APRIL 5, 2019

Approximating Geodesic Buffers with PyQGIS

When you want to buffer features that are spread across a large area (such as global layers), there is no suitable projection that can give you accurate results. This is the classic case for needing Geodesic Buffers – where the distances are measured on an ellipsoid or spherical globe. This post explains the basics of geodesic vs. planar buffers well.

QGIS lacks a way to do geodesic buffers natively. But one can approximate them by using a custom Azimuthal Equidistant projection for each point. Azimuthal Equidistant projection has the useful property that all points on the map are at proportionally correct distances from the center point. So we can write a custom processing script that implements the following algorithm for very accurate buffers – even for global layers.

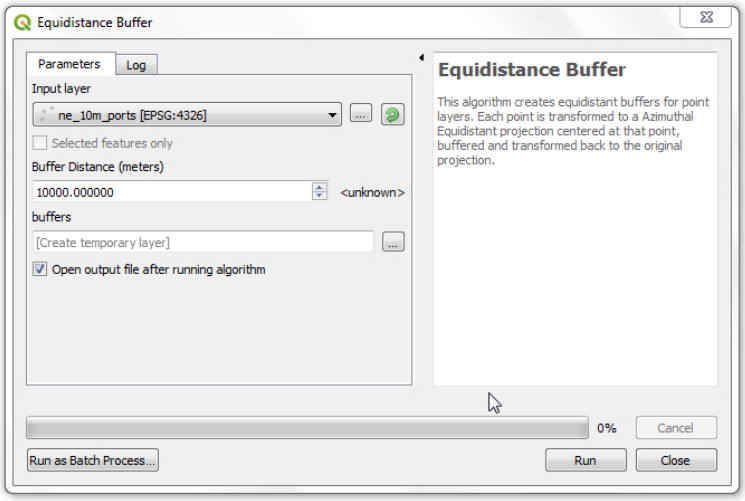
- For each feature in a layer, create a custom azimuthal equidistant projection centered at that geometry.
- Transform the geometry to the custom projection and do a planar buffer.
- Reverse transform the buffer to the original projection
- Repeat for each feature

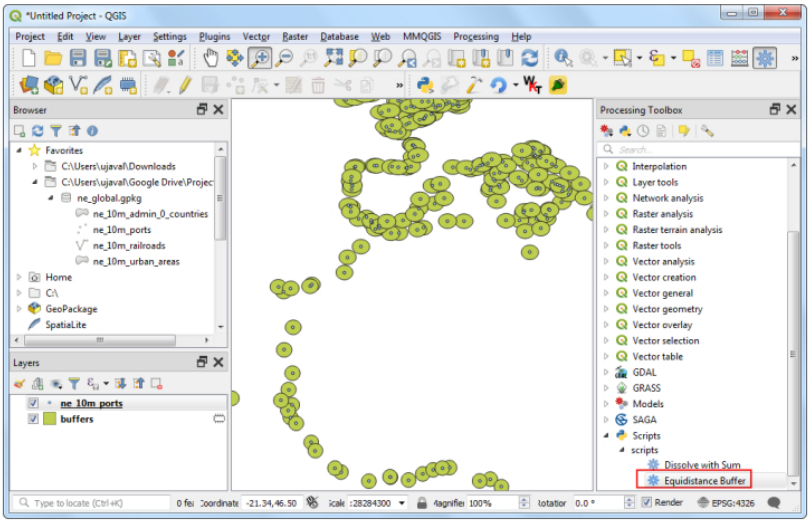
> *MMQGIS plugin has a similar implementation in the buffer tool, but the current version uses the World Equidistant Conic projection (instead of custom azimuthal equidistant projection for each feature) which is not as accurate.*

This post explains how to write a QGIS Processing script that implements this logic. It also serves as a reference for *QgsProcessingFeatureBasedAlgorithm* class which is the preferred and simplified way to implement algorithm that work on each feature independently. If you just want to use the script and are not interested in the implementation details, you can do the following and skip rest of the post

- Download the equidistance_buffer.py script
- In QGIS, open Settings → User Profiles → Open Active Profile Folder
- Copy the equidistance_buffers.py script to processing → scripts folder.

- Restart QGIS and launch the script from Processing Toolbox → Scripts → scripts → Equidistance Buffer





> *Tip: While buffering global layers, sometimes you may find that your buffer polygon is crossing the dateline, causing artifacts when displaying them in some projections. You can avoid this by splitting the buffer polygon into 2 halves at the dateline. ogr2ogr tool can do this automativcally using the* ***-wrapdateline*** *option*

Processing Script

Writing Processing Python scirpts in QGIS involves using the *QgsProcessingAlgorithm* class (see my tutorial if you are looking for an introduction). For algorithms which operate on features one-by-one, there's a class *QgsProcessingFeatureBasedAlgorithm*, which makes for a much simpler implementation. Since our problem operates on each feature independently, we can use this class instead.

QgsProcessingFeatureBasedAlgorithm requires a few different methods to be implemented for initializing, preparing and processing of the algorithm. We will look into each of them

Inputs and Outputs. There is no need to declare the input source or output sink, as these are automatically created by the class. You just need to declare what would be your output layer type, so an appropriate output sink can be created. Similarly, you can define valid input source types to filter the selection of layers that the user can

choose from. Here our input can any vector layer and the output is a polygon layer.

```
def inputLayerTypes(self):
        return [QgsProcessing.TypeVector]
def outputWkbType(self, input_wkb_type):
    return QgsWkbTypes.Polygon
```

If you need any extra parameters apart from input and output, you need to define them in the *initParameters* method. For our script, we need to take a distance input from the user.

```
def initParameters(self, config=None):
      self.addParameter(
          QgsProcessingParameterDistance(
              self.DISTANCE,
              'Buffer Distance (meters)',
              defaultValue=10000
              ))
```

We need to store the user supplied distance value in an instance variable (referred by *self.distance*) so it can be accessed when processing each feature. This is done in the *prepareAlgorithm* method which is for evaluating parameter values. We can also check if the chosen input layer is in a Geographic CRS and raise an error if it is not.

```
def prepareAlgorithm(self, parameters, context, feedback):
        self.distance = self.parameterAsDouble(
            parameters,
            self.DISTANCE,
            context)
        source = self.parameterAsSource(parameters, 'INPUT', conte
        self.source_crs = source.sourceCrs()
        if not self.source_crs.isGeographic():
            feedback.reportError('Layer CRS must be a Geograhpic C
            return False
        return super().prepareAlgorithm(parameters, context, feedb
```

Next is the implementation of the *processFeature* method. This method should contain the main logic of the script. You just need to define how a single feature should be processed, and QGIS takes care of iterating and writing the results to the output. Note that the *processFeature* method can create multiple output features, so the return value is a list. Here we implement the logic of transforming the feature geomety to a azimuthal equidistant projection, buffering it and converting the resulting geometry to the original projection.

```
def processFeature(self, feature, context, feedback):
      geometry = feature.geometry()
      # For point features, centroid() returns the point itself
      centroid = geometry.centroid()
      x = centroid.asPoint().x()
      y = centroid.asPoint().y()
      proj_string = 'PROJ4:+proj=aeqd +ellps=WGS84 +lat_0={} +lon_
      dest_crs = QgsCoordinateReferenceSystem(proj_string)
      xform = QgsCoordinateTransform(self.source_crs, dest_crs, Qg
      geometry.transform(xform)
      buffer = geometry.buffer(self.distance, self.SEGMENTS, self.
      buffer.transform(xform, QgsCoordinateTransform.ReverseTransf
      feature.setGeometry(buffer)
      return [feature]
```

Below is the complete script for reference.

```
from PyQt5.QtCore import QCoreApplication
from qgis.core import (QgsProcessing, QgsProcessingFeatureBasedAlg
    QgsProcessingParameterDistance, QgsPoint, QgsFeature, QgsGeome
    QgsWkbTypes, QgsCoordinateReferenceSystem, QgsCoordinateTransf

class EquidistanceBuffer(QgsProcessingFeatureBasedAlgorithm ):
    """
    This algorithm takes a vectorlayer and creates equidistance bu
```

```
    """
    DISTANCE = 'DISTANCE'
    SEGMENTS = 5
    END_CAP_STYLE = QgsGeometry.CapRound
    JOIN_STYLE = QgsGeometry.JoinStyleRound
    MITER_LIMIT = 2.0

    def tr(self, string):
        return QCoreApplication.translate('Processing', string)
    def createInstance(self):
        return EquidistanceBuffer()
    def name(self):
        return 'equidistance_buffer'
    def displayName(self):
        return self.tr('Equidistance Buffer')
    def group(self):
        return self.tr('scripts')
    def groupId(self):
        return 'scripts'
    def outputName(self):
        return self.tr('buffers')

    def shortHelpString(self):
        return self.tr(
            'This algorithm creates equidistant buffers for vector
            Each geometry is transformed to a Azimuthal Equidistan
            centered at that geometry, buffered and transformed ba
            original projection.')
    def __init__(self):
        super().__init__()

    def inputLayerTypes(self):
        return [QgsProcessing.TypeVector]

    def outputWkbType(self, input_wkb_type):
        return QgsWkbTypes.Polygon

    def initParameters(self, config=None):
        self.addParameter(
            QgsProcessingParameterDistance(
                self.DISTANCE,
                'Buffer Distance (meters)',
                defaultValue=10000
                ))

    def prepareAlgorithm(self, parameters, context, feedback):
        self.distance = self.parameterAsDouble(
            parameters,
            self.DISTANCE,
            context)
        source = self.parameterAsSource(parameters, 'INPUT', conte
        self.source_crs = source.sourceCrs()
        if not self.source_crs.isGeographic():
            feedback.reportError('Layer CRS must be a Geograhpic C
            return False
        return super().prepareAlgorithm(parameters, context, feedb
    def processFeature(self, feature, context, feedback):
        geometry = feature.geometry()
        # For point features, centroid() returns the point itself
        centroid = geometry.centroid()
        x = centroid.asPoint().x()
        y = centroid.asPoint().y()
        proj_string = 'PROJ4:+proj=aeqd +ellps=WGS84 +lat_0={} +lo
        dest_crs = QgsCoordinateReferenceSystem(proj_string)
        xform = QgsCoordinateTransform(self.source_crs, dest_crs,
        geometry.transform(xform)
        buffer = geometry.buffer(self.distance, self.SEGMENTS, sel
        buffer.transform(xform, QgsCoordinateTransform.ReverseTran
        feature.setGeometry(buffer)
        return [feature]
```


Posted in PYQGIS.

3 COMMENTS

LEAVE A COMMENT

Approximating Geodesic Buffers in @QGIS – GeoNe.ws

REPLY

APRIL 7, 2019 AT 1:55 AM

[…] https://spatialthoughts.com/2019/04/05/geodesic-buffers-in-qgis/ […]

★ Like

kyaw

REPLY

JULY 11, 2019 AT 9:59 PM

Having noticed that in ArcGIS users can enter distance in km or so for buffer (and selection) on features in GCS projection, tested buffering on a line extending form south 80 lat to north 80 lat to see how ArcGIS will process. ArcGIS really creates as Geodesic buffer. Then I wonder if QGIS has such functionality and stumbled on your post. Greate that you have created QGIS script for point geodesic buffer.
How to do geodesic buffer on a line or polygon in QGIS? What are to be considered in python scripts?

★ Like

DrBazUK (@DrBazUK)

REPLY

OCTOBER 30, 2019 AT 10:36 PM

Thank you! This python script and blog post has save me hours of headscratching and means that I can buffer without having to reproject every polygon I have which is set to WGS84 (EPSG 3857 or 4326). This has been a godsend.

Eventually I'll get to the point of scripting the changes I need and will use the second half of the blog post!

★ Like

Jesse

REPLY

JANUARY 17, 2020 AT 8:41 PM

Excellent post. Super helpful script, and an insight into writing very efficient QGIS code. Thank you!

★ Like

LEAVE A REPLY

Enter your comment here...