Workflow Explanation: Pre-Optimization Data Generation and SDU Methods with Algorithmic Changes

Lifeng Ren

2025-02-25

Overview

This document explains the workflow used to generate pre-optimization data for spatial optimization. The process has two major steps:

1. **Compilation of Cython Functions:**
   The `setup.py` script compiles performance-critical Cython modules (e.g., for scenario creation and carbon computations) so that heavy computations run efficiently.

2. **Preprocessing and Data Aggregation:**
   The main workflow is executed by **preprocess_flexible.py** which orchestrates the overall pipeline using a task graph. This script, in turn, calls functions defined in **preprocessing_flexible.py** that implement the details of data slicing, spatial decision unit (SDU) creation, and marginal value aggregation.

Integration of preprocess_flexible.py and preprocessing_flexible.py

preprocess_flexible.py serves as the entry point of the workflow. It: - Reads the configuration (from a file like `preprocess_config_flexible.yml`). - Sets up the task graph. - Dispatches tasks (such as creating SDUs) that rely on functions defined in **preprocessing_flexible.py**.

For example, in **preprocess_flexible.py** the function `tg_create_spatial_decision_units` selects the appropriate SDU generation method based on the `spatial_unit_type` configuration. The code snippet below shows this decision logic:

```
def       tg_create_spatial_decision_units(tg,       country_root,       args,
mask_base_raster_task, spatial_unit_type):
    c = os.path.basename(country_root)

    # Set target file based on spatial unit type
    if spatial_unit_type == "pixel":
        output_file = os.path.join(country_root, "pixel_map", "pixel_grid.tif")
    elif spatial_unit_type == "coarser_grid":
```

```
                output_file  =  os.path.join(country_root,  "coarser_grid_map",
"coarser_grid_detailed.tif")
    else:  # hexagon
        output_file = os.path.join(country_root, "sdu_map", "sdu.shp")

    make_sdu_task = tg.add_task(
                    func=create_spatial_decision_units,     #   Imported   from
preprocessing_flexible.py
        args=[country_root, args],
        task_name=f"create_spatial_decision_units_{c}",
        target_path_list=[output_file],
        dependent_task_list=[mask_base_raster_task]
    )
    return make_sdu_task
```

The function `create_spatial_decision_units` (defined in **preprocessing_flexible.py**) then examines the configuration and calls the appropriate sub-function: - For `"hexagon"`, it generates a hexagonal grid. - For `"pixel"`, it calls `create_pixel_grid`. - For `"coarser_grid"`, it calls `create_coarser_grid`.

Thus, the main workflow (in **preprocess_flexible.py**) acts as the orchestrator that delegates tasks to the implementation functions in **preprocessing_flexible.py**.

Detailed Explanation of SDU Methods

In the following sections, we describe the three SDU generation methods by combining their mathematical formulations with the exact code excerpts where these calculations are implemented.

1. Hexagon Method (Original)

Mathematical Formulation

- **Hexagon Dimensions:**
  With a given cell size:
  - $\delta_{short} = 0.25 \times \text{cell\_size}$
  - $\delta_{long} = 0.5 \times \text{cell\_size}$
  - $\delta_y = 0.25 \times \sqrt{3} \times \text{cell\_size}$
- **Grid Dimensions:**
  The number of columns and rows are calculated as:

$$n_{\text{cols}} = \left\lfloor \frac{\text{grid\_width}}{3 \times \delta_{long}} \right\rfloor + 1$$

$$n_{\text{rows}} = \left\lfloor \frac{\text{grid\_height}}{\delta_y} \right\rfloor + 1$$

Code Reference

In **preprocessing_flexible.py**, within the function `create_regular_sdu_grid`, the algorithm is implemented by: - Calculating the extents of the raster. - Defining `delta_short_x`, `delta_long_x`, and `delta_y` based on the cell size. - Determining the number of rows and columns using the formulas above. - Generating each hexagon by computing the centroid differently for odd and even rows and then applying offsets.

This section remains as originally implemented.

2. Pixel Method (New)

Mathematical Formulation

- **Unique Pixel ID Calculation:**
  For a raster with dimensions rows $\times$ cols:

$$pixel_{id} = \text{row} \times \text{cols} + \text{col} + 1$$

  This formula guarantees every valid pixel gets a unique identifier (ignoring nodata).

Code Changes and Implementation

In **preprocessing_flexible.py**, the function `create_pixel_grid` implements the pixel method:

- **Implementation of the Math:**
  The unique ID is computed by iterating over each row and column:

```
for r in range(rows):
    for c in range(cols):
        pixel_ids[r, c] = r * cols + c + 1  # +1 avoids 0 as an ID
```

  This loop is a direct translation of the mathematical formula.

- **Handling nodata:**
  The code applies a mask so that nodata pixels are set to -1:

```
mask = source_raster == nodata
pixel_ids[mask] = -1
```

Thus, the pixel method retains the full resolution by treating every pixel as an individual decision unit.

3. Coarser Grid Method (New)

Mathematical Formulation

- **Aggregation with Coarsening Factor:**
  With a coarsening factor f, each new cell aggregates $f \times f$ original pixels.

- **New Raster Dimensions:**
  If the original raster is $W \times H$, then:

$$\text{new\_width} = \frac{W}{f}, \quad \text{new\_height} = \frac{H}{f}$$

- **Scaling the Pixel Size:**
  The new pixel dimensions are:

  $$\text{new\_pixel\_width} = \text{base\_pixel\_width} \times f, \quad \text{new\_pixel\_height} = \text{base\_pixel\_height} \times f$$

Code Changes and Implementation

In **preprocessing_flexible.py**, the function `create_coarser_grid` carries out the following steps:

- **Computing New Dimensions:**

  ```
  new_width = original_width // coarsening_factor
  new_height = original_height // coarsening_factor
  ```

  This applies the mathematical division by f.

- **Adjusting the Geotransform:**

  ```
  new_geotransform[1] = base_geotransform[1] * coarsening_factor  # new pixel
  width
  new_geotransform[5] = base_geotransform[5] * coarsening_factor  # new pixel
  height
  ```

  This code scales the original pixel size.

- **Assigning Unique IDs:** A nested loop assigns unique IDs for each coarser cell:

  ```
  grid_id = 1
  for r in range(new_height):
      for c in range(new_width):
          id_array[r, c] = grid_id
          grid_id += 1
  ```

  This ensures every aggregated cell gets a unique identifier.

Running preprocess_flexible.py in VSCode

Follow these steps to run the preprocessing script in VSCode using your configuration file and folder structure:

1. **Folder Structure:**
   Ensure your project folder includes:

   - A `src` folder containing your scripts (including **preprocess_flexible.py** and the module **preprocessing_flexible.py**).

- A `configs` folder containing your configuration file (e.g., `preprocess_config_flexible.yml`).
- A `.vscode` folder with a `launch.json` file configured for debugging.

2. **Review launch.json:**
Your `launch.json` should include an entry like:

```
{
    "name": "Python: preprocess_flexible.py",
    "type": "python",
    "request": "launch",
    "program": "${workspaceFolder}/src/scripts/preprocess_flexible.py",
                            "args":       ["${workspaceFolder}/src/configs/
preprocess_config_flexible.yml"],
    "console": "integratedTerminal",
    "justMyCode": true,
    "env": {
        "PYTHONPATH": "${workspaceFolder}/src"
    }
}
```

This configuration launches the preprocessing script with the specified configuration file.

3. **Open the Project in VSCode:**
   - Open VSCode.
   - Select "File" > "Open Folder..." and choose your project folder.
4. **Start the Script:**
   - Press `F5` to start debugging, or choose "Run" > "Start Without Debugging".
   - The integrated terminal will display progress messages (e.g., "Created pixel grid with unique IDs at ..." or "Created coarser grid at ...").
5. **Monitor the Output:**
Output files will be created in folders such as `pixel_map`, `coarser_grid_map`, or `sdu_map` according to the configuration settings.

Conclusion

This document now provides a comprehensive explanation of the preprocessing workflow with: - **Mathematical formulations** for each SDU method. - **Code excerpts** from **preprocessing_flexible.py** showing the algorithmic changes (including the new pixel and coarser grid methods). - A clear description of the **link between preprocess_flexible.py and preprocessing_flexible.py**, where the main workflow dispatches tasks that are implemented in the preprocessing module. - Detailed instructions on **running the preprocessing script in VSCode** using your configuration file and folder structure.