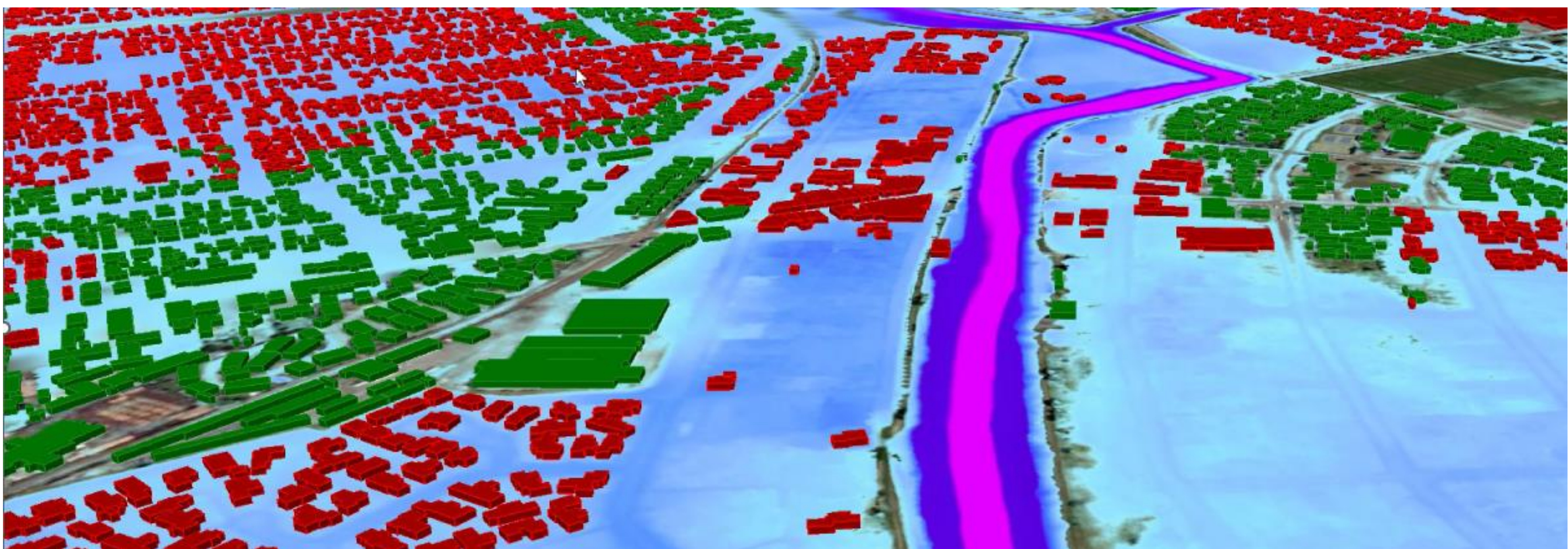
Flood Resilience Digital Twin



Final project report to FrontierSI (project number 3007)

6 November 2023

GRI
GEOSPATIAL RESEARCH INSTITUTE Toi Hangarau

BUILDING INNOVATION PARTNERSHIP

Executive Summary

This research project was a research collaboration between the Building Innovation Partnership (BIP) and the Geospatial Research Institute Toi Hangarau (GRI) at the University of Canterbury, New Zealand, conducted between December 2020 and August 2023.

Following engagement workshops, the project successfully developed and demonstrated a prototype digital twin for a site north of Christchurch, New Zealand. The focus the work was to build an environmental digital twin which brings together computational models of flood inundation with other data, for hazard assessment, management, and mitigation. A key objective was to enable the automation of flood risk assessment, such that multiple scenarios can be assessed rapidly, such as when given updated information. The fundamental design principle was that the digital twin should broadly adopt the “Findable, Accessible, Interoperable and Reusable” (FAIR) principles, and be open, extendable, interoperable, replicable, dynamic, versatile and scalable. Research was underpinned by methods developed in an aligned research programme, Mā te haumaru ō te wai, led by NIWA, which is assessing flood risk across New Zealand.

The conceptual design of the digital twin includes geospatial databases to hold the input data, including static data used for boundary conditions (e.g., LiDAR data, stopbanks, storm water, channel geometry and land cover), dynamic boundary data used for model water inputs (e.g., river gauge data, tide levels, rainfall data, design flows), observational data for model comparison (e.g., airborne flood imagery, SAR flood imagery, survey data) and data for infrastructure and population for impact assessment (e.g., buildings, census data, roads and rail). A design principle for the digital twin was that, for a selected area, these data are downloaded from their respective agencies and stored in a local database without modification (i.e., features are kept intact). A local copy of the data is maintained to avoid the high bandwidth which may be associated with multiple downloads, as well as speed up processing when the data are used for multiple analyses. To achieve this, a database of the metadata is maintained, which tracks areas which have been previously processed, and when.

After download and incorporation into local databases, data are further processed within the digital twin to create the “GeoFabric”, which refers to the model inputs and boundary conditions, including the scenarios assessed, in the appropriate model format (in this case, BG-Flood). Dynamic boundary condition data are obtained which represent flows into the domain and a downstream tide level. Rainfall and river flow input data are derived statistically from existing databases which have been generated based on historical observations. For the tide level, by default the annual maximum high tide for the last year is obtained. To account for sea level rise in the scenarios, differing levels of sea level can be included on top of the tide level. Climate scenarios can be included in both the rainfall and sea level. The model is then run, and the digital twin maintains metadata regarding simulations. Finally, model results are incorporated back into the digital twin for analysis, such as identifying flooded buildings. To complement the data processing and computational engine, which forms the foundation of the digital twin, we have developed a web-based visualisation environment based on the Cesium open platform for 3D geospatial data.

Future work will include additions to the backend functionality such as additional scenarios, connections to additional data, development of a storm drainage module, integration of machine learning methods, and a connection to RiskScape. For the frontend, an improved user interface will be developed alongside dynamic and immersive visualisations. Finally, we will develop a hosted digital twin solution and seek to scale across New Zealand and, longer term, internationally.

Contents

Executive Summary ... 3
Research team and funding ... 5
1 Introduction ... 6
2 Towards a Flood Resilience Digital Twin ... 7
2.1 Previous work in digital twins in disaster management ... 7
2.2 Purpose of our work and design principles ... 8
3 Development workshops ... 9
4 Spatial domain for prototype development ... 10
5 Flood Resilience Digital Twin: Software summary ... 12
5.1 Conceptual design ... 12
5.2 Current implementation ... 13
5.3 Current limitations ... 18
6 Flood Resilience Digital Twin: Visualisation frontend system ... 20
7 Key lessons and challenges ... 23
7.1 Data ... 23
7.2 Open-source software ... 25
7.3 Physics-based Digital Twins ... 25
8 Future work ... 26
9 References ... 27
10 Appendix A: Software installation ... 31
10.1 Introduction ... 31
10.2 Basic running instructions ... 31
10.3 Requirements ... 31
10.4 Required Credentials: ... 32
10.5 Starting the Digital Twin application (localhost) ... 32
10.6 Using the Digital Twin application ... 32
10.7 Setup for developers ... 33
10.7.1 Run single Docker service e.g. database ... 33
10.7.2 Create Conda environment ... 33
10.7.3 Run Celery locally ... 33
10.7.4 Running the backend without web interface. ... 33
10.8 Tests ... 33
10.8.1 Automated testing ... 33
10.8.2 Running tests locally ... 33
10.9 Vector Database ... 33
10.10 Raster Database ... 35
10.11 LiDAR Database ... 35
10.12 Create extensions in PostgreSQL: ... 35
10.12.1 Exploring the database created ... 37
11 Appendix B: Software licence ... 38
12 Appendix C: API Documentation ... 39

Research team and funding

The project was funded by BIP, FrontierSI (project number 3007) with in-kind support from Land Information New Zealand (LINZ) and the National Institute for Water and Atmospheric Research (NIWA). Additional funding for aligned work was provided by the GRI at the University of Canterbury, including the development of the front-end utilised by the digital twin, and contributions to methods for LiDAR processing utilised in the digital twin.

Main funding:

Project partners:

Research team members:

Organisation	Team member	Role
University of Canterbury	Prof. Matt Wilson	Principal investigator
	Greg Preston	Project manager
	Casey Li	Backend developer
	Luke Parkinson	Frontend developer
	Xander Cai	LiDAR code development
	Pooja Khosla	Early prototype development
NIWA	Dr. Emily Lane	Lead investigator for aligned programme
	Dr. Rose Pearson	Developer of the Geofabrics library
	Dr. Cyprien Bosserelle	Developer of the BG-Flood model software
LINZ	Rob Deakin	Key data partner and advisory role

1 Introduction

Flooding is frequent, widespread, and impactful, regularly causing damage along with disruption to communities (McDermott, 2022). This is expected to increase with climate change (Arnell & Gosling, 2016). Management of flood risk requires Flood Risk Assessment (FRA) to identify areas which may be impacted and develop mitigation measures (Ali et al., 2016; Bates et al., 2004; Tsakiris, 2014; Winsemius et al., 2013). In these assessments, geospatial data (e.g., high-accuracy elevation, land use data) are combined with meteorological and/or hydrological data to create flood inundation scenarios within hydraulic models of surface water flows. The outputs of these models are then intersected with additional geospatial data representing assets such as buildings and utilities, or data on population distribution, enabling the impact of the flood event to be assessed. Despite often being built on open data, there are barriers to FRAs being developed, accessed and used (Thieken et al., 2006). For example, the computational modelling and scenario assessments required for the FRAs used for management and mitigation requires substantial amounts of spatial data related to infrastructure and the environment, which can make it challenging and expensive, and leading to variability in the level of detail possible (Glas et al., 2020; Ocio et al., 2016). In this research, our focus has been to test whether the concept of a "digital twin" can address these critical challenges, and thereby enable decision-making in flood management by providing a means to rapidly assess flood risk, across wide areas, for a far greater number of scenarios.

The focus of our work has been to build an environmental digital twin which brings together computational models of flood inundation with other data, for hazard assessment, management, and mitigation. A key objective was to enable the automation of flood risk assessment, such that multiple scenarios can be assessed rapidly, such as when given updated information. The key objectives of the research were:

1. To assess existing and/or develop new standards and specifications for spatial data of relevance to flood resilience in urban areas, including but not limited to infrastructure such as pipes, storm water drainage systems, streamlines, culverts and stopbanks (levees), topographic data from terrestrial LiDAR, channel bathymetry, land cover and other infrastructure of relevance such as buildings and roads.
2. To test these standards within an interoperability experiment based on those of the Open Geospatial Consortium (OGC).
3. To use the specifications to develop a "digital twin" and implement this for automated generation of flood inundation models for rapid flood risk assessment, based on new methods of machine learning for automated feature extraction.
4. To test and demonstrate this digital twin for an urban flood event.

The first two objectives were address through two workshops with stakeholders in January 2021 and January 2022. These workshops are briefly summarised in Section 3, and previously reported. The bulk of this document is focussed on the final two objectives, and provides details of the approach we have taken, including the rational and design principles for this. In the Appendices, documentation is provided for setting up and running the software on a server, including the Application Programming Interface (API) which is part of the back-end system.

2 Towards a Flood Resilience Digital Twin

Over the last few years, the development of digital twins has accelerated greatly, especially within the manufacturing industry (Jones et al., 2020; Semeraro et al., 2021). Numerous definitions of the concept of a digital twin exist, with definitions tending to vary depending on the field (Fuller et al., 2020); in general terms, a digital twin is a dynamic virtual representation of a physical system (e.g., Madni et al., 2019), with automated data exchange being a key attribute. Digital twins are enabling the development of the next generation of smart cities (Deren et al., 2021); more recently, the concept has expanded to include digital twins of the natural environment (Blair, 2021) and there is, for example, significant investment by the European Union (EU) towards the creation of a digital twin of Earth, with the aim of ensuring climate neutrality by 2050 (Bauer et al., 2021). The EU's "Destination Earth" (DestinE) policy brings together computation and data lakes to create a "seamless fusion of real-time observations and high resolution predictive modelling" for critical application areas including extreme events and climate change adaptation (European Commission, 2022).

2.1 Previous work in digital twins in disaster management

The concept of the use of digital twins to improve disaster risk management is a rapidly evolving concept and is an amalgamation of several aligned technologies (Ariyachandra & Wedawatta, 2023). Digital twins are becoming widely recognized as an umbrella technology which will, for example, enable us to improve our resilience to climate change through smart cities (Riaz et al., 2023). Early examples of the use of the digital twin concept disaster risk assessment includes work by Ford & Wolf (2020) who demonstrated the linkage of a smart city system with a community simulation model in order to improve decision making related to evacuation, while accounting for real-time traffic information. Similarly, Ham & Kim (2020) developed a conceptual framework for the inclusion of crowd-sourced data in a 3D city model, in order contribute to the development of risk-informed decision making.

In the area of flood risk assessment, Ghaith et al. (2021) demonstrated a digital twin for Calgary, Canada, which included outputs from the HEC-RAS flood model (Brunner, 2002) as part of city visualisations. However, the digital twin didn't automate the application of the model, so it wouldn't be possible to include analysis using updated the simulations based on real-time information. Advanced statistical analysis has been included within digital twins to aid the rapid assessment of flood risk, reducing the computational overhead. For example, Alperen et al. (2021) developed a hydrological digital twin for flood simulation in a small catchment in France, using a hybrid physical and statistical approach, where a neural network was used to approximate the outputs from the physical model. However, the transferability of such an approach to other areas may be limited. Similarly, Jiang et al. (2021) used physics-informed machine learning for a coastal digital twin for assessing areas of flooding. Tarpanelli et al. (2023) demonstrated that the use of large volumes of satellite imagery for rapid, automated, mapping of flood inundation can be considered as a component of a larger digital twin system.

Digital twins have also been demonstrated as a way to develop smarter, nature-based solutions to hydro-meteorological hazards such as floods (Ruangpan et al., 2023), or as a way to act as an efficient flood prediction tool for emergency warning, reacting to observational data using AI (Manocha et al., 2023). In ongoing work, the FloodDAM-DT project aims to "provide an automated service to reliably detect, monitor and assess floods at global scale", using Earth observation data and modelling to provide rapid detection and assessments of ongoing flooding in near real-time (Suquet et al., 2023).

The FloodDAM-DT proof-of-concept project is a large collaboration between CNES and NASA/ JPL and demonstrates that the use of the digital twin concept in disaster risk management will likely receive increased research attention over the next few years.

2.2 Purpose of our work and design principles

The work outlined above demonstrates that that digital twins show great potential to aid in flood risk assessment and management. Limitations of previous work include a narrow application to one or two sites, the offline use of flood models meaning that they are unable to react to new information, and a focus on mapping or detection for emergency management, rather than preparedness through improved scenario assessment. While this latter approach is important, there is currently a gap in the availability of generally applicable digital twins for improving flood risk assessment ahead of events, to enable society to be better prepared for them.

The focus the research in this project was to develop an environmental digital twin which brings together computational models of flood inundation with other data, for hazard assessment, management, and mitigation. The developed digital twin was designed to enable the automation of flood risk assessment, such that multiple scenarios can be assessed rapidly. This may be particularly useful in response to updated information, such as in response to observed river levels or forecast rainfall (i.e., emergency management), or for regular updates given new data such as new scenarios (e.g., climatic), or new built infrastructure. The digital twin we have envisaged can bring together and processes the data needed for flood risk assessment and use these for scenario assessment within computational modelling. The digital twin can then analyse the impact of these scenarios and update them given new information. Such a digital twin would enable flood risk assessments to be completed more rapidly and at lower cost, and will facilitate detailed, standardised risk assessments at the national scale. The main purposes the flood resilience digital twin are to (i) automate the process of developing pluvial and fluvial models, (ii) capture and analyse topographical and infrastructure data to model inundation and flow information in an urban setting, and (iii) assess the impact of inundation on infrastructure.

The fundamental design principle of our work is that the digital twin should broadly adopt the “Findable, Accessible, Interoperable and Reusable” (FAIR) principles (Ivánová et al., 2019). In particular, the digital twin should be:

- ***Open***: the digital twin should be open by default; it should use and contribute to existing open-source libraries and be released as an open-source codebase to be built on. The data sources ingested by default should be open.
- ***Extendable***: the software libraries used within the digital twin should selected partly based on the possibilities of future work; for example, 3D visualisations which will enable future development of augmented and virtual reality applications.
- ***Interoperable***: using established data and API standards and best practices, the digital twin should be accessible by other server systems or digital twins, enabling it to form part of a future ecosystem of interconnected digital twins, rather being independent.
- ***Replicable***: rather than limited to one site as with many existing digital twins, for which significant additional effort is required to transfer to different sites, the digital twin should be designed to “self-generate” for sites of user interest, initially for anywhere in New Zealand but with the possibility of being replicable globally.
- ***Dynamic***: the hydraulic modelling engine needs to be embedded within the digital twin, through automated exchange of data and modelling results, enabling simulations to be run on demand.

- ***Versatile***: the digital twin should be agnostic towards which hydraulic modelling engine is selected, meaning that additional modelling engines can be embedded in future, for improvements or model benchmarking; it should be straightforward to add additional data sources to the digital twin, including those which may be provided by other twins.
- ***Scalable***: the methods used need to be efficient, with multiple scales of simulations possible and deployable on cloud-computing, while keeping track of and storing data and simulations performed to avoid unnecessary repetition of the same processing.

While ambitious, our research has been underpinned by methods developed in an aligned research programme, Mā te haumaru ō te wai, led by NIWA. This programme is assessing flood risk across New Zealand. Both research programmes have benefited from the engagement: the digital twin project has utilised the software libraries for LiDAR processing and the hydraulic modelling engine developed by the NIWA team, the digital twin team has in turn contributed improvements back to the upstream code. Furthermore, the standardising of the methods used between the two programmes will enable to the digital twin to act as a possible communication tool by Mā te haumaru ō te wai; it additionally democratises access to the assessment of additional scenarios, which could be run on demand rather than these being centralised.

The design principles above were developed in consultation with stakeholders from key organisations involved in flood risk management or industry. This consultation primarily consisted of two workshops which are summarised in the next section. Further input was provided as part of conference presentations (e.g., to the conferences of the Association of Local Government Information Managers, Stormwater New Zealand, and the UN World Geospatial Information Management Congress) and digital twin discussion workshops organized by FrontierSI and the Digital Twin Partnership.

3 Development workshops

Engagement and design workshops were held at the start of the project in January 2021 and a follow-up workshop in January 2022. The format and outcomes of these workshops are summarised here.

To initiate the project a Flood Interoperability Workshop was run which brought together a wide range of professionals from local council, engineering lifelines companies, engineering consultancies and researchers. The purpose of the workshop was to develop a proof-of-concept digital twin to assess the risk of urban flooding. The purpose of such an urban flood digital twin was specified to:

1. Automate the process of developing pluvial and fluvial models.
2. Capture and analyse topographical and infrastructure data to model inundation and flow information in an urban setting.
3. Assess the impact of inundation on infrastructure.

This workshop was designed to test the usability of infrastructure and environmental data to meet the outcomes stated above. The ultimate aim was to ensure data interoperability that will enable the full development of the digital twin a local to national scales. Furthermore, the workshop was run as a precursor to the aligned NIWA-led research programme, Mā te haumaru ō te wai, which aims to assess and mitigate flood inundation hazard and risk across Aotearoa/New Zealand.

The workshop focussed on the effects of a significant fluvial/ pluvial flood event on an urban area's physical infrastructure. Other concerns such as public safety, social, environmental and economic impacts of such an event was out of scope. Kaiapoi was chosen as the physical location to study based

on its moderate size, good variety of infrastructure in its boundaries, data availability for the region and the proximity to the workshop location.

The participants were chosen based on their expertise and ability to provide a good cross-section of the various agencies likely to be involved should such a flood event occur. The timing of the invites and the event likely impacted the number of people who attended. However, interest was high, and enough people with the right skills and background responded to make the event a success.

The workshop outputs included:

1. A prototype digital twin that modelled a flood scenario in Kaiapoi.
2. Identifying which data was missing, where these data may be found and areas where other techniques such as Artificial Intelligence (AI) may be needed to fill in those data.
3. A clear articulation of the issues that need to be addressed in the final model.
4. A validation of the prototyping methodology using the Feature Manipulation Engine (FME) software.
5. An understanding of the importance of data standards in the development process.

Based on the outcomes of the workshop, the most appropriate technology stack while enabling the digital twin to be open source was identified, and the information architecture for the digital twin was designed.

A follow up workshop was held a year later in January 2022, to update stakeholders on the progress made and to get their inputs on the future directions of code development. Key outcomes were:

1. The ability to generate multiple scenarios, particularly those which are user-specified, would be considered to be high value.
2. More complex geospatial analyses which could be addressed were identified, such as transport routing during a flood event, would be useful.
3. Additional dynamic data sources were identified, including rain radar and climate forcing data.
4. While a visualisation engine was welcomed, it was considered that most agencies would want to use their existing systems to present data.
5. Organisations would like to obtain model results and download them for use in their own risk models, which again highlighted the importance of flexibility in the software design.
6. The best approach to achieve a flexible system which is widely adopted was identified as via an Application Programming Interface (API), enabling organisations to connect their existing systems to the digital twin.

Following this feedback, the development timeline was extended and adjusted. In particular, an API was added to the design and the code was adjusted to reflect this.

4 Spatial domain for prototype development

To facilitate software development and stakeholder engagement, the digital twin was developed and tested as a prototype for the town of Kaiapoi in Canterbury, 12 km north of Christchurch (Figure 1). However, under the design principles the digital twin was built so that it can ingest data as needed, to “self-generate” for any location in New Zealand. With the prototype system completed, follow-on work will enable further development and deployment nationwide.

The Kaiapoi site provided a good test case for the prototype development since, in addition to being local to the development team, it has a complex flood risk, including high fluvial flood from the Waimakariri River which flows into the area from the west, fluvial flood risk from smaller rivers which flow through the town, pluvial risks from intense rainfall events, and coastal flood risk from high tides and storm surges. Further, there is a realistic prospect of each of these types of flooding occurring simultaneously, in compound flooding. The area is additionally extremely low relief (Figure 2), meaning that the prediction of flood extents and depths requires high precision.

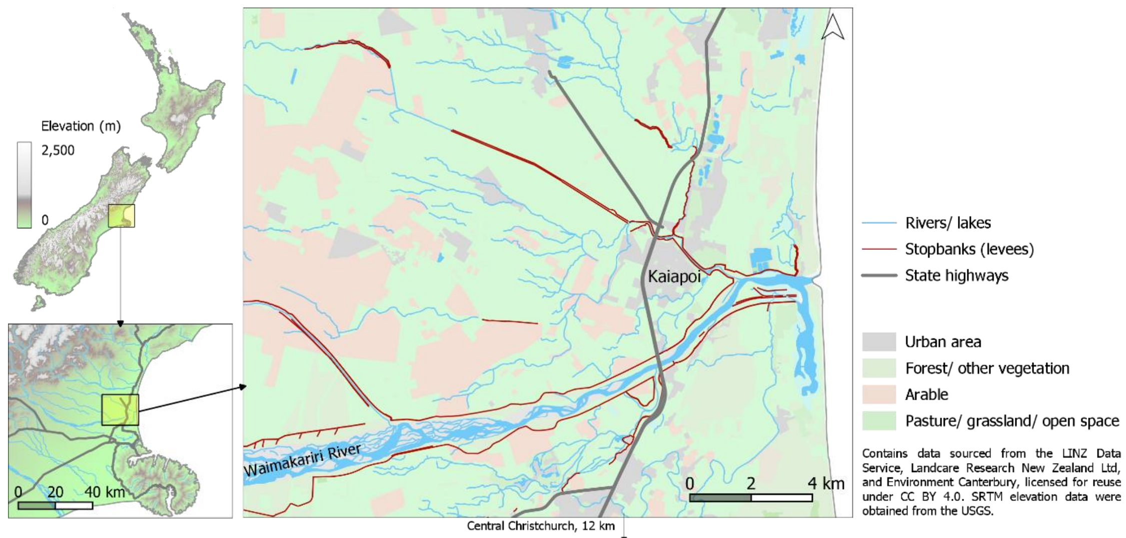


Figure 1: Kaiapoi location and situation to the north of the lower Waimakariri River. The town is protected from flooding by a stopbank (levee) system and pumping stations.

Figure 2: The area around Kaiapoi has extremely low topographic relief making it susceptible to widespread flooding. Areas of the town are part of a "red-zone" which have been retreated from due to liquefaction during the 2010-11 Canterbury earthquakes, which increased flood risk.

5 Flood Resilience Digital Twin: Software summary

In this section, an overview of the data processing (back-end system) is provided at a high level. For details of the software implementation, please refer to the Appendices which include documentation for the API and descriptions of all functions. The visualisation (front-end) is summarised in Section 6.

5.1 Conceptual design

The conceptual design of the digital twin developed at project inception (Figure 3), includes geospatial databases to hold the input data, including static data used for boundary conditions (e.g., LiDAR data, stopbanks, storm water, channel geometry and land cover), dynamic boundary data used for model water inputs (e.g., river gauge data, tide levels, rainfall data, design flows), observational data for model comparison (e.g., airborne flood imagery, SAR flood imagery, survey data) and data for infrastructure and population for impact assessment (e.g., buildings, census data, roads and rail). A design principle for the digital twin was that, for a selected area, these data are downloaded from their respective agencies and stored in a local database without modification (i.e., features are kept intact). A local copy of the data is maintained to avoid the high bandwidth which may be associated with multiple downloads, as well as speed up processing when the data are used for multiple analyses. To achieve this, a database of the metadata is maintained, which tracks areas which have been previously processed, and when.

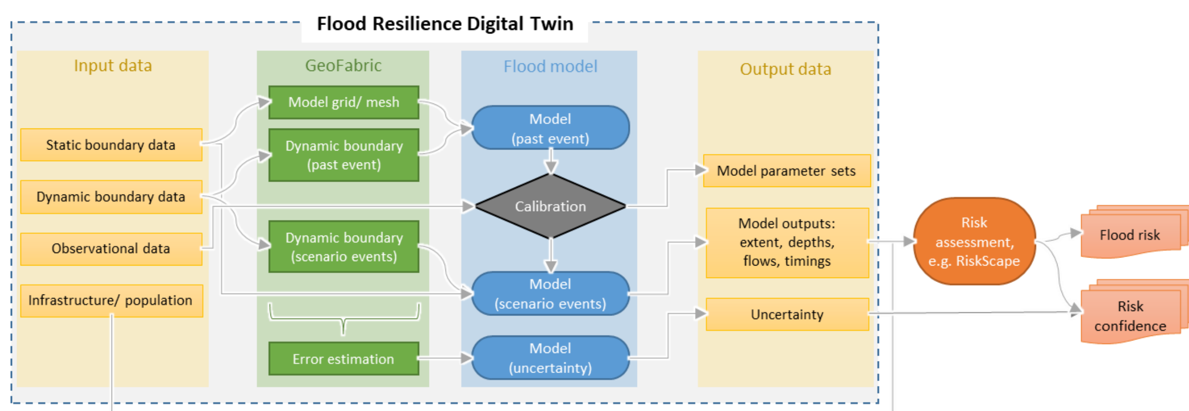


Figure 3: Flood resilience digital twin conceptual diagram

The items in the digital twin conceptual diagram (Figure 3) which are yet to be implemented are:

- Input data: observational data of flooding (e.g., from satellite imagery) and population data (e.g., from the census) are yet to be included.
- GeoFabric: error estimation based on the data processing to enable model uncertainty estimation.
- Flood model: calibration, which requires observational data of flood extent, and uncertainty analysis.
- Output data: model parameter sets and uncertainty not included.
- Post processing: impacted infrastructure identified by a full risk assessment needed, e.g., with a connection to RiskScape software.

Each of these will be implemented in future work. However, the current version has full end-to-end capability for flood risk assessment, as illustrated by Figure 4. A user specifies an area of interest (AOI) and a supported scenario, directly or by using the front end. In either case, an API request is sent to

the back-end system and the pipeline of processing commences. First, the system will check the database to determine whether data have already been obtained for the AOI. Only if they are missing will data be requested from remote servers, thereby avoiding excessive load. Periodically, the system will check providers for updated datasets. The system will also check whether a simulation for this AOI and scenario has already been performed. If it has, it will be able to quickly serve that simulation back to the user without the need to run the BG-Flood software (Bosserelle, 2018; Bosserelle et al., 2022). If the scenario has not been run, model input files will be generated, and the simulation run. For the Kaiapoi study site used in the prototype, each simulation takes around 30 seconds for a 2-day flood scenario. One of the key advantages of BG-Flood is that it will facilitate the scaling of the system to larger sites, as the software is designed to take advantage of GPU computing.

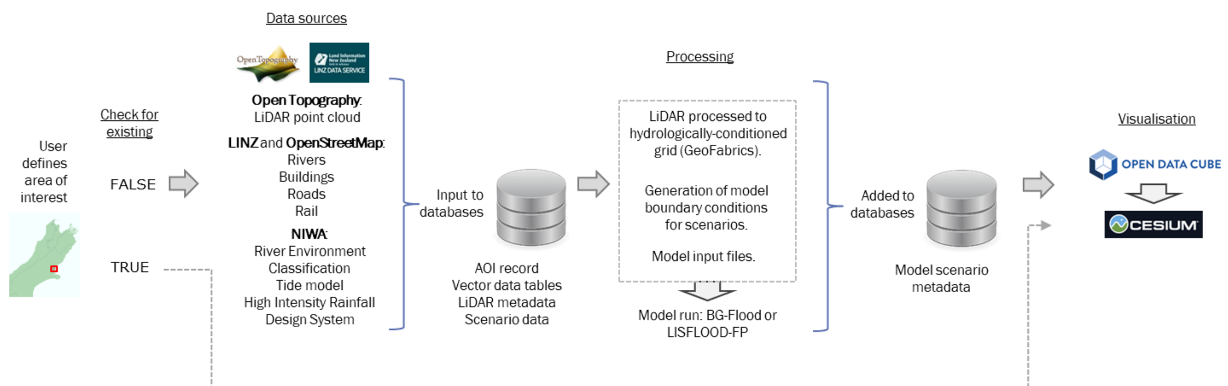


Figure 4. Overview of the current data processing and flood assessment processing pipeline.

5.2 Current implementation

In the current version, the list of included data is presented in Figure 5. After download and incorporation into local databases, data are further processed within the digital twin to create the “GeoFabric”, which refers to the model inputs and boundary conditions, including the scenarios assessed, in the appropriate model format (in this case, BG-Flood). The model is then run, and the digital twin maintains metadata regarding simulations. Finally, model results are incorporated back into the digital twin for analysis, such as identifying flooded buildings.

Currently, the software we have developed integrates spatial and other data from multiple vendors into databases, then extracts and processes the data for the automated simulation of a range of possible flood scenarios, using the BG-Flood hydraulic model. The data processing includes the extraction and conversion of LiDAR point cloud data into hydrologically conditioned digital elevation models, processed using the GeoFabrics Python package (R. Pearson, 2021; R. A. Pearson et al., 2023) (Figure 6 and Figure 7). These are then converted into the formats required to run BG-Flood, along with scenarios derived from statistically generated pluvial and fluvial boundary conditions.

To run the flood model, dynamic boundary condition data are obtained which represent flows into the domain and a downstream tide level (if the site is coastal) (see Figure 8 for example inputs). Rainfall and river flow input data are derived statistically from existing databases which have been generated based on historical observations. For rainfall, the High Intensity Rainfall Design System (HIRDS)[1] from NIWA is used, which enables depth-duration-frequency statistics to be obtained for rain

[1] https://hirds.niwa.co.nz

gauges across the country and provides estimates of future depths under different scenarios of climate change. The digital twin maintains a database of all sites for which these estimates are available and uses Thiessen polygons to determine the aerial coverage of each gauge. When a user selects an AOI, its extent is intersected with these polygons, and the data for each gauge needed for the requested scenario are obtained from the HIRDS server.

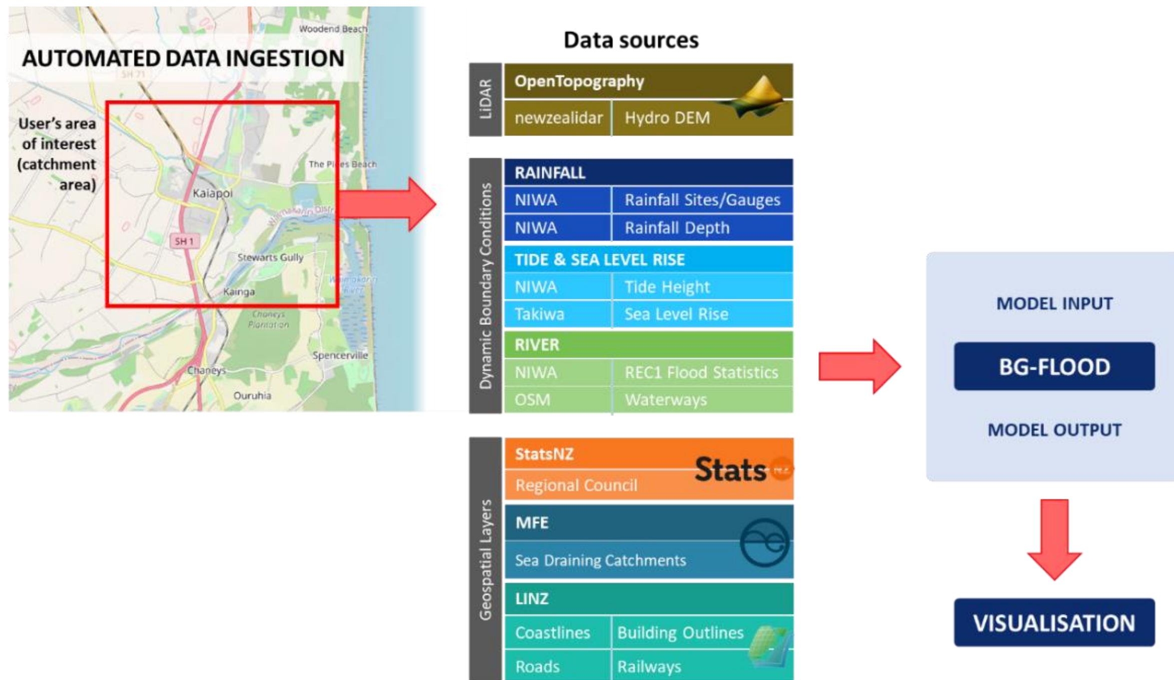


Figure 5: Data sources included in the current version of the digital twin. All are nationally available data and provide sufficient functionality to assess flood risk but may be supplemented by local data through extension of data specifications.

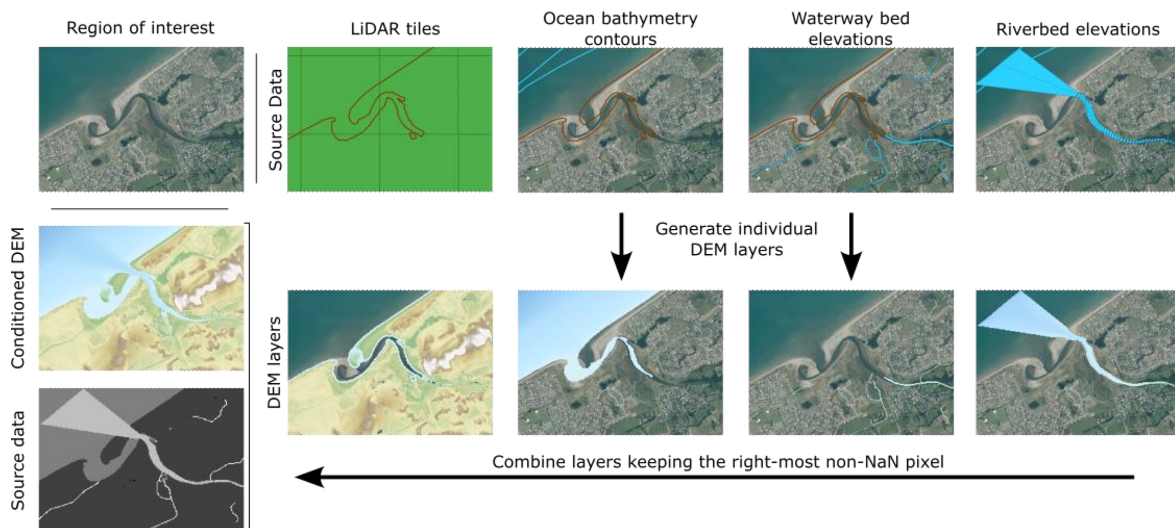


Figure 6. The GeoFabrics library (R. Pearson, 2021) is used to process LiDAR data to create a hydrologically conditioned DEM suitable for use in the BG-Flood model (Figure: R. A. Pearson et al., 2023).

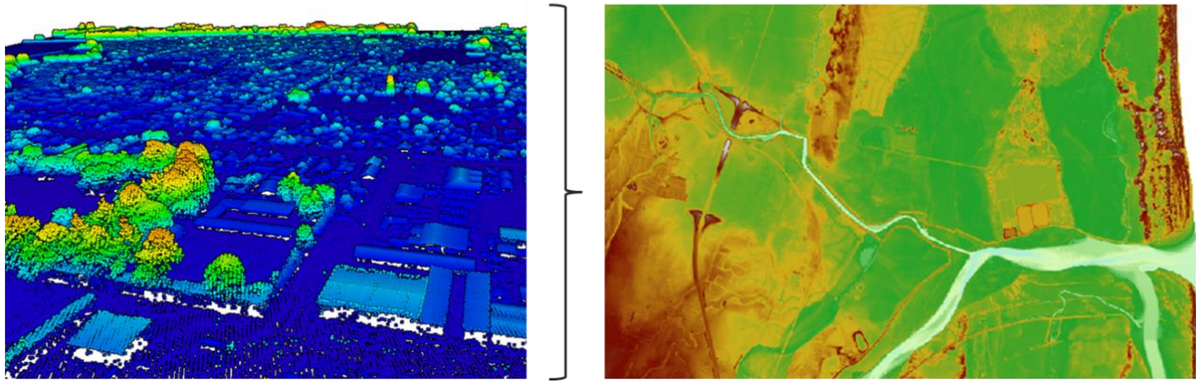


Figure 7. LiDAR point cloud data are downloaded by the digital twin from OpenTopography and processed into a model grid using GeoFabrics software, which removes surface features which would affect flow and ensures that a suitable bathymetry is assigned to river channels and coastal areas.

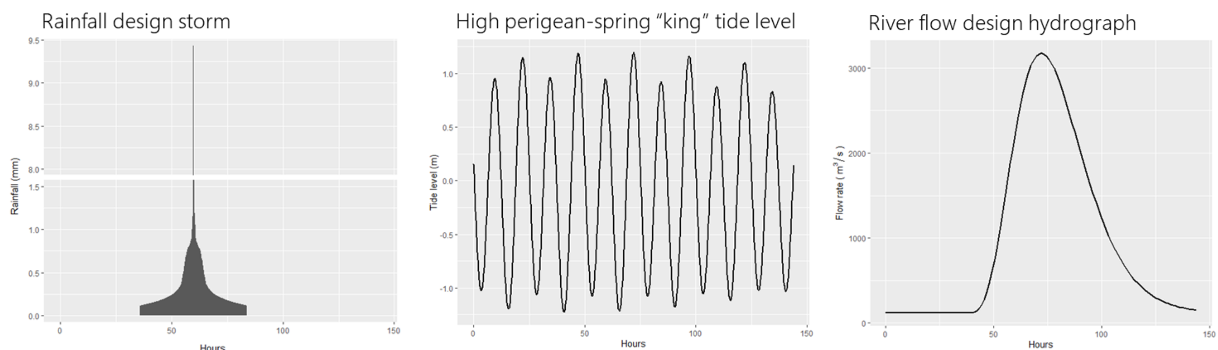


Figure 8. Dynamic boundary conditions are statistically derived for rainfall, tide levels and river flows, based on analysis of rainfall and flow levels by NIWA, and the annual maximum tide for the location selected (if coastal).

For river flow, data from NIWA's River Environment Classification (REC)[2], version 1 are used. These data contain flood level estimates for different annual exceedance probabilities, for river vector data across the country. However, climate scenarios are missing, and the data are further limited since the river vectors were derived from 90 m resolution topographic data (SRTM), meaning that their locations contain significant errors and they often do not exactly align with rivers in the LiDAR data. While a new version of the REC data is available, these do not contain flood level estimates. To be able to use the data, the digital twin needs to: identify which river vectors cross the boundary of the AOI requested, determine whether the river crosses the boundary multiple times and select the one furthest downstream, then search in the local neighbourhood for the correct location of the river based on the LiDAR elevation data. Data for each vector needed are then obtained from REC1 data for the requested scenario, and a hydrography generated using a design hydrograph approach. In future versions of the digital twin, improvements to the river network data and inflow statistics would be beneficial.

For the tide level, by default the annual maximum high tide for the last year is obtained from NIWA's Tide Forecaster[3] for the closest available location. A limitation is that these data do not contain observations of sea level, and the predicted tide will be different to actual due to weather conditions

[2] https://catalogue.data.govt.nz/dataset/flood-statistics-2018-rec1

[3] https://tides.niwa.co.nz/

particularly those associated with storm surges. In future versions of the digital twin, alternative sources of sea-level data may be included. The peaks of the rainfall, river flow and high tide data are temporally aligned within the scenarios assessed in the digital twin, and should therefore be considered a worst-case scenario, in which the tide has the greatest possible impact on river flooding.

To account for sea level rise in the scenarios, differing levels of sea level can be included on top of the tide level. The digital twin will obtain scenarios for the requested tidal location from the NZ SeaRise project[4]. This dataset accounts for both vertical land movement and sea level rise and provides data for multiple climate scenarios with projections out to the year 2300.

For the requested scenarios, the digital twin will run BG-Flood, record the necessary metadata for future retrieval of the model simulation. The digital twin will detect once the model has completed, then send the results in the requested format using the API. If this was the front-end system making the request, the results will be stored sent to the GeoServer so that they can be ingested into the Cesium platform. An example of model output for an extreme flood scenario is shown in Figure 9, which shows three time slices from the model scenario. The frequency of outputs can be adjusted depending on the level of detail required.

The digital twin will additionally generate spatial intersections with data of the built infrastructure to enable the assessment of impact. Figure 10 illustrates two examples for buildings and roads, for the maximum flood depth which occurred during the simulation. While these examples present the overall impact, the full dynamic flood data are also available from the model, meaning that the impacts can be assessed as they evolve during the flood event. Additional types of critical infrastructure can be assessed for flood impact, including specifying whether a depth threshold is reached. An example analysis which is possible is the assessment of depth inundation against critical depth thresholds for infrastructure such as electrical transformer stations, which would be useful for electricity grid management during flood events. A further example is that the road-depth analysis can be extended based on vehicle type, with automated re-routing of vehicles based on a road network analysis which accounts for a real-time estimation of likely flood impacts. These examples were identified during our engagement workshops with stakeholders.

While extensions to the analyses are possible internally in the digital twin software, its flexible framework provides organisations with the opportunity determine their own analyses of model outputs, by using the API to obtain model simulations and intersecting them with their own data. This also overcomes potential issues of data sovereignty, since not all data need to be included in the digital twin database.

Our development roadmap (see Section 8) includes the automated assessment of the impact of the predicted flooding, based on depth/ flow damage relationships, using a dynamic connection with RiskScape software. In addition, in future work we plan to continue to develop more advanced visualisations including the use of AR/VR.

[4] https://searise.takiwa.co

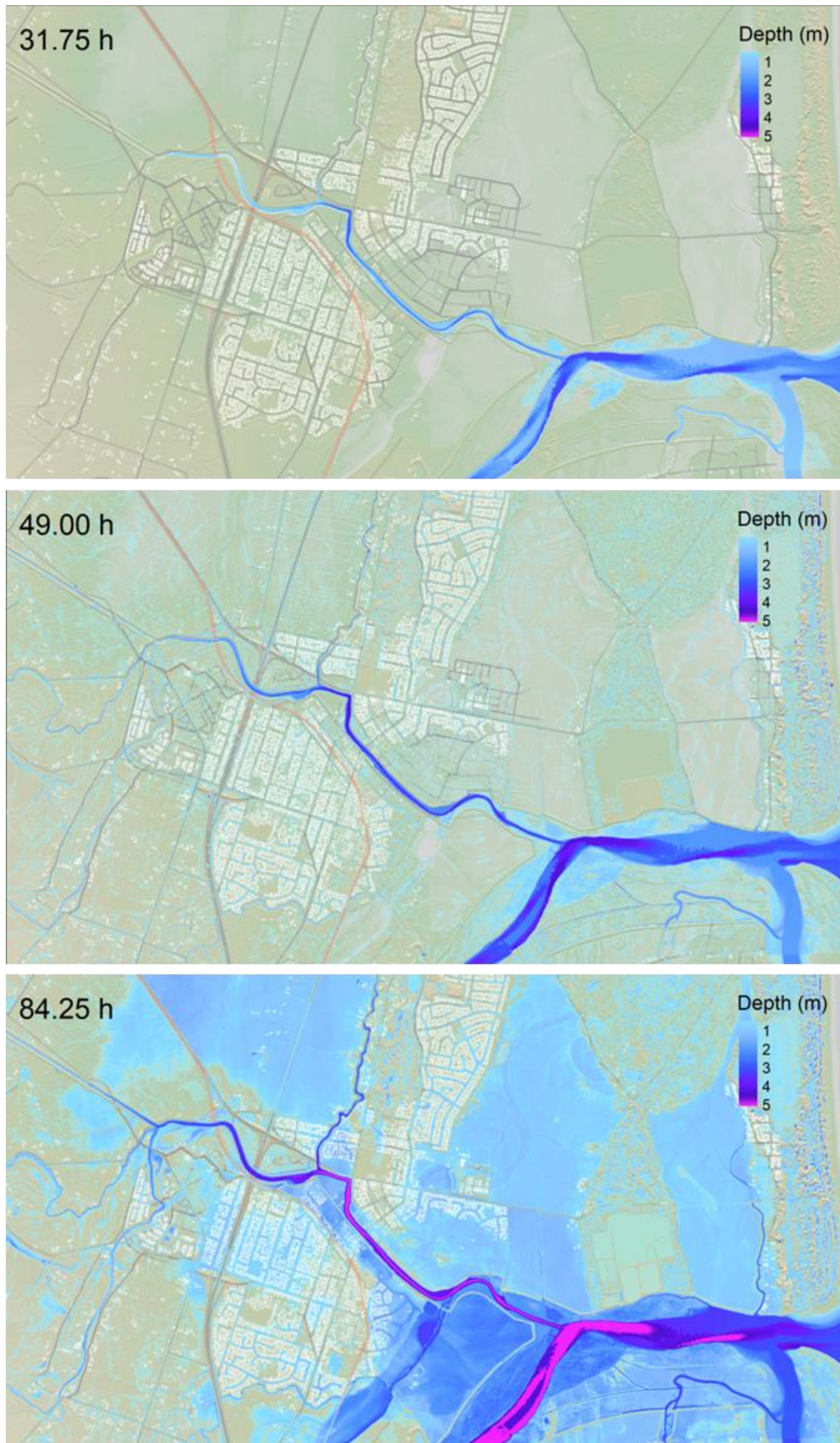


Figure 9. Example flood model depth predictions for an extreme flood scenario: (top) before the flood with normal tidal inflows, (middle) at the onset of heavy rainfall, and (bottom) during compound flooding with very high tide and river flows coinciding.

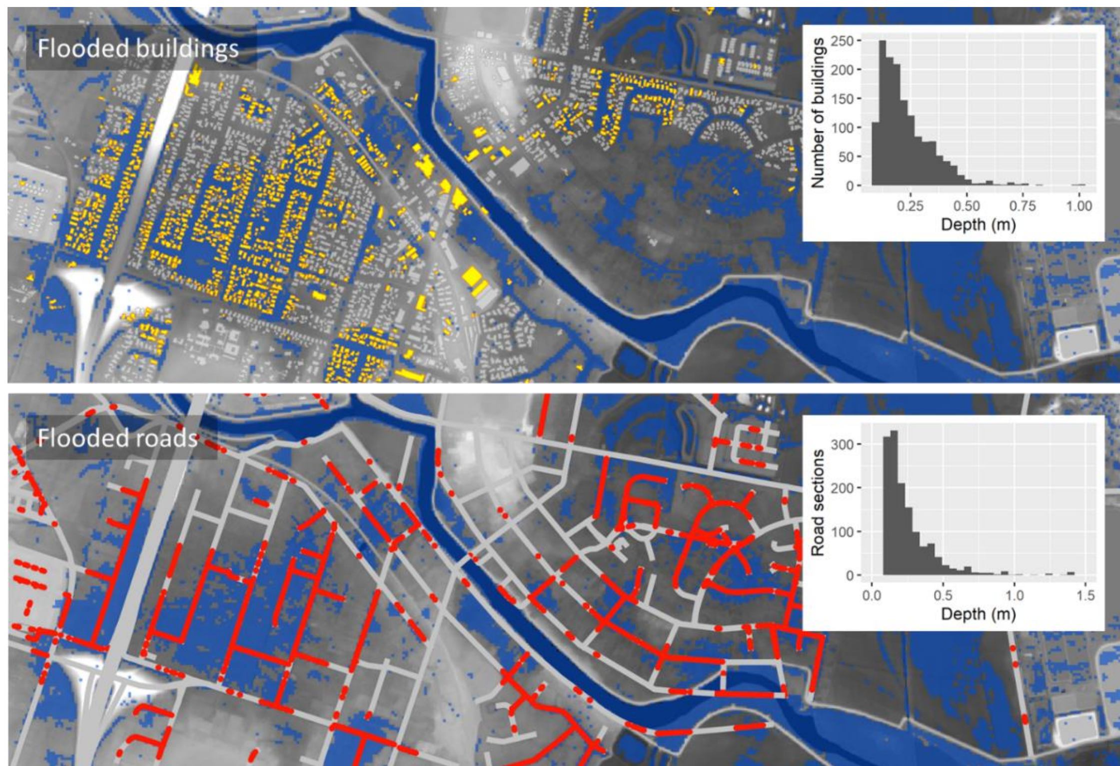


Figure 10. Examples of the analysis which is possible after the completion of the BG-Flood simulation: (top) maximum inundation depth intersected with buildings, identifying those flooded at 0.5 m depth or greater; (bottom) roads which are flooded and therefore present a risk to vehicles are identified.

5.3 Current limitations

There are several limitations in the current implementation of the digital twin, partly due to data availability, and partly resulting from limitations in the hydraulic modelling software engine used, BG-Flood. The main known limitations are provided in Table 1. Some of these are included in the current development pathway (see Section 8). Limitations which were already part of the existing development pathway are not included, such as functionality included in the conceptual design (Figure 1) but which were not coded within the current version (e.g., a dynamic connection to RiskScape software).

Table 1. Known limitations of the current flood resilience digital twin and possible solutions

Limitation	Implications and possible pathways to solutions	Estimated difficulty
Dynamic flood management infrastructure such as pumping stations are not included.	This may lead to a possible over-prediction of flooding for some areas. These data are not readily available, and the BG-Flood software does not currently support internal transfer of water, meaning that it is non-trivial to implement possible solutions. Alternative approaches may be to extend the BG-Flood software functionality or include alternative model software engines, such as using the approach of Cui et al. (2023), noting that these may bring different limitations such as limitations in computational efficiency of licencing restrictions. To overcome data availability issues, the most flexible approach would be to enable organisations to upload their own data to the digital twin, although this will create a significant	High

	additional layer of complexity since user-account management and security would need to be enabled.	
Stormwater drainage pipe infrastructure is not included.	While infrastructure such as stopbanks are included, and surface drainage channels are included implicitly within the LiDAR derived grid, underground stormwater pipe infrastructure is not included. This may lead to over-prediction for areas with significant stormwater drainage pipe infrastructure. The data are available nationally through an aligned programme of the Building Innovation Partnership, but the BG-Flood software does not currently include sub-surface pipe flow. As with pumping stations, consideration is needed as to whether the software is extended, and by what method (see Bulti & Abebe, 2020, for a review). However, the most flexible approach is likely to be to dynamically couple BG-Flood to an existing model such as the 1D Storm Water Management Model such as in the research by Yang et al. (2020).	Medium
Stopbank (levee) breaching is not included.	As is common in flood modelling, it is assumed that stopbanks do not fail (breach) during a flood event, so flooding only occurs where they overtop. This may lead to an underprediction of flooding, and can create a false sense of security in communities (Gissing et al., 2018; Hutton et al., 2019). While the failure mechanisms for levees are complex (Pol et al., 2023), the BG-Flood software supports levee breaching and scenarios can be included. Estimated software development difficulty is low as only a moderate extension of the digital twin software would be needed. However, the breaching methods to be used would need to be identified.	Low
Scenarios are based on statistics which may not be representative of current flood risk, and simplified dynamic boundary conditions used	Existing statistical analyses of rainfall and river flows were obtained for use in the digital twin, which greatly simplified the generation of appropriate alternative scenarios. Estimated flood peak levels were translated to a standardised design hydrography which enables a representation of a flood wave but is unlikely to represent actual river flows. The most flexible approach to overcoming these limitations will be to enable users to specify their own boundary conditions for testing, while providing further developing the default scenarios to be more realistic, such as through the development of an improved regional flood frequency analysis (Guo et al., 2023; Smith et al., 2015; e.g., Wright et al., 2020). In addition, a real-time connection to observational data is not currently enabled, and could include rain/river/tide gauges, weather model data, and rain radar data, to enable real-time or forecast flood prediction.	Low to medium
River network limitations	In the current digital twin implementation, NIWA's River Environment Classification (REC) version 1 data are used. These have a highly generalised spatial representation of the locations of the river networks, leading to additional complexity of the processing to ensure that the injection points for water entering the model domain are placed in the appropriate locations (i.e., in rivers within the model grid which is derived from LiDAR). Furthermore, there are "rivers" within the REC which appear to be artefacts of the processing, rather than existing, and these need to be identified and removed (the current implantation looks for connectivity to the downstream network). While there is are newer versions of the REC data, they lack the flood frequency statistics needed for scenario generation.	Medium

6 Flood Resilience Digital Twin: Visualisation frontend system

To complement the data processing and computational engine, which forms the foundation of the digital twin, we have developed a web-based visualisation environment based on the Cesium open platform for 3D geospatial data[5]. While the primary focus has been on the development of the backend system, with engagement with stakeholders suggesting that this was the correct approach, the implementation of a frontend system has been important to facilitate communication, both with stakeholders and during conference presentations. At the start of the project, it soon became apparent that diagrams of database systems were not sufficient, and additional developer resources were brought into the project to enable a frontend system. In this section, a summary of the work completed and remaining is provided.

The CesiumJS[6] library was selected because it is widely used in other digital twins, such as Digital Twin Victoria[7] which is built using the TerriaJS[8] library that incorporates CesiumJS. We have developed the current digital twin front end using the CesiumJS library directly, since it allows for greater flexibility to code future advanced functionality such as AR/VR connections. We have further experimented with the use of Open Data Cube (ODC) software[9] to enable temporal point query of simulation results. While the ODC is primarily designed for remote sensing software, the fundamental data structure of the model outputs and remote sensing imagery is the same: multi-temporal raster image stacks.

The software here is at a lower level of maturity to the backend system. It forms part of a wider-scoped software project within the Geospatial Research Institute to develop 3D visualisation applications[10], and some of the code developed for the digital twin will be integrated into that library. As part of this development, functionality is being developed for a user interface (UI) which will enable, at a minimum, selection of the AOI and standardised scenarios to be run.

The frontend system and its connection to the backend with the current software environment is illustrated in Figure 11. CesiumJS is used for the primary visualisation engine. For most visualisations, data are requested from GeoServer, into which spatial layers from flood simulations, and their intersections with built infrastructure, are inserted. This enables 3D visualisation of flood depth, with impacts to infrastructure highlighted. For example, Figure 12 shows the maximum flood depth during the simulation with flooded buildings highlighted.

Flood scenarios are currently saved in the native format of the model, within the server file system, with metadata maintained about the simulation to ensure that simulations are not repeated if already done. In the case of the BG-Flood model used here, the model results are stored in the netcdf scientific data format. To include all model outputs within the frontend visualisation, we have experimented with the use of the ODC engine, which primarily facilitates temporal query between each flood layer (i.e., each epoch or timeslice produced by the model, the period of which may be user-defined). This has enabled the frontend to include temporal point-query as illustrated in Figure 13. When a user clicks on a location within the model domain, the stack of raster flood depth layers can be queried at that location and the resulting temporal depth vector displayed as a time-series plot.

---

[5] https://cesium.com/
[6] https://cesium.com/platform/cesiumjs/
[7] https://vic.digitaltwin.terria.io/
[8] https://terria.io/
[9] https://www.opendatacube.org/
[10] https://github.com/GeospatialResearch/geo-visualisation-components

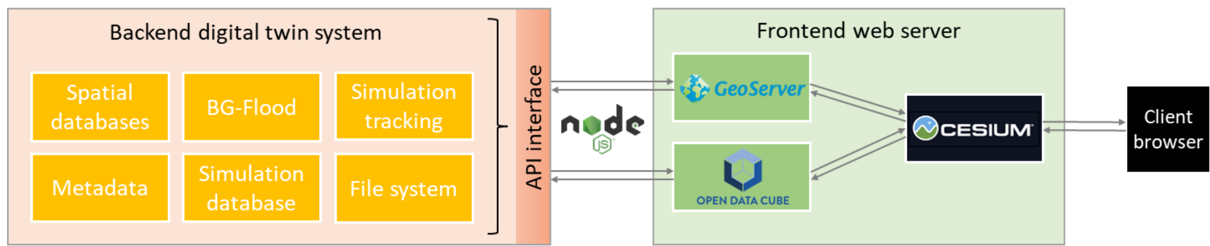


Figure 11. Sketch of the frontend system and its relationship to the backend.

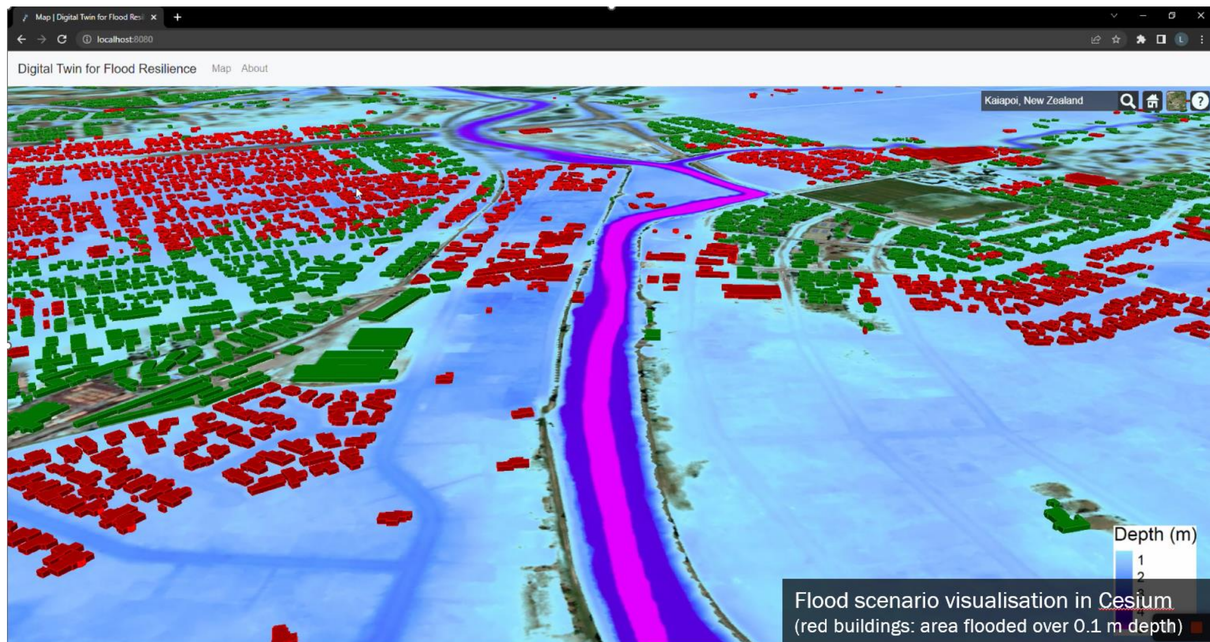


Figure 12. The default 3D visualisation is for the maximum flood depth during the scenario. Here, buildings are highlighted in red if they are flooded at a depth of 0.1 m or greater.

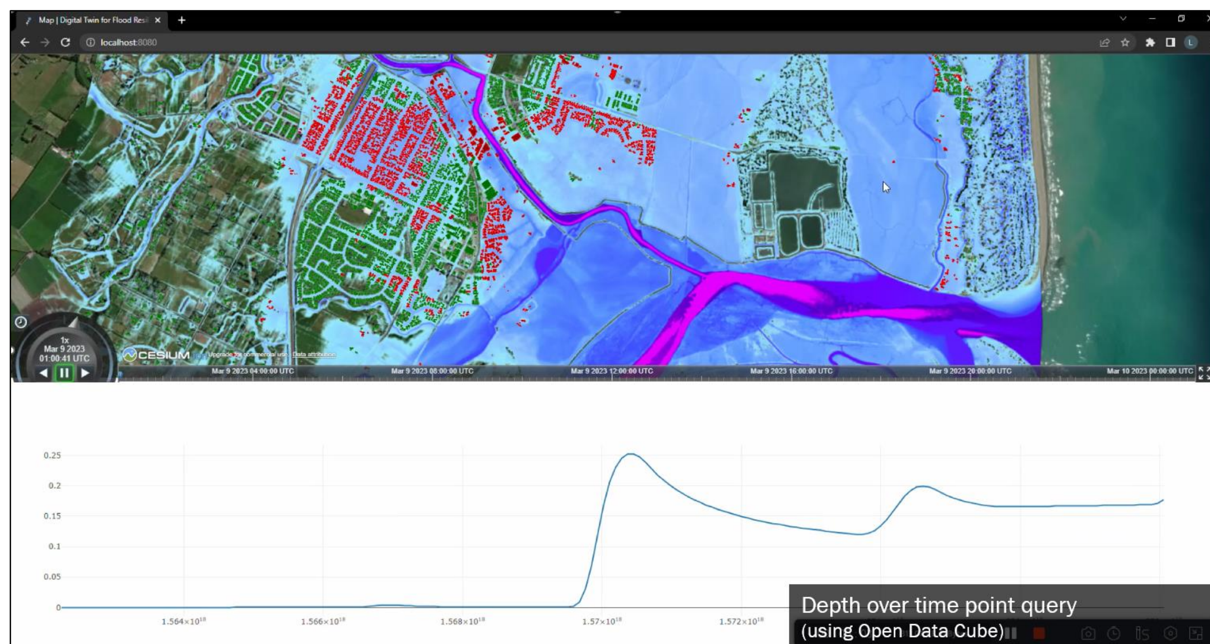


Figure 13. The use of the Open Data Cube facilitates user point-query of flood depth over time, extracting data for locations from the stack of predicted flood depths. In this case, the tidal influence on predicted flood depths is clear.

Since the flood resilience digital twin can run multiple scenarios, alongside the point query functionality for depth over time for a selected location, we have experimented with enabling a slider to compare between two different scenarios. While the functionality is still experimental, this will facilitate comparison between scenarios for issues such as the assessment of the impact of sea level rise on flood inundation, as illustrated in Figure 14. The user can move the slider between two scenarios to observe the difference made in changing a variable, such as the increase in the level of the downstream boundary condition in this case. The functionality still needs to be extended to work with a 3D view of the data, and to be integrated with the point-query functionality illustrated in Figure 13.

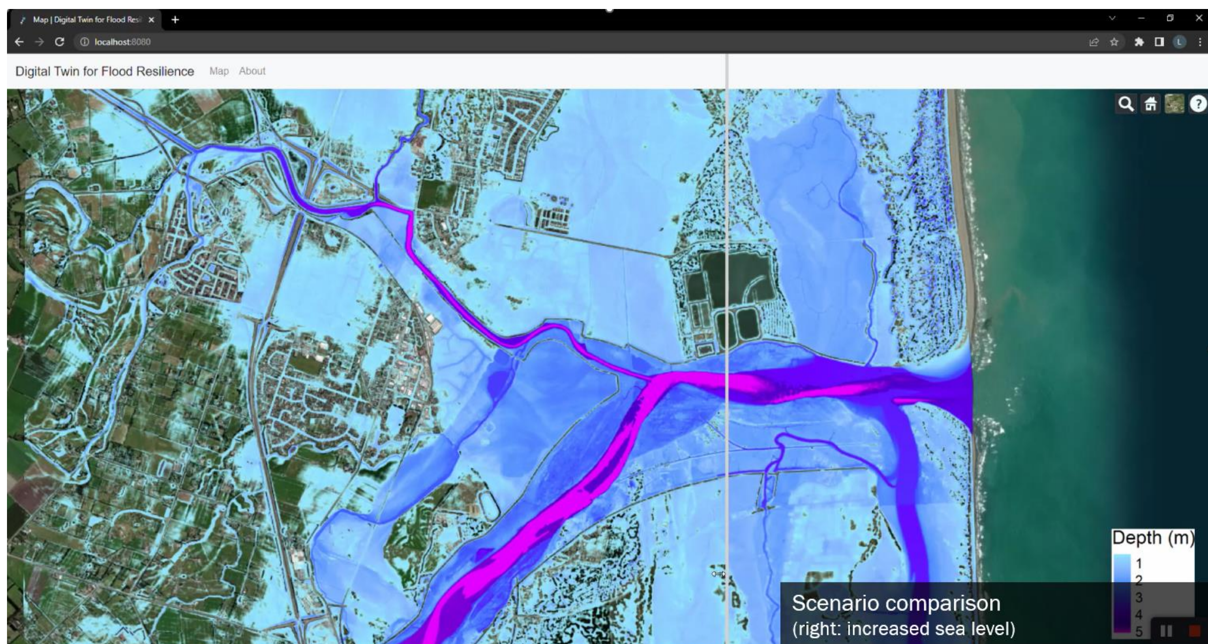


Figure 14. Comparison between two flood scenarios: (left of slider) baseline flood scenarios for requested flood likelihood; (right of slider) the same likelihood but with an added scenario of sea-level rise, leading to increased compound flooding.

The frontend environment is still under active development and there are several issues to be addressed. For example, a limitation of selecting a depth threshold to highlight impact is that buildings are assumed to be at the ground level. However, buildings are usually raised to some extent, known as the floor level, which provides some protection of the interior of the building against flooding. Shallow flooding is not likely to cause damage to the building, and this should be accounted for in the impact assessment. To address this, the depth threshold variable can be dynamically updated through the user interface, or data for flood levels could be used on a per-building basis to identify if the flood level is at or approaching this level. However, data on building flood levels are frequently not available publicly.

Secondly, currently missing from the visualisation are flood water flow rates, and the hazard levels caused by combined flood depth and flow rates. These data are available from the BG-Flood software and visualisations can be developed with only limited additional coding to the frontend system. This would also enable a point-query of the likely impact of a flood event, for example to identify buildings which are at greater risk of destruction during the flood event.

Another current limitation is that the full dynamics of the flood scenario are not currently captured using animation of predicted flood depths. While this is relatively straightforward to achieve in 2D, a dynamic animation of flood depth in 3D is more complex. We are currently exploring how we will include animations of requested scenarios in CesiumJS and this will be added to the code in a future version. It is likely that using the Cesium Markup Language (CZML) for temporally dynamic data, combined with a mesh which represents the water surface elevation, will be an appropriate approach, similar to the approach implemented by Kumar et al. (2018).

Finally, the software has been envisaged to be a flexible system which can adapt to the environment of end users. Cesium was selected as it fits with this functionality and already supports multiple virtual reality engines, including the Unity[11] and Unreal Engine[12] environments, which can enable future development of immersive visualisations for improved communication of flood risk. In addition, it connects with NVIDIA's Omniverse, which enables 3D visual effects that can enable much more realistic virtual environments.

7 Key lessons and challenges

The Flood Resilience Digital Twin was designed and developed to be reproducible for any location of interest within New Zealand by end users, through a process of "self-generation" once given an AOI by the user. The software was developed to automatically obtain the data needed from suppliers to generate, process them as needed, simulate flood scenarios, and assess flood risk. There was no available "off-the-shelf" example to build from nor was there an existing codebase to adopt; but digital twin we have created was only made possible by the availability of the data and software libraries which underpin it.

7.1 Data

New Zealand is acknowledged to have a strong open data foundation (World Wide Web Foundation, 2018), and this is a key reason why the creation of this digital twin was possible within the country. Substantial amounts of critical data are needed for flood risk assessment, including high-accuracy terrain data, river networks data, observations of river flow, rainfall and tide levels, and data for the built infrastructure. Of particular significance for this research is the national LiDAR programme which was initialised and managed by LINZ to provide NZD$19M of co-funding for LiDAR data collection to councils through a provincial growth fund[13]. Although this has now completed, it has led to around 80% of the country being covered by highly accurate topographic data[14], all available under an open licence. Before this programme, it would not have been possible to have developed the digital twin under its "self-generation" design philosophy: it would have to be restricted to the relatively few locations with available data.

Other data in the digital twin which are sourced from LINZ under an open data licence include buildings, roads, and other infrastructure. These open data are critical to enable technological developments such as the digital twin created in this research. They act as an enabler to these

[11] https://unity.com/
[12] https://www.unrealengine.com/en-US
[13] https://www.linz.govt.nz/products-services/data/types-linz-data/elevation-data/provincial-growth-fund-lidar-data-collection-now-progress
[14] https://www.linz.govt.nz/products-services/data/types-linz-data/elevation-data

developments for each of the different benefits they can bring. However, data standards are also critical to these developments: they facilitate the development of digital twins by greatly reducing the work required to use data from different sources, which need to be standardised to be useful. The digital twin here was developed using only nationally available datasets with clearly specified standards. This meant that some attributes were not possible to bring into the twin. An important example of this is the buildings data used: the current LINZ dataset only provides the building outline, which is a fundamental property for the use of the data. However, the data miss important information such as floor heights, and other properties including the building structure. These data are not available in a consistent, standardised way, which greatly increases the amount of work which is needed to develop digital twins.

LINZ has operated a programme which identifies priority datasets for resilience and climate change (LINZ, 2023), which each will be provided under open licencing for the benefit of the country. Many of these data sources (e.g., buildings, roads, and other infrastructure) will be directly useable by the existing digital twin with only minor updates to the data source definition table. Some of the issues present in current data (e.g., lack of building heights and other attributes) will be addressed, though not floor levels. An improved river network based on LiDAR data will also become available, though this will likely not include the flood frequency statistics needed. The fundamental challenge however remains the ingestion and standardisation of the data which underpin digital twins.

While these national datasets are an exemplar of best practice in open-data governance, at the sub-national level there is considerable variability in both the availability and the standards used for critical datasets. Many councils across New Zealand release data under an open licence (there are nearly 33,000 datasets on the data.govt.nz data catalogue[15]). While this is welcome, there is a lack of standardisation between councils regarding formats, the platforms used to host datasets differ, and not all councils release the same types of data. This makes the task of combining data into a consistent database laborious and places a significant barrier to the creation of technology which can build on these data for wide benefits. A good example of the collation of critical data such as observations of water quantity across the country is Land, Air and Water Aotearoa (LAWA)[16]. This site provides a live statistical summary of rainfall and river flow gauges across the country, operated by different regional councils; but the data are limited to only a recent period and are not available programmatically. Each council runs their own separate site for data distribution, with differing levels of access and different standards. NIWA operates its own, separate sites, with access provided via an API on its Hydro Web Portal[17], though its coverage is very limited.

This fragmented system for access to dynamic observational data creates a barrier to the implementation of technologies such as digital twins. This can be contrasted with the river gauge network operated by the United States Geological Service, which runs National Water Dashboard[18] through which all data are available under an open licence, and the National Water Model[19], operated by the National Oceanic and Atmospheric Administration, which provides open data access to river flow forecasting for short (1 day), medium (10 day) and long (30 day) ranges (Johnson et al., 2023).

[15] https://data.govt.nz/
[16] https://www.lawa.org.nz/explore-data/canterbury-region/water-quantity/
[17] https://hydrowebportal.niwa.co.nz/
[18] https://dashboard.waterdata.usgs.gov/app/nwd/en/?region=lower48&aoi=default
[19] https://water.noaa.gov/map

7.2 Open-source software

The digital twin for flood resilience was built on open-source code libraries: it has only been possible through the sharing of these codes. Similarly, in other projects open-source software is driving the development of digital twins (Barnstedt et al., 2021). However, we are further contributing back to the community by releasing the digital twin software as a fully open-source solution which can be used by others as a foundation for developing other digital twins, to either adopt our codebase or learn from the methods used within our software.

Although we have built on open source, we have needed to develop much code from scratch, specifically designed to be functional for the purpose intended. While there are software libraries orientated at digital twins which are released with open-source licencing (e.g., Ditto[20], iTwinJS[21] and Azure Digital Twins[22]), these are often limited in scope and do not support complex modelling such as those required by computational modelling. This may change with the development of a new domain working group for "Geo for Metaverse" within the Open Geospatial Consortium[23].

7.3 Physics-based Digital Twins

A key design principle we chose to adopt was that the digital twin should not only visualise and assess model predictions but also run the simulations. This means that the digital twin can run new simulations if given updated data or new scenarios, and follows the "self-generation" design principle, in that simulations do not need to be run ahead of digital twin deployment. Crucially, this allows the digital twin to make accurate predictions outside the observational record, such as an extreme flood event, since its predictions are grounded in physics. The concept of a physics-based digital twin has received increased research attention, particularly if operating in a hybrid mode where a machine learning algorithm either helps to improve data used in the computational model, or learns from the model itself to help make rapid predictions (Ritto & Rochinha, 2021; e.g., Sun & Shi, 2022). Away from the digital twin concept, machine learning is being integrated with physics-based models to enable more detailed, faster predictions of extremely large and complex systems such as the Earths climate, while remaining physically realistic (Kashinath et al., 2021; for example, Rolnick et al., 2023; Willard et al., 2020).

Arguably, a model of the climate system can be considered as a comprehensive digital twin, but it is one which should be connected to other local twins which can pull relevant data into their analyses. A network of digital twins is required, which each share data between them using standardised communication protocols. Work such as Twinbase (Autiosalo et al., 2021) towards the creation of a "Digital Twin Web" likely points the way to the future. This is further emphasized by the creation of Destination Earth (DestinE) digital twin programme of the European Union: "*By coupling the DestinE digital twins with […] highly specialized and localized [digital twin] tools through standardised interfaces many decision-making processes at local level will be able to benefit from the advanced capabilities that DestinE provides*" (Hoffmann et al., 2023).

[20] https://eclipse.dev/ditto/index.html
[21] https://www.itwinjs.org/
[22] https://azure.microsoft.com/en-us/products/digital-twins/
[23] https://www.ogc.org/press-release/ogc-announces-new-geo-for-metaverse-domain-working-group/

8 Future work

We have demonstrated a successful prototype digital twin with a specialisation on flood resilience. For the next phase of development, we will seek to address some of the limitations detailed in Section 5.3. An outline of the road map for development is given below.

Backend functionality:

- Additional scenarios will be included, and a mechanism to enable users to adjust scenarios or specify data to use will be developed.
- Connections to real-time data will be included, enabling the digital twin to be responsive.
- Connections to weather forcing data will be developed, such as those available from the ECMWF, and in preparation for the development of the DestinE system.
- A storm drainage system will be developed, integrating existing pipe data, and the necessary model engine produced, improving the representativeness of model scenarios.
- Machine learning methods will be integrated into the digital twin, for both uncertainty estimation of the predicted modelling and to enable rapid hybrid modelling based (these topics are currently the subject of research by PhD students within the GRI).
- Connection to the RiskScape software will be developed, enabling a more comprehensive assessment of flood impacts.

Frontend/ visualisations:

- An improved and integrated user interface will be developed, with a greater level of control provided.
- Dynamic visualisations will be developed, enabling animations in 3D within the Cesium environment.
- Immersive visualisations in VR and/or AR will be developed, enabling greater public engagement for flood risk education.

Deployment:

- Develop a hosted digital twin solution, based on scalable cloud computing to enable model simulations.
- Scale and test across New Zealand.
- Longer term, develop the digital twin in other countries, including a globally applicable version of the digital twin.

9 References

Ali, K., Bajracharya, R. M., & Koirala, H. Lal. (2016). A review of flood risk assessment. *International Journal of Environment, Agriculture and Biotechnology*, *1*(4), 1065–1077. https://doi.org/10.22161/ijeab/1.4.62

Alperen, C. I., Artigue, G., Kurtulus, B., Pistre, S., & Johannet, A. (2021). A Hydrological Digital Twin by Artificial Neural Networks for Flood Simulation in Gardon de Sainte-Croix Basin, France. *IOP Conference Series: Earth and Environmental Science*, *906*(1), 012112. https://doi.org/10.1088/1755-1315/906/1/012112

Ariyachandra, M. R. M. F., & Wedawatta, G. (2023). Digital twin smart cities for disaster risk management: A review of evolving concepts. *Sustainability*, *15*(15), 11910. https://doi.org/10.3390/su151511910

Arnell, N. W., & Gosling, S. N. (2016). The impacts of climate change on river flood risk at the global scale. *Climatic Change*, *134*(3), 387–401. https://doi.org/10.1007/s10584-014-1084-5

Autiosalo, J., Siegel, J., & Tammi, K. (2021). Twinbase: Open-Source Server Software for the Digital Twin Web. *IEEE Access : Practical Innovations, Open Solutions*, *9*, 140779–140798. https://doi.org/10.1109/ACCESS.2021.3119487

Barnstedt, E., Boss, B., Clauer, E., Isaacs, D., Lin, S. W., Malakuti, S., van Schalkwykm, P., & Martins, T. W. (2021). Open source drives digital twin adoption. *IIC J. Innov*.

Bates, P. D., Mason, D., Neelz, S., Pender, G., Villaneuva, G., & Wilson, M. D. (2004). A framework for flood inundation modelling. In *Flood risk assessment* (pp. 169–178). Institute of Mathematics and its Applications, Southend-on-Sea, UK.

Bauer, P., Stevens, B., & Hazeleger, W. (2021). A digital twin of Earth for the green transition. *Nature Climate Change*, *11*(2), 80–83. https://doi.org/10.1038/s41558-021-00986-y

Blair, G. S. (2021). Digital twins of the natural environment. *Patterns (New York, N.Y.)*, *2*(10), 100359. https://doi.org/10.1016/j.patter.2021.100359

Bosserelle, C., Lane, E. M., & Harang, A. (2022). BG-Flood: A GPU adaptive, open-source, general inundation hazard model. In *Australasian Coasts & Ports 2021: Te Oranga Takutai, Adapt and Thrive: Te Oranga Takutai, Adapt and Thrive* (pp. 152–158). New Zealand Coastal Society Christchurch, NZ.

Bosserelle, C. (2018). *GitHub: CyprienBosserelle/BG_Flood*. https://github.com/CyprienBosserelle/BG_Flood

Brunner, G. W. (2002). *HEC-RAS (river analysis system)*. 3782–3787.

Bulti, D. T., & Abebe, B. G. (2020). A review of flood modeling methods for urban pluvial flood application. *Modeling Earth Systems and Environment*, *6*(3), 1293–1302. https://doi.org/10.1007/s40808-020-00803-z

Cui, Y., Liang, Q., Xiong, Y., Wang, G., Wang, T., & Chen, H. (2023). Assessment of Object-Level Flood Impact in an Urbanized Area Considering Operation of Hydraulic Structures. *Sustainability*, *15*(5), 4589. https://doi.org/10.3390/su15054589

Deren, L., Wenbo, Y., & Zhenfeng, S. (2021). Smart city based on digital twins. *Computational Urban Science*, *1*(1), 4. https://doi.org/10.1007/s43762-021-00005-y

European Commission. (2022, March 31). *Destination Earth (DestinE)*. https://digital-strategy.ec.europa.eu/en/policies/destination-earth

Ford, D. N., & Wolf, C. M. (2020). Smart Cities with Digital Twin Systems for Disaster Management. *Journal of Management in Engineering*, *36*(4), 04020027. https://doi.org/10.1061/(ASCE)ME.1943-5479.0000779

Fuller, A., Fan, Z., Day, C., & Barlow, C. (2020). Digital twin: enabling technologies, challenges and open research. *IEEE Access : Practical Innovations, Open Solutions*, *8*, 108952–108971. https://doi.org/10.1109/ACCESS.2020.2998358

Ghaith, M., Yosri, A., & El-Dakhakhni, W. (2021). *Digital Twin: A city-scale flood imitation framework*. CSCE 2021 Annual Conference.

Gissing, A., Van Leeuwen, J., Tofa, M., & Haynes, K. (2018). Flood levee influences on community preparedness: a paradox?. *Australian Journal of Emergency Management, The*, *33*(3), 38–43.

Glas, H., De Maeyer, P., Merisier, S., & Deruyter, G. (2020). Development of a low-cost methodology for data acquisition and flood risk assessment in the floodplain of the river Moustiques in Haiti. *Journal of Flood Risk Management*, *13*(2). https://doi.org/10.1111/jfr3.12608

Guo, S., Xiong, L., Chen, J., Guo, S., Xia, J., Zeng, L., & Xu, C.-Y. (2023). Nonstationary regional flood frequency analysis based on the bayesian method. *Water Resources Management*, *37*(2), 659–681. https://doi.org/10.1007/s11269-022-03394-9

Ham, Y., & Kim, J. (2020). Participatory Sensing and Digital Twin City: Updating Virtual City Models for Enhanced Risk-Informed Decision-Making. *Journal of Management in Engineering*, *36*(3), 04020005. https://doi.org/10.1061/(ASCE)ME.1943-5479.0000748

Hoffmann, J., Bauer, P., Sandu, I., Wedi, N., Geenen, T., & Thiemert, D. (2023). Destination Earth – A digital twin in support of climate services. *Climate Services*, *30*, 100394. https://doi.org/10.1016/j.cliser.2023.100394

Hutton, N. S., Tobin, G. A., & Montz, B. E. (2019). The levee effect revisited: Processes and policies enabling development in Yuba County, California. *Journal of Flood Risk Management*, *12*(3), e12469. https://doi.org/10.1111/jfr3.12469

Ivánová, I., Brown, N., Fraser, R., Tengku, N., & Rubinov, E. (2019). Fair and standard access to spatial data as the means for achieving sustainable development goals. *ISPRS - International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences*, *XLII-4/W20*, 33–39. https://doi.org/10.5194/isprs-archives-XLII-4-W20-33-2019

Jiang, P., Meinert, N., Jordão, H., Weisser, C., Holgate, S., Lavin, A., Lütjens, B., Newman, D., Wainwright, H., Walker, C., & Barnard, P. (2021). Digital Twin Earth -- Coasts: Developing a fast and physics-informed surrogate model for coastal floods via neural operators. *ArXiv*. https://doi.org/10.48550/arxiv.2110.07100

Johnson, J. M., Blodgett, D. L., Clarke, K. C., & Pollak, J. (2023). Restructuring and serving web-accessible streamflow data from the NOAA National Water Model historic simulations. *Scientific Data*, *10*(1), 725. https://doi.org/10.1038/s41597-023-02316-7

Jones, D., Snider, C., Nassehi, A., Yon, J., & Hicks, B. (2020). Characterising the Digital Twin: A systematic literature review. *CIRP Journal of Manufacturing Science and Technology*, *29*, 36–52. https://doi.org/10.1016/j.cirpj.2020.02.002

Kashinath, K., Mustafa, M., Albert, A., Wu, J. L., Jiang, C., Esmaeilzadeh, S., Azizzadenesheli, K., Wang, R., Chattopadhyay, A., Singh, A., Manepalli, A., Chirila, D., Yu, R., Walters, R., White, B., Xiao, H., Tchelepi, H. A., Marcus, P., Anandkumar, A., … Prabhat. (2021). Physics-informed machine learning: case studies for weather and climate modelling. *Philosophical Transactions. Series A, Mathematical, Physical, and Engineering Sciences*, *379*(2194), 20200093. https://doi.org/10.1098/rsta.2020.0093

Kumar, K., Ledoux, H., & Stoter, J. (2018). Dynamic 3d visualization of floods: case of the netherlands. *ISPRS - International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences*, *XLII-4/W10*, 83–87. https://doi.org/10.5194/isprs-archives-XLII-4-W10-83-2018

LINZ. (2023). *Key Data for Resilience and Climate Change: Data Improvement Plan 2023/24* (No. A5879500). Land Information New Zealand.

Madni, A., Madni, C., & Lucero, S. (2019). Leveraging Digital Twin Technology in Model-Based Systems Engineering. *Systems*, *7*(1), 7. https://doi.org/10.3390/systems7010007

Manocha, A., Sood, S. K., & Bhatia, M. (2023). Digital Twin-assisted Fuzzy Logic-inspired Intelligent Approach for Flood Prediction. *IEEE Sensors Journal*, 1–1. https://doi.org/10.1109/JSEN.2023.3322535

McDermott, T. K. J. (2022). Global exposure to flood risk and poverty. *Nature Communications*, *13*(1), 3529. https://doi.org/10.1038/s41467-022-30725-6

Ocio, D., Stocker, C., Eraso, Á., Martínez, A., & de Galdeano, J. M. S. (2016). Towards a reliable and cost-efficient flood risk management: the case of the Basque Country (Spain). *Natural Hazards*, *81*(1), 617–639. https://doi.org/10.1007/s11069-015-2099-6

Pearson, R. (2021). *GitHub: rosepearson/GeoFabrics*. https://github.com/rosepearson/GeoFabrics

Pearson, R. A., Smart, G., Wilkins, M., Lane, E., Harang, A., Bosserelle, C., Cattoën, C., & Measures, R. (2023). GeoFabrics 1.0.0: An open-source Python package for automatic hydrological conditioning of digital elevation models for flood modelling. *Environmental Modelling & Software*, *170*, 105842. https://doi.org/10.1016/j.envsoft.2023.105842

Pol, J. C., Kindermann, P., van der Krogt, M. G., van Bergeijk, V. M., Remmerswaal, G., Kanning, W., Jonkman, S. N., & Kok, M. (2023). The effect of interactions between failure mechanisms on the reliability of flood defenses. *Reliability Engineering & System Safety*, *231*, 108987. https://doi.org/10.1016/j.ress.2022.108987

Riaz, K., McAfee, M., & Gharbia, S. S. (2023). Management of climate resilience: exploring the potential of digital twin technology, 3D city modelling, and early warning systems. *Sensors (Basel, Switzerland)*, *23*(5). https://doi.org/10.3390/s23052659

Ritto, T. G., & Rochinha, F. A. (2021). Digital twin, physics-based model, and machine learning applied to damage detection in structures. *Mechanical Systems and Signal Processing*, *155*, 107614. https://doi.org/10.1016/j.ymssp.2021.107614

Rolnick, D., Donti, P. L., Kaack, L. H., Kochanski, K., Lacoste, A., Sankaran, K., Ross, A. S., Milojevic-Dupont, N., Jaques, N., Waldman-Brown, A., Luccioni, A. S., Maharaj, T., Sherwin, E. D., Mukkavilli, S. K., Kording, K. P., Gomes, C. P., Ng, A. Y., Hassabis, D., Platt, J. C., … Bengio, Y. (2023). Tackling Climate Change with Machine Learning. *ACM Computing Surveys*, *55*(2), 1–96. https://doi.org/10.1145/3485128

Ruangpan, L., Mahgoub, M., Abebe, Y. A., Vojinovic, Z., Boonya-Aroonnet, S., Torres, A. S., & Weesakul, S. (2023). Real time control of nature-based solutions: Towards Smart Solutions and Digital Twins in Rangsit Area, Thailand. *Journal of Environmental Management*, *344*, 118389. https://doi.org/10.1016/j.jenvman.2023.118389

Semeraro, C., Lezoche, M., Panetto, H., & Dassisti, M. (2021). Digital twin paradigm: A systematic literature review. *Computers in Industry*, *130*, 103469. https://doi.org/10.1016/j.compind.2021.103469

Smith, A., Sampson, C., & Bates, P. (2015). Regional flood frequency analysis at the global scale. *Water Resources Research*, *51*(1), 539–553. https://doi.org/10.1002/2014WR015814

Sun, C., & Shi, V. G. (2022). PhysiNet: A combination of physics-based model and neural network model for digital twins. *International Journal of Intelligent Systems*, *37*(8), 5443–5456. https://doi.org/10.1002/int.22798

Suquet, R. R., Nguyen, T. H., Ricci, S., Piacentini, A., Bonassies, Q., Fatras, C., Lavergne, E., Brunato, S., Gaudissart, V., Guzzonatto, E., Froidevaux, A., Guiot, A., Valladeau, G., Poisson, J. C., Huang, T., Bretar, F., Kettig, P., & Blanchet, G. (2023). The SCO-Flooddam Project: Towards A Digital Twin for Flood Detection, Prediction and Flood Risk Assessments. *IGARSS 2023 - 2023 IEEE International Geoscience and Remote Sensing Symposium*, 1000–1003. https://doi.org/10.1109/IGARSS52108.2023.10282907

Tarpanelli, A., Bonaccorsi, B., Sinagra, M., Domeneghetti, A., Brocca, L., & Barbetta, S. (2023). Flooding in the digital twin earth: the case study of the enza river levee breach in december 2017. *Water*, *15*(9), 1644. https://doi.org/10.3390/w15091644

Thieken, A., Merz, B., Kreibich, H., & Apel, H. (2006). Methods for flood risk assessment: Concepts and challenges. *… Workshop on Flash Floods in …*.

Tsakiris, G. (2014). Flood risk assessment: concepts, modelling, applications. *Natural Hazards and Earth System Science*, *14*(5), 1361–1369. https://doi.org/10.5194/nhess-14-1361-2014

Willard, J., Jia, X., Xu, S., Steinbach, M., & Kumar, V. (2020). Integrating physics-based modeling with machine learning: A survey. *ArXiv Preprint ArXiv:2003.04919*, *1*(1), 1–34.

Winsemius, H. C., Van Beek, L. P. H., Jongman, B., Ward, P. J., & Bouwman, A. (2013). A framework for global river flood risk assessments. *Hydrology and Earth System Sciences*, *17*(5), 1871–1892. https://doi.org/10.5194/hess-17-1871-2013

World Wide Web Foundation. (2018). *Open Data Barometer - Leaders Edition*. World Wide Web Foundation. https://opendatabarometer.org/doc/leadersEdition/ODB-leadersEdition-Report.pdf

Wright, D. B., Yu, G., & England, J. F. (2020). Six decades of rainfall and flood frequency analysis using stochastic storm transposition: Review, progress, and prospects. *Journal of Hydrology*, *585*, 124816. https://doi.org/10.1016/j.jhydrol.2020.124816

Yang, Y., Sun, L., Li, R., Yin, J., & Yu, D. (2020). Linking a Storm Water Management Model to a Novel Two-Dimensional Model for Urban Pluvial Flood Modeling. *International Journal of Disaster Risk Science*, *11*(4), 508–518. https://doi.org/10.1007/s13753-020-00278-7

10 Appendix A: Software installation

All code is available at: https://github.com/GeospatialResearch/Digital-Twins. The documentation below is a drawn from the README.md file[24], which details the steps needed to install the software (using Docker) and get the databases setup.

10.1 Introduction

According to the National Emergency Management Agency, flooding is the greatest hazard in New Zealand, in terms of frequency, losses and civil defence emergencies. With major flood events occurring on average every 8 months (New Zealand – FloodList), it is necessary to produce high precision flood models and in order to do better planning, risk assessment and response to flood events, making plans in advance can make all the difference, not just to property owners at risks, it will also help insurance companies who make underwriting decisions on properties, the banks supplying the property finance, the telecommunications and utilities companies providing vital services to homes and offices, and the government agencies tasked with protecting communities and their assets. Digital Twin can provide a better understanding of the degree of impact flood events can have on physical assets like buildings, roads, railways, transmission lines, etc. Digital Twin is a real-time digital clone of a physical device. Anyone looking at the digital twin can see crucial information about how the physical thing is doing out there in the real world. Digital Twin not only represents the current status of the visualised assets but also how they will perform/react to future situations. The build twin when used to run flood models combined with other sources of information can allow us to make predictions. The first step of building a Digital Twin is data. Data is collected from an open data portal provided by multiple organisations or data providers such as LINZ, LRIS, stat NZ, ECAN, opentopography, NIWA, etc. The collected data is stored in the local database using PostgreSQL which is an open-source relational database system and supports both SQL (relational) and JSON (non-relational) querying. PostgreSQL is a highly stable database backed by more than 20 years of development by the open-source community. Spatial data is the primary data used for implementing the Digital Twin model. Therefore, PostgreSQL with the PostGIS extension which supports geospatial databases for geographic information systems (GIS) is the most preferable DBMS for this project. Also, it provides support for Python, C/C++ and JavaScript, the programming languages used for building Digital Twin. The spatial boundaries are currently limited to New Zealand with the potential of getting extended further. The reason for creating a database are: 1. Avoid unnecessary network overhead on the data providers 2. To avoid delays in fetching the same data from the API when required again and again to run the models. 3. To store the data only for the Area of Interest.

The Digital Twin stores API details and a local copy of data for the required Area of Interest provided by LINZ, ECAN, Stats NZ, KiwiRail, LRIS, opentopography, and NIWA in PostgreSQL.

10.2 Basic running instructions

The following list defines the basic steps required to setup and run the digital twin.

10.3 Requirements

- Docker
- Anaconda

[24] https://github.com/GeospatialResearch/Digital-Twins/blob/master/README.md

- [Node.js / NPM]

10.4 Required Credentials:

- Stats NZ API Key
- LINZ API Key
- Cesium access token

10.5 Starting the Digital Twin application (localhost)

1. Set up Docker, Anaconda, and NPM to work on your system.
2. In the project root, in an Anaconda prompt, run the following commands to initialise the environment:

   ```
   #!/usr/bin/env bash
   conda env create -f environment.yml
   conda activate digitaltwin
   ```

   While this is running, you can continue with the other steps until using the environment.
3. Create a file called `.env` in the project root, copy the contents of `.env.template` and fill in all blank fields.
4. Set any file paths in `.env` if needed, for example `FLOOD_MODEL_DIR` references a Geospatial Research Institute network drive, so you may need to provide your own implementation of `BG_flood` here.
5. Create a file `visualisation/.env.local`. In this, fill in `VUE_APP_CESIUM_ACCESS_TOKEN=[your_token_here]`, replace `[your_token_here]` with the Cesium Access Token
6. From project root, run the command `docker-compose up --build -d` to run the database, backend web servers, and helper services .
7. Currently, the `visualisation` and `celery_worker` services are not set up to work with Docker, so these will be set up manually.
   1. In one terminal, with the conda environment activated, go to the project root directory and run `celery -A src.tasks worker --loglevel=INFO --pool=solo` to run the backend celery service.
   2. In another terminal open the `visualisation` directory and run `npm ci && npm run serve` to start the development visualisation server.
8. You may inspect the logs of the backend in the celery window.
9. You may inspect the PostgreSQL database by logging in using the credentials you stored in the `.env` file and a database client such as `psql` or pgAdmin.

10.6 Using the Digital Twin application

1. Visit the address shown in the visualisation server window, default http://localhost:8080
2. To run a flood model, hold SHIFT and hold the left mouse button to drag a box around the area you wish to run the model for.
3. Once the model has completed running, you may need to click the button at the bottom of the screen requesting you to reload the flood model.

4. To see a graph for flood depths over time at a location, hold CTRL and click the left mouse button on the area you wish to query.

10.7 Setup for developers

10.7.1 Run single Docker service e.g. database

To run only one isolated service (services defined in `docker-compose.yml`) use the following command: `docker-compose up --build [-d] [SERVICES]`

e.g. To run only the database in detached mode:

```
#!/usr/bin/env bash
docker-compose up --build -d db_postgres
```

10.7.2 Create Conda environment

Setup a conda environment to allow intelligent code analysis and local development by using the following command run from the repository folder:

10.7.3 Run Celery locally

This set is recommended, since BG Flood does not yet work on Docker. With the conda environment activated run:

```
#!/usr/bin/env bash
celery -A src.tasks worker --loglevel=INFO --pool=solo
```

10.7.4 Running the backend without web interface.

For local testing, it may be useful to use the `src.run_all.py` script to run the processing.

10.8 Tests

Tests exist in the `tests/` folder.

10.8.1 Automated testing

Github Actions are used to run tests after each push to remote (i.e. github). Miniconda from the GitHub Actions marketplace is used to install the package dependencies. Linting with Flake8 and testing with PyTest is then performed. Several tests require an API key. This is stored as a GitHub secret and accessed by the workflow.

10.8.2 Running tests locally

See the geoapis wiki testing page for instructions for setting up a .env file and running the geofabrics test.

10.9 Vector Database

To store api details of vector data in the database, The following inputs are required:

1. Name of the dataset e.g. 104400-lcdb-v50-land-cover, 101292-nz-building-outlines. **Note:** make sure the names are unique.
2. Name of the region which is by default set to New Zealand but can be changed to regions e.g. Canterbury, Otago, etc. (regions can be further extended to other countries in future)

3. Geometry column name of the dateset, if required. for instance, for all LDS property and ownership, street address and geodetic data the geometry column is 'shape'. For most other layers including Hydrographic and Topographic data, the column name is 'GEOMETRY'. For more info: https://www.linz.govt.nz/data/linz-data-service/guides-and-documentation/wfs-spatial-filtering
4. Url i.e website from where the api is accessed.
5. Layer name of the dataset
6. Data provider name. For example: LINZ, LRIS, StatsNZ, etc. For more details on the format and structure of the inputs check out instructions_linz.json

Run run.py file from your IDE: 1. Creating json file

```
#!/usr/bin/env python
if __name__ == "__main__":
    from src.digitaltwin import insert_api_to_table
    from src.digitaltwin import setup_environment
    engine = setup_environment.get_database()
    config.get_env_variable("StatsNZ_API_KEY")
    # create region_geometry table if it doesn't exist in the db.
    # no need to call region_geometry_table function if region_geometry table exist in the
db
    insert_api_to_table.region_geometry_table(engine, Stats_NZ_KEY)

    record = input_data("src/instructions_linz.json")
    # call the function to insert record in apilinks table
    insert_api_to_table.insert_records(engine, record['data_provider'],
                                       record['source'],
                                       record['api'], record['region'],
                                       record['geometry_column'],
                                       record['url'],
                                       record['layer'])
```

StatsNZ Api key is only required if the region_geometry table doesn't exist in the database otherwise you can skip lines 5-9 of the above script.

This way data will be stored in the database which then will be used to make api requests for the desired Area of Interest. geometry column's name, url and layer name are not required if the data provider is not LINZ, LRIS or StatsNZ:

To get the data from the database:

1. Make sure .env file has the correct information stored in it.
2. The geometry (geopandas dataframe type) and source list (tuple data type) needs to passed as an argument to get_data_from_db() function. Check test1.json for more details on the format and structure of the arguments.
3. Run get_data_from_db.py file from your IDE:

```
#!/usr/bin/env python
if __name__ == "__main__":
    from src.digitaltwin import get_data_from_apis
    from src.digitaltwin import setup_environment
    engine = setup_environment.get_database()
    # load in the instructions, get the source list and polygon from the user
    FILE_PATH = pathlib.Path().cwd() / pathlib.Path(r"P:\GRI_codes\DigitalTwin2\src
\test3.json")
    with open(FILE_PATH, 'r') as file_pointer:
        instructions = json.load(file_pointer)
```

```
source_list = tuple(instructions['source_name'])
geometry = gpd.GeoDataFrame.from_features(instructions["features"])
get_data_from_db(engine, geometry, source_list)
```

get_data_from_db module allows the user to get data from the multiple sources within the required Area of Interest from the database and if data is not available in the database for the desired area of Interest, wfs request is made from the stored APIs, data is stored in the database and spatial query is done within the database to get the data for the desired Area of Interest. Currently data is collected from LINZ, ECAN, Stats NZ, KiwiRail, LRIS, NIWa and opentopography but will be extended to other sources.

10.10 Raster Database

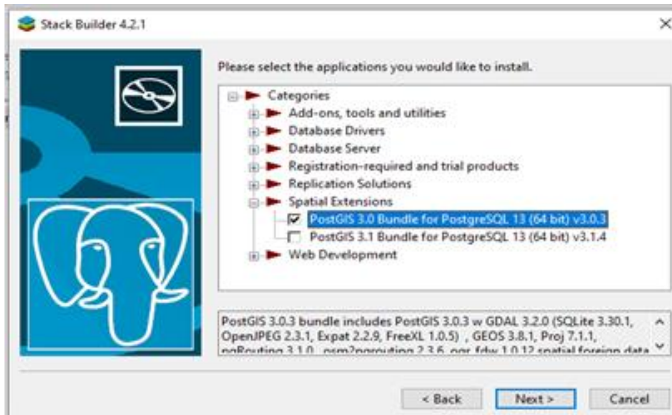
Hydrologically conditioned DEMs are generated using [geofabrics] (https://github.com/rosepearson/GeoFabrics) designed by NIWA which downloads the LiDAR data in the local directory from [opentopography] (https://portal.opentopography.org/dataCatalog) and generates DEM. These DEMs are stored in the local directory set by the user. The objective of the **dem_metadata_in_db.py** script is to store the metadata of the generated DEM in the database for the requested catchment area. Storing these details in the database helps in getting the DEM already generated using geofabrics rather than generating DEM for the same catchment, again and again, saving time and resources. The stored DEM is used to run the Flood model (BG Flood model)[https://github.com/CyprienBosserelle/BG_Flood)] designed by NIWA. The instruction file used to create hydrologically conditioned DEM is passed to the **get_dem_path(instruction)** function which checks if the DEM information exists in the database, if it doesn't exist, geofabrics is used to generate the hydrologically conditioned DEM which gets stored in the local directory and the metadata of the generated DEM is stored in the database and file path of the generated DEM is returned which is then used to run the flood model.

10.11 LiDAR Database

The data source for the LiDAR data is opentopography. The data for the requested catchment area is downloaded using geoapis in the local directory set by the user. To store the LiDAR metadata in the database, **lidar_metadata_in_db.py** script is used. The instruction file and path to the local directory where user wants to store the LiDAR data is passed as an argument to **store_lidar_path(file_path_to_store, instruction_file) function

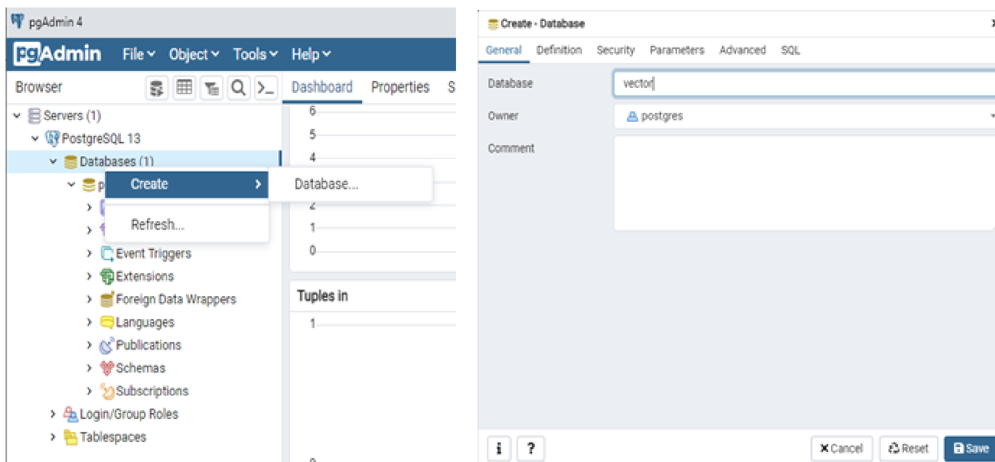
10.12 Create extensions in PostgreSQL:

1. Install Postgresql and selet PostGIS application to install along with PostgreSQL

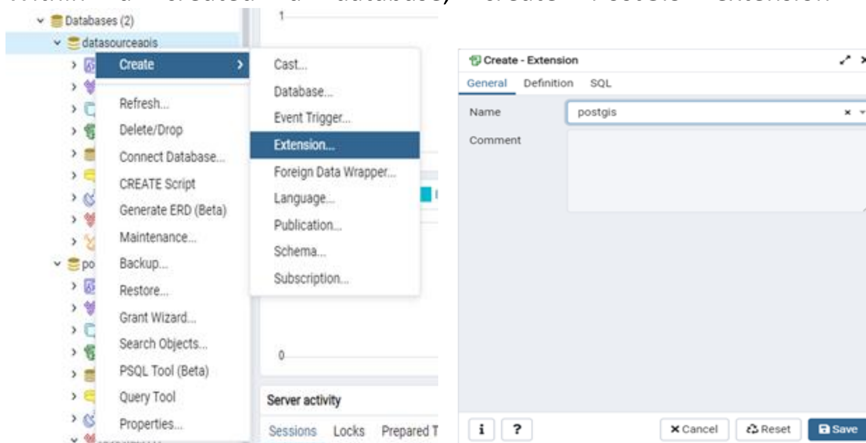


2. Open pgAdmin 4 and set your password which will be used for connecting to PostgreSQL using Python

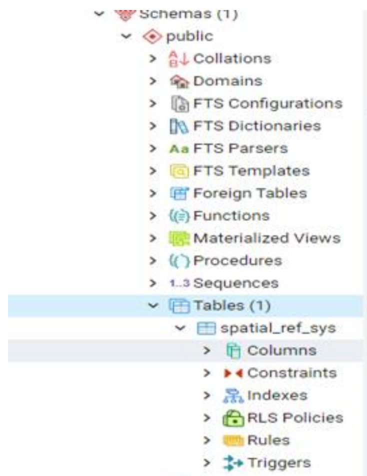
3. Create Database 'vector' as shown below:



4. Within a created a database, create PostGIS extension as shown below:



5. Once the extension is created, spatial_ref_sys table will appear under tables as shown below:



10.12.1 Exploring the database created

1. Run the container in the terminal using bash command, then using psql command to enter the database:

```bash
#!/usr/bin/env bash
docker exec -it db_postgres_digital_twin bash
psql -U [username]
```

```
P:\docker_digitaltwin>docker exec -it 09d974db2941 bash
root@09d974db2941:/# psql -U postgres
```

2. By default user will be connected to postgres database. You can change the database using the command: \c db

3. We can also check the list of tables stored in our database using the command: \dt

```
postgres=# \c db
You are now connected to database "db" as user "postgres".
db=# \dt
              List of relations
 Schema |       Name       | Type  |  Owner
--------+------------------+-------+----------
 public | apilinks         | table | postgres
 public | region_geometry  | table | postgres
 public | spatial_ref_sys  | table | postgres
(3 rows)
```

4. To check the data stored in the table: run the command:

```bash
#!/usr/bin/env bash
select * from region_geometry;
```

11 Appendix B: Software licence

All digital twin code is available under an open-source MIT licence (MIT), written below and available at: https://github.com/GeospatialResearch/Digital-Twins/blob/master/LICENSE

MIT License

© Copyright 2023 Geospatial Research Institute Toi Hangarau

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

12 Appendix C: API Documentation

The documentation below is autogenerated from the codebase (using `sphinx-autoapi`). It provides programmatic details of all API code functions within the digital twin (release 0.1.0). This is updated as soon as pull requests are landed in the repository. For the latest version, please see:

https://geospatialresearch.github.io/Digital-Twins/

Flood Resilience Digital Twin (FReDT)

Release 0.1.0

Geospatial Research Institute Toi Hangarau

Oct 31, 2023

CONTENTS:

1 API Reference **1**
- 1.1 src . . . 1
  - 1.1.1 Subpackages . . . 1
    - 1.1.1.1 src.digitaltwin . . . 1
    - 1.1.1.2 src.dynamic_boundary_conditions . . . 22
    - 1.1.1.3 src.flood_model . . . 98
  - 1.1.2 Submodules . . . 107
    - 1.1.2.1 src.app . . . 107
    - 1.1.2.2 src.config . . . 110
    - 1.1.2.3 src.datacube_data . . . 111
    - 1.1.2.4 src.run_all . . . 111
    - 1.1.2.5 src.tasks . . . 112
  - 1.1.3 Package Contents . . . 116

2 Indices and tables **117**

Python Module Index **119**

Index **121**

CHAPTER

ONE

API REFERENCE

This page contains auto-generated API reference documentation[1].

1.1 src

1.1.1 Subpackages

1.1.1.1 src.digitaltwin

Submodules

src.digitaltwin.data_to_db

This script fetches geospatial data from various providers using the 'geoapis' library and stores it in the database. It also saves user log information in the database.

[1] Created with sphinx-autoapi

Module Contents

Functions

get_nz_geospatial_layers(→ pandas.DataFrame)	Retrieve geospatial layers from the database that have a coverage area of New Zealand.
get_non_nz_geospatial_layers(→ pandas.DataFrame)	Retrieve geospatial layers from the database that do not have a coverage area of New Zealand.
get_geospatial_layer_info(→ Tuple[str, int, str, str])	Extracts geospatial layer information from a single layer entry.
get_vector_data_id_not_in_db(→ Set[int])	Get the IDs from the fetched vector_data that are not present in the specified database table
nz_geospatial_layers_data_to_db(→ None)	Fetches New Zealand geospatial layers data using 'geoapis' and stores it into the database.
get_non_intersection_area_from_db(→ geopandas.GeoDataFrame)	Get the non-intersecting area from the catchment area and user log information table in the database
process_new_non_nz_geospatial_layers(→ None)	Fetches new non-NZ geospatial layers data using 'geoapis' and stores it into the database.
process_existing_non_nz_geospatial_layers(→ None)	Fetches existing non-NZ geospatial layers data using 'geoapis' and stores it into the database.
non_nz_geospatial_layers_data_to_db(→ None)	Fetches non-NZ geospatial layers data using 'geoapis' and stores it into the database.
store_geospatial_layers_data_to_db(→ None)	Fetches geospatial layers data using 'geoapis' and stores it into the database.
user_log_info_to_db(→ None)	Store user log information to the database.

Attributes

log

`src.digitaltwin.data_to_db.`**`log`**

`exception` `src.digitaltwin.data_to_db.`**`NoNonIntersectionError`**

Bases: `Exception`

Exception raised when no non-intersecting area is found.

`src.digitaltwin.data_to_db.`**`get_nz_geospatial_layers`**(*engine: sqlalchemy.engine.Engine*) → pandas.DataFrame

Retrieve geospatial layers from the database that have a coverage area of New Zealand.

Parameters
engine (*Engine*) – The engine used to connect to the database.

Returns
Data frame containing geospatial layers that have a coverage area of New Zealand.

Return type
pd.DataFrame

src.digitaltwin.data_to_db.**get_non_nz_geospatial_layers**(*engine: sqlalchemy.engine.Engine*) → pandas.DataFrame

Retrieve geospatial layers from the database that do not have a coverage area of New Zealand.

Parameters
engine (*Engine*) – The engine used to connect to the database.

Returns
Data frame containing geospatial layers that do not have a coverage area of New Zealand.

Return type
pd.DataFrame

src.digitaltwin.data_to_db.**get_geospatial_layer_info**(*layer_row: pandas.Series*) → Tuple[str, int, str, str]

Extracts geospatial layer information from a single layer entry.

Parameters
layer_row (*pd.Series*) – A geospatial layer row that represents a single geospatial layer along with its associated information.

Returns
A tuple containing the values for data_provider, layer_id, table_name, and unique_column_name.

Return type
Tuple[str, int, str, str]

src.digitaltwin.data_to_db.**get_vector_data_id_not_in_db**(*engine: sqlalchemy.engine.Engine*, *vector_data: geopandas.GeoDataFrame*, *table_name: str*, *unique_column_name: str*, *area_of_interest: geopandas.GeoDataFrame*) → Set[int]

Get the IDs from the fetched vector_data that are not present in the specified database table for the area of interest.

Parameters

- **engine** (*Engine*) – The engine used to connect to the database.
- **vector_data** (*gpd.GeoDataFrame*) – A GeoDataFrame containing the fetched vector data.
- **table_name** (*str*) – The name of the table in the database.
- **unique_column_name** (*str*) – The name of the unique column in the table.
- **area_of_interest** (*gpd.GeoDataFrame*) – A GeoDataFrame representing the area of interest.

Returns
The set of IDs from the fetched vector_data that are not present in the specified table in the database.

Return type
Set[int]

src.digitaltwin.data_to_db.**nz_geospatial_layers_data_to_db**(*engine: sqlalchemy.engine.Engine*, *crs: int = 2193*, *verbose: bool = False*) → None

Fetches New Zealand geospatial layers data using 'geoapis' and stores it into the database.

Parameters

- **engine** (*Engine*) – The engine used to connect to the database.

- **crs** (*int = 2193*) – The coordinate reference system (CRS) code to use. Default is 2193.
- **verbose** (*bool = False*) – Whether to print messages. Default is False.

Returns
This function does not return any value.

Return type
None

`src.digitaltwin.data_to_db.`**`get_non_intersection_area_from_db`**(*engine: sqlalchemy.engine.Engine*, *catchment_area: geopandas.GeoDataFrame*, *table_name: str*) → geopandas.GeoDataFrame

Get the non-intersecting area from the catchment area and user log information table in the database for the specified table.

Parameters

- **engine** (*Engine*) – The engine used to connect to the database.
- **catchment_area** (*gpd.GeoDataFrame*) – A GeoDataFrame representing the catchment area.
- **table_name** (*str*) – The name of the table in the database.

Returns
The non-intersecting area, or the original catchment area if no intersections are found.

Return type
gpd.GeoDataFrame

Raises
NoNonIntersectionError – If the non-intersecting area is empty, it suggests that the catchment area is already fully covered.

`src.digitaltwin.data_to_db.`**`process_new_non_nz_geospatial_layers`**(*engine: sqlalchemy.engine.Engine*, *data_provider: str*, *layer_id: int*, *table_name: str*, *area_of_interest: geopandas.GeoDataFrame*, *crs: int = 2193*, *verbose: bool = False*) → None

Fetches new non-NZ geospatial layers data using 'geoapis' and stores it into the database.

Parameters

- **engine** (*Engine*) – The engine used to connect to the database.
- **data_provider** (*str*) – The data provider of the geospatial layer.
- **layer_id** (*int*) – The ID of the geospatial layer.
- **table_name** (*str*) – The database table name of the geospatial layer.
- **area_of_interest** (*gpd.GeoDataFrame*) – A GeoDataFrame representing the area of interest.
- **crs** (*int = 2193*) – The coordinate reference system (CRS) code to use. Default is 2193.
- **verbose** (*bool = False*) – Whether to print messages. Default is False.

Returns
This function does not return any value.

Return type
None

`src.digitaltwin.data_to_db.`**`process_existing_non_nz_geospatial_layers`**(*engine: sqlalchemy.engine.Engine*, *data_provider: str*, *layer_id: int*, *table_name: str*, *unique_column_name: str*, *area_of_interest: geopandas.GeoDataFrame*, *crs: int = 2193*, *verbose: bool = False*) → None

Fetches existing non-NZ geospatial layers data using 'geoapis' and stores it into the database.

Parameters

- **engine** (*Engine*) – The engine used to connect to the database.
- **data_provider** (*str*) – The data provider of the geospatial layer.
- **layer_id** (*int*) – The ID of the geospatial layer.
- **table_name** (*str*) – The database table name of the geospatial layer.
- **unique_column_name** (*str*) – The unique column name used for record identification in the database table.
- **area_of_interest** (*gpd.GeoDataFrame*) – A GeoDataFrame representing the area of interest.
- **crs** (*int = 2193*) – The coordinate reference system (CRS) code to use. Default is 2193.
- **verbose** (*bool = False*) – Whether to print messages. Default is False.

Returns
This function does not return any value.

Return type
None

`src.digitaltwin.data_to_db.`**`non_nz_geospatial_layers_data_to_db`**(*engine: sqlalchemy.engine.Engine*, *catchment_area: geopandas.GeoDataFrame*, *crs: int = 2193*, *verbose: bool = False*) → None

Fetches non-NZ geospatial layers data using 'geoapis' and stores it into the database.

Parameters

- **engine** (*Engine*) – The engine used to connect to the database.
- **catchment_area** (*gpd.GeoDataFrame*) – A GeoDataFrame representing the catchment area.
- **crs** (*int = 2193*) – The coordinate reference system (CRS) code to use. Default is 2193.
- **verbose** (*bool = False*) – Whether to print messages. Default is False.

Returns
This function does not return any value.

Return type
None

`src.digitaltwin.data_to_db.`**`store_geospatial_layers_data_to_db`**(*engine: sqlalchemy.engine.Engine*, *catchment_area: geopandas.GeoDataFrame*, *crs: int = 2193*, *verbose: bool = False*) → None

Fetches geospatial layers data using 'geoapis' and stores it into the database.

Parameters

- **engine** (*Engine*) – The engine used to connect to the database.
- **catchment_area** (*gpd.GeoDataFrame*) – A GeoDataFrame representing the catchment area.
- **crs** (*int = 2193*) – The coordinate reference system (CRS) code to use. Default is 2193.
- **verbose** (*bool = False*) – Whether to print messages. Default is False.

Returns
This function does not return any value.

Return type
None

`src.digitaltwin.data_to_db.`**`user_log_info_to_db`**(*engine: sqlalchemy.engine.Engine*, *catchment_area: geopandas.GeoDataFrame*) → None

Store user log information to the database.

Parameters

- **engine** (*Engine*) – The engine used to connect to the database.
- **catchment_area** (*gpd.GeoDataFrame*) – A GeoDataFrame representing the catchment area.

Returns
This function does not return any value.

Return type
None

src.digitaltwin.get_data_using_geoapis

This script provides functions to retrieve vector data from multiple providers, including StatsNZ, LINZ, LRIS, and MFE, using the 'geoapis' library. To access data from each provider, you'll need to set an API key in the environment variables.

Module Contents

Classes

MFE	A class to manage fetching Vector data from MFE.

Functions

clean_fetched_vector_data(→ geopandas.GeoDataFrame)	Clean the fetched vector data by performing necessary transformations.
fetch_vector_data_using_geoapis(→ geopandas.GeoDataFrame)	Fetch vector data using 'geoapis' based on the specified data provider, layer ID, and other parameters.

class src.digitaltwin.get_data_using_geoapis.**MFE**

Bases: geoapis.vector.WfsQueryBase

A class to manage fetching Vector data from MFE.

General details at: https://data.mfe.govt.nz/ API details at: https://help.koordinates.com/

Note that the 'GEOMETRY_NAMES' used when making WFS 'cql_filter' queries varies between layers. The MFE generally follows the LINZ LDS but uses 'Shape' in place of 'shape'. It still uses 'GEOMETRY'.

NETLOC_API = 'data.mfe.govt.nz'

GEOMETRY_NAMES = ['GEOMETRY', 'Shape']

src.digitaltwin.get_data_using_geoapis.**clean_fetched_vector_data**(*fetched_data: geopandas.GeoDataFrame*) → geopandas.GeoDataFrame

Clean the fetched vector data by performing necessary transformations.

Parameters

fetched_data (*gpd.GeoDataFrame*) – A GeoDataFrame containing the fetched vector data.

Returns

A GeoDataFrame containing the cleaned vector data.

Return type

gpd.GeoDataFrame

src.digitaltwin.get_data_using_geoapis.**fetch_vector_data_using_geoapis**(*data_provider: str*, *layer_id: int*, *crs: int = 2193*, *verbose: bool = False*, *bounding_polygon: geopandas.GeoDataFrame | None = None*) → geopandas.GeoDataFrame

Fetch vector data using 'geoapis' based on the specified data provider, layer ID, and other parameters.

Parameters

- **data_provider** (*str*) – The data provider to use. Supported values: “StatsNZ”, “LINZ”, “LRIS”, “MFE”.
- **layer_id** (*int*) – The ID of the layer to fetch.
- **crs** (*int = 2193*) – The coordinate reference system (CRS) code to use. Default is 2193.
- **verbose** (*bool = False*) – Whether to print messages. Default is False.
- **bounding_polygon** (*Optional[gpd.GeoDataFrame] = None*) – Bounding polygon for data fetching. Default is all of New Zealand.

Returns
A GeoDataFrame containing the fetched vector data.

Return type
gpd.GeoDataFrame

Raises
ValueError – If an unsupported ‘data_provider’ value is provided.

src.digitaltwin.instructions_records_to_db

This script processes ‘instructions_run’ records, validates URLs and instruction fields, and stores them in the ‘geospatial_layers’ table of the database.

Module Contents

Functions

validate_url_reachability(→ None)	Validate the URL by checking its format and reachability.
validate_instruction_fields(→ None)	Validate the fields of an instruction.
read_and_check_instructions_file(→ pandas.DataFrame)	Read and check the instructions_run file, validating URLs and instruction fields.
get_existing_geospatial_layers(→ pandas.DataFrame)	Retrieve existing geospatial layers from the 'geospatial_layers' table.
get_non_existing_records(→ pandas.DataFrame)	Get 'instructions_run' records that are not available in the database.
store_instructions_records_to_db(→ None)	Store 'instructions_run' records in the 'geospatial_layers' table in the database.

Attributes

log

src.digitaltwin.instructions_records_to_db.**log**

`src.digitaltwin.instructions_records_to_db.`**`validate_url_reachability`**(*section: str*, *url: str*) → None

Validate the URL by checking its format and reachability.

Parameters

- **`section`** (*`str`*) – The section identifier of the instruction.
- **`url`** (*`str`*) – The URL to validate.

Returns
This function does not return any value.

Return type
None

Raises
`ValueError` –

- If the URL is invalid.
- If the URL is unreachable.

`src.digitaltwin.instructions_records_to_db.`**`validate_instruction_fields`**(*section: str*, *instruction: Dict[str, str | int]*) → None

Validate the fields of an instruction. Each instruction should provide either 'coverage_area' or 'unique_column_name', but not both.

Parameters

- **`section`** (*`str`*) – The section identifier of the instruction.
- **`instruction`** (*`Dict[str, Union[str, int]]`*) – The instruction details.

Returns
This function does not return any value.

Return type
None

Raises
`ValueError` –

- If both 'coverage_area' and 'unique_column_name' are provided.
- If both 'coverage_area' and 'unique_column_name' are not provided.

`src.digitaltwin.instructions_records_to_db.`**`read_and_check_instructions_file`**() → pandas.DataFrame

Read and check the instructions_run file, validating URLs and instruction fields.

Returns
The processed instructions DataFrame.

Return type
pd.DataFrame

`src.digitaltwin.instructions_records_to_db.`**`get_existing_geospatial_layers`**(*engine: sqlalchemy.engine.Engine*) → pandas.DataFrame

Retrieve existing geospatial layers from the 'geospatial_layers' table.

Parameters
: `engine` (*Engine*) – The engine used to connect to the database.

Returns
: Data frame containing the existing geospatial layers.

Return type
: pd.DataFrame

`src.digitaltwin.instructions_records_to_db.`**`get_non_existing_records`**(*instructions_df: pandas.DataFrame*, *existing_layers_df: pandas.DataFrame*) → pandas.DataFrame

Get 'instructions_run' records that are not available in the database.

Parameters

- **`instructions_df`** (*pd.DataFrame*) – Data frame containing the 'instructions_run' records.
- **`existing_layers_df`** (*pd.DataFrame*) – Data frame containing the existing 'instructions_run' records from the database.

Returns
: Data frame containing the 'instructions_run' records that are not available in the database.

Return type
: pd.DataFrame

`src.digitaltwin.instructions_records_to_db.`**`store_instructions_records_to_db`**(*engine: sqlalchemy.engine.Engine*) → None

Store 'instructions_run' records in the 'geospatial_layers' table in the database.

Parameters
: `engine` (*Engine*) – The engine used to connect to the database.

Returns
: This function does not return any value.

Return type
: None

`src.digitaltwin.run`

This script automates the retrieval and storage of geospatial data from various providers using the 'geoapis' library. It populates the 'geospatial_layers' table in the database and stores user log information for tracking and reference.

Module Contents

Functions

main(→ None)	Connects to various data providers to fetch geospatial data for the selected polygon, i.e., the catchment area.

Attributes

sample_polygon

`src.digitaltwin.run.`**`main`**(*selected_polygon_gdf: geopandas.GeoDataFrame*, *log_level: src.digitaltwin.utils.LogLevel = LogLevel.DEBUG*) → None

Connects to various data providers to fetch geospatial data for the selected polygon, i.e., the catchment area. Subsequently, it populates the 'geospatial_layers' table in the database and stores user log information for tracking and reference.

Parameters

- **`selected_polygon_gdf`** (*`gpd.GeoDataFrame`*) – A GeoDataFrame representing the selected polygon, i.e., the catchment area.
- **`log_level`** (*`LogLevel = LogLevel.DEBUG`*) – The log level to set for the root logger. Defaults to LogLevel.DEBUG. The available logging levels and their corresponding numeric values are: - LogLevel.CRITICAL (50) - LogLevel.ERROR (40) - LogLevel.WARNING (30) - LogLevel.INFO (20) - LogLevel.DEBUG (10) - LogLevel.NOTSET (0)

Returns

This function does not return any value.

Return type

None

`src.digitaltwin.run.`**`sample_polygon`**

`src.digitaltwin.setup_environment`

This script provides functions to set up the database connection using SQLAlchemy and environment variables, as well as to create an SQLAlchemy engine for database operations.

Module Contents

Functions

get_database(→ sqlalchemy.engine.Engine)	Set up the database connection. Exit the program if connection fails.
get_connection_from_profile(→ sqlalchemy.engine.Engine)	Sets up database connection from configuration.
get_engine(→ sqlalchemy.engine.Engine)	Get SQLAlchemy engine using credentials.

Attributes

log
Base

src.digitaltwin.setup_environment.**log**

src.digitaltwin.setup_environment.**Base**

src.digitaltwin.setup_environment.**get_database**() → sqlalchemy.engine.Engine

Set up the database connection. Exit the program if connection fails.

Returns
The engine used to connect to the database.

Return type
Engine

Raises
OperationalError – If the connection to the database fails.

src.digitaltwin.setup_environment.**get_connection_from_profile**() → sqlalchemy.engine.Engine

Sets up database connection from configuration.

Returns
The engine used to connect to the database.

Return type
Engine

Raises
ValueError – If one or more connection credentials are missing in the .env file.

src.digitaltwin.setup_environment.**get_engine**(*host: str*, *port: str*, *db: str*, *username: str*, *password: str*) → sqlalchemy.engine.Engine

Get SQLAlchemy engine using credentials.

Parameters

- **host** (*str*) – Hostname of the database server.
- **port** (*str*) – Port number.

- **db** (*str*) – Database name.
- **username** (*str*) – Username.
- **password** (*str*) – Password for the database.

Returns
The engine used to connect to the database.

Return type
Engine

src.digitaltwin.tables

This script contains SQLAlchemy models for various database tables and utility functions for database operations.

Module Contents

Classes

GeospatialLayers	Class representing the 'geospatial_layers' table.
UserLogInfo	Class representing the 'user_log_information' table.
RiverNetworkExclusions	Class representing the 'rec1_network_exclusions' table.
RiverNetworkOutput	Class representing the 'rec1_network_output' table.
BGFloodModelOutput	Class representing the 'bg_flood_model_output' table.
BuildingFloodStatus	

Functions

create_table(→ None)	Create a table in the database if it doesn't already exist, using the provided engine.
check_table_exists(→ bool)	Check if a table exists in the database.
execute_query(→ None)	Execute the given query on the provided engine using a session.

Attributes

Base

src.digitaltwin.tables.**Base**

class src.digitaltwin.tables.**GeospatialLayers**

Bases: *Base*

Class representing the ‘geospatial_layers’ table.

`__tablename__`
Name of the database table.

Type
str

`unique_id`
Unique identifier for each geospatial layer entry (primary key).

Type
int

`data_provider`
Name of the data provider.

Type
str

`layer_id`
Identifier for the layer.

Type
int

`table_name`
Name of the table containing the data.

Type
str

`unique_column_name`
Name of the unique column in the table.

Type
Optional[str]

`coverage_area`
Coverage area of the geospatial data, e.g. 'New Zealand'.

Type
Optional[str]

`url`
URL pointing to the geospatial layer.

Type
str

`__tablename__ = 'geospatial_layers'`

`unique_id`

`data_provider`

`layer_id`

`table_name`

`unique_column_name`

`coverage_area`

url

class src.digitaltwin.tables.**UserLogInfo**

Bases: *Base*

Class representing the 'user_log_information' table.

__tablename__

Name of the database table.

Type

str

unique_id

Unique identifier for each log entry (primary key).

Type

int

source_table_list

A list of tables (geospatial layers) associated with the log entry.

Type

Dict[str]

created_at

Timestamp indicating when the log entry was created.

Type

datetime

geometry

Geometric representation of the catchment area coverage.

Type

Polygon

__tablename__ = 'user_log_information'

unique_id

source_table_list

created_at

geometry

class src.digitaltwin.tables.**RiverNetworkExclusions**

Bases: *Base*

Class representing the 'rec1_network_exclusions' table.

__tablename__

Name of the database table.

Type

str

rec1_network_id

An identifier for the river network associated with each new run.

Type
int

objectid

An identifier for the REC1 river object matching from the 'rec1_data' table.

Type
int

exclusion_cause

Cause of exclusion, i.e., the reason why the REC1 river geometry was excluded.

Type
str

geometry

Geometric representation of the excluded REC1 river features.

Type
LineString

__tablename__ = 'rec1_network_exclusions'

rec1_network_id

objectid

exclusion_cause

geometry

__table_args__ = ()

class src.digitaltwin.tables.**RiverNetworkOutput**

Bases: *Base*

Class representing the 'rec1_network_output' table.

__tablename__

Name of the database table.

Type
str

rec1_network_id

An identifier for the river network associated with each new run (primary key).

Type
int

network_path

Path to the REC1 river network file.

Type
str

network_data_path

Path to the REC1 river network data file.

Type
str

created_at

Timestamp indicating when the output was created.

Type
datetime

geometry

Geometric representation of the catchment area coverage.

Type
Polygon

__tablename__ = 'rec1_network_output'

rec1_network_id

network_path

network_data_path

created_at

geometry

class src.digitaltwin.tables.**BGFloodModelOutput**

Bases: *Base*

Class representing the 'bg_flood_model_output' table.

__tablename__

Name of the database table.

Type
str

unique_id

Unique identifier for each entry (primary key).

Type
int

file_name

Name of the flood model output file.

Type
str

file_path

Path to the flood model output file.

Type
str

created_at

Timestamp indicating when the output was created.

Type
datetime

geometry

Geometric representation of the catchment area coverage.

Type
Polygon

__tablename__ = 'bg_flood_model_output'

unique_id

file_name

file_path

created_at

geometry

class src.digitaltwin.tables.**BuildingFloodStatus**

Bases: *Base*

__tablename__ = 'building_flood_status'

unique_id

building_outline_id

is_flooded

flood_model_id

src.digitaltwin.tables.**create_table**(*engine: sqlalchemy.engine.Engine*, *table: Base*) → None

Create a table in the database if it doesn't already exist, using the provided engine.

Parameters

- **engine** (*Engine*) – The engine used to connect to the database.
- **table** (*Base*) – Class representing the table to create.

Returns
This function does not return any value.

Return type
None

src.digitaltwin.tables.**check_table_exists**(*engine: sqlalchemy.engine.Engine*, *table_name: str*, *schema: str = 'public'*) → bool

Check if a table exists in the database.

Parameters

- **engine** (*Engine*) – The engine used to connect to the database.
- **table_name** (*str*) – The name of the table to check for existence.

- **schema** (*str = "public"*) – The name of the schema where the table resides. Defaults to “public”.

Returns
True if the table exists, False otherwise.

Return type
bool

src.digitaltwin.tables.**execute_query**(*engine: sqlalchemy.engine.Engine*, *query*) → None

Execute the given query on the provided engine using a session.

Parameters

- **engine** (*Engine*) – The engine used to connect to the database.
- **query** – The query to be executed.

Returns
This function does not return any value.

Return type
None

Raises
Exception – If an error occurs during the execution of the query.

src.digitaltwin.utils

This script provides utility functions for logging configuration and geospatial data manipulation.

Module Contents

Classes

LogLevel	Enum class representing different logging levels mapped to their corresponding numeric values from the

Functions

log_execution_info(→ None)	Logs a debug message indicating the execution of the function in the script.
setup_logging(→ None)	Configures the root logger with the specified log level and formats, captures warnings, and excludes specific
get_catchment_area(→ geopandas.GeoDataFrame)	Convert the coordinate reference system (CRS) of the catchment area GeoDataFrame to the specified CRS.
get_nz_boundary(→ geopandas.GeoDataFrame)	Get the boundary of New Zealand in the specified Coordinate Reference System (CRS).

Attributes

log	

src.digitaltwin.utils.**log**

class src.digitaltwin.utils.**LogLevel**

Bases: `enum.IntEnum`

Enum class representing different logging levels mapped to their corresponding numeric values from the logging library.

CRITICAL

The critical logging level. Corresponds to logging.CRITICAL (50).

Type
int

ERROR

The error logging level. Corresponds to logging.ERROR (40).

Type
int

WARNING

The warning logging level. Corresponds to logging.WARNING (30).

Type
int

INFO

The info logging level. Corresponds to logging.INFO (20).

Type
int

DEBUG

The debug logging level. Corresponds to logging.DEBUG (10).

Type
int

NOTSET

The not-set logging level. Corresponds to logging.NOTSET (0).

Type
int

CRITICAL

ERROR

WARNING

INFO

DEBUG

NOTSET

src.digitaltwin.utils.**log_execution_info**() → None

Logs a debug message indicating the execution of the function in the script.

Returns
This function does not return any value.

Return type
None

src.digitaltwin.utils.**setup_logging**(*log_level: LogLevel = LogLevel.DEBUG*) → None

Configures the root logger with the specified log level and formats, captures warnings, and excludes specific loggers from propagating their messages to the root logger. Additionally, logs a debug message indicating the execution of the function in the script.

Parameters
log_level (*LogLevel = LogLevel.DEBUG*) – The log level to set for the root logger. Defaults to LogLevel.DEBUG. The available logging levels and their corresponding numeric values are: - LogLevel.CRITICAL (50) - LogLevel.ERROR (40) - LogLevel.WARNING (30) - LogLevel.INFO (20) - LogLevel.DEBUG (10) - LogLevel.NOTSET (0)

Returns
This function does not return any value.

Return type
None

src.digitaltwin.utils.**get_catchment_area**(*catchment_area: geopandas.GeoDataFrame*, *to_crs: int = 2193*) → geopandas.GeoDataFrame

Convert the coordinate reference system (CRS) of the catchment area GeoDataFrame to the specified CRS.

Parameters

- **catchment_area** (*gpd.GeoDataFrame*) – A GeoDataFrame representing the catchment area.
- **to_crs** (*int = 2193*) – Coordinate Reference System (CRS) code to convert the catchment area to. Default is 2193.

Returns
A GeoDataFrame representing the catchment area with the transformed CRS.

Return type
gpd.GeoDataFrame

src.digitaltwin.utils.**get_nz_boundary**(*engine: sqlalchemy.engine.Engine*, *to_crs: int = 2193*) → geopandas.GeoDataFrame

Get the boundary of New Zealand in the specified Coordinate Reference System (CRS).

Parameters

- **engine** (*Engine*) – The engine used to connect to the database.
- **to_crs** (*int = 2193*) – Coordinate Reference System (CRS) code to which the boundary will be converted. Default is 2193.

Returns
A GeoDataFrame representing the boundary of New Zealand in the specified CRS.

Return type
gpd.GeoDataFrame

1.1.1.2 src.dynamic_boundary_conditions

Subpackages

src.dynamic_boundary_conditions.rainfall

Submodules

src.dynamic_boundary_conditions.rainfall.hirds_rainfall_data_from_db

Retrieve all rainfall data for sites within the catchment area from the database.

Module Contents

Functions

filter_for_duration(→ pandas.DataFrame)	Filter the HIRDS rainfall data for a requested duration.
get_one_site_rainfall_data(→ pandas.DataFrame)	Get the HIRDS rainfall data for the requested site from the database and return the required data in
rainfall_data_from_db(→ pandas.DataFrame)	Get rainfall data for the sites within the catchment area and return it as a Pandas DataFrame.

src.dynamic_boundary_conditions.rainfall.hirds_rainfall_data_from_db.**filter_for_duration**(*rain_data: pandas.DataFrame*, *duration: str*) → pandas.DataFrame

Filter the HIRDS rainfall data for a requested duration.

Parameters

- **rain_data** (*pd.DataFrame*) – HIRDS rainfall data in Pandas DataFrame format.
- **duration** (*str*) – Storm duration. Valid options are: '10m', '20m', '30m', '1h', '2h', '6h', '12h', '24h', '48h', '72h', '96h', '120h', or 'all'.

Returns
Filtered rainfall data for the requested duration.

Return type
pd.DataFrame

src.dynamic_boundary_conditions.rainfall.hirds_rainfall_data_from_db.**get_one_site_rainfall_data**(*engine: sqlalchem site_id: str, rcp: float | None, time_peri str | None, ari: float, du- ra- tion: str, idf: bool*) → pandas.D

Get the HIRDS rainfall data for the requested site from the database and return the required data in Pandas DataFrame format.

Parameters

- **engine** (*Engine*) – The engine used to connect to the database.
- **site_id** (*str*) – HIRDS rainfall site ID.
- **rcp** (*Optional[float]*) – Representative Concentration Pathway (RCP) value. Valid options are 2.6, 4.5, 6.0, 8.5, or None for historical data.
- **time_period** (*Optional[str]*) – Future time period. Valid options are “2031-2050”, “2081-2100”, or None for historical data.
- **ari** (*float*) – Average Recurrence Interval (ARI) value. Valid options are 1.58, 2, 5, 10, 20, 30, 40, 50, 60, 80, 100, or 250.
- **duration** (*str*) – Storm duration. Valid options are: ‘10m’, ‘20m’, ‘30m’, ‘1h’, ‘2h’, ‘6h’, ‘12h’, ‘24h’, ‘48h’, ‘72h’, ‘96h’, ‘120h’, or ‘all’.
- **idf** (*bool*) – Set to False for rainfall depth data, and True for rainfall intensity data.

Returns
HIRDS rainfall data for the requested site and parameters.

Return type
pd.DataFrame

Raises
ValueError – If rcp and time_period arguments are inconsistent.

`src.dynamic_boundary_conditions.rainfall.hirds_rainfall_data_from_db.`**`rainfall_data_from_db`**`(`*engine: sqlalchemy.engi sites_in_catchm geopandas.GeoDataFr rcp: float | None*, *time_period: str | None*, *ari: float*, *idf: bool = False*, *duration: str = 'all'*`)` → pandas.DataFran

Get rainfall data for the sites within the catchment area and return it as a Pandas DataFrame.

Parameters

- **engine** (*Engine*) – The engine used to connect to the database.
- **sites_in_catchment** (*gpd.GeoDataFrame*) – Rainfall sites coverage areas (Thiessen polygons) within the catchment area.
- **rcp** (*Optional[float]*) – Representative Concentration Pathway (RCP) value. Valid options are 2.6, 4.5, 6.0, 8.5, or None for historical data.
- **time_period** (*Optional[str]*) – Future time period. Valid options are "2031-2050", "2081-2100", or None for historical data.
- **ari** (*float*) – Average Recurrence Interval (ARI) value. Valid options are 1.58, 2, 5, 10, 20, 30, 40, 50, 60, 80, 100, or 250.
- **idf** (*bool = False*) – Set to False for rainfall depth data, and True for rainfall intensity data.
- **duration** (*str = "all"*) – Storm duration. Valid options are: '10m', '20m', '30m', '1h', '2h', '6h', '12h', '24h', '48h', '72h', '96h', '120h', or 'all'. Default is 'all'.

Returns

A DataFrame containing the rainfall data for the sites within the catchment area.

Return type

pd.DataFrame

src.dynamic_boundary_conditions.rainfall.hirds_rainfall_data_to_db

Store the rainfall data for all the sites within the catchment area in the database.

Module Contents

Functions

db_rain_table_name(→ str)	Return the relevant rainfall data table name used in the database.
get_sites_id_in_catchment(→ List[str])	Get the rainfall site IDs within the catchment area.
get_sites_id_not_in_db(→ List[str])	Get the list of rainfall site IDs that are within the catchment area but not in the database.
add_rainfall_data_to_db(→ None)	Store the rainfall data for a specific site in the database.
add_each_site_rainfall_data(→ None)	Add rainfall data for each site in the sites_id_list to the database.
rainfall_data_to_db(→ None)	Store rainfall data of all the sites within the catchment area in the database.

Attributes

log

src.dynamic_boundary_conditions.rainfall.hirds_rainfall_data_to_db.**log**

src.dynamic_boundary_conditions.rainfall.hirds_rainfall_data_to_db.**db_rain_table_name**(*idf: bool*) → str

Return the relevant rainfall data table name used in the database.

Parameters
idf (*bool*) – Set to False for rainfall depth data, and True for rainfall intensity data.

Returns
The relevant rainfall data table name.

Return type
str

src.dynamic_boundary_conditions.rainfall.hirds_rainfall_data_to_db.**get_sites_id_in_catchment**(*sites_in_catch geopan- das.GeoData*
→
List[str]

Get the rainfall site IDs within the catchment area.

Parameters
sites_in_catchment (*gpd.GeoDataFrame*) – Rainfall site coverage areas (Thiessen polygons) that intersect or are within the catchment area.

Returns
The rainfall site IDs within the catchment area.

Return type
List[str]

`src.dynamic_boundary_conditions.rainfall.hirds_rainfall_data_to_db.`**`get_sites_id_not_in_db`**(*engine: sqlalchemy.engin sites_id_in_catch List[str]*, *idf: bool*) → List[str]

Get the list of rainfall site IDs that are within the catchment area but not in the database.

Parameters

- **engine** (*Engine*) – The engine used to connect to the database.
- **sites_id_in_catchment** (*List[str]*) – Rainfall site IDs within the catchment area.
- **idf** (*bool*) – Set to False for rainfall depth data, and True for rainfall intensity data.

Returns
The rainfall site IDs within the catchment area but not present in the database.

Return type
List[str]

`src.dynamic_boundary_conditions.rainfall.hirds_rainfall_data_to_db.`**`add_rainfall_data_to_db`**(*engine: sqlalchemy.engi site_id: str*, *idf: bool*) → None

Store the rainfall data for a specific site in the database.

Parameters

- **engine** (*Engine*) – The engine used to connect to the database.
- **site_id** (*str*) – HIRDS rainfall site ID.
- **idf** (*bool*) – Set to False for rainfall depth data, and True for rainfall intensity data.

Returns
This function does not return any value.

Return type
None

`src.dynamic_boundary_conditions.rainfall.hirds_rainfall_data_to_db.`**`add_each_site_rainfall_data`**(*engine: sqlalchemy sites_id_lis List[str]*, *idf: bool*) → None

Add rainfall data for each site in the sites_id_list to the database.

Parameters

- **engine** (*Engine*) – The engine used to connect to the database.
- **sites_id_list** (*List[str]*) – List of rainfall sites' IDs.
- **idf** (*bool*) – Set to False for rainfall depth data, and True for rainfall intensity data.

Returns
This function does not return any value.

Return type
None

src.dynamic_boundary_conditions.rainfall.hirds_rainfall_data_to_db.**rainfall_data_to_db**(*engine: sqlalchemy.engine.En sites_in_catchment: geopandas.GeoDataFrame*, *idf: bool = False*) → None

Store rainfall data of all the sites within the catchment area in the database.

Parameters

- **engine** (*Engine*) – The engine used to connect to the database.
- **sites_in_catchment** (*gpd.GeoDataFrame*) – Rainfall sites coverage areas (Thiessen polygons) that intersect or are within the catchment area.
- **idf** (*bool = False*) – Set to False for rainfall depth data, and True for rainfall intensity data.

Returns
This function does not return any value.

Return type
None

src.dynamic_boundary_conditions.rainfall.hyetograph

Get hyetograph data and generate interactive hyetograph plots for sites located within the catchment area.

Module Contents

Functions

get_transposed_data(→ pandas.DataFrame)	Clean and transpose the retrieved scenario data from the database for sites within the catchment area and
get_interpolated_data(→ pandas.DataFrame)	Perform temporal interpolation on the transposed scenario data for sites within the catchment area and
get_interp_incremental_data(→ pandas.DataFrame)	Get the incremental rainfall depths (difference between current and preceding cumulative rainfall)
get_storm_length_increment_data(→ pandas.DataFrame)	Get the incremental rainfall depths for sites within the catchment area for a specific storm duration.
add_time_information(→ pandas.DataFrame)	Add time information (seconds, minutes, and hours column) to the hyetograph data based on the
transform_data_for_selected_method(→ pandas.DataFrame)	Transform the storm length incremental rainfall depths for sites within the catchment area based on
hyetograph_depth_to_intensity(→ pandas.DataFrame)	Convert hyetograph depths data to hyetograph intensities data for all sites within the catchment area.
get_hyetograph_data(→ pandas.DataFrame)	Get hyetograph intensities data for all sites within the catchment area and return it in Pandas DataFrame format.
hyetograph_data_wide_to_long(→ pandas.DataFrame)	Transform hyetograph intensities data for all sites within the catchment area from wide format to long format.
hyetograph(→ None)	Create interactive individual hyetograph plots for sites within the catchment area.

src.dynamic_boundary_conditions.rainfall.hyetograph.**get_transposed_data**(*rain_depth_in_catchment: pandas.DataFrame*) → pandas.DataFrame

Clean and transpose the retrieved scenario data from the database for sites within the catchment area and return it in transposed Pandas DataFrame format.

Parameters

rain_depth_in_catchment (*pd.DataFrame*) – Rainfall depths for sites within the catchment area for a specified scenario retrieved from the database.

Returns

A DataFrame containing the cleaned and transposed scenario data.

Return type

pd.DataFrame

src.dynamic_boundary_conditions.rainfall.hyetograph.**get_interpolated_data**(*transposed_catchment_data: pandas.DataFrame*, *increment_mins: int*, *interp_method: str*) → pandas.DataFrame

Perform temporal interpolation on the transposed scenario data for sites within the catchment area and return it in Pandas DataFrame format.

Parameters

- **transposed_catchment_data** (*pd.DataFrame*) – Transposed scenario data retrieved from the database.
- **increment_mins** (*int*) – Time interval in minutes.
- **interp_method** (*str*) – Temporal interpolation method to be used. Refer to 'scipy.interpolate.interp1d()' for available methods. One of 'linear', 'nearest', 'nearest-up', 'zero', 'slinear', 'quadratic', 'cubic', 'previous', or 'next'.

Returns
A DataFrame containing the interpolated scenario data.

Return type
pd.DataFrame

Raises
`ValueError` –

- If the specified 'increment_mins' is out of range.
- If the specified 'interp_method' is not supported.

`src.dynamic_boundary_conditions.rainfall.hyetograph.`**`get_interp_incremental_data`**(*interp_catchment_data: pandas.DataFrame*) → pandas.DataFrame

Get the incremental rainfall depths (difference between current and preceding cumulative rainfall) for sites within the catchment area and return it in Pandas DataFrame format.

Parameters
interp_catchment_data (*pd.DataFrame*) – Interpolated scenario data for sites within the catchment area.

Returns
A DataFrame containing the incremental rainfall depths.

Return type
pd.DataFrame

`src.dynamic_boundary_conditions.rainfall.hyetograph.`**`get_storm_length_increment_data`**(*interp_increment_data: pandas.DataFrame*, *storm_length_mins: int*) → pandas.DataFrame

Get the incremental rainfall depths for sites within the catchment area for a specific storm duration.

Parameters

- **interp_increment_data** (*pd.DataFrame*) – Incremental rainfall depths for sites within the catchment area.
- **storm_length_mins** (*int*) – Storm duration in minutes.

Returns
Incremental rainfall depths for sites within the catchment area for the specified storm duration.

Return type
pd.DataFrame

Raises
`ValueError` – If the specified 'storm_length_mins' is less than the minimum storm duration available in the data.

src.dynamic_boundary_conditions.rainfall.hyetograph.**add_time_information**(*site_data: pandas.DataFrame*, *storm_length_mins: int*, *time_to_peak_mins: int | float*, *increment_mins: int*, *hyeto_method: src.dynamic_boundary_conditions.rainf* → pandas.DataFrame

Add time information (seconds, minutes, and hours column) to the hyetograph data based on the selected hyetograph method.

Parameters

- **site_data** (*pd.DataFrame*) – Hyetograph data for a rainfall site or gauge.
- **storm_length_mins** (*int*) – Storm duration in minutes.
- **time_to_peak_mins** (*Union[int, float]*) – The time in minutes when rainfall is at its greatest (reaches maximum).
- **increment_mins** (*int*) – Time interval in minutes.
- **hyeto_method** (HyetoMethod) – Hyetograph method to be used.

Returns
Hyetograph data with added time information.

Return type
pd.DataFrame

Raises
`ValueError` – If the specified 'time_to_peak_mins' is less than half of the storm duration.

src.dynamic_boundary_conditions.rainfall.hyetograph.**transform_data_for_selected_method**(*interp_increment_dat pandas.DataFrame*, *storm_length_mins: int*, *time_to_peak_mins: int | float*, *increment_mins: int*, *hyeto_method: src.dynamic_boundar* → pandas.DataFrame

Transform the storm length incremental rainfall depths for sites within the catchment area based on the selected

hyetograph method and return hyetograph depths data for all sites within the catchment area in Pandas DataFrame format.

Parameters

- **interp_increment_data** (*pd.DataFrame*) – Incremental rainfall depths for sites within the catchment area.
- **storm_length_mins** (*int*) – Storm duration in minutes.
- **time_to_peak_mins** (*Union[int, float]*) – The time in minutes when rainfall is at its greatest (reaches maximum).
- **increment_mins** (*int*) – Time interval in minutes.
- **hyeto_method** (HyetoMethod) – Hyetograph method to be used.

Returns
Hyetograph depths data for all sites within the catchment area.

Return type
pd.DataFrame

src.dynamic_boundary_conditions.rainfall.hyetograph.**hyetograph_depth_to_intensity**(*hyetograph_depth: pandas.DataFrame*, *increment_mins: int*, *hyeto_method: src.dynamic_boundary_con*) → pandas.DataFrame

Convert hyetograph depths data to hyetograph intensities data for all sites within the catchment area.

Parameters

- **hyetograph_depth** (*pd.DataFrame*) – Hyetograph depths data for sites within the catchment area.
- **increment_mins** (*int*) – Time interval in minutes.
- **hyeto_method** (HyetoMethod) – Hyetograph method to be used.

Returns
Hyetograph intensities data for all sites within the catchment area.

Return type
pd.DataFrame

src.dynamic_boundary_conditions.rainfall.hyetograph.**get_hyetograph_data**(*rain_depth_in_catchment: pandas.DataFrame*, *storm_length_mins: int*, *time_to_peak_mins: int | float*, *increment_mins: int*, *interp_method: str*, *hyeto_method: src.dynamic_boundary_conditions.rainfa*) → pandas.DataFrame

Get hyetograph intensities data for all sites within the catchment area and return it in Pandas DataFrame format.

Parameters

- **rain_depth_in_catchment** (*pd.DataFrame*) – Rainfall depths for sites within the catchment area for a specified scenario retrieved from the database.
- **storm_length_mins** (*int*) – Storm duration in minutes.
- **time_to_peak_mins** (*Union[int, float]*) – The time in minutes when rainfall is at its greatest (reaches maximum).
- **increment_mins** (*int*) – Time interval in minutes.
- **interp_method** (*str*) – Temporal interpolation method to be used. Refer to 'scipy.interpolate.interp1d()' for available methods. One of 'linear', 'nearest', 'nearest-up', 'zero', 'slinear', 'quadratic', 'cubic', 'previous', or 'next'.
- **hyeto_method** (HyetoMethod) – Hyetograph method to be used.

Returns
Hyetograph intensities data for all sites within the catchment area.

Return type
pd.DataFrame

`src.dynamic_boundary_conditions.rainfall.hyetograph.`**`hyetograph_data_wide_to_long`**(*hyetograph_data: pandas.DataFrame*) → pandas.DataFrame

Transform hyetograph intensities data for all sites within the catchment area from wide format to long format.

Parameters
hyetograph_data (*pd.DataFrame*) – Hyetograph intensities data for sites within the catchment area.

Returns
Hyetograph intensities data in long format.

Return type
pd.DataFrame

`src.dynamic_boundary_conditions.rainfall.hyetograph.`**`hyetograph`**(*hyetograph_data: pandas.DataFrame*, *ari: int*) → None

Create interactive individual hyetograph plots for sites within the catchment area.

Parameters

- **hyetograph_data** (*pd.DataFrame*) – Hyetograph intensities data for sites within the catchment area.
- **ari** (*int*) – Average Recurrence Interval (ARI) value. Valid options are 1.58, 2, 5, 10, 20, 30, 40, 50, 60, 80, 100, or 250.

Returns
This function does not return any value.

Return type
None

src.dynamic_boundary_conditions.rainfall.main_rainfall

Main rainfall script used to fetch and store rainfall data in the database, and to generate the requested rainfall model input for BG-Flood, etc.

Module Contents

Functions

remove_existing_rain_inputs(→ None)	Remove existing rain input files from the specified directory.
main(→ None)	Fetch and store rainfall data in the database, and generate the requested rainfall model input for BG-Flood.

Attributes

sample_polygon

src.dynamic_boundary_conditions.rainfall.main_rainfall.**remove_existing_rain_inputs**(*bg_flood_dir: pathlib.Path*) → None

Remove existing rain input files from the specified directory.

Parameters
bg_flood_dir (*pathlib.Path*) – BG-Flood model directory containing the rain input files.

Returns
This function does not return any value.

Return type
None

src.dynamic_boundary_conditions.rainfall.main_rainfall.**main**(*selected_polygon_gdf: geopandas.GeoDataFrame*, *rcp: float | None*, *time_period: str | None*, *ari: float*, *storm_length_mins: int*, *time_to_peak_mins: int | float*, *increment_mins: int*, *hyeto_method:* src.dynamic_boundary_conditions.rainfall.rainfall_enum *input_type:* src.dynamic_boundary_conditions.rainfall.rainfall_enum *log_level:* src.digitaltwin.utils.LogLevel = *LogLevel.DEBUG*) → None

Fetch and store rainfall data in the database, and generate the requested rainfall model input for BG-Flood.

Parameters

- **selected_polygon_gdf** (*gpd.GeoDataFrame*) – A GeoDataFrame representing the selected polygon, i.e., the catchment area.
- **rcp** (*Optional[float]*) – Representative Concentration Pathway (RCP) value. Valid options are 2.6, 4.5, 6.0, 8.5, or None for historical data.
- **time_period** (*Optional[str]*) – Future time period. Valid options are "2031-2050", "2081-2100", or None for historical data.
- **ari** (*float*) – Average Recurrence Interval (ARI) value. Valid options are 1.58, 2, 5, 10, 20, 30, 40, 50, 60, 80, 100, or 250.
- **storm_length_mins** (*int*) – Storm duration in minutes.
- **time_to_peak_mins** (*Union[int, float]*) – The time in minutes when rainfall is at its greatest (reaches maximum).
- **increment_mins** (*int*) – Time interval in minutes.
- **hyeto_method** (HyetoMethod) – Hyetograph method to be used. Valid options are HyetoMethod.ALT_BLOCK or HyetoMethod.CHICAGO.
- **input_type** (RainInputType) – The type of rainfall model input to be generated. Valid options are 'uniform' or 'varying', representing spatially uniform rain input (text file) or spatially varying rain input (NetCDF file).
- **log_level** (*LogLevel = LogLevel.DEBUG*) – The log level to set for the root logger. Defaults to LogLevel.DEBUG. The available logging levels and their corresponding numeric values are: - LogLevel.CRITICAL (50) - LogLevel.ERROR (40) - LogLevel.WARNING (30) - LogLevel.INFO (20) - LogLevel.DEBUG (10) - LogLevel.NOTSET (0)

Returns

This function does not return any value.

Return type

None

src.dynamic_boundary_conditions.rainfall.main_rainfall.**sample_polygon**

src.dynamic_boundary_conditions.rainfall.rainfall_data_from_hirds

Fetch rainfall data from the HIRDS website.

Module Contents

Classes

BlockStructure	Represents the layout structure of fetched rainfall data.

Functions

get_site_url_key(→ str)	Get the unique URL key of the requested rainfall site from the HIRDS website.
get_data_from_hirds(→ str)	Fetch rainfall data for the requested rainfall site from the HIRDS website.
get_layout_structure_of_data(→ List[BlockStructure])	Get the layout structure of the fetched rainfall data.
convert_to_tabular_data(→ pandas.DataFrame)	Convert the fetched rainfall data for the requested site into a Pandas DataFrame.

src.dynamic_boundary_conditions.rainfall.rainfall_data_from_hirds.**get_site_url_key**(*site_id: str*, *idf: bool*) → str

Get the unique URL key of the requested rainfall site from the HIRDS website.

Parameters

- **site_id** (*str*) – HIRDS rainfall site ID.
- **idf** (*bool*) – Set to False for rainfall depth data, and True for rainfall intensity data.

Returns

Unique URL key of the requested rainfall site.

Return type

str

src.dynamic_boundary_conditions.rainfall.rainfall_data_from_hirds.**get_data_from_hirds**(*site_id: str*, *idf: bool*) → str

Fetch rainfall data for the requested rainfall site from the HIRDS website.

Parameters

- **site_id** (*str*) – HIRDS rainfall site ID.
- **idf** (*bool*) – Set to False for rainfall depth data, and True for rainfall intensity data.

Returns

Rainfall data for the requested site as a string.

Return type

str

class src.dynamic_boundary_conditions.rainfall.rainfall_data_from_hirds.**BlockStructure**

Bases: NamedTuple

Represents the layout structure of fetched rainfall data.

skip_rows

Number of lines to skip at the start of the fetched rainfall site_data.

Type

int

rcp

There are four different representative concentration pathways (RCPs), and abbreviated as RCP2.6, RCP4.5, RCP6.0 and RCP8.5, in order of increasing radiative forcing by greenhouse gases, or nan for historical data.

Type
Optional[float]

time_period

Rainfall estimates for two future time periods (e.g. 2031-2050 or 2081-2100) for four RCPs, or None for historical data.

Type
Optional[str]

category

Historical data, Historical Standard Error or Projections (i.e. hist, hist_stderr or proj).

Type
str

skip_rows: int

rcp: float | None

time_period: str | None

category: str

`src.dynamic_boundary_conditions.rainfall.rainfall_data_from_hirds.`**`get_layout_structure_of_data`**(*site_data: str*) → List[*BlockS*

Get the layout structure of the fetched rainfall data.

Parameters
site_data (*str*) – Fetched rainfall data text string from the HIRDS website for the requested rainfall site.

Returns
List of BlockStructure named tuples representing the layout structure of the fetched rainfall data.

Return type
List[*BlockStructure*]

`src.dynamic_boundary_conditions.rainfall.rainfall_data_from_hirds.`**`convert_to_tabular_data`**(*site_data: str*, *site_id: str*, *block_structure:* Block-Struc-ture) → pandas.DataFram

Convert the fetched rainfall data for the requested site into a Pandas DataFrame.

Parameters

- **site_data** (*str*) – Fetched rainfall data text string from the HIRDS website for the requested rainfall site.

- **site_id** (*str*) – HIRDS rainfall site ID.
- **block_structure** (BlockStructure) – The layout structure of the fetched rainfall data, containing skip rows, RCP, time period, and category.

Returns
Rainfall data for the requested site in tabular format.

Return type
pd.DataFrame

src.dynamic_boundary_conditions.rainfall.rainfall_enum

Enum(s) used in the rainfall module.

Module Contents

Classes

HyetoMethod	Enum class representing different hyetograph methods.
RainInputType	Enum class representing different types of rain input used in the BG-Flood Model.

class src.dynamic_boundary_conditions.rainfall.rainfall_enum.**HyetoMethod**

Bases: enum.StrEnum

Enum class representing different hyetograph methods.

ALT_BLOCK

Alternating Block Method.

Type
str

CHICAGO

Chicago Method.

Type
str

ALT_BLOCK = 'alt_block'

CHICAGO = 'chicago'

class src.dynamic_boundary_conditions.rainfall.rainfall_enum.**RainInputType**

Bases: enum.StrEnum

Enum class representing different types of rain input used in the BG-Flood Model.

UNIFORM

Spatially uniform rain input.

Type
str

VARYING

Spatially varying rain input.

Type
str

UNIFORM = 'uniform'

VARYING = 'varying'

src.dynamic_boundary_conditions.rainfall.rainfall_model_input

Generate the requested rainfall model input for BG-Flood, which can be either spatially uniform rain input ('rain_forcing.txt' text file) or spatially varying rain input ('rain_forcing.nc' NetCDF file).

Module Contents

Functions

sites_voronoi_intersect_catchment(→ geopandas.GeoDataFrame)	Get the intersecting areas between the rainfall site coverage areas (Thiessen polygons) and the catchment area,
sites_coverage_in_catchment(→ geopandas.GeoDataFrame)	Get the intersecting areas between the rainfall site coverage areas (Thiessen polygons) and the catchment area,
mean_catchment_rainfall(→ pandas.DataFrame)	Calculate the mean catchment rainfall intensities (weighted average of gauge measurements)
spatial_uniform_rain_input(→ None)	Write the mean catchment rainfall intensities data (i.e., 'seconds' and 'rain_intensity_mmhr' columns)
create_rain_data_cube(→ xarray.Dataset)	Create rainfall intensities data cube (xarray data) for the catchment area across all durations,
spatial_varying_rain_input(→ None)	Write the rainfall intensities data cube in NetCDF format (rain_forcing.nc).
generate_rain_model_input(→ None)	Generate the requested rainfall model input for BG-Flood, either spatially uniform rain input

Attributes

log

src.dynamic_boundary_conditions.rainfall.rainfall_model_input.**log**

src.dynamic_boundary_conditions.rainfall.rainfall_model_input.**sites_voronoi_intersect_catchment**(*sites_in_c* *geopandas.GeoD* *catchment_are* *geopandas.GeoD*) → geopanda

Get the intersecting areas between the rainfall site coverage areas (Thiessen polygons) and the catchment area, i.e. return the overlapped areas.

Parameters

- **sites_in_catchment** (*gpd.GeoDataFrame*) – Rainfall site coverage areas (Thiessen polygons) that intersect or are within the catchment area.
- **catchment_area** (*gpd.GeoDataFrame*) – A GeoDataFrame representing the catchment area.

Returns

A GeoDataFrame containing the intersecting areas between the rainfall site coverage areas and the catchment area.

Return type

gpd.GeoDataFrame

src.dynamic_boundary_conditions.rainfall.rainfall_model_input.**sites_coverage_in_catchment**(*sites_in_catchme geopandas.GeoDataFra catchment_area: geopandas.GeoDataFra*) → geopandas.GeoDa

Get the intersecting areas between the rainfall site coverage areas (Thiessen polygons) and the catchment area, and calculate the size and percentage of the area covered by each rainfall site inside the catchment area.

Parameters

- **sites_in_catchment** (*gpd.GeoDataFrame*) – Rainfall sites coverage areas (Thiessen polygons) that intersect or are within the catchment area.
- **catchment_area** (*gpd.GeoDataFrame*) – A GeoDataFrame representing the catchment area.

Returns

A GeoDataFrame containing the intersecting areas between the rainfall site coverage areas and the catchment area, with calculated size and percentage of area covered by each rainfall site.

Return type

gpd.GeoDataFrame

src.dynamic_boundary_conditions.rainfall.rainfall_model_input.**mean_catchment_rainfall**(*hyetograph_data: pandas.DataFrame*, *sites_coverage: geopandas.GeoDataFrame*) → pandas.DataFrame

Calculate the mean catchment rainfall intensities (weighted average of gauge measurements) across all durations using the Thiessen polygon method.

Parameters

- **hyetograph_data** (*pd.DataFrame*) – Hyetograph intensities data for sites within the catchment area.
- **sites_coverage** (*gpd.GeoDataFrame*) – A GeoDataFrame containing information about the coverage area of each rainfall site within the catchment area, including the size and percentage of the catchment area covered by each site.

Returns
A DataFrame containing the mean catchment rainfall intensities across all durations.

Return type
pd.DataFrame

`src.dynamic_boundary_conditions.rainfall.rainfall_model_input.`**`spatial_uniform_rain_input`**(*hyetograph_data: pandas.DataFrame*, *sites_coverage: geopandas.GeoDataFrame*, *bg_flood_dir: pathlib.Path*) → None

Write the mean catchment rainfall intensities data (i.e., 'seconds' and 'rain_intensity_mmhr' columns) into a text file named 'rain_forcing.txt'. This file can be used as spatially uniform rain input for the BG-Flood model.

Parameters

- **hyetograph_data** (*pd.DataFrame*) – Hyetograph intensities data for sites within the catchment area.
- **sites_coverage** (*gpd.GeoDataFrame*) – A GeoDataFrame containing information about the coverage area of each rainfall site within the catchment area, including the size and percentage of the catchment area covered by each site.
- **bg_flood_dir** (*pathlib.Path*) – BG-Flood model directory.

Returns
This function does not return any value.

Return type
None

`src.dynamic_boundary_conditions.rainfall.rainfall_model_input.`**`create_rain_data_cube`**(*hyetograph_data: pandas.DataFrame*, *sites_coverage: geopandas.GeoDataFrame*) → xarray.Dataset

Create rainfall intensities data cube (xarray data) for the catchment area across all durations, i.e. convert rainfall intensities vector data into rasterized xarray data.

Parameters

- **hyetograph_data** (*pd.DataFrame*) – Hyetograph intensities data for sites within the catchment area.

- **sites_coverage** (*gpd.GeoDataFrame*) – A GeoDataFrame containing information about the coverage area of each rainfall site within the catchment area, including the size and percentage of the catchment area covered by each site.

Returns
Rainfall intensities data cube in the form of xarray dataset.

Return type
xr.Dataset

src.dynamic_boundary_conditions.rainfall.rainfall_model_input.**spatial_varying_rain_input**(*hyetograph_data: pandas.DataFrame*, *sites_coverage: geopandas.GeoDataFram*, *bg_flood_dir: pathlib.Path*) → None

Write the rainfall intensities data cube in NetCDF format (rain_forcing.nc). This file can be used as spatially varying rain input for the BG-Flood model.

Parameters

- **hyetograph_data** (*pd.DataFrame*) – Hyetograph intensities data for sites within the catchment area.
- **sites_coverage** (*gpd.GeoDataFrame*) – A GeoDataFrame containing information about the coverage area of each rainfall site within the catchment area, including the size and percentage of the catchment area covered by each site.
- **bg_flood_dir** (*pathlib.Path*) – BG-Flood model directory.

Returns
This function does not return any value.

Return type
None

src.dynamic_boundary_conditions.rainfall.rainfall_model_input.**generate_rain_model_input**(*hyetograph_data: pandas.DataFrame*, *sites_coverage: geopandas.GeoDataFrame,* *bg_flood_dir: pathlib.Path*, *input_type:* src.dynamic_bound → None

Generate the requested rainfall model input for BG-Flood, either spatially uniform rain input ('rain_forcing.txt' text file) or spatially varying rain input ('rain_forcing.nc' NetCDF file).

Parameters

- **hyetograph_data** (*pd.DataFrame*) – Hyetograph intensities data for sites within the catchment area.
- **sites_coverage** (*gpd.GeoDataFrame*) – A GeoDataFrame containing information about the coverage area of each rainfall site within the catchment area, including the size and percentage of the catchment area covered by each site.
- **bg_flood_dir** (*pathlib.Path*) – BG-Flood model directory.
- **input_type** (RainInputType) – The type of rainfall model input to be generated. Valid options are 'uniform' or 'varying', representing spatially uniform rain input (text file) or spatially varying rain input (NetCDF file).

Returns
This function does not return any value.

Return type
None

src.dynamic_boundary_conditions.rainfall.rainfall_sites

Fetch rainfall sites data from the HIRDS website and store it in the database.

Module Contents

Functions

get_rainfall_sites_data(→ str)	Get rainfall sites data from the HIRDS website.
get_rainfall_sites_in_df(→ geopandas.GeoDataFrame)	Get rainfall sites data from the HIRDS website and transform it into a GeoDataFrame.
rainfall_sites_to_db(→ None)	Store rainfall sites data from the HIRDS website in the database.

Attributes

log

src.dynamic_boundary_conditions.rainfall.rainfall_sites.**log**

src.dynamic_boundary_conditions.rainfall.rainfall_sites.**get_rainfall_sites_data**() → str

Get rainfall sites data from the HIRDS website.

Returns
The rainfall sites data as a string.

Return type
str

`src.dynamic_boundary_conditions.rainfall.rainfall_sites.`**`get_rainfall_sites_in_df`**`()` → geopandas.GeoDataFrame

Get rainfall sites data from the HIRDS website and transform it into a GeoDataFrame.

Returns
A GeoDataFrame containing the rainfall sites data.

Return type
gpd.GeoDataFrame

`src.dynamic_boundary_conditions.rainfall.rainfall_sites.`**`rainfall_sites_to_db`**`(`*engine: sqlalchemy.engine.Engine*`)` → None

Store rainfall sites data from the HIRDS website in the database.

Parameters
engine (*Engine*) – The engine used to connect to the database.

Returns
This function does not return any value.

Return type
None

src.dynamic_boundary_conditions.rainfall.thiessen_polygons

Calculate the area covered by each rainfall site throughout New Zealand and store it in the database. Retrieve the coverage areas (Thiessen polygons) for all rainfall sites located within the catchment area.

Module Contents

Functions

get_sites_within_aoi(→ geopandas.GeoDataFrame)	Get all rainfall sites within the area of interest from the database and return the required data as a
thiessen_polygons_calculator(→ geopandas.GeoDataFrame)	Create Thiessen polygons for rainfall sites within the area of interest and calculate the area covered by each
thiessen_polygons_to_db(→ None)	Store the data representing the Thiessen polygons, site information, and the area covered by
thiessen_polygons_from_db(→ geopandas.GeoDataFrame)	Get the coverage areas (Thiessen polygons) of all rainfall sites that intersect or are within the

Attributes

log

`src.dynamic_boundary_conditions.rainfall.thiessen_polygons.`**`log`**

src.dynamic_boundary_conditions.rainfall.thiessen_polygons.**get_sites_within_aoi**(*engine: sqlalchemy.engine.Engine*, *area_of_interest: geopandas.GeoDataFrame*) → geopandas.GeoDataFrame

Get all rainfall sites within the area of interest from the database and return the required data as a GeoDataFrame.

Parameters

- **engine** (*Engine*) – The engine used to connect to the database.
- **area_of_interest** (*gpd.GeoDataFrame*) – A GeoDataFrame representing the area of interest.

Returns

A GeoDataFrame containing the rainfall sites within the area of interest.

Return type

gpd.GeoDataFrame

src.dynamic_boundary_conditions.rainfall.thiessen_polygons.**thiessen_polygons_calculator**(*area_of_interest: geopandas.GeoDataFrame*, *sites_in_aoi: geopandas.GeoDataFrame*) → geopandas.GeoDataFrame

Create Thiessen polygons for rainfall sites within the area of interest and calculate the area covered by each rainfall site.

Parameters

- **area_of_interest** (*gpd.GeoDataFrame*) – A GeoDataFrame representing the area of interest.
- **sites_in_aoi** (*gpd.GeoDataFrame*) – Rainfall sites within the area of interest.

Returns

A GeoDataFrame containing the Thiessen polygons, site information, and area covered by each rainfall site.

Return type

gpd.GeoDataFrame

Raises

ValueError –

- If the provided 'area_of_interest' GeoDataFrame does not contain any data.
- If the provided 'sites_in_aoi' GeoDataFrame does not contain any data.

`src.dynamic_boundary_conditions.rainfall.thiessen_polygons.`**`thiessen_polygons_to_db`**(*engine: sqlalchemy.engine.Engine*, *area_of_interest: geopandas.GeoDataFrame*, *sites_in_aoi: geopandas.GeoDataFrame*) → None

Store the data representing the Thiessen polygons, site information, and the area covered by each rainfall site in the database.

Parameters

- **engine** (*Engine*) – The engine used to connect to the database.
- **area_of_interest** (*gpd.GeoDataFrame*) – A GeoDataFrame representing the area of interest.
- **sites_in_aoi** (*gpd.GeoDataFrame*) – Rainfall sites within the area of interest.

Returns
This function does not return any value.

Return type
None

`src.dynamic_boundary_conditions.rainfall.thiessen_polygons.`**`thiessen_polygons_from_db`**(*engine: sqlalchemy.engine.Engi*, *catchment_area: geopandas.GeoDataFrame*) → geopandas.GeoDataFra

Get the coverage areas (Thiessen polygons) of all rainfall sites that intersect or are within the specified catchment area.

Parameters

- **engine** (*Engine*) – The engine used to connect to the database.
- **catchment_area** (*gpd.GeoDataFrame*) – A GeoDataFrame representing the catchment area.

Returns
A GeoDataFrame containing the coverage areas (Thiessen polygons) of rainfall sites within the catchment area.

Return type
gpd.GeoDataFrame

src.dynamic_boundary_conditions.river

Submodules

src.dynamic_boundary_conditions.river.align_rec1_osm

This script handles the task of obtaining data for REC1 river inflow segments whose boundary points align with the boundary points of OpenStreetMap (OSM) waterways within a specified distance threshold.

Module Contents

Functions

get_rec1_network_data_on_bbox(→ geopandas.GeoDataFrame)	Obtain REC1 river network data that intersects with the catchment area boundary, along with the corresponding
get_single_intersect_inflows(→ geopandas.GeoDataFrame)	Identifies REC1 river segments that intersect the catchment boundary once, then retrieves the segments
get_exploded_multi_intersect(→ geopandas.GeoDataFrame)	Identifies REC1 river segments that intersect the catchment boundary multiple times,
determine_multi_intersect_inflow_index(→ int)	Determines the index that represents the position of the first inflow boundary point along a REC1 river segment.
categorize_exploded_multi_intersect(→ Dict[int, ...)	Categorizes boundary points of REC1 river segments that intersect the catchment boundary multiple times into
get_multi_intersect_inflows(→ geopandas.GeoDataFrame)	Identifies REC1 river segments that intersect the catchment boundary multiple times, then retrieves the segments
get_rec1_inflows_on_bbox(→ geopandas.GeoDataFrame)	Obtain REC1 river segments that are inflows into the specified catchment area, along with their corresponding
get_osm_waterways_on_bbox(→ geopandas.GeoDataFrame)	Retrieve OpenStreetMap (OSM) waterway data that intersects with the catchment area boundary,
align_rec1_with_osm(→ geopandas.GeoDataFrame)	Aligns the boundary points of REC1 river inflow segments with the boundary points of OpenStreetMap (OSM) waterways
get_rec1_inflows_aligned_to_osm(→ geopandas.GeoDataFrame)	Obtain data for REC1 river inflow segments whose boundary points align with the boundary points of

src.dynamic_boundary_conditions.river.align_rec1_osm.**get_rec1_network_data_on_bbox**(*engine: sqlalchemy.engine.Engine, catchment_area: geopandas.GeoDataFrame, rec1_network_data: geopandas.GeoDataFrame*) → geopandas.GeoDataFrame

Obtain REC1 river network data that intersects with the catchment area boundary, along with the corresponding intersection points on the boundary.

Parameters

- **engine** (*Engine*) – The engine used to connect to the database.
- **catchment_area** (*gpd.GeoDataFrame*) – A GeoDataFrame representing the catchment area.
- **rec1_network_data** (*gpd.GeoDataFrame*) – A GeoDataFrame containing the REC1 river network data.

Returns

A GeoDataFrame containing REC1 river network data that intersects with the catchment area boundary, along with the corresponding intersection points on the boundary.

Return type

gpd.GeoDataFrame

src.dynamic_boundary_conditions.river.align_rec1_osm.**get_single_intersect_inflows**(*rec1_on_bbox: geopandas.GeoDataFrame*) → geopandas.GeoDataFrame

Identifies REC1 river segments that intersect the catchment boundary once, then retrieves the segments that are inflows into the catchment area, along with their corresponding inflow boundary points.

Parameters

rec1_on_bbox (*gpd.GeoDataFrame*) – A GeoDataFrame containing REC1 river network data that intersects with the catchment area boundary, along with the corresponding intersection points on the boundary.

Returns

A GeoDataFrame containing the REC1 river segments that intersect the catchment boundary once and are inflows into the catchment area, along with their corresponding inflow boundary points.

Return type

gpd.GeoDataFrame

src.dynamic_boundary_conditions.river.align_rec1_osm.**get_exploded_multi_intersect**(*rec1_on_bbox: geopandas.GeoDataFrame*) → geopandas.GeoDataFrame

Identifies REC1 river segments that intersect the catchment boundary multiple times, transforms MultiPoint geometries into individual Point geometries (boundary points), calculates the distance along the river segment for each boundary point, and adds a new column containing boundary points sorted by their distance along the river.

Parameters

rec1_on_bbox (*gpd.GeoDataFrame*) – A GeoDataFrame containing REC1 river network data that intersects with the catchment area boundary, along with the corresponding intersection points on the boundary.

Returns

A GeoDataFrame containing the REC1 river segments that intersect the catchment boundary multiple times, along with the corresponding intersection points on the boundary, sorted by distance along the river.

Return type

gpd.GeoDataFrame

src.dynamic_boundary_conditions.river.align_rec1_osm.**determine_multi_intersect_inflow_index**(*multi_intersect pandas.Series*) → int

Determines the index that represents the position of the first inflow boundary point along a REC1 river segment.

Parameters
: **multi_intersect_row** (*pd.Series*) – A REC1 river segment that intersects the catchment boundary multiple times, along with the corresponding intersection points on the boundary, sorted by distance along the river.

Returns
: An integer that represents the position of the first inflow boundary point along a REC1 river segment.

Return type
: int

Raises
: **ValueError** – If the index that represents the position of the first inflow boundary point along a REC1 river segment cannot be determined.

src.dynamic_boundary_conditions.river.align_rec1_osm.**categorize_exploded_multi_intersect**(*multi_intersect: geopandas.GeoDataFram*) → Dict[int, Dict[str, List[shapely.geome

Categorizes boundary points of REC1 river segments that intersect the catchment boundary multiple times into 'inflow' and 'outflow' based on their sequential positions along the river segment etc.

Parameters
: **multi_intersect** (*gpd.GeoDataFrame*) – A GeoDataFrame containing the REC1 river segments that intersect the catchment boundary multiple times, along with the corresponding intersection points on the boundary, sorted by distance along the river.

Returns
: A dictionary where the keys represent the 'objectid' values of REC1 river segments, and the values are dictionaries. Each of these dictionaries contains two lists: 'inflow' and 'outflow,' which respectively represent the boundary points where water flows into and out of the catchment area.

Return type
: Dict[int, Dict[str, List[Point]]]

src.dynamic_boundary_conditions.river.align_rec1_osm.**get_multi_intersect_inflows**(*rec1_on_bbox: geopandas.GeoDataFrame*) → geopandas.GeoDataFrame

Identifies REC1 river segments that intersect the catchment boundary multiple times, then retrieves the segments that are inflows into the catchment area, along with their corresponding inflow boundary points.

Parameters
: **rec1_on_bbox** (*gpd.GeoDataFrame*) – A GeoDataFrame containing REC1 river network data

that intersects with the catchment area boundary, along with the corresponding intersection points on the boundary.

Returns
A GeoDataFrame containing the REC1 river segments that intersect the catchment boundary multiple times and are inflows into the catchment area, along with their corresponding inflow boundary points.

Return type
gpd.GeoDataFrame

`src.dynamic_boundary_conditions.river.align_rec1_osm.`**`get_rec1_inflows_on_bbox`**`(`*engine: sqlalchemy.engine.Engine*, *catchment_area: geopandas.GeoDataFrame*, *rec1_network_data: geopandas.GeoDataFrame*`)` → geopandas.GeoDataFrame

Obtain REC1 river segments that are inflows into the specified catchment area, along with their corresponding inflow boundary points.

Parameters

- **engine** (*Engine*) – The engine used to connect to the database.
- **catchment_area** (*gpd.GeoDataFrame*) – A GeoDataFrame representing the catchment area.
- **rec1_network_data** (*gpd.GeoDataFrame*) – A GeoDataFrame containing the REC1 river network data.

Returns
A GeoDataFrame containing REC1 river segments that are inflows into the catchment area, along with their corresponding inflow boundary points.

Return type
gpd.GeoDataFrame

`src.dynamic_boundary_conditions.river.align_rec1_osm.`**`get_osm_waterways_on_bbox`**`(`*engine: sqlalchemy.engine.Engine*, *catchment_area: geopandas.GeoDataFrame*`)` → geopandas.GeoDataFrame

Retrieve OpenStreetMap (OSM) waterway data that intersects with the catchment area boundary, along with the corresponding intersection points on the boundary.

Parameters

- **engine** (*Engine*) – The engine used to connect to the database.
- **catchment_area** (*gpd.GeoDataFrame*) – A GeoDataFrame representing the catchment area.

Returns
A GeoDataFrame containing OpenStreetMap (OSM) waterway data that intersects with the catchment boundary, along with the corresponding intersection points on the boundary.

Return type
gpd.GeoDataFrame

`src.dynamic_boundary_conditions.river.align_rec1_osm.`**`align_rec1_with_osm`**(*rec1_inflows_on_bbox: geopandas.GeoDataFrame*, *osm_waterways_on_bbox: geopandas.GeoDataFrame*, *distance_m: int = 300*) → geopandas.GeoDataFrame

Aligns the boundary points of REC1 river inflow segments with the boundary points of OpenStreetMap (OSM) waterways within a specified distance threshold.

Parameters

- **rec1_inflows_on_bbox** (*gpd.GeoDataFrame*) – A GeoDataFrame containing REC1 river network segments where water flows into the catchment area, along with their corresponding inflow boundary points.
- **osm_waterways_on_bbox** (*gpd.GeoDataFrame*) – A GeoDataFrame containing OpenStreetMap (OSM) waterway data that intersects with the catchment boundary, along with the corresponding intersection points on the boundary.
- **distance_m** (*int = 300*) – Distance threshold in meters for spatial proximity matching. The default value is 300 meters.

Returns
A GeoDataFrame containing the boundary points of REC1 river inflow segments aligned with the boundary points of OpenStreetMap (OSM) waterways within a specified distance threshold.

Return type
gpd.GeoDataFrame

`src.dynamic_boundary_conditions.river.align_rec1_osm.`**`get_rec1_inflows_aligned_to_osm`**(*engine: sqlalchemy.engine.Engi*, *catchment_area: geopandas.GeoDataFrame*, *rec1_network_data: geopandas.GeoDataFrame*, *distance_m: int = 300*) → geopandas.GeoDataFrar

Obtain data for REC1 river inflow segments whose boundary points align with the boundary points of OpenStreetMap (OSM) waterways within a specified distance threshold.

Parameters

- **engine** (*Engine*) – The engine used to connect to the database.
- **catchment_area** (*gpd.GeoDataFrame*) – A GeoDataFrame representing the catchment area.
- **rec1_network_data** (*gpd.GeoDataFrame*) – A GeoDataFrame containing the REC1 river network data.
- **distance_m** (*int = 300*) – Distance threshold in meters for spatial proximity matching. The default value is 300 meters.

Returns
A GeoDataFrame containing data for REC1 river inflow segments whose boundary points align with the boundary points of OpenStreetMap (OSM) waterways within a specified distance threshold.

Return type
gpd.GeoDataFrame

src.dynamic_boundary_conditions.river.hydrograph

This script handles the task of obtaining REC1 river inflow scenario data, whether it's Mean Annual Flood (MAF) or Average Recurrence Interval (ARI)-based, and generates corresponding hydrograph data for the requested scenarios.

Module Contents

Functions

clean_rec1_inflow_data(→ geopandas.GeoDataFrame)	Selects and renames specific columns that represent REC1 river inflow data from the input GeoDataFrame.
extract_valid_ari_values(→ List[int])	Extracts valid ARI (Annual Recurrence Interval) values from the column names of the REC1 river inflow data.
get_rec1_inflow_scenario_data(→ geopandas.GeoDataFrame)	Obtain the requested REC1 river inflow scenario data, which can be either Mean Annual Flood (MAF)-based or
get_hydrograph_data(→ geopandas.GeoDataFrame)	Generate hydrograph data for the requested REC1 river inflow scenario.

src.dynamic_boundary_conditions.river.hydrograph.**clean_rec1_inflow_data**(*rec1_inflows_w_input_points: geopandas.GeoDataFrame*) → geopandas.GeoDataFrame

Selects and renames specific columns that represent REC1 river inflow data from the input GeoDataFrame.

Parameters
rec1_inflows_w_input_points (*gpd.GeoDataFrame*) – A GeoDataFrame containing data for REC1 river inflow segments whose boundary points align with the boundary points of OpenStreetMap (OSM) waterways within a specified distance threshold, along with their corresponding river input points used in the BG-Flood model.

Returns
A GeoDataFrame with selected and renamed columns representing REC1 river inflow data.

Return type
gpd.GeoDataFrame

`src.dynamic_boundary_conditions.river.hydrograph.`**`extract_valid_ari_values`**(*rec1_inflow_data: geopandas.GeoDataFrame*) → List[int]

Extracts valid ARI (Annual Recurrence Interval) values from the column names of the REC1 river inflow data.

Parameters
`rec1_inflow_data` (*`gpd.GeoDataFrame`*) – A GeoDataFrame containing REC1 river inflow data with column names that include ARI values.

Returns
A list of valid ARI values extracted from the column names of the REC1 river inflow data.

Return type
List[int]

`src.dynamic_boundary_conditions.river.hydrograph.`**`get_rec1_inflow_scenario_data`**(*rec1_inflows_w_input_points: geopandas.GeoDataFrame*, *maf: bool = True*, *ari: int | None = None*, *bound: src.dynamic_boundary_conditio = BoundType.MIDDLE*) → geopandas.GeoDataFrame

Obtain the requested REC1 river inflow scenario data, which can be either Mean Annual Flood (MAF)-based or Average Recurrence Interval (ARI)-based scenario data.

Parameters

- **`rec1_inflows_w_input_points`** (*`gpd.GeoDataFrame`*) – A GeoDataFrame containing data for REC1 river inflow segments whose boundary points align with the boundary points of OpenStreetMap (OSM) waterways within a specified distance threshold, along with their corresponding river input points used in the BG-Flood model.
- **`maf`** (*`bool = True`*) – Set to True to obtain MAF-based scenario data or False to obtain ARI-based scenario data.
- **`ari`** (*`Optional[int] = None`*) – The Average Recurrence Interval (ARI) value. Valid options are 5, 10, 20, 50, 100, or 1000. Mandatory when 'maf' is set to False, and should be set to None when 'maf' is set to True.
- **`bound`** (*`BoundType = BoundType.MIDDLE`*) – Set the type of bound (estimate) for the REC1 river inflow scenario data. Valid options include: 'BoundType.LOWER', 'BoundType.MIDDLE', or 'BoundType.UPPER'.

Returns
A GeoDataFrame containing the requested REC1 river inflow scenario data.

Return type
gpd.GeoDataFrame

Raises
`ValueError` –

- If ‘ari’ is provided when ‘maf’ is set to True (i.e. ‘maf’ is True and ‘ari’ is not set to None).
- If ‘ari’ is not provided when ‘maf’ is set to False (i.e. ‘maf’ is False and ‘ari’ is set to None).
- If an invalid ‘ari’ value is provided.

`src.dynamic_boundary_conditions.river.hydrograph.`**`get_hydrograph_data`**(*rec1_inflows_w_input_points: geopandas.GeoDataFrame*, *flow_length_mins: int*, *time_to_peak_mins: int | float*, *maf: bool = True*, *ari: int | None = None*, *bound: src.dynamic_boundary_conditions.river.river = BoundType.MIDDLE*) → geopandas.GeoDataFrame

Generate hydrograph data for the requested REC1 river inflow scenario.

Parameters

- **rec1_inflows_w_input_points** (*gpd.GeoDataFrame*) – A GeoDataFrame containing data for REC1 river inflow segments whose boundary points align with the boundary points of OpenStreetMap (OSM) waterways within a specified distance threshold, along with their corresponding river input points used in the BG-Flood model.
- **flow_length_mins** (*int*) – Duration of the river flow in minutes.
- **time_to_peak_mins** (*Union[int, float]*) – The time in minutes when flow is at its greatest (reaches maximum).
- **maf** (*bool = True*) – Set to True to obtain MAF-based scenario data or False to obtain ARI-based scenario data.
- **ari** (*Optional[int] = None*) – The Average Recurrence Interval (ARI) value. Valid options are 5, 10, 20, 50, 100, or 1000. Mandatory when ‘maf’ is set to False, and should be set to None when ‘maf’ is set to True.
- **bound** (*BoundType = BoundType.MIDDLE*) – Set the type of bound (estimate) for the REC1 river inflow scenario data. Valid options include: ‘BoundType.LOWER’, ‘BoundType.MIDDLE’, or ‘BoundType.UPPER’.

Returns

A GeoDataFrame containing hydrograph data for the requested REC1 river inflow scenario.

Return type

gpd.GeoDataFrame

Raises

ValueError – If the specified ‘time_to_peak_mins’ is less than half of the river flow duration.

src.dynamic_boundary_conditions.river.main_river

Main river script used to read and store REC1 data in the database, fetch OSM waterways data, create a river network and its associated data, and generate the requested river model input for BG-Flood etc.

Module Contents

Functions

retrieve_hydro_dem_info(→ Tuple[xarray.Dataset, ...)	Retrieves the Hydrologically Conditioned DEM (Hydro DEM) data, along with its spatial extent and resolution,
get_hydro_dem_boundary_lines(→ geopandas.GeoDataFrame)	Get the boundary lines of the Hydrologically Conditioned DEM.
remove_existing_river_inputs(→ None)	Remove existing river input files from the specified directory.
main(→ None)	Read and store REC1 data in the database, fetch OSM waterways data, create a river network and its associated data,

Attributes

sample_polygon

src.dynamic_boundary_conditions.river.main_river.**retrieve_hydro_dem_info**(*engine: sqlalchemy.engine.Engine*, *catchment_area: geopandas.GeoDataFrame*) → Tuple[xarray.Dataset, shapely.geometry.LineString, int | float]

Retrieves the Hydrologically Conditioned DEM (Hydro DEM) data, along with its spatial extent and resolution, for the specified catchment area.

Parameters

- **engine** (*Engine*) – The engine used to connect to the database.
- **catchment_area** (*gpd.GeoDataFrame*) – A GeoDataFrame representing the catchment area.

Returns

A tuple containing the Hydro DEM data as a xarray Dataset, the spatial extent of the Hydro DEM as a LineString, and the resolution of the Hydro DEM as either an integer or a float.

Return type

Tuple[xr.Dataset, LineString, Union[int, float]]

src.dynamic_boundary_conditions.river.main_river.**get_hydro_dem_boundary_lines**(*engine: sqlalchemy.engine.Engine*, *catchment_area: geopandas.GeoDataFrame*) → geopandas.GeoDataFrame

Get the boundary lines of the Hydrologically Conditioned DEM.

Parameters

- **engine** (*Engine*) – The engine used to connect to the database.
- **catchment_area** (*gpd.GeoDataFrame*) – A GeoDataFrame representing the catchment area.

Returns
A GeoDataFrame containing the boundary lines of the Hydrologically Conditioned DEM.

Return type
gpd.GeoDataFrame

src.dynamic_boundary_conditions.river.main_river.**remove_existing_river_inputs**(*bg_flood_dir: pathlib.Path*) → None

Remove existing river input files from the specified directory.

Parameters
bg_flood_dir (*pathlib.Path*) – The BG-Flood model directory containing the river input files.

Returns
This function does not return any value.

Return type
None

src.dynamic_boundary_conditions.river.main_river.**main**(*selected_polygon_gdf: geopandas.GeoDataFrame*, *flow_length_mins: int*, *time_to_peak_mins: int | float*, *maf: bool = True*, *ari: int | None = None*, *bound: src.dynamic_boundary_conditions.river.river_enum.BoundType = BoundType.MIDDLE*, *log_level: src.digitaltwin.utils.LogLevel = LogLevel.DEBUG*) → None

Read and store REC1 data in the database, fetch OSM waterways data, create a river network and its associated data, and generate the requested river model input for BG-Flood.

Parameters

- **selected_polygon_gdf** (*gpd.GeoDataFrame*) – A GeoDataFrame representing the selected polygon, i.e., the catchment area.
- **flow_length_mins** (*int*) – Duration of the river flow in minutes.
- **time_to_peak_mins** (*Union[int, float]*) – The time in minutes when flow is at its greatest (reaches maximum).

- **maf** (*bool = True*) – Set to True to obtain MAF-based scenario data or False to obtain ARI-based scenario data.
- **ari** (*Optional[int] = None*) – The Average Recurrence Interval (ARI) value. Valid options are 5, 10, 20, 50, 100, or 1000. Mandatory when 'maf' is set to False, and should be set to None when 'maf' is set to True.
- **bound** (*BoundType = BoundType.MIDDLE*) – Set the type of bound (estimate) for the REC1 river inflow scenario data. Valid options include: 'BoundType.LOWER', 'BoundType.MIDDLE', or 'BoundType.UPPER'.
- **log_level** (*LogLevel = LogLevel.DEBUG*) – The log level to set for the root logger. Defaults to LogLevel.DEBUG. The available logging levels and their corresponding numeric values are: - LogLevel.CRITICAL (50) - LogLevel.ERROR (40) - LogLevel.WARNING (30) - LogLevel.INFO (20) - LogLevel.DEBUG (10) - LogLevel.NOTSET (0)

Returns
This function does not return any value.

Return type
None

`src.dynamic_boundary_conditions.river.main_river.`**`sample_polygon`**

src.dynamic_boundary_conditions.river.osm_waterways

This script handles the fetching of OpenStreetMap (OSM) waterways data for the defined catchment area.

Module Contents

Functions

configure_osm_cache(→ None)	Change the directory for storing the OSM cache files.
fetch_osm_waterways(→ geopandas.GeoDataFrame)	Fetches OpenStreetMap (OSM) waterways data for the specified catchment area.
get_osm_waterways_data(→ geopandas.GeoDataFrame)	Fetches OpenStreetMap (OSM) waterways data for the specified catchment area.

`src.dynamic_boundary_conditions.river.osm_waterways.`**`configure_osm_cache`**`()` → None

Change the directory for storing the OSM cache files.

Returns
This function does not return any value.

Return type
None

`src.dynamic_boundary_conditions.river.osm_waterways.`**`fetch_osm_waterways`**`(`*catchment_area: geopandas.GeoDataFrame*`)` → geopandas.GeoDataFrame

Fetches OpenStreetMap (OSM) waterways data for the specified catchment area.

Parameters
`catchment_area` (*gpd.GeoDataFrame*) – A GeoDataFrame representing the catchment area.

Returns
A GeoDataFrame containing the retrieved OSM waterways data for the specified catchment area.

Return type
gpd.GeoDataFrame

`src.dynamic_boundary_conditions.river.osm_waterways.`**`get_osm_waterways_data`**(*catchment_area: geopandas.GeoDataFrame*) → geopandas.GeoDataFrame

Fetches OpenStreetMap (OSM) waterways data for the specified catchment area. Only LineString geometries representing waterways of type "river" or "stream" are included.

Parameters
`catchment_area` (*gpd.GeoDataFrame*) – A GeoDataFrame representing the catchment area.

Returns
A GeoDataFrame containing only LineString geometries representing waterways of type "river" or "stream".

Return type
gpd.GeoDataFrame

`src.dynamic_boundary_conditions.river.river_data_to_from_db`

This script handles the following tasks: reading REC1 data from the NIWA REC1 dataset, storing REC1 data within the database, and retrieving REC1 data enriched with sea-draining catchment information from the database.

Module Contents

Functions

get_niwa_rec1_data(→ geopandas.GeoDataFrame)	Reads REC1 data from the NIWA REC1 dataset and returns a GeoDataFrame.
store_rec1_data_to_db(→ None)	Store REC1 data to the database.
get_sdc_data_from_db(→ geopandas.GeoDataFrame)	Retrieve sea-draining catchment data from the database that intersects with the given catchment area.
get_rec1_data_with_sdc_from_db(→ geopandas.GeoDataFrame)	Retrieve REC1 data from the database for the specified catchment area with an additional column that identifies

Attributes

log

`src.dynamic_boundary_conditions.river.river_data_to_from_db.`**`log`**

`src.dynamic_boundary_conditions.river.river_data_to_from_db.`**`get_niwa_rec1_data`**`()` → geopandas.GeoDataFrame

Reads REC1 data from the NIWA REC1 dataset and returns a GeoDataFrame.

Returns
A GeoDataFrame containing the REC1 data from the NZ REC1 dataset.

Return type
gpd.GeoDataFrame

Raises
`FileNotFoundError` – If the REC1 data directory does not exist or if there are no Shapefiles in the specified directory.

`src.dynamic_boundary_conditions.river.river_data_to_from_db.`**`store_rec1_data_to_db`**`(`*engine: sqlalchemy.engine.Engine*`)` → None

Store REC1 data to the database.

Parameters
`engine` (*`Engine`*) – The engine used to connect to the database.

Returns
This function does not return any value.

Return type
None

`src.dynamic_boundary_conditions.river.river_data_to_from_db.`**`get_sdc_data_from_db`**`(`*engine: sqlalchemy.engine.Engine, catchment_area: geopandas.GeoDataFrame*`)` → geopandas.GeoDataFrame

Retrieve sea-draining catchment data from the database that intersects with the given catchment area.

Parameters

- **`engine`** (*`Engine`*) – The engine used to connect to the database.
- **`catchment_area`** (*`gpd.GeoDataFrame`*) – A GeoDataFrame representing the catchment area.

Returns
A GeoDataFrame containing sea-draining catchment data that intersects with the given catchment area.

Return type
gpd.GeoDataFrame

src.dynamic_boundary_conditions.river.river_data_to_from_db.**get_rec1_data_with_sdc_from_db**(*engine: sqlalchemy.engi catchment_area: geopandas.GeoDataFra river_network_i int*) → geopandas.GeoI

Retrieve REC1 data from the database for the specified catchment area with an additional column that identifies the associated sea-draining catchment for each REC1 geometry. Simultaneously, identify the REC1 geometries that do not fully reside within sea-draining catchments and proceed to add these excluded REC1 geometries to the appropriate database table.

Parameters

- **engine** (*Engine*) – The engine used to connect to the database.
- **catchment_area** (*gpd.GeoDataFrame*) – A GeoDataFrame representing the catchment area.
- **river_network_id** (*int*) – An identifier for the river network associated with the current run.

Returns

A GeoDataFrame containing the retrieved REC1 data for the specified catchment area with an additional column that identifies the associated sea-draining catchment for each REC1 geometry.

Return type

gpd.GeoDataFrame

src.dynamic_boundary_conditions.river.river_enum

Enum(s) used in the river module.

Module Contents

Classes

BoundType	Enum class representing different types of estimates used in river flow scenarios.

class src.dynamic_boundary_conditions.river.river_enum.**BoundType**

Bases: enum.StrEnum

Enum class representing different types of estimates used in river flow scenarios.

LOWER

Lower bound of a confidence interval.

Type

str

MIDDLE

Point estimate or sample mean.

Type
str

UPPER

Upper bound of a confidence interval.

Type
str

LOWER = 'lower'

MIDDLE = 'middle'

UPPER = 'upper'

src.dynamic_boundary_conditions.river.river_inflows

This script handles the task of obtaining REC1 river inflow data along with the corresponding river input points used for the BG-Flood model.

Module Contents

Functions

get_elevations_near_rec1_entry_point(...)	Extracts elevation values and their corresponding coordinates from the Hydrologically Conditioned DEM in the
get_min_elevation_river_input_point(...)	Locate the river input point with the lowest elevation, used for BG-Flood model river input, from the
get_rec1_inflows_with_input_points(...)	Obtain data for REC1 river inflow segments whose boundary points align with the boundary points of

src.dynamic_boundary_conditions.river.river_inflows.**get_elevations_near_rec1_entry_point**(*rec1_inflows_row: pandas.Series, hydro_dem: xarray.Dataset*) → geopandas.GeoDat

Extracts elevation values and their corresponding coordinates from the Hydrologically Conditioned DEM in the vicinity of the entry point of the REC1 river inflow segment.

Parameters

- **rec1_inflows_row** (*pd.Series*) – Represents data pertaining to an individual REC1 river inflow segment, including its entry point into the catchment area and the boundary line it aligns with.

- **hydro_dem** (*xr.Dataset*) – Hydrologically Conditioned DEM for the catchment area.

Returns
A GeoDataFrame containing elevation values and their corresponding coordinates extracted from the Hydrologically Conditioned DEM in the vicinity of the entry point of the REC1 river inflow segment.

Return type
gpd.GeoDataFrame

src.dynamic_boundary_conditions.river.river_inflows.**get_min_elevation_river_input_point**(*rec1_inflows_row: pandas.Series*, *hydro_dem: xarray.Dataset*) → geopandas.GeoDataFrame

Locate the river input point with the lowest elevation, used for BG-Flood model river input, from the Hydrologically Conditioned DEM for the specific REC1 river inflow segment.

Parameters

- **rec1_inflows_row** (*pd.Series*) – Represents data pertaining to an individual REC1 river inflow segment, including its entry point into the catchment area and the boundary line it aligns with.
- **hydro_dem** (*xr.Dataset*) – Hydrologically Conditioned DEM for the catchment area.

Returns
A GeoDataFrame containing the river input point with the lowest elevation, used for BG-Flood model river input, from the Hydrologically Conditioned DEM for the specific REC1 river inflow segment.

Return type
gpd.GeoDataFrame

src.dynamic_boundary_conditions.river.river_inflows.**get_rec1_inflows_with_input_points**(*engine: sqlalchemy.engine.En*, *catchment_area: geopandas.GeoDataFrame*, *rec1_network_data: geopandas.GeoDataFrame*, *distance_m: int = 300*) → geopandas.GeoDataF

Obtain data for REC1 river inflow segments whose boundary points align with the boundary points of OpenStreetMap (OSM) waterways within a specified distance threshold, along with their corresponding river input points used for the BG-Flood model.

Parameters

- **engine** (*Engine*) – The engine used to connect to the database.
- **catchment_area** (*gpd.GeoDataFrame*) – A GeoDataFrame representing the catchment area.
- **rec1_network_data** (*gpd.GeoDataFrame*) – A GeoDataFrame containing the REC1 river network data.
- **distance_m** (*int = 300*) – Distance threshold in meters for spatial proximity matching. The default value is 300 meters.

Returns

A GeoDataFrame containing data for REC1 river inflow segments whose boundary points align with the boundary points of OpenStreetMap (OSM) waterways within a specified distance threshold, along with their corresponding river input points used for the BG-Flood model.

Return type

gpd.GeoDataFrame

src.dynamic_boundary_conditions.river.river_model_input

This script handles the task of generating the requested river model inputs for BG-Flood.

Module Contents

Functions

generate_river_model_input(→ None)	Generate the requested river model inputs for BG-Flood.

Attributes

log

src.dynamic_boundary_conditions.river.river_model_input.**log**

src.dynamic_boundary_conditions.river.river_model_input.**generate_river_model_input**(*bg_flood_dir: pathlib.Path*, *hydrograph_data: geopandas.GeoDataFrame*) → None

Generate the requested river model inputs for BG-Flood.

Parameters

- **bg_flood_dir** (*pathlib.Path*) – The BG-Flood model directory.

- **hydrograph_data** (*gpd.GeoDataFrame*) – A GeoDataFrame containing hydrograph data for the requested REC1 river inflow scenario.

Returns
This function does not return any value.

Return type
None

src.dynamic_boundary_conditions.river.river_network_for_aoi

This script processes REC1 data to construct a river network for the defined catchment area.

Module Contents

Functions

get_unique_nodes_dict(→ Dict[shapely.geometry.Point, int])	Generates a dictionary that contains the unique node coordinates in the REC1 data for the catchment area.
add_nodes_to_rec1(→ geopandas.GeoDataFrame)	Add columns for the first and last coordinates/nodes of each LineString in the REC1 data within the catchment area.
add_nodes_intersection_type(→ geopandas.GeoDataFrame)	Calculate and add an 'intersection_type' column to the GeoDataFrame that contains REC1 data with node information.
prepare_network_data_for_construction(...)	Prepares the necessary data for constructing the river network for the catchment area using the REC1 data.
add_nodes_to_network(→ None)	Add nodes to the REC1 river network along with their attributes.
add_initial_edges_to_network(→ None)	Add initial edges to the REC1 river network along with their attributes.
identify_absent_edges_to_add(→ geopandas.GeoDataFrame)	Identify edges that are absent from the REC1 river network and require addition.
add_absent_edges_to_network(→ None)	Add absent edges that are required for the current river network construction to the REC1 river network along with
add_edge_directions_to_network_data(...)	Add edge directions to the river network data based on the provided REC1 river network.
remove_unconnected_edges_from_network(...)	Remove REC1 river network edges that are not connected to their respective sea-draining catchment's end nodes.
build_rec1_river_network(→ Tuple[networkx.DiGraph, ...)	Builds a river network for the catchment area using the REC1 data.
get_rec1_river_network(→ Tuple[networkx.Graph, ...)	Retrieve or create REC1 river network for the specified catchment area.

Attributes

log

src.dynamic_boundary_conditions.river.river_network_for_aoi.**log**

src.dynamic_boundary_conditions.river.river_network_for_aoi.**get_unique_nodes_dict**(*rec1_data_w_node_coords: geopandas.GeoDataFrame*) → Dict[shapely.geometry.Point, int]

Generates a dictionary that contains the unique node coordinates in the REC1 data for the catchment area.

Parameters
rec1_data_w_node_coords (*gpd.GeoDataFrame*) – A GeoDataFrame containing the REC1 data for the catchment area with additional columns for the first and last coordinates of each LineString.

Returns
A dictionary that contains the unique node coordinates (Point objects) in the REC1 data for the catchment area.

Return type
Dict[Point, int]

src.dynamic_boundary_conditions.river.river_network_for_aoi.**add_nodes_to_rec1**(*rec1_data_with_sdc: geopandas.GeoDataFrame*) → geopandas.GeoDataFrame

Add columns for the first and last coordinates/nodes of each LineString in the REC1 data within the catchment area.

Parameters
rec1_data_with_sdc (*gpd.GeoDataFrame*) – A GeoDataFrame containing the REC1 data for the catchment area with an additional column that identifies the associated sea-draining catchment for each REC1 geometry.

Returns
A GeoDataFrame containing the REC1 data for the catchment area with additional columns for the first and last coordinates/nodes of each LineString.

Return type
gpd.GeoDataFrame

src.dynamic_boundary_conditions.river.river_network_for_aoi.**add_nodes_intersection_type**(*catchment_area: geopandas.GeoDataFrame, rec1_data_with_nod geopandas.GeoDataFrame*) → geopandas.GeoData

Calculate and add an 'intersection_type' column to the GeoDataFrame that contains REC1 data with node information.

Parameters

- **catchment_area** (*gpd.GeoDataFrame*) – A GeoDataFrame representing the catchment area.
- **rec1_data_with_nodes** (*gpd.GeoDataFrame*) – A GeoDataFrame containing the REC1 data for the catchment area with additional columns for the first and last coordinates/nodes of each LineString.

Returns
The input GeoDataFrame with the 'intersection_type' column added.

Return type
gpd.GeoDataFrame

src.dynamic_boundary_conditions.river.river_network_for_aoi.**prepare_network_data_for_construction**(*catchm geopan das.Ge rec1_d geopan das.Ge* → geopan

Prepares the necessary data for constructing the river network for the catchment area using the REC1 data.

Parameters

- **catchment_area** (*gpd.GeoDataFrame*) – A GeoDataFrame representing the catchment area.
- **rec1_data_with_sdc** (*gpd.GeoDataFrame*) – A GeoDataFrame containing the REC1 data for the catchment area with an additional column that identifies the associated sea-draining catchment for each REC1 geometry.

Returns
A GeoDataFrame containing the necessary data for constructing the river network for the catchment area.

Return type
gpd.GeoDataFrame

src.dynamic_boundary_conditions.river.river_network_for_aoi.**add_nodes_to_network**(*rec1_network: networkx.Graph*, *prepared_network_data: geopandas.GeoDataFrame*) → None

Add nodes to the REC1 river network along with their attributes.

Parameters

- **rec1_network** (*nx.Graph*) – The REC1 river network, a directed graph, to which nodes will be added.

- **prepared_network_data** (*gpd.GeoDataFrame*) – A GeoDataFrame containing the necessary data for constructing the river network for the catchment area.

Returns
This function does not return any value.

Return type
None

src.dynamic_boundary_conditions.river.river_network_for_aoi.**add_initial_edges_to_network**(*rec1_network: networkx.Graph, prepared_network_da geopandas.GeoDataFram*) → None

Add initial edges to the REC1 river network along with their attributes.

Parameters

- **rec1_network** (*nx.Graph*) – The REC1 river network, a directed graph, to which initial edges will be added.
- **prepared_network_data** (*gpd.GeoDataFrame*) – A GeoDataFrame containing the necessary data for constructing the river network for the catchment area.

Returns
This function does not return any value.

Return type
None

src.dynamic_boundary_conditions.river.river_network_for_aoi.**identify_absent_edges_to_add**(*rec1_network: networkx.Graph, prepared_network_da geopandas.GeoDataFram*) → geopandas.GeoDat

Identify edges that are absent from the REC1 river network and require addition.

Parameters

- **rec1_network** (*nx.Graph*) – The REC1 river network, a directed graph.
- **prepared_network_data** (*gpd.GeoDataFrame*) – A GeoDataFrame containing the necessary data for constructing the river network for the catchment area.

Returns
A GeoDataFrame containing edges that are absent from the REC1 river network and require addition.

Return type
gpd.GeoDataFrame

src.dynamic_boundary_conditions.river.river_network_for_aoi.**add_absent_edges_to_network**(*engine: sqlalchemy.engine.E catchment_area: geopandas.GeoDataFrame, rec1_network: networkx.Graph, prepared_network_data: geopandas.GeoDataFrame*) → None

Add absent edges that are required for the current river network construction to the REC1 river network along with their attributes.

Parameters

- **engine** (*Engine*) – The engine used to connect to the database.
- **catchment_area** (*gpd.GeoDataFrame,*) – A GeoDataFrame representing the catchment area.
- **rec1_network** (*nx.Graph*) – The REC1 river network, a directed graph, to which absent edges will be added.
- **prepared_network_data** (*gpd.GeoDataFrame*) – A GeoDataFrame containing the necessary data for constructing the river network for the catchment area.

Returns

This function does not return any value.

Return type

None

src.dynamic_boundary_conditions.river.river_network_for_aoi.**add_edge_directions_to_network_data**(*engine: sqlalchem rec1_netv int, rec1_netv networkx.Gr prepared_ne geopandas.GeoD*) → geopanda

Add edge directions to the river network data based on the provided REC1 river network. Subsequently, eliminate REC1 geometries from the network data where the edge direction is absent (None), and append these excluded REC1 geometries to the relevant database table.

Parameters

- **engine** (*Engine*) – The engine used to connect to the database.

- **rec1_network_id** (*int*) – An identifier for the river network associated with the current run.
- **rec1_network** (*nx.Graph*) – The REC1 river network, a directed graph, used to determine the edge directions.
- **prepared_network_data** (*gpd.GeoDataFrame*) – A GeoDataFrame containing the necessary data for constructing the river network for the catchment area.

Returns
A GeoDataFrame containing the updated river network data with added edge directions.

Return type
gpd.GeoDataFrame

src.dynamic_boundary_conditions.river.river_network_for_aoi.**remove_unconnected_edges_from_network**(*engine: sqlalch rec1_n int*, *rec1_n networkx.(rec1_n geopandas.Ge*) → geopan

Remove REC1 river network edges that are not connected to their respective sea-draining catchment's end nodes.

Parameters

- **engine** (*Engine*) – The engine used to connect to the database.
- **rec1_network_id** (*int*) – An identifier for the river network associated with the current run.
- **rec1_network** (*nx.Graph*) – The REC1 river network, a directed graph, used to identify edges that are connected to the end nodes of their respective sea-draining catchments.
- **rec1_network_data** (*gpd.GeoDataFrame*) – A GeoDataFrame containing the REC1 river network data with added edge directions.

Returns
A GeoDataFrame containing the modified river network data with REC1 geometries removed if they are not connected to their end nodes within their respective sea-draining catchments.

Return type
gpd.GeoDataFrame

src.dynamic_boundary_conditions.river.river_network_for_aoi.**build_rec1_river_network**(*engine: sqlalchemy.engine.Engi catchment_area: geopandas.GeoDataFrame*, *rec1_network_id: int*) → Tuple[networkx.DiGraph, geopandas.GeoDataFrar

Builds a river network for the catchment area using the REC1 data.

Parameters

- **engine** (*Engine*) – The engine used to connect to the database.
- **catchment_area** (*gpd.GeoDataFrame*) – A GeoDataFrame representing the catchment area.
- **rec1_network_id** (*int*) – An identifier for the river network associated with the current run.

Returns

A tuple containing the constructed REC1 river network, represented as a directed graph (DiGraph), along with its associated data in the form of a GeoDataFrame.

Return type

Tuple[nx.DiGraph, gpd.GeoDataFrame]

src.dynamic_boundary_conditions.river.river_network_for_aoi.**get_rec1_river_network**(*engine: sqlalchemy.engine.Engine*, *catchment_area: geopandas.GeoDataFrame*) → Tuple[networkx.Graph, geopandas.GeoDataFrame]

Retrieve or create REC1 river network for the specified catchment area.

Parameters

- **engine** (*Engine*) – The engine used to connect to the database.
- **catchment_area** (*gpd.GeoDataFrame*) – A GeoDataFrame representing the catchment area.

Returns

A tuple containing the REC1 river network as a directed graph (DiGraph) and its associated data as a GeoDataFrame.

Return type

Tuple[nx.Graph, gpd.GeoDataFrame]

src.dynamic_boundary_conditions.river.river_network_to_from_db

This script handles the following tasks: storing both the REC1 river network and its associated data in files along with their metadata in the database, retrieving the existing REC1 river network and its associated data from the database, and managing the addition of REC1 geometries that have been excluded from the river network in the database, as well as retrieving them for an existing REC1 river network.

Module Contents

Functions

get_next_network_id(→ int)	Get the next available REC1 River Network ID from the River Network Exclusions table.
add_network_exclusions_to_db(→ None)	Add REC1 geometries that are excluded from the river network for the current run in the database.
get_new_network_output_paths(→ Tuple[pathlib.Path, ...)	Get new file paths that incorporate the current timestamp into the filenames for storing both the REC1 Network and
get_network_output_metadata(→ Tuple[str, str, str])	Get metadata associated with the REC1 Network.
store_rec1_network_to_db(→ None)	Store both the REC1 river network and its associated data in files, and their metadata in the database.
get_existing_network_metadata_from_db(...)	Retrieve existing REC1 river network metadata for the specified catchment area from the database.
get_existing_network(→ Tuple[networkx.Graph, ...)	Retrieve existing REC1 river network and its associated data.

Attributes

log

src.dynamic_boundary_conditions.river.river_network_to_from_db.**log**

src.dynamic_boundary_conditions.river.river_network_to_from_db.**get_next_network_id**(*engine: sqlalchemy.engine.Engine*) → int

Get the next available REC1 River Network ID from the River Network Exclusions table.

Parameters
engine (*Engine*) – The engine used to connect to the database.

Returns
An identifier for the river network associated with each run, representing the next available River Network ID.

Return type
int

src.dynamic_boundary_conditions.river.river_network_to_from_db.**add_network_exclusions_to_db**(*engine: sqlalchemy.eng rec1_network_ int*, *rec1_network_ geopandas.GeoDataF exclusion_cause: str*) → None

Add REC1 geometries that are excluded from the river network for the current run in the database.

Parameters

- **engine** (*Engine*) – The engine used to connect to the database.
- **rec1_network_id** (*int*) – An identifier for the river network associated with the current run.
- **rec1_network_exclusions** (*gpd.GeoDataFrame*) – A GeoDataFrame containing the REC1 geometries that are excluded from the river network for the current run.
- **exclusion_cause** (*str*) – Cause of exclusion, i.e., the reason why the REC1 river geometry was excluded.

Returns
This function does not return any value.

Return type
None

src.dynamic_boundary_conditions.river.river_network_to_from_db.**get_new_network_output_paths**() → Tuple[pathlib.Pat pathlib.Path]

Get new file paths that incorporate the current timestamp into the filenames for storing both the REC1 Network and its associated data.

Returns
A tuple containing the file path to the REC1 Network and the file path to the REC1 Network data.

Return type
Tuple[pathlib.Path, pathlib.Path]

src.dynamic_boundary_conditions.river.river_network_to_from_db.**get_network_output_metadata**(*network_path: pathlib.Path*, *network_data_path: pathlib.Path*, *catchment_area: geopandas.GeoDataFra* → Tuple[str, str, str]

Get metadata associated with the REC1 Network.

Parameters

- **network_path** (*pathlib.Path*) – The path to the REC1 Network file.
- **network_data_path** (*pathlib.Path*) – The path to the REC1 Network data file.
- **catchment_area** (*gpd.GeoDataFrame*) – A GeoDataFrame representing the catchment area.

Returns

A tuple containing the absolute path to the REC1 Network file as a string, the absolute path to the REC1 Network data file as a string, and the Well-Known Text (WKT) representation of the catchment area's geometry.

Return type

Tuple[str, str, str]

src.dynamic_boundary_conditions.river.river_network_to_from_db.**store_rec1_network_to_db**(*engine: sqlalchemy.engine.E*, *catchment_area: geopandas.GeoDataFrame*, *rec1_network_id: int*, *rec1_network: networkx.Graph*, *rec1_network_data: geopandas.GeoDataFrame*) → None

Store both the REC1 river network and its associated data in files, and their metadata in the database.

Parameters

- **engine** (*Engine*) – The engine used to connect to the database.

- **catchment_area** (*gpd.GeoDataFrame*) – A GeoDataFrame representing the catchment area.
- **rec1_network_id** (*int*) – An identifier for the river network associated with the current run.
- **rec1_network** (*nx.Graph*) – The constructed REC1 river network, represented as a directed graph (DiGraph).
- **rec1_network_data** (*gpd.GeoDataFrame*) – A GeoDataFrame containing the REC1 river network data.

Returns
This function does not return any value.

Return type
None

src.dynamic_boundary_conditions.river.river_network_to_from_db.**get_existing_network_metadata_from_db**(*eng sql cat me geo das* → geo

Retrieve existing REC1 river network metadata for the specified catchment area from the database.

Parameters

- **engine** (*Engine*) – The engine used to connect to the database.
- **catchment_area** (*gpd.GeoDataFrame*) – A GeoDataFrame representing the catchment area.

Returns
A GeoDataFrame containing the existing REC1 river network metadata for the specified catchment area.

Return type
gpd.GeoDataFrame

src.dynamic_boundary_conditions.river.river_network_to_from_db.**get_existing_network**(*engine: sqlalchemy.engine.Engine existing_network_meta: geopandas.GeoDataFrame*) → Tuple[networkx.Graph, geopandas.GeoDataFram

Retrieve existing REC1 river network and its associated data.

Parameters

- **engine** (*Engine*) – The engine used to connect to the database.
- **existing_network_meta** (*gpd.GeoDataFrame*) – A GeoDataFrame containing the metadata for the existing REC1 river network.

Returns
A tuple containing the existing REC1 river network as a directed graph (DiGraph) and its associated data as a GeoDataFrame.

Return type
Tuple[nx.Graph, gpd.GeoDataFrame]

src.dynamic_boundary_conditions.tide

Submodules

src.dynamic_boundary_conditions.tide.main_tide_slr

Main tide and sea level rise script used to fetch tide data, read and store sea level rise data in the database, and generate the requested tide uniform boundary model input for BG-Flood etc.

Module Contents

Functions

remove_existing_boundary_inputs(→ None)	Remove existing uniform boundary input files from the specified directory.
get_or_load_tide_data_for_demo(tide_query_loc, ...)	Retrieve or load tide data for demonstration.
main(→ None)	Fetch tide data, read and store sea level rise data in the database, and generate the requested tide

Attributes

log
sample_polygon

src.dynamic_boundary_conditions.tide.main_tide_slr.**log**

src.dynamic_boundary_conditions.tide.main_tide_slr.**remove_existing_boundary_inputs**(*bg_flood_dir: pathlib.Path*) → None

Remove existing uniform boundary input files from the specified directory.

Parameters
bg_flood_dir (*pathlib.Path*) – BG-Flood model directory containing the uniform boundary input files.

Returns
This function does not return any value.

Return type
None

`src.dynamic_boundary_conditions.tide.main_tide_slr.`**`get_or_load_tide_data_for_demo`**(*tide_query_loc: geopandas.GeoDataFrame*, *tide_length_mins: int*, *time_to_peak_mins: int | float*, *interval_mins: int*)

Retrieve or load tide data for demonstration.

Parameters

- **tide_query_loc** (*gpd.GeoDataFrame*) – A GeoDataFrame containing the locations used to fetch tide data from NIWA using the tide API.
- **tide_length_mins** (*int*) – The length of the tide event in minutes.
- **time_to_peak_mins** (*Union[int, float]*) – The time in minutes when the tide is at its greatest (reaches maximum).
- **interval_mins** (*int*) – The time interval, in minutes, between each recorded tide data point.

Returns
A GeoDataFrame containing the retrieved or loaded tide data for demonstration.

Return type
gpd.GeoDataFrame

`src.dynamic_boundary_conditions.tide.main_tide_slr.`**`main`**(*selected_polygon_gdf: geopandas.GeoDataFrame*, *tide_length_mins: int*, *time_to_peak_mins: int | float*, *interval_mins: int*, *proj_year: int*, *confidence_level: str*, *ssp_scenario: str*, *add_vlm: bool*, *percentile: int*, *log_level: src.digitaltwin.utils.LogLevel = LogLevel.DEBUG*) → None

Fetch tide data, read and store sea level rise data in the database, and generate the requested tide uniform boundary model input for BG-Flood.

Parameters

- **selected_polygon_gdf** (*gpd.GeoDataFrame*) – A GeoDataFrame representing the selected polygon, i.e., the catchment area.
- **tide_length_mins** (*int*) – The length of the tide event in minutes.
- **time_to_peak_mins** (*Union[int, float]*) – The time in minutes when the tide is at its greatest (reaches maximum).
- **interval_mins** (*int*) – The time interval, in minutes, between each recorded tide data point.
- **proj_year** (*int*) – The projection year for which the combined tide and sea level rise data should be generated.

- **confidence_level** (*str*) – The desired confidence level for the sea level rise data. Valid values are 'low' or 'medium'.
- **ssp_scenario** (*str*) – The desired Shared Socioeconomic Pathways (SSP) scenario for the sea level rise data. Valid options for both low and medium confidence are: 'SSP1-2.6', 'SSP2-4.5', or 'SSP5-8.5'. Additional options for medium confidence are: 'SSP1-1.9' or 'SSP3-7.0'.
- **add_vlm** (*bool*) – Indicates whether Vertical Land Motion (VLM) should be included in the sea level rise data. Set to True if VLM should be included, False otherwise.
- **percentile** (*int*) – The desired percentile for the sea level rise data. Valid values are 17, 50, or 83.
- **log_level** (*LogLevel = LogLevel.DEBUG*) – The log level to set for the root logger. Defaults to LogLevel.DEBUG. The available logging levels and their corresponding numeric values are: - LogLevel.CRITICAL (50) - LogLevel.ERROR (40) - LogLevel.WARNING (30) - LogLevel.INFO (20) - LogLevel.DEBUG (10) - LogLevel.NOTSET (0)

Returns
This function does not return any value.

Return type
None

src.dynamic_boundary_conditions.tide.main_tide_slr.**sample_polygon**

src.dynamic_boundary_conditions.tide.sea_level_rise_data

This script handles the reading of sea level rise data from the NZ Sea level rise datasets, storing the data in the database, and retrieving the closest sea level rise data from the database for all locations in the provided tide data.

Module Contents

Functions

get_slr_data_from_nz_searise(→ geopandas.GeoDataFrame)	Read sea level rise data from the NZ Sea level rise datasets and return a GeoDataFrame.
store_slr_data_to_db(→ None)	Store sea level rise data to the database.
get_closest_slr_data(→ geopandas.GeoDataFrame)	Retrieve the closest sea level rise data for a single query location from the database.
get_slr_data_from_db(→ geopandas.GeoDataFrame)	Retrieve the closest sea level rise data from the database for all locations in the provided tide data.

Attributes

log

src.dynamic_boundary_conditions.tide.sea_level_rise_data.**log**

src.dynamic_boundary_conditions.tide.sea_level_rise_data.**get_slr_data_from_nz_searise**() → geopandas.GeoDataFra

Read sea level rise data from the NZ Sea level rise datasets and return a GeoDataFrame.

Returns
A GeoDataFrame containing the sea level rise data from the NZ Sea level rise datasets.

Return type
gpd.GeoDataFrame

Raises
FileNotFoundError – If the sea level rise data directory does not exist or if there are no CSV files in the specified directory.

src.dynamic_boundary_conditions.tide.sea_level_rise_data.**store_slr_data_to_db**(*engine: sqlalchemy.engine.Engine*) → None

Store sea level rise data to the database.

Parameters
engine (*Engine*) – The engine used to connect to the database.

Returns
This function does not return any value.

Return type
None

src.dynamic_boundary_conditions.tide.sea_level_rise_data.**get_closest_slr_data**(*engine: sqlalchemy.engine.Engine*, *single_query_loc: pandas.Series*) → geopandas.GeoDataFrame

Retrieve the closest sea level rise data for a single query location from the database.

Parameters

- **engine** (*Engine*) – The engine used to connect to the database.
- **single_query_loc** (*pd.Series*) – Pandas Series containing the location coordinate and additional information used for retrieval.

Returns
A GeoDataFrame containing the closest sea level rise data for the query location from the database.

Return type
gpd.GeoDataFrame

`src.dynamic_boundary_conditions.tide.sea_level_rise_data.`**`get_slr_data_from_db`**(*engine: sqlalchemy.engine.Engine*, *tide_data: geopandas.GeoDataFrame*) → geopandas.GeoDataFrame

Retrieve the closest sea level rise data from the database for all locations in the provided tide data.

Parameters

- **`engine`** (*`Engine`*) – The engine used to connect to the database.
- **`tide_data`** (*`gpd.GeoDataFrame`*) – A GeoDataFrame containing tide data with added time information (seconds, minutes, hours) and location details.

Returns

A GeoDataFrame containing the closest sea level rise data for all locations in the tide data.

Return type

gpd.GeoDataFrame

`src.dynamic_boundary_conditions.tide.tide_data_from_niwa`

Fetch tide data from NIWA using the Tide API based on the specified approach, datum, etc.

Module Contents

Functions

get_query_loc_coords_position(→ Tuple[float, float, str])	Get the latitude, longitude, and position of a query location.
get_date_ranges(, total_days, days_per_call, int])	Get the start date and duration, measured in days, for each API call used to fetch tide data for the
gen_api_query_param_list(→ List[Dict[str, Union[str, ...)	Generate a list of API query parameters used to retrieve tide data for the requested period.
fetch_tide_data(→ geopandas.GeoDataFrame)	Fetch tide data using the provided query parameters within a single API call.
fetch_tide_data_for_requested_period(...)	Iterate over the list of API query parameters to fetch tide data for the requested period.
convert_to_nz_timezone(→ geopandas.GeoDataFrame)	Convert the time column in the initially retrieved tide data for the requested period from UTC to NZ timezone.
fetch_tide_data_from_niwa(, total_days, interval_mins)	Retrieve tide data from NIWA for the requested time period using the Tide API.
get_highest_tide_datetime(→ pandas.Timestamp)	Get the datetime of the most recent highest tide that occurred within the requested time period.
get_highest_tide_datetime_span(...)	Get the start and end datetimes of a tide event centered around the datetime of the highest tide.
get_highest_tide_date_span(→ Tuple[datetime.date, int])	Get the start date and duration in days of a tide event centered around the datetime of the highest tide.
fetch_tide_data_around_highest_tide(...)	Fetch tide data around the highest tide from NIWA for the specified tide length and interval.
get_time_mins_to_add(→ List[Union[float, int]])	Get the time values in minutes to add to the tide data.
add_time_information(→ geopandas.GeoDataFrame)	Add time information (seconds, minutes, hours) to the tide data.
get_tide_data(, total_days, tide_length_mins, ...)	Fetch tide data from NIWA using the Tide API based on the specified approach, datum, and other parameters.

Attributes

TIDE_API_URL_DATA
TIDE_API_URL_DATA_CSV

src.dynamic_boundary_conditions.tide.tide_data_from_niwa.**TIDE_API_URL_DATA**

src.dynamic_boundary_conditions.tide.tide_data_from_niwa.**TIDE_API_URL_DATA_CSV**

src.dynamic_boundary_conditions.tide.tide_data_from_niwa.**get_query_loc_coords_position**(*query_loc_row: geopandas.GeoDataFrame*) → Tuple[float, float, str]

Get the latitude, longitude, and position of a query location.

Parameters
`query_loc_row` (*gpd.GeoDataFrame*) – A GeoDataFrame representing a query location used to fetch tide data from NIWA using the tide API.

Returns
A tuple containing the latitude, longitude, and position of the query location.

Return type
Tuple[float, float, str]

`src.dynamic_boundary_conditions.tide.tide_data_from_niwa.`**`get_date_ranges`**(*start_date: datetime.date = date.today()*, *total_days: int = 365*, *days_per_call: int = 31*) → Dict[datetime.date, int]

Get the start date and duration, measured in days, for each API call used to fetch tide data for the requested period.

Parameters

- **`start_date`** (*date = date.today()*) – The start date for retrieving tide data. It can be in the past or present. Default is today's date.
- **`total_days`** (*int = 365*) – The total number of days of tide data to retrieve. Default is 365 days (one year).
- **`days_per_call`** (*int = 31*) – The number of days to fetch in each API call. Must be between 1 and 31 inclusive. Default is 31, which represents the maximum number of days that can be fetched per API call.

Returns
A dictionary containing the start date as the key and the duration, in days, for each API call as the value.

Return type
Dict[date, int]

Raises
`ValueError` –

- If 'total_days' is less than 1.
- If 'days_per_call' is not between 1 and 31 inclusive.

src.dynamic_boundary_conditions.tide.tide_data_from_niwa.**gen_api_query_param_list**(*lat: int | float*, *long: int | float*, *date_ranges: Dict[datetime.date, int]*, *interval_mins: int | None = None*, *datum: src.dynamic_boundary_con… = DatumType.LAT*) → List[Dict[str, str | int]]

Generate a list of API query parameters used to retrieve tide data for the requested period.

Parameters

- **lat** (*Union[int, float]*) – Latitude in the range of -29 to -53 (e.g., -30.876).
- **long** (*Union[int, float]*) – Longitude in the range of 160 to 180 and -175 to -180 (e.g., -175.543).
- **date_ranges** (*Dict[date, int]*) – Dictionary of start date and number of days for each API call needed to retrieve tide data for the requested period.
- **interval_mins** (*Optional[int] = None*) – Output time interval in minutes, range from 10 to 1440 minutes (1 day). Omit to retrieve only the highest and lowest tide data.
- **datum** (*DatumType = DatumType.LAT*) – Datum used for fetching tide data from NIWA. Default value is LAT. Valid options are LAT for the Lowest Astronomical Tide and MSL for the Mean Sea Level.

Returns

A list of API query parameters used to retrieve tide data for the requested period.

Return type

List[Dict[str, Union[str, int]]]

Raises

ValueError –

- If the latitude is outside the range of -29 to -53.
- If the longitude is outside the range of 160 to 180 or -175 to -180.
- If the time interval is provided and outside the range of 10 to 1440.

async src.dynamic_boundary_conditions.tide.tide_data_from_niwa.**fetch_tide_data**(*session: aiohttp.ClientSession*, *query_param: Dict[str, str | int]*, *url: str = TIDE_API_URL_DATA*) → geopandas.GeoDataFrame

Fetch tide data using the provided query parameters within a single API call.

Parameters

- **session** (*aiohttp.ClientSession*) – An instance of *aiohttp.ClientSession* used for making HTTP requests.
- **query_param** (*Dict[str, Union[str, int]]*) – The query parameters used to retrieve tide data for a specific location and time period.
- **url** (*str = TIDE_API_URL_DATA*) – Tide API HTTP request URL. Defaults to 'https://api.niwa.co.nz/tides/data'. Can be either 'https://api.niwa.co.nz/tides/data' or 'https://api.niwa.co.nz/tides/data.csv'.

Returns
A GeoDataFrame containing the fetched tide data.

Return type
gpd.GeoDataFrame

async src.dynamic_boundary_conditions.tide.tide_data_from_niwa.**fetch_tide_data_for_requested_period**(*quer List[str | int]] url: str = TID* → geop

Iterate over the list of API query parameters to fetch tide data for the requested period.

Parameters

- **query_param_list** (*List[Dict[str, Union[str, int]]]*) – A list of API query parameters used to retrieve tide data for the requested period.
- **url** (*str = TIDE_API_URL_DATA*) – Tide API HTTP request URL. Defaults to 'https://api.niwa.co.nz/tides/data'. Can be either 'https://api.niwa.co.nz/tides/data' or 'https://api.niwa.co.nz/tides/data.csv'.

Returns
A GeoDataFrame containing the fetched tide data for the requested period.

Return type
gpd.GeoDataFrame

Raises

- **ValueError** – If an invalid URL is specified for the Tide API HTTP request.
- **RuntimeError** – If failed to fetch tide data.

src.dynamic_boundary_conditions.tide.tide_data_from_niwa.**convert_to_nz_timezone**(*tide_data_utc: geopandas.GeoDataFrame*) → geopandas.GeoDataFrame

Convert the time column in the initially retrieved tide data for the requested period from UTC to NZ timezone.

Parameters

tide_data_utc (*gpd.GeoDataFrame*) – The original tide data obtained for the requested period with the time column expressed in UTC.

Returns

The tide data with the time column converted to NZ timezone.

Return type

gpd.GeoDataFrame

src.dynamic_boundary_conditions.tide.tide_data_from_niwa.**fetch_tide_data_from_niwa**(*tide_query_loc: geopandas.GeoDataFrame*, *datum: src.dynamic_boundary_co = DatumType.LAT*, *start_date: datetime.date = date.today()*, *total_days: int = 365*, *interval_mins: int | None = None*) → geopandas.GeoDataFrame

Retrieve tide data from NIWA for the requested time period using the Tide API.

Parameters

- **tide_query_loc** (*gpd.GeoDataFrame*) – A GeoDataFrame containing the query coordinates and their positions.
- **datum** (*DatumType = DatumType.LAT*) – Datum used for fetching tide data from NIWA. Default value is LAT. Valid options are LAT for the Lowest Astronomical Tide and MSL for the Mean Sea Level.
- **start_date** (*date = date.today()*) – The start date for retrieving tide data. It can be in the past or present. Default is today's date.
- **total_days** (*int = 365*) – The total number of days of tide data to retrieve. Default is 365 days (one year).
- **interval_mins** (*Optional[int] = None*) – Output time interval in minutes, range from 10 to 1440 minutes (1 day). Omit to retrieve only the highest and lowest tide data.

Returns

A GeoDataFrame containing the fetched tide data from NIWA for the requested time period.

Return type

gpd.GeoDataFrame

src.dynamic_boundary_conditions.tide.tide_data_from_niwa.**get_highest_tide_datetime**(*tide_data: geopandas.GeoDataFrame*) → pandas.Timestamp

Get the datetime of the most recent highest tide that occurred within the requested time period.

Parameters
tide_data (*gpd.GeoDataFrame*) – The tide data fetched from NIWA for the requested time period. The time column is expressed in NZ timezone, which was converted from UTC.

Returns
The datetime of the most recent highest tide that occurred within the requested time period.

Return type
pd.Timestamp

src.dynamic_boundary_conditions.tide.tide_data_from_niwa.**get_highest_tide_datetime_span**(*highest_tide_datetim pandas.Timestamp, tide_length_mins: int*) → Tuple[pandas.Timestar pandas.Timestamp]

Get the start and end datetimes of a tide event centered around the datetime of the highest tide.

Parameters

- **highest_tide_datetime** (*pd.Timestamp*) – The datetime of the most recent highest tide that occurred within the requested time period.
- **tide_length_mins** (*int*) – The length of the tide event in minutes.

Returns
A tuple containing the start and end datetimes of the tide event centered around the datetime of the highest tide.

Return type
Tuple[pd.Timestamp, pd.Timestamp]

src.dynamic_boundary_conditions.tide.tide_data_from_niwa.**get_highest_tide_date_span**(*start_datetime: pandas.Timestamp, end_datetime: pandas.Timestamp*) → Tuple[datetime.date, int]

Get the start date and duration in days of a tide event centered around the datetime of the highest tide.

Parameters

- **start_datetime** (*pd.Timestamp*) – The start datetime of the tide event centered around the datetime of the highest tide.
- **end_datetime** (*pd.Timestamp*) – The end datetime of the tide event centered around the datetime of the highest tide.

Returns

A tuple containing the start date and the duration in days of a tide event centered around the datetime of the highest tide.

Return type

Tuple[date, int]

`src.dynamic_boundary_conditions.tide.tide_data_from_niwa.`**`fetch_tide_data_around_highest_tide`**`(`*tide_data: geopandas.GeoData*, *tide_length_m* *int*, *interval_mins: int = 10*, *datum: src.dynamic_* *= DatumType.LAT*`)` → geopandas.Ge

Fetch tide data around the highest tide from NIWA for the specified tide length and interval.

Parameters

- **`tide_data`** (*`gpd.GeoDataFrame`*) – The tide data fetched from NIWA for the requested time period. The time column is expressed in NZ timezone, which was converted from UTC.
- **`tide_length_mins`** (*`int`*) – The length of the tide event in minutes.
- **`interval_mins`** (*`int = 10`*) – The time interval, in minutes, between each recorded tide data point. The default value is 10 minutes.
- **`datum`** (*`DatumType = DatumType.LAT`*) – Datum used for fetching tide data from NIWA. Default value is LAT. Valid options are LAT for the Lowest Astronomical Tide and MSL for the Mean Sea Level.

Returns

The tide data around the highest tide, fetched from NIWA, for the specified tide length and interval.

Return type

gpd.GeoDataFrame

`src.dynamic_boundary_conditions.tide.tide_data_from_niwa.`**`get_time_mins_to_add`**(*tide_data: geopandas.GeoDataFrame*, *tide_length_mins: int*, *time_to_peak_mins: int | float*, *interval_mins: int = 10*) → List[float | int]

Get the time values in minutes to add to the tide data.

Parameters

- **tide_data** (*gpd.GeoDataFrame*) – The tide data for which time values in minutes will be calculated.
- **tide_length_mins** (*int*) – The length of the tide event in minutes.
- **time_to_peak_mins** (*Union[int, float]*) – The time in minutes when the tide is at its greatest (reaches maximum).
- **interval_mins** (*int = 10*) – The time interval, in minutes, between each recorded tide data point. The default value is 10 minutes.

Returns
A list containing the time values in minutes to add to the tide data.

Return type
List[Union[float, int]]

`src.dynamic_boundary_conditions.tide.tide_data_from_niwa.`**`add_time_information`**(*tide_data: geopandas.GeoDataFrame*, *time_to_peak_mins: int | float*, *interval_mins: int = 10*, *tide_length_mins: int | None = None*, *total_days: int | None = None*, *approach: src.dynamic_boundary_condition = ApproachType.KING_TIDE*) → geopandas.GeoDataFrame

Add time information (seconds, minutes, hours) to the tide data.

Parameters

- **tide_data** (*gpd.GeoDataFrame*) – The tide data for which time information will be added.
- **time_to_peak_mins** (*Union[int, float]*) – The time in minutes when the tide is at its greatest (reaches maximum).

- **interval_mins** (*int = 10*) – The time interval, in minutes, between each recorded tide data point. The default value is 10 minutes.
- **tide_length_mins** (*Optional[int] = None*) – The length of the tide event in minutes. Only required if the 'approach' is KING_TIDE.
- **total_days** (*Optional[int] = None*) – The total number of days for the tide event. Only required if the 'approach' is PERIOD_TIDE.
- **approach** (*ApproachType = ApproachType.KING_TIDE*) – The approach used to get the tide data. Default is KING_TIDE.

Returns

The tide data with added time information in seconds, minutes, and hours.

Return type

gpd.GeoDataFrame

Raises

ValueError – If 'time_to_peak_mins' is less than the minimum time to peak.

Notes

The minimum time to peak is calculated differently depending on the approach used: - For the KING_TIDE approach, it is half of the 'tide_length_mins'. - For the PERIOD_TIDE approach, it is half of the 'total_days' converted to minutes.

src.dynamic_boundary_conditions.tide.tide_data_from_niwa.**get_tide_data**(*tide_query_loc: geopandas.GeoDataFrame*, *time_to_peak_mins: int | float*, *approach: src.dynamic_boundary_conditions.tide.tid = ApproachType.KING_TIDE*, *start_date: datetime.date = date.today()*, *total_days: int | None = None*, *tide_length_mins: int | None = None*, *interval_mins: int = 10*, *datum: src.dynamic_boundary_conditions.tide.tid = DatumType.LAT*) → geopandas.GeoDataFrame

Fetch tide data from NIWA using the Tide API based on the specified approach, datum, and other parameters.

Parameters

- **tide_query_loc** (*gpd.GeoDataFrame*) – A GeoDataFrame containing the query coordinates and their positions.
- **time_to_peak_mins** (*Union[int, float]*) – The time in minutes when the tide is at its greatest (reaches maximum).

- **approach** (*ApproachType = ApproachType.KING_TIDE*) – The approach used to get the tide data. Default is KING_TIDE.
- **start_date** (*date = date.today()*) – The start date for retrieving tide data. It can be in the past or present. Default is today's date.
- **total_days** (*Optional[int] = None*) – The total number of days for the tide event. Only required if the 'approach' is PERIOD_TIDE.
- **tide_length_mins** (*Optional[int] = None*) – The length of the tide event in minutes. Only required if the 'approach' is KING_TIDE.
- **interval_mins** (*int = 10*) – The time interval, in minutes, between each recorded tide data point. The default value is 10 minutes.
- **datum** (*DatumType = DatumType.LAT*) – Datum used for fetching tide data from NIWA. Default value is LAT. Valid options are LAT for the Lowest Astronomical Tide and MSL for the Mean Sea Level.

Returns
The tide data with added time information in seconds, minutes, and hours.

Return type
gpd.GeoDataFrame

Raises
ValueError –

- If 'interval_mins' is None.
- If the 'approach' is KING_TIDE and 'tide_length_mins' is None or 'total_days' is not None.
- If the 'approach' is PERIOD_TIDE and 'total_days' is None or 'tide_length_mins' is not None.

src.dynamic_boundary_conditions.tide.tide_enum

Enum(s) used in the tide_slr module.

Module Contents

Classes

DatumType	Enum class representing different datum types.
ApproachType	Enum class representing different types of approaches.

class src.dynamic_boundary_conditions.tide.tide_enum.**DatumType**

Bases: enum.StrEnum

Enum class representing different datum types.

LAT

Lowest astronomical tide.

Type
str

MSL

Mean sea level.

Type
str

LAT = 'lat'

MSL = 'msl'

class src.dynamic_boundary_conditions.tide.tide_enum.**ApproachType**

Bases: enum.StrEnum

Enum class representing different types of approaches.

KING_TIDE

King Tide approach.

Type
str

PERIOD_TIDE

Period Tide approach.

Type
str

KING_TIDE = 'king_tide'

PERIOD_TIDE = 'period_tide'

src.dynamic_boundary_conditions.tide.tide_query_location

Get the locations used to fetch tide data from NIWA using the tide API. sli229

Module Contents

Functions

get_regional_council_clipped_from_db(...)	Retrieve regional council clipped data from the database based on the catchment area.
get_nz_coastline_from_db(→ geopandas.GeoDataFrame)	Retrieve the New Zealand coastline data within a specified distance of the catchment area from the database.
get_catchment_boundary_info(→ geopandas.GeoDataFrame)	Get information about the boundary segments of the catchment area.
get_catchment_boundary_lines(→ geopandas.GeoDataFrame)	Get the boundary lines of the catchment area.
get_catchment_boundary_centroids(→ geopandas.GeoDataFrame)	Get the centroids of the boundary lines of the catchment area.
get_non_intersection_centroid_position(...)	Determine the positions of non-intersection centroid points relative to the boundary lines of the catchment area.
get_tide_query_locations(→ geopandas.GeoDataFrame)	Get the locations used to fetch tide data from NIWA using the tide API.

exception src.dynamic_boundary_conditions.tide.tide_query_location.**NoTideDataException**

Bases: Exception

Exception raised when no tide data is to be used for the BG-Flood model.

src.dynamic_boundary_conditions.tide.tide_query_location.**get_regional_council_clipped_from_db**(*engine: sqlalchemy.e catch- ment_area: geopan- das.GeoDat* → geopandas.G

Retrieve regional council clipped data from the database based on the catchment area.

Parameters

- **engine** (*Engine*) – The engine used to connect to the database.
- **catchment_area** (*gpd.GeoDataFrame*) – A GeoDataFrame representing the catchment area.

Returns

A GeoDataFrame containing the regional council clipped data for the catchment area.

Return type

gpd.GeoDataFrame

src.dynamic_boundary_conditions.tide.tide_query_location.**get_nz_coastline_from_db**(*engine: sqlalchemy.engine.Engine*, *catch- ment_area: geopan- das.GeoDataFrame*, *dis- tance_km: int = 1*) → geopandas.GeoDataFrame

Retrieve the New Zealand coastline data within a specified distance of the catchment area from the database.

Parameters

- **engine** (*Engine*) – The engine used to connect to the database.
- **catchment_area** (*gpd.GeoDataFrame*) – A GeoDataFrame representing the catchment area.
- **distance_km** (*int = 1*) – Distance in kilometers used to buffer the catchment area for coastline retrieval. Default is 1 kilometer.

Returns

A GeoDataFrame containing the New Zealand coastline data within the specified distance of the catchment area.

Return type

gpd.GeoDataFrame

src.dynamic_boundary_conditions.tide.tide_query_location.**get_catchment_boundary_info**(*catchment_area: geopandas.GeoDataFrame*) → geopandas.GeoDataFrame

Get information about the boundary segments of the catchment area.

Parameters
`catchment_area` (*gpd.GeoDataFrame*) – A GeoDataFrame representing the catchment area.

Returns
A GeoDataFrame containing information about the boundary segments of the catchment area.

Return type
gpd.GeoDataFrame

Raises
`ValueError` – If the position of a catchment boundary line cannot be identified.

src.dynamic_boundary_conditions.tide.tide_query_location.**get_catchment_boundary_lines**(*catchment_area: geopandas.GeoDataFrame*) → geopandas.GeoDataFrame

Get the boundary lines of the catchment area.

Parameters
`catchment_area` (*gpd.GeoDataFrame*) – A GeoDataFrame representing the catchment area.

Returns
A GeoDataFrame containing the boundary lines of the catchment area.

Return type
gpd.GeoDataFrame

src.dynamic_boundary_conditions.tide.tide_query_location.**get_catchment_boundary_centroids**(*catchment_area: geopandas.GeoDataFrame*) → geopandas.GeoDataFrame

Get the centroids of the boundary lines of the catchment area.

Parameters
`catchment_area` (*gpd.GeoDataFrame*) – A GeoDataFrame representing the catchment area.

Returns
A GeoDataFrame containing the centroids of the boundary lines of the catchment area.

Return type
gpd.GeoDataFrame

src.dynamic_boundary_conditions.tide.tide_query_location.**get_non_intersection_centroid_position**(*catchmen geopandas.GeoD non_inter geopandas.GeoD*) → geopanda

Determine the positions of non-intersection centroid points relative to the boundary lines of the catchment area.

Parameters

- **catchment_area** (*gpd.GeoDataFrame*) – A GeoDataFrame representing the catchment area.
- **non_intersection_area** (*gpd.GeoDataFrame*) – A GeoDataFrame representing the non-intersection area.

Returns

A GeoDataFrame containing the positions of non-intersection centroid points relative to the catchment boundary lines. The GeoDataFrame includes the 'position' column denoting the relative position and the 'geometry' column representing the centroid points of the non-intersection areas.

Return type

gpd.GeoDataFrame

`src.dynamic_boundary_conditions.tide.tide_query_location.`**`get_tide_query_locations`**(*engine: sqlalchemy.engine.Engine*, *catchment_area: geopandas.GeoDataFrame*, *distance_km: int = 1*) → geopandas.GeoDataFrame

Get the locations used to fetch tide data from NIWA using the tide API.

Parameters

- **engine** (*Engine*) – The engine used to connect to the database.
- **catchment_area** (*gpd.GeoDataFrame*) – A GeoDataFrame representing the catchment area.
- **distance_km** (*int = 1*) – Distance in kilometers used to buffer the catchment area for coastline retrieval. Default is 1 kilometer.

Returns

A GeoDataFrame containing the locations used to fetch tide data from NIWA using the tide API.

Return type

gpd.GeoDataFrame

Raises

NoTideDataException – If no coastline is found within the specified distance of the catchment area.

src.dynamic_boundary_conditions.tide.tide_slr_combine

Generates combined tide and sea level rise (SLR) data for a specific projection year, taking into account the provided confidence level, SSP scenario, inclusion of Vertical Land Motion (VLM), percentile, and more.

Module Contents

Functions

split_slr_measurementname_column(→ geopandas.GeoDataFrame)	Split the 'measurementname' column in the sea level rise data to extract and add additional information.
get_slr_scenario_data(→ geopandas.GeoDataFrame)	Get sea level rise scenario data based on the specified confidence_level, ssp_scenario, add_vlm, and percentile.
get_interpolated_slr_scenario_data(...)	Interpolates sea level rise scenario data based on the specified year interval and interpolation method.
add_slr_to_tide(→ pandas.DataFrame)	Adds sea level rise (SLR) data to the tide data for a specific projection year and
get_combined_tide_slr_data(→ pandas.DataFrame)	Generates the combined tide and sea level rise (SLR) data for a specific projection year, considering the given

src.dynamic_boundary_conditions.tide.tide_slr_combine.**split_slr_measurementname_column**(*slr_data: geopandas.GeoDataFrame*) → geopandas.GeoDataF

Split the 'measurementname' column in the sea level rise data to extract and add additional information.

Parameters
slr_data (*gpd.GeoDataFrame*) – A GeoDataFrame containing the sea level rise data.

Returns
A GeoDataFrame containing the sea level rise data with additional columns for extracted information: 'confidence_level', 'ssp_scenario', and 'add_vlm'.

Return type
gpd.GeoDataFrame

src.dynamic_boundary_conditions.tide.tide_slr_combine.**get_slr_scenario_data**(*slr_data: geopandas.GeoDataFrame*, *confidence_level: str*, *ssp_scenario: str*, *add_vlm: bool*, *percentile: int*) → geopandas.GeoDataFrame

Get sea level rise scenario data based on the specified confidence_level, ssp_scenario, add_vlm, and percentile.

Parameters

- **slr_data** (*gpd.GeoDataFrame*) – A GeoDataFrame containing the sea level rise data.

- **confidence_level** (*str*) – The desired confidence level for the scenario data. Valid values are 'low' or 'medium'.
- **ssp_scenario** (*str*) – The desired Shared Socioeconomic Pathways (SSP) scenario for the scenario data. Valid options for both low and medium confidence are: 'SSP1-2.6', 'SSP2-4.5', or 'SSP5-8.5'. Additional options for medium confidence are: 'SSP1-1.9' or 'SSP3-7.0'.
- **add_vlm** (*bool*) – Indicates whether to include Vertical Land Motion (VLM) in the scenario data. Set to True if VLM should be included, False otherwise.
- **percentile** (*int*) – The desired percentile for the scenario data. Valid values are 17, 50, or 83.

Returns

A GeoDataFrame containing the sea level rise scenario data based on the specified confidence_level, ssp_scenario, add_vlm, and percentile.

Return type

gpd.GeoDataFrame

Raises

`ValueError` –

- If an invalid 'confidence_level' value is provided.
- If an invalid 'ssp_scenario' value is provided.
- If an invalid 'add_vlm' value is provided.
- If an invalid 'percentile' value is provided.

`src.dynamic_boundary_conditions.tide.tide_slr_combine.`**`get_interpolated_slr_scenario_data`**(*slr_scenario_data: geopandas.GeoDataFram increment_year: int = 1, interp_method: str = 'linear'*) → geopandas.GeoDat

Interpolates sea level rise scenario data based on the specified year interval and interpolation method.

Parameters

- **slr_scenario_data** (*gpd.GeoDataFrame*) – A GeoDataFrame containing the sea level rise scenario data.
- **increment_year** (*int = 1*) – The year interval used for interpolation. Defaults to 1 year.
- **interp_method** (*str = "linear"*) – Temporal interpolation method to be used. Defaults to 'linear'. Available methods: 'linear', 'nearest', 'nearest-up', 'zero', 'slinear', 'quadratic', 'cubic', 'previous', 'next'. Refer to 'scipy.interpolate.interp1d()' for more details.

Returns
A GeoDataFrame containing the interpolated sea level rise scenario data.

Return type
gpd.GeoDataFrame

Raises
`ValueError` –

- If the specified 'increment_year' is out of range.
- If the specified 'interp_method' is not supported.

`src.dynamic_boundary_conditions.tide.tide_slr_combine.`**`add_slr_to_tide`**(*tide_data: geopandas.GeoDataFrame*, *slr_interp_scenario: geopandas.GeoDataFrame*, *proj_year: int*) → pandas.DataFrame

Adds sea level rise (SLR) data to the tide data for a specific projection year and returns the combined tide and sea level rise value.

Parameters

- **`tide_data`** (*`gpd.GeoDataFrame`*) – A GeoDataFrame containing tide data with added time information (seconds, minutes, hours) and location details.
- **`slr_interp_scenario`** (*`gpd.GeoDataFrame`*) – A GeoDataFrame containing the interpolated sea level rise scenario data.
- **`proj_year`** (*`int`*) – The projection year for which sea level rise data should be added to the tide data.

Returns
A DataFrame that contains the combined tide and sea level rise data for the specified projection year.

Return type
pd.DataFrame

`src.dynamic_boundary_conditions.tide.tide_slr_combine.`**`get_combined_tide_slr_data`**(*tide_data: geopandas.GeoDataFrame*, *slr_data: geopandas.GeoDataFrame*, *proj_year: int*, *confidence_level: str*, *ssp_scenario: str*, *add_vlm: bool*, *percentile: int*, *increment_year: int = 1*, *interp_method: str = 'linear'*) → pandas.DataFrame

Generates the combined tide and sea level rise (SLR) data for a specific projection year, considering the given confidence_level, ssp_scenario, add_vlm, percentile, and more.

Parameters

- **tide_data** (*gpd.GeoDataFrame*) – A GeoDataFrame containing tide data with added time information (seconds, minutes, hours) and location details.
- **slr_data** (*gpd.GeoDataFrame*) – A GeoDataFrame containing the sea level rise data.
- **proj_year** (*int*) – The projection year for which the combined tide and sea level rise data should be generated.
- **confidence_level** (*str*) – The desired confidence level for the sea level rise data.
- **ssp_scenario** (*str*) – The desired Shared Socioeconomic Pathways (SSP) scenario for the sea level rise data.
- **add_vlm** (*bool*) – Indicates whether Vertical Land Motion (VLM) should be included in the sea level rise data.
- **percentile** (*int*) – The desired percentile for the sea level rise data.
- **increment_year** (*int = 1*) – The year interval used for interpolating the sea level rise data. Defaults to 1 year.
- **interp_method** (*str = "linear"*) – Temporal interpolation method used for interpolating the sea level rise data. Defaults to 'linear'. Available methods: 'linear', 'nearest', 'nearest-up', 'zero', 'slinear', 'quadratic', 'cubic', 'previous', 'next'. Refer to 'scipy.interpolate.interp1d()' for more details.

Returns

A DataFrame containing the combined tide and sea level rise data for the specified projection year, taking into account the provided confidence_level, ssp_scenario, add_vlm, percentile, and more.

Return type
pd.DataFrame

src.dynamic_boundary_conditions.tide.tide_slr_model_input

Generates the requested water level uniform boundary model input for BG-Flood.

Module Contents

Functions

generate_uniform_boundary_input(→ None)	Generates the requested water level uniform boundary model input for BG-Flood.

Attributes

log

src.dynamic_boundary_conditions.tide.tide_slr_model_input.**log**

src.dynamic_boundary_conditions.tide.tide_slr_model_input.**generate_uniform_boundary_input**(*bg_flood_dir: pathlib.Path*, *tide_slr_data: pandas.DataFrame*) → None

Generates the requested water level uniform boundary model input for BG-Flood.

Parameters

- **bg_flood_dir** (*pathlib.Path*) – The BG-Flood model directory.
- **tide_slr_data** (*pd.DataFrame*) – A DataFrame containing the combined tide and sea level rise data.

Returns
This function does not return any value.

Return type
None

1.1.1.3 src.flood_model

Submodules

src.flood_model.bg_flood_model

This script handles the processing of input files for the BG-Flood Model, executes the flood model, stores the resulting model output metadata in the database, and incorporates the model output into GeoServer for visualization.

Module Contents

Functions

get_valid_bg_flood_dir(→ pathlib.Path)	Get the valid BG-Flood Model directory.
get_new_model_output_path(→ pathlib.Path)	Get a new file path for saving the BG Flood model output with the current timestamp included in the filename.
get_model_output_metadata(→ Tuple[str, str, str])	Get metadata related to the BG Flood model output.
store_model_output_metadata_to_db(→ int)	Store metadata related to the BG Flood model output in the database.
model_output_from_db_by_id(→ pathlib.Path)	
add_crs_to_latest_model_output(→ None)	Add Coordinate Reference System (CRS) to the latest BG-Flood model output.
process_rain_input_files(→ None)	Process rain input files and write their parameter values to the BG-Flood parameter file.
process_boundary_input_files(→ None)	Process uniform boundary input files and write their parameter values to the BG-Flood parameter file.
process_river_input_files(→ None)	Process river input files, rename them, and write their parameter values to the BG-Flood parameter file.
prepare_bg_flood_model_inputs(→ None)	Prepare inputs for the BG-Flood Model.
run_bg_flood_model(→ None)	Run the BG-Flood Model for the specified catchment area.
main(→ int)	Generate BG-Flood model output for the requested catchment area, and incorporate the model output to GeoServer

Attributes

log
Base
sample_polygon

src.flood_model.bg_flood_model.**log**

src.flood_model.bg_flood_model.**Base**

src.flood_model.bg_flood_model.**get_valid_bg_flood_dir**() → pathlib.Path

Get the valid BG-Flood Model directory.

Returns
The valid BG-Flood Model directory.

Return type
pathlib.Path

Raises
FileNotFoundError – If the BG-Flood Model directory is not found or is not a valid directory.

src.flood_model.bg_flood_model.**get_new_model_output_path**() → pathlib.Path

Get a new file path for saving the BG Flood model output with the current timestamp included in the filename.

Returns
The path to the BG Flood model output file.

Return type
pathlib.Path

src.flood_model.bg_flood_model.**get_model_output_metadata**(*model_output_path: pathlib.Path*, *catchment_area: geopandas.GeoDataFrame*) → Tuple[str, str, str]

Get metadata related to the BG Flood model output.

Parameters

- **model_output_path** (*pathlib.Path*) – The path to the BG Flood model output file.
- **catchment_area** (*gpd.GeoDataFrame*) – A GeoDataFrame representing the catchment area.

Returns
A tuple containing three elements: the name of the BG Flood model output file, its absolute path as a string, and the Well-Known Text (WKT) representation of the catchment area's geometry.

Return type
Tuple[str, str, str]

src.flood_model.bg_flood_model.**store_model_output_metadata_to_db**(*engine: sqlalchemy.engine.Engine*, *model_output_path: pathlib.Path*, *catchment_area: geopandas.GeoDataFrame*) → int

Store metadata related to the BG Flood model output in the database.

Parameters

- **engine** (*Engine*) – The engine used to connect to the database.
- **model_output_path** (*pathlib.Path*) – The path to the BG Flood model output file.
- **catchment_area** (*gpd.GeoDataFrame*) – A GeoDataFrame representing the catchment area.

Returns
Returns the model id of the new flood_model produced

Return type
int

`src.flood_model.bg_flood_model.`**`model_output_from_db_by_id`**(*model_id: int*) → pathlib.Path

`src.flood_model.bg_flood_model.`**`add_crs_to_latest_model_output`**(*flood_model_output_id: int*) → None

Add Coordinate Reference System (CRS) to the latest BG-Flood model output.

Returns
This function does not return any value.

Return type
None

`src.flood_model.bg_flood_model.`**`process_rain_input_files`**(*bg_flood_dir: pathlib.Path*, *param_file: TextIO*) → None

Process rain input files and write their parameter values to the BG-Flood parameter file.

Parameters

- **bg_flood_dir** (*pathlib.Path*) – The BG-Flood model directory containing the rain input files.
- **param_file** (*TextIO*) – The file object representing the parameter file where the parameter values will be written.

Returns
This function does not return any value.

Return type
None

`src.flood_model.bg_flood_model.`**`process_boundary_input_files`**(*bg_flood_dir: pathlib.Path*, *param_file: TextIO*) → None

Process uniform boundary input files and write their parameter values to the BG-Flood parameter file.

Parameters

- **bg_flood_dir** (*pathlib.Path*) – The BG-Flood model directory containing the uniform boundary input files.
- **param_file** (*TextIO*) – The file object representing the parameter file where the parameter values will be written.

Returns
This function does not return any value.

Return type
None

`src.flood_model.bg_flood_model.`**`process_river_input_files`**(*bg_flood_dir: pathlib.Path*, *param_file: TextIO*) → None

Process river input files, rename them, and write their parameter values to the BG-Flood parameter file.

Parameters

- **bg_flood_dir** (*pathlib.Path*) – The BG-Flood model directory containing the river input files.
- **param_file** (*TextIO*) – The file object representing the parameter file where the parameter values will be written.

Returns
This function does not return any value.

Return type
None

`src.flood_model.bg_flood_model.`**`prepare_bg_flood_model_inputs`**(*bg_flood_dir: pathlib.Path*, *model_output_path: pathlib.Path*, *hydro_dem_path: pathlib.Path*, *resolution: int | float*, *output_timestep: int | float = 0*, *end_time: int | float = 0*, *mask: int | float = 9999*, *gpu_device: int = 0*, *small_nc: int = 0*) → None

Prepare inputs for the BG-Flood Model.

Parameters

- **bg_flood_dir** (*pathlib.Path*) – The BG-Flood Model directory.
- **model_output_path** (*pathlib.Path*) – The new file path for saving the BG Flood model output with the current timestamp included in the filename.
- **hydro_dem_path** (*pathlib.Path,*) – The file path of the Hydrologically conditioned DEM (Hydro DEM) for the specified catchment area.
- **resolution** (*Union[int, float]*) – The grid resolution in meters for metric grids, representing the size of each grid cell.
- **output_timestep** (*Union[int, float] = 0*) – Time step between model outputs in seconds. Default value is 0.0 (no output generated).
- **end_time** (*Union[int, float] = 0*) – Time in seconds when the model stops. Default value is 0.0 (model initializes but does not run).
- **mask** (*Union[int, float] = 9999*) – The mask value is used to remove blocks from computation where the topography elevation (zb) is greater than the specified value. Default value is 9999.0 (no areas are masked).
- **gpu_device** (*int = 0*) – Specify the GPU device to be used. Default value is 0 (the first available GPU). Set the value to -1 to use the CPU. For other GPUs, use values 2 and above.
- **small_nc** (*int = 0*) – Specify whether the output should be saved as short integers to reduce the size of the output file. Set the value to 1 to enable short integer conversion, or set it to 0 to save all variables as floats. Default value is 0.

Returns
This function does not return any value.

Return type
None

`src.flood_model.bg_flood_model.`**`run_bg_flood_model`**(*engine: sqlalchemy.engine.Engine*, *catchment_area: geopandas.GeoDataFrame*, *model_output_path: pathlib.Path*, *output_timestep: int | float = 0*, *end_time: int | float = 0*, *resolution: int | float | None = None*, *mask: int | float = 9999*, *gpu_device: int = 0*, *small_nc: int = 0*) → None

Run the BG-Flood Model for the specified catchment area.

Parameters

- **engine** (*Engine*) – The engine used to connect to the database.
- **catchment_area** (*gpd.GeoDataFrame*) – A GeoDataFrame representing the catchment area.
- **model_output_path** (*pathlib.Path*) – The new file path for saving the BG Flood model output with the current timestamp included in the filename.
- **output_timestep** (*Union[int, float] = 0*) – Time step between model outputs in seconds. Default value is 0.0 (no output generated).
- **end_time** (*Union[int, float] = 0*) – Time in seconds when the model stops. Default value is 0.0 (model initializes but does not run).
- **resolution** (*Optional[Union[int, float]] = None*) – The grid resolution in meters for metric grids, representing the size of each grid cell. If not provided (default is None), the resolution of the Hydrologically conditioned DEM will be used as the grid resolution.
- **mask** (*Union[int, float] = 9999*) – The mask value is used to remove blocks from computation where the topography elevation (zb) is greater than the specified value. Default value is 9999.0 (no areas are masked).
- **gpu_device** (*int = 0*) – Specify the GPU device to be used. Default value is 0 (the first available GPU). Set the value to -1 to use the CPU. For other GPUs, use values 2 and above.
- **small_nc** (*int = 0*) – Specify whether the output should be saved as short integers to reduce the size of the output file. Set the value to 1 to enable short integer conversion, or set it to 0 to save all variables as floats. Default value is 0.

Returns
This function does not return any value.

Return type
None

src.flood_model.bg_flood_model.**main**(*selected_polygon_gdf: geopandas.GeoDataFrame*, *output_timestep: int | float = 0*, *end_time: int | float = 0*, *resolution: int | float | None = None*, *mask: int | float = 9999*, *gpu_device: int = 0*, *small_nc: int = 0*, *log_level:* src.digitaltwin.utils.LogLevel *= LogLevel.DEBUG*) → int

Generate BG-Flood model output for the requested catchment area, and incorporate the model output to GeoServer for visualization.

Parameters

- **selected_polygon_gdf** (*gpd.GeoDataFrame*) – A GeoDataFrame representing the selected polygon, i.e., the catchment area.
- **output_timestep** (*Union[int, float] = 0*) – Time step between model outputs in seconds. Default value is 0.0 (no output generated).
- **end_time** (*Union[int, float] = 0*) – Time in seconds when the model stops. Default value is 0.0 (model initializes but does not run).
- **resolution** (*Optional[Union[int, float]] = None*) – The grid resolution in meters for metric grids, representing the size of each grid cell. If not provided (default is None), the resolution of the Hydrologically conditioned DEM will be used as the grid resolution.
- **mask** (*Union[int, float] = 9999*) – The mask value is used to remove blocks from computation where the topography elevation (zb) is greater than the specified value. Default value is 9999.0 (no areas are masked).
- **gpu_device** (*int = 0*) – Specify the GPU device to be used. Default value is 0 (the first available GPU). Set the value to -1 to use the CPU. For other GPUs, use values 2 and above.

- **small_nc** (*int = 0*) – Specify whether the output should be saved as short integers to reduce the size of the output file. Set the value to 1 to enable short integer conversion, or set it to 0 to save all variables as floats. Default value is 0.
- **log_level** (*LogLevel = LogLevel.DEBUG*) – The log level to set for the root logger. Defaults to LogLevel.DEBUG. The available logging levels and their corresponding numeric values are: - LogLevel.CRITICAL (50) - LogLevel.ERROR (40) - LogLevel.WARNING (30) - LogLevel.INFO (20) - LogLevel.DEBUG (10) - LogLevel.NOTSET (0)

Returns
Returns the model id of the new flood_model produced

Return type
int

src.flood_model.bg_flood_model.**sample_polygon**

src.flood_model.flooded_buildings

Module Contents

Functions

store_flooded_buildings_in_database(engine, buildings, ...)	
find_flooded_buildings(→ pandas.DataFrame)	Creates a building DataFrame with attribute "is_flooded",
categorise_buildings_as_flooded(→ geopandas.GeoDataFrame)	Identifies all buildings in building_polygons that intersect with areas in flooded_polygons.
retrieve_building_outlines(→ geopandas.GeoDataFrame)	Retrieve building outlines for an area of interest from the database
polygonize_flooded_area(→ geopandas.GeoDataFrame)	Takes a flood depth raster and applies depth thresholding on it so that only areas

Attributes

wkt

src.flood_model.flooded_buildings.**store_flooded_buildings_in_database**(*engine: sqlalchemy.engine.Engine*, *buildings: pandas.DataFrame*, *flood_model_id: int*)

src.flood_model.flooded_buildings.**find_flooded_buildings**(*area_of_interest: geopandas.GeoDataFrame*, *flood_model_output_path: pathlib.Path*, *flood_depth_threshold: float*) → pandas.DataFrame

Creates a building DataFrame with attribute "is_flooded", depending on if the area for each building is flooded to a depth greater than or equal to flood_depth_threshold. the index, building_outline_id, matches building_outline_id from nz_building_outline table/

Parameters

- **area_of_interest** (*gpd.GeoDataFrame*) – A GeoDataFrame with a polygon specifying the area to get buildings for.
- **flood_model_output_path** (*pathlib.Path*) – Path to the flood model output file to be read
- **flood_depth_threshold** (*float*) – The minimum depth required to designate a pixel in the raster as flooded.

Returns

A pd.DataFrame specifying if each building is flooded or not.

Return type

pd.DataFrame

`src.flood_model.flooded_buildings.`**categorise_buildings_as_flooded**(*building_polygons: geopandas.GeoDataFrame*, *flood_polygons: geopandas.GeoDataFrame*) → geopandas.GeoDataFrame

Identifies all buildings in building_polygons that intersect with areas in flooded_polygons. :param building_polygons: A GeoDataFrame with each polygon representing a building outline :type building_polygons: gpd.GeoDataFrame :param flood_polygons: A GeoDataFrame with each polygon representing a flooded area :type flood_polygons: gpd.GeoDataFrame

Returns

A copy of building_polygons with an additional boolean attribute "is_flooded"

Return type

gpd.GeoDataFrame

`src.flood_model.flooded_buildings.`**retrieve_building_outlines**(*area_of_interest: geopandas.GeoDataFrame*) → geopandas.GeoDataFrame

Retrieve building outlines for an area of interest from the database

Parameters

area_of_interest (*gpd.GeoDataFrame*) – A GeoDataFrame polygon specifying the area of interest to retrieve buildings in.

Returns

A GeoDataFrame containing all of the building outlines in the area

Return type

gpd.GeoDataFrame

`src.flood_model.flooded_buildings.`**polygonize_flooded_area**(*flood_raster: xarray.DataArray*, *flood_depth_threshold: float*) → geopandas.GeoDataFrame

Takes a flood depth raster and applies depth thresholding on it so that only areas flooded deeper than or equal to flood_depth_threshold are represented. Returns the data in a collection of polygons

Parameters

- **flood_raster** (*xarray.DataArray*) – Raster with each pixel representing flood depth at the point
- **flood_depth_threshold** (*float*) – The minimum depth specified to consider a pixel in the raster flooded

Returns
A GeoDataFrame containing all of the building outlines in the area

Return type
gpd.GeoDataFrame

src.flood_model.flooded_buildings.**wkt = 'POLYGON ((172.68346232258148 -43.39283883172603, 172.68346232258148 -43.37441484114113,...'**

src.flood_model.serve_model

Takes generated models and adds them to GeoServer so they can be retrieved by API calls by the frontend or other clients

Module Contents

Functions

convert_nc_to_gtiff(→ pathlib.Path)	Creates a GeoTiff file from a netCDF model output. The Tiff represents the max flood height in the model output.
upload_gtiff_to_store(→ None)	Uploads a GeoTiff file to a new GeoServer store, to enable serving.
create_layer_from_store(→ None)	Creates a GeoServer Layer from a GeoServer store, making it ready to serve.
get_geoserver_url(→ str)	Retrieves full GeoServer URL from environment variables.
add_gtiff_to_geoserver(→ None)	Uploads a GeoTiff file to GeoServer, ready for serving to clients.
add_model_output_to_geoserver(model_output_pat model_id)	Adds the model output max depths to GeoServer, ready for serving.

Attributes

GEOSERVER_REST_URL
log

src.flood_model.serve_model.**GEOSERVER_REST_URL = 'http://localhost:8088/geoserver/rest/'**

src.flood_model.serve_model.**log**

`src.flood_model.serve_model.`**`convert_nc_to_gtiff`**`(`*nc_file_path: pathlib.Path*`)` → pathlib.Path

Creates a GeoTiff file from a netCDF model output. The Tiff represents the max flood height in the model output.

Parameters
`nc_file_path` (*`pathlib.Patj`*) – The file path to the netCDF file.

Returns
The filepath of the new GeoTiff file.

Return type
pathlib.Path

`src.flood_model.serve_model.`**`upload_gtiff_to_store`**`(`*geoserver_url: str*, *gtiff_filepath: pathlib.Path*, *store_name: str*, *workspace_name: str*`)` → None

Uploads a GeoTiff file to a new GeoServer store, to enable serving.

Parameters

- **`geoserver_url`** (*`str`*) – The URL to the geoserver instance.
- **`gtiff_filepath`** (*`pathlib.Path`*) – The filepath to the GeoTiff file to be served.
- **`store_name`** (*`str`*) – The name of the new Geoserver store to be created.
- **`workspace_name`** (*`str`*) – The name of the existing GeoServer workspace that the store is to be added to.

Returns
This function does not return anything

Return type
None

`src.flood_model.serve_model.`**`create_layer_from_store`**`(`*geoserver_url: str*, *layer_name: str*, *native_crs: str*, *workspace_name: str*`)` → None

Creates a GeoServer Layer from a GeoServer store, making it ready to serve.

Parameters

- **`geoserver_url`** (*`str`*) – The URL to the geoserver instance.
- **`layer_name`** (*`str`*) – Defines the name of the layer in GeoServer.
- **`native_crs`** (*`str`*) – The WKT form of the CRS of the data being shown in the layer.
- **`workspace_name`** (*`str`*) – The name of the existing GeoServer workspace that the store is to be added to.

Returns
This function does not return anything

Return type
None

`src.flood_model.serve_model.`**`get_geoserver_url`**`()` → str

Retrieves full GeoServer URL from environment variables.

Returns
The full GeoServer URL

Return type
str

`src.flood_model.serve_model.`**`add_gtiff_to_geoserver`**(*gtiff_filepath: pathlib.Path*, *workspace_name: str*, *model_id: int*) → None

Uploads a GeoTiff file to GeoServer, ready for serving to clients.

Parameters

- **`gtiff_filepath`** (*`pathlib.Path`*) – The filepath to the GeoTiff file to be served.
- **`workspace_name`** (*`str`*) – The name of the existing GeoServer workspace that the store is to be added to.
- **`model_id`** (*`int`*) – The id of the model being added, to facilitate layer naming.

Returns

This function does not return anything

Return type

None

`src.flood_model.serve_model.`**`add_model_output_to_geoserver`**(*model_output_path: pathlib.Path*, *model_id: int*)

Adds the model output max depths to GeoServer, ready for serving. The GeoServer layer name will be f"Output_{model_id}" and the workspace name will be "dt-model-outputs"

Parameters

- **`model_output_path`** (*`pathlib.Path`*) – The file path to the model output to serve.
- **`model_id`** (*`int`*) – The database id of the model output.

Returns

This function does not return anything

Return type

None

1.1.2 Submodules

1.1.2.1 `src.app`

The main web application that serves the Digital Twin to the web through a Rest API.

Module Contents

Functions

check_celery_alive(→ Callable[Ellipsis, flask.Response])	Function decorator to check if the Celery workers are running and return INTERNAL_SERVER_ERROR if they are down.
health_check(→ flask.Response)	Ping this endpoint to check that the server is up and running
get_status(→ flask.Response)	Retrieves status of a particular Celery backend task.
remove_task(→ flask.Response)	Deletes and stops a particular Celery backend task.
generate_model(→ flask.Response)	Generates a flood model for a given area.
create_wkt_from_coords(→ str)	Takes two points and creates a wkt bbox string from them
get_depth_at_point(→ flask.Response)	Finds the depths and times at a particular point for a given completed model output task.
get_distinct_column_values(→ flask.Response)	
valid_coordinates(→ bool)	Validates coordinates are in the valid range of WGS84

Attributes

app
gunicorn_logger

src.app.**app**

src.app.**check_celery_alive**(*f: Callable[Ellipsis, flask.Response]*) → Callable[Ellipsis, flask.Response]

Function decorator to check if the Celery workers are running and return INTERNAL_SERVER_ERROR if they are down.

Parameters
f (*Callable[..., Response]*) – The view function that is being decorated

Returns
INTERNAL_SERVER_ERROR if the celery workers are down, otherwise continue to function f

Return type
Response

src.app.**health_check**() → flask.Response

Ping this endpoint to check that the server is up and running Supported methods: GET

Returns
The HTTP Response. Expect OK if health check is successful

Return type
Response

src.app.**get_status**(*task_id*) → flask.Response

Retrieves status of a particular Celery backend task. Supported methods: GET

Parameters
`task_id` (*str*) – The id of the Celery task to retrieve status from

Returns
JSON response containing taskStatus

Return type
Response

`src.app.`**`remove_task`**`(`*task_id*`)` → flask.Response

Deletes and stops a particular Celery backend task. Supported methods: DELETE

Parameters
`task_id` (*str*) – The id of the Celery task to remove

Returns
ACCEPTED is the expected response

Return type
Response

`src.app.`**`generate_model`**`()` → flask.Response

Generates a flood model for a given area. Supported methods: POST POST values: {“bbox”: {“lat1”: number, “lat2”: number, “lng1”: number, “lng2”: number}}

Returns
ACCEPTED is the expected response. Response body contains Celery taskId

Return type
Response

`src.app.`**`create_wkt_from_coords`**`(`*lat1: float*, *lng1: float*, *lat2: float*, *lng2: float*`)` → str

Takes two points and creates a wkt bbox string from them

Parameters

- **`lat1`** (*float*) – latitude of first point
- **`lng1`** (*float*) – longitude of first point
- **`lat2`** (*float*) – latitude of second point
- **`lng2`** (*float*) – longitude of second point

Returns
bbox in wkt form generated from the two coordinates

Return type
str

`src.app.`**`get_depth_at_point`**`(`*task_id: str*`)` → flask.Response

Finds the depths and times at a particular point for a given completed model output task. Supported methods: GET Required query param values: “lat”: float, “lng”: float

Parameters
`task_id` (*str*) – The id of the completed task for generating a flood model.

Returns
Returns JSON response in the form {“depth”: Arrau<number>, “time”: Array<number>} representing the values for the given point.

Return type
Response

src.app.**get_distinct_column_values**(*table_name: str*) → flask.Response

src.app.**valid_coordinates**(*latitude: float*, *longitude: float*) → bool

Validates coordinates are in the valid range of WGS84 (-90 < latitude <= 90) and (-180 < longitude <= 180)

Parameters

- **latitude** (*float*) – The latitude part of the coordinate
- **longitude** (*float*) – The longitude part of the coordinate

Returns

True if both latitude and longitude are within their valid ranges.

Return type

bool

src.app.**gunicorn_logger**

1.1.2.2 `src.config`

Module Contents

Functions

get_env_variable(→ T)	Reads an environment variable, with settings to allow defaults, empty values, and type casting
_cast_str(→ T)	Takes a string and casts it to necessary primitive builtin types. Tested with int, float, and bool.

Attributes

T

src.config.**T**

src.config.**get_env_variable**(*var_name: str*, *default: T = None*, *allow_empty: bool = False*, *cast_to: type = str*) → T

Reads an environment variable, with settings to allow defaults, empty values, and type casting To read a boolean EXAMPLE_ENV_VAR=False use get_env_variable("EXAMPLE_ENV_VAR", cast_to=bool)

Parameters

- **var_name** (*str*) – The name of the environment variable to retrieve.
- **default** (*T = None*) – Default return value if the environment variable does not exist. Doesn't override empty string vars.
- **allow_empty** (*bool*) – If False then a KeyError will be raised if the environment variable is empty.
- **cast_to** (*Callable[[str], T]*) – The type to cast to e.g. str, int, or bool

Return type
The environment variable, or default if it does not exist, as type T.

Raises

- **KeyError** – If allow_empty is False and the environment variable is empty string or None
- **ValueError** – If cast_to is not compatible with the value stored.

src.config.**_cast_str**(*str_to_cast: str*, *cast_to: T*) → T

Takes a string and casts it to necessary primitive builtin types. Tested with int, float, and bool. For bools, this detects if the value is in the case-insensitive sets {"True", "T", "1"} or {"False", "F", "0"} and raises a ValueError if not. For example _cast_str("False", bool) -> False

Parameters

- **str_to_cast** (*str*) – The string that is going to be casted to the type
- **cast_to** (*Callable[[str], T]*) – The type to cast to e.g. bool

Return type
The string casted to type T defined by cast_to.

Raises
ValueError if [cast_to] is not compatible with the value stored. –

1.1.2.3 src.datacube_data

1.1.2.4 src.run_all

This script runs each module in the Digital Twin using a Sample Polygon.

Module Contents

Functions

main(→ None)	Runs each module in the Digital Twin using the selected polygon and the defined parameters for each module's

Attributes

DEFAULT_MODULES_TO_PARAMETERS
sample_polygon

src.run_all.**main**(*selected_polygon_gdf: geopandas.GeoDataFrame*, *modules_to_parameters: Dict[types.ModuleType, Dict[str, str | int | float | bool | None | enum.Enum]]*) → None

Runs each module in the Digital Twin using the selected polygon and the defined parameters for each module's main function.

Parameters

- **selected_polygon_gdf** (*gpd.GeoDataFrame*) – A GeoDataFrame representing the selected polygon, i.e., the catchment area.
- **modules_to_parameters** (*Dict[ModuleType, Dict[str, Union[str, int, float, bool, None, Enum]]]*) – A dictionary that associates each module with the parameters necessary for its main function, including the option to set the log level for each module's root logger. The available logging levels and their corresponding numeric values are: - LogLevel.CRITICAL (50) - LogLevel.ERROR (40) - LogLevel.WARNING (30) - LogLevel.INFO (20) - LogLevel.DEBUG (10) - LogLevel.NOTSET (0)

Returns
This function does not return any value.

Return type
None

src.run_all.**DEFAULT_MODULES_TO_PARAMETERS**

src.run_all.**sample_polygon**

1.1.2.5 src.tasks

Runs backend tasks using Celery. Allowing for multiple long-running tasks to complete in the background. Allows the frontend to send tasks and retrieve status later.

Module Contents

Classes

OnFailureStateTask	Task that switches state to FAILURE if an exception occurs

Functions

create_model_for_area(→ celery.result.GroupResult)	Creates a model for the area using series of chained (sequential) and grouped (parallel) sub-tasks.
add_base_data_to_db(→ None)	Task to ensure static base data for the given area is added to the database
process_dem(selected_polygon_wkt)	Task to ensure hydrologically-conditioned DEM is processed for the given area and added to the database.
generate_rainfall_inputs(selected_polygon_wkt)	Task to ensure rainfall input data for the given area is added to the database and model input files are created.
generate_tide_inputs(selected_polygon_wkt, ...)	Task to ensure tide input data for the given area is added to the database and model input files are created.
generate_river_inputs(selected_polygon_wkt)	Task to ensure river input data for the given area is added to the database and model input files are created.
run_flood_model(→ int)	Task to run flood model using input data from previous tasks.
wkt_to_gdf(→ geopandas.GeoDataFrame)	Transforms a WKT string polygon into a GeoDataFrame
get_depth_by_time_at_point(→ Tuple[List[float], ...)	Task to query a point in a flood model output and return the list of depths and times.
get_distinct_column_values(→ dict)	

Attributes

message_broker_url
app
log
x

src.tasks.**message_broker_url**

src.tasks.**app**

src.tasks.**log**

class src.tasks.**OnFailureStateTask**

Bases: *app*

Task that switches state to FAILURE if an exception occurs

on_failure(*_exc*, *_task_id*, *_args*, *_kwargs*, *_einfo*)

src.tasks.**create_model_for_area**(*selected_polygon_wkt: str*, *scenario_options: dict*) → celery.result.GroupResult

Creates a model for the area using series of chained (sequential) and grouped (parallel) sub-tasks.

Parameters
`selected_polygon_wkt` (*str*) – The polygon defining the selected area to run the model for. Defined in WKT form.

Returns
The task result for the long-running group of tasks. The task ID represents the final task in the group.

Return type
result.GroupResult

`src.tasks.`**`add_base_data_to_db`**(*selected_polygon_wkt: str*) → None

Task to ensure static base data for the given area is added to the database

Parameters
`selected_polygon_wkt` (*str*) – The polygon defining the selected area to add base data for. Defined in WKT form.

Returns
This task does not return anything

Return type
None

`src.tasks.`**`process_dem`**(*selected_polygon_wkt: str*)

Task to ensure hydrologically-conditioned DEM is processed for the given area and added to the database.

Parameters
`selected_polygon_wkt` (*str*) – The polygon defining the selected area to process the DEM for. Defined in WKT form.

Returns
This task does not return anything

Return type
None

`src.tasks.`**`generate_rainfall_inputs`**(*selected_polygon_wkt: str*)

Task to ensure rainfall input data for the given area is added to the database and model input files are created.

Parameters
`selected_polygon_wkt` (*str*) – The polygon defining the selected area to add rainfall data for. Defined in WKT form.

Returns
This task does not return anything

Return type
None

`src.tasks.`**`generate_tide_inputs`**(*selected_polygon_wkt: str*, *scenario_options: dict*)

Task to ensure tide input data for the given area is added to the database and model input files are created.

Parameters
`selected_polygon_wkt` (*str*) – The polygon defining the selected area to add tide data for. Defined in WKT form.

Returns
This task does not return anything

Return type
None

src.tasks.**generate_river_inputs**(*selected_polygon_wkt: str*)

Task to ensure river input data for the given area is added to the database and model input files are created.

Parameters

selected_polygon_wkt (*str*) – The polygon defining the selected area to add river data for. Defined in WKT form.

Returns

This task does not return anything

Return type

None

src.tasks.**run_flood_model**(*selected_polygon_wkt: str*) → int

Task to run flood model using input data from previous tasks.

Parameters

selected_polygon_wkt (*str*) – The polygon defining the selected area to run the flood model for. Defined in WKT form.

Returns

The database ID of the flood model that has been run.

Return type

int

src.tasks.**wkt_to_gdf**(*wkt: str*) → geopandas.GeoDataFrame

Transforms a WKT string polygon into a GeoDataFrame

Parameters

wkt (*str*) – The WKT form of the polygon to be transformed. In WGS84 CRS (epsg:4326).

Returns

The GeoDataFrame form of the polygon after being transformed.

Return type

gpd.GeoDataFrame

src.tasks.**get_depth_by_time_at_point**(*model_id: int*, *lat: float*, *lng: float*) → Tuple[List[float], List[float]]

Task to query a point in a flood model output and return the list of depths and times.

Parameters

- **model_id** (*int*) – The database id of the model output to query.
- **lat** (*float*) – The latitude of the point to query.
- **lng** (*float*) – The longitude of the point to query.

Returns

Tuple of depths list and times list for the pixel in the output nearest to the point.

Return type

Tuple[List[float], List[float]]

src.tasks.**get_distinct_column_values**(*table_name: str*) → dict

src.tasks.**x**

1.1.3 Package Contents

src.**__version__** = **'0.1.0'**

CHAPTER

TWO

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

S

src, 1
src.app, 107
src.config, 110
src.datacube_data, 111
src.digitaltwin, 1
src.digitaltwin.data_to_db, 1
src.digitaltwin.get_data_using_geoapis, 6
src.digitaltwin.instructions_records_to_db, 8
src.digitaltwin.run, 10
src.digitaltwin.setup_environment, 11
src.digitaltwin.tables, 13
src.digitaltwin.utils, 19
src.dynamic_boundary_conditions, 22
src.dynamic_boundary_conditions.rainfall, 22
src.dynamic_boundary_conditions.rainfall.hirds_rainfall_data_from_db, 22
src.dynamic_boundary_conditions.rainfall.hirds_rainfall_data_to_db, 25
src.dynamic_boundary_conditions.rainfall.hyetograph, 27
src.dynamic_boundary_conditions.rainfall.main_rainfall, 33
src.dynamic_boundary_conditions.rainfall.rainfall_data_from_hirds, 34
src.dynamic_boundary_conditions.rainfall.rainfall_enum, 37
src.dynamic_boundary_conditions.rainfall.rainfall_model_input, 38
src.dynamic_boundary_conditions.rainfall.rainfall_sites, 42
src.dynamic_boundary_conditions.rainfall.thiessen_polygons, 43
src.dynamic_boundary_conditions.river, 46
src.dynamic_boundary_conditions.river.align_rec1_osm, 46
src.dynamic_boundary_conditions.river.hydrograph, 51
src.dynamic_boundary_conditions.river.main_river, 54
src.dynamic_boundary_conditions.river.osm_waterways, 56
src.dynamic_boundary_conditions.river.river_data_to_from_d
57
src.dynamic_boundary_conditions.river.river_enum, 59
src.dynamic_boundary_conditions.river.river_inflows, 60
src.dynamic_boundary_conditions.river.river_model_input, 62
src.dynamic_boundary_conditions.river.river_network_for_ac
63
src.dynamic_boundary_conditions.river.river_network_to_fro
69
src.dynamic_boundary_conditions.tide, 74
src.dynamic_boundary_conditions.tide.main_tide_slr, 74
src.dynamic_boundary_conditions.tide.sea_level_rise_data, 76
src.dynamic_boundary_conditions.tide.tide_data_from_niwa, 78
src.dynamic_boundary_conditions.tide.tide_enum, 88
src.dynamic_boundary_conditions.tide.tide_query_location, 89
src.dynamic_boundary_conditions.tide.tide_slr_combine, 93
src.dynamic_boundary_conditions.tide.tide_slr_model_input, 97
src.flood_model, 98
src.flood_model.bg_flood_model, 98
src.flood_model.flooded_buildings, 103
src.flood_model.serve_model, 105
src.run_all, 111
src.tasks, 112

INDEX

Symbols

__table_args__ (*src.digitaltwin.tables.RiverNetworkExclusions attribute*), 16
__tablename__ (*src.digitaltwin.tables.BGFloodModelOutput attribute*), 17, 18
__tablename__ (*src.digitaltwin.tables.BuildingFloodStatus attribute*), 18
__tablename__ (*src.digitaltwin.tables.GeospatialLayers attribute*), 13, 14
__tablename__ (*src.digitaltwin.tables.RiverNetworkExclusions attribute*), 15, 16
__tablename__ (*src.digitaltwin.tables.RiverNetworkOutput attribute*), 16, 17
__tablename__ (*src.digitaltwin.tables.UserLogInfo attribute*), 15
__version__ (*in module src*), 116
_cast_str() (*in module src.config*), 111

A

add_absent_edges_to_network() (*in module src.dynamic_boundary_conditions.river.river_network_for_aoi*), 66
add_base_data_to_db() (*in module src.tasks*), 114
add_crs_to_latest_model_output() (*in module src.flood_model.bg_flood_model*), 100
add_each_site_rainfall_data() (*in module src.dynamic_boundary_conditions.rainfall.hirds_rainfall_data_to_db*), 26
add_edge_directions_to_network_data() (*in module src.dynamic_boundary_conditions.river.river_network_for_aoi*), 67
add_gtiff_to_geoserver() (*in module src.flood_model.serve_model*), 106
add_initial_edges_to_network() (*in module src.dynamic_boundary_conditions.river.river_network_for_aoi*), 66
add_model_output_to_geoserver() (*in module src.flood_model.serve_model*), 107
add_network_exclusions_to_db() (*in module src.dynamic_boundary_conditions.river.river_network_to_from_db*), 70
add_nodes_intersection_type() (*in module src.dynamic_boundary_conditions.river.river_network_for_aoi*), 64
add_nodes_to_network() (*in module src.dynamic_boundary_conditions.river.river_network_for_aoi*), 65
add_nodes_to_rec1() (*in module src.dynamic_boundary_conditions.river.river_network_for_aoi*), 64
add_rainfall_data_to_db() (*in module src.dynamic_boundary_conditions.rainfall.hirds_rainfall_data_to*), 26
add_slr_to_tide() (*in module src.dynamic_boundary_conditions.tide.tide_slr_combine*), 95
add_time_information() (*in module src.dynamic_boundary_conditions.rainfall.hyetograph*), 30
add_time_information() (*in module src.dynamic_boundary_conditions.tide.tide_data_from_niwa*), 86
align_rec1_with_osm() (*in module src.dynamic_boundary_conditions.river.align_rec1_osm*), 50
ALT_BLOCK (*src.dynamic_boundary_conditions.rainfall.rainfall_enum.Hyet attribute*), 37
app (*in module src.app*), 108
app (*in module src.tasks*), 113
ApproachType (*class in src.dynamic_boundary_conditions.tide.tide_enum*), 89

B

Base (*in module src.digitaltwin.setup_environment*), 12
Base (*in module src.digitaltwin.tables*), 13
Base (*in module src.flood_model.bg_flood_model*), 98
BGFloodModelOutput (*class in src.digitaltwin.tables*), 17
BlockStructure (*class in src.dynamic_boundary_conditions.rainfall.rainfall_data_from_hi*), 35
BoundType (*class in src.dynamic_boundary_conditions.river.river_enum*), 59

build_rec1_river_network() (in module src.dynamic_boundary_conditions.river.river_network_for_aoi), 68
building_outline_id (src.digitaltwin.tables.BuildingFloodStatus attribute), 18
BuildingFloodStatus (class in src.digitaltwin.tables), 18

C

categorise_buildings_as_flooded() (in module src.flood_model.flooded_buildings), 104
categorize_exploded_multi_intersect() (in module src.dynamic_boundary_conditions.river.align_rec1_osm), 48
category (src.dynamic_boundary_conditions.rainfall.rainfall_data_from_hirds.BlockStructure attribute), 36
check_celery_alive() (in module src.app), 108
check_table_exists() (in module src.digitaltwin.tables), 18
CHICAGO (src.dynamic_boundary_conditions.rainfall.rainfall_enum.HyetoMethod attribute), 37
clean_fetched_vector_data() (in module src.digitaltwin.get_data_using_geoapis), 7
clean_rec1_inflow_data() (in module src.dynamic_boundary_conditions.river.hydrograph), 51
configure_osm_cache() (in module src.dynamic_boundary_conditions.river.osm_waterways), 56
convert_nc_to_gtiff() (in module src.flood_model.serve_model), 105
convert_to_nz_timezone() (in module src.dynamic_boundary_conditions.tide.tide_data_from_niwa), 82
convert_to_tabular_data() (in module src.dynamic_boundary_conditions.rainfall.rainfall_data_from_hirds), 36
coverage_area (src.digitaltwin.tables.GeospatialLayers attribute), 14
create_layer_from_store() (in module src.flood_model.serve_model), 106
create_model_for_area() (in module src.tasks), 113
create_rain_data_cube() (in module src.dynamic_boundary_conditions.rainfall.rainfall_model_input), 40
create_table() (in module src.digitaltwin.tables), 18
create_wkt_from_coords() (in module src.app), 109
created_at (src.digitaltwin.tables.BGFloodModelOutput attribute), 17, 18
created_at (src.digitaltwin.tables.RiverNetworkOutput attribute), 17
created_at (src.digitaltwin.tables.UserLogInfo attribute), 15
CRITICAL (src.digitaltwin.utils.LogLevel attribute), 20

D

data_provider (src.digitaltwin.tables.GeospatialLayers attribute), 14
DatumType (class in src.dynamic_boundary_conditions.tide.tide_enum), 88
db_rain_table_name() (in module src.dynamic_boundary_conditions.rainfall.hirds_rainfall_data_to_db), 25
DEBUG (src.digitaltwin.utils.LogLevel attribute), 20
DEFAULT_MODULES_TO_PARAMETERS (in module src.run_all), 112
determine_multi_intersect_inflow_index() (in module src.dynamic_boundary_conditions.river.align_rec1_osm), 47

E

ERROR (src.digitaltwin.utils.LogLevel attribute), 20
exclusion_cause (src.digitaltwin.tables.RiverNetworkExclusions attribute), 16
execute_query() (in module src.digitaltwin.tables), 19
extract_valid_ari_values() (in module src.dynamic_boundary_conditions.river.hydrograph), 52

F

fetch_osm_waterways() (in module src.dynamic_boundary_conditions.river.osm_waterways), 56
fetch_tide_data() (in module src.dynamic_boundary_conditions.tide.tide_data_from_niwa), 81
fetch_tide_data_around_highest_tide() (in module src.dynamic_boundary_conditions.tide.tide_data_from_niwa), 83
fetch_tide_data_for_requested_period() (in module src.dynamic_boundary_conditions.tide.tide_data_from_niwa), 82
fetch_tide_data_from_niwa() (in module src.dynamic_boundary_conditions.tide.tide_data_from_niwa), 83
fetch_vector_data_using_geoapis() (in module src.digitaltwin.get_data_using_geoapis), 7
file_name (src.digitaltwin.tables.BGFloodModelOutput attribute), 17, 18
file_path (src.digitaltwin.tables.BGFloodModelOutput attribute), 17, 18
filter_for_duration() (in module src.dynamic_boundary_conditions.rainfall.hirds_rainfall_data_from_db), 22

find_flooded_buildings() (*in module src.flood_model.flooded_buildings*), 103
flood_model_id (*src.digitaltwin.tables.BuildingFloodStatus attribute*), 18

G

gen_api_query_param_list() (*in module src.dynamic_boundary_conditions.tide.tide_data_from_niwa*), 80
generate_model() (*in module src.app*), 109
generate_rain_model_input() (*in module src.dynamic_boundary_conditions.rainfall.rainfall_model_input*), 41
generate_rainfall_inputs() (*in module src.tasks*), 114
generate_river_inputs() (*in module src.tasks*), 114
generate_river_model_input() (*in module src.dynamic_boundary_conditions.river.river_model_input*), 62
generate_tide_inputs() (*in module src.tasks*), 114
generate_uniform_boundary_input() (*in module src.dynamic_boundary_conditions.tide.tide_slr_model_input*), 97
geometry (*src.digitaltwin.tables.BGFloodModelOutput attribute*), 18
geometry (*src.digitaltwin.tables.RiverNetworkExclusions attribute*), 16
geometry (*src.digitaltwin.tables.RiverNetworkOutput attribute*), 17
geometry (*src.digitaltwin.tables.UserLogInfo attribute*), 15
GEOMETRY_NAMES (*src.digitaltwin.get_data_using_geoapis.WFS attribute*), 7
GEOSERVER_REST_URL (*in module src.flood_model.serve_model*), 105
GeospatialLayers (*class in src.digitaltwin.tables*), 13
get_catchment_area() (*in module src.digitaltwin.utils*), 21
get_catchment_boundary_centroids() (*in module src.dynamic_boundary_conditions.tide.tide_query_location*), 91
get_catchment_boundary_info() (*in module src.dynamic_boundary_conditions.tide.tide_query_location*), 90
get_catchment_boundary_lines() (*in module src.dynamic_boundary_conditions.tide.tide_query_location*), 91
get_closest_slr_data() (*in module src.dynamic_boundary_conditions.tide.sea_level_rise_data*), 77
get_combined_tide_slr_data() (*in module src.dynamic_boundary_conditions.tide.tide_slr_combine*), 95
get_connection_from_profile() (*in module src.digitaltwin.setup_environment*), 12
get_data_from_hirds() (*in module src.dynamic_boundary_conditions.rainfall.rainfall_data_from_hi*
35
get_database() (*in module src.digitaltwin.setup_environment*), 12
get_date_ranges() (*in module src.dynamic_boundary_conditions.tide.tide_data_from_niwa*), 80
get_depth_at_point() (*in module src.app*), 109
get_depth_by_time_at_point() (*in module src.tasks*), 115
get_distinct_column_values() (*in module src.app*), 109
get_distinct_column_values() (*in module src.tasks*), 115
get_elevations_near_rec1_entry_point() (*in module src.dynamic_boundary_conditions.river.river_inflows*), 60
get_engine() (*in module src.digitaltwin.setup_environment*), 12
get_env_variable() (*in module src.config*), 110
get_existing_geospatial_layers() (*in module src.digitaltwin.instructions_records_to_db*), 9
get_existing_network() (*in module src.dynamic_boundary_conditions.river.river_network_to_from_d*
73
get_existing_network_metadata_from_db() (*in module src.dynamic_boundary_conditions.river.river_network_to*
73
get_exploded_multi_intersect() (*in module src.dynamic_boundary_conditions.river.align_rec1_osm*), 47
get_geoserver_url() (*in module src.flood_model.serve_model*), 106
get_geospatial_layer_info() (*in module src.digitaltwin.data_to_db*), 3
get_highest_tide_date_span() (*in module src.dynamic_boundary_conditions.tide.tide_data_from_niwa*), 84
get_highest_tide_datetime() (*in module src.dynamic_boundary_conditions.tide.tide_data_from_niwa*), 83
get_highest_tide_datetime_span() (*in module src.dynamic_boundary_conditions.tide.tide_data_from_niwa*), 84
get_hydro_dem_boundary_lines() (*in module src.dynamic_boundary_conditions.river.main_river*), 54
get_hydrograph_data() (*in module src.dynamic_boundary_conditions.river.hydrograph*), 53
get_hyetograph_data() (*in module*

src.dynamic_boundary_conditions.rainfall.hyetograph), 31
get_interp_incremental_data() (*in module src.dynamic_boundary_conditions.rainfall.hyetograph*), 29
get_interpolated_data() (*in module src.dynamic_boundary_conditions.rainfall.hyetograph*), 28
get_interpolated_slr_scenario_data() (*in module src.dynamic_boundary_conditions.tide.tide_slr_combined*), 94
get_layout_structure_of_data() (*in module src.dynamic_boundary_conditions.rainfall.rainfall_data_from_hirds*), 36
get_min_elevation_river_input_point() (*in module src.dynamic_boundary_conditions.river.river_inflows*), 61
get_model_output_metadata() (*in module src.flood_model.bg_flood_model*), 99
get_multi_intersect_inflows() (*in module src.dynamic_boundary_conditions.river.align_rec1_osm*), 48
get_network_output_metadata() (*in module src.dynamic_boundary_conditions.river.river_network_to_from_db*), 71
get_new_model_output_path() (*in module src.flood_model.bg_flood_model*), 99
get_new_network_output_paths() (*in module src.dynamic_boundary_conditions.river.river_network_to_from_db*), 71
get_next_network_id() (*in module src.dynamic_boundary_conditions.river.river_network_to_from_db*), 70
get_niwa_rec1_data() (*in module src.dynamic_boundary_conditions.river.river_data_to_from_db*), 58
get_non_existing_records() (*in module src.digitaltwin.instructions_records_to_db*), 10
get_non_intersection_area_from_db() (*in module src.digitaltwin.data_to_db*), 4
get_non_intersection_centroid_position() (*in module src.dynamic_boundary_conditions.tide.tide_query_location*), 91
get_non_nz_geospatial_layers() (*in module src.digitaltwin.data_to_db*), 2
get_nz_boundary() (*in module src.digitaltwin.utils*), 21
get_nz_coastline_from_db() (*in module src.dynamic_boundary_conditions.tide.tide_query_location*), 90
get_nz_geospatial_layers() (*in module src.digitaltwin.data_to_db*), 2
get_one_site_rainfall_data() (*in module src.dynamic_boundary_conditions.rainfall.hirds_rainfall_data_from_db*), 22
get_or_load_tide_data_for_demo() (*in module src.dynamic_boundary_conditions.tide.main_tide_slr*), 75
get_osm_waterways_data() (*in module src.dynamic_boundary_conditions.river.osm_waterways*), 57
get_osm_waterways_on_bbox() (*in module src.dynamic_boundary_conditions.river.align_rec1_osm*), 49
get_query_loc_coords_position() (*in module src.dynamic_boundary_conditions.tide.tide_data_from_niwa*), 79
get_rainfall_sites_data() (*in module src.dynamic_boundary_conditions.rainfall.rainfall_sites*), 42
get_rainfall_sites_in_df() (*in module src.dynamic_boundary_conditions.rainfall.rainfall_sites*), 42
get_rec1_data_with_sdc_from_db() (*in module src.dynamic_boundary_conditions.river.river_data_to_from_db*), 58
get_rec1_inflow_scenario_data() (*in module src.dynamic_boundary_conditions.river.hydrograph*), 52
get_rec1_inflows_aligned_to_osm() (*in module src.dynamic_boundary_conditions.river.align_rec1_osm*), 50
get_rec1_inflows_on_bbox() (*in module src.dynamic_boundary_conditions.river.align_rec1_osm*), 49
get_rec1_inflows_with_input_points() (*in module src.dynamic_boundary_conditions.river.river_inflows*), 61
get_rec1_network_data_on_bbox() (*in module src.dynamic_boundary_conditions.river.align_rec1_osm*), 46
get_rec1_river_network() (*in module src.dynamic_boundary_conditions.river.river_network_for_aoi*), 69
get_regional_council_clipped_from_db() (*in module src.dynamic_boundary_conditions.tide.tide_query_location*), 90
get_sdc_data_from_db() (*in module src.dynamic_boundary_conditions.river.river_data_to_from_db*), 58
get_single_intersect_inflows() (*in module src.dynamic_boundary_conditions.river.align_rec1_osm*), 47
get_site_url_key() (*in module src.dynamic_boundary_conditions.rainfall.rainfall_data_from_hirds*), 35
get_sites_id_in_catchment() (*in module src.dynamic_boundary_conditions.rainfall.hirds_rainfall_data_to_db*),

25
get_sites_id_not_in_db() (*in module src.dynamic_boundary_conditions.rainfall.hirds_rainfall_data_to_db*), 26
get_sites_within_aoi() (*in module src.dynamic_boundary_conditions.rainfall.thiessen_polygons*), 43
get_slr_data_from_db() (*in module src.dynamic_boundary_conditions.tide.sea_level_rise_data*), 77
get_slr_data_from_nz_searise() (*in module src.dynamic_boundary_conditions.tide.sea_level_rise_data*), 77
get_slr_scenario_data() (*in module src.dynamic_boundary_conditions.tide.tide_slr_combine*), 93
get_status() (*in module src.app*), 108
get_storm_length_increment_data() (*in module src.dynamic_boundary_conditions.rainfall.hyetograph*), 29
get_tide_data() (*in module src.dynamic_boundary_conditions.tide.tide_data_from_niwa*), 87
get_tide_query_locations() (*in module src.dynamic_boundary_conditions.tide.tide_query_location*), 92
get_time_mins_to_add() (*in module src.dynamic_boundary_conditions.tide.tide_data_from_niwa*), 85
get_transposed_data() (*in module src.dynamic_boundary_conditions.rainfall.hyetograph*), 28
get_unique_nodes_dict() (*in module src.dynamic_boundary_conditions.river.river_network_for_aoi*), 64
get_valid_bg_flood_dir() (*in module src.flood_model.bg_flood_model*), 99
get_vector_data_id_not_in_db() (*in module src.digitaltwin.data_to_db*), 3
gunicorn_logger (*in module src.app*), 110

H

health_check() (*in module src.app*), 108
hyetograph() (*in module src.dynamic_boundary_conditions.rainfall.hyetograph*), 32
hyetograph_data_wide_to_long() (*in module src.dynamic_boundary_conditions.rainfall.hyetograph*), 32
hyetograph_depth_to_intensity() (*in module src.dynamic_boundary_conditions.rainfall.hyetograph*), 31
HyetoMethod (*class in src.dynamic_boundary_conditions.rainfall.rainfall_enum*), 37

I

identify_absent_edges_to_add() (*in module src.dynamic_boundary_conditions.river.river_network_for_aoi*), 66
INFO (*src.digitaltwin.utils.LogLevel attribute*), 20
is_flooded (*src.digitaltwin.tables.BuildingFloodStatus attribute*), 18

K

KING_TIDE (*src.dynamic_boundary_conditions.tide.tide_enum.ApproachTy attribute*), 89

L

LAT (*src.dynamic_boundary_conditions.tide.tide_enum.DatumType attribute*), 88, 89
layer_id (*src.digitaltwin.tables.GeospatialLayers attribute*), 14
log (*in module src.digitaltwin.data_to_db*), 2
log (*in module src.digitaltwin.instructions_records_to_db*), 8
log (*in module src.digitaltwin.setup_environment*), 12
log (*in module src.digitaltwin.utils*), 20
log (*in module src.dynamic_boundary_conditions.rainfall.hirds_rainfall_da*), 25
log (*in module src.dynamic_boundary_conditions.rainfall.rainfall_model_i*), 38
log (*in module src.dynamic_boundary_conditions.rainfall.rainfall_sites*), 42
log (*in module src.dynamic_boundary_conditions.rainfall.thiessen_polygon*), 43
log (*in module src.dynamic_boundary_conditions.river.river_data_to_from*), 58
log (*in module src.dynamic_boundary_conditions.river.river_model_input*), 62
log (*in module src.dynamic_boundary_conditions.river.river_network_for_*), 64
log (*in module src.dynamic_boundary_conditions.river.river_network_to_f*), 70
log (*in module src.dynamic_boundary_conditions.tide.main_tide_slr*), 74
log (*in module src.dynamic_boundary_conditions.tide.sea_level_rise_data*), 77
log (*in module src.dynamic_boundary_conditions.tide.tide_slr_model_inpu*), 97
log (*in module src.flood_model.bg_flood_model*), 98
log (*in module src.flood_model.serve_model*), 105
log (*in module src.tasks*), 113
log_execution_info() (*in module src.digitaltwin.utils*), 21
LogLevel (*class in src.digitaltwin.utils*), 20
LOWER (*src.dynamic_boundary_conditions.river.river_enum.BoundType attribute*), 59, 60

M

main() (*in module src.digitaltwin.run*), 11
main() (*in module src.dynamic_boundary_conditions.rainfall.main_rainfall*), 33
main() (*in module src.dynamic_boundary_conditions.river.main_river*), 55
main() (*in module src.dynamic_boundary_conditions.tide.main_tide_slr*), 75
main() (*in module src.flood_model.bg_flood_model*), 102
main() (*in module src.run_all*), 111
mean_catchment_rainfall() (*in module src.dynamic_boundary_conditions.rainfall.rainfall_model_input*), 39
message_broker_url (*in module src.tasks*), 113
MFE (*class in src.digitaltwin.get_data_using_geoapis*), 7
MIDDLE (*src.dynamic_boundary_conditions.river.river_enum.BoundType attribute*), 59, 60
model_output_from_db_by_id() (*in module src.flood_model.bg_flood_model*), 100
module
- src, 1
- src.app, 107
- src.config, 110
- src.datacube_data, 111
- src.digitaltwin, 1
- src.digitaltwin.data_to_db, 1
- src.digitaltwin.get_data_using_geoapis, 6
- src.digitaltwin.instructions_records_to_db, 8
- src.digitaltwin.run, 10
- src.digitaltwin.setup_environment, 11
- src.digitaltwin.tables, 13
- src.digitaltwin.utils, 19
- src.dynamic_boundary_conditions, 22
- src.dynamic_boundary_conditions.rainfall, 22
- src.dynamic_boundary_conditions.rainfall.hirds_rainfall_data_from_db, 22
- src.dynamic_boundary_conditions.rainfall.hirds_rainfall_data_to_db, 25
- src.dynamic_boundary_conditions.rainfall.hyetograph, 27
- src.dynamic_boundary_conditions.rainfall.main_rainfall, 33
- src.dynamic_boundary_conditions.rainfall.rainfall_data_from_hirds, 34
- src.dynamic_boundary_conditions.rainfall.rainfall_enum, 37
- src.dynamic_boundary_conditions.rainfall.rainfall_model_input, 38
- src.dynamic_boundary_conditions.rainfall.rainfall_sites, 42
- src.dynamic_boundary_conditions.rainfall.thiessen_poly 43
- src.dynamic_boundary_conditions.river, 46
- src.dynamic_boundary_conditions.river.align_rec1_osm, 46
- src.dynamic_boundary_conditions.river.hydrograph, 51
- src.dynamic_boundary_conditions.river.main_river, 54
- src.dynamic_boundary_conditions.river.osm_waterways, 56
- src.dynamic_boundary_conditions.river.river_data_to_fr 57
- src.dynamic_boundary_conditions.river.river_enum, 59
- src.dynamic_boundary_conditions.river.river_inflows, 60
- src.dynamic_boundary_conditions.river.river_model_inpu 62
- src.dynamic_boundary_conditions.river.river_network_fo 63
- src.dynamic_boundary_conditions.river.river_network_to 69
- src.dynamic_boundary_conditions.tide, 74
- src.dynamic_boundary_conditions.tide.main_tide_slr, 74
- src.dynamic_boundary_conditions.tide.sea_level_rise_da 76
- src.dynamic_boundary_conditions.tide.tide_data_from_ni 78
- src.dynamic_boundary_conditions.tide.tide_enum, 88
- src.dynamic_boundary_conditions.tide.tide_query_locati 89
- src.dynamic_boundary_conditions.tide.tide_slr_combine, 93
- src.dynamic_boundary_conditions.tide.tide_slr_model_in 97
- src.flood_model, 98
- src.flood_model.bg_flood_model, 98
- src.flood_model.flooded_buildings, 103
- src.flood_model.serve_model, 105
- src.run_all, 111
- src.tasks, 112

MSL (*src.dynamic_boundary_conditions.tide.tide_enum.DatumType attribute*), 88, 89

N

NETLOC_API (*src.digitaltwin.get_data_using_geoapis.MFE attribute*), 7
network_data_path (*src.digitaltwin.tables.RiverNetworkOutput attribute*), 16, 17
network_path (*src.digitaltwin.tables.RiverNetworkOutput attribute*), 16, 17

non_nz_geospatial_layers_data_to_db() (*in module src.digitaltwin.data_to_db*), 5
NoNonIntersectionError, 2
NoTideDataException, 89
NOTSET (*src.digitaltwin.utils.LogLevel attribute*), 20
nz_geospatial_layers_data_to_db() (*in module src.digitaltwin.data_to_db*), 3

O

objectid (*src.digitaltwin.tables.RiverNetworkExclusions attribute*), 16
on_failure() (*src.tasks.OnFailureStateTask method*), 113
OnFailureStateTask (*class in src.tasks*), 113

P

PERIOD_TIDE (*src.dynamic_boundary_conditions.tide.tide_enum.ApproachType attribute*), 89
polygonize_flooded_area() (*in module src.flood_model.flooded_buildings*), 104
prepare_bg_flood_model_inputs() (*in module src.flood_model.bg_flood_model*), 101
prepare_network_data_for_construction() (*in module src.dynamic_boundary_conditions.river.river_network_for_aoi*), 65
process_boundary_input_files() (*in module src.flood_model.bg_flood_model*), 100
process_dem() (*in module src.tasks*), 114
process_existing_non_nz_geospatial_layers() (*in module src.digitaltwin.data_to_db*), 5
process_new_non_nz_geospatial_layers() (*in module src.digitaltwin.data_to_db*), 4
process_rain_input_files() (*in module src.flood_model.bg_flood_model*), 100
process_river_input_files() (*in module src.flood_model.bg_flood_model*), 100

R

rainfall_data_from_db() (*in module src.dynamic_boundary_conditions.rainfall.hirds_rainfall_data_from_db*), 23
rainfall_data_to_db() (*in module src.dynamic_boundary_conditions.rainfall.hirds_rainfall_data_to_db*), 27
rainfall_sites_to_db() (*in module src.dynamic_boundary_conditions.rainfall.rainfall_sites*), 43
RainInputType (*class in src.dynamic_boundary_conditions.rainfall.rainfall_enum*), 37
rcp (*src.dynamic_boundary_conditions.rainfall.rainfall_data_from_hirds.BlockStructure attribute*), 35, 36
read_and_check_instructions_file() (*in module src.digitaltwin.instructions_records_to_db*), 9
rec1_network_id (*src.digitaltwin.tables.RiverNetworkExclusions attribute*), 15, 16
rec1_network_id (*src.digitaltwin.tables.RiverNetworkOutput attribute*), 16, 17
remove_existing_boundary_inputs() (*in module src.dynamic_boundary_conditions.tide.main_tide_slr*), 74
remove_existing_rain_inputs() (*in module src.dynamic_boundary_conditions.rainfall.main_rainfall*), 33
remove_existing_river_inputs() (*in module src.dynamic_boundary_conditions.river.main_river*), 55
remove_task() (*in module src.app*), 109
remove_unconnected_edges_from_network() (*in module src.dynamic_boundary_conditions.river.river_network_fo*
68
retrieve_building_outlines() (*in module src.flood_model.flooded_buildings*), 104
retrieve_hydro_dem_info() (*in module src.dynamic_boundary_conditions.river.main_river*), 54
RiverNetworkExclusions (*class in src.digitaltwin.tables*), 15
RiverNetworkOutput (*class in src.digitaltwin.tables*), 16
run_bg_flood_model() (*in module src.flood_model.bg_flood_model*), 101
run_flood_model() (*in module src.tasks*), 115

S

sample_polygon (*in module src.digitaltwin.run*), 11
sample_polygon (*in module src.dynamic_boundary_conditions.rainfall.main_rainfall*), 34
sample_polygon (*in module src.dynamic_boundary_conditions.river.main_river*), 56
sample_polygon (*in module src.dynamic_boundary_conditions.tide.main_tide_slr*), 76
sample_polygon (*in module src.flood_model.bg_flood_model*), 103
sample_polygon (*in module src.run_all*), 112
setup_logging() (*in module src.digitaltwin.utils*), 21
sites_coverage_in_catchment() (*in module src.dynamic_boundary_conditions.rainfall.rainfall_model_input*)
39
sites_voronoi_intersect_catchment() (*in module src.dynamic_boundary_conditions.rainfall.rainfall_model_input*)
38
skip_rows (*src.dynamic_boundary_conditions.rainfall.rainfall_data_from_*
attribute), 35, 36

source_table_list (*src.digitaltwin.tables.UserLogInfo attribute*), 15

spatial_uniform_rain_input() (*in module src.dynamic_boundary_conditions.rainfall.rainfall_model_input*), 40

spatial_varying_rain_input() (*in module src.dynamic_boundary_conditions.rainfall.rainfall_model_input*), 41

split_slr_measurementname_column() (*in module src.dynamic_boundary_conditions.tide.tide_slr_combine*), 93

src
    module, 1

src.app
    module, 107

src.config
    module, 110

src.datacube_data
    module, 111

src.digitaltwin
    module, 1

src.digitaltwin.data_to_db
    module, 1

src.digitaltwin.get_data_using_geoapis
    module, 6

src.digitaltwin.instructions_records_to_db
    module, 8

src.digitaltwin.run
    module, 10

src.digitaltwin.setup_environment
    module, 11

src.digitaltwin.tables
    module, 13

src.digitaltwin.utils
    module, 19

src.dynamic_boundary_conditions
    module, 22

src.dynamic_boundary_conditions.rainfall
    module, 22

src.dynamic_boundary_conditions.rainfall.hirds_rainfall_data_from_db
    module, 22

src.dynamic_boundary_conditions.rainfall.hirds_rainfall_data_to_db
    module, 25

src.dynamic_boundary_conditions.rainfall.hyetograph
    module, 27

src.dynamic_boundary_conditions.rainfall.main_rainfall
    module, 33

src.dynamic_boundary_conditions.rainfall.rainfall_data_from_hirds
    module, 34

src.dynamic_boundary_conditions.rainfall.rainfall_enum
    module, 37

src.dynamic_boundary_conditions.rainfall.rainfall_model_input
    module, 38

src.dynamic_boundary_conditions.rainfall.rainfall_sites
    module, 42

src.dynamic_boundary_conditions.rainfall.thiessen_polygons
    module, 43

src.dynamic_boundary_conditions.river
    module, 46

src.dynamic_boundary_conditions.river.align_rec1_osm
    module, 46

src.dynamic_boundary_conditions.river.hydrograph
    module, 51

src.dynamic_boundary_conditions.river.main_river
    module, 54

src.dynamic_boundary_conditions.river.osm_waterways
    module, 56

src.dynamic_boundary_conditions.river.river_data_to_from_d
    module, 57

src.dynamic_boundary_conditions.river.river_enum
    module, 59

src.dynamic_boundary_conditions.river.river_inflows
    module, 60

src.dynamic_boundary_conditions.river.river_model_input
    module, 62

src.dynamic_boundary_conditions.river.river_network_for_ac
    module, 63

src.dynamic_boundary_conditions.river.river_network_to_fro
    module, 69

src.dynamic_boundary_conditions.tide
    module, 74

src.dynamic_boundary_conditions.tide.main_tide_slr
    module, 74

src.dynamic_boundary_conditions.tide.sea_level_rise_data
    module, 76

src.dynamic_boundary_conditions.tide.tide_data_from_niwa
    module, 78

src.dynamic_boundary_conditions.tide.tide_enum
    module, 88

src.dynamic_boundary_conditions.tide.tide_query_location
    module, 89

src.dynamic_boundary_conditions.tide.tide_slr_combine
    module, 93

src.dynamic_boundary_conditions.tide.tide_slr_model_input
    module, 97

src.flood_model
    module, 98

src.flood_model.bg_flood_model
    module, 98

src.flood_model.flooded_buildings
    module, 103

src.flood_model.serve_model
    module, 105

src.run_all
    module, 111

src.tasks
    module, 112

store_flooded_buildings_in_database() (*in module src.flood_model.flooded_buildings*), 103
store_geospatial_layers_data_to_db() (*in module src.digitaltwin.data_to_db*), 6
store_instructions_records_to_db() (*in module src.digitaltwin.instructions_records_to_db*), 10
store_model_output_metadata_to_db() (*in module src.flood_model.bg_flood_model*), 99
store_rec1_data_to_db() (*in module src.dynamic_boundary_conditions.river.river_data_to_from_db*), 58
store_rec1_network_to_db() (*in module src.dynamic_boundary_conditions.river.river_network_to_from_db*), 72
store_slr_data_to_db() (*in module src.dynamic_boundary_conditions.tide.sea_level_rise_data*), 77

T

T (*in module src.config*), 110
table_name (*src.digitaltwin.tables.GeospatialLayers attribute*), 14
thiessen_polygons_calculator() (*in module src.dynamic_boundary_conditions.rainfall.thiessen_polygons*), 44
thiessen_polygons_from_db() (*in module src.dynamic_boundary_conditions.rainfall.thiessen_polygons*), 45
thiessen_polygons_to_db() (*in module src.dynamic_boundary_conditions.rainfall.thiessen_polygons*), 44
TIDE_API_URL_DATA (*in module src.dynamic_boundary_conditions.tide.tide_data_from_niwa*), 79
TIDE_API_URL_DATA_CSV (*in module src.dynamic_boundary_conditions.tide.tide_data_from_niwa*), 79
time_period (*src.dynamic_boundary_conditions.rainfall.rainfall_data_from_hirds.BlockStructure attribute*), 36
transform_data_for_selected_method() (*in module src.dynamic_boundary_conditions.rainfall.hyetograph*), 30

U

UNIFORM (*src.dynamic_boundary_conditions.rainfall.rainfall_enum.RainInputType attribute*), 37, 38
unique_column_name (*src.digitaltwin.tables.GeospatialLayers attribute*), 14
unique_id (*src.digitaltwin.tables.BGFloodModelOutput attribute*), 17, 18
unique_id (*src.digitaltwin.tables.BuildingFloodStatus attribute*), 18
unique_id (*src.digitaltwin.tables.GeospatialLayers attribute*), 14
unique_id (*src.digitaltwin.tables.UserLogInfo attribute*), 15
upload_gtiff_to_store() (*in module src.flood_model.serve_model*), 106
UPPER (*src.dynamic_boundary_conditions.river.river_enum.BoundType attribute*), 60
url (*src.digitaltwin.tables.GeospatialLayers attribute*), 14
user_log_info_to_db() (*in module src.digitaltwin.data_to_db*), 6
UserLogInfo (*class in src.digitaltwin.tables*), 15

V

valid_coordinates() (*in module src.app*), 110
validate_instruction_fields() (*in module src.digitaltwin.instructions_records_to_db*), 9
validate_url_reachability() (*in module src.digitaltwin.instructions_records_to_db*), 8
VARYING (*src.dynamic_boundary_conditions.rainfall.rainfall_enum.RainInputType attribute*), 37, 38

W

WARNING (*src.digitaltwin.utils.LogLevel attribute*), 20
wkt (*in module src.flood_model.flooded_buildings*), 105
wkt_to_gdf() (*in module src.tasks*), 115

X

x (*in module src.tasks*), 115