# User Manual for numgeo (0.0.1a0)

Numgeo is a package of Python modules that primarily supports skeletonization (at present).

## Quick start

If you're keen to get started, consider the following code (*but don't run it just yet!*):

```
from numgeo.scripts.skel import process
process(r"path/to/polygon_shapefile.shp", interval=0.1, min_normalized_length=0.)
```

`interval`, in effect, specifies the desired resolution of the output skeleton, in map units (e.g., meters). If the narrowest constriction in your input polygons is $x$ map units, you should specify `interval` $< 0.5x$.

`min_normalized_length`, in effect, specifies how simple you want the output skeleton to be, with higher values corresponding to greater simplicity. Any value less than 1. results in no simplification, so the "raw" skeleton itself is output, which may have a lot of unwanted bits.

Therefore, if you were to run the example code, you'd derive a "raw" skeleton at a resolution of 0.1 m (if your map unit is one meter). Because you did not specify an output path, the skeleton would default to *path/to/polygon_shapefile_skel.shp*.

### Quick tips:

1. It is *highly* recommended that you read Installation before installing numgeo.
2. Processing time and memory use both increase exponentially with progressively smaller `interval` values, so be careful when choosing a value.
3. You might consider calling `process(...)` a few times, each time with different `interval` and `min_normalized_length` values, to build an intuition for how these arguments affect the output skeleton.
   - In that case, you can specify `out_path_prefix` to avoid overwriting previous outputs. For example, `out_path_prefix="A_"` would output to *path/to/A_polygon_shapefile_skel.shp*.

## Installation

Ideally, installing numgeo would be as easy as running the following from a console:

```
pip install numgeo
```

This would install numgeo and all of the modules it depends on. If any of those modules were already installed, it would upgrade them if necessary to ensure that numgeo would be supported. Unfortunately, you may well find that things are not that simple.

### Managed distributions

If your Python environment is a managed distribution, such as Anaconda or Enthought Canopy, pip commands can create problems. As just one example, such managed distributions keep track of what modules are installed and can make installation of tricky packages much easier. However, if you install numgeo via pip, pip may then install or upgrade other modules, possibly even corrupting the ecosystem that managed distributions are meant to maintain. If you use a managed distribution, you are strongly encouraged to read its documentation before installing numgeo. You should also consider using the manual option.

## Prepackaged distributions

Prepackaged distributions, such as OSGeo4W and the Python environment that ships with ArcGIS, come preinstalled with a rich set of useful modules but (unlike managed distributions) may not provide full support for the managed installation of modules outside that set. Similar to managed distributions, installing such modules or even upgrading already installed modules could potentially corrupt the carefully constructed ecosystem of the distribution. Therefore, you should read all relevant documentation and consider using the manual option.

## Gdal challenges

Gdal is a powerful module, and numgeo relies on it (by default) for all of its reading and writing. However, gdal can be tricky to get installed and working. If you run into any problems, or (wisely) just want to do your homework before installing gdal, you may find the following links helpful:

https://gis.stackexchange.com/questions/9553/installing-gdal-and-ogr-for-python

https://gis.stackexchange.com/questions/2276/installing-gdal-with-python-on-windows

## The manual option

One option that is often safer than a straightforward pip install is to manually install any tricky dependencies first. For example, if you're using a managed distribution, you could use its built-in tools to install whatever dependencies are supported by those tools. If gdal or other tricky dependencies are not supported, you could read up on how best to install each of those dependencies and proceed to do so. Alternatively, you could find a prepackaged geospatial distribution to your liking that already has as many of the tricky dependencies preinstalled as possible, and then proceed to install each remaining tricky dependency (if any) manually, after doing your homework on how best to do that. In any case, after all the tricky dependencies are installed, you could execute

```
pip install numgeo --upgrade --upgrade-strategy only-if-needed
```

or, if every required dependency has been installed (not just the tricky ones),

```
pip install numgeo --no-dependencies
```

## Dependencies

The current version of numgeo depends on the following packages and modules, as indicated:

1. numpy (can be tricky; required)
2. scipy (depends on numpy; can be tricky; required)
3. gdal (can be tricky; required)

4. psutil (optional but recommend)
5. bottleneck (optional but can improve performance)
6. arcpy (commercial; optional)

### Arcpy

Arcpy is the Python module that ships with the ArcGIS commercial software made by ESRI. Numgeo will use gdal by default for all reading and writing. However, if gdal does not support an operation, numgeo will resort to using arcpy for that operation, if it is installed. Most notably, numgeo will use arcpy for write access to file geodatabases if gdal does not support this access. For existing ArcGIS users, this may avoid the potentially tricky task of enabling write access to file geodatabases in gdal.

### Write access to file geodatabases in gdal

A basic installation of gdal includes the OpenFileGDB driver, which enables read access to file geodatabases. However, write access to file geodatabases can only be enabled in gdal by installing the FileGDB driver, which can be tricky. If you choose to install the FileGDB driver, the following links may prove helpful:

https://gis.stackexchange.com/a/292542

https://glenbambrick.com/2017/03/10/setting-up-gdal-with-filegdb/

## Supported formats

At present, numgeo supports two formats: shapefiles and file geodatabases. A basic installation of gdal supports read/write access for shapefiles but only read access to file geodatabases. Enabling gdal support for write access to file geodatabases takes a bit of extra effort. Alternatively, if available, numgeo can resort to arcpy.

## Numgeo modules

### scripts (`numgeo.scripts`)

Contains submodules, each of which serves as a demonstration of the relatively mature functionality of top-level modules. Each submodule, in turn, contains no more than a few top-level functions, each of which supports high-level processing. *Note: This is the only top-level module that most users are expected to need.*

#### scripts.skel (`numgeo.scripts.skel`)

[*Maturity: Release Candidate*] Demonstration code to support high-level skeletonization via the functions `numgeo.scripts.skel.process` and `numgeo.scripts.skel.process.external`.

### geom (`numgeo.geom`)

[*Maturity: Alpha*] Defines geometric types (e.g., `Polygon`) that are used internally for geometric operations (e.g., testing whether a point lies in a polygon). At present ,functionality is narrowly focused on supporting skeletonization.

### opt (`numgeo.opt`)

[*Maturity: Alpha*] Defines package-level user options.

### skel (`numgeo.skel`)

[*Maturity: Beta*] Provides low-level functionality to support skeletonization. This module is well documented and serves as a reference (1) for the details of how skeletonization is implementation within numgeo and (2) of the options available for advanced skeletonization, including arguments that may be passed to `numgeo.scripts.skel.process` (see `numgeo.skel.EasySkeleton`).

### util (`numgeo.skel.util`)

[*Maturity: Alpha/Beta*] Provides core functionality used by multiple top-level modules. Private functions and classes have names that begin with an underscore and are intended for internal use only. They may be poorly documented or undocumented, may have very narrow intended scopes (outside of which, behavior is arbitrary), and may be modified between versions without any consideration for backward compatibility. Public functions have names that do not begin with an underscore and are intentionally exposed for external use. They are well documented, with clearly defined scopes and intended behavior, and will not be modified arbitrarily between versions.

# Skeletonization

## Introduction

### What's a skeleton?

Consider the outline of a river. This outline has a polygonal shape. For example, it has a measurable width. However, we can also think of this river as a (squiggly) line, similar to how major rivers are represented on a globe. That squiggly line is effectively the skeleton of the river outline. In other words, "skeletonization" collapses an elongate polygon to a linear representation.

### Why do skeletons matter?

The present algorithm is designed for use in geomorphology, a branch of geology that studies landforms, though the algorithm likely has value well outside that field. In geomorphology, the skeleton of a river, often called a "center line", is used in a wide range of geometric measurements, including sinuosity (which measures how squiggly the river is) or the radius of curvature (which measures how tightly the river turns). It also can be used as a coordinate axis to measure distance downstream, orient the cross-river direction for measuring the local width, etc.

### The Algorithm, in brief

The algorithm builds on a long history of work by others. It also has a lot of moving parts, which are explained in greater detail further below, but these are the highlights:

1. Sample points along the boundary of an input polygon, including around any holes.
2. Compute a Voronoi diagram. (https://en.wikipedia.org/wiki/Voronoi_diagram)

3. Isolate the "graph skeleton" from the other extraneous bits in the Voronoi diagram (which together make up the graph skeleton's "complement"). For example, each hole in polygon has its own skeleton that should be discarded.
4. "Partition out" paths from the graph skeleton to incrementally construct the "partitioned skeleton".
    - This is a bit like moving Lego blocks (segments) from one toy box (skeleton) to another, stacking some of them together during the transfer (to form continuous paths of many segments).
5. Optionally add "tails" to the paths so that they extend to polygon's boundary rather than stopping short.
6. Optionally prune away undesired paths to simplify the skeleton and remove noise.
    - This is accomplished using the normalized length metric.

## Phase 1

The main goal of this phase is to generate a raw "graph skeleton" that is organized like a network (or formally, a graph). The phase follows a classic procedure that well predates numgeo, though some implementation details may be unique. This phase includes steps 1–3 as enumerated above.

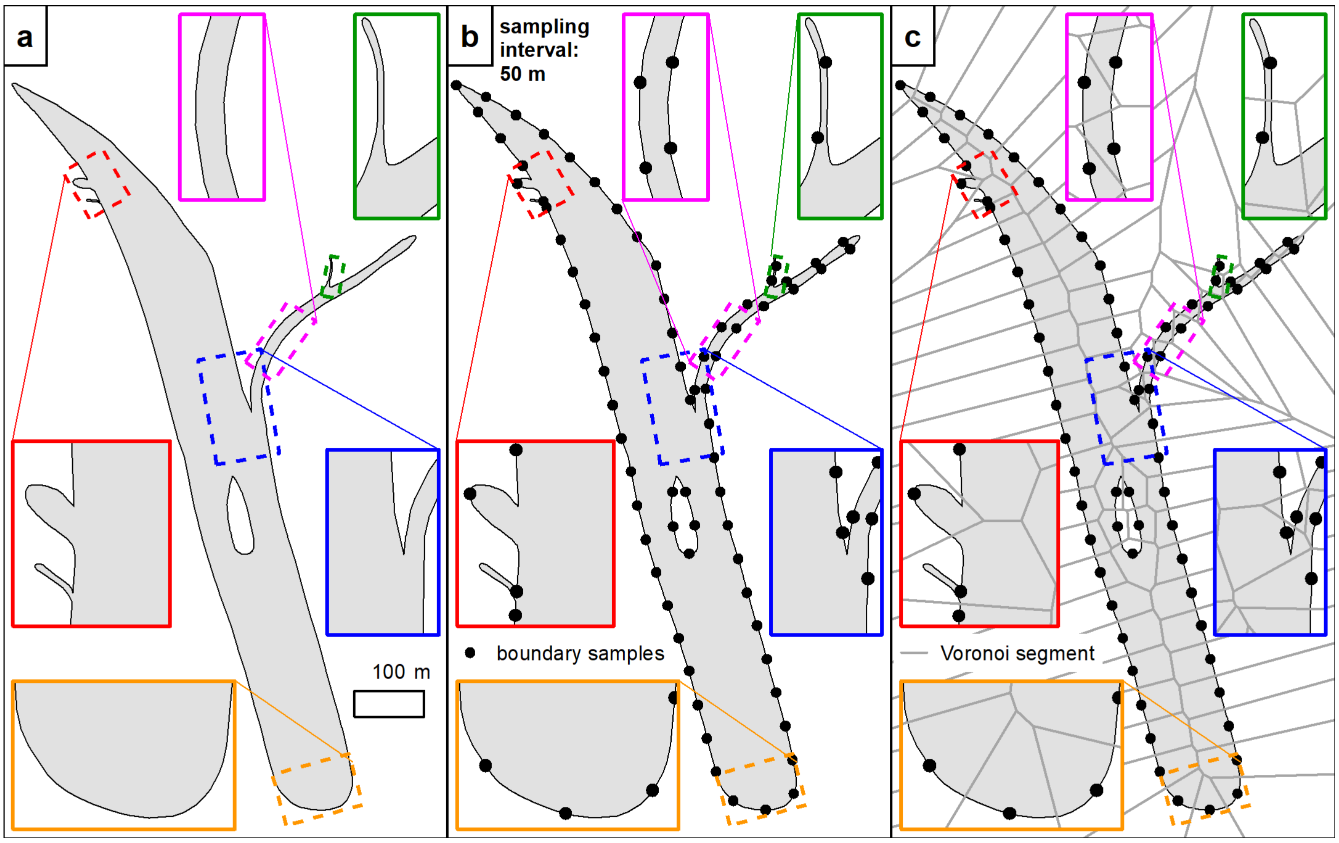

**Fig. 1.** *a) An example input polygon. b) Boundary samples. c) Voronoi diagram.*

### Boundary sampling

The boundary of the input polygon is sampled at a regular interval, specified by `interval`, including the boundary around each hole, if any (**Fig. 1b**). Ideally, if the narrowest constriction in your input polygons is $x$ map units, you should specify `interval` $< 0.5x$. (Otherwise, the boundary is considered "under-sampled".) Ultimately this boundary sampling interval acts like a resolution, with finer sampling (smaller `interval`) yielding a smoother skeleton that is better centered (**Fig. 2**). However, memory use grows exponentially with finer sampling interval, so keep that in mind!

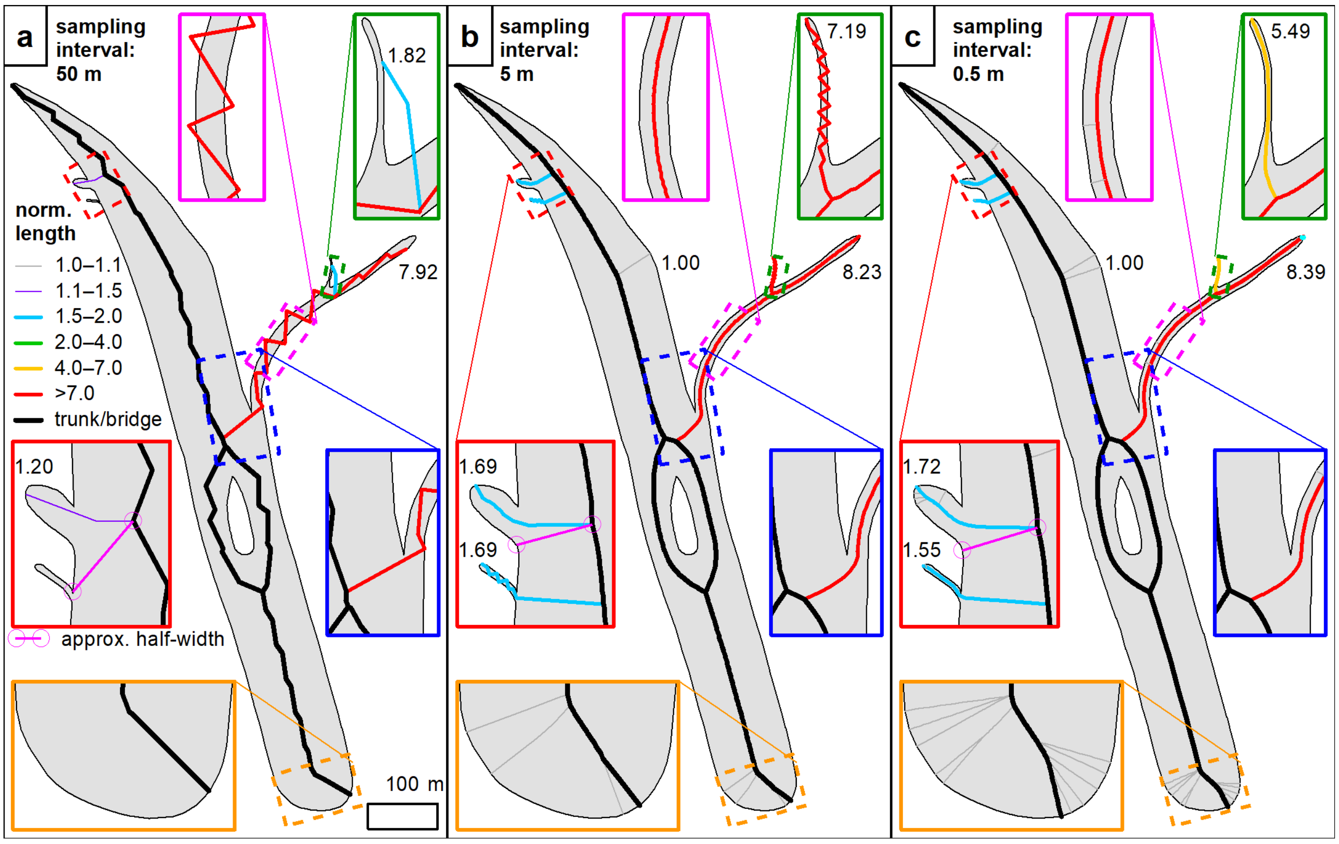

**Fig. 2.** *Note, especially in insets, how smoothness and centeredness of partitioned skeleton increase with fining sampling interval. The accuracy with which the normalized length scores the relative importance of branches also increases with fining sampling interval (center-left (red) and top-right (green) insets).*

## Voronoi analysis

The Voronoi diagram for these boundary samples is then generated (**Fig. 1c**). The Voronoi diagram (Wikipedia link) describes, for each boundary sample, the polygonal neighborhood in which all enclosed coordinates are closer to that boundary sample than to any other boundary sample. We'll call each segment that forms a facet of these polygon neighborhoods a "Voronoi segment" for brevity.

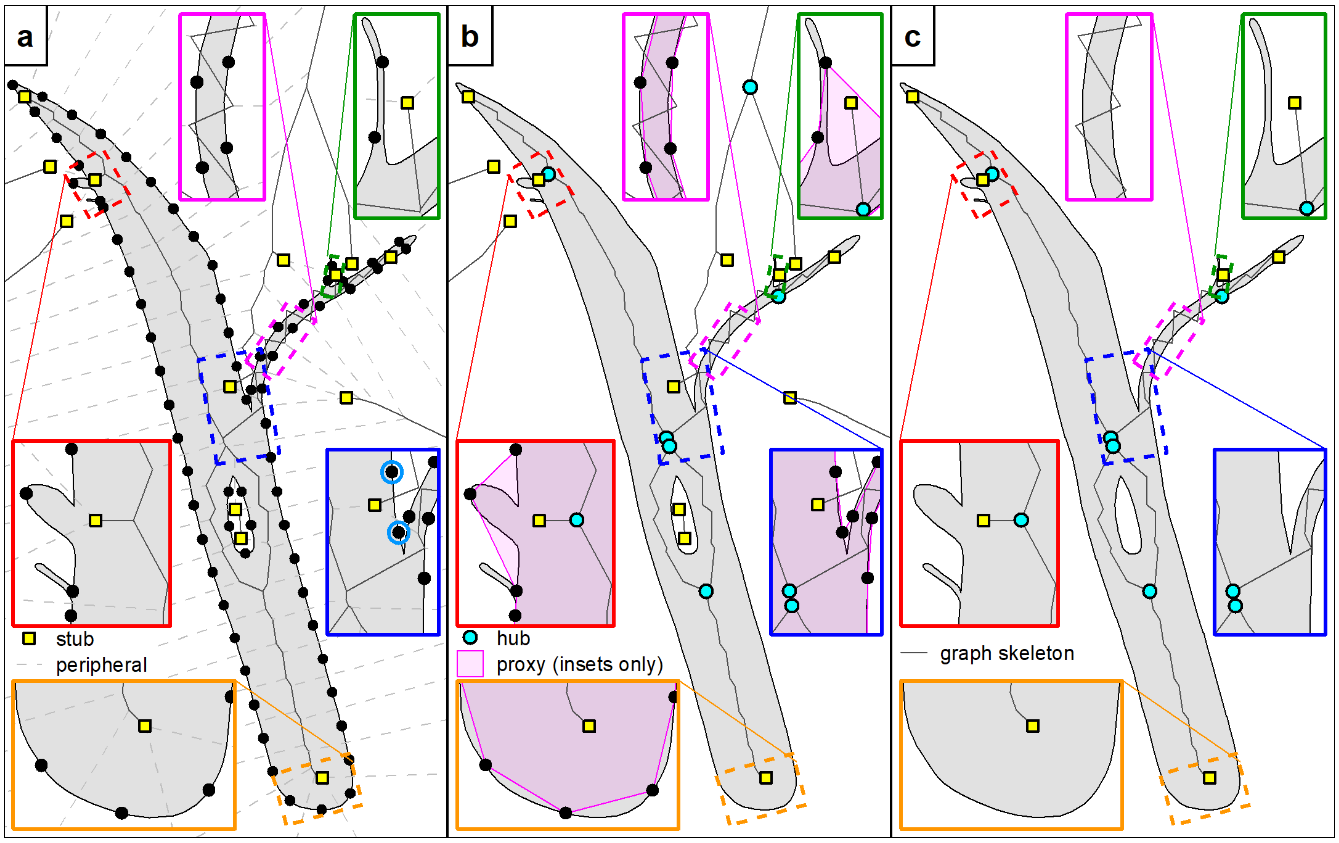

**Fig. 3.** *a) The skeleton (mostly within the input polygon) and its complement (mostly outside the input polygon) are separated by "peripheral" Voronoi segments. The skeleton exits the input polygon in narrow places (top-center (magenta) and top-right (green) insets), and the complement enters the input polygon in a tight corner (between blue-encircled boundary samples in center-right (blue) inset). b) Whether or not hubs are contained within the proxy polygon discriminates the skeleton from its complement. c) The isolated graph skeleton.*

## Graph skeleton isolation

Approximately, the graph skeleton is composed of all Voronoi segments that are completely contained by the input polygon, with all Voronoi segments completely outside the input polygon forming the skeleton's complement and the remaining segments, which intersect the input polygon's boundary, constituting "peripheral" segments (**Fig. 3a**). However, if the input polygon locally narrows to less than double the sampling interval (`interval`), even the raw skeleton may exit the input polygon (**Fig. 3a**). To partially accommodate such coarse sampling (which may be required by memory constraints), the algorithm adopts two (heuristic) strategies:

1. Containment is tested against a "proxy polygon" (rather than the input polygon*), which is an approximation of the input polygon whose vertices are boundary samples (**Fig. 3b**).
    - In essence, the proxy polygon represents the polygon "seen" by boundary sampling.
    - *In the special case that the smallest inter-vertex segment of the input polygon has a length greater than the sampling interval, the input polygon is used.
2. The graph skeleton is identified as that connected component (network, or graph) which has the greatest number of hubs (intersection nodes) contained by the proxy polygon (**Figs. 3b,c**). In addition, *all* of its hubs must be so contained, or graph isolation fails*.
    - In testing, evaluating hub containment against the proxy polygon, rather than the input polygon, tended to better distinguish the raw skeleton from its complement, which is essential to Phase 2.
    - *Unless the cutting heuristic is used.

## Phase 2

The main goal of this phase is to generate a "partitioned skeleton" that is made up of paths. Each path is a series of links (or formally, edges) from the graph skeleton that have been "glued" together. The purpose of the partitioned skeleton is to support simplification of the raw graph skeleton from Phase 1, including the removal of extraneous bits. This phase is largely novel and implemented in numgeo for the first time. It includes steps 4–6 as enumerated above.

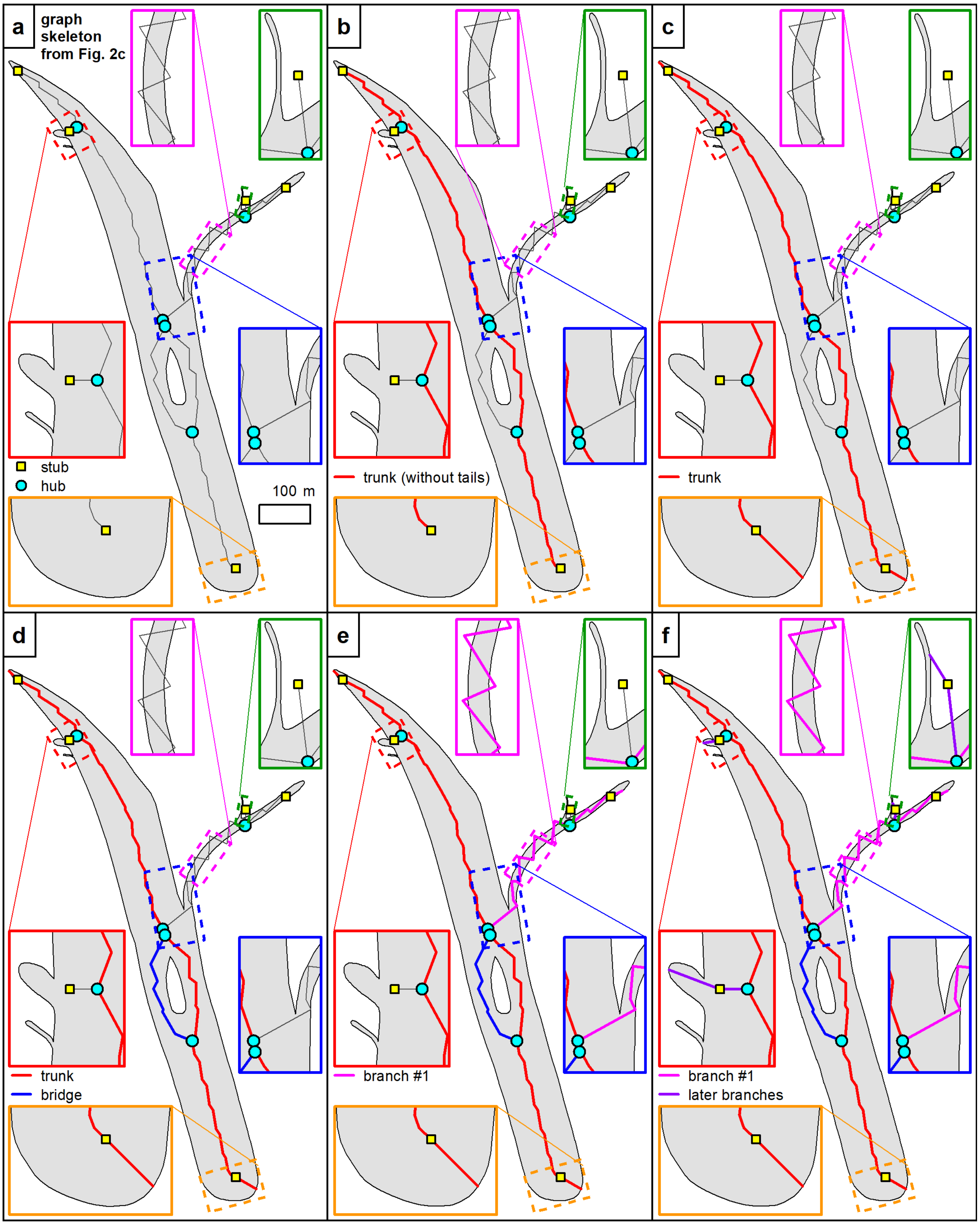

**Fig. 4.** *"Partitioning out", in which the graph skeleton is iteratively segmented into paths.*

## Partitioning out

This step is iterative. On each "partitioning out" iteration, a length-optimized path between (the remaining) graph skeleton nodes is identified, its constituent links (edges) removed from the graph skeleton, and the path deposited into the partitioned skeleton as a single unit (**Fig. 4**). By default, the iterations have the following order, which is designed to approximate the partitioning out that a human might expect (e.g., what bits should be designated part of a bridge rather than part of a branch):

1. Partition out the longest stub–stub path as the "trunk" (**Fig. 4b**).
    - A "stub" is an end node.
2. Partition out each hub–hub path as a "bridge" (**Fig. 4d**).
    - A "hub" is an intersection node.
    - Throughout partitioning out, the identity of a node as a hub or stub is determined by the geometry it would have within the the partitioned skeleton if it were partitioned out, which will often differ from its geometry within the remaining graph skeleton.
3. Partition out the longest hub–stub path as a "branch" (**Fig. 4e**).
    1. If the branch does not satisfy user criteria (e.g., `min_normalized_length`), it may be immediately pruned.
4. Repeat #2, then #3 (**Fig. 4f**).
5. Repeat #4 until no more paths can be partitioned out.
    - Partitioning out ends when either every link (edge) from the graph skeleton has been partitioned out, or whenever all links (edges) that remain in the graph skeleton could not possibly satisfy the constraints specified by the user.

Note: For improved performance, branches and bridges that can be trivially identified are also partitioned out between the formal steps enumerated above. A branch or bridge can be trivially identified if it is completely detached within the graph skeleton but would touch the partitioned skeleton if it were partitioned out.

## Tail addition

Because the graph skeleton was isolated by first removing peripheral Voronoi segments, its ends generally* stop short of the input polygon's boundary (**Fig. 4b**). To extend the ends of the partitioned skeleton to this boundary, a "tail" segment is added to each stub (**Fig. 4c**). The tail's opposite end is a boundary sample . This boundary sample is somewhat arbitrarily chosen but, like the skeleton itself, is "correct" within the effective resolution determined by the sampling interval. A tail is added to each path immediately after it is partitioned out, as the addition of a tail modifies the calculated normalized length.

*Strictly, an end can extend beyond the input polygon's boundary due to noise, likely because the local width of the polygon is much smaller than the sampling interval. (This is why graph skeleton isolation is based on hubs rather that stubs.) However, a tail is added even in such cases, which may be preferable (**Fig. 4f, top-right (green) inset**).

Tail addition can be disabled by specifying `tails=False`, which is a `numgeo.skel.EasySkeleton` argument and therefore supported by the `numgeo.scripts.skel.process` function.

## Pruning

Each path that does not satisfy the criteria specified by the user is pruned. When a path is pruned, it is removed from the partitioned skeleton and permanently discarded, along with all the paths that depend on it. For example, if a particular branch is pruned, any branches that stem from that branch are likewise pruned. This ensures that the partitioned skeleton remains a single connected unit.

Why is pruning important? Unless the boundary of the input polygon is extremely smooth, the raw skeleton will include many bits that are generally considered noise and called "spurious branches" in the technical literature. It is not uncommon for these spurious branches to make up most of the length and the vast majority of the partitioned-out branches if no pruning is enabled (**Figs. 5, 6**). Therefore, a prerequisite for useful, largely automated geomorphic skeletonization is some means to facilitate removal of these spurious branches, such as a pruning criterion.

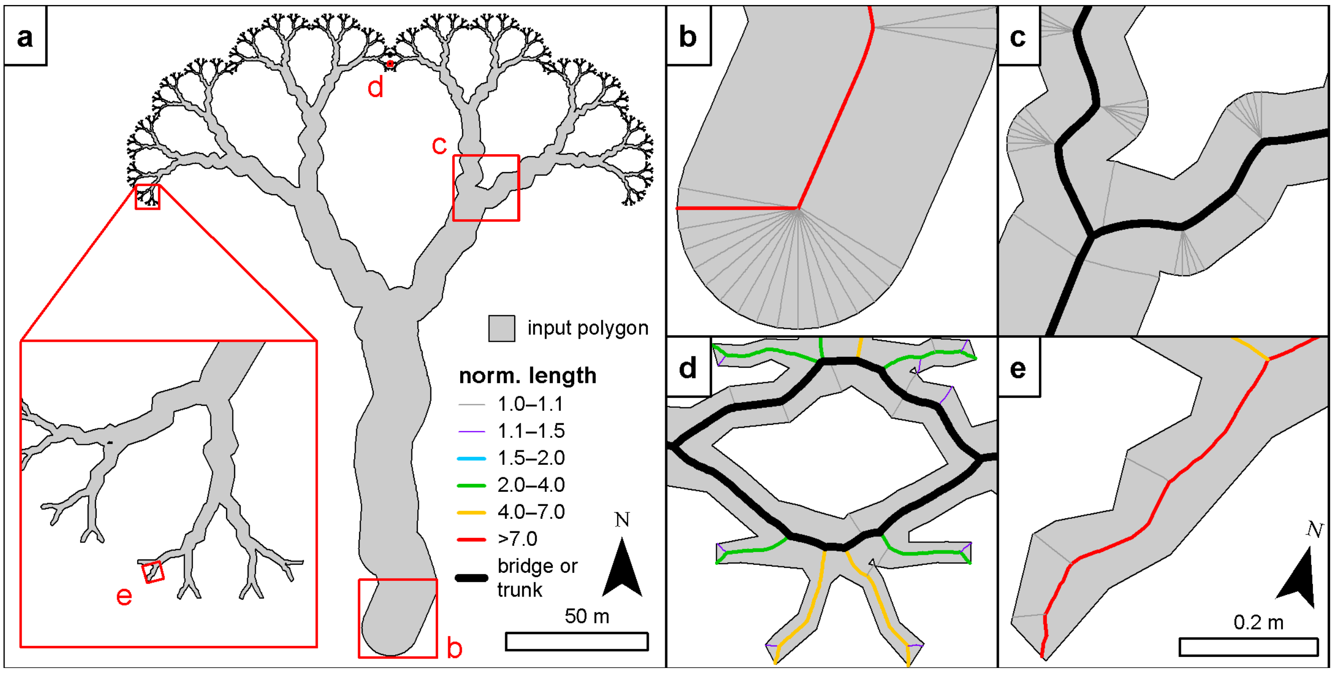

**Fig. 5.** *a) Synthetic input polygon (and key). b)–e) Note abundant spurious branches (66.7% of total length, 4646 of 5141 partitioned-out branches), all of which have normalized length <1.38, whereas all non-spurious branches have normalized length >2.56.*

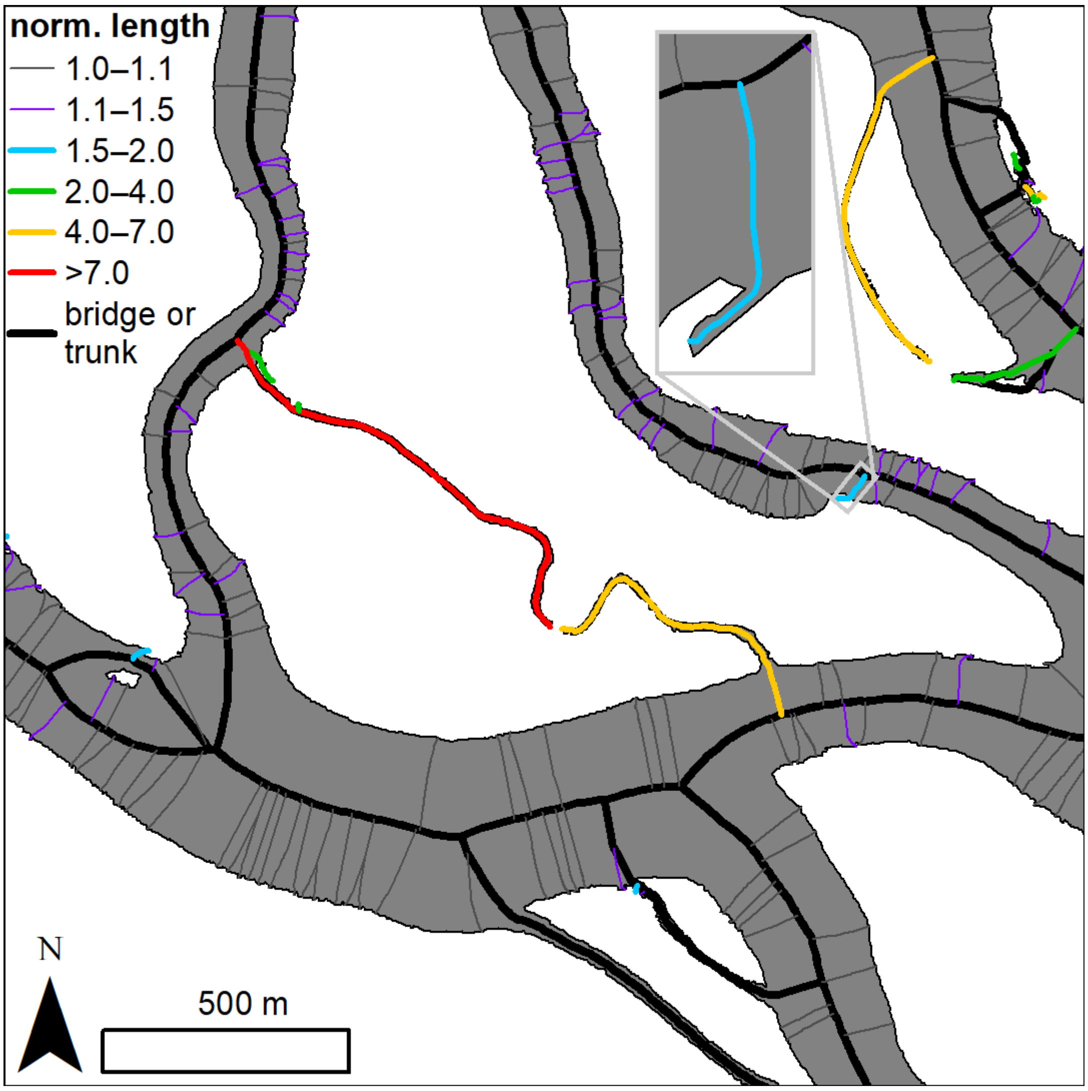

**Fig. 6.** *Raw partitioned skeleton of an intentionally noisy and artifact-ridden input polygon. The input polygon was generated by thresholding a 1 m/pixel raster of the Yukon River, Alaska, USA (center 145.092° W, 66.433° N). Note how normalized length reasonably approximates branch importance. Spurious branches would be even more abundant if not for templating at a coarse sampling interval of 15 m.*

#### Normalized length

Probably the most generally useful pruning criterion (as explained in the following paragraphs) is the minimum permitted normalized length (specified by `min_normalized_length`). The normalized length, which is only defined for branches, is a branch's length divided by half the approximate width of the input polygon at the branch's stem (hub) end. More precisely, the half-width at the stem is approximated by the distance from the branch's stem (hub) node to the nearest boundary sample (**Fig. 2, center-left (red) insets**). (By definition of the Voronoi diagram, there are always exactly three nearest boundary samples, each at the same distance.) Naturally, this value better approximates the local half-width at finer sampling intervals. (The use of this approximated half-width, rather than the true half-width, is merely for performance. The nearest boundary sample is known from earlier Voronoi analysis and therefore the approximation can be made very efficiently.)

The purpose of the normalized length is to approximately score the importance of a branch (**Figs. 5, 6**), which (as often interpreted) depends on both its length and the scale of the local geometry. Note that a branch must at least span the local half-width, and therefore the normalized lengths of branches are invariably >1 (unless tail addition is disabled). Because spurious branches frequently have a normalized length very near 1, the normalized length facilitates their identification and the minimum permitted normalized length pruning criterion (`min_normalized_length`) can be used to facilitate their removal. Moreover, if still further simplification of the skeleton is desired, the normalized length can similarly be leveraged to facilitate identification and removal of the least important (non-spurious) branches. The accuracy with which the normalized length scores the relative importance of branches increases with fining sampling interval (**Fig. 2, center-left (red) insets**).

#### How to choose the "right" normalized length

The normalized length may be the most important innovation of the implemented algorithm, and it is a major reason for the entire partitioning out procedure. However, realizing its potential often depends on identifying some critical normalized length for your particular application. If you're unsure of how to go about that, consider the following strategies:

1. Output a raw skeleton by enabling no pruning criteria. Symbolize branches based on normalized length (which is stored in the *NormLength* field if `numgeo.scripts.skel.process` is used) and visually determine what normalized length appears to be critical for your needs (**Figs. 2, 5, 6**).
    - Then either regenerate the skeleton with that critical value specified for `min_normalized_length` or else manually delete branches with smaller normalized lengths, being careful to also delete any regions that become detached by those manual deletions.
2. Output a skeleton at each of several minimum permitted normalized length values and determine which best achieves your desired level of simplification.
3. Reason what critical value is best for your particular application, based on the geometric definition of the normalized length.

#### Other pruning criteria

*Warning: The functionality described in this subsection has not been as thoroughly tested as* `min_normalized_length`.

Additional pruning criteria are supported by keyword arguments to `numgeo.scripts.skel.process`, which are passed to `numgeo.skel.EasySkeleton.__init__`. See that method's documentation for details. If you're comfortable with programming and desire still lower-level functionality, note that each graph skeleton link (edge) and partitioned skeleton path is a `numgeo.skel.SkeletalLineString2D` instance that supports some useful attributes. See the `numgeo.skel.EasySkeleton.get_lines` method for details.

## Memory use: considerations and heuristics

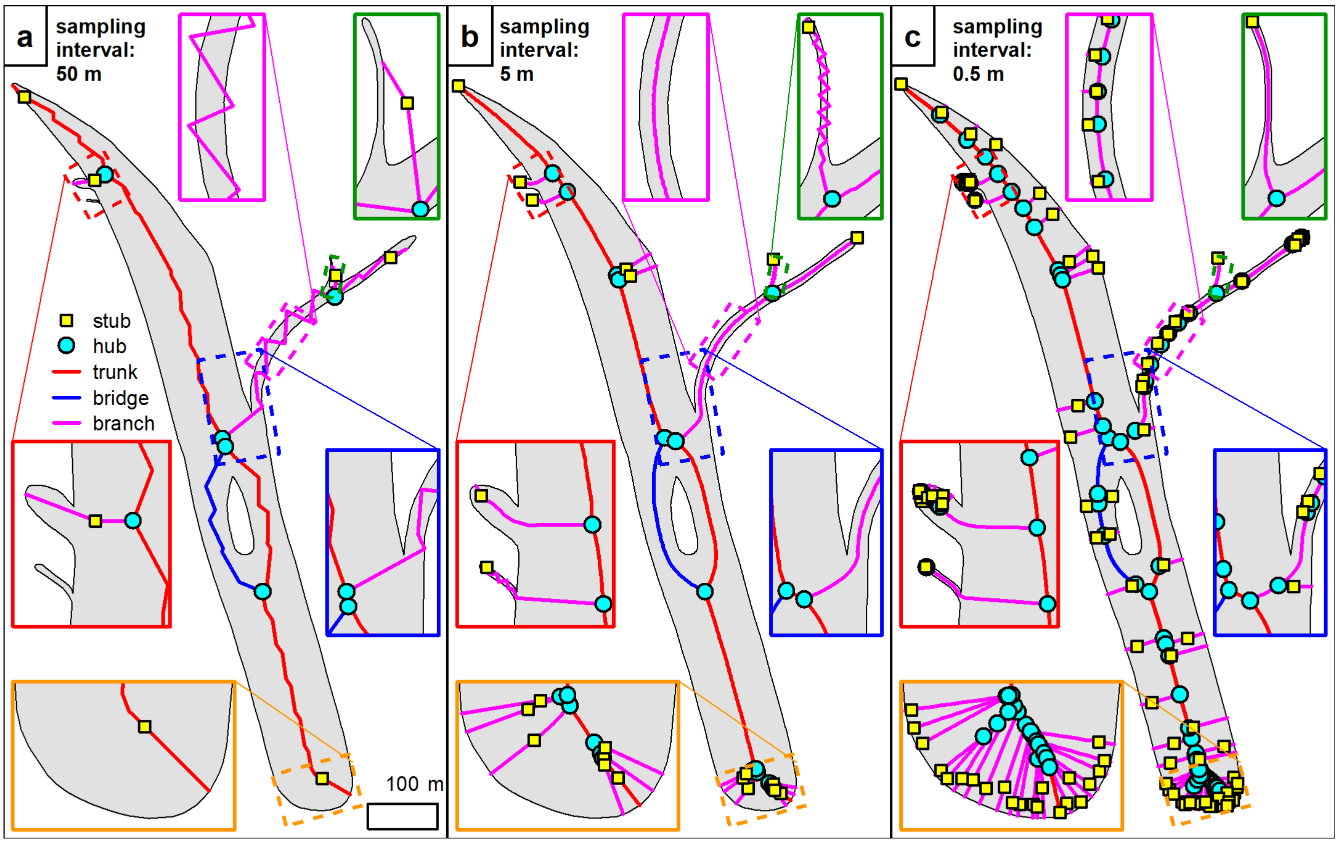

**Fig. 7.** *The total number of hubs and stubs increases with fining sampling interval in a way that cannot be predicted.*

## Bottlenecks

There are two points during processing at which a memory bottleneck may occur:

1. During generation of the Voronoi diagram.
   - Footprint scales approximately linearly with the number of boundary samples.
   - Footprint can be reasonably estimated before any significant processing.
2. During partitioning out of the trunk (specifically, when the graph skeleton is solved in a graph traversal sense).
   - Footprint scales approximately with the total number of hubs and stubs to the second power.
   - Footprint cannot be reasonably estimated until hubs and stubs are identified midway through processing (**Fig. 7**).
   - Except for very simple input polygons, this bottleneck is usually the limiting factor.

To make skeletonization as easy as possible on the user, processing is halted as soon as it is anticipated that available memory may not accommodate completion of processing. (This is the default, and recommended, behavior. However, overriding this safeguard is supported via the `isolation_mode` argument that may be passed by keyword to `numgeo.scripts.skel.process`. See documentation for `numgeo.skel.Skeleton.__init__` for details.)

## Templating

To retain the small memory footprint of a coarse sampling interval (large `interval`) but partially recover the geometric advantages (**Fig. 2**) of a fine sampling interval (small `interval`), consider templating. This (heuristic) technique (**Fig. 8**) essentially simplifies a skeleton generated at some fine sampling interval (specified by `interval`) until it approximates the general shape of a skeleton generated at some coarse sampling interval (specified by `template_interval`). Although the gross shape of the final hybrid skeleton will only be as complete as the coarsely-sampled skeleton, its local geometry will have the smoothness and centeredness of the finely-sampled skeleton.

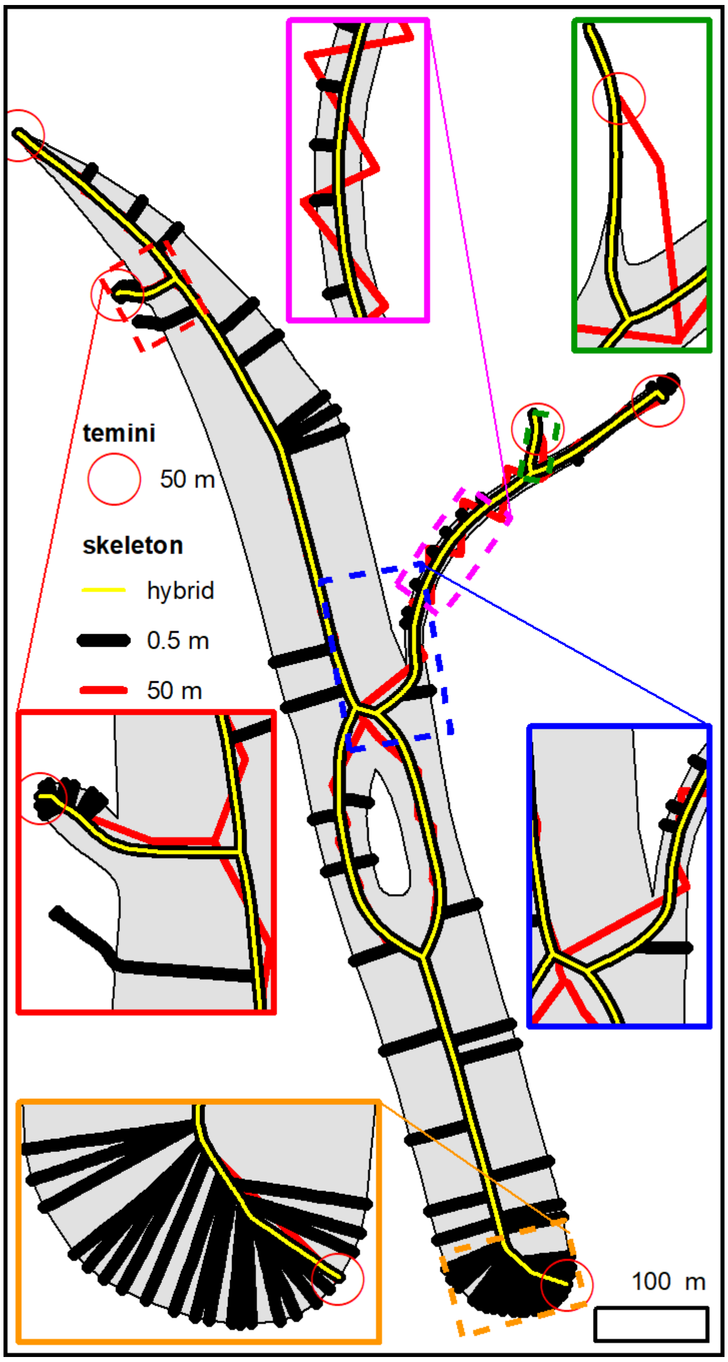

**Fig. 8.** *Templating example. Skeleton finely sampled at 0.5 m is simplified such that its termini approximately correspond to the termini of a skeleton coarsely sampled at 50 m (red circles). The resulting hybrid skeleton is smooth and centered like the finely-sampled skeleton but only as complete as the coarsely-sampled skeleton, namely, the lower protrusion in the center-left (red) inset is absent in the hybrid skeleton.*

If the specified coarse sampling interval (`template_interval`) is still too fine in combination with the specified fine sampling interval (`interval`) for their combined processing to be accommodated within the targeted memory footprint (specified by `targ_GB`), branches in the coarsely-sampled skeleton will be discarded in order of increasing normalized length until processing can be so accommodated. The smallest normalized length that was retained, called the "template normalized length cutoff", is reported in the *TmplNLXOff* field if `numgeo.scripts.skel.process` is used.

By default, if standard processing at the fine sampling interval (`interval`) would not exhaust the targeted memory footprint (`targ_GB`), the coarse sampling interval (`template_interval`) is ignored. (This behavior can be overridden. See documentation for the `template_interval` argument of `numgeo.skel.EasySkeleton`.) Note that the targeted memory footprint defaults to ~80% of the memory available on the system at the start of processing.

If templating is resorted to and the coarse sampling interval (`template_interval`) is anywhere greater than half the input polygon's width, as is usually the case, you may have to enable cutting to complete processing.

## Cutting

Cutting is a (heuristic) technique of last resort intended to accommodate coarser sampling than would otherwise be possible. In particular, cutting is useful wherever the input polygon narrows to less than double the sampling interval. At such "under-sampled" locations, two scenarios can occur, each of which violates the expectations that underlie the standard procedure for isolation of the graph skeleton:

- Scenario 1: hubs that ought to belong to the skeleton lie outside the proxy polygon
- Scenario 2: hubs that ought to belong to the skeleton's complement lie within the proxy polygon

If either of these scenarios occur, the standard procedure for isolating the graph skeleton may fail. Then, *and only then*, if cutting is enabled, "problematic" hubs are identified and permanently discarded, and then all links (edges) that terminate at the now deleted hub are also discarded. These problematic hubs are those that lie outside the proxy polygon but whose association with the skeleton is ambiguous. Consequently, *all Scenario 1 hubs will be excluded from the graph skeleton*, whether by cutting or because, due to the the noise of under-sampling, they appear associated with the complement. In either case, important portions of the true skeleton could become detached and fail to be represented in the graph skeleton. Additionally, although intentional side-effects of cutting may correctly exclude from the graph skeleton some Scenario 2 hubs , it is possible that some Scenario 2 hubs will survive cutting. In that case, graph isolation will fail, or in a worst-case scenario, graph isolation may appear to succeed but the graph and partitioned skeletons will in fact correspond to part of the complement.

Due to the potential hazards of cutting just described, you are *strongly encouraged to manually vet the output skeleton at whatever level of detail you require for your application whenever cutting is triggered*. That said, cutting is implemented in numgeo precisely because it generally works surprisingly well, especially when the sampling interval is not exceedingly coarse.

## Footprint optimization

It is not possible (even for the designer of this algorithm!) to reasonably guess the memory footprint required to skeletonize a given input polygon at a given sampling interval (except for very simple input polygons, such as that in **Fig. 1a**). For that reason, the user can specify the sampling interval implicitly by using the value zero (i.e, `interval=0.`). In that case, a recursive search is triggered that will identify, to within a factor of two, that sampling interval that would exactly consume the targeted memory footprint (`targ_GB`), which defaults to ~80% of the memory available on the system at the time that the search is initiated. Skeletonization then proceeds at that sampling interval.

Note that because the memory footprint depends exponentially on the sampling interval (except for very simple input polygons), the actual memory footprint of processing at the "optimized" interval can be much less than half of the targeted memory footprint. Furthermore, note that, as currently implemented, the recursive search for an optimized interval is best suited for complex polygons, for which the functionality is presumed to be most helpful. *If the input polygon is simple, the search may take a prohibitively long time.* Even for complex polygons, the search may take a considerable amount of time, as many recursions require a full execution of Phase 1.

The coarse sampling interval used in templating (`template_interval`) also supports optimization, and both the fine and coarse sampling intervals can be optimized together (i.e., `interval=0., template_interval=0.`). After optimization, there may still be, by design, a (nonzero) template normalized length cutoff. For a detailed explanation of how special values for each interval argument are interpreted, see the documentation for the `template_interval` argument of `numgeo.skel.EasySkeleton.__init__`.

## Additional strategies

In addition to templating, additional strategies that you might consider to reduce the memory footprint include

1. Remove unimportant (small?) holes from the input polygon.
    - More generally, you should always clean the input polygon prior to skeletonization, removing any holes, narrow protrusions, or other geometries that you do not want considered in skeletonization.
2. Manually chop the input polygon into smaller polygons, and skeletonize each polygon individually.
    - The biggest drawback here is that you will have to knit these skeletons back together. That knitting requires manual effort and compromises the objectivity and reproducibility of the result.
3. Smooth the input polygon's boundary.
    - Ideally, you should always skeletonize the smoothest version of a polygon that satisfies your analysis needs, as long as that version has no more artifacts than less smooth versions.
    - The biggest potential drawback here is that smoothing algorithms often create artifacts, and these artifacts can have significant consequences. For example, automated boundary smoothing may create narrow protrusions that require a fine sampling interval and/or cutting. Fine sampling could swell the memory footprint, at least partially offsetting the memory footprint reduction accomplished by smoothing, and cutting introduces the possibility of new artifacts. Templating is crudely analogous to an implicit smoothing of the boundary, and may yield similar memory footprint reduction with fewer hazards.
    - Manual smoothing, on the other hand, has no drawbacks (as long as the smoothed polygon still meets your analysis needs).

Although none of these strategies is implemented in numgeo, each can be easily executed in GIS software.

# Getting started

Having no doubt carefully read every word to this point (way to go!), you're ready to get started! For most use-cases, you'll only use one of two nearly identical functions. Each of these functions is documented in detail in their docstrings, so only the most important points are highlighted below.

`numgeo.scripts.skel.process` takes dozens of arguments, but only three are required. The remaining arguments are best understood in categories.

- required arguments

    - `in_path` is a string that specifies the path to a shapefile or file geodatabase feature class that contains the input polygons.
    - `interval` is a float that specifies the sampling interval (**Fig. 1**), in map units (e.g., meters), which is analogous to a resolution (**Fig. 2**).
        - Ideally, if the narrowest constriction in your input polygons is *x* map units, you should specify a sampling interval <*0.5x*.
        - However, specifying a small sampling interval for a complex polygon can use a lot of memory, potentially causing processing to abort if available memory is insufficient.
        - To have the sampling interval optimized to be approximately as small as possible, given available memory, you can specify `interval=0.`, but optimization will take additional time and is not currently advisable for very simple polygons (e.g., **Fig. 1a**).
    - `min_normalized_length` is a float that specifies the minimum normalized length that a branch may have and not be pruned.
        - Higher values (starting at 1.) result in a simpler skeleton (**Figs. 5, 6**).
        - There are a few strategies for choosing an optimal value for your application .
- output paths
    - `out_path_prefix` is a string that is prepended to the automatically generated name of each output shapefile or file geodatabase feature class.
    - `out_path_suffix` is a string that is appended to the automatically generated name of each output shapefile or file geodatabase feature class.
    - `out_dir_path` is a string that specifies the directory into which outputs should be placed.
    - `*_out_path` describes a set of string arguments that may be used to override the path of each output.
- output options
    - `output_*` describes a set of boolean arguments that control what outputs will be generated and written.
- keyword arguments passed to `numgeo.skel.EasySkeleton`
    - `isolation_mode` is a string that specifies options relating to isolation of the graph skeleton and general memory safeguards.
        - The default ("SAFE") is recommended unless you wish to enable cutting, in which case "SAFE_CUT" is recommended.
    - `tails` is a boolean that specifies whether tail addition is enabled.
    - `template_interval` is a float that specifies the coarse sampling interval used in templating.
    - `targ_GB` is a float that specifies the memory footprint, in gigabytes, that is permitted for processing.

`numgeo.scripts.skel.process.external` takes all the same arguments as `numgeo.scripts.skel.process`, except for its first argument.

- `settings_path` is a string that specifies a path to a text file that specifies any argument defaults that should be used instead of the built-in defaults. The format of the log file generated (by default) with each call to `numgeo.scripts.skel.process` or `numgeo.scripts.skel.process.external` is suitable for use as the target text file. This argument is intended to provide the user with a way to avoid repetitive long argument lists, easily execute a processing run similar to an earlier processing run, and effectively override hard-coded defaults with the user's preferred default behavior.
    - Note: You will probably want to delete any explicit paths in a log file before using it to define defaults.

- Any additional arguments override the defaults, if any, specified in the text file at `settings_path`.

## In case of emergency

If you find that something is not working (e.g., processing aborts or hangs), you might try one or more of the following to disable functionalities that can have some unusual side-effects:

- `monitor_memory=False`
  - Note: Memory logging will be disabled.
- `reduce_peak_memory=False`
  - Note: For some outputs with many geometries (excluding the skeleton), the memory footprint may increase dramatically. No safeguards are applied to prevent this increase from exhausting available memory and potentially freezing your system.