-----
public | geography_columns     | view  | postgres
public | geometry_columns      | view  | postgres
public | raster_columns        | view  | postgres
public | raster_overviews      | view  | postgres
public | spatial_ref_sys       | table | postgres
(5 rows)
```

These tables are required for the handling of spatial constructs (geometries, raster data, projection information).

11.2.1 Optimizing

The main configuration settings for PostgreSQL are in a text file `postgresql.conf` (`/etc/postgresql/"version"/main/postgresql.conf`). PostgreSQL ships with a basic configuration tuned for wide compatibility rather than performance.

It is strongly recommended to configure the settings of the PostgreSQL database based on your hardware configuration and application, suggested configurations can be found at [PGTune](#) or at [PGConfig](#).

11.2.2 Essential CbM tables

We now need to create the tables that we will use in the CbM context:

```
CREATE TABLE public.aois (name text);
SELECT addgeometrycolumn('aois', 'wkb_geometry', 4326, 'POLYGON', 2);

CREATE TABLE public.dias_catalogue (
    id serial NOT NULL,
    obstime timestamp without time zone NOT NULL,
    reference character varying(120) NOT NULL,
    sensor character(2) NOT NULL,
    card character(2) NOT NULL,
    status character varying(24) DEFAULT 'ingested'::character varying NOT NULL,
    footprint public.geometry(Polygon,4326)
);

ALTER TABLE ONLY public.dias_catalogue
    ADD CONSTRAINT dias_catalogue_pkey PRIMARY KEY (id);

CREATE INDEX dias_catalogue_footprint_idx ON public.dias_catalogue USING gist
    ↪(footprint);

CREATE UNIQUE INDEX dias_catalogue_reference_idx ON public.dias_catalogue USING btree
    ↪(reference);

CREATE TABLE public.aoi_s2_signatures (
    pid integer,
    obsid integer,
    band character(3),
    count real, mean real, std real,
    min real, max real,
    p25 real, p50 real, p75 real
);

CREATE INDEX aoi_s2_signatures_bidx ON public.aoi_s2_signatures USING btree (band);
CREATE INDEX aoi_s2_signatures_obsidx ON public.aoi_s2_signatures USING btree (obsid);
CREATE INDEX aoi_s2_signatures_pidx ON public.aoi_s2_signatures USING btree (pid);
```

The table *aois* is an ancillary table in which one can define the geometries of the areas of interest. The *dias_catalogue* is an essential table that stores the metadata for the relevant Sentinel-1 and -2 image frames. The table *aoi_s2_signatures* will store the time series extracts which will be linked to the parcel ID (pid) from the to-be-uploaded parcel reference table for each observation id (obsid) in the *dias_catalogue*.

Generate a new *aoi_s2_signatures* table for each aoi. This will typically be needed for separate years, as parcel references change. For instance, a table name like *nld2019_s2_signatures* would store all S2 records for the NL reference for 2019.

For Sentinel-1 time series create the equivalent tables with *bs* (backscattering coefficients) and *c6* (6-day coherence) instead of *s2* in the table name.

11.3 Jupyter server

The Jupyter Server is an open source web application that allows to create and share documents that contain live code, equations, visualizations and narrative text. Uses include: data cleaning and transformation, numerical simulation, statistical modeling, data visualization, machine learning, and much more (<https://jupyter.org>). JupyterLab is the next-generation user interface for Project Jupyter offering all the familiar building blocks of the classic Jupyter Notebook (notebook, terminal, text editor, file browser, rich outputs, etc.) in a flexible and powerful user interface. JupyterLab will eventually replace the classic Jupyter Notebook (<https://jupyterlab.readthedocs.io>).

Instaling DIAS Jupyter (Jupyter Notebook Tensorflow Python Stack for CbM)

GTCAP cbm_jupyter docker image is based on the tensorflow-notebook of Jupyter Notebook Scientific Python Stack and configured for Copernicus DIAS for CAP “checks by monitoring” with all the requirements. This is the recommended way to run a Jupyter server. Some DIAS providers may provide preinstalled Jupyter environments as well.

Run GTCAP Jupyter docker image

To run a jupyter server with the default setup:

```
docker run --name cbm_jupyter -p 8888:8888 gtcap/cbm_jupyter
```

This will run the jupyter server on port ‘8888’ and can be accessed from a web browser on ‘localhost:8888’.

To expose the jupyter server to port 80, change -p 8888:8888 to -p 80:8888, or to any other port.

More options

To pull the docker image from [dockerhub](https://hub.docker.com/r/gtcap/cbm_jupyter) use:

```
docker pull gtcap/cbm_jupyter
```

To configure and access the current local directory within the jupyter server run:

```
docker run -it -p 8888:8888 -v "$PWD":/home/jovyan --name=cbm_jupyter gtcap/cbm_jupyter
```

To run the Jupyter server with a predefined token, add at the end of the command:

```
start-notebook.sh --ServerApp.token='abcdefg h i j k l 1234567890'
```

Note: JupyterLab can be accessed by adding /lab at the url, instead of /tree (e.g. localhost/lab).

To run with enabled JupyterLab by default add -e JUPYTER_ENABLE_LAB=yes flag.

To run with enabled JupyterLab by default and mount the current directory run:

```
docker run -it -e JUPYTER_ENABLE_LAB=yes -p 8888:8888 -v "$PWD":/home/jovyan --name=cbm_
↪ jupyter gtcap/cbm_jupyter
```

For more options visit jupyter-docker-stacks.readthedocs.io

To access jupyter server, open in the web browser the link with the token that is provided in the terminal (e.g. <http://localhost/tree?token=abcdefg h i j k l 1234567890>).

Usage Instructions

All Jupyter Notebooks files have the extension ‘.ipynb’ and are identifiable by the notebook icon next to their name. To create a new notebook in JupyterLab, go to File -> New and select ‘New Notebook’. Notebooks currently running will have a green dot, while non-running ones will not. To run a cell with a script, click on the run icon or press Shift+Enter

More information can be found at: <https://jupyter.org/documentation>

The token to access the jupyter server will be in the command line output:

```
[I 08:51:48.705 NotebookApp] Use Control-C to stop this server and shut down all kernels.↵
↵(twice to skip confirmation).
[C 08:51:48.708 NotebookApp]
```

```
To access the notebook, open this file in a browser:
    file:///home/jovyan/.local/share/jupyter/runtime/nbserver-8-open.html
Or copy and paste one of these URLs:
    http://abcd12345678:8888/?token=abcd12345678
    or http://127.0.0.1:8888/?token=abcd12345678
```

You will be able to access the Jupyter server on port 8888 (or any other port) on VM’s public ip e.g.: **0.0.0.0:8888**
Copy the token from the command line and add it to the web interface.

11.3.1 Build Jupyter image from source

To build cbm_jupyter docker image from source see the [Jupyter for cbm README.md](#) file.

DATABASE FOR JRC CBM

12.1 DB in the CbM architecture

Sentinel images offer the possibility to monitor in real time and on a continuous basis the conditions of the agricultural parcels and their coherence with the declarations made by the farmers within the Common Agricultural Policy (CAP). At the same time, this huge amount of data poses technical challenges for the extraction and handling of the information (signatures) that is relevant for the Checks by Monitoring (CbM). In this respect the **database** (DB) plays a central role in the demonstrative [JRC CbM architecture](#), as it is the tool used to store and manage the data involved in the process (except satellite images). Satellite data are made available in the Object Storage of the [Copernicus Data and Information Access Services \(DIAS\)](#) infrastructure and processed in that environment by Python-based modules. The base layers (particularly, parcels and image metadata, the latter generated by the DIAS) are stored in a database installed in the Member State (MS) DIAS space, inside the same environment of the satellite image archive. The database also stores the signatures (i.e. number of pixels, mean, std, min, max, 25th percentile, 50th percentile, 75th percentile) that result from the intersection of satellite image bands and the parcels. This information can then be used by analysts to verify the consistency of the farmer declarations against the conditions detected by the satellites. In the JRC CbM system, the open-source Relational DataBase Management System (RDBMS) [PostgreSQL](#) with its spatial extension [PostGIS](#) are used as database software. S(patial)RDBMS can efficiently manage very large spatial and non spatial datasets with complex structure in a multi-user and secure environment. In this documentation page, we introduce SRDBMS, we describe the database that has been set up for the JRC CbM, particularly for the Outreach project, we describe how to access, retrieve and export data stored in the DB. In the last section, we illustrate some procedures to optimize performance. In an operational national Paying Agency (PA) CbM system, a relational database can be used not only to support signatures extraction and management data but is also a good candidate as central repository for all the information relevant for the CAP process. In such a context, the technical solution depends on the specific goals and constraints of each PA. Learning database administration and advanced use (CbM backend) require a technical background and dedicated training, while for basic interaction (data retrieval and visualization, CbM frontend) limited expertise is needed. The scope of this documentation is to show how the database is used in the JRC CbM system and demonstrates its potentialities to support CbM.

12.2 Spatial database in a nutshell

In computer science, a database (or DB) refers to **a set of data organized in such a way as to facilitate its management, use and updating, stored in a computer**. The relational model is a logical model for structuring data in a database. All data are represented as relationships (tables) that are linked together. The data is manipulated with the operators of relational algebra using the **SQL** (Structured Query Language) language. Spatial database are database that can manage the spatial attribute (raster or vector) of an object. The main features of (spatial) relational databases are:

1. Storage capacity
2. Retrieval performance
3. Server/client structure (modular approach)

4. Remote access
5. Concurrency control
6. Permission policy
7. Data formalization
8. Relational environment (data modelling)
9. Data integrity controls
10. Data consistency (normalization)
11. Data preservation
12. Easy automation of processes
13. Integration with other data repository
14. Industrial standard
15. Cost effective
16. Backup/recovery
17. Efficient management of spatio-temporal data
18. Mature technology

In a nutshell, we can define relational database as a tool to securely store and preserve large volume of standardized and consistent data (including spatio-temporal data) that can be efficiently retrieved by multiple remote users with different interfaces with no data duplication.

12.2.1 Main database elements

The basic element of a database is called a **TABLE**. It is composed of columns and rows, but unlike what happens in spreadsheet, a table is declaratively created with a structure: each column has a defined **DATA TYPE**, and the rows (also called *records*) must respect this type: the system enforces this constraint, and does not allow the wrong kind of data to slip in. Some of the frequently used data types are: **integer** for whole numbers, **numeric** for decimal numbers, **text** for character strings, **date** for dates, **boolean** for yes/no values, **timestamp** for attributes containing date and time. Each data type has specific properties and functions associated. For instance, a column declared to be of a numerical type will not accept arbitrary text strings, and the data stored in such a column can be used for mathematical computations. By contrast, a column declared to be of a character string type will accept almost any kind of data but it does not lend itself to mathematical calculations, although other operations such as string concatenation are available. The number and order of the columns is fixed, and each column has a name. The number of rows is variable — it reflects how much data is stored at a given moment. Each row of a table must be identified by the value of one or more columns. The same value (or set of values) cannot be repeated in two different rows. The attributes that identify a record are called **PRIMARY KEY**. The primary key must be explicitly defined for all the tables (although this is not strictly required to create the table, it is necessary for a correct use of the database). Tables can be linked to one another (the jargon term for this kind of link is *relation*, which accounts for the *R* in *RDBMS*): you can explicitly ask that the value to put in a specific record column comes from another table by using **FOREIGN KEYS**. This helps reduce data replication (redundancy), and explicitly keeps track of inter-table structure in a formalized way. A database contains one or more **SCHEMAS**, which in turn contain tables. Schemas also contain other kinds of objects, including data types, functions, and operators. The same object name can be used in different schemas without conflict. Schemas are analogous to directories at the operating system level, except that schemas cannot be nested. Schema are used:

- to allow many users to use one database without interfering with each other
- to organize database objects into logical groups to make them more manageable
- third-party applications can be put into separate schemas so they do not collide with the names of other objects

It is worth mentioning another very useful feature offered by database: **VIEWS**. Views are queries (i.e. SQL statements, see SQL subsection) permanently stored in the database. For users (and client applications), views work like normal tables, but their data are calculated at query time and not physically stored. Changing the data in a table alters the data shown in subsequent invocations of related views. Views are useful because they can represent a subset of the data contained in a table; can join and simplify multiple tables into a single virtual table; take very little space to store, as the database contains only the definition of a view (i.e. the SQL query), not a copy of all the data it presents; and provide extra security, limiting the degree of exposure of tables to the outer world. On the other hand, a view might take some time to return its data content. For complex computations that are often used, it is more convenient to store the information in a permanent table.

12.2.2 The spatial bit

Until a few years ago, spatial information was managed and analysed exclusively with dedicated software (GIS) in file-based data formats (e.g. shapefiles). In spatial databases, the spatial component of an object (e.g. the boundaries of a parcel) is managed as one of its many attributes and stored in a column. From a data management point of view, spatial information is no different from a date or a quantitative measure (number). The spatial extension adds spatial data types (vectors such as points, lines, polygons, curves, in 2 or 3 dimensions, and raster) to the standard data types that store other associated (non-spatial) attributes of objects. It also introduces spatial indexes to improve performance involving these attributes. Spatial data types can be manipulated with the SQL language through a comprehensive set of functions to analyse geographical components (e.g. calculate the area, reproject into a different system), determine spatial relationships (e.g. the intersection of two spatial objects) and modify geometries (e.g. create the line of a transect from individual plots). The list of functions available in a spatial database is very extensive. A set of very common functions is defined by the **OGC SFSQL**. This essentially allows GIS functionality to be built using the capabilities of a relational database integrating spatial and non spatial data in the same environment. A spatial database does not replace GIS software, especially for visualisation, map creation and some advanced functions, but facilitates the integration and management of spatial data with other available information. Moreover, spatial data are generally based on widely used shared standards, which makes the exchange of data between different platforms simple and straightforward and allows a seamless integration between spatial databases and GIS software, which can be used as database clients. To visualise spatial data, like all other data in a database, a client application is needed that requests the data from the server and displays it to the user in the required format. For spatial data, the best client is QGIS. PgAdmin provides a very fast way of displaying spatial objects in a table, although it does not offer the functionality of GIS software.

12.2.3 PostgreSQL and PostGIS

PostgreSQL represents the state of the art as regards relational databases, and in particular spatial databases thanks to its extension **PostGIS**. First of all, this is **open source** software. This has the following advantages:

- Use of standards
- Interoperability with other tools
- No vendor-lock policies
- No limitations in its use
- No costs for licenses

PostgreSQL and PostGIS have been chosen as database software for JRC CbM because they are characterized by:

- Great spatial tools for data management and analysis
- Geography data type, raster, topology, 3D, ...
- Great non-spatial tools for data management and analysis
- Good documentation
- Fast development

- Native support by many software
- Supporting, collaborative and active (large) community
- Strong commercial support
- Multi-platform support
- Many procedural languages
- Stable and secure
- Mature project with long history
- Used by many large companies

Another important PostgreSQL feature is the possibility to scale it beyond running on a single server, exploiting cloud based infrastructures. There are many possible approaches, for example replication, database clustering and connection pooling. Based on your specific requirements you can identify the solution that best fit them. Many companies provide commercial support for advanced PostgreSQL high performance, multi-server solutions. As relational database are based on industrial standard, it is usually easy to move data from a specific software to another. Many other relational database software exist. PostgreSQL is the recommended choice, but if a database already exists in the organization that wants to implement a CbM system, it is probably better to keep the same platform. Other popular database with a spatial extension are for example MySQL, SQLServer and Oracle. For local, simple and single-user database SQLite with its spatial extension SpatiaLite is also a possibility (this is usually non the case for CbM). Relational database is not the only option to store and manage data in a “modern” database. NoSQL database (as opposed to relational databases based on SQL) offers different features compared to relational database. It is more suited for contexts where the evolving data structure requires a flexible data model. Given the well structured and defined database schema of CbM, and because data integrity and consistency is better granted by SQL database, we opted for a solution based on relation database. However, given its ability to scale up easily, the NoSQL database can be considered as an alternative in specific cases.

12.2.4 SQL

SQL (Structured Query Language) is the universally used data definition and manipulation language in relational databases, i.e. the way for user to interact with SRDBMS. It does not require sequences of operations to be written, but to specify the logical properties of the information sought (declarative language). SQL statements are used to perform tasks such as retrieve data from a database (in this case they are commonly called “queries”) or update and create database objects. SQL is highly standardized and while each database platform has some kind of SQL dialect, it can be used with any SRDBMS software with minimal changes. While complex queries can be hard to design, SQL itself is a simple language that combines a very limited sets of commands in a way that is similar to the natural language. You can run SQL commands, explore database objects and see tables using any client that is able to connect with your DB (i.e. most of the software that deals with data). The reference graphical interface for PostgreSQL database management and query is [PgAdmin](#). **Psql** is the interactive terminal for working with Postgres (command line). The operation of choosing the records from a table or a combination of tables is called *selection*: the **SELECT** command allows you to express clearly which columns you need, which rows, and in which order; you can also generate computed data on the fly. The basic structure of a query is the following:

```
SELECT
  <one or more columns, or * for all columns of a table>
FROM
  <one or more tables>
WHERE
  <conditions to filter retrieved records on and to combine multiple tables>;
```

Some practical examples are provided in the section on database access. Many tutorials on SQL are available on the web. Among the others:

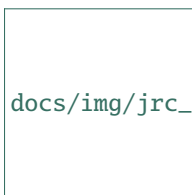
- [PostgreSQL official tutorial](#)
- [postgresqtutorial](#)
- [w3resource](#)
- [sqlbolt](#)
- [webcheatsheet](#)
- [www.sql.org](#)

12.3 JRC CbM DB structure

The primary goal of the JRC CbM database is to store and make available the time series of Sentinel bands signature for each parcel. This is calculated by a Python module by rasterizing parcel polygons stored in the database and intersecting them with satellite images stored in the DIAS [Object Storage](#). Images are selected and retrieved based on the metadata provided by the DIAS and stored in the database (particularly the area covered and the acquisition date and time). Basic statistics are calculated from the set of pixels belonging to the each parcel, which are then stored back in the database. The number of pixels with cloudy flags for Sentinel 2 is extracted for each parcel in order to identify images affected by atmospheric disturbance (and thus with not reliable statistics). This information is also stored back in the database. To sum up, the main database objects (tables) are:

- Parcels (e.g. `parcels_2020`)
- Images metadata (`dias_catalogue`)
- Signatures (e.g. `sigs_2020`)
- Cloud flags histogram (`hists_2020`)

Parcels, signatures and histograms are year-specific because parcels change every year. The year used as suffix in the names is thus related to the year of the parcel definition, not to the year of the Sentinel data. For example, for the parcels declared in 2020 it is possible to extract the complete set of Sentinel 2 from 2016). The *dias_catalogue* table is not “timestamped” because it contains information about satellite images for all years. The evolution of parcels can be potentially tracked with a object-oriented versioned approach where only changing elements are recorded keeping track of all the history giving the possibility of queering both current and history record. This solution has not been implemented in the JRC CbM system. In the Outreach database, a single *dias_catalogue* table for all countries is generated and another table (*aois*) stores the information about pilot areas, particularly their extension. In the JRC CbM database, tables are organized in schemas. The general tables (e.g. *dias_catalogue*) are stored in the *public* schema. The other working tables are stored in a specific schema. In the case of the Outreach project, the country specific tables (parcels, signatures, histograms) are stored in a dedicated schema identified by the 2-letters [ISO code](#) of each country (for example *hr* in case of Hungary). When multiple Paying Agencies of the same Member State are joining, two additional letters are used to identify the PA. In this way it is easy to move data if the database has to be transferred to another local or cloud-based system. The corresponding database data model (i.e. the conceptual representation of the real world in the structures of a relational database: tables and their relationships) in form of a simplified Entity-Relationship (ER) diagram is illustrated in Figure 1. It describes what types of data are stored and how they are organised. Columns used as primary key are indicated in bold and external keys are represented as connecting lines between tables. For each column is also indicated the data type.



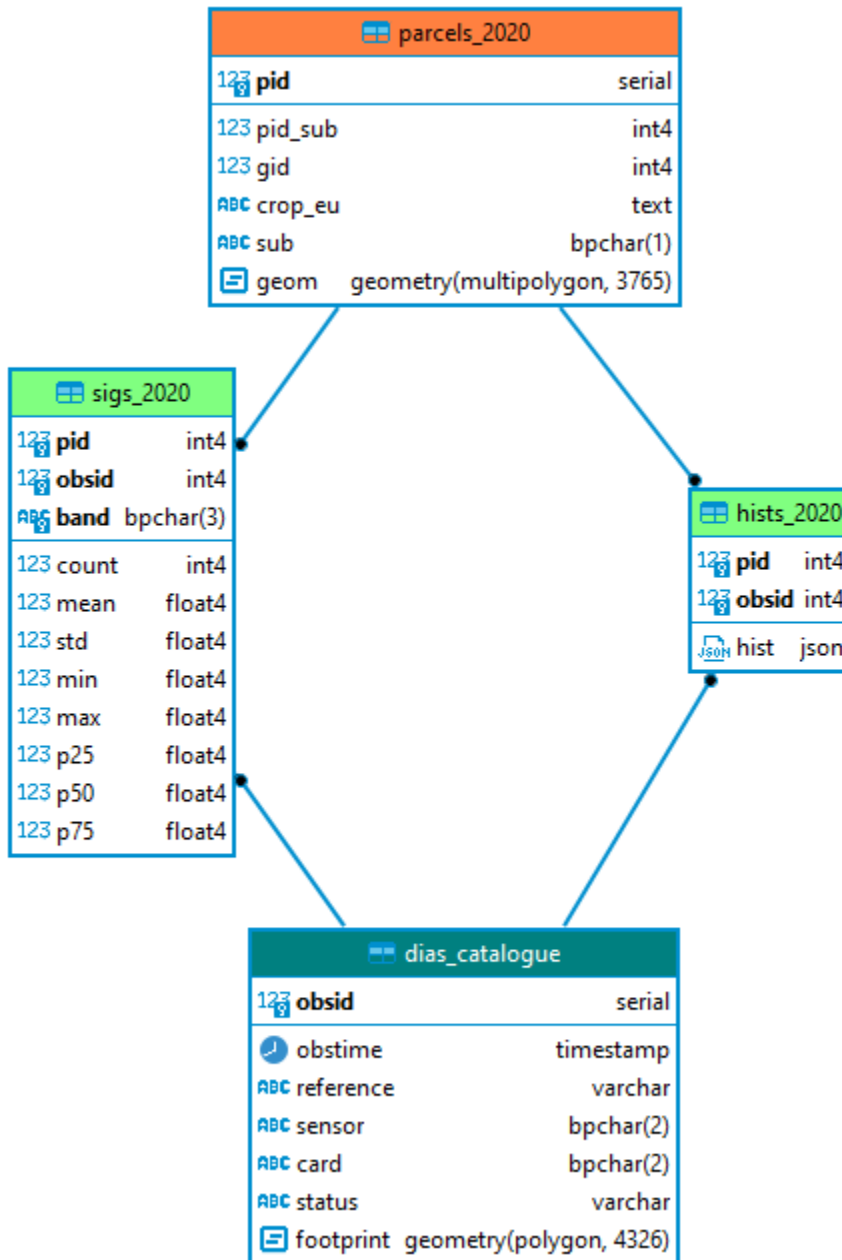


Figure 1. Outreach database data model (core tables)

In some cases, the parcels table can be split in multiple tables according to the specific declaration (e.g. grazing: parcels_2020_g, mowing: parcels_2020_m, bare soil: parcels_2020_b) or to the pilot area (e.g. south: parcels_2020_s, north: parcels_2020_n). These differences can also be accommodated using specific columns in the parcels table (e.g. the *sub* column in the ER diagram) or using partitioned tables (see section on optimization of performances). If the parcels table is divided in physical tables, an approach must be used to generate unique ID in the different tables that are referred to in the signatures and histograms or specific hist and sigs tables must be created (e.g. sigs_2020_g, hist_2020_g for grazing). What is reported in the diagram and used in this demonstrative JRC CbM database is only part of the information involved in the CbM processing. The database can be easily extended to include for example additional data like present and past declarations, parcel classification according to traffic light classes and based on

markers, other relevant environmental layers. The information from different tables can be combined in a SQL statement to generate the complex information required for a specific task in form of a single table as a view. Practical examples are provided in the next section.

12.4 Database access

12.4.1 Server/client structure

The database architecture is based on a client-server structure. It is divided into two distinct components: a “server” program that provides a service and a “client” program that accesses the service. The database server (PostgreSQL) is the back-end system of the database application and provides database functionality to client applications. The client is usually an interface through which a user makes a request to the server through SQL commands. The client then converts the server’s response into the form requested by the user. Some examples of possible clients are PgAdmin, QGIS, Python through Psycopg2, R, Calc, Excel, PHP web interfaces, MS Access, ArcGIS. In this architecture the data management and storage layer is physically separated from data use. Data are displayed by the clients but are stored in the DB and not duplicated. A database server is installed on a physical computer (server), or in services in the cloud. Several database servers can be installed on the same machine. Several databases can be created within the same database server. In the JRC CbM, the database is installed on the DIAS (although installation on a local machine is also an option). For more information on installation and initialization (e.g. creation of the image database table and enabling of the PostGIS spatial extension) of the database check the [specific documentation](#).

12.4.2 Access parameters

In order to remotely access a database from any client, five parameters are usually necessary:

- Server IP address
- Port
- (Database name)
- User name
- User password

In the Outreach project, if requested, these information are provided directly to users limiting access to the specific MS data sets, but in general DB access to the Outreach database is granted through an intermediate layer. In an internal MS/PA CbM system, we recommend to limit direct DB access to specific cases (i.e. database administrators, users with advanced skills in SQL and good knowledge of the database structure). In all the other cases an intermediate layer can be used, as in the case of the Outreach DB. This ensures performance and security by preventing poorly designed resource-intensive queries and facilitates access to basic users with no knowledge of SQL who can be guided by predefined queries offered as a simple graphical interface where only defined parameters need to be defined. This can be implemented using a RestFUL API, as in the case of the Outreach database.

12.4.3 User access policy

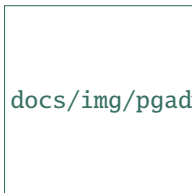
One of the most important functions of a database is the possibility of restricting access and possible operations (e.g. read, edit, insert, delete) on the data according to the different types of users, and in a differentiated manner for the different tables/rows of the database. PostgreSQL manages access permissions to the database through **ROLES**. A role is an entity that can own objects and have privileges on the database. It can be an individual user or a group of users. Users can be grouped to facilitate privilege management: privileges can be granted to, or revoked from, a group as a whole simplifying a lot the management of the permission policy. This is done by creating a role that represents the group (for example *administrators* that can create/delete objects in the database, *editors* that can add, delete and update data in existing tables, and *viewer* that can only view all or part of the data but not change it), and then granting membership of the group role to individual user roles (administrators, analysts, final users). Database roles apply to all databases in a cluster, but permissions are then associated with the individual objects in each database. Each user is assigned a password together with the role to ensure data security. Access to the server itself can be restricted to certain IP addresses to further reinforce security. In PostgreSQL, permissions can go down to the single record level. In the Outreach project, access is managed by the intermediate layer connected to a read-only role defined in the database (*api_bot*).

12.5 Data retrieval

To interact with a PostgreSQL database, it is not necessary to install PostgreSQL, but it is sufficient to install a client that connects to the database (server). The database with the data is instead physically installed on the server (e.g. in the DIAS). In this Section we show examples of connection of typical clients for spatial and non spatial data. Only a limited set of information is provided here. Practical demonstrations are done during the MS trainings on JRC CbM. In addition, many tutorials are available on the Internet for further study.

12.5.1 PgAdmin

PgAdmin is the most common graphical interface for querying data and managing PostgreSQL. You can view tables, query and download data, create new tables and views, edit and insert data, manage users, make backups. PgAdmin is an open source software and has very frequent updates. If you install PostgreSQL, PgAdmin is installed automatically. PgAdmin is not the only client that you can use to manipulate data and objects in the database. An alternative to PgAdmin for database management is for example **dBeaver** (Community version), which is also very useful for generating ER diagrams. Note that when you open PgAdmin for the first time, it asks you to create a password. This is not the password for accessing the databases, but only the password for accessing PgAdmin (since PgAdmin stores all database access passwords internally). The PgAdmin interface is organised into 5 main sections (see Figure 2). The display of the elements can be optimised through the customisation options.



`docs/img/pgadmin_panels.png`



`img/pgadmin_panels.png`

Figure 2. PgAdmin interface (numbers correspond to the description given below)

The 5 sections (or panels) are:

1. The menu bar
2. The toolbar
3. The tab bar
4. The tree menu with the database objects
5. Contents of the selected tab (object properties, tables, SQL editor)

Each part is used to perform different types of management tasks. The most common example of an operation is selecting a database object in the tree menu of panel 4 (e.g. a table) and displaying it in panel 5 (tab contents). The *Help* buttons in the bottom left-hand corner of each dialogue box open the online help for that box. Additional information can be accessed by navigating through the Help menu.

To create a connection to a database, select *Object/Create/Server* from the menu (or right-click on the Server icon in the menu and select *Create/Server*). A window opens in which the connection parameters must be entered. On the *General* tab you only need to enter a name (any name) for the connection. In the tab *Connection* you have to enter the IP address of the server, the port, the user and the password (see Figure 3).

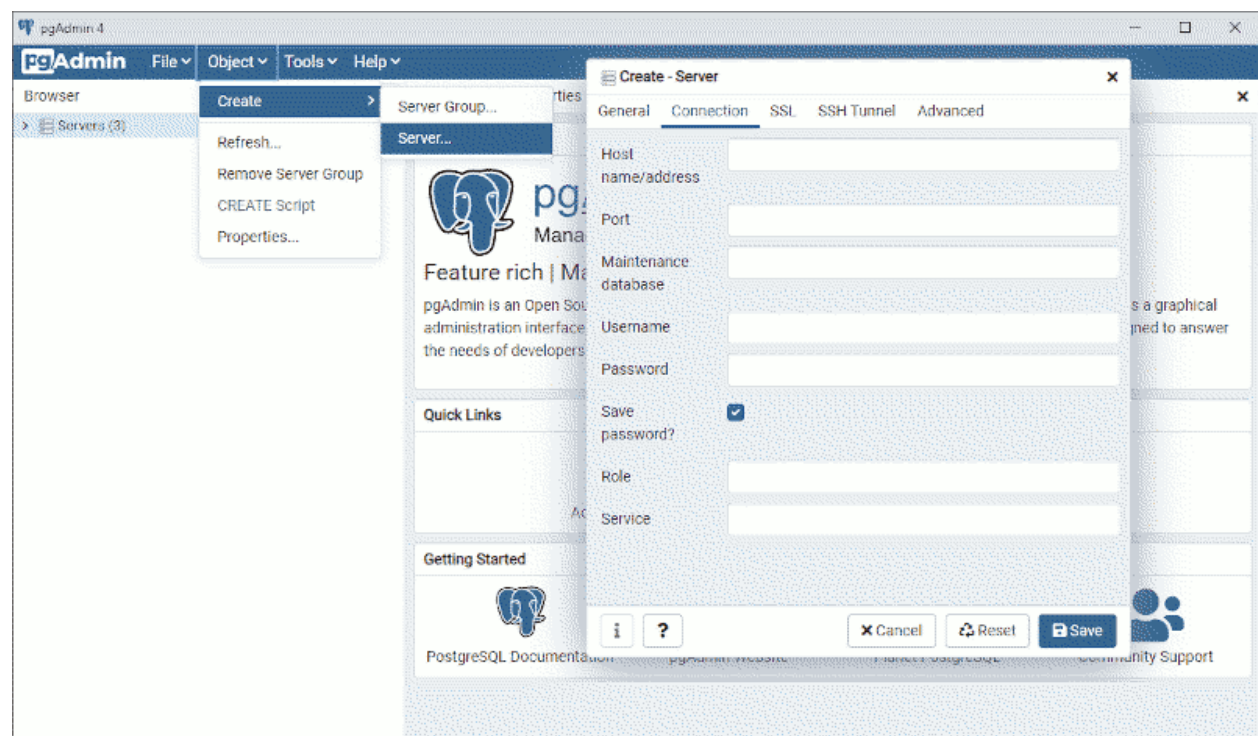


Figure 3. Creating a connection to a database server

Once saved, the connection will appear in the menu tree by expanding the *Server* icon.

12.5.2 phpPgAdmin

There is an alternative to PgAdmin that does not require the installation of any local client because it uses a tool installed on the server: [PhpPgAdmin](#). The interface is similar to PgAdmin, but it has less functionalities and can have problems with unstable connections, and it has limited development support.

12.5.3 Psql

[Psql](#) is the interactive terminal for working with PostgreSQL. It enables to type in queries interactively, issue them to PostgreSQL, and see the query results. Alternatively, input can be from a file or from command line arguments. In addition, psql provides a number of meta-commands and various shell-like features to facilitate writing scripts and automating a wide variety of tasks. For example `\d` list tables in the database and `\d public.dias_catalogue` list all the columns of the specified table.

12.5.4 QGIS

PostgreSQL/PostGIS itself offers no tool for spatial data visualization, but this can be done by a number of client applications, in particular GIS desktop software like QGIS (a powerful and complete open source software). It offers all the functions needed to deal with spatial data and offers many tools specifically for managing and visualizing PostGIS data. Connecting to the database is pretty simple and the process is well documented. Data can be accessed in three steps: 1) create a connection to the db (the first time that you connect to the db, see Figure 4) using the database access parameters, 2) open the connection, 3) get the data. Once the connection is created, you can use the dedicated DB Manager interface to explore, preview, visualize in the main canvas and also export spatial data (both vector and raster).



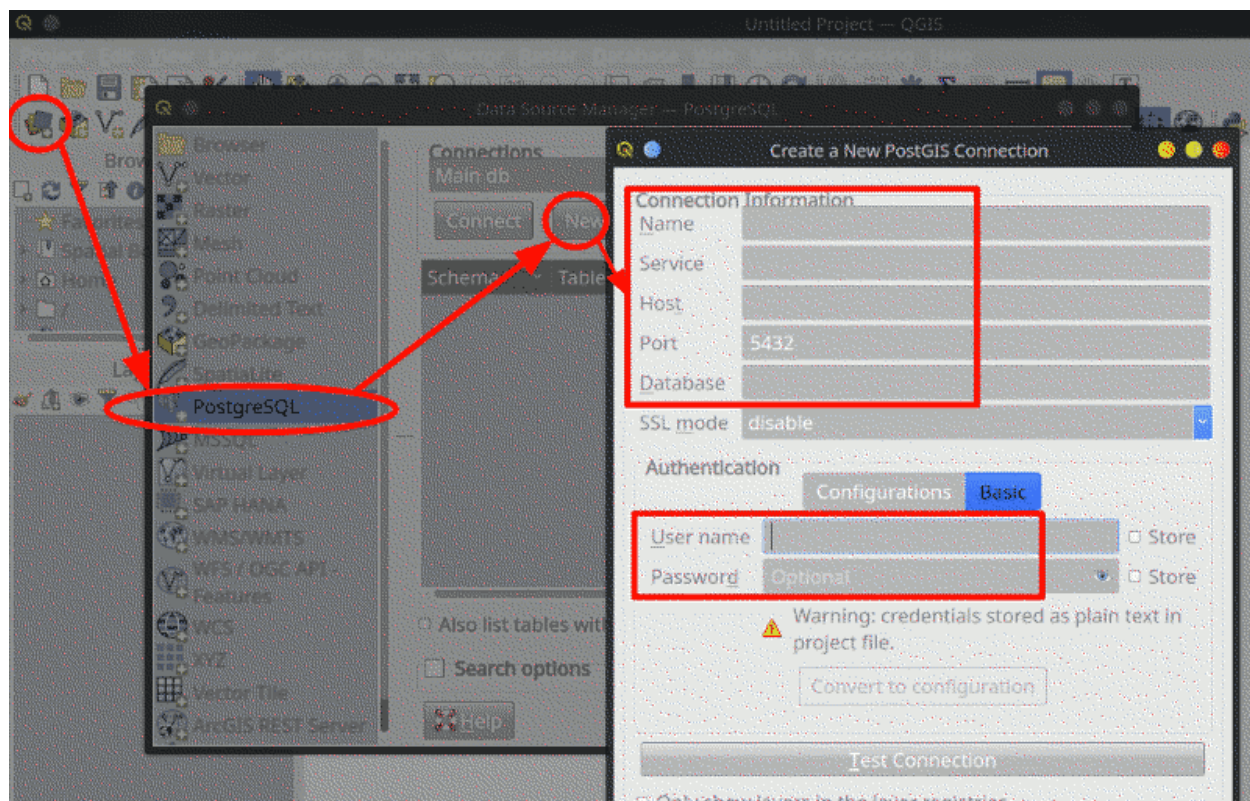


Figure 4. Connecting with the database from QGIS

12.5.5 Jupiter Notebook

Python is the suggested tool to access the JRC CbM and the examples provided in the system documentation are always based on it. Here a simple example of connection with the database using the Python psycopg2 is reported:

```
import psycopg2
import pandas as pd
from datetime import datetime
from ipywidgets import widgets

DB_HOST = 'IP_ADDRESS'
DB_NAME = 'DB_NAME'
DB_USER = 'YOUR_USER'
DB_PORT = 'PORT'
DB_PASS = 'YOUR_PASSWORD'

conn_str = f"host={DB_HOST} dbname={DB_NAME} user={DB_USER} port={DB_PORT} password={DB_
PASS}"
conn = psycopg2.connect(conn_str)

getSampleList = f"""
    SELECT * FROM dias_catalogue LIMIT 2;
    """
```

(continues on next page)

(continued from previous page)

```
cur = conn.cursor()
cur.execute(getSampleList)

for rows in cur:
    print(rows)
```

12.5.6 RESTful API

Several examples of interaction with the database using a RESTful api are provided in the [dedicated JRC CbM documentation](#).

12.5.7 R

To import data from the database into [R](#), simply use the code below (via the RPostgreSQL library):

```
library(RPostgreSQL)
drv <- dbDriver("PostgreSQL")
con <- dbConnect(drv, dbname="YOURDB",
host="IP_ADDRESS",
port="PORT",
user="YOUR_USER",
password="YOUR_PASSWORD")
rs <- dbSendQuery(con, "SELECT * FROM public.dias_catalogue;")
df <- fetch(rs,-1)
df[1:4,]
str(df)
dbClearResult(rs)
```

In the *dbSendQuery* command, you can insert any SQL code that will be executed by the database and then inserted into a dataframe as specified by the user (in the code above, *df*). In the example, the Sentinel metadata is loaded into the *df* dataframe. It is also possible to import data into the database from R. You can find all the documentation and a list of the most interesting packages for working with a PostgreSQL database from R on the Internet.

12.6 Export and import data

There are different ways to export a table or the results of a query to an external file. The easiest one is to use the pgAdmin interface: in the SQL console select **Query/Execute to file**, the results will be saved to a local file instead of being visualized. Other database interfaces have similar tools. This can be applied to any query. For small tables, you can select the data visualized and do copy/past. For spatial data, the easiest option is to load the data in QGIS and then save as shapefile (or any other format) on your computer. Shapefiles can be imported into the database with a simple drag and drop operation in the QGIS interface. You can also import/export spatial data using [ogr2ogr](#) command from the GDAL library.

Form command line, you can use [COPY \(TO\)](#). **COPY TO** (similarly to what happens with the command **COPY FROM** used to import data) with a file name directly write the content of a table or the result of a query to a file, for example in .csv format. The file must be accessible by the PostgreSQL user (i.e. you have to check the permission on target folder by the user ID the PostgreSQL server runs as) and the name (path) must be specified from the viewpoint of the server. This means that files can be read or write only in folders ‘visible’ to the database servers. If you want to

remotely connect to the database and save data into your local machine, you should use the command `COPY` instead. It performs a frontend (client) copy. `\COPY` is not an SQL command and must be run from a PostgreSQL interactive terminal `PSQL`. This is an operation that runs an SQL `COPY` command, but instead of the server reading or writing the specified file, PSQL reads or writes the file and routes the data between the server and the local file system. This means that file accessibility and privileges are those of the local user, not the server, and no SQL superuser privileges are required.

If you want to export a very large table or an entire database, you can use the `pg_dump` and `pg_restore` commands. These commands are used to create backup of part of all database content. The backup and restore can also be run using PgAdmin GUI. From pgAdmin, the operation of making a database dump is extremely simple: right click the database and choose Backup. There are a few output formats, apart from the default Custom one. With Plain the file will be plain (readable) SQL commands that can be opened (and edit, if needed) with a text editor. Tar will generate a compressed file that is convenient if you have frequent backups and you want to maintain an archive.

More information about the backup of the JRC CbM database is available in the [db backup documentation](#).

12.7 Performance optimization

12.7.1 Basic optimization

Given the size of the tables generated by CbM, the time required to upload and retrieve the data can be long. The table that is typically most demanding in this respect and most used for analysis is the one with the signature (image statistics) per parcel and Sentinel scene/band. The number of records in this table is:

number of parcels * number of images * number of bands

To give an order of magnitude, if the parcels are 500,000, the Sentinel 2 images are 73 (an image every 5 days in a single year) and the bands are 7, the number of signatures records generated is about **260 millions**. This corresponds to a size (without indexes and primary keys) of about **19 GB**. As mentioned in the introduction to relational database, primary key is a compulsory element in a relational table. Technically, it is possible to create a table without a primary key, but this is a bad practice. Primary keys, where these involves columns commonly used to retrieve data, is the first optimization of time of data extraction (an index is created together with the primary key, and in case of multiple columns primary key the order of the columns matters). In the case of the `sigs_YYYY` tables, the primary key is based on the id of the parcel (`pid`), the id of the image (`obsid`) and the id of the band (`band`), that are good candidate as criteria to extract data. The combination of these three elements is unique for each row. In the case of our example, the primary key adds about 10 GB to the table size, for a total of about **30 GB**. Indexes are a common way to enhance database performance. An index allows the database server to find and retrieve specific rows much faster than it could do without an index, similar to what happens with a book summary. Indexes also add overhead to the database system as a whole, so they should be used sensibly. If a specific index is added for each of the three columns of the primary that are likely to be the most queries, the size increase of about 1.5 GB per index, for a total size of about **35 GB**. Additional index can be added if other fields are used often in queries (a.g. a mean value used to detect markers). Indexes slow down upload of new data into the table, so in case of bulk insert this can increase the loading time. If this an issue, you can consider to drop the indexes and recreate them once the upload is finished). To give an idea, for standard queries with/without keys and indexes (retrieve all the signatures for a parcel and a specific band) like:

```
SELECT pid, obsid, band, count, mean, std, min, max, p25, p50, p75
FROM sigs_2020
where pid = "mypid" and band = "myband";
```

the time needed to get the data is:

- Table with primary key and indexes: 0.2 second
- Table with primary key but no indexes: 2 second
- Table with no primary key and no indexes: 2 minutes

while for a query based on the id of the image (all parcels for a specific image/band) performances are:

- Table with primary key and indexes: 2 second
- Table with primary key but no indexes: 2 minutes
- Table no primary key and no indexes: 30 minutes

These numbers depend on many factors on top of table size, but they give an idea of the importance of basic optimization.

12.7.2 Advanced optimization

If the performance achieved with indexes are not satisfactory, other actions are possible. This can be the case when many more parcels and images are managed. In this case, additional actions that can be taken. Among the others:

- **Tune configuration parameters:** PostgreSQL default configuration is tuned for wide compatibility rather than performance. Parameters can be changed in the `postgresql.conf` file to optimize the configuration for each specific case. This file is located in the `data` subfolder inside the folder where you installed PostgreSQL. Many tools exist to help this operation (for example: **PGtunes** and **PGconfig**).
- **Partitioned tables:** splitting what is logically one large table into smaller physical pieces can improve query performance dramatically in certain situations, particularly when most of the heavily accessed rows of the table are in a single partition or a small number of partitions. Partitioning effectively substitutes for the upper tree levels of indexes, making it more likely that the heavily-used parts of the indexes fit in memory. These benefits will normally be worthwhile only when a table would otherwise be very large. The exact point at which a table will benefit from partitioning depends on the application, although a rule of thumb is that the size of the table should exceed the physical memory of the database server.
- **Table Clustering:** when a table is clustered, it is physically reordered based on the index information. This speeds up queries that are based on the criteria used for clustering. Clustering is a one-time operation: when the table is subsequently updated, the changes are not clustered. That is, no attempt is made to store new or updated rows according to their index order and tables must be periodically reclustered.
- **Increase hardware resources:** important improvements can be achieved if more hardware resources are available (e.g. CPU, RAM). If a database is very big in size, pure optimization might not be enough if the hardware is weak. In this case adequate resources are usually needed.
- **Multiple-Server Parallel Query Execution:** even if this features in not available natively in PostgreSQL, some companies provide commercial support for this kind of optimization where a query is execute in parallel on multiple server.

DATA PREPARATION

Data preparation

13.1 Adding shapefiles (.shp) to PostGIS database

More information at: <http://postgis.net/documentation/>

To add a shapefile to a postgis database using ogr2ogr, do the following:

```
sudo apt-get install gdal-bin
```

to install the GDAL binaries, which is highly recommended also for other geospatial data analysis tools. You can also use ogr2ogr to import from other formats (e.g. GML, GeoJSON, etc.).

```
ogr2ogr -f "PostgreSQL" PG:"host=localhost port=5432 user=postgres dbname=postgres" \  
-nln "roi_YYYY" -nlt PROMOTE_TO_MULTI parcels_2018.shp
```

This will import the shape file into the table *roi_YYYY* (choose a recognisable roi name and replace YYYY with the year of interest). The option *-nlt PROMOTE_TO_MULTI* may not be needed if all shape features are single polygons. If (some) features fail to upload (e.g. due to corrupt geometries), add the option *-skipfailures*, but check the completeness of the uploaded table afterwards.

See

```
ogr2ogr --help
```

for more information.

The table *roi_YYYY* will be spatially indexed after upload, which significantly speeds up spatial querying.

You are now ready to [start transferring metadata from the catalogue](#) and then [run extracts](#).

13.2 Transfer metadata from the DIAS catalog

Each DIAS maintains a catalog of the Level 1 and Level 2 CARD data sets available in the S3 store. Even with the use of standard catalog APIs (e.g. OpenSearch), the actual metadata served by the catalog may contain different, or differently named, attributes. Thus, in order to minimize portability issues, we implement a metadata transfer step, which makes metadata available in a single, consistent, database table **dias_catalogue** across the various DIAS instances.

As is the case for the later parcel extraction stage, metadata transfers can be done in (1) burst mode, e.g. for a long time series of processed CARD data for a new area on interest or (2) in update mode, for newly generated CARD data set, e.g. on a daily basis. Since database transfers are not particularly demanding on compute resources, there is no need

for parallelization and both modes can run on a single VM. “Burst mode” can be run interactively from the command line interface (terminal) or as a notebook, while “update mode” can run as a scheduled python task in cron.

13.2.1 OpenSearch compliant catalogs (CREODIAS, MUNDI)

Both CREODIAS and MUNDI provide online catalogs that implement the [OpenSearch](#) standard. This is the same standard as implemented by the [Copernicus Open Access Hub](#). In short, this standard ensures that the catalog can be queried via so-called REST requests, typically with standard request parameters to define the area of interest and search period, product type and some other parameters (e.g. platform, sensor, etc.). The response is usually in a well-known parseable format, such as JSON or XML. It is relatively straightforward to build graphical user interfaces (GUI) to capture the request parameters interactively and display the response results (e.g. [CREODIAS FINDER](#), [MUNDI Web Services](#)). More importantly, though, the OpenSearch standard allows to capture search results in scripts as well, so that catalog updates can be automatically processed.

The transfer of metadata consists of the following steps:

- Build up the OpenSearch query for the area of interest, the search start and end date and the product type;
- Fire off the query to the catalog server
- Check whether the response type is as expected (XML in our case) and if so
- Parse the relevant attributes from the XML response and reformat to **dias_catalogue** compatible table fields (e.g. the gml geometry for the CARD image footprint needs to become a WKT polygon)
- Insert each record into the **dias_catalogue** table (duplicate records will fail to insert)

Batch processing is then simply scheduled to run overnight for a fixed incremental period (e.g. yesterday to today).

13.2.2 Database catalog (SOBLOO)

SOBLOO maintains a dedicated database for processed CARD data, on a project-by-project basis. While this is a second-best solution, it is relatively straightforward to use, as we only need to transfer records between databases. All metadata is stored in a single table **datas**, so all relevant CARD metadata records can be transferred, for the area of interest, with a single spatial query.

The transfer is modified as follows:

- Determine the database connection details for the SOBLOO **datas** table (provided with the SOBLOO account)
- Build a spatial query for the area of interest, the search start and end date
- Fire off the query to the SOBLOO database
- Parse the relevant attributes in the query response and reformat to **dias_catalogue** compatible table fields
- (Note that the geometries in the **datas** table have their lon, lat coordinates flipped. We use the postgis function **ST_FlipCoordinates()** to correct this).
- Insert each record into the **dias_catalogue** table (duplicate records will fail to insert)

13.3 Benchmarking data formats for fast image processing

(Work in progress)

Problem statement: for fast image processing and visualization, the image data organization play an essential role. Whereas many standard formats exist, there is still little information on how formats relate to “typical” timing of standard image processing and visualization needs. This is further complicated by the fact that image processing is now moving from single platform to cloud based solutions, and from limited selections of locally stored image data to global coverages of full data archives. Thus distributed data storage needs to be combined with massively parallel execution of processing code, and render meaningful results in increasingly demanding analytics and visualization workflow.

Pre-conditions

Data formats/solutions need to adhere to a minimum number of conditions:

- No changes to the original resolution of the source data: the new format shall neither change spatial resolution (pixel spacing), nor radiometric resolution (drop bands), nor data precision (bit depth of the image bands). It should also keep the original projection. In other words, there is full reproducibility between the original data and the newly formatted data.
- The format shall ensure equal access delays to any arbitrary sub-part of the image.
- The format must support use on local storage (e.g. HDD, SSD) and distributed storage (e.g. S3 cloud storage).

Minimum processing requirements:

- Seamless recomposition of processing across image sub parts, without introduction of border artefacts
- Support standard projections “on-the-fly”

Qualified solutions:

- Tiled GeoTIFF or its current Cloud Optimized GeoTIFF variant, which extends GeoTIFF with upsampled overviews. In a tiled GeoTIFF, image sub-parts are indexed inside the GeoTIFF header information.
- Tiled formats: the image is broken up in smaller tiles and either indexed by an explicit metadata format (e.g. VRT) or by an implicit tile file naming. Individual tiles are stored with a common format (e.g. GeoTIFF). Tiled formats may also store upsampled overviews, e.g. in a hierarchical file directory structure ordered according to overview (zoom) level.

We limit ourselves to full resolution, so we will not deal with upsampled overviews (for now)

Disqualified solutions:

- Data cubes are already out, because they prescribe resampling to a common grid, which does not adhere to the first pre-condition, and does not provide a globally consistent solution.

Benchmark

- Tiled GeoTIFF: single file, tile indexing in internal LUT, many standard tools read/write GeoTIFF;
- Tiled composites: many tiles, in directory structure, composed implicit index or through companion metadata file (e.g. VRT), possibly benefit from ZIP compression and archiving. VRT with virtual file system access protocols emerging standard.

The benchmark test is to select an arbitrary sized image subset for 100 random locations inside the image bounds. Mean and standard deviation across image bands are calculated to check data consistency. Tests to be run with single and multiple image bands.

PARCEL EXTRACTION

14.1 Overview

This page describes the use of parcel extraction routines. The general concept of parcel extraction is linked to the marker concept of **CAP Check by Monitoring**, which assumes that the agricultural parcel is the unit for which we want to confirm the particular agricultural activity.

In computing terms, parcel extraction leads to data reduction: we generate aggregate values from an ensemble of samples by applying some algorithm. The ensemble of samples are the pixels contained within the parcel boundary. The algorithm should, as far as possible, retain the characteristics of the parcel that we are interested in, which is typically the mean, standard deviation and percentiles statistics for the image bands.

The marker principle requires the extraction to run consistently over every CARD image that is available for the period of interest and for all parcels in the region of interest (ROI). Since image cover is usually partial, i.e. only a part of the ROI is covered by a specific image frame (or scene), different combinations of parcel ensembles can be extracted for each individual image.

There are 2 scenarios for parcel extraction: (1) a “burst” scenario, in which all scenes for a significant period (e.g. a year) need to be processed and (2) a “continuous” scenario, in which new scenes are processed incrementally as they arrive on the DIAS. Both scenarios are typically non-interactive, i.e. they should run automatically in the background. Since parcel extractions for distinct scenes are independent processes, they can be easily parallelized. This is particularly important for scenario (1). The typical workload for scenario (2) may not require parallelization.

In the next sections, we first detail the implementation of the parcel extraction routines. We then discuss parallelization using docker swarm (for scenario (1)). Finally, we list some caveats that relate to the use of the extraction routines.

This document assumes familiarity with docker and the database set up used in the JRC-DIAS project.

14.2 Implementation details

The high level implementation details of the parcel extraction routines are the following:

- The **dias_catalogue** database table is required to find the metadata for the scenes that cover the ROI, and to keep track of those scenes’ processing status. Other processes may insert new records, e.g. by scanning the DIAS catalogue for newly acquired scenes.
- The **parcel_set** database table is required to get the parcel boundaries and attributes.
- As a first step, the oldest scene that intersects the ROI and with status *ingested* is selected. The status of this scene is changed to *inprogress*.
- The *reference* attribute for the selected scene is used to compose the key for finding the scene in the S3 store. The scene is downloaded to local disk. Depending on CARD type, more than one object needs to be downloaded (e.g. several bands for S2, the .img and .hdr objects for S1).

- All parcel boundaries that are intersecting both the ROI and the footprint of the selected scene are selected from **parcel_set**.
- Using the *rasterio* and *rasterstats* python modules, zonal statistics are extracted for each parcel and each band. This is done in chunks of 2000 records.
- The extraction results (which include *count*, *mean*, *stdev*, *min*, *max*, *p25*, *p50* and *p75*) are copied into the database table **results_table**. This table uses foreign keys that reference the unique parcel id in the **parcel_set** and unique scene id in **dias_catalogue**.
- Upon successful completion of the extraction, the scene status in **dias_catalogue** is changed to *extracted* and the local copies of the image file are removed.

Separate versions are kept for each of the ‘bs’, ‘c6’ and ‘s2’ CARD sets, to address the different data formats used. For ‘s2’, the extraction currently handles only the 10m bands B04 (Red) and B08 (NIR) (used for NDVI calculation) and the 20m SCL scene classification band to integrate information on cloud conditions and data quality. This can obviously be reconfigured to include other or more bands. For ‘bs’ and ‘c6’ both VV and VH bands are extracted.

14.3 Configuration files

The extraction routines must be configured via 2 parameters files that use the JSON format. In `_db_config.json` the database parameters are defined. This includes the set of connection parameters, the tables used in the relevant queries and the arguments used as query parameters:

```
{
  "database": {
    "connection": {
      "host": "ip.of.the.dbserver",
      "dbname": "database_name",
      "dbuser": "database_user",
      "dbpasswd": "database_passwd",
      "port": 5432
    },
    "tables": {
      "aoi_table": "tables_with_aois",
      "parcel_table": "table_with_parcel",
      "results_table": "table_with_extracted_signatures"
    },
    "args": {
      "aoi_field": "aoi_name",
      "name": "my_aoi",
      "startdate": "2018-01-01",
      "enddate": "2019-01-01"
    }
  }
}
```

The **aoi_table** is a convenience table in which specific ROIs can be defined. It MUST have column *name* and define its geometry as `_wkb_geometry_`. This table serves to extract either full national coverages or smaller subsets, e.g. for a province.

parcel_table is assumed to be imported into the database with *ogr2ogr* and MUST have column names `_ogc_fid_` (a unique integer identifier) and `_wkb_geometry_`. These are normally assigned by *ogr2ogr*.

It is recommended to name **results_table** for separate CARD types and the period of extraction. For instance, **roi_2018_bs_signatures**.

The second parameter file `_s3_config.json_` is needed for S3 access, both for the extraction routine and the `_download_with_boto3.py_` support script.

```
{
  "s3": {
    "access_key": "s3_acces_key",
    "secret_key": "s3_secret_key",
    "host": "endpoint_url",
    "bucket": "BUCKET"
  }
}
```

14.4 Run with docker

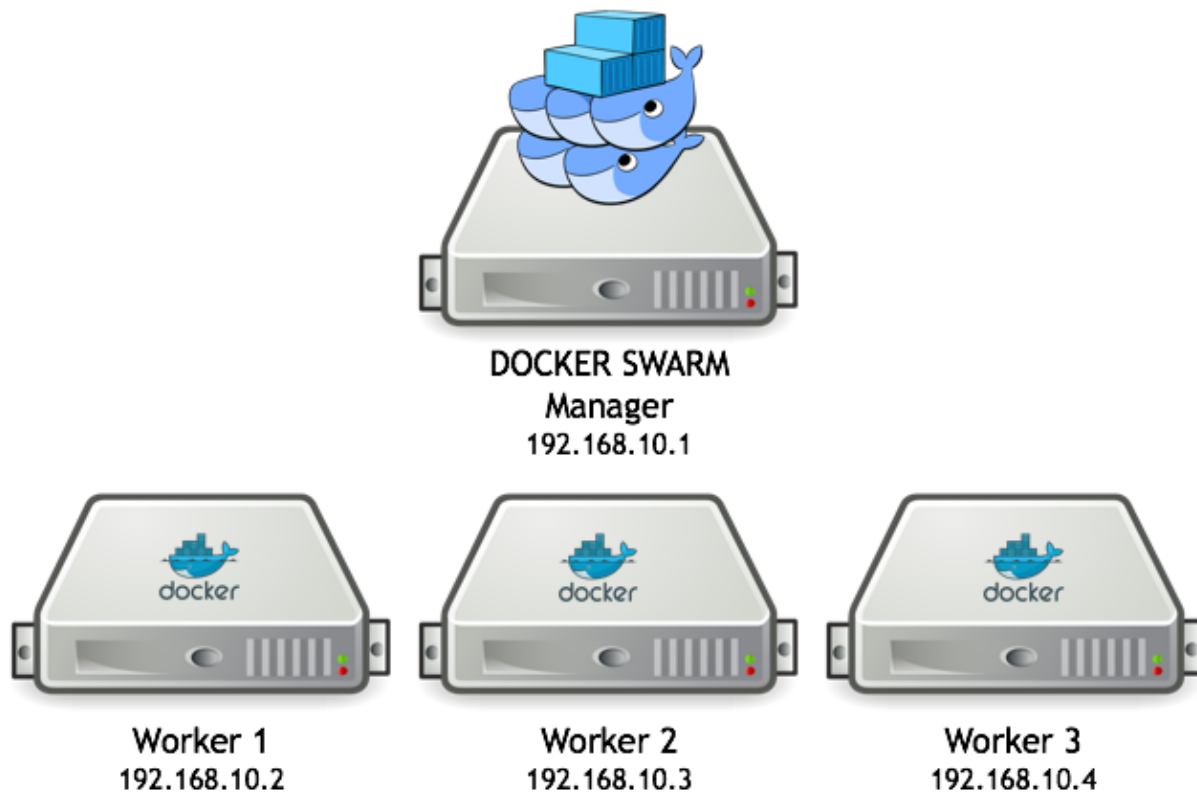
The extraction code is python3 compatible and requires a number of python modules which are packaged in the `glemoine62/dias_py` image. A single run can be launched as follows (`_working_directory_` is the location of the extraction routine):

```
cd working_directory
docker run -it --rm -v`pwd`: /usr/src/app glemoine62/dias_py python postgisS2Extract.py
```

The output will report progress on downloading relevant files, selection of the parcels and zonal statistics extraction. The total number of parcels selected for extraction primarily depends on the location of the image scene. Obviously, this number can range from a few to several 100,000s, and, thus, extraction, can take between 20 and 2000 seconds. The script eliminates parcels for which the extraction results in no data. This can happen in scene boundary areas or fractional scene coverages (common for Sentinel-2).

14.5 Parallelization with docker swarm

The extraction routines can easily run in parallel on several VMs with the use of docker swarm. We normally run the configuration as in the figure below:



The database is running on the master node and the worker nodes will run one or more extraction tasks. The worker VMs need to have a minimal configuration (a docker installation and the `glemoine62/dias_py` image), get copies of the relevant extraction routines and configuration files and then join the swarm as workers.

14.5.1 Create VMs

On the DIAS, create 4 new VMs, either through the console, or with the use of the OpenStack command line interface. Minimum configuration should be:

- VM of type `s3.xlarge.4` (with 4 vCPUs | 16 GB RAM)
- Ubuntu 18.04
- System disk size of 40 GB (only needed for temporary storage)
- No need for external IP. Ensure access from the permanent VM (with external access), using its public key.

Write down the different (internal) IP addresses assigned to the new VMs and enter these in `.ssh/config` on the permanent VM:

```
Host vm_new1
  HostName 192.168.*.*
  User cloud
  ServerAliveInterval 10
```

You can ssh in to each of the new VM instances by using its alias, e.g.:

```
ssh vm_new1
```

14.5.2 Install docker and set up docker swarm

On each of the newly created VM, install the minimum configuration, as follows:

```
ssh vm_new1
sudo apt-get update
sudo apt -y install docker.io
sudo usermod -aG docker $USER
```

The latter setting avoids the need to run docker commands as *sudo* and only takes effect after logout/login.

After re-login, get the python3 docker image with the configuration to run the extraction workflow and create a simple directory structure:

```
ssh vm_new1
docker pull glemoine62/dias_py:latest
mkdir -p dk/data
```

Copy the required code and configuration files to each of the VMs:

```
scp postgisS2Extract.py download_with_boto3.py db_config.json s3_config.json vm_new1:/
↪ home/eouser/dk
```

Set up the permanent VM as docker swarm manager:

```
docker swarm init
```

This will confirm that the current VM is now the manager and outputs a command line that should be executed on each of the new VMs, to let them join the swarm as “workers”

```
ssh vm_new1
docker swarm join --token SWMTKN-1-38wcnesfi230xn6p63izm3kr2c4ir6xi4cihvez23grd0phh6u-
↪ e01b90hoqlsm2670yts7vpp1k 192.168.30.116:2377
```

Check whether all is well:

```
docker node ls
```

ID	HOSTNAME	STATUS	AVAILABILITY
↪ MANAGER STATUS	ENGINE VERSION		
7kd7022sf1hzdf9zaqdce86jf *	bastion2	Ready	Active
↪ Leader	18.09.7		
z3sghiy1y6kvefaacysgv8cww	project-2019-0001	Ready	Active
↪	18.09.7		
c7qjl1cxfrcqbxas2fq9opm6z	project-2019-0002	Ready	Active
↪	18.09.7		
tozs74jeexdpiazhc3xqpn7q5	project-2019-0003	Ready	Active
↪	18.09.7		
fmvm1ylz2lrfn45upoisz0p7ex	project-2019-0004	Ready	Active
↪	18.09.7		

i.e. all looks fine.

14.5.3 Configure and run

In order to allow the swarm nodes to communicate with each other, an overlay network needs to be created:

```
docker network create -d overlay my-overlay
```

The docker swarm is configured with the following `_docker_compose_s2.yml_` configuration file (assuming S2 extraction):

```
version: '3.5'
services:
  pg_spat:
    image: mdillon/postgis:latest
    volumes:
      - database:/var/lib/postgresql/data
    networks:
      - overnet
    ports:
      - "5432:5432"
    deploy:
      replicas: 1
      placement:
        constraints: [node.role == manager]

  vector_extractor:
    image: glemoine62/dias_py:latest
    volumes:
      - /home/eouser/es:/usr/src/app
    networks:
      - overnet
    depends_on:
      - pg_spat
    deploy:
      replicas: 8
      placement:
        constraints: [node.role == worker]
    command: python postgisS2Extract.py

networks:
  overnet:
volumes:
  database:
```

This will create one task, `_pg_spat_`, on the manager node (i.e. the permanent VM), that runs the database server, using the mounted volume where the database outputs are written. The `_vector_extractor_` task is started on the worker nodes, as 8 replicas (thus 2 tasks on each node).

The swarm can be started as follows:

```
docker stack deploy -c docker_compose_s2.yml s2swarm
```

The swarm runs a stack of services in the background, for which the status can be checked with:

```
docker service ls
```

or the service logs displayed with:

```
docker service logs -f s2swarm_vector_extractor
```

Note that the stack continues to launch new processes. After some time there are no longer *ingested* candidate images left in the **dias_catalogue**. Check the logs to ensure that all processes return with the message “No image with status ‘ingested’ found”. Stop the stack with the command:

```
docker stack rm s2swarm
```

14.6 Caveats

The extraction routines assume standard naming of a number of required columns in the various data base table (see above). Errors will be thrown if this is not applied consistently.

The extraction routines use the copy statement to write to the **results_table** signatures data base, because this is much faster than insert statements. The **results_table** should preferable NOT have indices defined on the foreign key columns, as this will slow down mass copying into the table.

Docker stack continues to run even if no data available to process.

BACKING UP THE DATABASE

15.1 Overview

This page explains how to back up database tables. This may be needed as a precaution to avoid loss of data, or to transfer the data to another system.

The following tables are important in any backup, as they determine the values of the foreign keys in the extracted signature tables:

- The **dias_catalogue** database table is required to relate the metadata for the scenes that cover the ROI and which have 'extracted' processing status. This means the *id* column, which is a sequential number generated when records are ingested, is used as the foreign key *obsid* in the **roiYYYY_CARD_signatures** database table. This foreign key is needed to cross-reference, for instance, the *reference* and *obstime* column values for each record.
- The **parcel_set** database table is required to get the parcel *_ogc_fid_* which is the foreign key *pid* in the **roiYYYY_CARD_signatures** database table, which is required, for instance, to retrieve the geometry of the parcel and crop type attributes.
- The **roiYYYY_CARD_signatures** database tables, which contain the extracted values for each combination of *pid* and *obsid* for each of the bands of the image sets for the respective CARD sets.

Thus, in order not to loose the explicit relation defined by the foreign keys, a backup must, as a minimum, includes these tables.

15.2 Use pg_dump

The database is assumed to run within a docker container, for instance, the one derived from the *mdillon/postgis:latest* image.

Check whether the container is running:

docker ps -a				
CONTAINER ID	IMAGE	COMMAND	CREATED	
↪ STATUS	PORTS	NAMES		
7f700d1c67a2	mdillon/postgis	"docker-entrypoint.s..."	20 hours ago	
↪ Up 20 hours	0.0.0.0:5433->5432/tcp	pg_spat		

The default client tool to make database backups for PostgreSQL/Postgis database instances is `_pg_dump_`. Client tools can be directly installed on the VM that runs the docker container. However, there may be version differences between the `_pg_dump_` version of the docker host and the database running inside the docker container.

For instance, trying to back up the table **dias_catalogue** may cause the following message:

```
pg_dump -h localhost -d postgres -U postgres -p 5433 -t dias_catalogue > dias_catalogue.
↪sql
pg_dump: server version: 11.2 (Debian 11.2-1.pgdg90+1); pg_dump version: 10.6 (Ubuntu 10.
↪6-0ubuntu0.18.04.1)
pg_dump: aborting because of server version mismatch
```

This can be resolved by making sure the client tool is upgraded to 11.2, or, running `_pg_dump_` inside the docker container. The latter is preferred, as it ensures that client tools and database server versions are always aligned.

The sequence is as follows

```
docker exec -it pg_spat bash
root@7f700d1c67a2:/# pg_dump -d postgres -U postgres -t dias_catalogue > /tmp/dias_
↪catalogue.sql
root@7f700d1c67a2:/# exit
docker cp pg_spat:/tmp/dias_catalogue.sql .
```

The first command starts a *bash* shell inside the container. This causes the command prompt to change to the root user inside the container (`root@7f700d1c67a2`). `_pg_dump_` can now be used with the database that runs inside the container, e.g. to dump the table **dias_catalogue** to the file `_/tmp/dias_catalogue.sql_` inside the container. After this has finished, exit the container internal *bash* shell (command prompt turns to the one of the docker host). The last command copies the data table dump file from the docker container directory */tmp* to the current directory (`.`)

This file can now be further handled (e.g. compressed) and transferred.

BUILD A RESTFUL API

Build a RESTful API for CbM.

In order to facilitate the access to parcel time series, also for users who do not have a DIAS account, a RESTful API with [Flask](#) can be build. Flask is a micro web frameworks servers that can handle [RESTful](#) requests/responses written in Python.

16.1 Prerequisites

- Installed Docker (see: [Software requirements](#)).
- Database with extractions (see: [Parcel extraction](#)).

To build a RESTful API with flask, first [access the virtual machine](#) and download the CbM repository:

```
git clone https://github.com/ec-jrc/cbm.git
```

Then go to the api folder of the downloaded package:

```
cd cbm/api
```

16.2 Create RESTful API users

To create and manage users that can access the RESTful API for CbM, execute in the terminal:

```
python3 scripts/users.py add username password dataset # To Create a new user.
python3 scripts/users.py delete username              # Delete a user.
python3 scripts/users.py list                          # Print a list of the users.
```

Change the 'username' and 'password' with a username and password of the user. Set the dataset to the data that the user will be restricted to.

Example:

```
python3 scripts/users.py add john_doe Pass4John aoi
```

Alternatively import the module in a python script or notebook cell:

```

from scripts import users # Import the users module (set the import accordingly to your
↳path)
# Create a new user with:
users.add('username', 'password', 'dataset')

# To delete a user.
users.delete('username')

# Print a list of the users.
print(users.get_list())

```

16.3 Database connection

Open the config/db.json file with a text editor (e.g. **nano config/main.json**) and set the database connection information. e.g.:

```

{
  "db": {
    "main": {
      "desc": "Main database connection information.",
      "host": "0.0.0.0",
      "port": "5432",
      "name": "postgres",
      "sche": "public",
      "user": "postgres",
      "pass": "MyPassword"
    }
  },
  "s3": {
    "dias": "EOSC",
    "host": "http://data.cloudferro.com",
    "bucket": "DIAS",
    "access_key": "anystring",
    "secret_key": "anystring"
  }
}

```

16.4 Dataset configuration

All dataset configuration are set in the config/dataset.json file.

```

{
  "default_2020": {
    "db": "main",
    "description": "Dataset description",
    "center": "51.0,14.0",
    "zoom": "5",
    "year": "",
    "start_date": "",

```

(continues on next page)

(continued from previous page)

```

    "end_date": "",
    "extent": "",
    "flip_coordinates": "False",
    "tables": {
        "parcels": "par",
        "dias_catalog": "dias_catalogue",
        "scl": "hists_2020",
        "s2": "s2_sigs_2020",
        "bs": "bs_sigs_2020",
        "c6": "c6_sigs_2020",
        "bs_tf": "bs_tensorflow"
    },
    "pcolumns": {
        "parcel_id": "id",
        "crop_name": "name",
        "crop_code": "code"
    }
}

```

16.5 Deploy the RESTful API docker container

A docker image is available on [docker hub](#). This image includes flask and all the required python libraries needed to build a RESTful API For CbM. It can be easily deployed with:

```
docker run -it --name api -v "$PWD":/app -p 80:80 gtcap/cbm_api
```

*Must run from within the 'cbm/api/' folder

To expose the RESTful server to another port, change the port parameter -p [PORT]:80.

16.5.1 Build from source

To build the cbm_api docker image from source, go to the docker folder of cbm package

```
git clone https://github.com/ec-jrc/cbm.git
cd cbm/docker/cbm_api
```

And run:

```
docker build --tag gtcap/cbm_api .
```

Go back to api folder **cbm/api** and deployed the container with:

```
docker run -it --name api -v "$PWD":/app -p 5000:80 gtcap/cbm_api
```

16.6 Provide available options (Optional)

Add in the file “config/options.json” the available RESTful API options in the below format:

```
{
  "aois": {
    "aio": {
      "desc": "Area of Interest",
      "year": ["2019", "2018"]
    },
    "test": {
      "desc": "Sample test dataset",
      "year": ["2019"]
    }
  }
}
```

This can be retrieved from the users with the request:

```
http(s)://Host.Name.Or.IP/query/options
```

16.7 Adding orthophotos (Optional)

It is often handy to have a high resolution overview of the parcel situation. A (globally) useful set are the Google and Bing (or Virtual Earth) background image sets. To add new base maps add in the ‘tms/’ folder the xml files.

Bing base maps example:

```
<GDAL_WMS>
  <Service name="VirtualEarth">
    <ServerUrl>http://a${server_num}.ortho.tiles.virtualearth.net/tiles/a${quadkey}.
    ↪ jpeg?g=90</ServerUrl>
  </Service>
  <DataWindow>
    <UpperLeftX>-20037508.34</UpperLeftX>
    <UpperLeftY>20037508.34</UpperLeftY>
    <LowerRightX>20037508.34</LowerRightX>
    <LowerRightY>-20037508.34</LowerRightY>
    <TileLevel>19</TileLevel>
    <TileCountX>1</TileCountX>
    <TileCountY>1</TileCountY>
    <YOrigin>top</YOrigin>
  </DataWindow>
  <Projection>EPSG:900913</Projection>
  <BlockSizeX>256</BlockSizeX>
  <BlockSizeY>256</BlockSizeY>
  <BandsCount>3</BandsCount>
  <MaxConnections>4</MaxConnections>
  <Cache />
</GDAL_WMS>
```

Important notes

- In case the tables names are different and not as proposed in the ‘essential-cbm-tables’ chapter the table names of the Postges queries in the file `query_handler.py` will need to be changed accordingly with the database tables names.
- This is for testing and development purposes, for production use a more secure method to store the password should be considered.

OVERVIEW

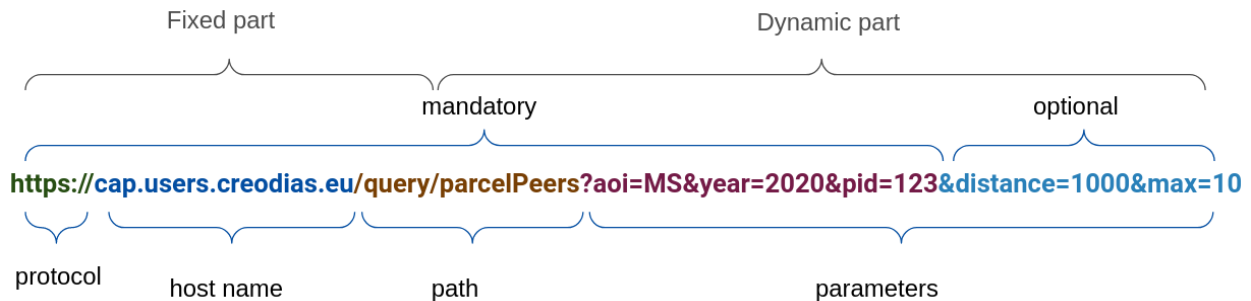
JRC RESTful service example requests have predefined logical query names that need to be configured with a set parameters. All requests return a response as a JSON formatted dictionary. The values in this dictionary are always lists, i.e. even if the query produced no (empty list) or only 1 value.

If the query is not valid, an empty dictionary will be returned (`{}`).

In this page, we keep a list of actual queries and provide some use examples.

The server of JRC RESTful example currently runs from a CloudFerro server. The root URL of the RESTful server is <https://cap.users.creodias.eu/query/>. The URL requires authentication with a username and password (which has been provided to users upon request).

Query parameters are either required or optional. The order of parameters is not significant.



Current queries

- Parcel information
 - `parcelByLocation`
 - `parcelByID`
- Parcel signatures time series
 - `parcelTimeSeries`
 - `weatherTimeSeries`
- Parcel sentinel images
 - `chipByLocation`
 - `rawChipByLocation`
 - `rawChipByParcelID`
- Parcel orthophotos

- backgroundByLocation
 - backgroundByParcelID
- Parcels lists,
 - parcelPeers
 - parcelsByPolygon

PARCEL INFORMATION

parcelByLocation - parcelByID

Get parcel information for a geographical location or a by the parcel ID. The parcels in the annual declaration sets have unique IDs, but these are not consistent between years (also, the actual parcel geometries may have changed). Thus, a query is needed to “discover” which parcel ID is at a particular geographical coordinate. Agricultural parcels are supposed to be without overlap, a unique ID will normally be returned (this is, however, not a pre-condition).

Table: **parcelByLocation** Parameters

Parameters	Description	Values	Default Value
aoi	Area of Interest (Member state or region code)	e.g.: at, pt, ie, etc.	
year	year of parcels dataset	e.g.: 2018, 2019	
lon	longitude in decimal degrees	e.g.: 6.31	
lat	latitude in decimal degrees	e.g.: 52.34	
ptype	parcels type	b, g, m, etc.	
withGeometry	adds geometry	True or False	False

Table: **parcelByID** Parameters

Parameters	Description	Example call	Values
aoi	Area of Interest (Member state or region code)	e.g.: at, pt, ie, etc.	
year	year of parcels dataset	e.g.: 2018, 2019	
pid	parcel id		
ptype	parcels type	b, g, m, etc.	
withGeometry	adds geometry	True or False	False

returns

Key	Values	Description
pid	a list of parcel ID	Normally, only 1 ID should be returned. Empty list is no parcel found
cropname	a list of crop names	Placeholder for the crop name, mapped to the original parcel table
cropcode	a list of crop code	Placeholder for the crop code, mapped to the original parcel table
srid	a list of EPSG codes	Describes the projection for the parcel geometry
area	a list of area	The area, in square meters, of the parcel geometry
clon	a list of centroid longitude	The longitude of the parcel centroid
clat	a list of centroid latitude	The latitude of the parcel centroid
geom	a list of geometries	WKT representations of the parcel geometry

ptype is used only in case there are different datasets dedicated to different type of analysis for the same year. For example datasets dedicated to grazing use **g**, for mowing **m** etc.

PARCEL TIME SERIES

19.1 parcelTimeSeries

Using RESTful services to get parcel time series

Extract time series statistics from the Sentinel data are available, both actual and archived, for all parcels in an annual declaration set.

The table below shows the parameters for time series selection.

Parameters	Description	Values	Default value
aoi	Area of Interest (Member state or region code)	e.g.: at, pt, ie, etc.	
year	year of parcels dataset	e.g.: 2018, 2019	
pid	parcel ID		
ptype	parcels type	b, g, m, atc.	
tstype	Sentinel-2 Level 2A, S1 CARD Backscattering Coefficients, S1 CARD 6-day Coherence	s2, bs, c6, scl	s2
scl	Include scl in the s2 extraction, for use in cloud screening	True or False	True
ref	Include Sentinel image reference in time series	True or False	False
tsformat	parcels type	csv, json	json

Examples: **Change the parameters aoi, year and pid based on your provided parcels data.**

- Example 1, returns the S2 time series of the parcel with SCL histograms and image reference, <https://cap.users.creodias.eu/query/parcelTimeSeries?aoi=ms&year=2020&pid=123&tstype=s2&scl=True&ref=True>
- Example 2, returns the S1 Backscattering Coefficients time series of the parcel, <https://cap.users.creodias.eu/query/parcelTimeSeries?aoi=ms&year=2020&pid=123&tstype=bs>
- Example 3, returns the S1 6-day Coherence time series of the parcel, <https://cap.users.creodias.eu/query/parcelTimeSeries?aoi=ms&year=2020&pid=123&tstype=c6>
- Example 4, returns only the SCL histogram for the selected parcel, <https://cap.users.creodias.eu/query/parcelTimeSeries?aoi=ms&year=2020&pid=123&tstype=scl>

returns

Key	Values	Description
date_part	a list of timestamps	as seconds since ‘Unix epoch’ (1970-01-01 00:00:00 UTC)
band	a list of image bands	Missing if <i>band</i> query parameter is provided
count	a list of counts	count of pixels extracted for each parcel and observation
mean	a list of means	mean etc.
std	a list of stds	standard deviation etc.
min	a list of mins	min etc.
max	a list of maxs	max etc.
p25	a list of p25s	25% histogram percentile etc.
p50	a list of p50s	50% histogram percentile etc.
p75	a list of p75s	75% histogram percentile etc.

19.2 weatherTimeSeries

Example: **Change the parameters aoi, year and pid based on your provided parcels data.**

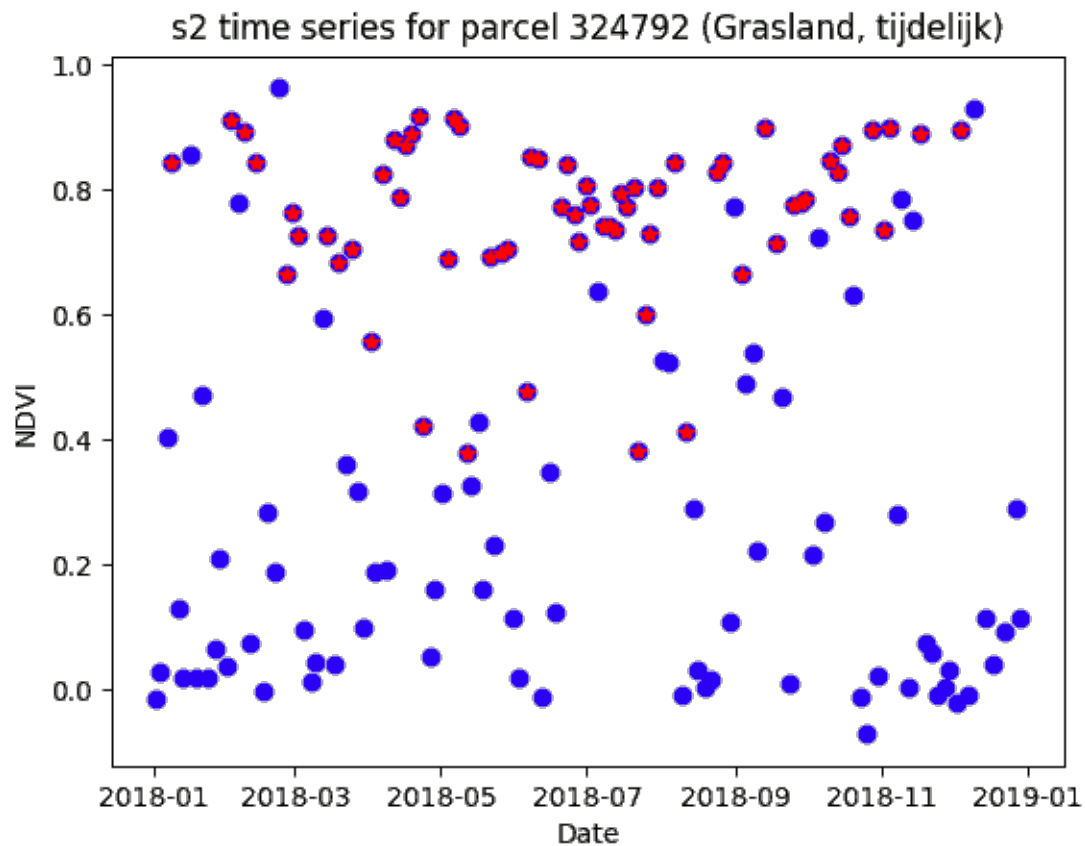
- Example 1, returns the weather time series of the parcel, <https://cap.users.creodias.eu/query/weatherTimeSeries?aoi=ms&year=2020&pid=123>

returns

Key	Description
meteo_date	a list of timestamps as seconds since ‘Unix epoch’ (1970-01-01 00:00:00 UTC)
tmean	a list of means
tmin	a list of mins
tmax	a list of maxs
prec	precipitation

19.3 Simple python client to plot parcel Time Series

We show a complete example on how to use the RESTful queries in a python script. The script first requests the parcel details at the geographical location, and then retrieves the Sentinel-2 time series in a second request. The response of the latter query is parsed into a pandas DataFrame, which allows some data reorganisation and cleanup. The cleaned data is used to generate an NDVI profile which is then plotted, resulting in a figure as the one below. The blue dots are showing all NDVI values, those with a red inset are for observations that are cloud-free according to the “scene classifier” band of Sentinel 2 Level 2A.



[Get the source](#)

```

"""
NDVI plot example
"""

import json
import requests
import pandas as pd
import matplotlib.pyplot as plt
import matplotlib.dates as mdates
from datetime import timedelta

# Define your credentials here
username = 'YOURUSERNAME'
password = 'YOURPASSWORD'
host = 'http://0.0.0.0'

# Set parcel query parameters
aoi = 'ms'
year = '2020'
ptype = ''
lon = '5.1234'

```

(continues on next page)

(continued from previous page)

```

lat = '50.1234'

# Set the cloud Scene Classification (SC) values.
scl = '3_8_9_10_11'

# ----- Plot NDVI
tstype = 's2'

# Get parcel information for a given location
parcelurl = """/query/parcelByLocation?aoi={}&year={}&lon={}&lat={}&withGeometry=True&
↳ptype={}""
response = requests.get(parcelurl.format(
    host, aoi, year, lon, lat, ptype), auth=(username, password))
parcel = json.loads(response.content)

pid = parcel['pid'][0]
crop_name = parcel['cropname'][0]
area = parcel['area'][0]

# Get the parcel Time Series
tsurl = """/query/parcelTimeSeries?aoi={}&year={}&pid={}&tstype={}&ptype={}""
response = requests.get(tsurl.format(
    host, aoi, year, pid, tstype, ptype), auth=(username, password))
# Directly create a pandas DataFrame from the json response
# This should work even if the response is and empty dictionary
df = pd.read_json(response.content)

df['date'] = pd.to_datetime(df['date_part'], unit='s')
start_date = df.iloc[0]['date'].date()
end_date = df.iloc[-1]['date'].date()
print(f"From '{start_date}' to '{end_date}'")

pd.set_option('max_colwidth', 200)
pd.set_option('display.max_columns', 20)

# Plot settings are confirm IJRS graphics instructions
plt.rcParams['axes.titlesize'] = 16
plt.rcParams['axes.labelsize'] = 14
plt.rcParams['xtick.labelsize'] = 12
plt.rcParams['ytick.labelsize'] = 12
plt.rcParams['legend.fontsize'] = 14

df.set_index(['date'], inplace=True)

dfB4 = df[df.band.isin(['B4', 'B04'])].copy()
dfB8 = df[df.band.isin(['B8', 'B08'])].copy()
datesFmt = mdates.DateFormatter('%-d %b %Y')

# Plot Cloud free NDVI.
dfNDVI = (dfB8['mean'] - dfB4['mean']) / (dfB8['mean'] + dfB4['mean'])

```

(continues on next page)

(continued from previous page)

```

df['cf'] = pd.Series(dtype='str')
scls = scl.split('_')
for index, row in df.iterrows():
    if any(x in scls for x in [*row['hist']]):
        df.at[index, 'cf'] = 'False'
    else:
        df.at[index, 'cf'] = 'True'
cloudfree = (df['cf'] == 'True')
cloudfree = cloudfree[~cloudfree.index.duplicated()]

fig = plt.figure(figsize=(16.0, 10.0))
axb = fig.add_subplot(1, 1, 1)

axb.set_title(f"Parcel {pid} (crop: {crop_name}, area: {area:.2f} sqm)")

axb.set_xlabel("Date")
axb.xaxis.set_major_formatter(datesFmt)

axb.set_ylabel(r'NDVI')
axb.plot(dfNDVI.index, dfNDVI, linestyle=' ', marker='s',
        markersize=10, color='DarkBlue',
        fillstyle='none', label='NDVI')

axb.plot(dfNDVI[cloudfree].index, dfNDVI[cloudfree],
        linestyle=' ', marker='P',
        markersize=10, color='Red',
        fillstyle='none', label='Cloud free NDVI')

axb.set_xlim(start_date, end_date + timedelta(1))
axb.set_ylim(0, 1.0)

axb.legend(frameon=False)

plt.show()

```


SENTINEL IMAGE CHIPS

Using RESTful services to get parcel time series as image chips

Please read up on the [generic characteristics of the RESTful service](#) before using the queries described here.

WARNING: The query described here is **resource intensive**. It starts a complex task that involves communication between docker containers and parallel execution on several DIAS VMs. The following limitations apply:

- Chip extraction is for Level-1C (L1C) and Level-2A (L2A) Sentinel-2. L2A is not available everywhere, but generally in the CbM onboarding Member States and those in other pilots (e.g. SEN4CAP, EOSC-EAP) where CREODIAS was used. L1C is available globally.
- Selection is by date range. The current limit per request is 24 chips. No cloud filtering is applied. Shorten the time window if it leads to > 24 chips being selected (error output is still rudimentary).
- The service applies a simple rule to exclude duplicate chips for the same date (e.g. in tile overlaps). However, a location may be in a multiple orbit overlap, which will generate denser time series than the usual 5 day repeat cycle of the A and B sensors. Thus, you may need to shorten the time window in order not to exceed the 24 chip limit.
- The server has a simple caching mechanism. If you run the query again, for the same location, with a different time window, it will only generate the new chips and return the rest from cache. This can be used to create longer time series faster. In case you do not want the cached chips, you can request only the new chips by changing slightly the location.
- The cache is flushed regularly, typically at midnight (CET).
- **Do not refresh the link** if it is still running, as it will start a new selection process and currently screws up the sequence.

NOTE: The use of these RESTful queries on JRC RESTful example is meant for occasional use and testing. If you have a use case that requires many chip generations, please contact us for alternative methods.

All requests are logged, using your internet address (IP). Furthermore, the caching is based on a combination of your IP and query parameters. We regularly check the cache and will contact you in case we detect excessive use.

20.1 chipByLocation

Generates a series of extracted Sentinel-2 LEVEL2A segments of 128x128 pixels as a composite of 3 bands.

Currently, parameter values can be as follows:

Pa- rame- ters	Format	Description
lon, lat	a string representing a float number	Any geographical coordinate where Level-2A Sentinel-2 is available
start_date, end_date	YYYY-mm-dd	Time window for which Level-2A Sentinel-2 is available (after 27 March 2018)
<i>lut</i>	Low_High	A pair of float values between 0 and 100 separated by an underscore (_). Low must be smaller than High. Defaults to 5_95
<i>lut</i>	LB1_HB1_LB2_HB2_LB3_HB3	A pair of float values, with each value separated by an underscore (_). Each pair are the absolute low and high thresholds, applied to the band selection
<i>bands</i>	Bn1_Bn2_Bn3	33 Sentinel-2 band names. One of ['B02', 'B03', 'B04', 'B08'] (10 m bands) or ['B05', 'B06', 'B07', 'B8A', 'B11', 'B12'] (20 m bands). 10m and 20m bands can be combined. The first band determines the resolution in the output composite. Defaults to B08_B04_B03.
<i>plevel</i>	string	LEVEL2A, LEVEL1C. Use LEVEL1C where LEVEL2A is not available

returns

An HTML page that displays the selected chips in a table with max. 8 columns.

20.2 rawChipByLocation

rawChipByLocation - rawChipByParcelID

Generates a series of extracted Sentinel-2 LEVEL2A segments of 128x128 (10m resolution bands) or 64x64 (20 m) pixels as list of full resolution GeoTIFFs.

Table: **rawChipByLocation** Parameters

Parameters	Description	Values	Default Value
lon	longitude in decimal degrees	e.g.: 6.31	
lat	latitude in decimal degrees	e.g.: 52.34	
start_date, end_date	Time window for which Level-2A Sentinel-2 is available (after 27 March 2018)	Format: YYYY-mm-dd	
band	Sentinel-2 band name. One of ['B02', 'B03', 'B04', 'B08'] (10 m bands) or ['B05', 'B06', 'B07', 'B8A', 'B11', 'B12', 'SCL'] (20 m bands).	Format: BXX	
chipsize	size of the chip in pixels	< 5120	1280
plevel	Processing levels. Use LEVEL1C where LEVEL2A is not available	LEVEL2A, LEVEL1C	LEVEL2A

Table: **rawChipByParcelID** Parameters

Parameters	Description	Example call	Values
aoi	Area of Interest (Member state or region code)	e.g.: at, pt, ie, etc.	
year	year of parcels dataset	e.g.: 2018, 2019	
pid	parcel id		
ptype	parcels type	b, g, m, etc.	
start_date, end_date	Time window for which Level-2A Sentinel-2 is available (after 27 March 2018)	Format: YYYY-mm-dd	
band	Sentinel-2 band name. One of ['B02', 'B03', 'B04', 'B08'] (10 m bands) or ['B05', 'B06', 'B07', 'B8A', 'B11', 'B12', 'SCL'] (20 m bands).	Format: BXX	
chipsize	size of the chip in pixels	< 5120	1280
plevel	Processing levels. Use LEVEL1C where LEVEL2A is not available	LEVEL2A, LEVEL1C	LEVEL2A

Examples:

- Example 1: List of downloadable Sentinel images for a given location: https://cap.users.creodias.eu/query/rawChipByLocation?lon=5.123&lat=55.123&start_date=2020-01-01&end_date=2020-01-30&band=SCL&chipsize=920
- Example 2: List of downloadable Sentinel images for a given parcel ID: https://cap.users.creodias.eu/query/rawChipByParcelID?aoi=MS&year=2020&pid=1234&start_date=2020-01-01&end_date=2020-01-30&band=B04&chipsize=920

returns

A JSON dictionary with date labels and relative URLs to cached GeoTIFFs e.g.:

```
{"dates": ["20200101T105441", "..."], "chips": ["/dump/..._20200101T121106.SCL.tif", "..."]}
```

You will need to add the host to download the image e.g.:

```
https://cap.users.creodias.eu/dump/..._20200129T130207.SCL.tif
```

The preferred way is to use a client script to transfer the GeoTIFFs and run analysis on it.

20.3 Example client code.

The code example builds on the [client code of the basic RESTful services](#), and integrates some more advanced processing concepts that help build up to more advanced logic in the next steps.

First, we locate a parcel by location, as before, but now use the geometry that is (optionally) returned by *parcelByLocation* to refine the positioning of the chip selection. This introduces some basic feature geometry handling that is possible with the GDAL python libraries, including geometry creation for JSON, reprojection and centroid extraction. For the centroid position, the *rawChipByLocation* is launched to retrieve the SCL band for the Sentinel-2 Level-2A data. SCL is produced in the atmospheric correction step of SEN2COR, and contains mask values that link to the Scene Classification. These mask values are useful to determine whether a pixel (in the parcel) is cloud free.

rawChipByLocation first stages the selections, as GeoTIFFs, in the user cache on the server and then returns a JSON dictionary with the file locations. This information is then used to download the GeoTIFFs to local disk. Finally, we iterate through the list of downloaded GeoTIFFs and extract the histogram for the parcel pixels.

Get the code

```
import sys
import glob
import json
import requests
import rasterio
import pandas as pd
from datetime import datetime
from rasterstats import zonal_stats
from osgeo import osr, ogr

# Define your credentials here
username = 'YOURUSERNAME'
password = 'YOURPASSWORD'
host = 'http://0.0.0.0'

# Set parcel query parameters
aoi = 'ms'
year = '2020'
ptype = ''
lon = '5.1234'
lat = '50.1234'

# Parse the response with the standard json module
# Get parcel information for a given location
parcelurl = "{}{}query/parcelByLocation?aoi={}&year={}&lon={}&lat={}&withGeometry=True&
↳ ptype={}"
response = requests.get(parcelurl.format(host, aoi, year, lon, lat, ptype), auth =
↳ (username, password))
parcel = json.loads(response.content)

# Check response
if not parcel:
    print("Parcel query returned empty result")
    sys.exit()
elif not parcel.get(list(parcel.keys())[0]):
    print(f"No parcel found in {parcels} at location ({lon}, {lat})")
    sys.exit()

print(parcelurl.format(host, aoi, year, lon, lat, ptype))

# Create a valid geometry from the returned JSON withGeometry
geom = ogr.CreateGeometryFromJson(parcel.get('geom')[0])

source = osr.SpatialReference()
source.ImportFromEPSG(parcel.get('srid')[0])

# Assign this projection to the geometry
geom.AssignSpatialReference(source)

target = osr.SpatialReference()
```

(continues on next page)

(continued from previous page)

```

target.ImportFromEPSG(4326)

transform = osr.CoordinateTransformation(source, target)

# And get the lon, lat for its centroid, so that we can center the chips on
# the parcel
centroid = geom.Centroid()
centroid.Transform(transform)

# Use pid for next request
pid = parcel['pid'][0]
cropname = parcel['cropname'][0]

# Set up the rawChip request
rawurl = """/query/rawChipByLocation?lon={}&lat={}&start_date={}&end_date={}&band={}&
↳chipsize={}""

# Images query parameter values
start_date='2019-06-01'
end_date='2019-06-30'
band='SCL'      # Start with the SCL scene, to check cloud cover conditions
chipsize=2560 # Size of the extracted chip in meters (5120 is maximum)

print(rawurl.format(host, str(centroid.GetY()), str(centroid.GetX()), start_date, end_
↳date, band, chipsize))
response = requests.get(rawurl.format(host, str(centroid.GetY()), str(centroid.GetX()),
↳start_date, end_date, band, chipsize), auth=(username, password))

# Directly create a pandas DataFrame from the json response
df = pd.read_json(response.content)

# Check for an empty dataframe
if df.empty:
    print(f"rawChip query returned empty result")
    sys.exit()

print(df)

# Download the GeoTIFFs that were just created in the user cache
for c in df.chips:
    url = f"{host}/{c}"
    res = requests.get(url, stream=True)
    outf = c.split('/')[-1]
    print(f"Downloading {outf}")
    handle = open(outf, "wb")
    for chunk in res.iter_content(chunk_size=512):
        if chunk: # filter out keep-alive new chunks
            handle.write(chunk)
    handle.close()

# Check whether our parcel is cloud free

```

(continues on next page)

(continued from previous page)

```

# We should have a list of GeoTIFFs ending with .SCL.tif
tiflist = glob.glob('*.SCL.tif')

for t in tiflist:
    with rasterio.open(t) as src:
        affine = src.transform
        CRS = src.crs
        data = src.read(1)

    # Reproject the parcel geometry in the image crs
    imageCRS = int(str(CRS).split(':')[1])

    # Cross check with the projection of the geometry
    # This needs to be done for each image, because the parcel could be in a
    # straddle between (UTM) zones
    geomCRS = int(geom.GetSpatialReference().GetAuthorityCode(None))

    if geomCRS != imageCRS:
        target = osr.SpatialReference()
        target.ImportFromEPSG(imageCRS)
        source = osr.SpatialReference()
        source.ImportFromEPSG(geomCRS)
        transform = osr.CoordinateTransformation(source, target)
        geom.Transform(transform)

    # Format as a feature collection (with only 1 feature) and extract the histogram
    features = { "type": "FeatureCollection",
                  "features": [{"type": "feature", "geometry": json.loads(geom.ExportToJson()),
    ↪ "properties": {"pid": pid}}]}
    zs = zonal_stats(features, data, affine=affine, prefix="", nodata=0,
    ↪ categorical=True, geojson_out=True)

    # This has only one record
    properties = zs[0].get('properties')

    # pid was used as a dummy key to make sure the histogram values are in 'properties'
    del properties['pid']

    histogram = {int(float(k)):v for k, v in properties.items()}
    print(t, histogram)

```

The code creates the following output:

```

S2A_MSIL2A_20190615T105031_N0212_R051_T31UFU_20190615T121337.SCL.tif {9: 187}
S2A_MSIL2A_20190602T104031_N0212_R008_T31UFU_20190602T140340.SCL.tif {8: 187}
S2A_MSIL2A_20190605T105031_N0212_R051_T31UFU_20190605T152839.SCL.tif {9: 187}
S2B_MSIL2A_20190617T104029_N0212_R008_T31UFU_20190617T131553.SCL.tif {4: 187}
S2B_MSIL2A_20190607T104029_N0212_R008_T31UFU_20190607T135245.SCL.tif {8: 187}
S2A_MSIL2A_20190622T104031_N0212_R008_T31UFU_20190622T120726.SCL.tif {4: 1, 5: 44, 7: 42,
    ↪ 8: 63, 9: 37}
S2B_MSIL2A_20190610T105039_N0212_R051_T31UFU_20190610T133632.SCL.tif {8: 26, 10: 161}
S2A_MSIL2A_20190612T104031_N0212_R008_T31UFU_20190612T133140.SCL.tif {8: 187}

```

(continues on next page)

(continued from previous page)

```

S2A_MSIL2A_20190625T105031_N0212_R051_T31UFU_20190625T134744.SCL.tif {4: 186, 5: 1}
S2B_MSIL2A_20190627T104029_N0212_R008_T31UFU_20190627T135004.SCL.tif {9: 187}
S2B_MSIL2A_20190620T105039_N0212_R051_T31UFU_20190620T140845.SCL.tif {4: 187}

```

i.e., for each image set, the histogram (a dictionary) shows how many of the 187 pixels included in the parcel (at 20 m resolution) are in each SCL class. The SCL class keys have the following meaning:

Scene Classification Legend		
	1	 Saturated or defective pixels
	2	 Dark features / Shadows
x	3	 Cloud shadows
	4	 Vegetation
	5	 Not-vegetated
	6	 Water
	7	 Unclassified
x	8	 Cloud medium probability
x	9	 Cloud high probability
	10	 Thin cirrus
x	11	 Snow or ice

(Courtesy of Csaba Wernhardt, JRC)

thus, only 3 out of the 11 image in June 2018 are cloud free. One has mixed values, and 7 are completely cloud covered.

You can now decide to go pick up the relevant bands for the 4 cloud free acquisitions. This can, in principle, be done with *rawChipByLocation* but that is not a very efficient procedure, because that query does not benefit well from parallel processing.

Continue on the [next WIKI page](#) to find a more advanced solution.

PARCEL ORTHOPHOTOS

backgroundByLocation - backgroundByParcelID

It is often handy to have a high resolution overview of the parcel situation. A (globally) useful set are the Google and Bing (or Virtual Earth) background image sets.

This query generates an extract from either Google or Bing. It uses the WMTS standard to grab and compose, which makes it fast (does not depend on DIAS S3 store).

Currently, parameter values can be as follows:

Table: **backgroundByLocation** Parameters

Parameters	Description	Values	Default Value
lon	longitude in decimal degrees	e.g.: 6.31	.
lat	latitude in decimal degrees	e.g.: 52.34	.
chipsize	size of the chip in pixels	< 1280	256
extend	size of the chip in meters	.	256
tms	tile map server	Google, Bing, OSM or Orthophotos	Google
ifformat	image format	tif or png	png

Table: **backgroundByParcelID** Parameters

Parameters	Description	Example call	Values
aoi	Area of Interest (Member state or region code)	e.g.: at, pt, ie, etc.	•
year	year of parcels dataset	e.g.: 2018, 2019	•
pid	parcel id		•
ptype	parcels type	b, g, m, atc.	•
chipsize	size of the chip in pixels	< 1280	256
extend	size of the chip in meters	•	256
tms	tile map server	Google, Bing, OSM or MS Orthophotos	Google
ifformat	image format	tif or png	png
withGeometry	show parcel polygon overlay if ifformat=png	True or False	False

Examples:

- Example 1, returns a background google earth image of a selected location, <https://cap.users.creodias.eu/query/backgroundByLocation?lon=1.333370&lat=41.557789&chipsize=256&extend=500&tms=es2020&ifformat=png>
- Example 2, returns a background google earth image of a selected parcel, <https://cap.users.creodias.eu/query/backgroundByParcelId?aoi=ms&year=2020&pid=123&chipsize=256&extend=512&ifformat=png>

returns

An HTML page that displays the selected chip as a PNG tile. Generates a GeoTIFF as well, for use in rasterio processing.

PARCEL PEERS

22.1 parcelPeers

Get the parcel “peers” for a known parcel ID, i.e. parcels with the same crop type as the reference within a certain distance. This can be useful for checking the relative behavior of a parcel against its nearest neighbors (with the use of *parcelTimeSeries*)

Parameters	Description	Values	Default Value
aoi	Area of Interest (Member state or region code)	e.g.: at, pt, ie, etc.	
year	year of parcels dataset	e.g.: 2018, 2019	
pid	parcel id		
ptype	parcels dedicated to different analyses	b, g, m, atc.	
distance	maximum distance to search around parcel with pid	< 5000.0 atc.	1000.0
max	maximum number of peers to return	< 100	10

Example: <https://cap.users.creodias.eu/query/parcelPeers?parcels=ms&year=2020&pid=315141>

returns

Key	Values	Description
pid	a list of parcel IDs	
distance	a list of distances	In ascending order

22.2 parcelStatsPeers

Get the parcel “peers” for a known parcel ID, i.e. parcels with the same crop type as the reference within a certain distance. This can be useful for checking the relative behavior of a parcel against its nearest neighbors (with the use of *parcelTimeSeries*)

Parameters	Description	Values	Default Value
aoi	Area of Interest (Member state or region code)	e.g.: at, pt, ie, etc.	
year	year of parcels dataset	e.g.: 2018, 2019	
ptype	parcels dedicated to different analyses	b, g, m, etc.	
band	Sentinel 1 or 2 band	B02, B03, B04, B05, B08, B11, VVc, VVb	
values	a specific value or value range e.g.: '100-200'.	b, g, m, etc.	
stype	the stats type	mean, max, min, p25, p50, p75	mean
max	maximum number of peers to return	•	100

Example: https://cap.users.creodias.eu/query/parcelStatsPeers?aoi=at&ptype=m&year=2020&start_date=2020-05-01&end_date=2020-06-01&band=B08&values=10000-11000

returns

22.3 parcelsByPolygon

Get a list of parcels within a given polygon.

Parameters	Description	Values	Default Value
aoi	Area of Interest (Member state or region code)	e.g.: at, pt, ie, etc.	
year	year of parcels dataset	e.g.: 2018, 2019	
pid	parcel id		
ptype	parcels dedicated to different analyses	b, g, m, etc.	
polygon	polygon coordinates		
max	maximum number of parcels to return	< 100	10

Example: [https://cap.users.creodias.eu/query/parcelsByPolygon?aoi=AA&year=2020&polygon=\[\[\]polygon_coordinates\]](https://cap.users.creodias.eu/query/parcelsByPolygon?aoi=AA&year=2020&polygon=[[]polygon_coordinates])

returns

ptype is used only in case there are different datasets dedicated to different type of analysis for the same year. For example datasets dedicated to grazing use **g**, for mowing **m** etc.

DATA ANALYTICS

Using the RESTful services to support data analytics for CbM

On this page, we work out a case study for a practical CbM case. For a limited area of interest, we select all parcel IDs. For each of the parcel IDs, we first check the extracted signature statistics. Given some criteria, we sort on expected heterogeneity of the parcels. This is calculated from the signature statistics. Heterogeneity must be meaningful with respect to the crop stage for which we expect a homogenous parcel to have minimum variability.

For the top 10 heterogeneous parcels, we then extract chips for further testing.

23.1 parcel selection

To illustrate that we are dealing with a realistic scenario, we select an arbitrary area in NRW in 2018, and do a direct database extraction of all parcels in a 5000x5000 sqm square buffer around a coordinate of interest. We do not have a dedicated RESTful service for this, but in reality, the parcel IDs should come from the existing registry of the PA.

```
SELECT ogc_fid, mon10_cr_2, st_area(wkb_geometry)/10000.0 As area,
       st_x(st_transform(st_centroid(wkb_geometry), 4326)) as clon,
       st_y(st_transform(st_centroid(wkb_geometry), 4326)) as clat
FROM nrw2018
WHERE st_intersects(wkb_geometry,
                    st_envelope(st_buffer(st_transform(st_geomfromtext('POINT(6.365 51.0198)',
                                                                    4326), 25832), 2500)))
ORDER by st_area(wkb_geometry) desc;
```

this produces a list of 840 parcel IDs, sorted by descending parcel area size, which ranges from 30.6 to 0.013 ha. Only 39 parcels are smaller than 0.2 ha, so “small parcel occurrence” is not a very significant issue. In fact, it helps us appreciate the added value of our heterogeneity analysis: detecting a non-compliant subdivide of several hectares in any of the larger parcels is more significant than checking all the “small parcels” together (**focus!**)

23.2 time series statistics

For each parcel ID, we can now collect the S2 (or S1) time series. In both, we assume heterogeneity is expressed in:

- a larger than “normal” standard deviation (or stdev ratio over mean) of the signal values
- a skewed distribution of the signal values in the parcel histogram
- at the phenology phase at which we expect a full crop cover
- a clustering of values in distinct subsegments

We deal with clustering in the next section, because we can consider it as a post-processing step to confirm the findings in time series statistics. This is the main reason why the JRC DIAS implementation includes automated parcel extraction routines: most of the relevant heterogeneity information is in the statistical extracts. More complex analysis and visual inspection of chips is only needed to resolve the less obvious cases (**cost!**)

The steps outlined above are already served by the existing RESTful routines, in particular the *parcelTimeSeries* routine. The challenge is to define what is a “normal” standard deviation of the signal and at which phenology stage (i.e. time window) we want to test. Obviously, the latter is depended on the crop type, e.g. we expect winter crops to have a different timing than summer crops. To a lesser extent, we need be aware of heterogeneity effects that may occur in crop covers (e.g. alfalfa, grass due to partial mowing).

23.3 histogram analysis, clustering

23.4 ...

POST REQUESTS

Taking the next step: Using POST with RESTful services to get tailor made selections

Please read up on the [generic characteristics of the RESTful service](#) AND [RESTful queries for chip selections](#) **BEFORE** using the query described here.

24.1 rawChipsBatch

This query extends *rawChipByLocation* by providing a faster parallel extraction.

It requires the POST method, instead of the GET methods used in the previous pages, because the parameters are now lists of detailed references to individual chip selections. This also means that the query no longer works in a standard browser (unless you install special extensions like [postman](#) in your browser, though that is mostly useful as a debug tool).

Using POST has the additional advantage that it is much easier to provide request parameters as more complex structures, e.g. including lists. We will post JSON dictionary structures.

The overall idea behind using *rawChipBatch* is that you already have made a selection of specific items that you want to collect from the server. The items are passed as a list, which is then extracted on parallel VMs on the server. A common scenario for Sentinel-2 selection would be to first screen the SCL information for a parcel (either using *parcelTimeSeries* or *rawChipByLocation*) and then POST the cloud-free selections to retrieve the bands of interest.

The cartesian product of selected references and unique bands is returned (e.g. if 4 references and 3 bands are provided as parameters, 12 GeoTIFF chips are produced).

Currently, parameter values can be as follows:

Pa- rame- ters	Format	Description
lon, lat	float num- bers	Any geographical coordinate where Level-2A Sentinel-2 is available
tiles	list of strings	Time window for which Level-2A Sentinel-2 is available (after 27 March 2018)
bands	list of strings	Sentinel-2 band name. Selection out of ['B02', 'B03', 'B04', 'B08'] (10 m bands) or ['B05', 'B06', 'B07', 'B8A', 'B11', 'B12', 'SCL'] (20 m bands).
<i>chip- size</i>	int	1280 (default), truncated to 5120, if larger

returns

A JSON dictionary with date labels and relative URLs to cached GeoTIFFs.

You need a client script to transfer the GeoTIFFs and run analysis on it.

24.2 Example test script

Based on the result of the example script for *rawChipByLocation* the following script retrieves the bands B04, B08 and B11 for the 3 scenes for which the parcel histogram showed that they are without cloud masked SCL pixels. Instead of downloading the bands from cache, we generate NDVI directly from the cached bands B08 and B04, and scale the result to a byte image which will be given a [matplotlib color map](#) of choice.

For the simple client side image processing tasks, you need:

Simple client-side image processing

Get the code

```
import requests

import imgLib

username = 'YOURUSERNAME'
password = 'YOURPASSWORD'
host = '0.0.0.0'

# Authenticate directly in URL
tifUrlBase = f"http://{username}:{password}@{host}"
url = tifUrlBase + "/query/rawChipsBatch"

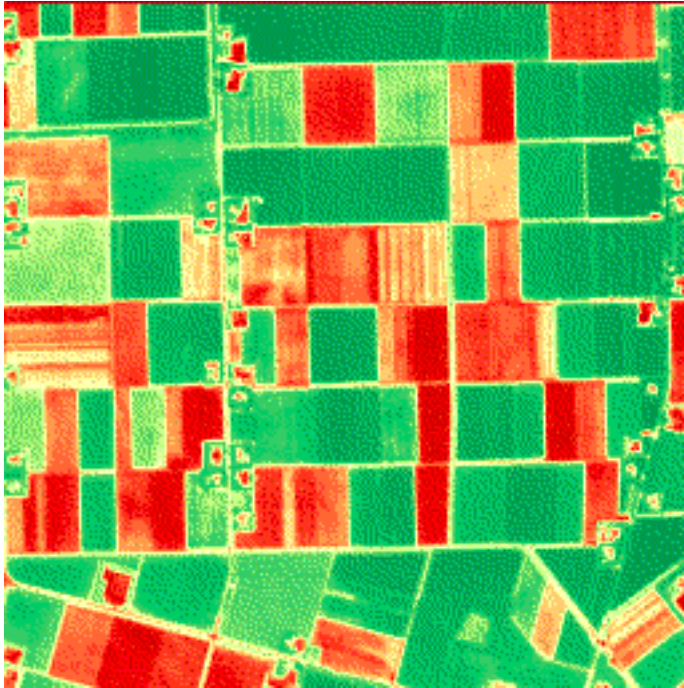
payloadDict = {
    "lon": 5.66425,
    "lat": 52.69444,
    "tiles": ["S2B_MSIL2A_20190617T104029_N0212_R008_T31UFU_20190617T131553",
             "S2A_MSIL2A_20190625T105031_N0212_R051_T31UFU_20190625T134744",
             "S2B_MSIL2A_20190620T105039_N0212_R051_T31UFU_20190620T140845"],
    "bands": ["B08", "B04", "B11"],
    "chipsize": 2560
}

response = requests.post(url, json=payloadDict)

jsonOut = response.json()

# Instead of transferring the image bands from cache
# generate an normalizedDifference output directly from the cache URLs
for c in jsonOut.get('chips'):
    if c.endswith('B08.tif'):
        name = c.split('/')[1].replace('B08', 'NDVI')
        url0 = tifUrlBase + c
        url1 = url0.replace('B08.tif', 'B04.tif')

        imgLib.normalizedDifference(url0, url1, name)
        imgLib.scaled_palette(name, 0, 1, 'RdYlGn')
```

The scaled NDVI output of 2019-06-25 rendered as a PNG.

Home work: derive the NDVI histogram for parcel at this location and create a plot with the color map of the scaled version.

24.3 rawS1ChipsBatch

In analogue to the *rawChipBatch* query, this query retrieves Sentinel-1 chips. Since Sentinel-1 chips never have cloud issues, we don't need to cloud check first and make a specific selection. Instead, we can fall back to a date range selection. However, we prefer the convenience of POSTing JSON structures to pass on the request parameters. We further simplify by always returning both polarization (VV and VH for IW image mode over land).

WARNING

There are generally more Sentinel-1 observations than Sentinel-2 observations, where both ascending and descending orbits are acquired (e.g. most of Europe). Thus, the chip limit is generally reached with shorted time windows. Caching rules apply as usual.

Sentinel-1 chips are **ONLY** available where the imagery is processed to CARD (Copernicus Analysis Ready Data). This is only the case for selected European areas (on CREODIAS).

Sentinel-1 chips are selected from larger image sets and are always generated as float32 images. This (currently) takes more time than for Sentinel-2 and leads to larger file size (e.g. for transfer).

Request parameters are as follows:

Currently, parameter values can be as follows:

Parameters	Format	Description
lon, lat	float numbers	Any geographical coordinate where Level-2A Sentinel-2 is available
dates	list of strings	Start and end date of selection time window formatted as YYYY-mm-dd
<i>chipsize</i>	int	1280 (default), truncated to 5120, if larger
<i>plevel</i>	string	'CARD-BS' for geocoded GRD backscattering coefficient, or 'CARD-COH6' for 6-day coherence

returns

A JSON dictionary with date labels and relative URLs to cached GeoTIFFs.

In the client code script below, we grab the GeoTIFFs for the Sentinel-1 CARD-BS and CARD-COH6 sets for the same area as above. We then compose some of that data into a false colour composite of mean VV and VH backscattering coefficients and the 6-day coherence for the period 2019-06-14 to 2019-06-20. The composite is byte-scaled for display purposes.

Get the code

```
import requests

import rasterio
import numpy as np

# Should move to imgLib later
def byteScale(img, min, max):
    return np.clip(255 * (img - min) / (max - min), 0, 255).astype(np.uint8)

username = 'YOURUSERNAME'
password = 'YOURPASSWORD'
host = '0.0.0.0'

tifUrlBase = f"http://{username}:{password}@{host}"
url = tifUrlBase + "/query/rawS1ChipsBatch"
payloadDict = {
    "lon": 5.66425,
    "lat": 52.69444,
    "dates": ["2019-06-10", "2019-06-25"],
    "chipsize": 2560,
    "plevel": 'CARD-BS'
}

# First get the CARD-BS
response = requests.post(url, json=payloadDict)
json_BS = response.json()

# then get CARD-COH6
payloadDict['plevel'] = 'CARD-COH6'
response = requests.post(url, json=payloadDict)
json_C6 = response.json()

# Now let's do some arty compositing
```

(continues on next page)

(continued from previous page)

```

# We'll generate a mean VV and VH for the period
# and add coherence

g0 = [f for f in json_BS.get('chips') if f.find('20190614')!=-1]
g1 = [f for f in json_BS.get('chips') if f.find('20190620')!=-1]
c0 = [f for f in json_C6.get('chips') if f.find('20190614')!=-1 and f.find('20190620')!=-
→ 1]

g0VV = [g for g in g0 if g.find('VV')!=-1][0]
g0VH = g0VV.replace('VV', 'VH')
g1VV = [g for g in g1 if g.find('VV')!=-1][0]
g1VH = g1VV.replace('VV', 'VH')

# We only open the VV coherence band
c0VV = [c for c in c0 if c.find('VV')!=-1][0]

with rasterio.open(tifUrlBase + g0VV) as src:
    vv = src.read(1)
    kwargs = src.meta.copy()

# Take the average of the 2 VV bands and convert to dB
with rasterio.open(tifUrlBase + g1VV) as src:
    vv = 10.0*np.log10((vv + src.read(1))/2)

with rasterio.open(tifUrlBase + g0VH) as src:
    vh = src.read(1)

with rasterio.open(tifUrlBase + g1VH) as src:
    vh = 10.0*np.log10((vh + src.read(1))/2)

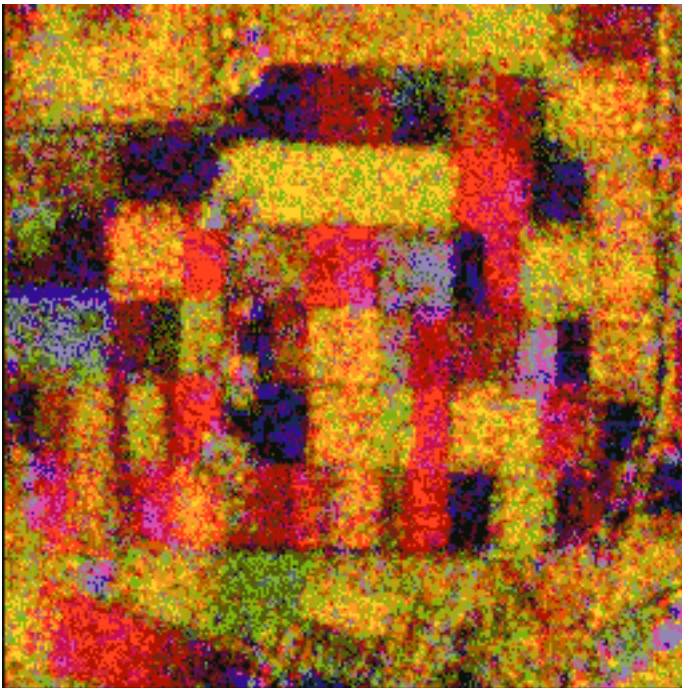
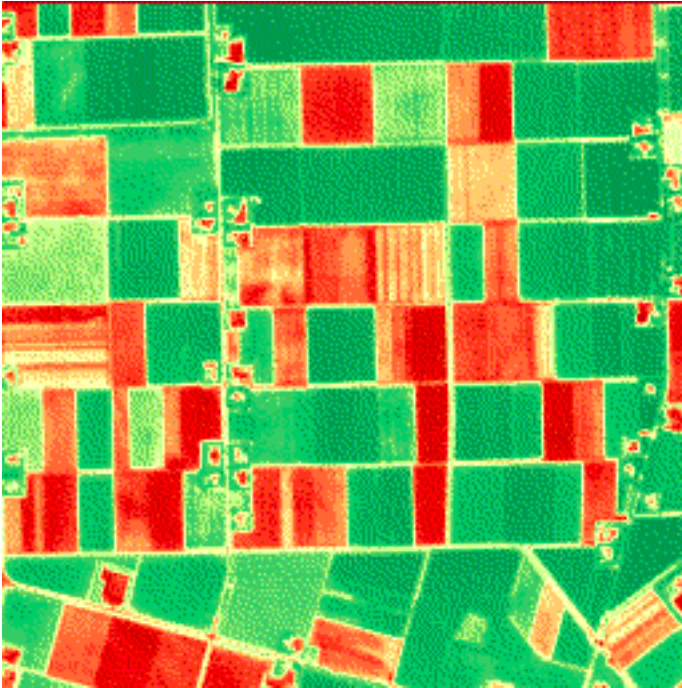
with rasterio.open(tifUrlBase + c0VV) as src:
    c6 = src.read(1)

kwargs.update({'count': 3, 'dtype': np.uint8})
with rasterio.open(f"test.tif", 'w', **kwargs) as sink:
    sink.write(byteScale(vv, -15.0, -3.0), 1) # R
    sink.write(byteScale(vh, -20.0, -8.0), 2) # R
    sink.write(byteScale(c6, 0.2, 0.9), 3) # R

```

The result is compared to the NDVI image of 2019-06-25.

At first sight, you will notice the speckly appearance of the S1 composite versus the crisp Sentinel-2 NDVI, which even shows fine inner-parcel details. But look closer, and you find more variation in the S1 composite. You can easily separate winter cereals from broadleaf crops, for instance, something that is nearly impossible in the S2 NDVI. Sparsely vegetated fields show some coherence (blue tints), etc.



SALMS R-PACKAGE

25.1 Introduction

SALMS (Signal Analyses for Land Monitoring Systems) is an R package that can help with analyses of remote sensing signals. The text below is mainly taken from the vignette of the package. The package is currently in the CBM repository in GitHub: <https://github.com/ec-jrc/cbm/tree/main/scripts/SALMS> can be installed directly in R with the command:

```
install_github("ec-jrc/cbm/scripts/SALMS")
```

The background for this package is that land cover objects are commonly observed through remote sensing. The change in land cover leads to changes in the observed remote sensing signal for an area of interest. This area could be any spatially represented feature (i.e., an agricultural parcel, forest unit). Ideally, it should have a homogeneous cover, for which we will expect all parts to react similar to an activity, with normal statistical variability.

The package was developed for help with detection of agricultural activities (typically mowing, ploughing, harvesting etc.), but can also be used for other applications. An example could be help for detection of illegal activities in natural reserves. The functions are most efficient when the single activity takes place within the entire area or feature of interest (FOI), otherwise the response depends on the proportion of the land that was changed by the activity.

This package covers much of the functionality described in the report “Proposed workflow for optimization of land monitoring systems” by Zielinski et al. (2022). The following text is mainly copied from the abstract of the report.

The background of the report is that Member States in the European Union are responsible for implementing the payment schemes to farmers. Checks of farmer activity have traditionally been performed with on-the-spot-checks, but with the free access to wall-to-wall satellite imagery from the European Copernicus programme, Checks by Monitoring was introduced to reduce of the controls burden and facilitate the management of early warnings in combination with an option to correct the aid application that could prevent non-compliances. The automation of this process helped to move it from a sample-based approach to covering the full population.

Much work has already been done regarding cloud-based solutions and large-scale detection of activities, but less attention was given to understanding the relationship between the activity and the corresponding signals, and in general the signal behaviour. This report describes a framework that could be followed to analyse the signal behaviour from a large number of bands and indices from remote sensing sensors, and to select one or a few of them as potential candidates for marker development. In the analysed data sets, indices from Sentinel-2 appears to have better discriminative power than those from Sentinel-1, but also the latter can give good results for countries where clouds reduce the availability of Sentinel-2 images. Several different statistical descriptors of the remote sensing pixels appeared to give equally good results (P25, P50, P75 and mean), but because that the median is less affected by outliers, it would be the recommended choice.

The result of signal behaviour analyses highly depends on quality and level of details of input data and are based on ground truth data, the signal time series are computed from image pixels located inside the boundaries of a feature of interest (agricultural parcel) and dates of activities. The signal selection approach is generic and can be applied to any land monitoring system where there is an interest in changes on a preselected field. The method is adjustable,

exploring (by providing a sequence of statistical methods) links between local knowledge and practices on the one side, and remote sensing images and the derived signals on the other side.

Throughout this package, we will refer to these areas as a feature of interest, FOI. Instead of analysing all pixels within the FOI object, the functionality in this package is based on a reduction of the areal observations by different statistical descriptors for each signal source.

25.2 Descriptors

The most common statistical descriptors are the ones such as mean, quantiles, standard deviation. However, there are many more, including higher order moments (skewness and kurtosis) or entropy. The default version uses 19 different descriptors, from the functions `fivenum`, `stat.desc`, and the `entropy`. If `vec` is the set of a signal values for the different pixels within a Feature of Interest (FOI), the descriptors would be:

```
# Example data for showing the descriptors
library(pastecs)
library(entropy)
vec = 1:10
fivenum(vec)
#> [1] 1.0 3.0 5.5 8.0 10.0
stat.desc(vec, basic = FALSE, norm = TRUE)
#>      median      mean    SE.mean CI.mean.0.95      var    std.dev
#> 5.50000000 5.50000000 0.9574271 2.1658506 9.1666667 3.0276504
#>  coef.var  skewness  skew.2SE  kurtosis  kurt.2SE  normtest.W
#> 0.5504819 0.0000000 0.0000000 -1.5616364 -0.5852118 0.9701646
#> normtest.p
#> 0.8923673
entropy.empirical(vec)
#> [1] 2.151282
```

The package does not include the functionality for extracting these the time series from the remote sensing images.

25.3 Input data

The analyses start with two types of .csv-files:

- A file with the dates and types of the activities, the ground truth (in-situ data, observed on the ground)
- Several files with time series of remote sensing observations

The intention is to include more flexibility regarding names, but for the moment, the time series need to have names reflecting their content based sensor source i.e. Sentinel-1 and Sentinel-2.

25.3.1 FOIinfo

The ground truth are read into `FOIinfo` which is a `data.frame`. This should contain an overview of the activities and corresponding FOIs of interest. The columns with the following names are necessary (also lower case accepted):

- `ID`: The ID of the FOI (the function also accepts `PSEUDO_ID` or `FIELD_ID`)
- `EVENT_DATE`: When did the activity happen (or start)? Also `ACTIVITY_DATE` or `EVENT_START_DATE` are accepted names
- `EVENT_TYPE`: This column should have a code for each type of activities that should be analysed

The package is quite flexible with date formats (mm-dd-yyyy, yyyy-mm-dd, etc), and will try to parse with different formats and see which one works.

The following shows the first 6 events of the example data set.

```
ddir = system.file("extdata", package = "SALMS")
FOIinfo = read.csv(paste0(ddir, "/gtdir/FOIinfo.csv"))
head(FOIinfo)
#>   X ID area Event_date Event_type
#> 1 0 1 1.3 10/07/2020      1
#> 2 1 2 0.9 13/06/2020      1
#> 3 2 3 0.8 01/07/2020      1
#> 4 3 4 1.3 27/06/2020      1
#> 5 4 5 1.2 25/07/2020      1
#> 6 5 6 1.3 30/06/2020      1
```

This example also includes the area (in ha in this case). One FOI can have several activities observed (i.e. throughout the season) .

25.3.2 ts-files

The .csv-files with time series should be stored in a single directory, for each area of interest and for each group of remote sensing signals/indexes. These files should have names according to their content. The intention is to include more flexibility regarding names, but for the moment, the time series need to have names reflecting their content based on Sentinel-1 and Sentinel-2.

Five different file names are expected (where ID is the ID of the FOI, the same as the ID of the `FOIinfo` object above):

- `ID_s1_coh6_ts.csv` - The different time series for the coherence (including ratios between orbits)
- `ID_s1_bs_ts.csv` - The different time series for the back scatter (including ratios between orbits)
- `ID_s2_ts.csv` - The different time series for S2-bands
- `ID_s2_idx_ts.csv` - The different time series for different derived S2-indexes
- `ID_s1_bsidx_ts.csv` - The different time series for indexes derived from the backscatter

All these time series need some particular columns for the function `createSignals` to be able to create the activity centred time series:

- `Date`: the date (and time) of the acquisition. This date might be an average for example for `s1_coherence` signals that are averages of different orbits or different polarizations. Ideally, these data sets should already also include the `week` column, which identifies the week of the year of the observations
- `Week`: this column is optional, but should be available for data sets that already include weekly averaged signals, such as averages of different polarizations and orbits.

- **Band:** One file can include many different bands, and this column should include the band name for each observation.
- **Orbit:** This one is only relevant for S1-signals. It can be omitted if the orbit can be found from the date (if it also includes the time of the acquisition).
- **Count:** This is optional, and only necessary if the analyses should use a weighting factor (`weighted = TRUE`) in the analyses
- **Descriptors:** The remaining columns should have the time series for the different descriptors, with the name of each descriptor as column name.
- **Histogram values:** This is an optional value, mainly used for S2-observations. This indicates the number of pixels within each class of the classification mask, for example shown towards the bottom of <https://sentinels.copernicus.eu/web/sentinel/technical-guides/sentinel-2-msi/level-2a/algorithm>.

The first lines of this file can look like this (with an example from the package)

```
ddir = system.file("extdata", package = "SALMS")
dat = read.csv(paste0(ddir, "/tsdir/1_s2_idx_ts.csv"))
head(dat)
```

#>	cwithin	chalf	call	date	band	count	MIN	P25	P50
#> 1	97	129	162	2020-01-04	GARI	97	-0.003617167	0.02064336	0.03525393
#> 2	97	129	162	2020-01-04	NDVI	97	0.056822895	0.07085278	0.07647305
#> 3	97	129	162	2020-01-04	BSI	97	-0.137596417	-0.13050421	-0.12518195
#> 4	97	129	162	2020-01-09	GARI	97	-0.108171637	-0.01309887	0.03287543
#> 5	97	129	162	2020-01-09	NDVI	97	-0.027041357	0.02374670	0.03428296
#> 6	97	129	162	2020-01-09	BSI	97	-0.089059923	-0.06720757	-0.05393071
#>	P75	MAX	MEDIAN	MEAN	SE.MEAN	CI.MEAN.0.95			
#> 1	0.04863408	0.0674882385	0.03525393	0.03439160	0.0017027393	0.003379911			
#> 2	0.08255319	0.0914504466	0.07647305	0.07634766	0.0007639961	0.001516520			
#> 3	-0.11880331	-0.1102410376	-0.12518195	-0.12463916	0.0006962173	0.001381980			
#> 4	0.06209981	0.1299435028	0.03287543	0.02446510	0.0054877141	0.010893026			
#> 5	0.04962647	0.0731707317	0.03428296	0.03427784	0.0021900500	0.004347215			
#> 6	-0.04062024	0.0001388696	-0.05393071	-0.05181195	0.0021638233	0.004295155			
#>	VAR	STD.DEV	COEF.VAR	SKEWNESS	SKEW.2SE	KURTOSIS			
#> 1	2.812341e-04	0.016770037	0.48762009	0.08456523	0.17259951	-0.8771574			
#> 2	5.661794e-05	0.007524489	0.09855559	-0.24484214	-0.49972826	-0.8114368			
#> 3	4.701770e-05	0.006856946	-0.05501438	0.04777411	0.09750801	-1.0238479			
#> 4	2.921156e-03	0.054047715	2.20917624	-0.34100108	-0.69599078	-0.5627923			
#> 5	4.652429e-04	0.021569491	0.62925474	-0.52477203	-1.07107139	-0.1101928			
#> 6	4.541667e-04	0.021311188	-0.41131801	0.52590902	1.07339200	-0.4396198			
#>	KURT.2SE	NORMTEST.W	NORMTEST.P	ENTROPY	hist.0	hist.1	hist.2	hist.3	hist.4
#> 1	-0.9036716	0.9744203	0.054855823	4.449000	0	0	0	0	0
#> 2	-0.8359644	0.9746543	0.057152028	4.569847	0	0	0	0	0
#> 3	-1.0547961	0.9719646	0.035732764	4.573210	0	0	0	0	0
#> 4	-0.5798040	0.9780403	0.103555426	5.516926	0	0	0	0	0
#> 5	-0.1135236	0.9719588	0.035697146	4.468723	0	0	0	0	0
#> 6	-0.4529084	0.9634395	0.008412672	4.472277	0	0	0	0	0
#>	hist.5	hist.6	hist.7	hist.8	hist.9	hist.10	hist.11		
#> 1	0	0	0	0	97	0	0		
#> 2	0	0	0	0	97	0	0		
#> 3	0	0	0	0	97	0	0		
#> 4	0	0	0	0	97	0	0		
#> 5	0	0	0	0	97	0	0		
#> 6	0	0	0	0	97	0	0		

The first column refers to the number of pixels that are within the outline of the FOI, depending on the approach for selecting pixels on the border (the example only uses pixels completely within the boundary, the number which is the also repeated under count).

25.4 plotTimeSeries

Before starting the analyses, it would be advisable to plot some of the original time series first, with the function `plotTimeSeries`. The function can typically print to two different devices (usually two pdfs), where the first will include a high number of plots and the second one fewer.

```
library(SALMS)
# Read data from package directory
ddir = system.file("extdata", package = "SALMS")
tsdir = paste0(ddir, "/tsdir")

# Output to tempdir, change to something local for easier access to pdfs
bdir = tempdir()

# File with activity information
FOIinfo = read.csv(paste0(ddir, "/gtmdir/FOIinfo.csv"))

# Two output files
devs = list()
for (ipdf in 1:2) {pdf(paste0(bdir, "/ts_", ipdf, ".pdf")) ; devs[ipdf] = dev.cur()}

# Plot the individual signals to see their behaviour
plotTimeSeries(FOIinfo, devs = devs, tsdir = tsdir, plotAllStats = FALSE, iyear = 2020,
  legend = "before")
#> $s1_coh6
#> list()
#>
#> $s2
#> list()
#>
#> $s1_bs
#> list()
#>
#> $s2_idx
#> list()
#>
#> $s1_bsidx
#> list()
#>
#> $s1_coh6rat
#> list()
#>
#> $s1_bsrat
#> list()
for (ipdf in 1:2) dev.off(devs[ipdf])

# Plot of one of the individual time series
plotTimeSeries(FOIinfo[2,], obands = list(s2_idx = "NDVI"), stats = "P50", tsdir = tsdir,
```

(continues on next page)

(continued from previous page)

```

→ iyear = 2020)
#> $s2_idx
#> list()

```

Figure 1 above shows the time series for NDVI, for the P50 percentile. The black line shows the date of an activity (mowing of grass in this case), as indicated in the FOIinfo-file. For the NDVI, a drop in the signal value is observed after the activity took place.

The function can produce plots for all indexes and all descriptors, normally plotted in one or several pdfs. The legend is usually good for pdfs, less good in this example.

25.5 createSignals

The function `createSignals` will take the time series above and extract activity centred and weekly time series, with a length equal to $2 \times \text{nweeks} + 1$, meaning that the new time series will have the same number of weeks before and after the activity. The newly created time series will have the same columns as the time series in the input, but also a few extra, as it will include some of the information also from `FOIinfo`. Instead of being separate for each ID, they are separated for each signal, with the possibility of having data for all FOIs in the same file. The additional column names will be:

- `EVENT_TYPE`: This column is necessary if there is more than one activity type in the data set
- `WEEK`: The week number of the observation relative to the start of the activity centred time series
- `DOY`: Which day of the year the activity happened

Some data can also be provided in a .pid-file, which are available from some downloading services. These can contain additional information, but are not mandatory. If not available, the function will mention the number of missing .pid-files at the end of the process.

Also this function will produce some plots if the variable `plotit = TRUE`.

```

odir = paste0(bdir, "/signals")
if (!dir.exists(odir)) dir.create(odir)
pdf(paste0(bdir, "/sigCreate.pdf"))
iyear = 2020
createSignals(FOIinfo, nweeks = 7, iyear, plotit = TRUE, tsdir = tsdir, odir = odir,
              iprint = 0, idxs = 1:5)
dev.off()
#> png
#> 2
print(getwd())
#> [1] "D:/git/jskoien/cbm/scripts/SALMS/vignettes"

```

Figure 2 shows an example of the output of the `createSignals`-function. The header refers to the FOI id and the day of the activity which the extracted time series is centred around. The example shows three different coherence signals for the first activity, centred around the week of day 253. The plots shows the mean and the quantiles of the signals, together with dots for the original observations.

25.6 What is week0

The analyses consider the difference between `week0` and the surrounding weeks. The `week0` is the week of the activity observed. The date of activity (`week0`) is crucial information for the analyses, and inaccuracies will influence the results. In addition the following constraints should be considered:

- First of all, depending on how manifestations have been observed on the ground, there is often some uncertainty about the exact date.
- Second, analyses in this package are done on a weekly basis. The week of the activity will therefore include observations both before and after the activity.

As a result, it is recommended to compare with the week before the activity. The default variables will create 15 week time series where the activity takes place in week 8. If ground observations are collected weakly, the recommendation is to use `week0 = 7` in all analyses.

25.7 Produce and plot means, standard deviations and t-tests

The first part of the analyses is, for a given activity type, to estimate the means and standard deviations for each of the tested signals, and also to run t-tests to see if the means are significantly different from zero. The analyses are done on a weekly basis. The easiest is to use the functions `allSignalMeans` and `allSignalTTests` for these analyses. They are both wrapper functions - around `signalMeans` and `signalTTest`. The last two functions have to be called separately for each signal and each activity type, whereas the wrapper functions will keep the different activity types separate, but include all different signals in the analyses. The output from these functions can be used as input in the functions further down. Both of the functions can plot to one or several pdf-documents. They can also open the .pdf-files based on a base name.

The `signalMean`-function will normally send all plots to two different devices, whereas `signalTTest` will send the plots to one device. The first of the devices for the `signalMean`-function will receive confidence plots, if the `plotit` argument is either "confidence", TRUE or "both". The second device will receive three additional types of plots which might be used for extra analyses.

1. A comparison between mean and median of the signals
2. The median signal value for all the FOIs in the same plot. This plot can become difficult to interpret (many lines) if there are many FOIs.
3. Boxplots of the signal values for all weeks.

As for several of the other functions, it is possible to send subsets of the plots to different devices, by setting the `sepPlots` argument. `allSignalMeans` will then need two extra plotting devices for each subset.

```
# Read all signals and find the names of all signals/indexes
allvar = readSignals(odir)
inds = names(allvar)

# Limit to the first 10 of the default descriptors
stats = SALMS::bnms[1:10]

# Open two pdfs for plotting the results from the mean function
devs = list()
for (ipdf in 1:2) {pdf(paste0(bdir, "/meanPlot", ipdf, ".pdf")) ; devs[ipdf] = dev.cur()}

# Store all the results from the calculations in evress, which contains an
# array with the means and standard deviations
```

(continues on next page)

(continued from previous page)

```

evress = allSignalMeans(allvar, events = unique(FOIinfo$Event_type), first = FALSE,
                        dels = 0, weighted = FALSE, inds = inds, weeks = 1:15, week0 = 7,
                        weekcorrect = 0, plotit = TRUE, stats = stats,
                        devs = devs, zeroshift = TRUE, FOIinfo = FOIinfo, legend =
→ "before",
                        addNum = FALSE, iprint = 1)
evres = evress$evres
cases = evress$cases
for (ipdf in 1:2) dev.off(devs[ipdf])

# Create three pdfs for t-test output
for (ipdf in 1:3) {pdf(paste0(bdir, "/ttest", ipdf, ".pdf")) ; devs[ipdf] = dev.cur()}

dfs = allSignalTTests(allvar, first = FALSE, dels = 0, weighted = FALSE,
                      inds = inds, dev = devs, weeks = 1:15, stats = stats,
                      week0 = 7, weekcorrect = 0, plotit = TRUE,
                      legend = "before")
# Close the pdfs
for (ipdf in 1:3) dev.off(devs[ipdf])

```

Figure 3 shows two plots as examples of the output of the `signalMeans` and `signalTTest` functions.

The first one shows the mean of the `s1_coh6` for all FOIs, with confidence bands based on the standard deviations of the weekly values. The header refers to the activity type (1 in this case), the statistical descriptor and the maximum number of activities used in the plot (the actual number of observations can be lower for some of the weeks).

The second plot shows the means of the NDVI for all FOIs, with the color of the dots according to the confidence level. The lines represent the 95% confidence interval of the mean. The break in the graphs are due to missing values, which can particularly happen for S2-based signals (i.e., from presence of clouds). The header is the same as for the `signalMeans`.

25.8 Analysing signal probabilities

The plots produced by the functions above are useful step-by-step analyses, but too detailed for getting an overview of the capabilities of the different signals/descriptors. The function `signalProbability` is better for analysing a large set of signals.

First of all, it estimates the probability of the signal being above a threshold. The threshold can either be zero (i.e. the probability that the signal actually increase after an activity), or half the largest change in the weeks following the activity, as a simple threshold which could be used to detect an activity.

When we refer to increases above, the function will internally change the sign of decreasing signals to positive, so that all probabilities will be about being larger than the threshold.

Then it can produce two types of plots, either to the same device, or to two different devices.

The first type of plots are the probability rasters. The first set of these rasters will have the descriptors on the x-axis and the different indexes on the y-axis. The second set will have the weeks on the x-axis, for a particular choice of descriptors.

The probability maps, particularly the first set, give an indication of which signal and which descriptor is the most suitable for the FOIs in the sample. A set of plots are produced. First of all, the plots show different values for $p > 0$ and $p > M$, where M is a threshold value that can be used to identify an activity.

For the first set of plots with descriptors on the x-axis, there will be separate plots for each week of the `selwks` parameter. The week number is the number after “W” in the heading. If there are more than one group of activity types, these are given after “G”. If there is only one activity type, this part of the header is dropped.

The next number is the highest probability in this map. The number after “P” indicates the maximum number of FOIs/activities that have been used for this raster. The last part depends if `sortedRasters = TRUE`. If yes, the function will also generate a range of plots where the rasters are subsequently ordered according to the values of each individual descriptor/week. Particularly for the descriptors, this gives the opportunity to compare the different descriptors with each other. The sorted rasters can be identified with “S.ind” as a part of the header, followed by the sorting variable.

The second group of plots are graphs with the average value of the change together with the probabilities of the signal increasing. The function will produce four panels on each page. The upper ones show the value and the cumulative value for a particular signal, either as a function of week (only for selected descriptors), or as a function of the descriptor (for selected weeks). The lower panels will show the probability of being above a threshold of zero for the value itself, or for the cumulative sums.

The graphs might be easier to interpret if `computeAll = TRUE`. This means that the probability values will be estimated for all indexes, not only the ones that the `t.test` has estimated to have a significant change.

```
pdf(paste0(bdir, "/sigProb.pdf"))
events = unique(FOIinfo$Event_type)
evres1 = signalProbability(evress$evres, dfs, stats = stats, week0 = 7, weeks = 1:15,
  ↪ selwks = 9:13, selstats = "P50",
                        events = events, inds = inds, cases = cases)
#> [1] "Computing probabilities"
#> [1] " "
#> [1] "plotting raster probabilities"
#> [1] " "
#> [1] "plotting probability graphs"
#> [1] " "
dev.off()
#> png
#> 2
```

Figure 4 shows an example of a raster probability plot. It shows the maximum probabilities of the signal being above the threshold “M” for each index and statistical descriptor for the `selwks` period sorted according to the P50 descriptor. Indexes where none of the descriptors show a significant change are not plotted, so in this case this means that B04, s1_bs_rAD and s1_bs_VH_rAD are missing. Sorting the indexes like this, we can see that P50 seems to be good for most indexes for identifying possible activities, although “P25” is better for “BSI” in this particular case.

Figure 5 shows an example of the graph plots for different statistical descriptors for week 10 for the coherence-based signals. The two panels on the top show the signal change and the cumulative signal change. The two panels on the bottom show the probabilities of the index being above zero for the different descriptors. For this dataset, we can see that the probabilities are highest for the ordinary descriptors, i.e., the mean and the quantiles, except for min and max. Second, we can notice that there is no advantage of using cumulative values (panels to the right) for this data set. This could be different for other data.

25.9 Summarizing the results

The functions above let us analyse signals separately. `optimalSignal` is a function which can help in identifying which descriptor is in general most useful for describing the differences in signal behavior caused by an activity.

It will do this in three different ways, and only for the cases where at least one of the descriptors show a probability above 0.7 of increasing the value after the activity.

1. For how many indexes is a certain descriptor ranked as the first one, for threshold zero or “M”? The weeks are treated separately, so the number can be higher than the number of signals.
2. For how many indexes is a certain descriptor ranked as the last?
3. How good is each descriptor, relatively to the one that is ranked first for a signal? This indicates the relative goodness of a descriptor. Contrary to finding only the first ranked - where two descriptors might be almost similar, but only one included, this gives the relative goodness of each descriptor.

One plot is given for each of the three comparisons, for the two different thresholds.

The output of the function is a data.frame giving the numbers above. Each column with the type describes the three simple classes best (be), worst (bw) and good (bg) descriptors for the different thresholds (0 and T).

```
pdf(paste0(bdir, "/optimalSignal.pdf"))
dd = optimalSignal(1, inds, evres1, stats)
#> Warning: Removed 4 rows containing missing values (position_stack()).
#> Warning: Removed 7 rows containing missing values (position_stack()).
#> Warning: Removed 4 rows containing missing values (position_stack()).
#> Warning: Removed 6 rows containing missing values (position_stack()).
dev.off()
#> png
#> 2
```

Figure 6 show an example of each of the plots. The left panel shows how “P50” is the most optimal with the highest number of cases where it is ranked as number one. However, it also shows the disadvantage of only counting the highest one, as Median (same as P50) is not the first for any case. The central panel shows that the variance and the confidence interval of the mean are the poorest performing descriptors to be used in this case. The last panel rather looks at the relative goodness of the descriptor. We can here see that there is not a huge difference between the quantiles and the mean although the median is still the most optimal.

25.10 Correlations

There are many situations where it would be possible to use several signals for detection of a single activity. In this case, it is better to choose signals that are not highly correlated. The function `signalCorrelation` helps visualizing correlations between the different signals, and will also return a correlation array with the selected descriptors.

```
pdf(paste0(bdir, "/correlations2.pdf"))
sigCor = signalCorrelations(allvar, stats = c("MIN", "P50"), mar = c(3,3,2,2))
#> [1] "Creating correlation table - activity type 1 indicator: 1"
#> [1] "Creating correlation table - activity type 1 indicator: 2"
#> [1] "Creating correlation table - activity type 1 indicator: 3"
#> [1] "Creating correlation table - activity type 1 indicator: 4"
#> [1] "Creating correlation table - activity type 1 indicator: 5"
#> [1] "Creating correlation table - activity type 1 indicator: 6"
```

(continues on next page)

(continued from previous page)

```
#> [1] "Creating correlation table - activity type 1 indicator: 7"  
#> [1] "Creating correlation table - activity type 1 indicator: 8"  
#> [1] "Creating correlation table - activity type 1 indicator: 9"  
#> [1] "Creating correlation table - activity type 1 indicator: 10"  
#> [1] "done correlation 1 1"  
dev.off()  
#> png  
#> 2
```

Figure 7 shows the correlations between the different signals for the P50 descriptor.

25.11 References

Zieliński, Rafal, Jon Olav Skøien, Laura Acquafresca, and Wim Devos. Proposed workflow for optimization of land monitoring systems. JRC130659. Ispra: European Commission. 2022. https://marswiki.jrc.ec.europa.eu/wikicap/images/9/90/JRC130659_final.pdf.

CALENDAR VIEW

26.1 1. Introduction

The “**calendar_view**” python package provides a set of functions that allows one to download, process and display Sentinel-1 and Sentinel-2 data products. The package is designed to operate on “**parcels**” provided as input in a ESRI shape file. Sentinel-1 and -2 data are extracted for the specific parcels and are displayed in a “**calendar view**”, which has been designed and optimized to provide an immediate and intuitive access to both temporal and spatial dimensions of Sentinel-derived data.

While the scripts exploit the [RESTful APIs](#) developed by the JRC D5 unit to download raw Sentinel imageries tailored to the extent of the parcels provided as input, the code is general and can be easily adapted to work, for example, on geotiff downloaded from other sources.

26.2 2. Dependencies

The calendar view script depends on several standard python libraries that can be found in the *requirements.txt* file. The dependencies can be installed with:

```
pip install -r requirements.txt
```

As well the `gdal_merge.py` module that can be download from https://github.com/geobox-infrastructure/gbi-client/blob/master/app/geobox/lib/gdal_merge.py and should be added to the `utils` folder.

26.2.1 2.1 Calendar view Modules

The script can be launched from the “`calendar_view_gui.ipynb`” Jupyter notebook.

On the first tab of the GUI you can select the products that you want to produce:

What to run?	Set dates	Vector/Output folder	Other parameters
<input checked="" type="checkbox"/> Get and download SCL imageries	<input checked="" type="checkbox"/> Create NDVI imageries	<input type="checkbox"/> Get coherence imageries	
<input checked="" type="checkbox"/> Get and download band imageries	<input type="checkbox"/> Calendar view of NDVI imageries	<input type="checkbox"/> Calculate coherence statistics	
<input checked="" type="checkbox"/> Merge band imageries	<input checked="" type="checkbox"/> Calculate NDVI statistics	<input type="checkbox"/> Create coherence graphs	
<input checked="" type="checkbox"/> LUT stretch magic	<input checked="" type="checkbox"/> Create NDVI graphs	<input type="checkbox"/> Get backscatter imageries	
<input checked="" type="checkbox"/> Calendar view LUT magic	<input type="checkbox"/> Create BSI imageries	<input type="checkbox"/> Calendar view of backscatter	
<input type="checkbox"/> LUT stretch dynamic	<input type="checkbox"/> Calendar view of BSI imageries	<input type="checkbox"/> Calculate backscatter statistics	
<input type="checkbox"/> Calendar view LUT dynamic	<input type="checkbox"/> Calculate BSI statistics	<input type="checkbox"/> Create backscatter graphs	
<input type="checkbox"/> Calculate band statistics	<input type="checkbox"/> Create BSI graphs		
<input type="checkbox"/> Create band graphs	<input type="checkbox"/> Calendar view of NDVI histograms		
	<input type="checkbox"/> Calendar view of Red-NIR scatterplot		
Select minimum		Select all	

Run

On the second tab of the GUI you can set the date range for data download and visualisation:

What to run?	Set dates	Vector/Output folder	Other parameters
Search window start date	2017 . 10 . 01	Index graph start date	2017 . 10 . 01
Search window end date	2018 . 12 . 31	Index graph end date	2018 . 12 . 31

Run

On the third tab of the GUI you can define the ESRI shapefile with the parcel polygons (“Vector filename”), you can select the folder for the outputs produced by the script (“Base folder outputs”), the attribute table column holding the parcel ids (“Parcel id column”) and the attribute table column holding the crop names (“Crop name column”):

What to run?	Set dates	Vector/Output folder	Other parameters
Select	No file selected		
Vector filename:	C:\Users\Csaba\ownCloud\GTCAP\Outreach_2021\MS_directry\NL\mowing_detection\vector\parcels_2018_m.shp		
Select	No file selected		
Base folder for outputs:	C:\Users\Csaba\ownCloud\GTCAP\Outreach_2021\MS_directry\NL\mowing_detection\chips\2018_for_webinar		
Parcel id column:	ogc_fid (eg.1)	Crop name column:	crop_name (eg.Grassland)
Parcel ids	<div> 1 2 3 4 5 6 7 8 9 10 </div>		

Run

On the forth tab of the GUI you can define all additional parameters, such as the title for the time series graphs, whether to include or exclude the cirrus cloud mask from cloud cover calculations, the buffer size around the parcel for chip extract (in meters) and the potential shift of the parcel centroid in degrees. (This latter can be useful when you want to

make sure the image chips are re-generated and not read from the cash):

What to run?	Set dates	Vector/Output folder	Other parameters
Title for graphs: <input type="text" value="NL 2018"/>			
<input checked="" type="checkbox"/> Exclude cirrus from stats calculation			
Buffer size around parcel: <input type="text" value="50"/>			
Centroid shift (degrees): <input type="text" value="0,00001"/>			
<input type="button" value="Run"/>			

The following custom modules for the `calendar_view` package are called by the `run_calendar_view_from_jupyter.py` script:

- **batch_utils**: provides general functions for the processing of parcels and Sentinel data. The set of functions provided in this module allows the selection of parcels, the determination of the list of imageries to be downloaded on the basis of cloud mask criteria, the download of imageries, processing of Sentinel data including image color stretching (lookup table stretch or LUT stretch) and computation of derived indexes such as NDVI and NDWI. For download operations, the module uses the functions developed under `download_utils`.
- **download_utils**: module for downloading and processing Sentinel-1 and 2 imageries.
- **plot_utils**: functions for plotting Sentinel-1 and 2 imageries in structured calendar views. Sentinel products are displayed in a coherent way, in order to provide an enhanced accessibility to both temporal and spatial dimensions of Sentinel-1 and 2 products.
- **graph_utils**: module providing functions for displaying and plotting temporal profiles such as NDVI and NDWI. The module focuses on temporal (1-D) signals and complements the utilities in `plot_utils`, which focuses on displaying images (set of 2-D signals).
- **extract_utils**: module for the calculation of the NDVI and NDWI indexes. The functions in this module are called by `batch_utils`.

The call graph of the `run_calendar_view` script is provided in the figure below.

26.3 3 Structure of the code

The `run_calendar_view` script is organized in two main parts: the initialization block and the main processing loop. The input parameters and processing settings are defined in the initialization block whereas the actual processing is performed in the main loop that performs the different tasks by considering individual parcels. The main processing loop is further divided in two parts: download and processing of Sentinel-2 data and download and processing of Sentinel-1 data. The overall structure of the script and the different tasks performed by the main processing loop are illustrated in Figure 2. The different parts of the script are better detailed in the following sections.

Overall structure of the `run_calendar_view` script.

```
run_calendar_view
```

Initialization:

- Authentication and RESTful API access
- Cloud masking settings
- Parcel data input (shape file)

- Band selection
- Time interval selection
- Output image properties

Main processing loop *For each parcel:*

- **Sentinel-2 processing**
 - get and download SCL imageries
 - create a list tiles to be downloaded (based on cloud cover)
 - get and download band imageries
 - merge bands and apply stretching
 - create calendar views
 - generate derived products: NDVI profiles, histograms, Red-NIR scatter plots, ...
- **Sentinel-1 processing**
 - get and download Sentinel-1 backscattering imageries
 - rescale and stretch imageries
 - for each polarization (VV and VH) and for the two orientations (D and A)
 - * provide calendar views
 - * compute statistics
 - plot joint profiles

26.4 3.1 Initialization

The variables in the initialization part of the script allows one to define inputs and outputs and to customize the different operations performed by the script.

Authentication and data access

These scripts are using RESTful API for CbM. See the [documentation on how to “Build RESTful API with Flask for CbM”](#). To access the RESTful services it is necessary to provide login information. In this respect, the username, password and the data url (`url_base` variable) have to be properly configured. Alternatively JRC’s RESTful API for CbM provides limited sample open datasets and can be used for testing and demonstration purposes. A temporary account can be requested from JRC GTCAP group.

Cloud categories

Different SCL classes can be specified in the `cloud_categories` list. These classes, mainly specifying cloud types, are used to filter out Sentinel-2 images affected by cloud covers within the parcel.

Parcel data

The parcel data including polygons, parcel IDs and crop types, are specified through the `vector_file_name` variable. This variable should point to the ESRI shape file containing all the needed information. The variables `parcel_id_column` and `crop_name_column` specify the columns in the shape file containing the parcel IDs and crop names. This information will be printed on the images generated by the script. In case crop name is not available add any other information in that column (can be even an empty string) you want to be printed on the outputs.

Sentinel-2 imageries

The bands and the maximum size of the Sentinel-2 imageries are specified through the *bands* and *chipsize* variables. Default chip size is replaced by the maximum extent of the parcel plus the buffer size provided in *buffer_size_meter* variable.

Date range The date range for the data search can be set through the *search_window_start_date* and *search_window_end_date* variables.

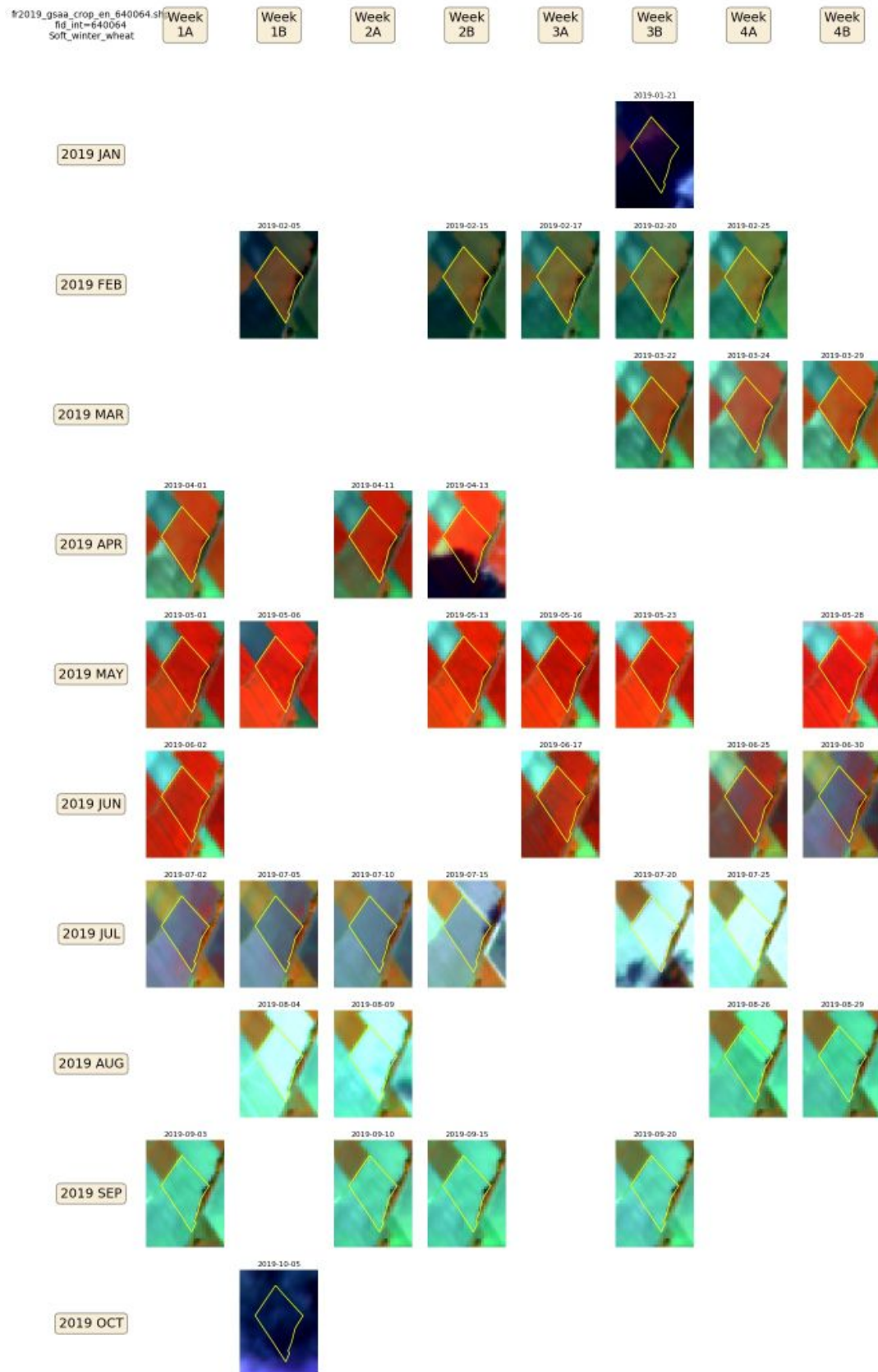
26.5 3.2 Main Processing Loop

The different operations performed in the main loop are listed in the second part of the “Overall structure of the *run_calendar_view* script” section above. These operations are performed through the functions called in Figure 1. While the *run_calendar_view* script demonstrates most of the functionalities implemented in the different libraries, the main loop has a modular structure and functions can be commented if specific operations are not needed.

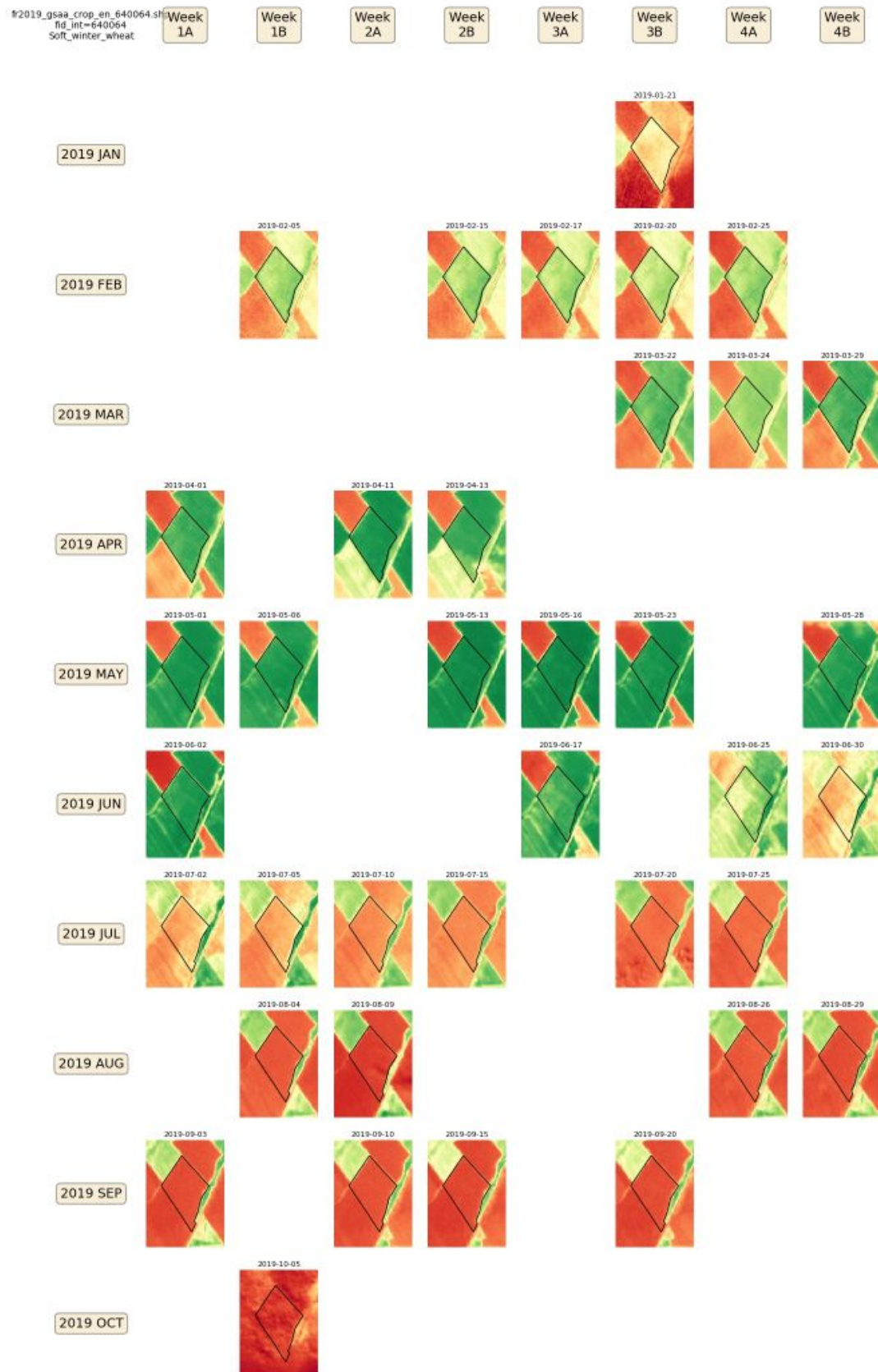
26.6 3.3 Output examples

Calendar view of Sentinel-2 imageries

False Colour Composite (FCC), 8,11,4 RGB, LUT stretched (generic)

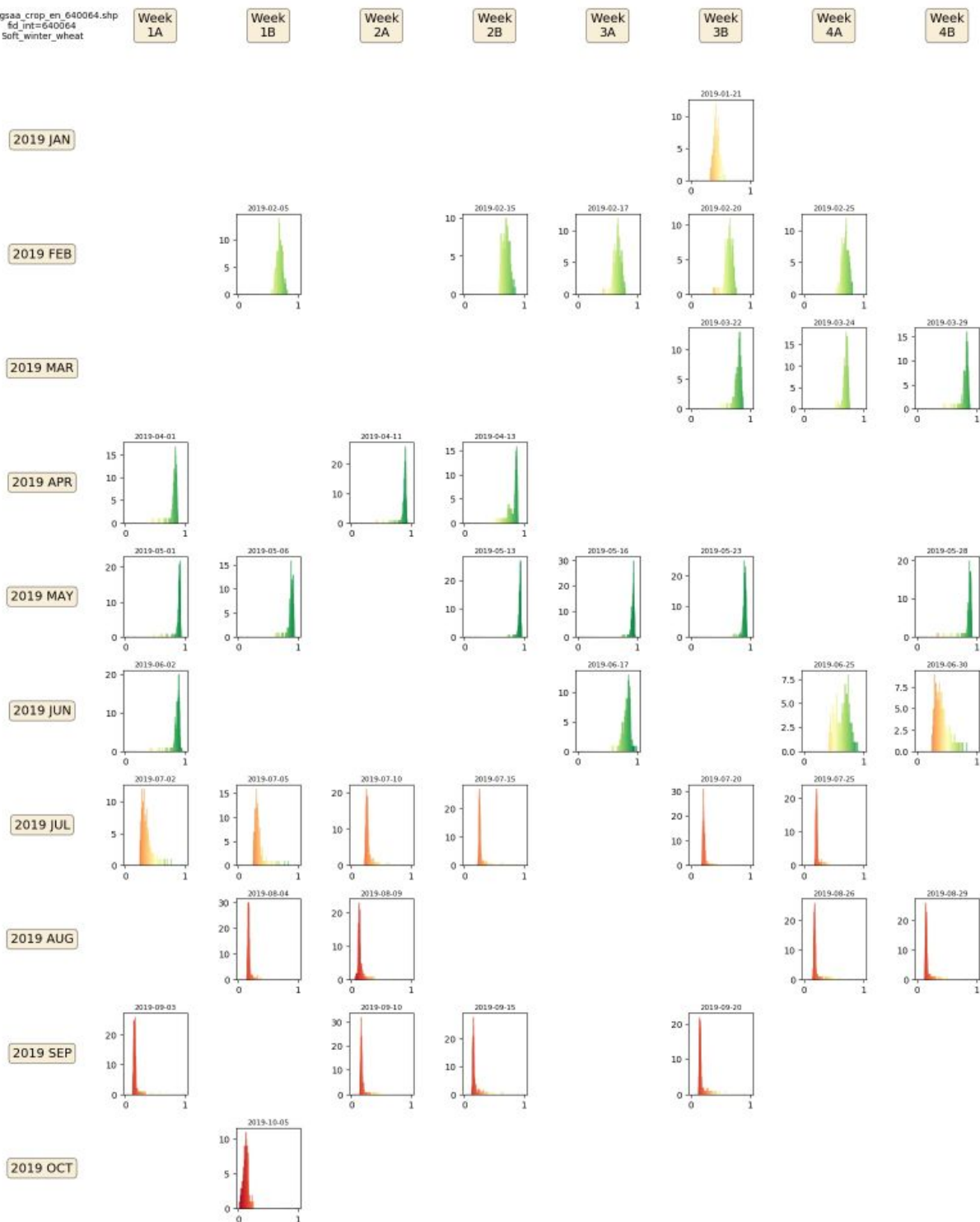


Calendar view of Sentinel-2 NDVI imageries



Calendar view of Sentinel-2 NDVI histograms

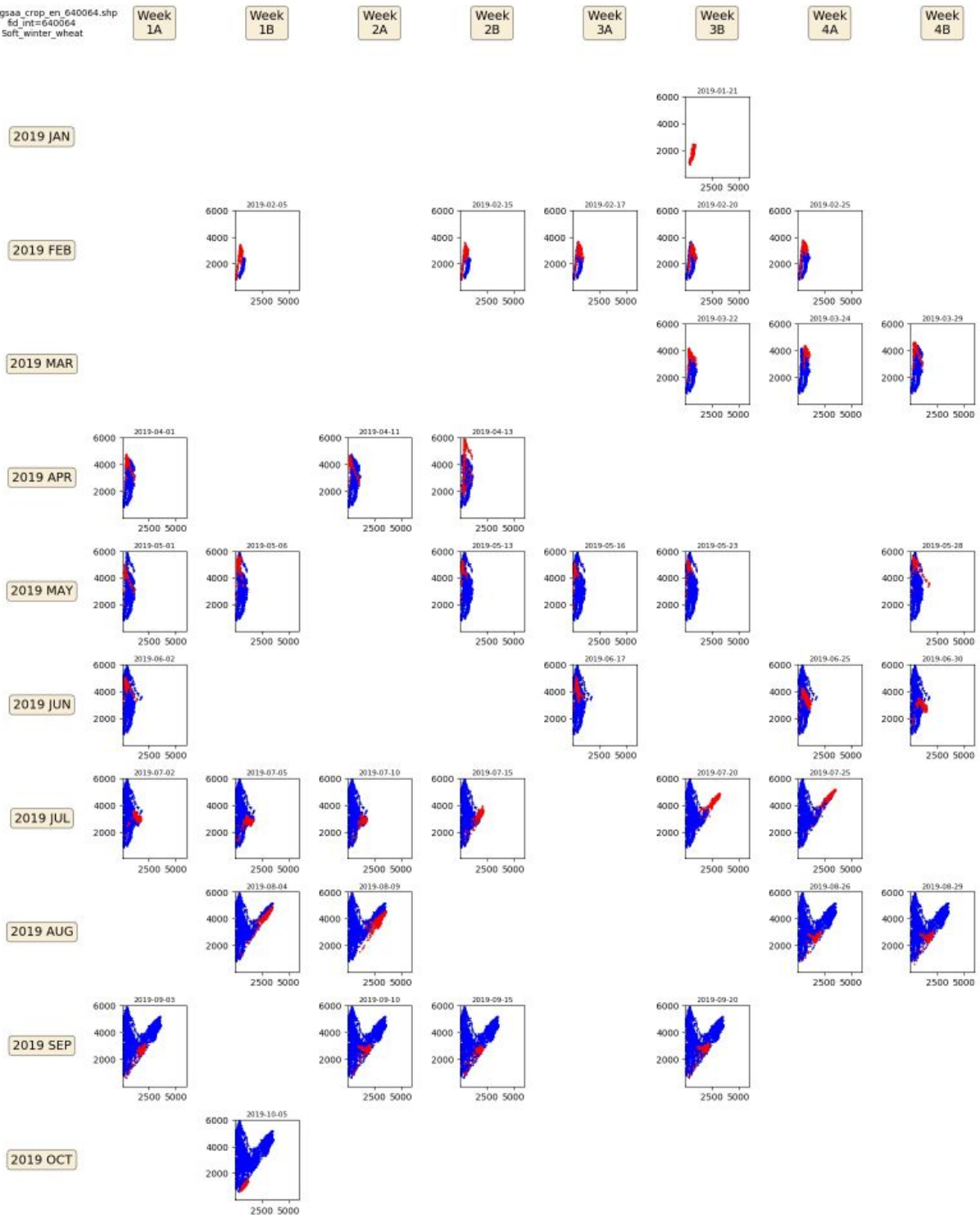
fr2019_graa_crop_en_640064.shp
 fd_int=640064
 Soft_winter_wheat



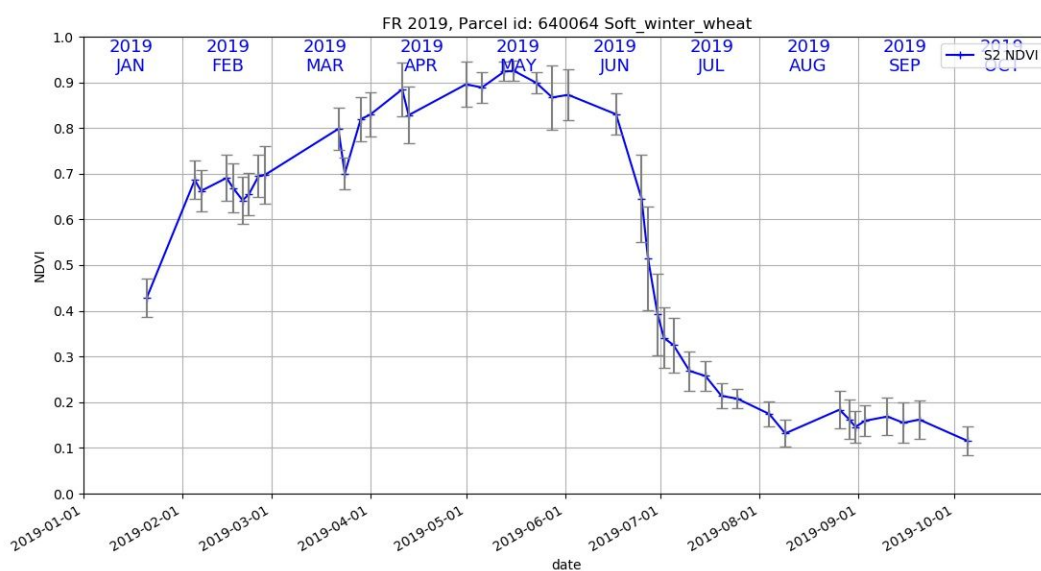
Calendar view of Sentinel-2 cumulative scatterplots

Cumulative scatter plot of Red (horizontal axis) and NIR (vertical axis) bands within the parcel Red dots: scatter plot of current date Blue dots: scatter plot of all previous dates

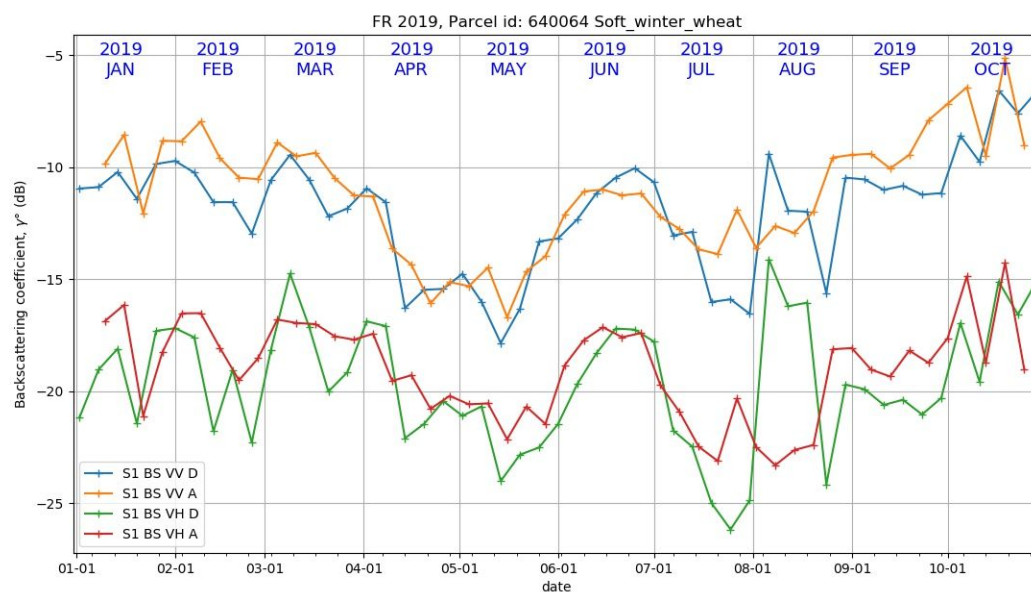
fr2019_gsa_crop_en_640064.shp
fd_int=640064
Soft_winter_wheat



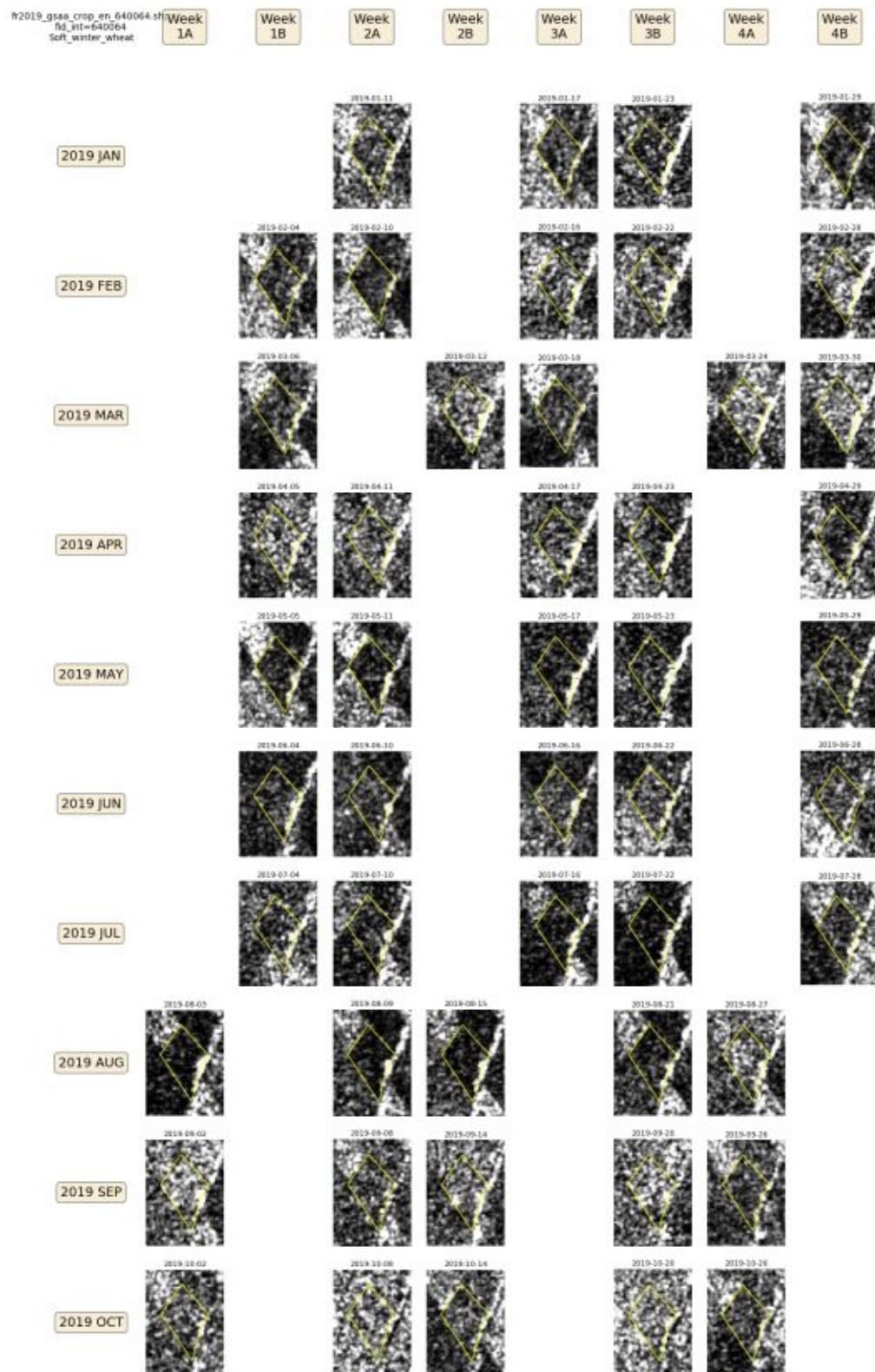
Graph of Sentinel-2 NDVI values



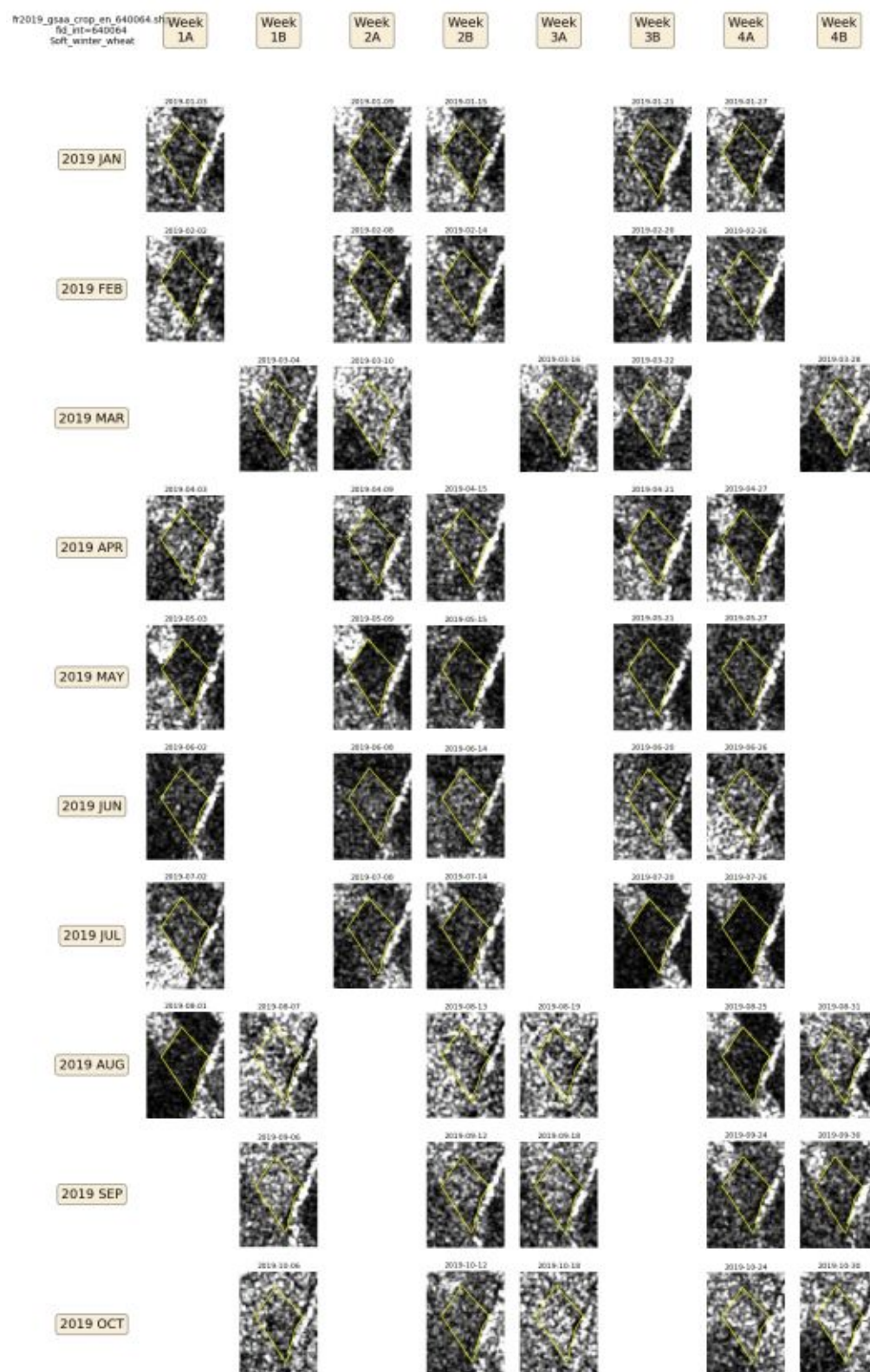
Graph of Sentinel-1 backscatter values



Calendar view of Sentinel-1 backscatter imageries, VH polarisation, Ascending orbit



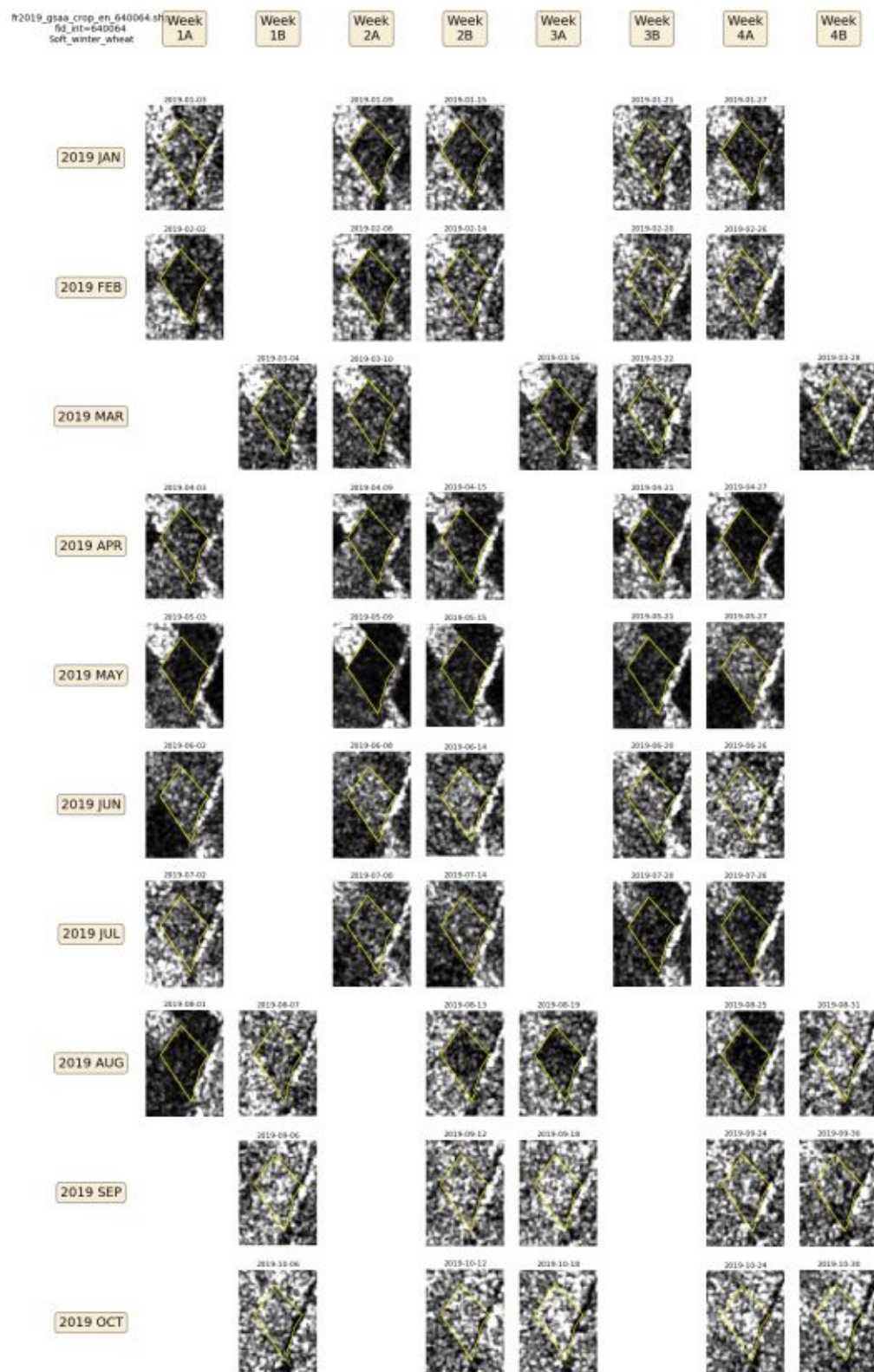
Calendar view of Sentinel-1 backscatter imaggates, VH polarisation, Descending orbit



Calendar view of Sentinel-1 backscatter imaggates, VV polarisation, Ascending orbit



Calendar view of Sentinel-1 backscatter imaggates, VV polarisation, Descending orbit



FOI ASSESSMENT

27.1 Concept

The spatial dimension and Feature Of Interest

Checks by Monitoring (CbM) introduced the concept Feature of Interest (FOI) to deal with the spatial aspect of the bio-physical phenomenon present on Earth. FOI is in fact the “space” occupied by the observable physical object on the ground, the “field”, and its spatial “footprint”. In the EU CAP context, it often coincides with the single unit of agricultural management (single crop or particular land use, on homogeneous agricultural land cover).

In the CbM system, the FOI has two spatial representations (features). The first is derived from the Geo-Spatial Aid Application (GSAA) and uses a polygon as geometric primitive. The second is derived from the Sentinel data and can be expressed in different “formats” - statistical metric, clusters of image pixels, image segments. Both representations serve to represent that single true physical object or FOI.

The information collected from Sentinel on a given FOI could reveal the presence or persistence of “things” of different nature inside. This could shed light on the characteristics of those bio-physical phenomena and their spatial distribution. It could also indicate whether the FOI representation from GSAA reflect the true physical object behind.

Spatial heterogeneity within the FOI representation detected by Sentinel signals, could relate to three cases:

- Inherent and expected variations within the physical entity: A typical example is the pro-rata grassland where the herbaceous and woody land forms co-exist and form a stable “intrinsic mix”.
- Alien physical entities present in the same unit of management: A typical example is the presence of an object of non-agricultural nature (for example buildings) within the FOI representation.
- Several physical entities corresponding to different units of management: A typical example is the presence of arable land and permanent grassland within the FOI representation.

The latter two require particular attention, since they reject the initial hypothesis that:

1. the FOI representation from the GSAA relates to one and only one physical object,
2. this object is of agricultural origin (land cover);
3. and it is spatially congruent to both FOI representation (GSAA and Sentinel).

The validity of this initial hypothesis is an important boundary condition of the CbM, as it ensures that:

1. the area component - officially known hectares of agricultural land cover - provided by IACS, is correct
2. data derived from Sentinel is exclusively associated to the monitored physical object, which guarantees meaningful analysis and CbM decisions

The truthfulness of the FOI representation in relation to the actual FOI depends on the system design and quality of LPIS/GSAA datasets; the closer the representation to the actual FOI, the better the performance of any processing in CbM.

The importance of the spatial heterogeneity in the FOI and the spatial congruency of the FOI representations to the physical object on the ground, has triggered the design a special set of methods to detect and eventually quantify the spatial heterogeneity and congruency for CbM purposes. Since the spatial congruency of the FOI representation with the actual FOI is manifested through “one to one” cardinality between the representation and the true phenomenon, the term applied in the CbM is “spatial cardinality”, expressed with code G1. It is the first spatial type of information extraction in the CbM; a second, generic type of information extraction dealing with FOI heterogeneity, is called “spatial variability” and expressed with code G2.

Once integrated within the CbM, both G1 and G2 are expected to provide key information in the decision process. A confirmation of the “one to one” spatial cardinality (validity of the initial hypothesis) will confirm suitability of any given GSAA associated with the payments schemes as FOI representation for the CbM process. A rejection of the “one to one” spatial cardinality will require either

1. interaction with the farmer, when the GSAA-FOI cardinality is “one to many”; or
2. spatial aggregation process when GSAA-FOI cardinality is “many to one”.

The “spatial variability”, G2, in other hand, confirms the anticipated presence of heterogeneous properties within the physical object (pro-rata grassland, presence of landscape features) or the detection of the presence of multiple land uses within FOI, typical for the scenarios, related to greening payment scheme.

Figure 3: Illustration of the “spatial congruency” in the context of CbM

27.2 Methods and algorithms

GTCAP launched a series of activities to develop and test different methods for the detection of the spatial cardinality and spatial variability within the FOI representation. There were three phases:

1. Elaborate several methods using different Sentinel signals, at the level of Proof of Concept;
2. Test the performance of the developed prototypes;
3. Select and compile the most successful methods in python scripts.

A tiered-approach for the development of the method was established, with the aim to cover both spatial cardinality (G1) and spatial variability (G2); with the former considered to be a special case of the latter. For spatial cardinality G1, there were two sub-cases identified depending on the severity of the spatial mismatch between the FOI representations. The approach comprised the following three steps (tiers):

1. Step 1: Search for “different things” within the FOI representation from GSAA – spatial variability (G1)
2. Step 2: Search for “different physical entities” that invalidate scenario – spatial congruency (G1) not respected to a degree that it jeopardizes the Sentinel signal with a given scenario
3. Step 3: Search for “alien entities” big enough to invalidate the area – spatial congruency (G1) not respected to a degree it jeopardizes the area component as provided by IACS.

The implementation of step 3 certainly required a numeral for the size of the different physical entities within the FOI representation, beyond which the area component would be jeopardized. Based on earlier technical research and following the logic applied in the LPIS update, it was agreed that occurrence of continuous clusters with more than 20 pixels within the FOI representation indicate the presence of physical objects big enough to challenge the area component provided by IACS. The pixel size and thus final area depends on sensor used. More information on the subject is provided in a separate technical report.

For the G1 method, further assumptions have been made on the detection of “one to many” cardinality.

- persistence on the ground for sufficient time to invalidate the given scenario.
- manifestation through the presence of entities having different behaviour in time.
- distinct behaviour can be captured by the Sentinel signal.

- These different entities are sufficiently large in image cluster size to challenge the area component.

Although the G1 methods were designed to detect issues with the spatial congruency, the ultimate outcome was a confirmation of the validity of the FOI with respect to:

1. the “officially known area” and the type of agricultural land cover recorded in IACS;
2. correspondence of graphical representation of the FOI (one-to-one spatial match with reality or correct portion of larger unit)

In phase 1, five methods were developed and four were tested:

- Analysis of S1 backscattering’s speckle noise: Takes into consideration that in homogeneous fields the SAR backscattering speckle is following a Gamma distribution
- Threshold on S2 signal-to-noise ratio: Uses the ratio between the observed NDVI average and observed NDVI standard deviation
- Histogram analysis of S2 signal: Monitors the difference in 25 and 75 percentile values relative to the mean of the histogram
- Unsupervised clustering through S2 image segmentation: Looks for clusters of pixels (grouped in segments) with distinct behaviour in time
- Multi-temporal S2 supervised classification: Assesses the land cover types found within the FOI representation derived from pixel-based supervised machine learning

More information on these methods is documented in a separate technical report in progress.

27.3 Test results

In phase 2, all methods were run on a pre-defined test area in Catalonia (Spain) with GSAA data from 2019, where relevant Sentinel CARD products (Copernicus Application Ready Data) were available for both Sentinel 1 (S1) and Sentinel 2 (S2), with sufficient temporal density.

The setup for evaluating the results was based on the conceptual elements, incorporated later on in the Technical Guidance on QA CbM (v.1.0). 365 randomly selected FOI representations from 2019 GSAA were visually checked (with optical Sentinel data) for presence of spatial variability and the spatial cardinality issues over time, and at regular intervals, depending on the land cover type given in the GSAA. These checks were made before running the methods and can be considered completely “blind” (not biased by the outcomes from the different methods).

The comparison of the results from the methods with the collected ground truth provided a diverse picture, with no standalone method scoring sufficiently well for both commission and omission errors. Methods seem to be complementary rather than competing, each revealing different aspects of the observed phenomena.

The particular LPIS design (based on cadastral parcel) and the rugged landscape in Catalonia was another important factor that adversely influenced the performance of the methods. Detection performance varied also depending on the type of observed agricultural land cover. Yet, the clustering methods seemed to score better in comparison with the two statistical methods.

FOI methods	Expectations			
	Detection performance	Sentinel data (other data needs) needs	Complexity of method	Portability to JRC DIAS notebooks
S1 Gamma	Not promising (unsuitable for rough landscape)	S1 GRD	Simple	Easy
Statistical methods				
S2 S/N	Explore further (other metrics – DIP test)	S2 L2A	Simple	Easy
S2 segmentation				
Clustering methods	Promising (need fine-tuning of segmentation technique, spatial context rules)	S2 L2A	Moderate	Difficult
S2 supervised ML				
	Promising (need more complete training data)	S2 L2A Training data	Complex	Moderate

Figure 4: Overview table with the methods tested (the histogram analysis method is missing, since it was not subject to this evaluation)

The results also provided useful insights of certain methodological and technical challenges related to the (weather) cloud cover filtering, the appropriateness of the statistical methods, the parameters of the segmentation process, the completeness of the training data and the need for introduction of elevation data (DTM, DSM).

A promising discovery from the quality assessment was the fact that the majority of the observed “FOI inconsistencies” were visible already by the month of May, which is still within the GSAA application period. This suggests that both G1 and G2 detections could and probably should be incorporated as supporting tools during the declaration process to help farmers improve their GSAAAs.

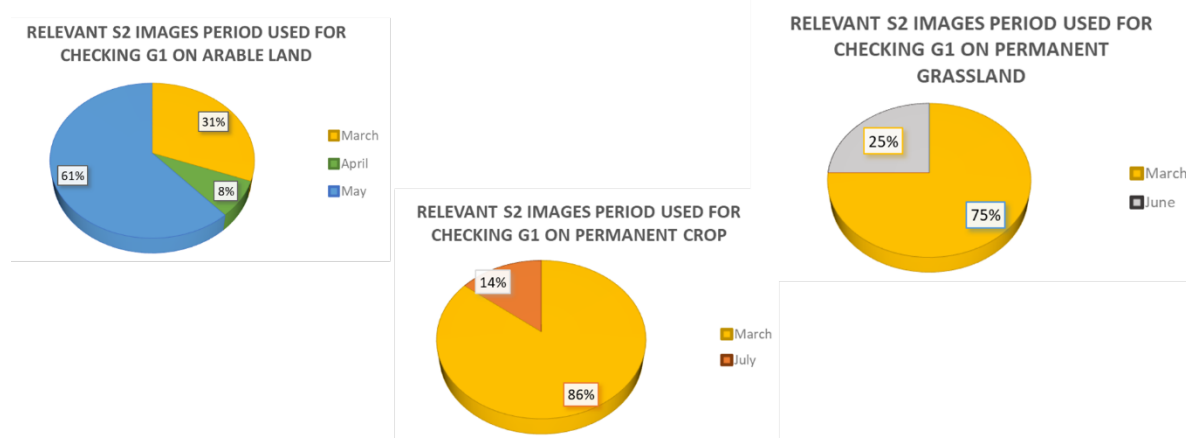


Figure 5: Relevant period for the detection of issues related to spatial cardinality (G1), as collected for the validation dataset in Catalonia (Spain)

27.4 Cardinality

Towards a spatial cardinality, G1 code example

The main conclusion of phase 2 was that clustering methods were more promising and, in fact, essential for the detection of issues in relation to FOI spatial cardinality. However, both the segmentation and supervised classification needed additional tuning. GTCAP decided to invest, in phase 3, in the development of generic “best practice” python scripts for the detection of clusters within FOI representation, provided by any thematic raster. A prototype of such python module already existed as part of the supervised machine learning method.

This resulted in the so-called “FOI assessment python module” that combined these methodological elements:

- Cluster assessment within FOI relies completely on information provided by thematic raster
- Focussed exclusively on detecting, whether GSAA > FOI (GSAA split case)
- Assumption that the CbM signal alone is used to capture distinct cluster behaviour
- Deals with GSAA FOIs having cluster representation of at least 40 pixels
- Does not (yet) apply negative buffer prior to cluster assessment
- Customizable to address both G1 and G2 aspects

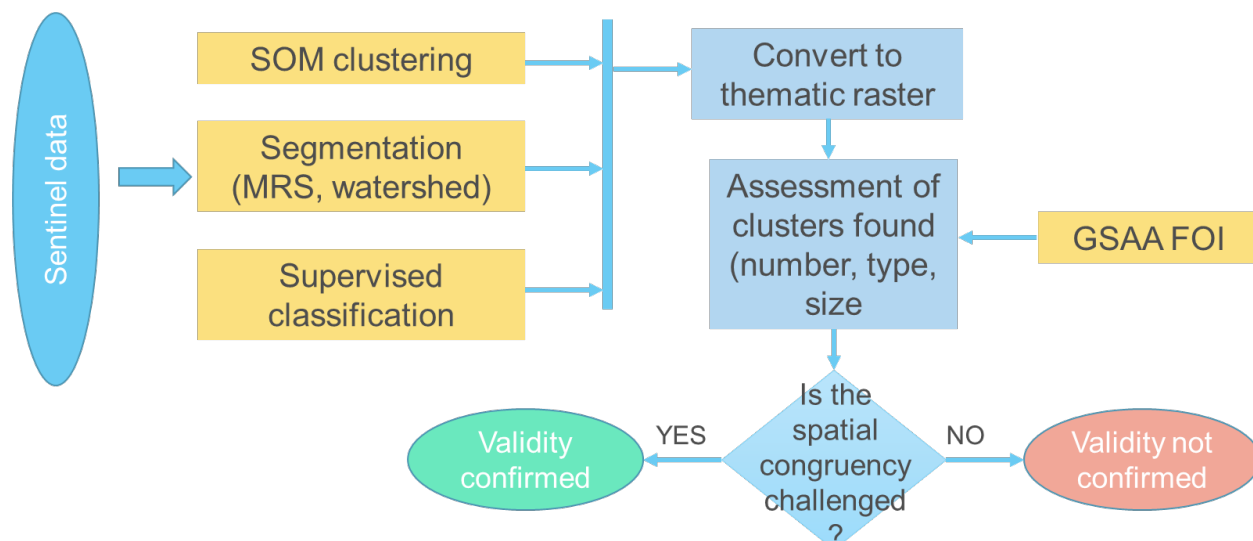


Figure 6: Workflow of the FOI assessment python module

The cluster assessment python module is being tested with thematic raster data generated from the unsupervised image segmentation method (both in Catalonia, ES, and Belgium-Flanders). Although having relatively high commission errors for the spatial cardinality, this segmentation method proved to add value by detecting variabilities within the FOI, such as terraces, landscape features and shrub encroachment within natural grasslands – all very relevant in the context of compliance scenarios that cover CAP greening aspects.

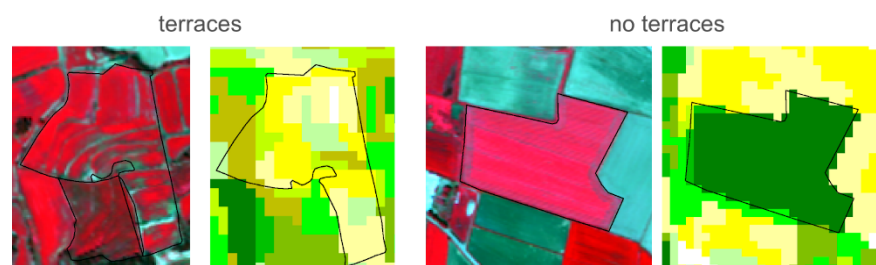


Figure 7: Parcels with terraces results with more images segments than those without (JRC Study case in Catalonia, ES)

The python module also supports an “enhanced” version of the “histogram analysis” method. It monitors the pixel value distribution of the NIR and RED bands of Sentinel-2 and compares them to a uniform or normal distribution. Although not yet formally evaluated, some initial trials evidence its ability to detect heterogeneous FOI representations. The algorithm is fully automated, easy to deploy, and generates both indices and image chip highlights.

27.5 FOI Clustering

MACHINE LEARNING

Machine learning with tensorflow in 8 easy notebooks.

28.1 Introduction

The core data set in CAP Checks by Monitoring (CbM) is the annual parcel declaration set compiled from the beneficiaries' registration process. A pre-condition for CbM is that the Member State has a system/systems in place that allow(s) such declarations to be specific for the agricultural parcel (the management unit) and the specific activities that comply with the scheme under which the declaration is made. The obvious benefit of continuous, territory-wide Copernicus Sentinel data use is that we can now cross check the information given in the declarations, by generating relevant markers that confirm compliance.

Given the low overall "material errors" in the annual parcel declaration sets in most Member States, this makes the set an ideal source of information in machine learning. The way we apply machine learning is complementary to classification, and a much better choice indeed. After all, what is the contribution of a classification with an overall accuracy of 90% (at the end of the season), if the parcel declaration set is typically 95-98% correct. What we're interested in is identifying the small percentage of parcels that are not confirmed (i.e. in the 2-5% range), and understand the reasons. Let's leave sorting out classification confusion to the scientists.

A key factor for machine learning is that it needs a regular and consistent sample of observations, both for the learning (or training) stage and the prediction (or testing) stage. This is why Sentinel-1 time series are a prime candidate, because they are regular, consistent (calibrated, no cloud issues) and with a density (at least every 1-3 day for European latitudes) that befits the dynamics of the agricultural growing season. The regularity of Sentinel-1 also applies across several seasons, whereas it would be very difficult to rely on consistent Sentinel-2 time series from one season to the next (esp. at mid to Northern latitudes). We can still use Sentinel-2 after machine learning, e.g. to explain and confirm why certain outliers were detected.

28.2 Data preparation

We choose to apply machine learning to the parcel averages, which were extracted in a previous step. This is not really a pre-requisite for machine learning, which can be applied to pixel time stacks, if needed. However, the sheer size of the data volume to process would be prohibitive and, for Sentinel-1, the results would be strongly affected by speckle.

28.2.1 Determine which classes to include

A typical annual parcel declaration set may include several 100s of different crop types. However, only a small subset of those are cultivated on 90-95% of the overall surface. Machine learning will typically allow the sorting of classes that are distinct in some specific way. A certain robustness is required so that not all minor deviations are assigned to distinct classes. This is achieved by providing a sufficiently large sample of features for “typical” crop types, i.e. those that have at least 1% in the selected area.

We typically try to group crop types that are expected to show similar radiometric behavior over the season. This may include cases like “maize” and “sweet maize”, “seed potatoes” and “consumption potatoes”, which are expected to have roughly the same crop season and phenological behavior. This is a practical choice, and may not be applied if it is of interest to keep exactly those classes separate. In fact, machine learning can be easily tailored, for instance, by eliminating confirmed classes and outliers from a first run with group crop types, and then refine in a second run. In our example, we focus on separating the main crop types.

Small parcels are obviously undesired, as they will pollute the training set with mixed feature extractions. In our examples, we eliminate all parcels < 0.3 ha. However, there would be no issue re-inserting small parcels in the testing set, in order to get a first order estimate of their predicted classes.

Machine learning requires many samples to do proper training. Thus, we recommend to select areas in which at least 100,000 parcels are present in the sample. More is better. For very large territories, it may be a good approach to subdivide in large regions that have some agronomic commonality (e.g. similar distribution of the major crop types). An overall precondition is that a sufficient number of samples for the selected classes is present in the sample.

– insert notebook here –

28.2.2 Composing the features for machine learning

While Sentinel-1 observations are (very) frequent, they are not regularly sampled. Most EU locations are covered by both ASCending and DESCending orbits and often by more than one overlapping orbit (the more so towards Northern latitudes). We choose to sample these observations over a regular 5 (e.g. North of 50 degrees latitude) or 7 day interval, so that at least 3 observations are averaged.

– insert notebook here –

The regularly sampled means now be organised in a feature vector and be exported for use in follow up routines. At this stage, it should be determined what part of the growing season is most likely to contain the most distinctive features that will help separate the crop types that were identified in the previous step. In our example, we use 5-day means from period 11 (end of February) till period 54 (end of October).

– insert notebook here –

The feature set must be matched with unique crop labels (defined as a sequence of integers starting at 0).

– insert notebook here –

28.2.3 Splitting in training and testing sets

The data set for the entire area of interest is now split into training and testing sets. We choose to select random subsets of N% for training with their complementary (100-N)% part for testing. N is typically 20 or 25 percent. In this way, we can execute 100/N separate machine learning runs and combine the results. Each parcel will be labelled with N-1 predictions this way (it is used as training sample in 1 case). This is a practical choice. Scientists prefer to select 80% for training and use 20% for testing, which seems to be somewhat skewed. In fact, training accuracy does not really improve much after a certain percentage of samples is used. We prefer to check the consistency between multiple runs as an indicator for outliers.

– insert notebook here –

28.3 Running the training and testing

We're now set to run the actual machine learning step. We use tensorflow with the tflearn module. The step consists in reading in the training data and prepare it for the training step. The machine learning model is chosen to be a deep neural network with 2 layers of 32 fully connected nodes. The training step is then run for 50 epochs using the softmax criterion for convergence and interactive review of the accuracy assessment. In most cases, convergence happens well before the end of the 50 epochs.

The trained model is then used to predict the class probabilities for each of the testing sample. These are then stored to disk.

– insert notebook here –

The tensorflow model is run for each of the 100/N runs to produce 100/N distinct predictions.

28.4 Checking the results

28.4.1 Confusion matrix for testing sets

28.4.2 Combining different tensorflow runs

28.4.3 Identifying outliers

28.5 Next steps

OVERVIEW OF THE MARKER DETECTION TOOL

DOCUMENTATION IN PROGRESS

- GOALS
- TARGET USERS
- SKILLS NEEDED

Identification of agricultural practices is a key step for the CbM and for the future area monitoring system (AMS). Several algorithms based on Copernicus Sentinel-1 and Sentinel-2 satellites have already been published in the literature. However, the application of these methods for operational monitoring requires tools that can efficiently analyse the large volume and streams of Sentinel data. In addition, for some practices proper methods has still to be defined and tuned on specific conditions of the territory assessed. This requires a platform where analysts can test their hypothesis, measures the uncertainty and consolidate the methods. In this context we developed a processing framework to detect markers, defined as an observation of a relevant bio-physical change of state, made on the Sentinel signal, on a given date. The markers can then be associated to agricultural activities on a parcel and used to confirm/reject the farmer declaration of a CAP scheme. This framework is built using a set of integrated Python modules and Object Oriented Programming (OOP) to facilitate its potential extension with new modules and algorithms designed to detect specific land practices. The JRC CbM is designed to reduce the data volume from Sentinel satellites and transform it into manageable information. Following this logic, the marker analysis is based on information extracted in the backend from satellite bands for each homogeneous land management unit (declared parcel), moving from pixel to object analysis. The time series band statistics (mean, standard deviation, min, max, median, quantiles) and the histogram derived from the Scene Classification Layer (SCL) for Sentinel-2, and statistics of backscattering and 6-day coherence for Sentinel-1, are then further reduced to a set of detected markers. The processing architecture is built on four main modules: 1) import of band statistics, 2) data preprocessing, 3) identification of relevant changes in the signal time series (marker), 4) aggregation of markers from different sensors and association with land management practice, through the relevant bio-physical stages. The first module loads the object-based statistics generated by the backend and formats them according to the analytical requirements. Different sources can be set as data source, including direct database access, flat files (csv for statistics and shapefile/geojson for spatial units) and RestFul Application Programming Interfaces (APIs). The second module offers a set of preprocessing tools to deal with incomplete and irregular time series (i.e., resampling, interpolation, smoothing, signal combination, cloudy observation removal, noise reduction). These signal processing blocks can be chained to allow the implementation of complex processing schemes before actual marker detection. Several processing chains can be allocated in parallel to simultaneously process Sentinel-1 and Sentinel-2 data. In addition, the code architecture of this framework offers the possibility to improve the overall computational performance of each module by exploiting multi-threading to process observations from several spatial objects in parallel. The third module analyses the temporal profiles of the relevant signals to detect markers resulting from land management practices. For example, aboveground biomass reduction associated with a mowing activity is expected to result in a drop in NDVI. Similarly, an increase in coherence is also expected. In this respect, several algorithms (detectors) able to identify, in a given time series, a specific feature corresponding to a marker have been implemented based on minima and maxima values of band statistics. For each marker, parameters such as duration, amplitude and period of occurrence are determined. These parameters can be used to exclude false positives and improve detection performance and fine tune the identification of relevant variations according to the specific environmental and climatic context and the practices monitored. Several graphical outputs help users to explore the results and build the most appropriate model. In the

last module, the results from the marker detection run on all the specific bands and derived indices are combined to identify the relevant stages of the land management practices under assessment and obtain a more reliable detection. This approach is demonstrated for mowing detection using NDVI from Sentinel-2 and coherence from Sentinel-1. If an NDVI drop and a coherence increase are found on close time intervals, they can be associated leading to a more reliable activity detection. In a similar way, coherence can be used to compensate for data gaps in the NDVI time series. The framework proposed demonstrates how to fully exploit the complementary nature of Sentinel-1 and Sentinel-2 observations. It is constantly under development and new features are frequently added.

MODULES DESCRIPTION

DOCUMENTATION IN PROGRESS

- Detailed description of module 1
- Detailed description of module 2
- Detailed description of module 3
- Detailed description of module 4

OPTION FILE

DOCUMENTATION IN PROGRESS

- What is an option file
- Settings
- Examples
- Diagrams

GRAPHICAL USER INTERFACE

DOCUMENTATION IN PROGRESS

- Description
- How to

EXAMPLES OF MARKER DETECTION

DOCUMENTATION IN PROGRESS

- List of examples with short description and link to Jupyter notebooks

OVERVIEW

The cbm Python library provides an easy and organized way to run a variety of different tasks for checks by monitoring.

Python library for Checks by Monitoring, includes:

- card2db : Transfer metadata from the DIAS catalog (direct access required)
- extract : Parcel extraction routines (direct access required)
- get : To download data locally
- show : to show plots
- foi : FOI analysis module
- report : For reports creation
- ipycbm : Interactive notebook tools, includes:
 - ext : Graphical interactive notebook widget for extraction procedures.
 - foi : Graphical interactive notebook widget for FOI analysis.
 - get : Graphical interactive notebook widget to download data.
 - qa : Graphical interactive notebook widget for Quality Assessment (QA) procedures.
 - view : Graphical interactive notebook widget to view data (graphs, images).

The first time the cbm library is imported it will create:

- config/main.json # The main configuration file
- temp # Folder to store all the intermediate data.
- data # Folder to store the user data

All data can be stored in the temporary folder 'temp' or the 'data' folder. The difference is that every time the notebook is started, it will check if there is old data in the temporary folder and ask to delete them.

There are two methods to get parcel data, one is with the use of a RESTful API and the other with direct access to the database and object storage, RESTful API is the preferred method to retrieve and view the parcels data. To get data from a RESTful API a relevant server is needed see [build a RESTful API server](#). To run the extract functions direct access is required.

34.1 Notebook widgets

A subpackage ‘ipycbm’ is available for use in Jupyter Notebooks and provides interactive graphical configuration panels and data visualization tools for Checks by Monitoring.

Main functions

Panels	Description	Use
config()	To configure the config/main.json file interactively	D,R
get()	Get data from servers with different methods (coordinates, parcels ids, map marker, polygon*)	D,R
show()	View the data with many different ways**, with easy selection of the view method.	D,R
extract()	For running extraction routines and other cbm tasks	D
foi()	The FOI procedures notebook graphical interface	D,R
qa()	The Quality Assessment (QA) notebook graphical interface	D,R

Use: D=Can be used with direct access, R=Can be used with RESTful API

34.2 Data stracture

Example folder structure for parcel with ID 12345 stored in temp folder:


```
temp/
  aoi/2020/12345/info.json           # Parcel information in .json
  ↳ format
  aoi/2020/12345/time_series_s2.csv  # Time series form the parcel
  ↳ in .csv format
  aoi/2020/12345/backgrounds/*      # Background images
  aoi/2020/12345/chipimages/images_list.B04.csv # A list of downloaded images
  ↳ in .csv format
  aoi/2020/12345/chipimages/S2A_MSIL2A_2019---.B04.tif # The downloaded chip images
  aoi/2020/12345/chipimages/S2A_MSIL2A_...           # ...
```

34.3 Get widget



The get() function of ipycbm library provides an interactive Jupyter Notebook widget to get data from different sources, with variety of different methods (coordinates, parcels ids, map marker, polygon).

```
from cbm import ipycbm
ipycbm.get()
```

Get Data
Help
Settings


- Select the region and the year to get parcel information.
AOI: Spain (NOUR Subset) ▼  Year: 2019 ▼
- Select a method to download parcel data.

Parcel ID
Coordinates
Map marker
Polygon
- Select datasets to download.

 Time series
 Chip images
- Download the selected data.

By default data will be stored in the temp folder (temp/), you will be asked to empty the temp folder each time you start the notebook. In your personal data folder (data/) you can permanently store the data.

Select folder: ☒ Temporary folder: 'temp/'.
☐ Personal data folder: 'data/'.

 Download

34.4 View widget

```
from cbm import ipychm
ipychm.view()
```


INSTALLATION

35.1 Dependencies

The ‘cbm’ python library has one C library dependency: GDAL >=2.x. GDAL itself depends on many of other libraries provided by most major operating systems and also depends on the non standard GEOS and PROJ4 libraries, see: [Install GDAL](#).

35.2 Installing from PyPI

The cbm python library is available at the [Python Package Index](#) software repository.

Can be installed with:

```
pip3 install cbm
```

To upgrade use the -U flag

```
pip3 install cbm -U
```

35.3 Installing for development

For development install cbm in [editable mode](#) with:

```
git clone https://github.com/ec-jrc/cbm.git
cd cbm
pip3 install -e .
```

To update cbm with the local changes run in the cbm folder:

```
pip3 install -U -f -e .
```

35.4 Installing from source:

Install from source to get the latest updates (Not editable).

```
git clone https://github.com/ec-jrc/cbm.git
cd cbm
python setup.py install
```

35.5 Uninstallation

To uninstall cbm run:

```
pip3 uninstall cbm
```

35.6 Install GDAL

Open a terminal window and enter the command to update repositories:

```
sudo apt-get update && sudo apt-get upgrade
```

To get the add-apt-repository command, install the software-properties-common package:

```
sudo apt-get install software-properties-common
```

To get the latest GDAL/OGR version, add the PPA to sources, then install the gdal-bin package:

```
sudo add-apt-repository ppa:ubuntugis/ppa
```

Update your source packages:

```
sudo apt-get update
```

Install the GDAL/OGR package and other required packages as well:

```
sudo apt-get install -y \
    binutils \
    libproj-dev \
    gdal-bin \
    libgdal-dev \
    python3-gdal \
    python3-numpy \
    python3-scipy \
    python3-tk \
    graphviz \
    python3-dev \
    nano \
    g++ \
    gcc \
    libgdal-dev
```

Include paths using the environment variables with:

```
export CPLUS_INCLUDE_PATH=/usr/include/gdal
export C_INCLUDE_PATH=/usr/include/gdal
```

Then you will be able to install python gdal with:

```
pip install GDAL==$(gdal-config --version) --global-option=build_ext --global-option="-I/
↳usr/include/gdal"
```

For more information on GDAL installation see: [pypi GDAL Installation](#) or [How To Install GDAL/OGR Packages on Ubuntu](#).

35.7 Troubleshooting

In case of permission errors try using 'user' flag:

```
pip install cbm --user
```

On Windows GDAL may need to be installed with the [osgeo4w](#) installation package

In some cases the rasterio package may need to be installed with:

```
conda install -c conda-forge rasterio=1.1.5
```


CONFIGURATION

The main configurations for cbm are stored in the config/main.json file

To easily set the RESTful API account use:

```
cbm.set_api_account('http://0.0.0.0/', 'YOUR_USERNAME', 'YOUR_PASSWORD')
```

The account credentials will be stored automatically in the config/main.json file

You can configure the main configuration file (config/main.json) with a text editor of your choice. e.g.:

```
python3 -c "import cbm"
nano config/main.json
```

The main json configuration file has different sectors to store the settings for cbm:

```
{
  "set": {}, // General configurations
  "paths": {}, // The data and temp path are configurable and can be changed globally
  "files": {}, // Location of files used it some functions
  "api": {}, // The RESTful API credentials
  "db": {}, // Database access information (only if direct access is available)
  "s3": {} // the object storage credentials (only if direct access is available)
}
```


36.1 Configuration widget

To configure the config/main.json file interactively, in the jupyterlab environment create a new notebook and run in a cell:

```
# Import ipycbm
from cbm import ipycbm

# Open the configuration widget
ipycbm.config()
```

This will provide a widget with all configuration within a graphical interface

 Empty temp folder Your temp folder 'temp/' has old files: '['test2019', '.ipynb_checkpoints']', do you want to delete them?

DataSource General

Data sources: ☒ RESTful API for CbM.
☐ Direct access to database and object storage.

RESTful API Settings.

API URL: Format: http://0.0.0.0/ or https://0.0.0.0/

API User:

API Passw...

 Save

DIAS CATALOG

Transfer metadata from the DIAS catalog

Each DIAS maintains a catalog of the Level 1 and Level 2 CARD data sets available in the S3 store. Even with the use of standard catalog APIs (e.g. OpenSearch), the actual metadata served by the catalog may contain different, or differently named, attributes. Thus, in order to minimize portability issues, we implement a metadata transfer step, which makes metadata available in a single, consistent, database table `dias_catalogue` across the various DIAS instances. (see [data_preparation](#) for more information)

(Under development)

```
import cbm

tb_prefix = 'aoi2019' # The database table prefix
aoi = 'aoi'           # Area of interest e.g.: es, nld (str)
start = '2019-06-01'  # Start date (str)
end = '2019-06-30'    # End date (str)
card = 's2'           # s2, c6 or bs
option = 'LEVEL2A'    # s1 ptype CARD-COH6 or CARD-BS, s2 plevel : LEVEL2A or LEVEL2AP

cbm.card2db.creodias(tb_prefix, aoi, start, end, card, option)
```

Other DIAS options (future implementations):


- CREODIAS
- MUNDI
- SOBLOO
- WEKEO
- ONDA

37.1 Notebook widget


```
from cbm import ipycbm
ipycbm.card2db()
```


DIAS Provi...

AOI: ☐ Predefined MS polygons
☒ Get polygon from dataset extent


Start date 

card: ☐ Sentinel 1
☒ Sentinel 2

Table: 

End date 

pLevel: ☒ LEVEL2A
☐ LEVEL2AP

 Add CARD to db

EXTRACTION

If not configured already, the database and object storage credentials can be set with:

```
# Import the cbm python library
import cbm

# Add the database credentials
db_host = 'http://0.0.0.0'
db_port = '5432'
db_name = 'postgres'
db_user = 'postgres'
db_pass = ''
cbm.config.set_value(['db', 'main', 'host'], db_host)
cbm.config.set_value(['db', 'main', 'port'], db_port)
cbm.config.set_value(['db', 'main', 'name'], db_name)
cbm.config.set_value(['db', 'main', 'user'], db_user)
cbm.config.set_value(['db', 'main', 'pass'], db_pass)

# Add the object store credentials. Should be given from the DIAS provider.
osdias = ''      # The name of the dias provider
oshost = ''      # The host address
bucket = ''      # The bucket name
access_key = ''  # The access key of the object store
secret_key = ''  # The secret key of the object store
cbm.config.set_value(['obst', 'osdias'], osdias)
cbm.config.set_value(['obst', 'oshost'], oshost)
cbm.config.set_value(['obst', 'bucket'], bucket)
cbm.config.set_value(['obst', 'access_key'], access_key)
cbm.config.set_value(['obst', 'secret_key'], secret_key)
```

This can be done manually as well by opening the config/main.json file.

If the credentials are set properly the extractions can be done with:

```
import cbm

start = '2019-06-01'
end = '2019-06-30'
dias_catalogue = 'aoi2019_dias_catalogue'
parcels_table = 'aoi2019'
results_table = 'aoi2019_s2_signatures'
dias = 'dias'
```

(continues on next page)

(continued from previous page)

```
cbm.extract.s2(start, end, dias_catalogue, parcels_table, results_table, dias)
```

38.1 Extract widget

This widget provides the credentials configuration widget as well

```
from cbm import ipychm
ipychm.extract()
```

Extract Data

Help

Settings

- Connect to database and object storage.

Configure: mai
- Create the essential CbM tables. The parcels table name will be added as prefix.

Add a short name as prefix for the tables, max 15 characters e.g.:escat2020.

Table prefix:

☒ DIAS Catalogue
 ☒ S2 signatures
 ☒ S1 Backscattering
 ☒ S1 6-day coherence
 ☐ AOIs (Optional) - R...

Create tables
- Upload .shp, with all the required files (.shp, .cpg, .dbf, .prj, .shx).
 - Upload .shp to the server.

Folder:

+ Select files: (0)
 - Import uploaded .shp to the database. Add a short name, max 15 characters for the parcels table e.g.:escat2020.

Table name:
 Select .shp:

Import .shp file

☐ Remove old entries
- Add CARD metadata to database tabel 'xx_dias_catalogue'.

DIAS Provi...:

EOSC

AOI:

☐ Predefined MS polygons
 ☒ Get polygon from dataset extent

Table:

bev12018

Start date:

mm/dd/yyyy

 End date:

mm/dd/yyyy

card:

☐ Sentinel 1
 ☒ Sentinel 2

pLevel:

☒ LEVEL2A
 ☐ LEVEL2AP

Add CARD to db
- Run parcel extraction routines.

Start date:

mm/dd/yyyy

 End date:

mm/dd/yyyy

Sentinel 2

S1 Backscattering

S1 6-day cohere...

FOI EXAMPLES

FOI Assessment: Heterogeneity and Cardinality

The FOI assessment python package is based on the principle that inside of a homogenous FOI there should be only one type of land use on the same type of land cover; thus, if we apply a given image classification technique on Sentinel data, it would label the pixels to the same class value. In the same manner, a FOI which respects the 1-1 cardinality should not include clusters of pixels, labelled with different class values, larger than a specified threshold (we can consider dispersed pixels labelled differently than the main class value as “noise”).

The FOI Assessment performs a spatial analysis on such “thematic raster” produced in advance. The thematic raster can be the result of any image/raster processing method yielding a class label for each pixel - supervised (land cover/crop) classification, unsupervised clustering based on the behavior of land phenomenon, gridded data on any terrain characteristics, such as soil type, slope, hydrology, etc.

As an example, if the thematic raster is the result of a crop classification, a homogeneous FOI should enclose only pixels with class value (or label) that represent the respective crop; a spatially congruent (1-1 cardinality) FOI should not include any cluster of pixels from other class label larger than a specified threshold (for example, expressed as maximum number of pixels in the cluster). If the thematic raster is the result of an unsupervised behavior analysis, all the pixels inside an FOI should be labeled with the value corresponding to a particular behavior during a pre-defined period of time. For both heterogeneity and cardinality, the python package provides two methods for the analysis: one based area calculation (version 1) and one based on cluster size calculation (version 2). Both methods yield similar results.

To run the FOI functions the configuration file need to be set, see `CbM Package Configuration <https://jrc-cbm.readthedocs.io/en/latest/cbm_config.html>` for more details.

39.1 Version 1

The first version requires the connection to a database server (PostgreSQL >v12 with PostGIS extension). For the heterogeneity analysis the following steps are required.

1. Connect to the database (at the moment only “Database connection settings” are required)
 1. Upload the reference data shapefile to the server.
 2. Import uploaded shapefile to the database, specifying the name for the table that will be created in the database.
2. Upload the “thematic raster”. The accepted files are tif or tiff files. The thematic raster should be a one band raster file, with the pixel values representing the class labels (representing land cover, crop type or type of behaviour)
3. Create the necessary function and stored procedures on the database server.
4. Select the required files for analysis

1. Vector file: the data containing the FOI geometries on which the analysis will be applied. In case that we have more shapefiles uploaded on the server, this functionality allows us to select the one that we want to analyze.
- #. Thematic raster: the thematic raster provided. In case that we have more rasters uploaded on the server, this functionality allows us to select the one that we want to use on the analysis. #.

YAML file that holds the classes from the thematic raster file: this file specifies the classes of pixels from the thematic raster and can also provide the meaning of those classes. It should have the following structure:

example.yml

```
category_map:
  0: Unclassified
  1: Class1
  2: Class2
  3: Class3
  4: Class4
  5: Class5
  6: Class6
  7: Class7
  8: Class8
  9: Class9
  10: Class10
```

Class1, Class2, Class_n can be replaced by the meaning of the class (like Wheat, Maize, etc. or by behavior name or any other). The YAML file should include all the classes that exist in the thematic raster.

5. Analysis parameters: Heterogeneity thresholds: in order to exclude the influence of „noise” pixels, the user can specify the heterogeneity thresholds. For example, only the FOIs where one class of pixels have a percentage between 30 and 70 is considered heterogeneous (this implies that the remaining pixels are labelled with different class). Minimum area for clusters selection: the user can specify the minimum area of the cluster that are considered a cardinality issue, in square meters. Of example the clusters smaller than 2000 square meters can be considered as not influencing the FOI cardinality.
6. Run the FOI analysis.

The result of the analysis is represented by three shapefiles that are stored on the “output_data” folder (/cbm/tmp/foi/output_data).

“name of the shapefile dataset” (without extension) that needs to be tested + foi_h_v1.shp – represents the initial shapefile and during the analysis the following attributes are added:

- foi_h – heterogeneity flag (0 for homogeneous FOIs and 1 for heterogeneous FOIs)
- number of pixels for each class (the name of the attribute is the name of the class)
- total number of pixel for the respective FOI
- percentage of pixels from each class (number of pixels for each class / total number of pixels inside the FOI)

“name of the shapefile dataset” (without extension) that needs to be tested + foi_c_v1.shp - represents the initial shapefile and during the analysis the following attributes are added:

- foi_c – cardinality flag (0 for FOIs respecting the 1-1 cardinality and 1 for FOIs not respecting the 1-1 cardinality). As a result of this analysis, the FOIs that include more than one cluster of pixel from different classes bigger than the threshold are considered not respecting 1-1 cardinality. For example a FOI that includes two clusters of pixels from different classes (arable land and non-agricultural area), each of the clusters bigger than the threshold (ex. 2000 square meters), is considered as not respecting the 1-1 cardinality.

“name of the shapefile dataset” (without extension) that needs to be tested + foic_clusters_v1.shp – represents only the clusters of pixels that are defining the FOI cardinality (for example if an FOI includes three clusters of pixels bigger than the threshold, only those clusters will be saved in this shapefile)

Code non interactive

```
import cbm

vector_file = "data/parcels2020.shp"
raster_file = "data/raster.tif"
yaml_file = "data/pixelvalues_classes.yml"
pre_min_het = 30
pre_max_het = 70
area_threshold = 2000

cbm.foi.v1(vector_file, raster_file, yaml_file,
           pre_min_het, pre_max_het, area_threshold)
```

Interactive Jupyter Notebook widget

The subpackage ipycbm.foi() provides a graphical interface for the required steps.

```
from cbm import ipycbm
ipycbm.foi()
```

FOI Assessment V1
FOI Assessment V2
Help
Settings

FOI procedures need direct access to the database.

In case there no image is provided, access to object storage will be needed to generate the base image from sentinel images.

1. Connect to database and object storage.

Configure: 1

a. Upload .shp to the server.

Folder: tmp/foi/vector Select files: (0)

b. Import uploaded .shp to the database. Add a short name, max 15 characters for the parcels table e.g.:escat2020.

Table name: ms2010 Select .shp: ☐ Remove old entries

3. Upload or generate raster base image. (Only upload is currently available)

Folder: tmp/foi/raster/ Select file: (0)

4. Prepare FOI procedure.

Add functions to database:

Select the required files:

a. Spatial data to be tested - parcels that will be checked for heterogeneity and cardinality.

Vector file:

b. Thematic raster - classification raster, or raster from other source that will be used for testing heterogeneity and cardinality.

Raster file:

c. YAML file that holds the classes from the thematic raster file - can be also a simple list of values in the notebook correspondence between pixel values and names for the classes

yaml file: Select file: (0)

Minimum and maximum thresholds for heterogeneity checks. In the example, any parcel with percentage of pixels for one class between 30 and 70 from the total, will be considered heterogenous.

MIN: 30 MAX: 70

area: 2000 Minimum area for clusters selection - only clusters bigger from this threshold will be counted.

5. Run FOI procedure.

39.2 Version 2

The second version does not require a database server. All the calculations are made directly at pixel level using Python function (without the creation of intermediate vectors for the area calculation). The interface and the steps are similar to the ones from the Version 1. The main difference is that it does not include the functionality for database connection and creating the functions on the database server.

The different options available:

Connectivity type: 8 or 4 connected pixels with the same class label (4 indicating that diagonal pixels are not considered directly adjacent for polygon membership purposes or 8 indicating they are) Negative buffer: user can apply a negative buffer on the FOI in order to reduce in the analysis, the role of the pixels located at the boundary (assumed influenced by roads, adjacent FOIs, etc.) Cluster size (in pixels): the minimum number of pixels with the same class label in cluster, in order to be taken into account in the cardinality check.

The result of the analysis is represented by two shapefiles that are stored on the “output_data” folder (/cbm/data/foi/output_data). Name of the shapefile dataset (without extension) that needs to be tested + foih_v2.shp – represents the initial shapefile and during the analysis the following attributes are added:

- foi_h – heterogeneity flag (0 for homogeneous FOIs and 1 for heterogeneous FOIs)
 - number of pixels for each class (the name of the attribute is the name of the class)
 - total number of pixel for the respective FOI
 - percentage of pixels from each class (number of pixels for each class / total number of pixels inside the FOI)
- name of the shapefile dataset (without extension) that needs to be tested + foic_v2.shp - represents the initial shapefile and during the analysis the following attributes are added:
- foi_c – cardinality flag (0 for FOIs respecting the 1-1 cardinality and 1 for FOIs not respecting the 1-1 cardinality). As a result of this analysis, the FOIs that include more than one cluster of pixels from different classes bigger than the threshold are considered not respecting the 1-1 cardinality. For example and FOI that includes two clusters of pixels from different classes (arable land and non-agricultural area), each of the clusters bigger than the threshold (ex. 20 pixels), is considered as not respecting the 1-1 cardinality.
 - Clusters – the information about the clusters of pixels identified inside the FOI, as pair of pixel class and cluster size: for example (3, 25), (5, 120) means that inside the FOI we have identified two clusters: one of pixels from class 3 and the cluster size is 25 pixels and another one with pixels of class 5 and cluster size 120 pixels.

Code non interactive

```
import cbm

vector_file = "data/parcels.shp"
raster_file = "data/raster.tif"
yaml_file = "data/pixelvalues_classes.yml"
negative_buffer = -10
min_heterogeneity_threshold = 30
max_heterogeneity_threshold = 70
connectivity_option = 8
cluster_threshold = 20

cbm.foi.v2(vector_file, raster_file, yaml_file, negative_buffer, min_heterogeneity_
↪ threshold,
          max_heterogeneity_threshold, connectivity_option, cluster_threshold)
```



Interactive Jupyter Notebook widget

The subpackage `ipycbm.foi()` provides a graphical interface for the required steps.



```
from cbm import ipycbm
ipycbm.foi()
```

FOI Assessment V1
FOI Assessment V2
Help
Settings

a. Upload .shp to the server.



Folder: + Select files: (0)  

b. Import uploaded .shp to the database. Add a short name, max 15 characters for the parcels table e.g.:escat2020.

Table name: Select .shp:   Import .shp file ☐ Remove old entries

3. Upload or generate raster base image. (Only upload is currently available)

Upload Generate

Folder: + Select file: (0)  



4. Prepare FOI procedure.

Select the required files:

Vector file: Spatial data to be tested - parcels that will be checked for heterogeneity and cardinality.

Raster file: Thematic raster - classification raster, or raster from other source that will be used for testing heterogeneity and cardinality.

YAML file that holds the classes from the thematic raster file - can be also a simple list of values in the notebook correspondence between pixel values and names for the classes

yaml file: + Select file: (0)  

Minimum and maximum thresholds for heterogeneity checks. In the example, any parcel with percentage of pixels for one class between 30 and 70 from the total, will be considered heterogenous.

MIN: MAX:

connectivit...

negative b...

pixels: Minimum area for clusters selection - only clusters bigger from this threshold will be counted.

5. Run FOI procedure.

▶ Run FOI procedu...

DATA ACCESS

CbM data can be accessed through RESTful API or directly accessing the database and object storage, direct access is only for administrators.

Preferable way to access the data is through RESTful API. To easily set the RESTful API account you can use the below python command:

```
cbm.set_api_account('http://0.0.0.0/', 'YOUR_USERNAME', 'YOUR_PASSWORD')
```

Alternatively see ‘[cbm package configuration](#)’ for other options.

40.1 Parcel information

Download parcel information by ID.

```
import cbm

aoi = 'ms'           # area of interest (str)
year = 2020          # the year of the parcels dataset (int)
pid = '12345'        # parcel id (str)
ptype = None         # (optional) specify the dataset*
with_geometry = True # get the parcel geometry
debug = False        # show debugging output

cbm.get.parcel_info.by_pid(aoi, year, pid, ptype, with_geometry, debug)
```

Download parcel information by location.

```
import cbm

aoi = 'ms'           # area of interest (str)
year = 2020          # the year of the parcels dataset (int)
lat = 50.12345       # latitude in decimal degrees
lon = 5.12345        # longitude in decimal degrees
ptype = None         # (optional) specify the dataset*
with_geometry = True # get the parcel geometry
debug = False        # show debugging output

cbm.get.parcel_info.by_location(aoi, year, pid, ptype, with_geometry, debug)
```

40.2 Time series

ptype is used only in case there are different datasets dedicated to different type of analysis for the same year. For example datasets dedicated to grazing use **g**, for mowing **m** etc.

scl Scene Classification Values

The classification mask is generated along with the process of generating the cloud mask quality indicator and by merging the information obtained from cirrus cloud detection and cloud shadow detection.

The classification map is produced for each SENTINEL-2 Level-1C product at 60 m resolution and byte values of the classification map are organised as shown below:

Label	Classification
0	NO_DATA
1	SATURATED_OR_DEFECTIVE
2	DARK_AREA_PIXELS
3	CLOUD_SHADOWS
4	VEGETATION
5	NOT_VEGETATED
6	WATER
7	UNCLASSIFIED
8	CLOUD_MEDIUM_PROBABILITY
9	CLOUD_HIGH_PROBABILITY
10	THIN_CIRRUS
11	SNOW

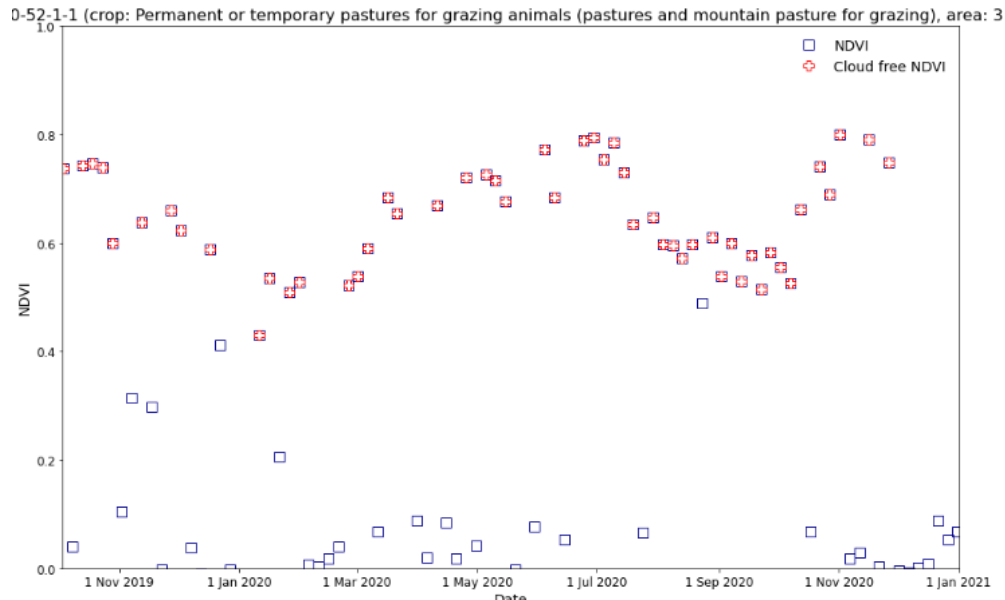
40.2.1 Show time series

Plot NDVI time series

```
import cbm

aoi = 'ms'           # area of interest (str)
year = 2020          # the year of the parcels dataset (int)
pid = '12345'        # parcel id (str)
ptype = None         # (optional) specify the dataset*
cloud_free = True    # mark cloud free images based on scl values
scl = '3_8_9_10_11'  # scene classification values to be excluded
debug = False        # show debugging output

cbm.show.time_series.ndvi(aoi, year, pid, ptype, cloud_free, scl, debug)
```



Plot S2 bands

```
import cbm

aoi = 'ms'           # area of interest (str)
year = 2020          # the year of the parcels dataset (int)
pid = '12345'        # parcel id (str)
ptype = None         # (optional) specify the dataset*
bands = ['B03', 'B04'] # bands to plot in a list
cloud_free = True    # mark cloud free images based on scl values
scl = '3_8_9_10_11'  # scene classification values to be excluded
debug = False        # show debugging output

cbm.show.time_series.s2(aoi, year, lat, lon, ptype, bands, cloud_free, scl, debug)
```

40.2.2 Download time series

Download the time series without plotting for a knowing parcel ID.

```
import cbm

aoi = 'ms'           # area of interest (str)
year = 2020          # the year of the parcels dataset (int)
pid = '12345'        # parcel id (str)
ptype = None         # (optional) specify the dataset*
tstype = 's2'        # time series type (str)

import cbm
cbm.get.time_series.by_pid(aoi, year, pid, tstype, ptype)
```

Download the time series without plotting based on the parcel location.

```
import cbm

aoi = 'ms'           # area of interest (str)
year = 2020          # the year of the parcels dataset (int)
pid = '12345'        # parcel id (str)
ptype = None         # (optional) specify the dataset*
lat = 50.12345       # latitude in decimal degrees
lon = 5.12345        # longitude in decimal degrees
tstype = 's2'        # time series type (str)

cbm.get.time_series.by_location(aoi, year, lat, lon, tstype, ptype)
```

The files will be saved by default in the temp/ms/year/parcel/... folder.

40.3 Background images

40.3.1 Show background images

```
import cbm

aoi = 'ms'           # area of interest (str)
year = 2020          # the year of the parcels dataset (int)
pid = '12345'        # parcel id (str)
chipsize = 512       # size of the chip in pixels
extend = 512         # size of the chip in meters
tms = ['Google']     # tile map server, Google, Bing or MS orthophotos
ptype = None         # (optional) specify the dataset*
columns = 4          # the number of grid columns
debug = False        # show debugging output

cbm.show.background.by_pid(aoi, year, pid, ptype, chipsize, extend, tms, ptype, columns,
↪ debug)
```

```
[1]: import cbm
      cbm.show.background.by_pid('ms', '2020', '12345', 256, 256, ['bing', 'google'])
```



40.3.2 Download background images

To download background images without plotting the images for a knowing parcel ID.

```
import cbm

aoi = 'ms'           # area of interest (str)
year = 2020          # the year of the parcels dataset (int)
pid = '12345'        # parcel id (str)
chipsize = 512       # size of the chip in pixels
extend = 512         # size of the chip in meters
tms = ['Google']     # tile map server, Google, Bing or MS orthophotos
ptype = None         # (optional) specify the dataset*
axis = True          # show axis
debug = False        # show debugging output

cbm.get.background.by_pid(aoi, year, pid, ptype, chipsize, extend, tms, ptype, axis,
    ↪ debug)
```

To download background images without plotting the images based on the parcel location.

```
import cbm

aoi = 'ms'           # area of interest (str)
year = 2020          # the year of the parcels dataset (int)
pid = '12345'        # parcel id (str)
lat = 50.12345       # latitude in decimal degrees
lon = 5.12345        # longitude in decimal degrees
chipsize = 512       # size of the chip in pixels
extend = 512         # size of the chip in meters
tms = ['Google']     # tile map server, Google, Bing or MS orthophotos
ptype = None         # (optional) specify the dataset*
axis = True          # show axis
debug = False        # show debugging output

cbm.get.background.by_location(aoi, year, lat, lon, tstype, axis, ptype)
```

The files will be saved by default in the temp/ms/year/parcel/... folder.

40.4 Sentinel chips

Download chip images by selected parcel id.

Example

```
import cbm

cbm.get.chip_images.by_pid(
    aoi='dk',          # Area of interest (str)
    year='2020',       # Year of the parcels dataset (int)
    pid='92794229',    # Parcel id (int)
    start_date='2020-06-01', # Start date '2019-06-01' (str)
    end_date='2020-06-30',  # End date '2019-06-01' (str)
```

(continues on next page)

(continued from previous page)

```
band='SCL',          # Selected band
chipsize=2560        # size of the chip in pixels (int)
)
```

Preview chip images by selected parcel id.

Example

```
import cbm

cbm.show.chip_images.by_pid(
    aoι='dk',          # Area of interest (str)
    year='2020',       # Year of the parcels dataset (int)
    pid='92794229',    # Parcel id (int)
    start_date='2020-06-01', # Start date '2019-06-01' (str)
    end_date='2020-06-30',  # End date '2019-06-01' (str)
    band='SCL',        # Selected band
    chipsize=2560      # size of the chip in pixels (int)
)
```

Arguments

band: 3 Sentinel-2 band names. One of ['B02', 'B03', 'B04', 'B08'] (10 m bands) or ['B05', 'B06', 'B07', 'B8A', 'B11', 'B12'] (20 m bands). 10m and 20m bands can be combined.

CONTRIBUTING TO CBM

Please take a moment to review this document in order to make the contribution process easy and effective for everyone involved.

Following these guidelines helps to communicate that you respect the time of the developers managing and developing this open source project. In return, they should reciprocate that respect in addressing your issue or assessing patches and features.

41.1 Using the issue tracker

The [issue tracker](#) is the preferred channel for submitting *pull requests* and *bug reports*, but please **do not** use the issue tracker for personal support requests. Consider one of the following alternatives instead:

- [JRC GTCAP Wikis platform](#) for Q&A as well as support for troubleshooting, installation and debugging. Do remember to tag your questions with the tag `cbm`.
- [Contact directly JRC GTCAP team](#)
- [CbM forum](#) (Under development)

41.2 Pull requests

Good pull requests - patches, improvements, new features - are helpful. They should remain focused in scope and avoid containing unrelated commits.

Please ask first before embarking on any significant pull request (e.g. implementing features, refactoring code), otherwise you risk spending a lot of time working on something that the project's developers might not want to merge into the project. Read the [tutorial on writing new CbM functions](#) if you want to contribute a brand new feature.

If you are new to Git, GitHub, or contributing to an open-source project, you may want to consult our [guide on preparing and submitting a pull request](#) or check our list of [useful related documentations](#).

41.3 Bug reports

A bug is a *demonstrable problem* that is caused by the code in the repository. Good bug reports are extremely helpful - thank you!

Guidelines for bug reports:

1. **Check if the issue has been reported** — use GitHub issue search and mailing list archive search.
2. **Check if the issue has been fixed** — try to reproduce it using the latest `main` branch in the repository.
3. **Isolate the problem** — ideally create a reduced test case.

A good bug report shouldn't leave others needing to chase you up for more information. Please try to be as detailed as possible in your report. What is your environment? What steps will reproduce the issue? What would you expect to be the outcome? All these details will help people to fix any potential bugs.

Example:

Short and descriptive example bug report title

A summary of the issue and the OS environment in which it occurs. If suitable, include the steps required to reproduce the bug.

1. This is the first step
2. This is the second step
3. Further steps, etc.

Any other information you want to share that is relevant to the issue being reported. This might include the lines of code that you have identified as causing the bug, and potential solutions (and your opinions on their merits).

41.4 License

CbM is 3-Clause BSD licensed, and by submitting a patch, you agree to allow Open Perception, Inc. to license your work under the terms of the BSD 3-Clause License. The link of the license should be inserted as a comment on top of each `.py` file e.g.:

```
# This file is part of CbM (https://github.com/ec-jrc/cbm).  
# Copyright : 2021 European Commission, Joint Research Centre  
# License   : 3-Clause BSD
```

Note that if the academic institution or company you are affiliated with does not allow to give up the rights, you should contact us for further details.

PULL REQUESTS

Below are the steps on how to Fork a GitHub Repository and Submit a Pull Request.

Before you continue you may want to check how to [Creating a personal access token](#) on your GitHub account.

1. Fork the project, clone your fork, and configure the remotes:

```
# Clone your fork of the repo into the current directory
git clone https://github.com/<your-username>/cbm.git
# Navigate to the newly cloned directory
cd cbm
# Assign the original repo to a remote called "upstream"
git remote add upstream https://github.com/ec-jrc/cbm.git
```

2. If you cloned a while ago, get the latest changes from upstream:

```
git checkout main
git pull upstream main
```

3. (Optional) Create a new topic branch (off the main project development branch) to contain your feature, change, or fix:

```
git checkout -b <topic-branch-name>
```

4. Commit your changes in logical chunks. For any Git project, some good rules for commit messages are

- the first line is commit summary, 50 characters or less,
- followed by an empty line
- followed by an explanation of the commit, wrapped to 72 characters.

See [a note about git commit messages](#) for more.

5. Push your topic branch up to your fork:

```
git push origin <topic-branch-name>
```

6. [Open a Pull Request](#) with title and description.

7. When your commit is merged to main branch you can delete your branch:

```
git checkout main
git branch -D <topic-branch-name>
```