

PyQGIS plugin de geoprocessamento

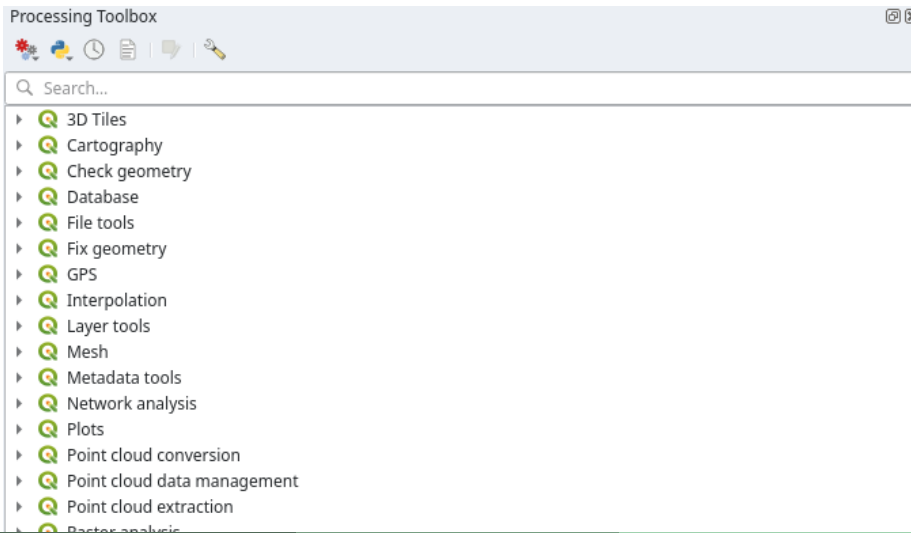
Disciplina: Programação aplicada à engenharia cartográfica

Maurício C. M. de Paulo - D.Sc.

18 de fevereiro de 2026

Objetivos

- Ilustrar operações de geoprocessamento
- Trabalhar com dados Matriciais (Raster) e Vetoriais (Vector)



Sistema de Processing no QGIS

Processing é o framework de execução de algoritmos do QGIS.

Permite:

- Executar algoritmos nativos
- Executar algoritmos do GDAL, GRASS, SAGA
- Criar algoritmos customizados
- Automatizar fluxos de geoprocessamento

A API é acessível via PyQGIS.

Executando Algoritmos via Python

```
import processing

params = {
    "INPUT": layer,
    "DISTANCE": 100,
    "SEGMENTS": 8,
    "OUTPUT": "memory:"
}

result = processing.run("native:buffer", params)
```

Formato:

```
processing.run("provider:algorithm", parametros)
```

Estrutura de um Algoritmo Customizado

Todo algoritmo herda de:

`QgsProcessingAlgorithm`

Métodos principais:

- `initAlgorithm()` → define parâmetros
- `processAlgorithm()` → lógica principal
- `name()`, `displayName()`

Definindo Parâmetros

```
self.addParameter(  
    QgsProcessingParameterVectorLayer(  
        "INPUT",  
        "Camada de entrada"  
    )  
)  
  
self.addParameter(  
    QgsProcessingParameterNumber(  
        "DIST",  
        "Distância",  
        type=QgsProcessingParameterNumber.Double  
    )  
)
```

Tipos comuns:

- VectorLayer
- RasterLayer

Lógica do processAlgorithm()

```
def processAlgorithm(self, parameters, context, feedback):  
    layer = self.parameterAsVectorLayer(  
        parameters, "INPUT", context  
    )  
  
    dist = self.parameterAsDouble(  
        parameters, "DIST", context  
    )  
  
    for feat in layer.getFeatures():  
        if feedback.isCanceled():  
            break  
  
        geom = feat.geometry()  
        new_geom = geom.buffer(dist, 8)  
  
    return {}
```

feedback permite:

Saídas com FeatureSink

Para gerar nova camada:

- Usar `QgsProcessingParameterFeatureSink`
- Criar feições
- Adicionar ao sink

Vantagens:

- Integração automática com o projeto
- Suporte a arquivos ou memória

Model Builder e Exportação

Alternativa prática:

- Criar fluxo no Model Builder
- Exportar para Python
- Adaptar o código

Útil para:

- Prototipagem rápida
- Aprender estrutura interna
- Construir plugins Processing

Exercício 1 — Executar Buffer via Python

Objetivo: executar algoritmo nativo via console.

- 1 Abrir o console Python do QGIS
- 2 Selecionar uma camada vetorial ativa
- 3 Executar:

```
import processing

layer = iface.activeLayer()

params = {
    "INPUT": layer,
    "DISTANCE": 100,
    "SEGMENTS": 8,
    "OUTPUT": "memory:"
}

result = processing.run("native:buffer", params)
```

Pergunta: Onde está armazenada a camada resultante?

Exercício 2 — Buffer Parametrizado

Objetivo: usar valor da seleção do usuário.

```
layer = iface.activeLayer()
selected = layer.selectedFeatures()

processing.run("native:buffer", {
    "INPUT": selected,
    "DISTANCE": 50,
    "OUTPUT": "memory:"
})
```

Desafio: Modifique para usar distância diferente por atributo.

Exercício 3 — Encadeando Algoritmos

Objetivo: criar fluxo Buffer → Dissolve

```
buffered = processing.run("native:buffer", {  
    "INPUT": layer,  
    "DISTANCE": 100,  
    "OUTPUT": "memory:"  
})  
  
dissolved = processing.run("native:dissolve", {  
    "INPUT": buffered["OUTPUT"],  
    "OUTPUT": "memory:"  
})
```

Pergunta: Como evitar criar camadas intermediárias visíveis?

Exercício 4 — Criar Algoritmo Simples

Objetivo: criar algoritmo próprio.

Passos:

- 1 Criar classe herdando de QgsProcessingAlgorithm
- 2 Definir parâmetros
- 3 Implementar processAlgorithm()

Tarefa: Criar algoritmo que:

- Recebe camada vetorial
- Cria campo "area_calc"
- Calcula área geométrica

Exercício 5 — Feedback e Performance

Objetivo: usar feedback corretamente.

```
for i, feat in enumerate(layer.getFeatures()):  
    if feedback.isCanceled():  
        break  
  
feedback.setProgress(i * 100 / layer.featureCount())
```

Discussão:

- Quando evitar loops Python?
- Quando delegar para algoritmo nativo?

Executando Algoritmos pelo Console

O QGIS permite executar algoritmos diretamente via console Python.

Documentação oficial:

https://docs.qgis.org/3.40/en/docs/user_manual/processing/console.html

Exemplo:

- `processing.run("native:buffer", {...})`

```
from qgis.core import *

# Supply path to qgis install location
QgsApplication.setPrefixPath("/path/to/qgis/installation", True)
#QgsApplication.setPrefixPath("/home/mauricio/miniforge3/envs/qgis/bin/", True)

# Create a reference to the QgsApplication. Setting the
# second argument to False disables the GUI.
qgs = QgsApplication([], False)

# Load providers
qgs.initQgis()

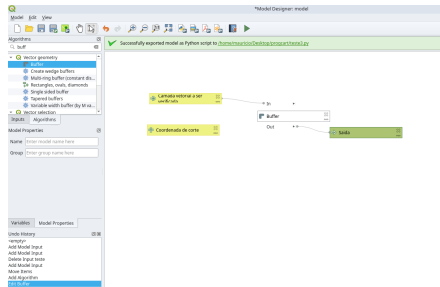
from teste3 import Model

# Write your code here to load some layers, use processing
# algorithms, etc.
```


Construindo um Script com Model Builder

Estratégia recomendada:

- 1 Abrir o **Model Builder**
- 2 Criar um modelo com as entradas do plugin
- 3 Inserir processos que serão utilizados
- 4 Testar o fluxo
- 5 Exportar para Python:
 - Model → Export → Export as Python Script

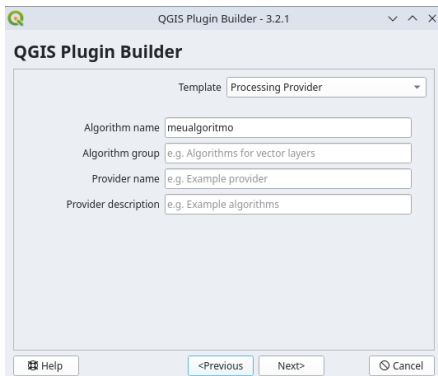


Plugin Builder

Criar um plugin do tipo **Processing Provider**

Passos:

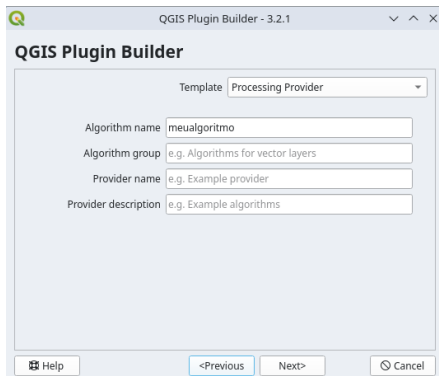
- Gerar estrutura do plugin pelo Plugin Builder
- Copiar para pasta de plugins do QGIS
- Ativar no Instalador de Plugins



The screenshot shows the 'QGIS Plugin Builder - 3.2.1' window. The 'Template' dropdown is set to 'Processing Provider'. The form contains the following fields:

- Algorithm name:
- Algorithm group:
- Provider name:
- Provider description:

At the bottom, there are buttons for 'Help', '<Previous', 'Next>', and 'Cancel'.



This is an identical screenshot to the one on the left, showing the 'QGIS Plugin Builder - 3.2.1' window with the 'Processing Provider' template selected. The fields for 'meualgoritmo', 'e.g. Algorithms for vector layers', 'e.g. Example provider', and 'e.g. Example algorithms' are filled in. The bottom buttons are 'Help', '<Previous', 'Next>', and 'Cancel'.

Adicionar o Script ao Plugin

- Copiar o arquivo exportado pelo Model Builder
- Inserir na pasta do plugin
- Adicionar o import no provider

```
from qgis.core import QgsProcessingProvider
from .proc_algorithm import procAlgorithm
from .teste3 import Model

class procProvider(QgsProcessingProvider):

    def __init__(self):
        """
        Default constructor.
        """
        QgsProcessingProvider.__init__(self)

    def unload(self):
        """
        Unloads the provider. Any tear-down steps required by the provider
        should be implemented here.
        """
        pass

    def loadAlgorithms(self):
        """
        Loads all algorithms belonging to this provider.
        """
        self.addAlgorithm(procAlgorithm())
        self.addAlgorithm(Model())
        # add additional algorithms here
        # self.addAlgorithm(MyOtherAlgorithm())
```

Processing Scripts no QGIS

Referência essencial:

https://docs.qgis.org/3.40/en/docs/user_manual/processing/scripts.html

```
def processAlgorithm(self, parameters, context, feedback):  
    """  
    Here is where the processing itself takes place.  
    """  
    # First, we get the count of features from the INPUT layer.  
    # This layer is defined as a QgsProcessingParameterFeatureSource  
    # parameter, so it is retrieved by calling  
    # self.parameterAsSource.  
    input_featuresource = self.parameterAsSource(parameters,  
                                                'INPUT',  
                                                context)  
    numfeatures = input_featuresource.featureCount()  
  
    # Retrieve the buffer distance and raster cell size numeric  
    # values. Since these are numeric values, they are retrieved  
    # using self.parameterAsDouble.  
    bufferdist = self.parameterAsDouble(parameters, 'BUFFERDIST',  
                                         context)  
    rastercellsize = self.parameterAsDouble(parameters, 'CELLSIZE',  
                                             context)  
  
    if feedback.isCanceled():  
        return {}
```

Leitura interessante:

https://qgis-tuts-wu.readthedocs.io/en/latest/land_degradation_development/scripts/rasterizing.html

Entradas e Saídas no Processing

Principais tipos de parâmetros:

Entradas

QgsProcessingParameterRasterLayer

QgsProcessingParameterVector-
Layer

QgsProcessingParameterNumber

QgsProcessingParameterNum-
ber.Type.Double

Saídas

QgsProcessingOutputNumber

QgsProcessingOutputVectorLayer

QgsProcessingOutputRasterLayer

[Link com parâmetros disponíveis no Processing.](#)

Acessando as Feições do VectorLayer

Dentro do método `processAlgorithm()`:

- Recuperar camada:
 - `self.parameterAsVectorLayer(...)`
- Iterar feições:
 - `for feat in layer.getFeatures():`
- Acessar geometria:
 - `feat.geometry()`

Acessando as Feições do VectorLayer

```
class Model(QgsProcessingAlgorithm):  
  
    def initAlgorithm(self, config: Optional[dict[str, Any]] = None):  
        self.addParameter(QgsProcessingParameterVectorLayer('camada_vetorial_a_ser_verificada', 'Camada vetorial a ser verificada', defaultValue=None))  
        self.addParameter(QgsProcessingParameterNumber('coordenada_de_corte', 'Coordenada de corte', type=QgsProcessingParameterNumber.Double, defaultValue=None))  
        self.addParameter(QgsProcessingParameterFeatureSink('Saida', 'Saida', type=QgsProcessing.TypeVectorPolygon, createByDefault=True, supportsAppend=True, default=QgsProcessingParameterFeatureSink.NoOptions))  
  
    def processAlgorithm(self, parameters: dict[str, Any], context: QgsProcessingContext, model_feedback: QgsProcessingFeedback) -> dict[str, Any]:  
        # Use a multi-step feedback, so that individual child algorithm progress reports are adjusted for the  
        # overall progress through the model  
        feedback = QgsProcessingMultiStepFeedback(1, model_feedback)  
        results = {}  
        outputs = {}  
  
        input_featuresource = self.parameterAsSource(parameters, 'camada_vetorial_a_ser_verificada', context)  
  
        numfeatures = input_featuresource.featureCount()  
        print(numfeatures)  
        for feat in input_featuresource.getFeatures():  
            print(feat)  
  
        bufferedist = self.parameterAsDouble(parameters, 'coordenada_de_corte', context)  
  
        if feedback.isCanceled():  
            return {}
```

```
input_featuresource = self.parameterAsSource(parameters, 'camada_vetorial_a_ser_verificada', context)  
for feat in input_featuresource.getFeatures():  
    print(feat)
```

Exemplo de Algoritmo Processing (Buffer)

```
from qgis.core import (
    QgsProcessing,
    QgsProcessingAlgorithm,
    QgsProcessingParameterVectorLayer,
    QgsProcessingParameterNumber,
    QgsProcessingParameterFeatureSink,
    QgsFeature
)

class SimpleBuffer(QgsProcessingAlgorithm):

    INPUT = 'INPUT'
    DIST = 'DIST'
    OUTPUT = 'OUTPUT'

    def initAlgorithm(self, config=None):
        self.addParameter(
            QgsProcessingParameterVectorLayer(
                self.INPUT,
                'Camada de entrada'
            )
        )

        self.addParameter(
```