Санкт-Петербургский политехнический университет Петра Великого

Институт машиностроения, материалов и транспорта

Высшая школа автоматизации и робототехники

**Отчёт**

**по курсовой работе по теме  
«»**

Дисциплина: «Объектно-ориентированное программирование»

Студент гр. 3331506/20102 Коннов К. Г.

Преподаватель Ананьевский М. С.

Санкт-Петербург

2025

**СОДЕРЖАНИЕ**

[ВВЕДЕНИЕ 3](#_Toc197379407)

[АЛГОРИТМИЧЕСКАЯ ЧАСТЬ 4](#_Toc197379408)

[Постановка задачи 4](#_Toc197379409)

[Реализуемые функциональности 4](#_Toc197379410)

[Используемые библиотеки 5](#_Toc197379411)

[Программный код 6](#_Toc197379412)

[ЗАКЛЮЧЕНИЕ 9](#_Toc197379413)

# ВВЕДЕНИЕ

Космическая аэрофотосъёмка (дистанционное зондирование Земли) сегодня является ключевым источником данных для мониторинга окружающей среды, сельского хозяйства, городского планирования и других областей. Съёмка с помощью спутников или БПЛА генерирует огромные массивы многоспектральных изображений, требующих эффективных методов анализа. В таких условиях автоматизация обработки данных с помощью программных средств становится крайне актуальной задачей. Python — один из наиболее популярных языков программирования для анализа геопространственных данных благодаря богатому набору библиотек и простой интеграции различных инструментов. Использование Python позволяет быстро прототипировать алгоритмы обработки изображений, а готовые библиотеки ускоряют решение типовых задач и уменьшают количество рутинных операций.

Автоматизированная обработка космических изображений с помощью Python обладает существенными преимуществами. Во-первых, она значительно ускоряет обработку больших объёмов данных по сравнению с ручными или полуавтоматическими методами в геоинформационных системах (GIS). Во-вторых, скриптовая автоматизация обеспечивает воспроизводимость результатов: один и тот же алгоритм можно многократно применять к разным наборам снимков. В-третьих, использование открытых библиотек и стандартов (GDAL, GeoTIFF, GeoJSON и т. д.) позволяет обрабатывать данные из различных источников без преобразований между проприетарными форматами. Все эти факторы делают разработку специализированного ПО для автоматической обработки аэрофотосъёмки на Python важной и перспективной задачей.

# АЛГОРИТМИЧЕСКАЯ ЧАСТЬ

## Постановка задачи

Космические и аэрофотоснимки представляют собой растровые геопривязанные изображения: каждый пиксель имеет координаты на земной поверхности и может содержать данные по нескольким спектральным каналам (например, красному, зелёному, инфракрасному). Основные этапы обработки таких данных включают преобразование проекций, выделение областей интереса (ROI) с помощью векторных масок, анализ значений пикселей и учёт метаданных (привязка, разрешение, система координат и т. д.). При этом важно работать с большими массивами чисел (пикселями), выполнять геометрические операции над полигонами и применять математические преобразования.

В качестве задачи данной курсовой работы было решено создать универсальные базовые классы для хранения, обработки и конвертации космических данных. Это базовый класс *Raster*, содержащий все необходимые методы, а также базовый класс *Layer*,необходимый для обработки многоканальных снимков, а также их геометрической обработки с помощью векторных типов данных.

## Реализуемые функциональности

Далее приведен список основных функциональностей в базовых классах, а также описание :

* Работа с проекциями и системами координат.

Изображения могут быть записаны в разных системах координат (например, UTM или WGS84). Понимание и преобразование проекций обеспечивают корректное наложение растров и векторных данных. Для этого используется библиотека ***pyproj*** [5], позволяющая конвертировать координаты между EPSG-кодами и произвольными CRS.

* Маскирование и обрезка растра.

Часто необходимо «вырезать» часть изображения по границе интересующего полигона или маски. Библиотека ***rasterio*** предоставляет модуль ***rasterio.mask***, который позволяет применить геометрические маски: все пиксели, выходящие за пределы заданного полигона, можно обнулить или пометить как *NoData*. При этом опция *crop=True* обеспечивает «отрезание» растра по минимальному охватывающему прямоугольнику маски.

* Работа с метаданными и координатными преобразованиями.

Растровый файл содержит метаданные — информацию о геопривязке (преобразование *transform*), системе координат (CRS), размерах и т. д. При чтении через ***rasterio*** [2] можно получить эти атрибуты и, например, определить географические координаты любого пикселя *(src.xy(row, col))*, затем преобразовать их в другую систему с помощью ***pyproj*** [5]. Такие операции необходимы для анализа точных координат объектов на снимке.

## Используемые библиотеки

Стоит более детально рассмотреть основные используемые библиотеки. Их описание приведено ниже в виде списка:

* NumPy — фундаментальный пакет для научных вычислений с Python. Он предоставляет эффективные N-мерные массивы и математические функции. Растровые данные при чтении обычно преобразуются в объекты *numpy.ndarray* [4], что даёт возможность быстрого векторного и матричного анализа (фильтрации, агрегации, арифметики над пикселями).
* Rasterio — высокоуровневая библиотека для чтения, записи и обработки растровых геоданных. Она основана на GDAL и интегрируется с NumPy. Rasterio «читает и записывает» форматы GeoTIFF и другие, предоставляя Python-API на основе Numpy-массивов. С помощью Rasterio можно открывать файлы, читать отдельные полосы изображения, получать метаданные (*dataset.meta*, *dataset.transform*, *dataset.crs* и др.), а также выполнять пространственные операции (изменение проекции, выделение снимков и т. д.).
* GeoPandas [3] — расширение библиотеки Pandas для работы с геопространственными данными. GeoPandas объединяет табличные возможности Pandas с возможностями библиотеки Shapely по работе с геометрическими объектами. Это позволяет легко читать и обрабатывать векторные форматы (shapefile, GeoJSON и др.), фильтровать и трансформировать полигоны. В рамках задачи обработки аэрофотоснимков GeoPandas используется для хранения границ областей интереса, объединения атрибутивной информации с пространственной или для создания новых векторных объектов (буферизация, объединение и т. д.).
* Pyproj — библиотека для преобразования систем координат. Она необходима, когда растровые данные и векторные полигоны имеют разные проекции. Например, координаты, полученные из растра через Rasterio (обычно в метрах в местной системе), преобразуют в широту/долготу (WGS84) с помощью pyproj. Pyproj поддерживает стандарты EPSG и PROJ, что позволяет гибко задавать и конвертировать CRS.

## Комментарии к программному коду

Реализованный код представлен в приложении. Он имеет структуру, соответствующую базовым принципам ООП. Также код содержит все необходимые комментарии, типизацию, обработку ошибок. Описание структуры алгоритма и основных реализованных методов представлено далее.

Класс Layer представляет собой отдельный слой растровых данных, инкапсулирующий сами данные и связанные с ними метаданные. Он предоставляет методы для валидации, доступа к данным и выполнения геопространственных операций. Его основные методы и функциональность:

1. Инициализация (\_\_*init*\_\_): создает экземпляр Layer с растровыми данными (в формате *DatasetReader* или массива *numpy*) и метаданными (в виде объекта *Metadata* или словаря). Если метаданные переданы в виде словаря, они преобразуются в объект *Metadata*. Данные проверяются с помощью метода *\_validate\_data* [1].
2. Управление системой координат **(***crs***)**: Свойство *crs* предоставляет методы для получения и установки системы координат (CRS). Установка CRS приводит к перепроецированию набора данных и обновлению метаданных, что обеспечивает геопространственную согласованность [2].
3. Обрезка (*crop*): выполняет обрезку слоя с использованием шейп-файла, *GeoDataFrame* или *GeoSeries*. Поддерживает выбор конкретных геометрических объектов и обновляет метаданные после обрезки, что позволяет проводить точный пространственный анализ [3].
4. Доступ к данным (*to*\_*numpy*, *get*\_*by*\_*index*): Метод *to*\_*numpy* преобразует растровые данные в массив *numpy*, а *get\_by\_index* позволяет извлечь значение по указанным индексам строки и столбца, упрощая манипуляции на уровне пикселей [1].
5. Статистические свойства (*max*, *min*, *mean*): вычисляет максимальное, минимальное и среднее значения растровых данных, предоставляя быстрый доступ к статистическим характеристикам слоя [4].
6. Управление метаданными (*metadata*): Свойство *metadata* позволяет получать и обновлять метаданные, обеспечивая согласованность геопространственных атрибутов [2].

Класс *Raster* представляет многослойный растровый набор данных, управляя коллекцией объектов Layer. Он поддерживает операции с несколькими полосами (*bands*) и предоставляет геопространственные функции для комплексной обработки растров. Его основные методы и функциональность

1. Инициализация (\_\_*init*\_\_): создает экземпляр *Raster* либо на основе *DatasetReader*, либо на основе списка объектов Layer. Гарантирует, что предоставлены либо данные, либо полосы, извлекает слои из набора данных при необходимости и инициализирует метаданные [1].
2. Доступ к слоям и итерация (\_\_*getitem*\_\_, \_\_*setitem*\_\_, \_\_*iter*\_\_, \_\_*len*\_\_): обеспечивает индексацию, установку, итерацию и определение количества слоев, упрощая работу с многослойными растровыми данными [1].
3. Координаты (*coordinates*): генерирует массивы широт и долгот для каждого пикселя в растре, что полезно для геопространственного анализа [2, 5].
4. Обрезка (*crop*): выполняет обрезку указанных полос растра с использованием шейп-файла или геометрических данных, обновляя метаданные для сохранения согласованности [3].
5. Сохранение в файл (*to\_file*): сохраняет растр в файл, позволяя выбрать определенные полосы для сохранения, что удобно для экспорта данных [4].
6. Поиск ближайшего пикселя (*nearest*\_*pixel*, *pixel*\_*info*): находит пиксель, ближайший к заданным координатам (широта и долгота), и возвращает информацию о нем, включая значение в указанной полосе, что полезно для точечного анализа [2].
7. Статистические методы (*max*, *min*, *mean*): вычисляет статистические показатели (максимум, минимум, среднее) для указанной полосы, предоставляя быстрый доступ к характеристикам данных [4].

# ЗАКЛЮЧЕНИЕ

В ходе курсовой работы были реализованы базовые классы для растрового типа данных и слоя многоканального спутникового снимка. Описаны все необходимые методы для расчётов и анализа растра.

Благодаря заложенное модульной архитектуре систему можно гибко дорабатывать и расширять. Среди направлений дальнейшей доработки: добавление алгоритмов машинного обучения для классификации или детектирования объектов на снимках, поддержка новых источников данных (другие форматы или прямой доступ к космическим архивам), оптимизация скорости (параллельная обработка или работа с аппаратным ускорением), а также создание удобного графического интерфейса пользователя.

В целом, создание небольшой, но функциональной библиотеки для бесшовной обработки космических данных различных типов и их анализа является перспективной и востребованной задачей в этой отрасли.

# СПИСОК ЛИТЕРАТУРЫ

1. Хантер, Дж. Д. Matplotlib: среда для двумерной графики // Вычисления в науке и технике. — 2007. — Т. 9, № 3. — С. 90–95. — DOI: 10.1109/MCSE.2007.55.
2. Гиллис, С. и др. Rasterio: доступ к геопространственным растровым данным для программистов на Python [Электронный ресурс]. — GitHub Repository, 2013. — URL: https://github.com/rasterio/rasterio (дата обращения: 06.05.2025).
3. Джордал, К. и др. GeoPandas: инструменты Python для работы с географическими данными [Электронный ресурс]. — GitHub Repository, 2019. — URL: https://github.com/geopandas/geopandas (дата обращения: 06.05.2025).
4. Ван дер Валт, С., Колберт, С. К., Варокво, Г. Массив NumPy: структура для эффективных числовых вычислений // Вычисления в науке и технике. — 2011. — Т. 13, № 2. — С. 22–30. — DOI: 10.1109/MCSE.2011.37.
5. Сноу, К. и др. Pyproj: Python-интерфейс для PROJ — библиотеки картографических проекций [Электронный ресурс]. — GitHub Repository, 2020. — URL: https://github.com/pyproj4/pyproj (дата обращения: 06.05.2025).

# ПРИЛОЖЕНИЕ

from typing import Any, Iterator

import numpy as np

from geopandas import GeoDataFrame, GeoSeries

from pyproj import CRS

from rasterio import DatasetReader, open as open\_raster, transform as raster\_transform

from rasterio.mask import mask as mask\_band

from .constants import DRIVERS

from .metadata import Metadata

from .utils import reproject\_dataset, to\_dataset

from ..shapes.polygon import PolygonShapefile

class Layer:

"""

Represents a single layer of raster data.

"""

def \_\_init\_\_(

self,

data: DatasetReader | np.ndarray | None = None,

metadata: Metadata | dict | None = None

) -> None:

"""

Initializes a new Layer instance.

Args:

data (DatasetReader | np.ndarray | None): The raster data.

metadata (Metadata | dict | None): Metadata associated with the layer.

"""

self.\_metadata = metadata

# Convert metadata to Metadata object if it's a dictionary

if isinstance(self.\_metadata, dict):

self.\_metadata = Metadata(\*\*self.\_metadata)

# Convert data to a DatasetReader and validate it

self.\_data = to\_dataset(data, self.\_metadata.to\_dict())

self.\_validate\_data()

def \_validate\_data(self) -> None:

"""

Validates the data in the layer.

This method can be overridden by subclasses to add specific validation logic.

"""

pass

@staticmethod

def \_unpack(stack: np.ndarray) -> np.ndarray:

"""

Unpacks a single-layer stack into a 2D array if necessary.

Args:

stack (np.ndarray): The input array.

Returns:

np.ndarray: The unpacked array.

"""

if stack.shape[0] == 1 and len(stack.shape) == 3:

return stack[0]

return stack

@property

def crs(self) -> CRS | str:

"""

Gets the Coordinate Reference System (CRS) of the layer.

Returns:

CRS | str: The CRS of the layer.

"""

return self.\_metadata.crs

@crs.setter

def crs(self, crs: CRS | str) -> None:

"""

Sets the Coordinate Reference System (CRS) of the layer.

Args:

crs (CRS | str): The CRS to set.

"""

if isinstance(crs, str):

crs = CRS.from\_user\_input(crs)

elif not isinstance(crs, CRS):

raise ValueError("Invalid CRS type.")

# Reproject the dataset and update metadata

self.\_data = reproject\_dataset(self.\_data, crs)

self.\_metadata.update(metadata=self.\_data.meta)

def crop(self, shapefile: PolygonShapefile | GeoDataFrame | GeoSeries, \*\*kwargs) -> "Layer":

"""

Crops the layer using a given shapefile or GeoDataFrame.

Args:

shapefile (PolygonShapefile | GeoDataFrame | GeoSeries): The shapefile or GeoDataFrame to use for cropping.

\*\*kwargs: Additional keyword arguments for cropping.

Returns:

Layer: The cropped layer.

"""

if self.\_data is None:

raise ValueError("Layer is empty.")

# Handle specific geometry number if provided

features = kwargs.pop("features", None)

if isinstance(features, list) and isinstance(shapefile, PolygonShapefile):

shapefile = [shapefile.geometry\_feature(number) for number in features]

elif isinstance(features, int) and isinstance(shapefile, PolygonShapefile):

shapefile = shapefile.geometry\_feature(features)

elif isinstance(shapefile, PolygonShapefile):

shapefile = shapefile.geometry

# Ensure shapefile is a list of geometries

if isinstance(shapefile, (GeoDataFrame, GeoSeries)):

shapefile = [shapefile]

# Crop the data using the provided shapefile

for geometry in shapefile:

data, transform = mask\_band(self.data, geometry, crop=True, \*\*kwargs)

self.\_metadata.update\_after\_crop(transform=transform, crop=data[0])

self.\_data = to\_dataset(data, self.\_metadata.to\_dict())

return self

@classmethod

def from\_layer(cls, \*args, \*\*kwargs) -> "Layer":

"""

Creates a new Layer instance from an existing layer.

Args:

\*args: Positional arguments.

\*\*kwargs: Keyword arguments.

Returns:

Layer: A new Layer instance.

Raises:

NotImplementedError: This method should be implemented by subclasses.

"""

raise NotImplementedError

def get\_by\_index(self, row: int, col: int):

"""

Gets the value at a specific row and column index in the raster data.

Args:

row (int): The row index.

col (int): The column index.

Returns:

The value at the specified index.

"""

array = self.to\_numpy()

if row >= self.shape[0] or col >= self.shape[1]:

raise ValueError("Invalid row or column index.")

return array[row][col]

@property

def data(self) -> DatasetReader:

"""

Gets the raster data.

Returns:

DatasetReader: The raster data.

"""

return self.\_data

@property

def max(self) -> int | float:

"""

Gets the maximum value in the raster data.

Returns:

int | float: The maximum value.

"""

return self.to\_numpy().max()

@property

def mean(self) -> int | float:

"""

Gets the mean value in the raster data.

Returns:

int | float: The mean value.

"""

return self.\_data.read().mean()

@property

def metadata(self) -> Metadata:

"""

Gets the metadata associated with the layer.

Returns:

Metadata: The metadata.

"""

return self.\_metadata

@metadata.setter

def metadata(self, metadata: Metadata) -> None:

"""

Sets the metadata associated with the layer.

Args:

metadata (Metadata): The metadata to set.

"""

self.\_metadata = metadata

@property

def min(self) -> int | float:

"""

Gets the minimum value in the raster data.

Returns:

int | float: The minimum value.

"""

return self.\_data.read().min()

@property

def shape(self) -> tuple[int, ...]:

"""

Gets the shape of the raster data.

Returns:

tuple: The shape of the raster data.

"""

return self.\_data.shape

def to\_numpy(self) -> np.ndarray:

"""

Converts the raster data to a NumPy array.

Returns:

np.ndarray: The raster data as a NumPy array.

"""

return self.\_unpack(self.\_data.read().astype(np.float32))

class Raster:

"""

Represents a raster with multiple layers.

"""

\_data: DatasetReader

\_metadata: Metadata

\_bands: list[Layer]

def \_\_init\_\_(

self,

data: DatasetReader | None = None,

bands: list[Layer] | None = None,

metadata: Metadata | dict | None = None,

\*\*kwargs

) -> None:

"""

Initializes a new Raster instance.

Either `data` or `bands` must be provided. If `data` is provided, it initializes the raster

using the dataset reader and extracts layers from it. If `bands` is provided, it initializes

the raster using the list of layers and constructs the dataset from these layers.

Args:

data (DatasetReader | None): The raster data. If provided, `bands` should be None.

bands (list[Layer] | None): The layers of the raster. If provided, `data` should be None.

metadata (Metadata | dict | None): Metadata associated with the raster. If provided as a dictionary,

it will be converted into a `Metadata` object.

Raises:

ValueError: If neither `data` nor `bands` is provided.

"""

if (data is None and bands is None) or (data is not None and bands is not None):

raise ValueError("Either data or bands must be provided.")

self.\_metadata = metadata

# Convert metadata to Metadata object if it's a dictionary

if isinstance(self.\_metadata, dict):

self.\_metadata = Metadata(\*\*self.\_metadata)

if data is not None:

self.\_data = data

self.\_bands = self.\_extract\_layers()

if self.\_metadata is None:

self.\_metadata = Metadata(\*\*self.\_data.meta)

else:

self.\_bands = bands # noqa

self.\_data = to\_dataset(np.dstack([layer.to\_numpy() for layer in bands]), self.\_metadata.to\_dict()) # noqa

if crs := kwargs.pop("crs", None):

self.crs = crs

def \_\_getitem\_\_(self, item: int) -> Layer:

"""

Gets a specific layer from the raster.

Args:

item (int): The index of the layer to get.

Returns:

Layer: The specified layer.

"""

return self.\_bands[item]

def \_\_setitem\_\_(self, key: int, value: Layer) -> None:

"""

Sets a specific layer in the raster.

Args:

key (int): The index of the layer to set.

value (Layer): The layer to set.

"""

self.\_bands[key] = value

def \_\_iter\_\_(self) -> Iterator[Layer]:

"""

Iterates over the layers in the raster.

Returns:

iterator: An iterator over the layers.

"""

return iter(self.\_bands)

def \_\_len\_\_(self) -> int:

"""

Gets the number of layers in the raster.

Returns:

int: The number of layers.

"""

return len(self.\_bands)

def \_extract\_layers(self) -> list[Layer]:

"""

Extracts individual layers from the raster data.

Returns:

list[Layer]: A list of layers.

"""

return [

Layer(data=self.\_data.read(i + 1).astype(np.float32), metadata=self.\_metadata) for i in range(self.\_data.count)

]

def \_get\_bands\_to\_save(self, bands\_nums: list[int] | int | None = None) -> np.ndarray:

"""

Gets the bands to save.

Args:

bands\_nums (list[int] | int | None): The indices of the bands to save. If None, all bands are saved.

Returns:

np.ndarray: The bands to save as a NumPy array.

"""

if bands\_nums is None:

bands\_nums = list(range(self.\_data.count))

elif isinstance(bands\_nums, int):

bands\_nums = [bands\_nums]

return np.array([self[band].to\_numpy() for band in bands\_nums])

@staticmethod

def \_unpack(stack: np.ndarray) -> np.ndarray:

"""

Unpacks a single-layer stack into a 2D array if necessary.

Args:

stack (np.ndarray): The input array.

Returns:

np.ndarray: The unpacked array.

"""

if stack.shape[0] == 1 and len(stack.shape) == 3:

return stack[0]

return stack

def coordinates(self) -> tuple[np.ndarray, np.ndarray]:

"""

Generates the latitude and longitude coordinates for each pixel in the raster.

Returns:

tuple: Two arrays representing the latitude and longitude coordinates.

"""

cols, rows = np.meshgrid(

np.arange(self.\_data.width),

np.arange(self.\_data.height),

indexing='ij'

)

longitudes, latitudes = raster\_transform.xy(self.\_data.transform, rows, cols) # noqa

return np.array(latitudes), np.array(longitudes)

def bands(self, merge: bool = False) -> np.ndarray:

"""

Retrieves the bands of the raster as numpy array.

Args:

merge (bool): Whether to merge the bands into a single array.

Returns:

np.ndarray: The bands as a NumPy array.

"""

bands = [layer.to\_numpy() for layer in self.layers]

if merge:

bands = np.stack(bands, axis=0)

return self.\_unpack(bands[0] if isinstance(bands, list) else bands)

@property

def crs(self) -> CRS | str:

"""

Gets the Coordinate Reference System (CRS) of the raster.

Returns:

CRS: The CRS of the raster.

"""

return self.metadata.crs

@crs.setter

def crs(self, crs: CRS | str) -> None:

"""

Sets the Coordinate Reference System (CRS) of the raster.

Args:

crs (CRS | str): The CRS to set.

"""

if isinstance(crs, str):

crs = CRS.from\_user\_input(crs)

elif not isinstance(crs, CRS):

raise ValueError("CRS must be a string or a `CRS` object")

# Reproject the dataset and update metadata

self.\_data = reproject\_dataset(self.\_data, crs)

self.\_metadata.update(metadata=self.\_data.meta)

# Update CRS for each band

for band in self.\_bands:

band.crs = crs

def crop(

self,

shapefile: PolygonShapefile | GeoDataFrame | GeoSeries,

bands: list[int] | int | None = None,

\*\*kwargs

) -> "Raster":

"""

Crops the raster using a given shapefile or GeoDataFrame.

Args:

shapefile (PolygonShapefile | GeoDataFrame | GeoSeries): The shapefile or GeoDataFrame to use for cropping.

bands (list[int] | int | None): The indices of the bands to crop. If None, all bands are cropped.

\*\*kwargs: Additional keyword arguments for cropping.

Returns:

Raster: The cropped raster.

"""

if isinstance(bands, int):

bands = [bands]

elif bands is None:

bands = list(range(self.\_data.count))

# Crop each specified band

for band in bands:

self[band] = self[band].crop(shapefile, \*\*kwargs)

self.\_metadata.update(metadata=self[band].metadata)

return self

@classmethod

def from\_file(cls, filepath: str, \*\*kwargs) -> "Raster":

"""

Creates a new Raster instance from a file.

Args:

filepath (str): The path to the file.

\*\*kwargs: Additional keyword arguments for opening the file.

Returns:

Raster: A new Raster instance.

"""

options = kwargs.pop("options", {})

# Determine the appropriate driver based on the file extension

driver = options.pop("driver", None)

if not driver:

driver = DRIVERS.get(filepath.split(".")[-1], None)

data = open\_raster(filepath, driver=driver, \*\*options)

return cls(data=data, metadata=data.meta, \*\*kwargs)

@classmethod

def from\_raster(cls, \*args, \*\*kwargs) -> "Raster":

"""

Creates a new Raster instance from an existing raster.

Args:

\*args: Positional arguments.

\*\*kwargs: Keyword arguments.

Returns:

Raster: A new Raster instance.

Raises:

NotImplementedError: This method should be implemented by subclasses.

"""

raise NotImplementedError

def intersect\_raster(self, other: "Raster") -> None:

"""

Intersects this raster with another raster.

Args:

other (Raster): The other raster to intersect with.

Raises:

NotImplementedError: This method is not yet implemented.

"""

raise NotImplementedError

@property

def layers(self) -> list[Layer]:

"""

Gets the layers in the raster.

Returns:

list[Layer]: The layers in the raster.

"""

return self.\_bands

def max(self, band: int = 0) -> int | float:

"""

Returns the maximum value in the specified band of the dataset.

Args:

band (int): The index of the band from which to retrieve the maximum value. Default is 0.

Returns:

int | float: The maximum value in the specified band of the dataset.

"""

return self[band].max

def mean(self, band: int = 0) -> int | float:

"""

Compute the mean value for a specified band of data.

Args:

band (int): The band for which the mean is to be computed. Defaults to 0.

Returns:

int | float: The calculated mean value of the data in the specified band.

"""

return self[band].mean

@property

def metadata(self) -> Metadata:

"""

Gets the metadata associated with the raster.

Returns:

Metadata: The metadata.

"""

return self.\_metadata

def min(self, band: int = 0) -> int | float:

"""

Computes the minimum value of the specified band in a data structure that contains multi-band data.

Parameters:

band (int): The index of the band for which the minimum value is to be computed. Defaults to 0.

Returns:

int | float: The minimum value found in the specified band of the data structure.

"""

return self[band].min

def nearest\_pixel(self, latitude: float, longitude: float) -> tuple[int, tuple[float, float]]:

"""

Finds the nearest pixel to the given longitude and latitude.

Args:

latitude (float): The latitude coordinate.

longitude (float): The longitude coordinate.

Returns:

tuple: A tuple containing the index and coordinates of the nearest pixel.

"""

latitudes, longitudes = self.coordinates()

lat\_diff = np.abs(latitudes - latitude)

lon\_diff = np.abs(longitudes - longitude)

combined\_diff = np.sqrt(lat\_diff \*\* 2 + lon\_diff \*\* 2)

idx = np.argmin(combined\_diff)

return idx, (latitudes[idx], longitudes[idx]) # noqa

def pixel\_info(self, latitude: float, longitude: float, band\_num: int = 0) -> dict[str, Any]:

"""

Gets information about the pixel closest to the given longitude and latitude.

Args:

latitude (float): The latitude coordinate.

longitude (float): The longitude coordinate.

band\_num (int): The band number to get information from. Default is 0.

Returns:

dict: A dictionary containing the band number, latitude, longitude, and value of the pixel.

"""

index, coords = self.nearest\_pixel(latitude, longitude)

value = self[band\_num].get\_by\_index(index // self.shape[-1], index % self.shape[-1])

return {

"band": band\_num,

"latitude": coords[1],

"longitude": coords[0],

"value": value

}

@property

def shape(self) -> tuple[int, ...]:

"""

Gets the shape of the raster data.

Returns:

tuple: The shape of the raster data.

"""

return self.\_data.shape

def to\_file(

self,

filepath: str,

bands\_to\_save: list[int] | int | None = None,

\*\*kwargs

) -> None:

"""

Saves the raster to a file.

Args:

filepath (str): The path to the file.

bands\_to\_save (list[int] | int | None): The indices of the bands to save. If None, all bands are saved.

\*\*kwargs: Additional keyword arguments for saving the file.

"""

bands = self.\_get\_bands\_to\_save(bands\_to\_save)

with open\_raster(fp=filepath, mode='w', \*\*self.\_metadata.to\_dict(), \*\*kwargs) as file:

try:

file.write(bands)

except ValueError:

file.write(bands[0])