Juliane Mai, Ph.D.
Technical Consulting

– Documentation –

Improving the characterization of the flood-producing precipitation events in the Historic Flood Event (HFE) database by using the CaPA reanalysis/analysis

Contents

1. **Request, read, and plot data from GeoMet** 2
2. **Request, read, and plot data from CaSPAr** 6
3. **Request and read data from HFE database** 10
4. **Bilinear interpolation** 13
5. **Determine bounding box to process** 16
6. **Determine dates to process** 18
7. **Identify precipitation event** 20
8. **Analyze occurrence** 22
9. **Analyze event** 25

1. Request, read, and plot data from GeoMet

The assumption is that we want to retrieve CaPA data (RDPA; 24h total precipitation) from GeoMet for Aug 9, 2018 around Montreal ($\text{lat} = 45.5019°\text{N}$, $\text{lon} = -73.5674°\text{E}$).

The URL to request is the following: `https://api.weather.gc.ca/collections/weather:rdpa:10km:24f/coverage?f=json&datetime=2022-08-24T12Z&CRS=EPSG:4326&bbox=-74,45,-73,46`

This returns the following JSON string (not geo-referenced):


```
{"type": "Coverage", "domain": {"type": "Domain", "domainType":
"Grid", "axes": {"x": {"start": -74.0, "stop": -73.0, "num": 15},
"y": {"start": 46.0, "stop": 45.0, "num": 16}}, "referencing":
[{"coordinates": ["x", "y"], "system": {"type": "GeographicCRS", "id":
"http://www.opengis.net/def/crs/OGC/1.3/CRS84"}}]}, "parameters": {"APCP":
{"type": "Parameter", "description": "Total precipitation [kg/(m^2)]", "unit":
{"symbol": "[kg/(m^2)]"}, "observedProperty": {"id": "0-SFC", "label":
{"en": "Total precipitation [kg/(m^2)]"}}}}, "ranges": {"APCP": {"type":
"NdArray", "dataType": "float", "axisNames": ["y", "x"], "shape": [16, 15],
"values": [null, null, null, null, null, null, null, null, null, null, null,
null, null, null, null, null, null, null, null, null, 9.300250053405762,
12.344250679016113, null, null, null, null, null, null, null, null, null,
null, null, null, 5.607500076293945, 9.330750465393066, 13.382000923156738,
16.639751434326172, null, null, null, null, null, null, ... , null, null, null,
null, null, null, null, 10.574250221252441, 11.665250778198242, 11.032000541687012,
null, null, null, null, null, null, null, null, null, null, null, null,
9.582500457763672, 10.17750072479248, null, null, null, null, null, null, null,
null, null, null, null, null, null, null, null, null, null, null, null, null,
null]}}}
```


This corresponds to the following 2D view of the data (not geo-referenced):

-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
-	-	-	-	-	-	9	12	-	-	-	-	-	-	-
-	-	-	-	-	6	9	13	17	-	-	-	-	-	-
-	-	-	-	4	6	9	13	17	21	-	-	-	-	-
-	-	3	3	4	6	11	16	19	22	-	-	-	-	-
-	4	3	3	3	6	13	18	22	25	23	-	-	-	-
-	5	4	3	3	5	12	19	23	28	26	28	-	-	-
-	-	5	3	2	4	12	18	23	26	27	32	38	-	-
-	-	-	3	2	4	10	16	20	22	26	35	38	27	-
-	-	-	-	4	5	8	16	18	20	27	36	35	24	-
-	-	-	-	6	6	7	14	16	20	31	41	35	20	-
-	-	-	-	-	5	8	11	13	16	27	37	33	-	-
-	-	-	-	-	-	10	10	12	11	19	26	-	-	-
-	-	-	-	-	-	-	11	12	11	-	-	-	-	-
-	-	-	-	-	-	-	10	10	-	-	-	-	-	-
-	-	-	-	-	-	-	-	-	-	-	-	-	-	-

To retrieve the corresponding PNG file for this data use (see Fig. 1a) (not geo-referenced):

```
https://geo.weather.gc.ca/geomet?SERVICE=WMS&VERSION=1.3.0&REQUEST=GetMap&LAYERS=
RDPA.24F_PR&STYLES=RDPA-WXO&CRS=EPSG:4326&BBOX=45,-74,46,-73&WIDTH=400&HEIGHT=400&
FORMAT=image/png&TIME=2022-08-24T12:00:00Z&DIM_REFERENCE_TIME=2022-08-24T12:00:
00Z
```

To retrieve the legend use (see Fig. 1a):

```
https://geo.weather.gc.ca/geomet?SERVICE=WMS&VERSION=1.3.0&REQUEST=
GetLegendGraphic&LAYERS=RDPA.24F_PR&STYLES=RDPA-WXO&CRS=EPSG:4326&BBOX=45,-74,46,
-73&WIDTH=400&HEIGHT=400&FORMAT=image/png&TIME=2022-08-24T12:00:00Z&DIM_REFERENCE_
TIME=2022-08-24T12:00:00Z
```

To actually retrieve the geo-referenced data used for those plots, the GRIB2 file needs to be retrieved using the following command:

```
https://api.weather.gc.ca/collections/weather:rdpa:10km:24f/coverage?f=GRIB&
datetime=2022-08-24T12Z&CRS=EPSG:4326&bbox=-74,45,-73,46
```

This seems to be the only possibility to obtain geo-referenced data. The library developed here now provides three functions to obtain and plot the data. The results are shown in Fig. 1b.

Step A1: Request data using the Geomet API (`https://api.weather.gc.ca`).

```
# see module for detailed documentation and example
from a1_request_geomet_grib2 import request_geomet_grib2

files_geomet = request_geomet_grib2(
      # Mandatory arguments:
      product=product,      # name of product, e.g., "rdpa:10km:24f"
      date=date,            # datetime object specifying date (or list of dates)
      bbox=bbox,            # dictionary specifying bounding box
      # Optional arguments:
      crs=crs,              # coordinate reference system, e.g., "EPSG:4326"
      filename=filename,    # base filename of output file (can include path but
                            # no file extension); date will be added to filename
      overwrite=overwrite,  # If true, file will be downloaded
                            # again overwriting existing file
      silent=silent,        # True for no printing on screen
      )
```

The return variable `files_geomet` is a dictionary where the keys are the provides date(s) and the values are lists of filenames that contain this time step. If the file already existed, it will not be overwritten unless `overwrite` is set to `True`. The filename will be returned nonetheless. In case no file is found for a date, the list will be empty. If the list contains multiple files, the time step was found in several files. If no file was found for any date, an error is raised since this indicates that something in the request string to Geomet is likely wrong.

Step B1: Read data from files requested.

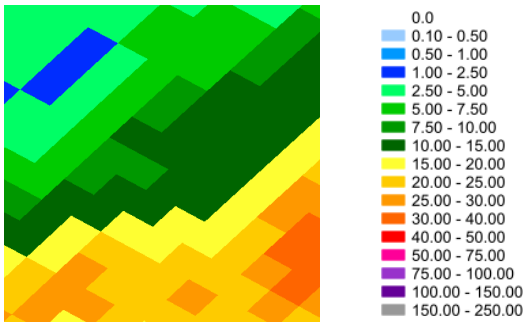
```
# see module for detailed documentation and example
from b1_read_geomet_grib2 import read_geomet_grib2

data_geomet = read_geomet_grib2(
      # Mandatory arguments:
      filenames=filenames,          # dictionary specifying filename per time
                                    # step, e.g.,
                                    # { date_1:  [ filename_1, filename_2 ],
                                    #   date_2:  [ filename_2 ],
                                    #   date_3:  [ ], ...  }
                                    # ↷ output of "request_geomet_grib2()"
      # Optional arguments:
      lintransform=lintransform,    # dictionary to specify linear transform of
                                    # data, e.g., to allow for unit conversions
      silent=silent,                # True for no printing on screen
      )
```

The returned variable `data_geomet` is a dictionary that will contain the attributes `time`, `lat`, `lon`, and `var`. The latter will be 3-dimensional (time, lat, lon). The latitude and longitudes of the files are checked for consistency. The time steps returned are only the time steps where data was available.

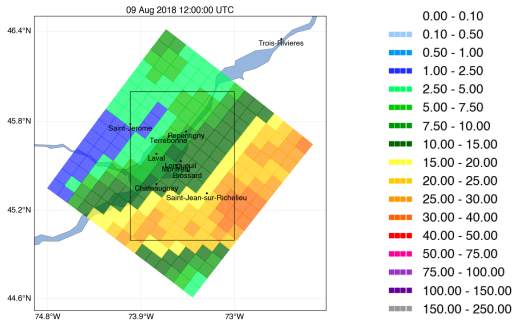
Step Cx: Plot data.

```
# see module for detailed documentation and example
from cx_plot_data import plot_data

plot_geomet = plot_data(
      # Mandatory arguments:
      var=var,                      # 2D/3D array of values for variable
      lat=lat,                      # 2D array of latitudes
      lon=lon,                      # 2D array of longitudes
      date=date,                    # date or list of dates as datetime objects
      # Optional arguments:
      png=png,                      # True if PNG file(s) should be created
      pngsum=pngsum,                # True if PNG file (map) with sum over all
                                    # time steps should be created
      gif=gif,                      # True if GIF should be created
      legend=legend,                # True if PNG of legend should be created
      cities=cities,                # True to display cities on maps
      bbox=bbox,                    # Bounding box will be plotted if provided
      basefilename=basefilename,    # String specifying basename of files created
      overwrite=overwrite,          # True to overwrite existing plots
      label=label,                  # Overwrite default label which is date of
                                    # time step plotted by this label
                                    # (same label added to all plots produced)
      language=language,            # language version used for plot;
                                    # e.g., 'en_CA' or 'fr_CA'
      locations=locations,          # locations that are to be added
                                    # as labels to plot; dictionary with keys
                                    # "lat", "lon", and "name"
      silent=silent,                # True for no printing on screen
      )
```

The returned variable `plot_geomet` is a dictionary that will contain the attributes `png`, `gif`, and `legend`. Each of them are assigned list of the according files created. If no file was created (e.g., no legend), the respective list will be empty.



(a) Raw PNG and legend for RDPA-24h data retrieved using `https://geo.weather.gc.ca` (only displayed for reference)



(b) PNG and legend produced using NRCan-HFE library based on GRIB2 data (RDPA-24h) retrieved using `https://api.weather.gc.ca/`

Figure 1: RDPA (24h total precipitation) on Aug 9, 2018 (noon UTC) around Montreal

2. Request, read, and plot data from CaSPAr

Step A2: Request data using CaSPAr (`https://caspar-data.ca/caspar`).

The following function is just a dummy to provide some information on how to retrieve data from CaSPAr manually. Fig. 2 shows the setting used to retrieve RDRS-v2.1 data from CaSPAr. The shapefile containing the domain of the HFE database is provided under `data/caspar/RDRS_v21_hfe_outline.zip` (see shape as blue polygon in Fig. 4). Some example data are made available (see `src/test-data/*.nc`). See more information on how to create a CaSPAr account, request and dwonload data under `https://github.com/julemai/CaSPAr/wiki/How-to-get-started-and-download-your-first-data`.

This function basically only checks if the files for the product specified and the (list of) date(s) provided are available under the specified foldername.

```
# see module for detailed documentation and example
from a2_request_caspar_nc import request_caspar_nc

files_caspar = request_caspar_nc(
       # Mandatory arguments:
       product=product,        # name of product in CaSPAr (e.g., 'RDRS-v2.1')
       variable=variable,      # name of variable in CaSPAr
                               # (e.g., 'RDRS_v2.1_A_PR0_SFC')
       date=date,              # datetime object specifying date
                               # (can be list of dates)
       # Optional arguments:
       foldername=foldername, #
       silent=silent,          # True for no printing on screen
       )
```

The return variable `files_caspar` is a dictionary where the keys are the provides date(s) and the values are lists of filenames that contain this time step. In case no file is found for a date, the list will be empty. If the list contains multiple files, the time step was found in several files. If no file was found for any date, an error is raised since this indicates that the files have not been requested and downloaded from CaSPAr yet or the `foldername` is not the location where those files have been downloaded to.

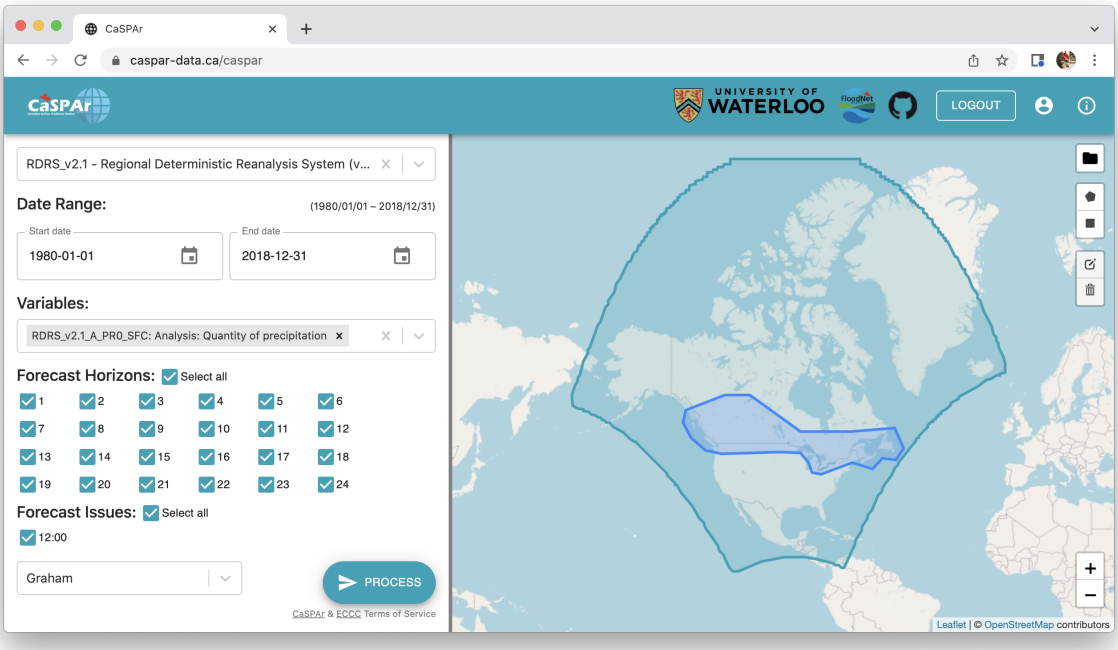


Figure 2: Screenshot of CaSPAr (`https://caspar-data.ca/caspar`) frontend displaying setting to request RDRS-v2.1 for the HFE domain. The shapefile used to define the domain (blue highlighted area) within the data available for RDRS-v2.1 (turquoises highlighted area) is available under `data/caspar/RDRS_v21_hfe_outline.zip` and has been used via file upload (top button in right top corner of the map).

Step B2: Read data from CaSPAr files.

```
# see module for detailed documentation and example
from b2_read_caspar_nc import read_caspar_nc

data_caspar = read_caspar_nc(
      # Mandatory arguments:
      variable=variable,           # name of variable in CaSPAr
      filenames=filenames,         # dictionary specifying filename per time
                                   # step, e.g.,
                                   # { date_1:  [ filename_1, filename_2 ],
                                   #   date_2:  [ filename_2 ],
                                   #   date_3:  [ ], ...  }
                                   # ↷ output of "request_caspar_nc()"
      # Optional arguments:
      bbox=bbox,                   # dictionary specifying bounding box
      lintransform=lintransform,   # dictionary to specify linear transform of
                                   # data, e.g., to allow for unit conversions
      silent=silent,               # True for no printing on screen
      )
```

The returned variable `data_caspar` is a dictionary that will contain the attributes `time`, `lat`, `lon`, and `var`. The latter will be 3-dimensional (time, lat, lon). The latitude and longitudes of the files are checked for consistency. The time steps returned are only the time steps where data was available.

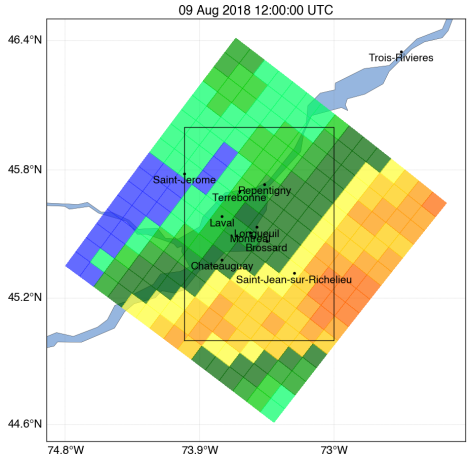
Step Cx: Plot data.

Since the data `data_caspar` have the same structure as the data read from Geomet (i.e., `data_geomet`) the plotting of the data is now exactly the same. See **Step Cx** above.

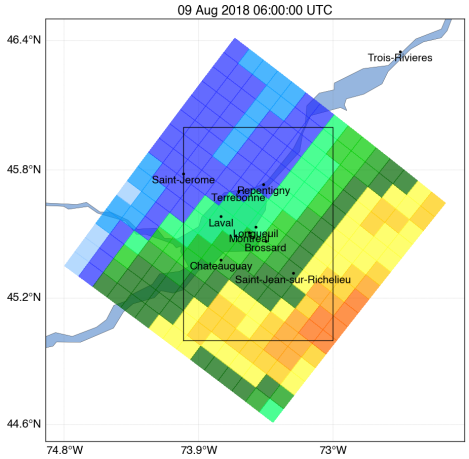
```
# see module for detailed documentation and example
from cx_plot_data import plot_data

plot_caspar = plot_data(
      # Mandatory arguments:
      var=data_caspar["var"],       # 2D/3D array of values for variable
      lat=data_caspar["lat"],       # 2D array of latitudes
      lon=data_caspar["lon"],       # 2D array of longitudes
      ...                           # see above "Step Cx" for more arguments
      )
```

The returned variable `plot_caspar` is a dictionary that will contain the attributes `png`, `gif`, and `legend`. Each of them are assigned list of the according files created. If no file was created (e.g., no legend), the respective list will be empty.

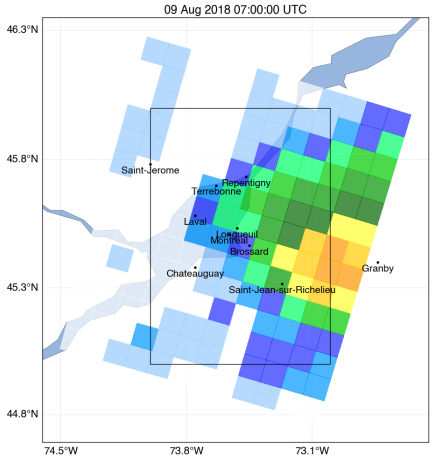
Fig. 3c shows the result of a plot of precipitation (`RDRS-v2.1_A_PR0_SFC`) in the RDRS-v2.1 product for a domain around Montreal on Aug 9, 2018 at 7:00 am (UTC). For comparison, the other two panels in this figure (i.e., Fig. 3a and Fig. 3b) show the results of plotting data requested from Geomet for the same day but as 24-h and 6-h precipitation accumulations of the RDPA product.



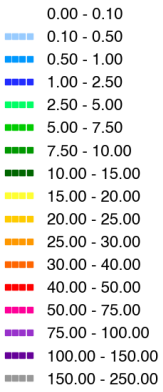
(a) PNG produced using NRCan-HFE library based on GRIB2 data (RDPA-24h) retrieved using `https://api.weather.gc.ca/`



(b) PNG produced using NRCan-HFE library based on GRIB2 data (RDPA-6h) retrieved using `https://api.weather.gc.ca/`



(c) PNG produced using NRCan-HFE library based on NetCDF data (RDRS v2.1-1h) retrieved using `https://caspar-data.ca`



(d) Legend for all plots generated.

Figure 3: RDPA (24h and 6h total precipitation) from Geomet and RDRS_v2.1_A_PR0_SFC (1h accumulations) from CaSPAr for data available for Aug 9, 2018. (a) RDPA-24h is accumulated precipitation for noon Aug 8, 2018 (UTC) until noon Aug 9, 2018 (UTC), (b) RDPA-6h is accumulated precipitation for midnight Aug 9, 2018 (UTC) until 6am Aug 9, 2018 (UTC) while (c) RDRS-v2.1 is precipitation between Aug 9, 2018 6:00 am and 7:00 am (UTC). The same bounding box around Montreal has been requested from the three products. (d) The legend is the same for all plots. The panel in Fig. 3a is the same as panel in Fig. 1b.

3. Request and read data from HFE database

Step A3: Request HFE database.

The database is available here. It has a web map service (WMS) that can be used to retrieve a GeoJSON containing information about flood events.

However, the database in JSON format was provided by Philippe Aussant on Aug 29, 2022. The data are available under `data/hfe/`.

The following data were provided:

`historical_flood.json ...` This layer contains individual flood occurrences (`points`). The points in this layer have a unique ID (`uuid`) and a secondary key to their historical_flood_events (`event_id`). We need precipitation for these points. You can write the precipitation for each in the `rainfall_mm` attribute. The maps produced in 2.4.1 of SOW are for these points.

`historical_flood_event.json ...` This layer contains the same flood occurrences as the previous layer but now they are grouped into events (`multipoints`). A flood event contains one or more occurrence. The unique id of this layer is `event_id`. The maps produced in 2.4.2 and 2.4.3 of SOW are for these multipoints.

The two files above were used to create some test data under `src/test-data`. The first few features of each file was taken to create the sample files.

The following function is currently only checking if the files exist at the indicated location, i.e. checking for the existence of the two files <jsonfilebase>.json and <jsonfilebase>_event.json. It is highly recommended that this function is adjusted by NRCan to replicate the formatting of the original data found on the website into these JSON files.

```
# see module for detailed documentation and example
from a3_request_hfe_json import request_hfe_json

files_hfe = request_hfe_json(
       # Mandatory arguments:
       filename=filename,            # filename of HFE database (currently not used)
       jsonfilebase=jsonfilebase,    # basename (w/o file ending) of JSON files that
                                     # will be produced (currently just checked for
                                     # existence).  Can include path.
       # Optional arguments:
       silent=silent,                # True for no printing on screen
)
```

The returned variable `files_hfe` is a dictionary with one key (`json`) containing the list of JSON files created (currently just found; not created).

Step B3: Read data from HFE files.

```
# see module for detailed documentation and example
from b3_read_hfe_json import read_hfe_json

data_hfe = read_hfe_json(
      # Mandatory arguments:
      filename=filename,              # Name of JSON file of HFE database to read
      # Optional arguments:
      filtering=filtering,            # if True, content of JSON will be checked
                                      # to be consistent and filtered dataset
                                      # returned; checks applied are listed below
      polygon=polygon,                # list of coordinates (lon,lat) of polygon;
                                      # if specified and filtering is
                                      # True, coordinates in JSON are located
                                      # inside this polygon
      return_filtered=return_filtered, # if True, a dictionary will be returned
                                      # listing features that were discarded
                                      # and reason they were filtered;
                                      # filtered = {reason:  list of feature IDs}
      silent=silent,                  # True for no printing on screen
      )
```

The returned variable `data_hfe` is a dictionary that will contain the attributes `data` and `filtered`. The latter will be an empty list if `return_filtered` is False. Filtering will only be applied if `filtering` is set to True which is highly recommended. Otherwise, the `data` are all features found in the JSON file and there is no guarantee that processing using these data will be successful.

The checks currently applied when `filtering` is set to True are:

Check #1. Required keys are available.

Check #2. Start and end dates specified are in format YYYY-MM-DD (not just MM or YYYY, etc.).

Check #3. Start date is 1980-01-01 or later. This is the date RDRS-v2.1 becomes available. For events before this date, no data could be retrieved.

Check #4. Start date is not before end date (given that end date is specified).

Check #5. Points and multipoints are within the specified `polygon` if specified.

Table 1 shows the number of features available in the files provided. Different levels of checks are applied, i.e., (i) no checking, (ii) checking everything but not location of coordinates within a polygon (check #1 to #3 but not #4), and (iii) all checks including the test that the features are located within the specified polygon (checks #1 to #4). The polygon is set to the extent used to request the CaSPAr data (see `data/caspar/RDRS_v21_hfe_outline.zip`). Fig. 4 shows the location of all events and occurrences as well as the polygon specifying the domain of interest for CaSPAr.

	no check
	`filtering=False`
		check 1-4
		`filtering=True`
		`polygon=None`
			check 1-5
			`filtering=True`
			`polygon=polygon`

historical_flood.json	1999	1949[a]	1949[c]
historical_flood_event.json	376	363[b]	363[c]

[a] ... in total, n=50 features are flagged;
↷ n=11 due to check #2 (only year but not date specified)
↷ n=39 due to check #4 (start-date after end-date)
[b] ... in total, n=14 features are flagged;
↷ n=9 due to check #2 (only year but not date specified)
↷ n=5 due to check #4 (start-date after end-date)
[c] ... in total, n=0 features are flagged due to check #4
↷ i.e., all features are inside polygon
↷ i.e., polygon used to request data from CaSPAr was large enough to cover all events

Table 1: Features available in the HFE JSON files provided. Different degrees of filtering show how many features are not conform with requirements to extract precipitation data.

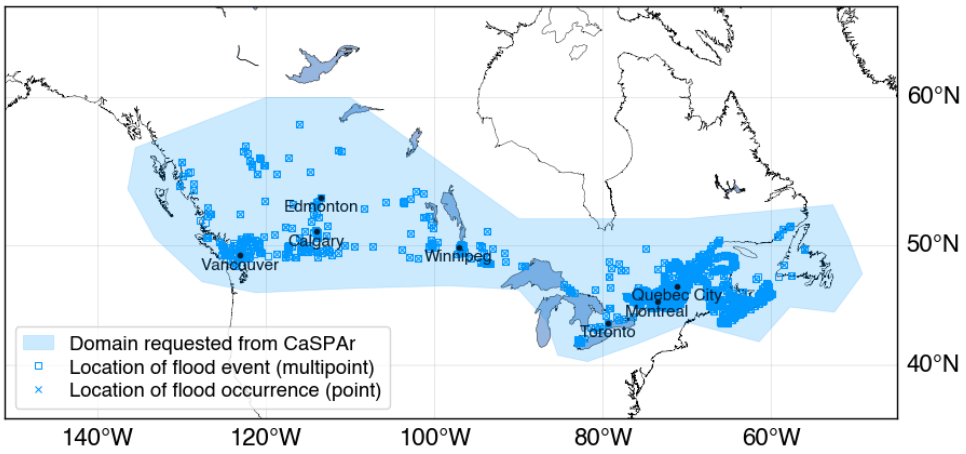


Figure 4: Locations specified in HFE database as multi-point events (squares) or single-point occurrences (crosses) as well as the domain (blue polygon) used to request RDRS-v2.1 from CaSPAr.

4. Bilinear interpolation

Step Dx: a – Interpolate data to specific location.

This method is used to extract time series for points defining the flooding event (located within the bounding box used to request data). The raster is interpolated at the flooding event points using bilinear interpolation. The base of this function is `scipy.interpolate.LinearNDInterpolator` (function `scipy.interpolate.interp2d` was tested but turned out to be too unstable). If a location is requested that is not available in data, an error is raised. It is recommended to set `post_process` to True which will round negative interpolated (precipitation) data to zero unless they are "too negative". The latter will raise an error indicating that the interpolation does not seem to work properly.

```
# see module for detailed documentation and example
from dx_interpolate_data import interpolate_data

interpolated_points = interpolate_data(
      # Mandatory arguments:
      var=var,                  # 2D/3D array of values for variable
      lat=lat,                  # 2D array of latitudes
      lon=lon,                  # 2D array of longitudes
      locations=locations,      # dict containing "lat" as list of latitudes
                                # and "lon" as list of longitudes for points
                                # where field(s) of "var" will be interpolated
      # Optional arguments:
      bbox=bbox,                # if provided locations will be checked to fall
                                # within; in any case points that are not covered
                                # by the data will result in an error message of
                                # the function
      return_tmp=return_tmp,    # if True, data and location of the points used
                                # for interpolation will be returned as dictionary
      post_process=post_process, # if True, negative data will be set to zero;
                                # unless they are smaller than -0.1 which will
                                # raise an error
      silent=silent,            # True for no printing on screen
      )
```

The returned variable `interpolated_points` is a dictionary that will contain the attributes `lat`, `lon`, and `var`. The latter will be 2-dimensional with the size of the first dimension being the number of time steps and the size of the second dimension being the number of locations.

Fig. 5 is showing the result of a bilinear interpolation of two locations over the course of 9 days.

Step Dx: b – Plot interpolated data.

The module `dx_interpolate_data` also contains a convenience function to plot the interpolated data using `plot_interpolated()` similar to what is shown in the left two panels of Fig. 5. An example output of the function with highlighting some time steps is shown in Fig. 6.

```
# see module for detailed documentation and example
from dx_interpolate_data import plot_interpolated

plots_interpolated = plot_interpolated(
      # Mandatory arguments:
      locations=locations,                    # dict containing "lat" as list of
                                              # latitudes and "lon" as list of
                                              # longitudes for points
      dates=dates,                            # list of dates "data" are provided
      data=data,                              # dictionary of data containing
                                              # "lat", "lon", and "var" (return of
                                              # "interpolate_data()")
      start_date_buffer=start_date_buffer,    # start date incl.  buffer
      end_date_buffer=end_date_buffer,        # end date incl.  buffer
      pngfile=pngfile,                        # name of PNG file to be created
      # Optional arguments:
      highlight_dates_idx=highlight_dates_idx, # indexes of timesteps to highlight
                                              # in plot
      start_date=start_date,                  # start date listed in HFE database
                                              # (only used to plot vertical line)
      end_date=end_date,                      # end date listed in HFE database
                                              # (only used to plot vertical line)
      label=label,                            # custom label to add below plot
      language=language,                      # language version used for plot;
                                              # e.g., 'en_CA' or 'fr_CA'
      silent=silent,                          # True for no printing on screen
      )
```

The returned variable `plots_interpolated` is a dictionary that will contain the key `"png"` and its value is a list of the PNG file created.

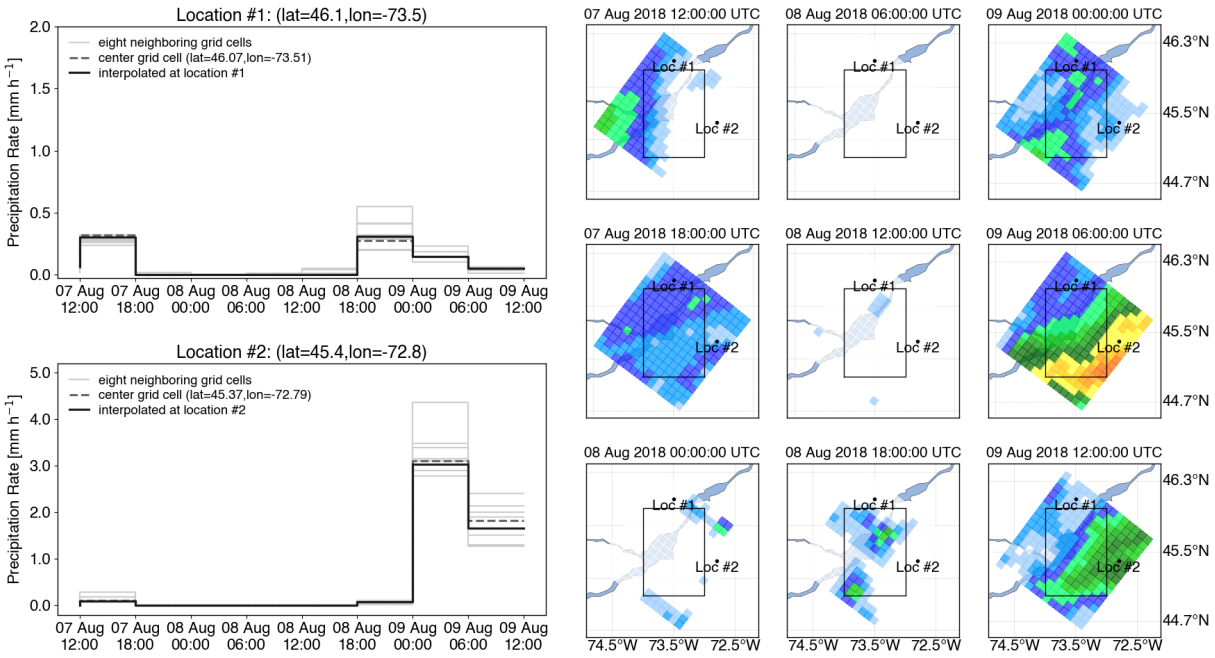


Figure 5: RDPA (6h total precipitation) around Montreal requested from Geomet (`https://api.weather.gc.ca/`) is displayed for nine time steps (Aug 7, 2018 12:00 until Aug 9, 2018 12:00). The nine maps on the right show the spatial distribution of precipitation within the requested bounding box (outline of black rectangle) and beyond. The legend is the same for the nine plots showing maps (see Fig. 3d). Two locations have been randomly selected and are displayed on the maps as well (black dots). The left two panels show the precipitation rate over time for the center cell that contains the locations (dashed black line) as well as the bilinearly interpolated precipitation rate at the exact location (solid black line). For reference also the other 8 neighboring cells used for the bilinear interpolation are shown (gray lines).

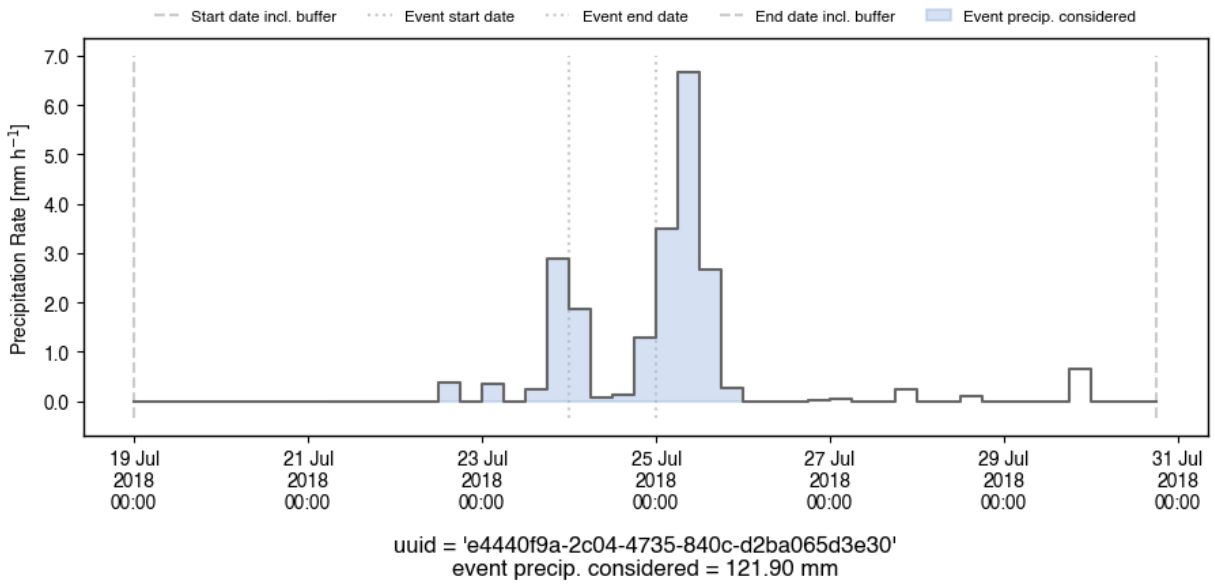


Figure 6: Output of function `plot_interpolated()` in module `dx_interpolate_data`. Some timesteps of the analysed time period from July 19, 2018 to July 31, 2018 have been specified to be highlighted ("`highlight_dates_idx`"; highlighted in blue color). The sum of those datapoints is 121.90 [mm] which had been passed as a custom label ("`label`") including the UUID of the feature analyzed.

5. Determine bounding box to process

Step Ex: Determine bounding box of single-point occurrences or multi-point events.

This function is to determine the bounding box `bbox` to request data from Geomet and read data retrieved from CaSPAr (see steps A1 and B2, respectively; function argument `bbox` for both functions). The function is able to handle lists of latitudes and longitudes as well as features directly taken from the HFE database. The features can be single-point occurrences or multi-point events. The bounding box is by default the bounding box around all locations provided with an additional buffer `bbox_buffer` of 0.5 degrees in all directions. The buffer can be set to any non-negative value.

Note: The approach might be adjusted/refined once we start processing all flood occurrences and events.

```
# see module for detailed documentation and example
from ex_determine_bbox import determine_bbox

bbox = determine_bbox(
       # Mandatory arguments:
       lat=lat,                    # list of latitude of location(s) that are
                                   # required to be within the bounding box;
                                   # can be scalar if bbox is determined for
                                   # only one location
                                   # either lat/lon or feature needs to be specified
       lon=lon,                    # list of longitude of location(s) that are
                                   # required to be within the bounding box;
                                   # can be scalar if bbox is determined for
                                   # only one location
                                   # either lat/lon or feature needs to be specified
       feature=feature,            # dict containing feature as obtained from
                                   # HFE database;
                                   # either lat/lon or feature needs to be specified
       # Optional arguments:
       bbox_buffer=bbox_buffer,    # minimum distance between locations provided and
                                   # outline of bounding box (in degrees); needs to
                                   # be non-negative; unit:  degree; default 0.5
       silent=silent,              # True for no printing on screen
       )
```

The return value the bounding box in the format of a dictionary which is conform with what is needed for requesting data from Geomet (step A1; `request_geomet_grib2`), read data retrieved from CaSPAr (step B2; `read_caspar_nc`), plotting data (step Cx; `plot_data`), and to interpolate data (step Dx; `interpolate_data`).

Fig. 7 shows the bounding box derived for a single-point occurrence and a multi-point event using different settings for the bounding box buffer `bbox_buffer`.

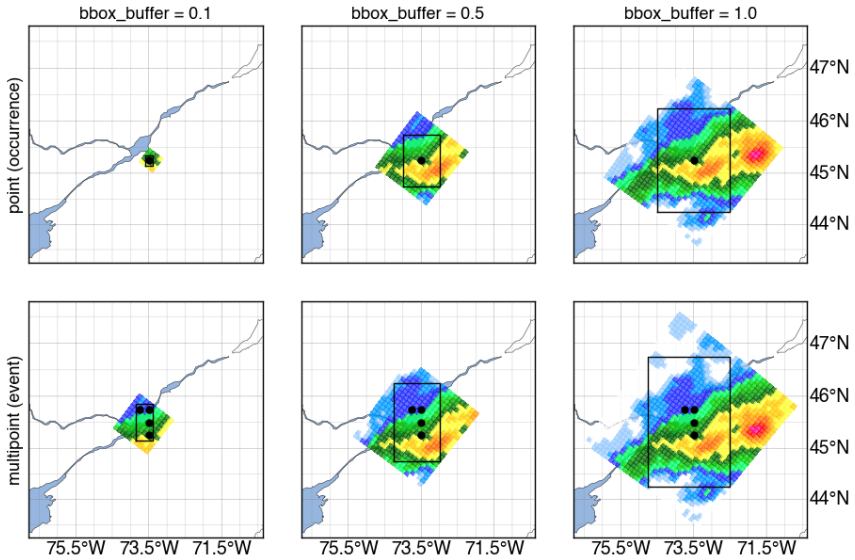


Figure 7: Bounding box derived using `determine_bbox` using different settings for the buffer between locations and bounding box, i.e., buffer of 0.1 (first column of panels), buffer of 0.5 (second column of panels), and buffer of 1.0 degrees (third column of panels). The results are shown for a single point defining flood occurrences (first row of panels) and multiple points defining flood events (second row).

6. Determine dates to process

Step Fx: Determine time steps for single-point occurrences or multi-point events.

This function is to determine all time steps to request data from Geomet and read data retrieved from CaSPAr (see steps A1 and B2, respectively; function argument `date` for both functions). The function is aware of the timesteps that are available for different products (`product`) and will return only those valid time steps. A buffer before the specified start date (function argument `start_date` or `feature` property `start_date`) and after the specified end date (function argument `end_date` or `feature` property `end_date`) is specified via `dates_buffer`. The dates are all assumed to be in UTC time. Start dates are assumed to be at 00:00 while end dates are assumed to be at 23:59.

Note: The approach might be adjusted/refined once we start processing all flood occurrences and events.

```
# see module for detailed documentation and example
from fx_determine_dates import determine_dates

dates = determine_dates(
       # Mandatory arguments:
       start_date=start_date,      # start date of event/occurr.  as datetime object;
                                   # either start_date or start_date and end_date or
                                   # feature needs to be specified
       end_date=end_date,          # end date of event/occurrence as datetime object;
                                   # if not specified but start_date is, end_date will
                                   # be set to start_date + 1 day;
                                   # either start_date or start_date and end_date or
                                   # feature needs to be specified
       feature=feature,            # dict containing feature as obtained from
                                   # HFE database; needs to contain feature property
                                   # "start_date" and "end_date";
                                   # either start_date or start_date and end_date or
                                   # feature needs to be specified
       product=product,            # name of product, e.g., "rdpa:10km:24f"
                                   # must be one of the following:
                                   # [ "RDRS_v2.1", "rdpa:10km:24f", "rdpa:10km:6f" ]
       # Optional arguments:
       dates_buffer=dates_buffer, # used to offset start_date to
                                   # start_date - dates_buffer[0] and end_date to
                                   # end_date + dates_buffer[1] in order to retrieve
                                   # precipitation data prior to and after flooding
                                   # event; needs to be non-negative; unit:  days
                                   # default [3.0,1.0]
       silent=silent,              # True for no printing on screen
       )
```

The return value is a list of datetime objects which is conform with what is needed for requesting data from Geomet (step A1; `request_geomet_grib2`), reading data retrieved from CaSPAr (step B2; `read_caspar_nc`), and plotting data (step Cx; `plot_data`). The function argument for these three functions is `date`.

The following examples are to show the dates determined using this function.

Example 1a: Dates to retrieve data from RDRS_v2.1 (hourly dataset) without buffer.

```
product = "RDRS_v2.1"
dates_buffer = [0.0,0.0]
period = [datetime.datetime(2000,5,28,3,0),datetime.datetime(2000,5,29,20,30)]

# call function
tsteps = determine_dates(start_date=period[0],end_date=period[1],
                         product=product,dates_buffer=dates_buffer)
print(tsteps)

[Out]:  [datetime.datetime(2000, 5, 28, 3, 0),
         datetime.datetime(2000, 5, 28, 4, 0), ...,
         datetime.datetime(2000, 5, 29, 20, 0) ]
```

Example 1b: Dates to retrieve data from RDRS_v2.1 (hourly dataset) with buffer.

```
product = "RDRS_v2.1"
dates_buffer = [3.0,1.0]
period = [datetime.datetime(2000,5,28,3,0),datetime.datetime(2000,5,29,20,30)]

# call function
tsteps = determine_dates(start_date=period[0],end_date=period[1],
                         product=product,dates_buffer=dates_buffer)
print(tsteps)

[Out]:  [datetime.datetime(2000, 5, 25, 3, 0),
         datetime.datetime(2000, 5, 25, 4, 0), ...,
         datetime.datetime(2000, 5, 30, 20, 0) ]
```

Example 2a: Dates to retrieve data from RDPA-6h (6-hourly dataset) without buffer.

```
product = "rdpa:10km:6f"
dates_buffer = [0.0,0.0]
period = [datetime.datetime(2000,5,28,3,0),datetime.datetime(2000,5,29,20,30)]

# call function
tsteps = determine_dates(start_date=period[0],end_date=period[1],
                         product=product,dates_buffer=dates_buffer)
print(tsteps)

[Out]:  [datetime.datetime(2000, 5, 28, 6, 0),
         datetime.datetime(2000, 5, 28, 12, 0), ...,
         datetime.datetime(2000, 5, 29, 18, 0) ]
```

7. Identify precipitation event

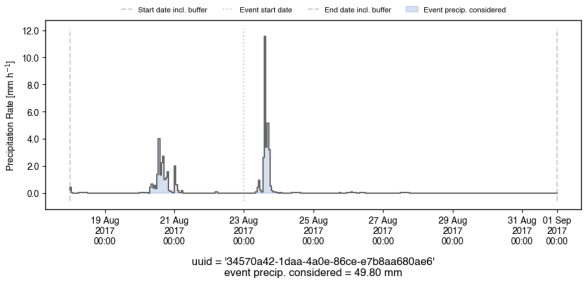
Step Gx: Identify the major precipitation event in a time series between a specified start and end date as well as shortly before and after.

It is assumed that a precipitation time series is provided ("data") at given dates ("dates") for a feature ("feature") as it is stored in the HFE database. The main event will then be identified starting at the start date given in the "feature". A rolling window of a certain length ("length_window_d") will be analyzed. If the accumulated precipitation of the current window is large enough (larger than "min_prec_window"), all time steps of the window will be added and the window moved by one timestep (earlier). This is continued until no new time step is added prior to the start date (because the cumulative precipitation is too small, i.e., event start seems to have been found). Then only single time steps (prior to first current index) are checked and added until the first time step smaller than a threshold ("min_prec") is detected. This guarantees that the entire onset of the event is added. The same approach is then repeated starting the rolling window with the current last detected index (current end of event). After no more indexes are added by using the window, single time steps larger than a threshold ("min_prec") are added.

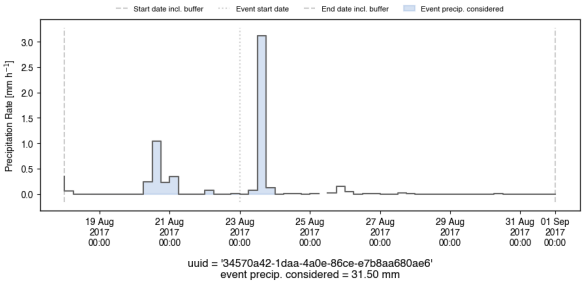
```
# see module for detailed documentation and example
from gx_identify_precipitation_event import identify_precipitation_event

time_idx = identify_precipitation_event(
      # Mandatory arguments:
      feature=feature,                  # dict containing feature as obtained
                                        # from HFE database
      product=product,                  # name of product, e.g., "rdpa:10km:24f"
                                        # must be one of the following:
                                        # [ "RDRS_v2.1", "rdpa:10km:24f",
                                        #   "rdpa:10km:6f" ]
      dates=dates,                      # list of dates "data" are provided for
      data=data,                        # dictionary of data containing "lat",
                                        # "lon", and "var" (return of
                                        # "interpolate_data()")
      # Optional arguments:
      length_window_d=length_window_d,  # Length of rolling window precipitation
                                        # is iteratively analyzed for.
                                        # Given in [days].
      min_prec_window=min_prec_window,  # Minimum precipitation accumulated over
                                        # rolling window to be considered being
                                        # added to event.
                                        # Given in [mm] within length_window_d.
      min_prec=min_prec,                # Minimal amount of precipitation to be
                                        # considered at one time step.
                                        # Given in [mm/h].
      silent=silent,                    # True for no printing on screen
      )
```

The return value "time_idx" is a list of list of indexes for time steps that should be considered as major precipitation event. The indexes can be applied to "dates" and the time axis of "data". The list of list of indexes can be helpful to be specified as "highlight_dates_idx" in plot_interpolated().

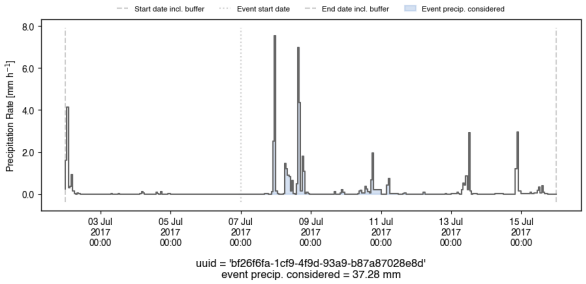
The result of three example events that occurred in 2017 comparing data retrieved from Geomet (RDPA-6h) and CaSPAr (RDRS_v2.1) are shown in Fig. 8.



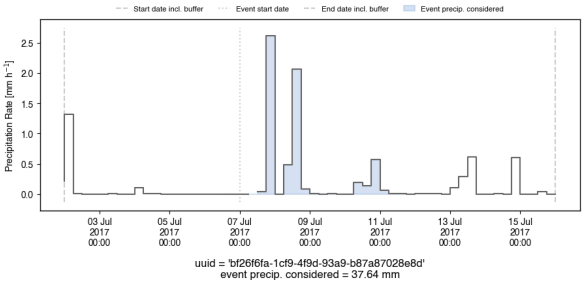
(a) UUID "34570a42-1daa-4a0e-86ce-e7b8aa680ae6" using CaSPAr data (RDRS_v2.1; 1hr accum.)



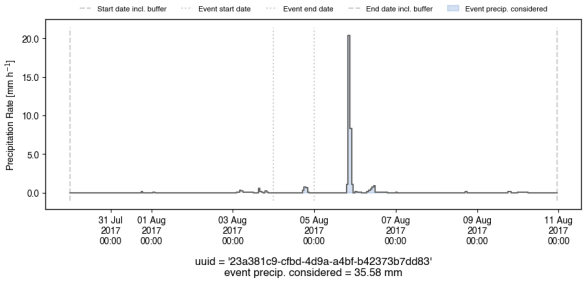
(b) UUID "34570a42-1daa-4a0e-86ce-e7b8aa680ae6" using GeoMet data (RDPA; 6hr accum.)



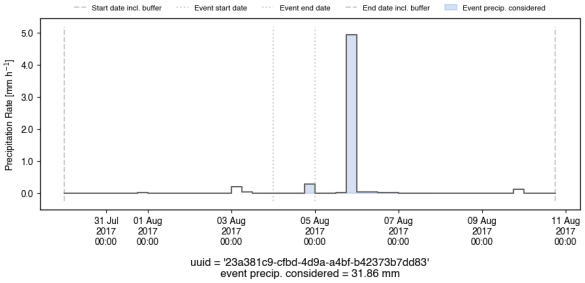
(c) UUID "bf26f6fa-1cf9-4f9d-93a9-b87a87028e8d" using CaSPAr data (RDRS_v2.1; 1hr accum.)



(d) UUID "bf26f6fa-1cf9-4f9d-93a9-b87a87028e8d" using GeoMet data (RDPA; 6hr accum.)



(e) UUID "23a381c9-cfbd-4d9a-a4bf-b42373b7dd83" using CaSPAr data (RDRS_v2.1; 1hr accum.)



(f) UUID "23a381c9-cfbd-4d9a-a4bf-b42373b7dd83" using GeoMet data (RDPA; 6hr accum.)

Figure 8: Three example flood occurrences (rows) that occurred in 2017 from HFE database (historical_flood.json). The major precipitation has been identified using CaSPAr data (left column) and GeoMet data (right column). The identified for each event (UUID) corresponds to the UUID specified in `historical_flood.json`.

8. Analyze occurrence

Workflow: Sequence of steps to analyze occurrences (single-point feature).

The script/function summarizes the workflow to analyze single-point events (occurrences) for the HFE database, i.e., `"historical_flood.json"`. The user needs to specify a feature index (or several). The other arguments are optional and set to default values. This script can be called directly from the command-line with command-line arguments or as a function within another Python script. There is no new functionality added here. This wrapper script is using only the functions described above.

A. Usage as Python function:

```
# see module for detailed documentation and example
from analyse_occurrence import analyse_occurrence

files_created = analyse_occurrence(
      # Mandatory arguments:
      ifeatures=ifeatures,          # list of indexes of feature(s) to analyze;
                                    # numbering starts with 0; these are not
                                    # feature IDs themselves but the order they
                                    # appear in "historical_flood.json"
      # Optional arguments:
      tmpdir=tmpdir,                # Name of directory to save downloaded and
                                    # produced files under; default:  "/tmp/"
      bbox_buffer=bbox_buffer,      # minimum distance between locations provided and
                                    # outline of bounding box (in degrees); needs to
                                    # be non-negative; unit:  degree; default 0.5
      dates_buffer=dates_buffer,    # used to offset start_date to
                                    # start_date - dates_buffer[0] and end_date to
                                    # end_date + dates_buffer[1] in order to retrieve
                                    # precipitation data prior to and after flooding
                                    # event; needs to be non-negative; unit:  days
                                    # default [5.0,5.0]
      silent=silent,                # True for no printing on screen
      )
```

The return value `"files_created"` is a dictionary of files created. It has the keys `"png-ts"` containing PNG files created showing time series, `"png"` with a list of PNG files showing maps, `"legend"` with a list of legends used for map PNG files, and `"json"` containing JSON files that have the summarized analysis stored in the format they will be used to augment the HFE database. It also contains a key `"gif"` which is added for consistency but should usually only be an empty list as no GIFs are created.

B. Usage as commandline function:

```
# command-line terminal; '\' at the end of lines below indicate line breaks
# for readability; remove before copying to terminal

# List of feature indexes to analyse
features='2,12,22'

# Call of Python workflow
python analyse_occurrence.py --ifeatures "${features}" \
    --tmpdir "/project/6070465/julemai/nrcan-hfe/data/output/" \
    --bbox_buffer 0.5 --dates_buffer 2.0,2.0 --silent

# with:

    # Mandatory arguments:
    --ifeatures ifeatures         # list of indexes of feature(s) to analyze;
                                  # numbering starts with 0; these are not
                                  # feature IDs themselves but the order they
                                  # appear in "historical_flood.json"
                                  # example:  "2,12,22"
    # Optional arguments:
    --tmpdir tmpdir               # Name of directory to save downloaded and
                                  # produced files under
                                  # example:  "/tmp/"
    --bbox_buffer bbox_buffer     # minimum distance between locations provided and
                                  # outline of bounding box (in degrees); needs to
                                  # be non-negative; unit:  degree
                                  # example:  "0.5"
    --dates_buffer dates_buffer   # used to offset start_date to
                                  # start_date - dates_buffer[0] and end_date to
                                  # end_date + dates_buffer[1] in order to retrieve
                                  # precipitation data prior to and after flooding
                                  # event; needs to be non-negative; unit:  days
                                  # example:  "5.0,5.0"
    --silent                      # if given, no printing on screen
```

There is no standard return of this command-line call. You will need to gather yourself where file(s) are stored.

A standard JSON file containing the summary of the analysis looks like the following:

```
# JSON file produced by analyse_occurrence()
# Note:  gray text has been added as comment here but is not part of the
#        actual file produced

{
    # accumulated precipitation over identified precipitation event period P;
    # value is rounded to 2 digits
    "accumulated_mm":  90.06,
    # start date of files analyzed to find precipitation event period P, i.e.
    # feature start date including "dates_buffer[0]"
    "start_date_w_buffer":  "2018-07-19 00:00:00",
    # end date of files analyzed to find precipitation event period P, i.e.
    # feature end date including "dates_buffer[1]"
    "end_date_w_buffer":  "2018-07-30 18:00:00",
    # number of time steps no data is available during identified
    # precipitation event period P
    "missing_timesteps_n":  1,
    # percentage of time steps no data is available during identified
    # precipitation event period P
    "missing_timesteps_%":  3.12,
    # number of time steps available during identified
    # precipitation event period P
    "available_timesteps_n":  31,
    # percentage of time steps available during identified
    # precipitation event period P
    "available_timesteps_%":  96.88,
    # list of time steps available during identified precipitation
    # event period P incl.  precipitation at this (period ending) time step;
    # unit is [mm/dt] where dt is the temporal resolution of the data, i.e.,
    # 6h for RDPA-6h and 1h for RDRS_v2.1; values are rounded to 2 digits;
    # ↷ sum of the "available_timesteps_n" values equals "accumulated_mm"
    "available_timesteps_precip_mm/dt":  {
        "2018-07-22 18:00:00":  5.07,
        "2018-07-23 00:00:00":  0.1,
        "2018-07-23 06:00:00":  0.73,
        ...
        "2018-07-30 00:00:00":  2.88,
        "2018-07-30 06:00:00":  0.15,
        "2018-07-30 12:00:00":  0.02
    },
    # product used as data source for precipitation
    "data_source":  "rdpa:10km:6f"
}
```

9. Analyze event

Workflow: Sequence of steps to analyze events (multi-point feature).

The script/function summarizes the workflow to analyze multi-point events (events) for the HFE database, i.e., `"historical_flood_event.json"`. The user needs to specify a feature index (or several). The other arguments are optional and set to default values. This script can be called directly from the command-line with command-line arguments or as a function within another Python script. There is no new functionality added here. This wrapper script is using only the functions described above.

A. Usage as Python function:

```
# see module for detailed documentation and example
from analyse_event import analyse_event

files_created = analyse_event(
      # Mandatory arguments:
      ifeatures=ifeatures,          # list of indexes of feature(s) to analyze;
                                    # numbering starts with 0; these are not
                                    # feature IDs themselves but the order they
                                    # appear in "historical_flood_event.json"
      # Optional arguments:
      tmpdir=tmpdir,                # Name of directory to save downloaded and
                                    # produced files under; default:  "/tmp/"
      bbox_buffer=bbox_buffer,      # minimum distance between locations provided and
                                    # outline of bounding box (in degrees); needs to
                                    # be non-negative; unit:  degree; default 0.5
      dates_buffer=dates_buffer,    # used to offset start_date to
                                    # start_date - dates_buffer[0] and end_date to
                                    # end_date + dates_buffer[1] in order to retrieve
                                    # precipitation data prior to and after flooding
                                    # event; needs to be non-negative; unit:  days
                                    # default [5.0,5.0]
      silent=silent,                # True for no printing on screen
      )
```

The return value `"files_created"` is a dictionary of files created. It has the keys `"png"` containing PNG files created (one for each timestep showing the spatial distribution of precipitation) and `"json"` containing JSON files that have the summarized analysis stored in the format they will be used to augment the HFE database. It also contains the key `"png-ts"` which is the PNG showing the time series of precipitation for all locations with the identified precipitation event highlighted in blue (same as `"png"` returned for `"analyse_occurrence()"`). The key `"gif"` contains the GIF of all the merged files (listed under `"png"`). Finally, it has the key `"legend"` which is a PNG showing the color legend of the PNGs and GIFs.

Please be aware that a lot of files will be created for long events. Due to that, events that are longer than 90 days are skipped at the moment. If the hourly RDRS_v2.1 product would be used, in total 24×90 files would be created and listed under `"png"`. The merged GIF would likely be very large. The setting can be adjusted in `src/analyse_event.py` around line 233 (`"if (length_event > 90.)"`).

B. Usage as commandline function:

```
# command-line terminal; '\' at the end of lines below indicate line breaks
# for readability; remove before copying to terminal

# List of feature indexes to analyse
features='2,12,22'

# Call of Python workflow
python analyse_event.py --ifeatures "${features}" \
    --tmpdir "/project/6070465/julemai/nrcan-hfe/data/output/" \
    --bbox_buffer 0.5 --dates_buffer 2.0,2.0 --silent

# with:

    # Mandatory arguments:
    --ifeatures ifeatures          # list of indexes of feature(s) to analyze;
                                   # numbering starts with 0; these are not
                                   # feature IDs themselves but the order they
                                   # appear in "historical_flood_event.json"
                                   # example:  "2,12,22"
    # Optional arguments:
    --tmpdir tmpdir                # Name of directory to save downloaded and
                                   # produced files under
                                   # example:  "/tmp/"
    --bbox_buffer bbox_buffer      # minimum distance between locations provided and
                                   # outline of bounding box (in degrees); needs to
                                   # be non-negative; unit:  degree
                                   # example:  "0.5"
    --dates_buffer dates_buffer    # used to offset start_date to
                                   # start_date - dates_buffer[0] and end_date to
                                   # end_date + dates_buffer[1] in order to retrieve
                                   # precipitation data prior to and after flooding
                                   # event; needs to be non-negative; unit:  days
                                   # example:  "5.0,5.0"
    --silent                       # if given, no printing on screen
```

There is no standard return of this command-line call. You will need to gather yourself where file(s) are stored.

A standard JSON file containing the summary of the analysis looks like the following (similar to the JSON created using `analyse_occurrence()` but for multiple locations):

```
# JSON file produced by analyse_event()
# Note:  gray text has been added as comment here but is not part of the
#        actual file produced

{
    # accumulated precipitation over identified precipitation event period P;
    # value is rounded to 2 digits
    "accumulated_mm":  [ 90.06, 41.76, ... ],
    # start date of files analyzed to find precipitation event period P, i.e.
    # feature start date including "dates_buffer[0]"
    "start_date_w_buffer":  "2018-07-19 00:00:00",
    # end date of files analyzed to find precipitation event period P, i.e.
    # feature end date including "dates_buffer[1]"
    "end_date_w_buffer":  "2018-07-30 18:00:00",
    # number of time steps no data is available during identified
    # precipitation event period P
    "missing_timesteps_n":  [ 1, 0, ...],
    # percentage of time steps no data is available during identified
    # precipitation event period P
    "missing_timesteps_%":  [ 3.12, 0.0, ...],
    # number of time steps available during identified
    # precipitation event period P
    "available_timesteps_n":  [ 31, 36, ...],
    # percentage of time steps available during identified
    # precipitation event period P
    "available_timesteps_%":  [ 96.88, 100.0, ...],
    # list of time steps available during identified precipitation
    # event period P incl.  precipitation at this (period ending) time step;
    # unit is [mm/dt] where dt is the temporal resolution of the data, i.e.,
    # 6h for RDPA-6h and 1h for RDRS_v2.1; values are rounded to 2 digits;
    # ↷ sum of the "available_timesteps_n" values equals "accumulated_mm"
    "available_timesteps_precip_mm/dt":  [
        {
            "2018-07-22 18:00:00":  5.07,
            ...
            "2018-07-30 12:00:00":  0.02
        },
        {
            "2018-07-21 18:00:00":  15.0,
            ...
            "2018-07-30 18:00:00":  1.05
        },
        ...
        ],
    # product used as data source for precipitation
    "data_source":  "rdpa:10km:6f"
}
```