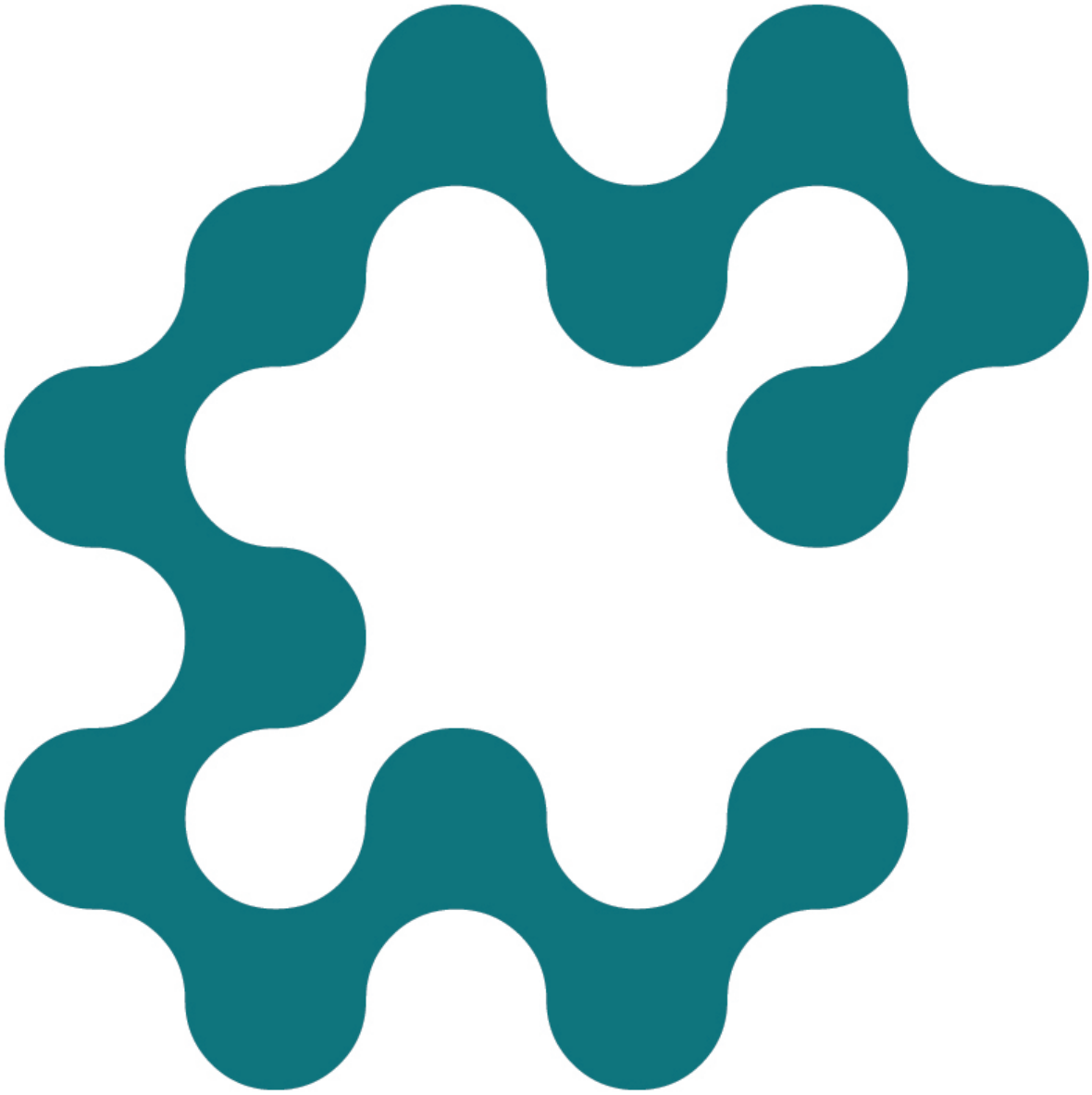


Cockatoo

Release 0.1.0.0-alpha

Max Eschenbach

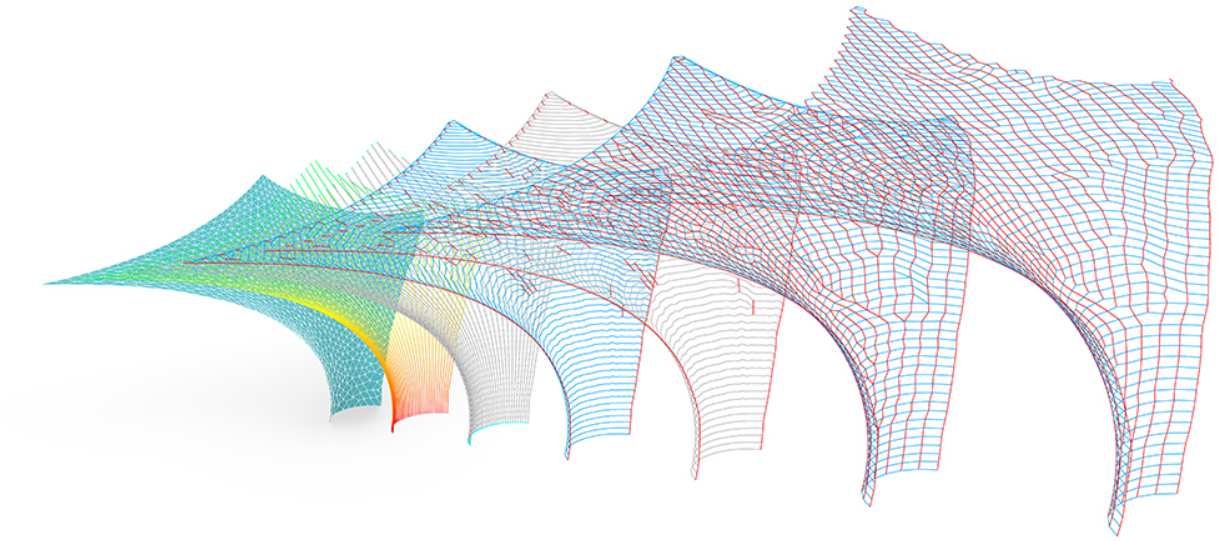
Aug 14, 2020

CONTENTS:

CHAPTER

ONE

COCKATOO



- Cockatoo is a prototypical open-source software toolkit for generating (3d-)knitting patterns from NURBS surface and mesh geometry.
- It is implemented as a Python module for use within McNeel Rhinoceros 6 aswell as Grasshopper.
- *Yeah, yeah… Knitting… Rhino… Python… I get it. Just tell me how to install and use it! 1*

1.1 Purpose & Origins

The purpose of this project is to enable Rhino and Grasshopper to automatically derive (3d-)knitting patterns for computerized knitting machines from NURBS surfaces and unstructured triangle meshes. The absence of such a freely available open-source toolkit marks the origin point for this project. Programming Cockatoo was only possible thanks to some brilliant research done by lots of other people. Please check the *Sources & References* section if you're curious.

This open-source software prototype constitutes the practical part of my diploma project *Knit Relaxation - Knit Membranes for Textile (Interior) Architecture* (original german title: *Knit Relaxation - Membrangestricke für Textile (Innen-)Architektur*) in the product design department at Kunsthochschule Kassel.

1.2 Software Structure

1.2.1 Python Module

- All datastructures, core logic and algorithms are defined in the `cockatoo` python module.
- This module is developed to be compatible with IronPython (for more info, see the *Pecularities* section).
- The RhinoCommon API is used to handle all geometric operations.
- The `networkx` module is used to handle all the necessary graph operations (for more info, see the *Pecularities* section).

1.2.2 Rhino Integration

The `cockatoo` module can be used from within Rhino.Python scripts as well as from within Grasshopper through the GHPython scriptable component.

1.2.3 Grasshopper Components

Cockatoo includes a set of Grasshopper components (`UserObjects`), which provide a user interface to the underlying python module without the need of scripting.

1.2.4 Extendability

The python module as well as the UserObjects are designed to be open for extension. Everything is open-source.

1.3 Pecularities

1.3.1 Development Environment

The RhinoPython and GHPython development environments are very, very special. I am not going to write in-depth about this here. Everybody who is working with these tools on a regular basis should have come accross their oddities. If not - most information about these topics is available in the Rhino Developer Docs.

1.3.2 Graph Library

To do all the juicy graph stuff, Cockatoo uses NetworkX. To be more specific, an older version - NetworkX 1.5 is used for... well, reasons. **This specific networkx module was modified in some places and is therefore bundled with Cockatoo! Using a different version might be possible but may also lead to errors.**

1.3.3 Partial Dependencies

- Some of the UserObjects rely on Kangaroo 2. Since this is shipped with Rhino since Version 6, everything should work smoothly. The Kangaroo 2 installation should be found by the UserObjects automagically. If any hiccups occur, please let me know.
- There is one UserObject that relies on Plankton being installed, although it's just a small utility. If Plankton is already installed everything should be found automagically, otherwise you'd first have to install Plankton. If any hiccups occur with this, please also let me know.

1.4 Installation

1.4.1 1. Download release files

- Go to releases and download the newest release
- Unzip the downloaded archive. You should get the folders: `modules` and `Cockatoo`.

1.4.2 2. Install python modules

- Open the scripts folder of Rhino 6
  - On **Windows**: `C:\Users\%USERNAME%\AppData\Roaming\McNeel\Rhinoceros\6.0\scripts`
  - On **Mac OSX**: `~/Library/Application Support/McNeel/Rhinoceros/6.0/scripts`
- Move all the Content from inside the `modules` directory to this scripts folder.

1.4.3 3. Install Cockatoo UserObjects

- Navigate to the Grasshopper UserObjects folder
  - On **Windows**: `C:\Users\%USERNAME%\AppData\Roaming\Grasshopper\UserObjects`
  - On **Mac OSX**: `~/Library/Application Support/McNeel/Rhinoceros/6.0/scripts`
  - *Alternative:* Open Rhino & Grasshopper and in the Grasshopper Window click on `File` > `Special Folders` > `User Object Folder`
- Move the whole `Cockatoo` directory to the UserObjects folder.

1.4.4 4. Unblock the new UserObjects!

- Go into the `Cockatoo` folder inside Grasshoppers UserObjects folder
- Right click onto the first UserObject and go to **Properties**
- If the text *This file came from another computer [. . .]* is displayed click on **Unblock**!
- **Unfortunately you have to do this for EVERY UserObject in the folder!**

1.4.5 5. Restart Rhino & Grasshopper

- If Rhino was running during the installation process, you'll have to restart it for the changes to take effect!

1.5 Examples & Usage

If everything is installed correctly, you should be able to open the example file provided in `Examples`. For a demo, you can also have a look at the demonstration video.

For guidance on using the API provided through the python module directly, please have a look at the documentation

1.6 Testing & Contributing

1.6.1 You are invited to participate!

Contributing is easy as Pi (well... easier, actually). First off, Cockatoo needs software testing to find bugs and make it more robust. So just by trying out Cockatoo out of curiosity, you can actually help!

If you find a bug (which is very likely because they always sneak in somewhere) please tell me about it by submitting an issue so I can improve Cockatoo further.

1.6.2 Testing

A sad truth is that I currently don't have access to a computerized knitting machine. As a consequence, it was not possible to actually test or verify a knitting pattern generated by Cockatoo in the real world, yet. If you have access to a machine and you would be willing to collaborate with me in testing, I would be more than happy!

Also, if you know a thing or two about computational knitting and find a fundamental (or minor) mistake in the workings of Cockatoo, please let me know. I'm always eager to learn from others and fix mistakes.

1.6.3 Code

If you're willing to contribute to Cockatoo by writing new code or improving existing code, that's great! Please have a look at the contribution guidelines.

1.7 Sources & References

This section states the most important sources used in writing this software. The full and proper list of sources is - of course - available in the written version of the diploma thesis.

- Cherif, Chokri: Textile Werkstoffe für den Leichtbau. Techniken - Verfahren - Materialien - Eigenschaften.
- CITAstudio: Textile. Light. Architecture. SOFT SPACES
- Hagberg, Aric; Schult, Dan; Swart, Pieter: NetworkX 1.5
- McCann, James; Albaugh, Lea; Narayanan, Vidya; Grow, April; Matusik, Wojciech; Mankoff, Jen; Hodgins, Jessica: A Compiler for 3D Machine Knitting
- Narayanan, Vidya; Albaugh, Lea; Hodgins, Jessica; Coros, Stelian; McCann, James: Automatic Machine Knitting of 3D Meshes

- Narayanan, Vidya; Wu, Kui; Yuksel, Cem; McCann, James: Visual Knitting Machine Programming
- Popescu, Mariana; Rippmann, Matthias; van Mele, Tom; Block, Philippe: Automated Generation of Knit Patterns for Non-developable Surfaces
- Popescu, Mariana: KnitCrete - Stay-in-place knitted formworks for complex concrete structures
- Thomsen, Mette Ramsgaard; Tamke, Martin; Deleuran, Anders Holden; Tinning, Ida Katrine Friis; Evers, Henrik Leander; Gengnagel, Christoph; Schmeck, Michel: *»Hybrid Tower, Designing Soft Structures«*. In: Modelling behaviour. Design Modelling Symposium 2015, hrsg. von Mette Ramsgaard Thomsen, Martin Tamke, Christoph Gengnagel, Billie Faircloth und Fabian Scheurer. Cham/Heidelberg/New York/Dordrecht/London 2015
- Ramsgaard Thomsen, Mette; Tamke, Martin; Ayres, Phil; Nicholas, Paul: CITA Complex Modelling, Toronto 2019
- Van Mele, Tom; others, many: COMPAS: A framework for computational research in architecture and structures

1.8 Licensing

- Original code is licensed under the MIT License.
- NetworkX is licensed under the 3-clause BSD license which can be found in `licenses/networkx`.
- Some code snippets from the COMPAS framework are used within this software. This code is licensed under the MIT License which can be found in `licenses/COMPAS`.
- Some code snippets by Anders Holden Deleuran are used with permission. They originate from gists and the *FAHS* pipeline, kindly provided by Anders. This code is licensed under the Apache License 2.0 which can be found in `licenses/ahd`.

1.9 Misc

- [1] This is a hommage to David Ruttens delightful sense of humor which has changed many of my darker days for the better.

CHAPTER

TWO

COCKATOO MODULE API

2.1 Submodules

2.1.1 cockatoo.environment module

is_rhino_inside	Check if Rhino is running using rhinoinside.
RHINOINSIDE	Will be `True` if Rhino is running using rhinoinside, `False` otherwise.
networkx_version	Return the version of the used networkx module.
NXVERSION	The version string of the networkx module that is being used.

cockatoo.environment.**is_rhino_inside**()
Check if Rhino is running using rhinoinside.

Returns *bool* – `True` if Rhino is running using rhinoinside, otherwise `False`.

Raises ***RhinoNotPresentError*** – If import of Rhino fails.

cockatoo.environment.**RHINOINSIDE = False**
Will be `True` if Rhino is running using rhinoinside, `False` otherwise.

Type bool

cockatoo.environment.**networkx_version**()
Return the version of the used networkx module.

Returns *str* – The version string of the used networkx module.

Raises ***NetworkXNotPresentError*** – If the networkx module cannot be found.

cockatoo.environment.**NXVERSION = '1.5'**
The version string of the networkx module that is being used.

Type str

2.1.2 cockatoo.exception module

CockatooException	Base class for exceptions in Cockatoo.
CockatooImportException	Base class for import errors in Cockatoo.
RhinoNotPresentError	Exception raised when import of Rhino fails.
SystemNotPresentError	Exception raised when import of System fails.
NetworkXNotPresentError	Exception raised when import of NetworkX fails.
NetworkXVersionError	Exception raised when NetworkX version is not 1.5.
KnitNetworkError	Exception for a serious error in a KnitNetwork of Cockatoo.
KnitNetworkGeometryError	Exception raised when vital geometry operations fail.
MappingNetworkError	Exception raised by methods relying on a mapping network if no mapping network has been assigned to the current KnitNetwork instance yet.
KnitNetworkTopologyError	Exception raised by methods which rely on a certain topology of a network if that topology could not be verified.
NoWeftEdgesError	Exception raised by methods relying on 'weft' edges if there are no 'weft' edges in the network.
NoWarpEdgesError	Exception raised by methods relying on 'warp' edges if there are no 'warp' edges in the network.
NoEndNodesError	Exception raised by methods relying on 'end' nodes if there are no 'end' nodes in the network.

exception cockatoo.exception.**CockatooException**
Bases: Exception

Base class for exceptions in Cockatoo.

exception cockatoo.exception.**CockatooImportException**
Bases: ImportError

Base class for import errors in Cockatoo.

exception cockatoo.exception.**RhinoNotPresentError**
Bases: *cockatoo.exception.CockatooImportException*

Exception raised when import of Rhino fails.

exception cockatoo.exception.**SystemNotPresentError**
Bases: *cockatoo.exception.CockatooImportException*

Exception raised when import of System fails.

exception cockatoo.exception.**NetworkXNotPresentError**
Bases: *cockatoo.exception.CockatooImportException*

Exception raised when import of NetworkX fails.

exception cockatoo.exception.**NetworkXVersionError**
Bases: *cockatoo.exception.CockatooException*

Exception raised when NetworkX version is not 1.5.

exception cockatoo.exception.**KnitNetworkError**
Bases: *cockatoo.exception.CockatooException*

Exception for a serious error in a KnitNetwork of Cockatoo.

exception cockatoo.exception.**KnitNetworkGeometryError**
Bases: *cockatoo.exception.KnitNetworkError*

Exception raised when vital geometry operations fail.

exception cockatoo.exception.**MappingNetworkError**
Bases: *cockatoo.exception.KnitNetworkError*

Exception raised by methods relying on a mapping network if no mapping network has been assigned to the current KnitNetwork instance yet.

exception cockatoo.exception.**KnitNetworkTopologyError**
Bases: *cockatoo.exception.KnitNetworkError*

Exception raised by methods which rely on a certain topology of a network if that topology could not be verified.

exception cockatoo.exception.**NoWeftEdgesError**
Bases: *cockatoo.exception.KnitNetworkError*

Exception raised by methods relying on 'weft' edges if there are no 'weft' edges in the network.

exception cockatoo.exception.**NoWarpEdgesError**
Bases: *cockatoo.exception.KnitNetworkError*

Exception raised by methods relying on 'warp' edges if there are no 'warp' edges in the network.

exception cockatoo.exception.**NoEndNodesError**
Bases: *cockatoo.exception.KnitNetworkError*

Exception raised by methods relying on 'end' nodes if there are no 'end' nodes in the network.

2.1.3 cockatoo.utilities module

blend_colors	Blend between two colors using the square root of photon flux.
break_polyline	Breaks a polyline at kinks based on a specified angle.
map_values_as_colors	Make a list of HSL colors where the values are mapped onto a targetMin-targetMax hue domain.
tween_planes	Tweens between two planes using quaternion rotation.
is_ccw_xy	Determine if c is on the left of ab when looking from a to b, and assuming that all points lie in the XY plane.
resolve_order_by_backtracking	Resolve topological order of a networkx DiGraph through backtracking of all nodes in the graph.

cockatoo.utilities.**blend_colors**(*col_a*, *col_b*, *t=0.5*)
Blend between two colors using the square root of photon flux. For more info see *Algorithm for additive color mixing for RGB values*[18].

Parameters

- **col_a** (sequence of `int`) – Sequence of (R, G, B) that defines the color value.
- **col_b** (sequence of `int`) – Sequence of (R, G, B) that defines the color value.
- **t** (*float, optional*) – Parameter to define the blend location between the two colors.

  Defaults to `0.5`.

[18] *Algorithm for additive color mixing for RGB values*
See: Thread on stackoverflow

Returns **color** (*tuple*) – 3-tuple of (R, G, B) that defines the new color.

References

cockatoo.utilities.**break_polyline**(*polyline*, *break_angle*, *as_crv=False*)

Breaks a polyline at kinks based on a specified angle. Will move the seam of closed polylines to the first kink discovered.

Parameters

- **polyline** (Rhino.Geometry.Polyline) – Polyline to break apart at angles.
- **break_angle** (*float*) – The angle at which to break apart the polyline (in radians).
- **as_crv** (*bool, optional*) – If True, will return a Rhino.Geometry.PolylineCurve object.

  Defaults to False.

Returns

- **polyline_segments** (list of Rhino.Geometry.Polyline) – A list of the broken segments as Polylines if as_crv is False.
- **polyline_segments** (list of Rhino.Geometry.PolylineCurve) – A list of the broken segments as PolylineCurves if as_crv is True.

cockatoo.utilities.**map_values_as_colors**(*values*, *src_min*, *src_max*, *target_min=0.0*, *target_max=0.7*)

Make a list of HSL colors where the values are mapped onto a targetMin-targetMax hue domain. Meaning that low values will be red, medium values green and large values blue if target_min is 0.0 and target_max is 0.7.

Parameters

- **values** (*list*) – List of values to map as colors.
- **src_min** (*float*) – Lower bounds of the value domain.
- **src_max** (*float*) – Upper bounds of the value domain.
- **target_min** (*float, optional*) – Lower bounds of the target (color) domain.

  Defaults to 0.
- **target_max** (*float, optional*) – Upper bounds of the target (color) domain.

  Defaults to 0.7 .

Returns **colors** (*list*) – List of RGB colors corresponding to the input values.

Notes

Based on code by Anders Holden Deleuran. Code was only changed in regards of defaults and names. For more info see *mapValuesAsColors.py*[10] .

[10] Deleuran, Anders Holden *mapValuesAsColors.py*
See: mapValuesAsColors.py gist

References

cockatoo.utilities.**tween_planes**(*pa*, *pb*, *t*)

Tweens between two planes using quaternion rotation. Based on code by Chris Hanley.[19]

Parameters

- **pa** (Rhino.Geometry.Plane) – The start plane for the tween.
- **pb** (Rhino.Geometry.Plane) – The end plane for the tween.
- **t** (*float*) – The parameter for the tweened plane. 0.5 will result in the average between the two input planes.

Returns **tweened_plane** (Rhino.Geometry.Plane) – The plane between pa and pb at parameter t.

Raises ***SystemNotPresentError*** – If the System module cannot be imported.

References

cockatoo.utilities.**is_ccw_xy**(*a*, *b*, *c*, *colinear=False*)

Determine if c is on the left of ab when looking from a to b, and assuming that all points lie in the XY plane.

Parameters

- **a** (*sequence of float*) – XY(Z) coordinates of the base point.
- **b** (*sequence of float*) – XY(Z) coordinates of the first end point.
- **c** (*sequence of float*) – XY(Z) coordinates of the second end point.
- **colinear** (*bool, optional*) – Allow points to be colinear. Default is False.

Returns *bool* – True if ccw. False otherwise.

Notes

Based on an implementation inside the COMPAS framework. For more info, see[14] and[15].

References

Examples

```
>>> print(is_ccw_xy([0,0,0], [0,1,0], [-1, 0, 0]))
True
>>> print(is_ccw_xy([0,0,0], [0,1,0], [+1, 0, 0]))
False
>>> print(is_ccw_xy([0,0,0], [1,0,0], [2,0,0]))
False
>>> print(is_ccw_xy([0,0,0], [1,0,0], [2,0,0], True))
True
```

[19] *Average between two planes*
See: Thread on discourse.mcneel.com
[14] Van Mele, Tom et al. *COMPAS: A framework for computational research in architecture and structures*.
See: is_ccw_xy() inside COMPAS
[15] Marsh, C. *Computational Geometry in Python: From Theory to Application*.
See: Computational Geometry in Python

cockatoo.utilities.**resolve_order_by_backtracking**(*G*)

Resolve topological order of a networkx DiGraph through backtracking of all nodes in the graph. Nodes are only inserted into the output list if all their dependencies (predecessor nodes) are already inside the output list, otherwise the algorithm will first resolve all open dependencies.

Parameters **G** (networkx.Graph) – The graph on which to perform topological sorting.

Returns **ordered_nodes** (*list*) – List of hashable node identifiers.

Raises **ValueError** – If the input graph is not directed.

Warning: For this to work, the input gaph must be a DAG (directed acyclic graph). For more info,see[11] and[12].

References

cockatoo.utilities.**pairwise**(*iterable*)

Returns the data of iterable in pairs (2-tuples).

Parameters **iterable** (*iterable*) – An iterable sequence of items.

Yields *tuple* – Two items per iteration, if there are at least two items in the iterable.

Examples

```
>>> print(pairwise(range(4))):
...
[(0, 1), (1, 2), (2, 3)]
```

Notes

For more info see[16] .

References

2.2 Classes

cockatoo.KnitConstraint	Datastructure for representing constraints derived from a mesh.
cockatoo.KnitNetworkBase	Abstract datastructure for representing a network (graph) consisting of nodes with special attributes aswell as 'warp' edges, 'weft' edges and contour edges which are neither 'warp' nor 'weft'.

continues on next page

[11] Directed acyclic graph on Wikipedia.
See: Directed acyclic graph
[12] Topological sorting on Wikipedia.
See: Topological sorting
[16] Python itertools Recipes
See: Python itertools Recipes

Table 4 – continued from previous page

cockatoo.KnitNetwork	Datastructure for representing a network (graph) consisting of nodes with special attributes aswell as 'warp' edges, 'weft' edges and contour edges which are neither 'warp' nor 'weft'.
cockatoo.KnitDiNetwork	Datastructure representing a directed graph of nodes aswell as 'weft' and 'warp' edges.
cockatoo.KnitMappingNetwork	Datastructure representing a mapping between connected chains of 'weft' edges in a KnitNetwork for final creation of 'weft' and 'warp' edges.

2.2.1 cockatoo.KnitConstraint

class cockatoo.**KnitConstraint**(*start_course*, *end_course*, *left_boundary*, *right_boundary*)

Bases: object

Datastructure for representing constraints derived from a mesh. Used for the automatic generation of knitting patterns.

ToString()

Return a textual description of the constraint.

Returns **description** (*str*) – A textual description of the constraint.

Notes

Used for overloading the Grasshopper display in data parameters.

property end_course

The end course of the KnitConstraint

property left_boundary

The left boundary of the KnitConstraint

property right_boundary

The right boundary of the KnitConstraint

property start_course

The start course of the KnitConstraint

2.2.2 cockatoo.KnitNetworkBase

class cockatoo.**KnitNetworkBase**(*data=None*, ***attr*)

Bases: networkx.classes.graph.Graph

Abstract datastructure for representing a network (graph) consisting of nodes with special attributes aswell as 'warp' edges, 'weft' edges and contour edges which are neither 'warp' nor 'weft'.

Used as a base class for sharing behaviour between the KnitNetwork, KnitMappingNetwork and KnitDiNetwork classes.

Inherits from networkx.Graph. For more info, see *NetworkX*[13].

[13] Hagberg, Aric A.; Schult, Daniel A.; Swart, Pieter J. *Exploring Network Structure, Dynamics, and Function using NetworkX* In: *Varoquaux, Vaught et al. (Hg.) 2008 - Proceedings of the 7th Python in Science Conference* pp. 11-15
See: NetworkX 1.5

References

ToString()
Return a textual description of the network.

Returns **description** (*str*) – A textual description of the network.

Notes

Used for overloading the Grasshopper display in data parameters.

all_ends_by_position(*data=False*)
Gets all 'end' nodes ordered by their 'position' attribute.

Parameters **data** (*bool, optional*) – If `True`, found nodes will be returned with their attribute data.

Defaults to `False`.

Returns **nodes** (`list` of `list`) – All nodes for which the attribute 'end' is true, grouped by their 'position' attribute

all_leaves_by_position(*data=False*)
Gets all 'leaf' nodes ordered by their 'position' attribute.

Parameters **data** (*bool, optional*) – If `True`, found nodes will be returned with their attribute data.

Defaults to `False`.

Returns **nodes** (`list` of `list`) – All nodes for which the attribute 'leaf' is true, grouped by their 'position' attribute

all_nodes_by_position(*data=False*)
Gets all the nodes of the network, ordered by the values of their 'position' attribute.

Parameters **data** (*bool, optional*) – If `True`, found nodes will be returned with their attribute data.

Defaults to `False`.

Returns **nodes** (`list` of `list`) – All nodes grouped by their 'position' attribute

property contour_edges
The contour edges of the network marked neither 'weft' nor 'warp'.

create_contour_edge(*from_node*, *to_node*)
Creates an edge neither 'warp' nor 'weft' between two nodes in the network.

Parameters

- **from_node** (`tuple`) – 2-tuple of (node_identifier, node_data) that represents the edges' source node.
- **to_node** (`tuple`) – 2-tuple of (node_identifier, node_data) that represents the edges' target node.

Returns **success** (*bool*) – `True` if the edge has been successfully created, `False` otherwise.

create_segment_contour_edge(*from_node*, *to_node*, *segment_value*, *segment_geo*)
Creates a mapping edge between two 'end' nodes in the network. The geometry of this edge will be a polyline built from all the given former 'weft' edges. returns True if the edge has been successfully created.

Parameters

- **from_node** (tuple) – 2-tuple of (node_identifier, node_data) that represents the edges' source node.
- **to_node** (tuple) – 2-tuple of (node_identifier, node_data) that represents the edges' target node.
- **segment_value** (tuple of int) – 3-tuple that will be used to set the 'segment' attribute of the 'weft' edge.
- **segment_geo** (list of Rhino.Geometry.Line) – the geometry of all 'weft' edges that make this segment contour edge

Returns **success** (*bool*) – True if the edge has been successfully created, False otherwise

create_warp_edge(*from_node*, *to_node*)

Creates a 'warp' edge between two nodes in the network.

Parameters

- **from_node** (tuple) – 2-tuple of (node_identifier, node_data) that represents the edges' source node.
- **to_node** (tuple) – 2-tuple of (node_identifier, node_data) that represents the edges' target node.

Returns **success** (*bool*) – True if the edge has been successfully created. False otherwise.

create_weft_edge(*from_node*, *to_node*, *segment=None*)

Creates a 'weft' edge between two nodes in the network.

Parameters

- **from_node** (tuple) – 2-tuple of (node_identifier, node_data) that represents the edges' source node.
- **to_node** (tuple) – 2-tuple of (node_identifier, node_data) that represents the edges' target node.
- **segment** (tuple) – 3-tuple that will be used to set the 'segment' attribute of the 'weft' edge.

Returns **success** (*bool*) – True if the edge has been successfully created. False otherwise.

edge_geometry_direction(*u*, *v*)

Returns a given edge in order with reference to the direction of the associated geometry (line).

Parameters

- **u** (*hashable*) – Hashable identifier of the edges source node.
- **v** (*hashable*) – Hashable identifier of the edges target node.

Returns **edge** (*2-tuple*) – 2-tuple of (u, v) or (v, u) depending on the directions

end_node_segments_by_end(*node*, *data=False*)

Get all the edges with a 'segment' attribute marked neither 'weft' nor 'warp' and share a given 'end' node at the end, sorted by the values of their 'segment' attribute.

Parameters

- **node** (*hashable*) – Hashable identifier of the node to check for connected segments.
- **data** (*bool, optional*) – If True, the edges will be returned as 3-tuples with their associated attribute data.

Defaults to `False`.

Returns **edges** (*list*) – List of edges. Each item will be either a 2-tuple of (u, v) identifiers or a 3-tuple of (u, v, d) where d is the attribute data of the edge, depending on the data parameter.

end_node_segments_by_start (*node*, *data=False*)
Get all the edges with a 'segment' attribute marked neither 'weft' nor 'warp' and share a given 'end' node at the start, sorted by the values of their 'segment' attribute.

Parameters

- **node** (*hashable*) – Hashable identifier of the node to check for connected segments.
- **data** (*bool, optional*) – If `True`, the edges will be returned as 3-tuples with their associated attribute data.

  Defaults to `False`.

Returns **edges** (`list`) – List of edges. Each item will be either a 2-tuple of (u, v) identifiers or a 3-tuple of (u, v, d) where d is the attribute data of the edge, depending on the data parameter.

property end_nodes
All 'end' nodes of the network

ends_on_position (*position*, *data=False*)
Gets all 'end' nodes which share the supplied value as their 'position' attribute.

Parameters

- **position** (*hashable*) – The index / identifier of the position
- **data** (*bool, optional*) – If `True`, found nodes will be returned with their attribute data.

  Defaults to `False`.

Returns **nodes** (`list`) – List of all nodes for which the attribute 'end' is `True` and which share the supplied value as their 'position' attribute

geometry_at_position_contour (*position*, *as_crv=False*)
Gets the contour polyline at a given position by making a polyline from all nodes which share the specified 'position' attribute.

Parameters

- **position** (*hashable*) – The index / identifier of the position
- **as_crv** (*bool, optional*) – If `True`, will return a PolylineCurve instead of a Polyline.

  Defaults to `False`.

Returns

- **contour** (`Rhino.Geometry.Polyline`) – The contour as a Polyline if `as_crv` is `False`.
- **contour** (`Rhino.Geometry.PolylineCurve`) – The contour as a PolylineCurve if `as_crv` is `True`.

property leaf_nodes
All 'leaf' nodes of the network.

leaves_on_position (*position*, *data=False*)
Gets all 'leaf' nodes which share the supplied value as their 'position' attribute.

Parameters

- **position** (*hashable*) – The index / identifier of the position
- **data** (*bool, optional*) – If `True`, found nodes will be returned with their attribute data.

  Defaults to `False`.

Returns **nodes** (`list`) – List of all nodes for which the attribute 'leaf' is `True` and which share the supplied value as their 'position' attribute

longest_position_contour()

Gets the longest contour 'position', geometry andgeometric length.

Returns **contour_data** (`tuple`) – 3-tuple of the 'position' identifier, the contour geometry and its length.

node_contour_edges(*node*, *data=False*)

Gets the edges marked neither 'warp' nor 'weft' connected to the given node.

Parameters

- **node** (*hashable*) – Hashable identifier of the node to check for edges marked neither 'warp' nor 'weft'.
- **data** (*bool, optional*) – If `True`, the edges will be returned as 3-tuples with their associated attribute data.

  Defaults to `False`.

Returns **edges** (`list`) – List of edges marked neither 'warp' nor 'weft' connected to the given node. Each item in the list will be either a 2-tuple of (u, v) identifiers or a 3-tuple of (u, v, d) where d is the attribute data of the edge, depending on the data parameter.

node_coordinates(*node_index*)

Gets the node coordinates from the 'x', 'y' and 'z' attributes of the supplied node.

Parameters **node_index** (*hashable*) – The unique identifier of the node, an int in most cases.

Returns **xyz** (`tuple` of `int`) – The XYZ coordinates of the node as a 3-tuple.

node_from_point3d(*node_index*, *pt*, *position=None*, *num=None*, *leaf=False*, *start=False*, *end=False*, *segment=None*, *increase=False*, *decrease=False*, *color=None*)

Creates a network node from a Rhino Point3d and attributes.

Parameters

- **node_index** (*hashable*) – The index of the node in the network. Usually an integer is used.
- **pt** (`Rhino.Geometry.Point3d`) – A RhinoCommon Point3d object.
- **position** (*hashable, optional*) – The 'position' attribute of the node identifying the underlying contour edge of the network.

  Defaults to `None`.
- **num** (*int, optional*) – The 'num' attribute of the node representing its index in the underlying contour edge of the network.

  Defaults to `None`.

- **leaf** (*bool, optional*) – The 'leaf' attribute of the node identifying it as a node on the first or last course of the knitting pattern.

  Defaults to `False`.

- **start** (*bool, optional*) – The 'start' attribute of the node identifying it as the start of a course.

  Defaults to `False`.

- **end** (*bool, optional*) – The 'end' attribute of the node identifying it as the end of a segment or course.

  Defaults to `False`.

- **segment** (`tuple` of `int`, optional) – The 'segment' attribute of the node identifying its position between two 'end' nodes.

  Defaults to `None`.

- **increase** (*bool, optional*) – The 'increase' attribute identifying the node as an increase (needed for translation from dual to 2d knitting pattern).

  Defaults to `False`.

- **decrease** (*bool, optional*) – The 'decrease' attribute identifying the node as a decrease (needed for translation from dual to 2d knitting pattern).

  Defaults to `False`.

- **color** (`System.Drawing.Color`, optional) – The 'color' attribute of the node, representing the color of the pixel when translating the network to a 2d knitting pattern.

  Defaults to `None`.

node_geometry (*node_index*)
Gets the geometry from the 'geo' attribute of the supplied node.

Parameters **node_index** (*hashable*) – The unique identifier of the node, an int in most cases.

Returns **geometry** (*data*) – The data of the 'geo' attribute of the specified node or `None` if the node is not present or has no 'geo' attribute.

node_warp_edges (*node*, *data=False*)
Gets the 'warp' edges connected to the given node.

Parameters

- **node** (*hashable*) – Hashable identifier of the node to check for 'warp' edges.
- **data** (*bool, optional*) – If `True`, the edges will be returned as 3-tuples with their associated attribute data.

  Defaults to `False`.

Returns **edges** (`list`) – List of 'warp' edges connected to the given node. Each item in the list will be either a 2-tuple of (u, v) identifiers or a 3-tuple of (u, v, d) where d is the attribute data of the edge, depending on the data parameter.

node_weft_edges (*node*, *data=False*)
Gets the 'weft' edges connected to a given node.

Parameters

- **node** (*hashable*) – Hashable identifier of the node to check for 'weft' edges.

- **data** (*bool, optional*) – If `True`, the edges will be returned as 3-tuples with their associated attribute data.

  Defaults to `False`.

Returns **edges** (`list`) – List of 'weft' edges connected to the given node. Each item in the list will be either a 2-tuple of (u, v) identifiers or a 3-tuple of (u, v, d) where d is the attribute data of the edge, depending on the data parameter.

nodes_on_position(*position*, *data=False*)

Gets the nodes on a given position (i.e. contour) by returning all nodes which share the given value as their 'position' attribute.

Parameters

- **position** (*hashable*) – The index of the position.
- **data** (*bool, optional*) – If `True`, found nodes will be returned with their attribute data.

  Defaults to `False`.

Returns **nodes** (`list`) – The nodes sharing the supplied 'position' attribute.

nodes_on_segment(*segment*, *data=False*)

Gets all nodes on a given segment by finding all nodes which share the specified value as their 'segment' attribute, ordered by the value of their 'num' attribute.

Parameters

- **segment** (*hashable*) – The identifier of the segment to look for.
- **data** (*bool, optional*) – If `True`, found nodes will be returned with their attribute data.

  Defaults to `False`.

Returns **nodes** (`list`) – List of nodes sharing the supplied value as their 'segment' attribute, ordered by their 'num' attribute.

prepare_for_gephi()

Creates a new graph with attributes for visualising this network using Gephi.

Based on code by Anders Holden Deleuran

prepare_for_graphviz()

Creates a new graph with attributes for visualising this network using GraphViz.

Based on code by Anders Holden Deleuran

property segment_contour_edges

The edges of the network marked neither 'warp' nor 'weft' and which have a 'segment' attribute assigned to them.

property total_positions

The total number of positions (i.e. contours) inside the network

property warp_edges

The edges of the network marked 'warp'.

property weft_edges

The edges of the network marked 'weft'.

2.2.3 cockatoo.KnitNetwork

class cockatoo.**KnitNetwork**(*data=None*, ***attr*)
Bases: cockatoo._knitnetworkbase.KnitNetworkBase

Datastructure for representing a network (graph) consisting of nodes with special attributes aswell as 'warp' edges, 'weft' edges and contour edges which are neither 'warp' nor 'weft'.

Used for the automatic generation of knitting patterns based on mesh or NURBS surface geometry.

Inherits from *KnitNetworkBase*.

Notes

The implemented algorithms are strongly based on the paper *Automated Generation of Knit Patterns for Non-developable Surfaces*[1]. Also see *KnitCrete - Stay-in-place knitted formworks for complex concrete structures*[2].

The implementation was further influenced by concepts and ideas presented in the papers *Automatic Machine Knitting of 3D Meshes*[3], *Visual Knitting Machine Programming*[4] and *A Compiler for 3D Machine Knitting*[5].

References

ToString()
Return a textual description of the network.

Returns **description** (*str*) – A textual description of the network.

Notes

Used for overloading the Grasshopper display in data parameters.

all_nodes_by_segment(*data=False*, *edges=False*)
Returns all nodes of the network ordered by 'segment' attribute. Note: 'end' nodes are not included!

Parameters

- **data** (*bool, optional*) – If True, the nodes contained in the output will be represented as 2-tuples in the form of (node_identifier, node_data).

  Defaults to False
- **edges** (*bool, optional*) – If True, the returned output list will contain 3-tuples in the form of (segment_value, segment_nodes, segment_edge).

  Defaults to False.

[1] Popescu, Mariana et al. *Automated Generation of Knit Patterns for Non-developable Surfaces*
See: Automated Generation of Knit Patterns for Non-developable Surfaces
[2] Popescu, Mariana *KnitCrete - Stay-in-place knitted formworks for complex concrete structures*
See: KnitCrete - Stay-in-place knitted formworks for complex concrete structures
[3] Narayanan, Vidya; Albaugh, Lea; Hodgins, Jessica; Coros, Stelian; McCann, James *Automatic Machine Knitting of 3D Meshes*
See: Automatic Machine Knitting of 3D Meshes
[4] Narayanan, Vidya; Wu, Kui et al. *Visual Knitting Machine Programming*
See: Visual Knitting Machine Programming
[5] McCann, James; Albaugh, Lea; Narayanan, Vidya; Grow, April; Matusik, Wojciech; Mankoff, Jen; Hodgins, Jessica *A Compiler for 3D Machine Knitting*
See: A Compiler for 3D Machine Knitting

Returns **nodes_by_segment** (`list` of `tuple`) – List of 2-tuples in the form of (segment_value, segment_nodes) or 3-tuples in the form of (segment_value, segment_nodes, segment_edge) depending on the `edges` argument.

Raises ***MappingNetworkError*** – If the mapping network is not available for this instance.

assign_segment_attributes()
Get the segmentation for loop generation and assign 'segment' attributes to 'weft' edges and nodes.

attempt_warp_connection(*node*, *candidate*, *source_nodes*, *max_connections=4*, *verbose=False*)
Method for attempting a 'warp' connection to a candidate node based on certain parameters.

Parameters

- **node** (*node*) – The starting node for the possible 'weft' edge.
- **candidate** (*node*) – The target node for the possible 'weft' edge.
- **source_nodes** (`list`) – List of nodes on the position contour of node. Used to check if the candidate node already has a connection.
- **max_connections** (*int, optional*) – The new 'weft' connection will only be made if the candidate nodes number of connected neighbors is below this.

  Defaults to `4`.
- **verbose** (*bool, optional*) – If `True`, this routine and all its subroutines will print messages about what is happening to the console.

  Defaults to `False`.

Returns **result** (*bool*) – True if the connection has been made, otherwise false.

Notes

Closely resembles the implementation described in *Automated Generation of Knit Patterns for Non-developable Surfaces*[1]. Also see *KnitCrete - Stay-in-place knitted formworks for complex concrete structures*[2].

attempt_weft_connection(*node*, *candidate*, *source_nodes*, *max_connections=4*, *verbose=False*)
Method for attempting a 'weft' connection to a candidate node based on certain parameters.

Parameters

- **node** (`tuple`) – 2-tuple representing the source node for the possible 'weft' edge.
- **candidate** (`tuple`) – -tuple representing the target node for the possible 'weft' edge.
- **source_nodes** (`list`) – List of nodes on the position contour of node. Used to check if the candidate node already has a connection.
- **max_connections** (*int, optional*) – The new 'weft' connection will only be made if the candidate nodes number of connected neighbors is below this.

  Defaults to `4`.
- **verbose** (*bool, optional*) – If `True`, this routine and all its subroutines will print messages about what is happening to the console.

  Defaults to `False`.

Returns *bool* – `True` if the connection has been made, `False` otherwise.

Notes

Closely resembles the implementation described in *Automated Generation of Knit Patterns for Non-developable Surfaces*[1]. Also see *KnitCrete - Stay-in-place knitted formworks for complex concrete structures*[2].

create_dual(*mode=- 1*, *merge_adj_creases=False*, *mend_trailing_rows=False*)

Creates the dual of this KnitNetwork while translating current edge attributes to the edges of the dual network.

Parameters

- **mode** (*int, optional*) – Determines how the neighbors of each node are sorted when finding cycles for the network.

  `-1` equals to using the world XY plane.

  `0` equals to using a plane normal to the origin nodes closest point on the reference geometry.

  `1` equals to using a plane normal to the average of the origin and neighbor nodes' closest points on the reference geometry.

  `2` equals to using an average plane between a plane fit to the origin and its neighbor nodes and a plane normal to the origin nodes closest point on the reference geometry.

  Defaults to `-1`.

- **merge_adj_creases** (*bool, optional*) – If `True`, will merge adjacent 'increase' and 'decrease' nodes connected by a 'weft' edge into a single node. This effectively simplifies the pattern, as a decrease is unneccessary to perform if an increase is right beside it - both nodes can be replaced by a single regular node (stitch).

  Defaults to `False`.

- **mend_trailing_rows** (*bool, optional*) – If `True`, will attempt to mend trailing rows by reconnecting nodes.

  Defaults to `False`.

Returns **dual_network** (*KnitDiNetwork*) – The dual network of this KnitNetwork.

Warning: Modes other than -1 (default) are only possible if this network has an underlying reference geometry in form of a Mesh or NurbsSurface. The reference geometry should be assigned when initializing the network by assigning the geometry to the 'reference_geometry' attribute of the network.

Notes

Closely resembles the implementation described in *Automated Generation of Knit Patterns for Non-developable Surfaces*[1]. Also see *KnitCrete - Stay-in-place knitted formworks for complex concrete structures*[2].

create_final_warp_connections(*max_connections=4*, *include_end_nodes=True*, *precise=False*, *verbose=False*)

Create the final 'warp' connections by building chains of segment contour edges and connecting them.

For each source chain, a target chain is found using an 'educated guessing' strategy. This means that the possible target chains are guessed by leveraging known topology facts about the network and its special 'end' nodes.

Parameters

- **max_connections** (*int, optional*) – The number of maximum previous connections a candidate node for a 'warp' connection is allowed to have.

  Defaults to `4`.

- **include_end_nodes** (*bool, optional*) – If `True`, 'end' nodes between adjacent segment contours in a source chain will be included in the first pass of connecting 'warp' edges.

  Defaults to `True`.

- **precise** (*bool*) – If `True`, the distance between nodes will be calculated using the Rhino.Geometry.Point3d.DistanceTo method, otherwise the much faster Rhino.Geometry.Point3d.DistanceToSquared method is used.

  Defaults to `False`.

- **verbose** (*bool, optional*) – If `True`, this routine and all its subroutines will print messages about what is happening to the console. Great for debugging and analysis.

  Defaults to `False`.

Notes

Closely resembles the implementation described in *Automated Generation of Knit Patterns for Non-developable Surfaces*[1]. Also see *KnitCrete - Stay-in-place knitted formworks for complex concrete structures*[2].

create_final_weft_connections()
Loop through all the segment contour edges and create all 'weft' connections for this network.

Notes

Closely resembles the implementation described in *Automated Generation of Knit Patterns for Non-developable Surfaces*[1]. Also see *KnitCrete - Stay-in-place knitted formworks for complex concrete structures*[2].

classmethod create_from_contours(*contours*, *course_height*, *reference_geometry=None*)
Create and initialize a KnitNetwork based on a set of contours, a given course height and an optional reference geometry. The reference geometry is a mesh or surface which should be described by the network. While it is optional, it is **HIGHLY** recommended to provide it!

Parameters

- **contours** (`list` of `Rhino.Geometry.Polyline` – or `Rhino.Geometry.Curve` Ordered contours (i.e. isocurves, isolines) to initialize the KnitNetwork with.
- **course_height** (*float*) – The course height for sampling the contours.
- **reference_geometry** (`Rhino.Geometry.Mesh` – or `Rhino.Geometry.Surface` Optional underlying geometry that this network is based on.

Returns **KnitNetwork** (*KnitNetwork*) – A new, initialized KnitNetwork instance.

Notes

This method will automatically call initialize_position_contour_edges() on the newly created network!

Raises ***KnitNetworkGeometryError*** – If a supplied contour is not a valid instance of `Rhino.Geometry.Polyline` or `Rhino.Geometry.Curve`.

create_mapping_network()

Creates the corresponding mapping network for the final loop generation from a KnitNetwork instance with fully assigned 'segment' attributes.

The created mapping network will be part of the KnitNetwork instance. It can be accessed using the mapping_network property.

Notes

All nodes without an 'end' attribute as well as all 'weft' edges are removed by this step. Final nodes as well as final 'weft' and 'warp' edges can only be created using the mapping network.

Returns **success** (*bool*) – `True` if the mapping network has been successfully created, `False` otherwise.

Notes

Closely resembles the implementation described in *Automated Generation of Knit Patterns for Non-developable Surfaces*[1]. Also see *KnitCrete - Stay-in-place knitted formworks for complex concrete structures*[2].

create_mesh(*mode=- 1*, *max_valence=4*)

Constructs a mesh from this network by finding cycles and using them as mesh faces.

Parameters

- **mode** (*int, optional*) – Determines how the neighbors of each node are sorted when finding cycles for the network.

  `-1` equals to using the world XY plane.

  `0` equals to using a plane normal to the origin nodes closest point on the reference geometry.

  `1` equals to using a plane normal to the average of the origin and neighbor nodes' closest points on the reference geometry.

  `2` equals to using an average plane between a plane fit to the origin and its neighbor nodes and a plane normal to the origin nodes closest point on the reference geometry.

  Defaults to `-1`.

- **max_valence** (*int, optional*) – Sets the maximum edge valence of the faces. If this is set to > 4, n-gon faces (more than 4 edges) are allowed. Otherwise, their cycles are treated as invalid and will be ignored.

  Defaults to `4`.

Warning: Modes other than `-1` are only possible if this network has an underlying reference geometry in form of a Mesh or NurbsSurface. The reference geometry should be assigned when initializing the network by assigning the geometry to the "reference_geometry" attribute of the network.

find_cycles(*mode=-1*)
Finds the cycles (faces) of this network by utilizing a wall-follower mechanism.

Parameters **mode** (*int, optional*) – Determines how the neighbors of each node are sorted when finding cycles for the network. `-1` equals to using the world XY plane.

`0` equals to using a plane normal to the origin nodes closest point on the reference geometry.

`1` equals to using a plane normal to the average of the origin and neighbor nodes' closest points on the reference geometry.

`2` equals to using an average plane between a plane fit to the origin and its neighbor nodes and a plane normal to the origin nodes closest point on the reference geometry.

Defaults to `-1`.

Warning: Modes other than `-1` are only possible if this network has an underlying reference geometry in form of a Mesh or NurbsSurface. The reference geometry should be assigned when initializing the network by assigning the geometry to the "reference_geometry" attribute of the network.

Notes

Based on an implementation inside the COMPAS framework. For more info see[16].

initialize_leaf_connections()
Create all initial connections of the 'leaf' nodes by iterating over all position contours and creating 'weft' edges between the 'leaf' nodes of the position contours.

Notes

Closely resembles the implementation described in *Automated Generation of Knit Patterns for Non-developable Surfaces*[1]. Also see *KnitCrete - Stay-in-place knitted formworks for complex concrete structures*[2].

initialize_position_contour_edges()
Creates all initial position contour edges as neither 'warp' nor 'weft' by iterating over all nodes in the network and grouping them based on their 'position' attribute.

Notes

This method is automatically called when creating a KnitNetwork using the create_from_contours method!

Closely resembles the implementation described in *Automated Generation of Knit Patterns for Non-developable Surfaces*[1]. Also see *KnitCrete - Stay-in-place knitted formworks for complex concrete structures*[2].

initialize_warp_edges(*contour_set=None*, *verbose=False*)
Method for initializing first 'warp' connections once all preliminary 'weft' connections are made.

Parameters

- **contour_set** (`list`, optional) – List of lists of nodes to initialize 'warp' edges. If none are supplied, all nodes ordered by thei 'position' attributes are used.

  Defaults to `None`.

- **verbose** (*bool, optional*) – If `True`, will print verbose output to the console.

  Defaults to `False`.

Notes

Closely resembles the implementation described in *Automated Generation of Knit Patterns for Non-developable Surfaces*[1]. Also see *KnitCrete - Stay-in-place knitted formworks for complex concrete structures*[2].

initialize_weft_edges(*start_index=None*, *propagate_from_center=False*, *force_continuous_start=False*, *force_continuous_end=False*, *angle_threshold=0.10471975511965978*, *max_connections=4*, *least_connected=False*, *precise=False*, *verbose=False*)

Attempts to create all the preliminary 'weft' connections for the network.

Parameters

- **start_index** (*int, optional*) – This value defines at which index the list of contours is split. If no index is supplied, will split the list at the longest contour.

  Defaults to `None`.

- **propagate_from_center** (*bool, optional*) – If `True`, will propagate left and right set of contours from the center contour defined by start_index or the longest contour (< | >). Otherwise, the propagation of the contours left to the center will start at the left boundary (> | >).

  Defaults to `False`

- **force_continuous_start** (*bool, optional*) – If `True`, forces the first row of stitches to be continuous.

  Defaults to `False`.

- **force_continuous_end** (*bool, optional*) – If `True`, forces the last row of stitches to be continuous.

  Defaults to `False`.

- **max_connections** (*int, optional*) – The maximum connections a node is allowed to have to be considered for an additional 'weft' connection.

  Defaults to `4`.

- **least_connected** (*bool, optional*) – If `True`, uses the least connected node from the found candidates.

  Defaults to `False`

- **precise** (*bool, optional*) – If `True`, the distance between nodes will be calculated using the Rhino.Geometry.Point3d.DistanceTo method, otherwise the much faster Rhino.Geometry.Point3d.DistanceToSquared method is used.

  Defaults to `False`.

- **verbose** (*bool, optional*) – If `True`, this routine and all its subroutines will print messages about what is happening to the console. Great for debugging and analysis.

  Defaults to `False`.

Raises ***KnitNetworkError*** – If the supplied splitting index is too high.

Notes

Closely resembles the implementation described in *Automated Generation of Knit Patterns for Non-developable Surfaces*[1]. Also see *KnitCrete - Stay-in-place knitted formworks for complex concrete structures*[2].

property mapping_network
The associated mapping network of this KnitNetwork instance.

sample_segment_contours(*stitch_width*)
Samples the segment contours of the mapping network with the given stitch width. The resulting points are added to the network as nodes and a 'segment' attribute is assigned to them based on their origin segment contour edge.

Parameters **stitch_width** (*float*) – The width of a single stitch inside the knit.

Raises ***MappingNetworkError*** – If the mapping network is not available for this instance.

Notes

Closely resembles the implementation described in *Automated Generation of Knit Patterns for Non-developable Surfaces*[1]. Also see *KnitCrete - Stay-in-place knitted formworks for complex concrete structures*[2].

to_KnitDiNetwork()
Constructs and returns a directed KnitDiNetwork based on this network by duplicating all edges so that [u -> v] and [v -> u] for every edge [u - v] in this undirected network.

Returns **directed_network** (*KnitDiNetwork*) – The directed representation of this network.

traverse_weft_edges_and_set_attributes(*start_end_node*)
Traverse a path of 'weft' edges starting from an 'end' node until another 'end' node is discovered. Set 'segment' attributes to nodes and edges along the way.

start_end_node [tuple] 2-tuple representing the node to start the traversal.

2.2.4 cockatoo.KnitDiNetwork

class cockatoo.**KnitDiNetwork**(*data=None*, ***attr*)
Bases: networkx.classes.digraph.DiGraph, cockatoo._knitnetworkbase.KnitNetworkBase

Datastructure representing a directed graph of nodes aswell as 'weft' and 'warp' edges. Used in the automatic generation of knitting patterns.

Inherits from networkx.DiGraph, *KnitNetworkBase*. For more info, see *NetworkX*[13].

Notes

The implemented algorithms are strongly based on the paper *Automated Generation of Knit Patterns for Non-developable Surfaces*[1]. Also see *KnitCrete - Stay-in-place knitted formworks for complex concrete structures*[2].

The implementation was further influenced by concepts and ideas presented in the papers *Automatic Machine Knitting of 3D Meshes*[3], *Visual Knitting Machine Programming*[4] and *A Compiler for 3D Machine Knitting*[5].

ToString()
Return a textual description of the network.

Returns **description** (*str*) – A textual description of the network.

Notes

Used for overloading the Grasshopper display in data parameters.

create_mesh(*mode=-1*, *max_valence=4*)
Constructs a mesh from this network by finding cycles and using them as mesh faces.

Parameters

- **mode** (*int, optional*) – Determines how the neighbors of each node are sorted when finding cycles for the network.

  `-1` equals to using the world XY plane.

  `0` equals to using a plane normal to the origin nodes closest point on the reference geometry.

  `1` equals to using a plane normal to the average of the origin and neighbor nodes' closest points on the reference geometry.

  `2` equals to using an average plane between a plane fit to the origin and its neighbor nodes and a plane normal to the origin nodes closest point on the reference geometry.

  Defaults to `-1`.

- **max_valence** (*int, optional*) – Sets the maximum edge valence of the faces. If this is set to > 4, n-gon faces (more than 4 edges) are allowed. Otherwise, their cycles are treated as invalid and will be ignored.

  Defaults to `4`.

Warning: Modes other than `-1` are only possible if this network has an underlying reference geometry in form of a Mesh or NurbsSurface. The reference geometry should be assigned when initializing the network by assigning the geometry to the "reference_geometry" attribute of the network.

find_cycles(*mode=-1*)
Finds the cycles (faces) of this network by utilizing a wall-follower mechanism.

Parameters **mode** (*int, optional*) – Determines how the neighbors of each node are sorted when finding cycles for the network.

`-1` equals to using the world XY plane.

`0` equals to using a plane normal to the origin nodes closest point on the reference geometry.

`1` equals to using a plane normal to the average of the origin and neighbor nodes' closest points on the reference geometry.

`2` equals to using an average plane between a plane fit to the origin and its neighbor nodes and a plane normal to the origin nodes closest point on the reference geometry.

Defaults to `-1`.

Warning: Modes other than -1 (default) are only possible if this network has an underlying reference geometry in form of a Mesh or NurbsSurface. The reference geometry should be assigned when initializing the network by assigning the geometry to the "reference_geometry" attribute of the network.

Notes

Based on an implementation inside the COMPAS framework. For more info see[17].

References

make_pattern_data(*consolidate=False*)

Topological sort this network to represent it as 2d knitting pattern consisting of rows and columns.

Parameters **consolidate** (*bool*) – If `True`, will consolidate the final pattern data. Defaulst to `False`.

Returns **pattern_data** (`list` of `list`) – List (rows) of lists (column values) where every value represents a node.

Raises ***KnitNetworkTopologyError*** – if the network does not satisfy the topology constraints needed for this operation and the outcome would be unfeasible or unpredictable.

Notes

Closely resembles the implementation described in *Automated Generation of Knit Patterns for Non-developable Surfaces*[1]. Also see *KnitCrete - Stay-in-place knitted formworks for complex concrete structures*[2].

node_contour_edges(*node*, *data=False*)

Gets the incoming and outcoing edges marked neither 'warp' nor 'weft' connected to the given node.

Parameters

- **node** (*hashable*) – Hashable identifier of the node to check for incoming and outgoing edges neither 'weft' nor 'warp'.
- **data** (*bool, optional*) – If `True`, will also return the edges associated data attribute dictionary.

  Defaults to `False`.

Returns **weft_edges** (`list`) – List of incoming and outgoing edges neither 'weft' nor 'warp'.

node_contour_edges_in(*node*, *data=False*)

Gets the incoming edges marked neither 'warp' nor 'weft' connected to the given node.

Parameters

- **node** (*hashable*) – Hashable identifier of the node to check for incoming edges neither 'weft' nor 'warp'.

[17] Van Mele, Tom et al. *COMPAS: A framework for computational research in architecture and structures*. See: find_cycles() inside COMPAS

- **data** (*bool, optional*) – If `True`, will also return the edges associated data attribute dictionary.

  Defaults to `False`.

Returns **weft_edges** (`list`) – List of incoming edges neither 'weft' nor 'warp'.

node_contour_edges_out(*node*, *data=False*)
Gets the outgoing edges marked neither 'warp' nor 'weft' connected to the given node.

Parameters

- **node** (*hashable*) – Hashable identifier of the node to check for outgoing edges neither 'weft' nor 'warp'.
- **data** (*bool, optional*) – If `True`, will also return the edges associated data attribute dictionary.

  Defaults to `False`.

Returns **weft_edges** (`list`) – List of outgoing edges neither 'weft' nor 'warp'.

node_warp_edges(*node*, *data=False*)
Gets the incoming and outgoing 'warp' edges connected to the given node.

Parameters

- **node** (*hashable*) – Hashable identifier of the node to check for incoming and outgoing 'warp' edges.
- **data** (*bool, optional*) – If `True`, will also return the edges associated data attribute dictionary.

  Defaults to `False`.

Returns **weft_edges** (`list`) – List of incoming and outgoing 'warp' edges.

node_warp_edges_in(*node*, *data=False*)
Gets the incoming 'warp' edges connected to the given node.

Parameters

- **node** (*hashable*) – Hashable identifier of the node to check for incoming 'warp' edges.
- **data** (*bool, optional*) – If `True`, will also return the edges associated data attribute dictionary.

  Defaults to `False`.

Returns **weft_edges** (`list`) – List of incoming 'warp' edges.

node_warp_edges_out(*node*, *data=False*)
Gets the outgoing 'warp' edges connected to the given node.

Parameters

- **node** (*hashable*) – Hashable identifier of the node to check for outgoing 'warp' edges.
- **data** (*bool, optional*) – If `True`, will also return the edges associated data attribute dictionary.

  Defaults to `False`.

Returns **weft_edges** (`list`) – List of outgoing 'warp' edges.

node_weft_edges(*node*, *data=False*)
Gets incoming and outgoing 'weft' edges connected to the given node.

Parameters

- **node** (*hashable*) – Hashable identifier of the node to check for incoming and outgoing 'weft' edges.
- **data** (*bool, optional*) – If `True`, will also return the edges associated data attribute dictionary.

  Defaults to `False`.

Returns **weft_edges** (`list`) – List of incoming and outgoing 'weft' edges.

node_weft_edges_in(*node*, *data=False*)

Gets the incoming 'weft' edges connected to the given node.

Parameters

- **node** (*hashable*) – Hashable identifier of the node to check for incoming 'weft' edges.
- **data** (*bool, optional*) – If `True`, will also return the edges associated data attribute dictionary.

  Defaults to `False`.

Returns **weft_edges** (`list`) – List of incoming 'weft' edges.

node_weft_edges_out(*node*, *data=False*)

Gets the outgoing 'weft' edges connected to the given node.

Parameters

- **node** (*hashable*) – Hashable identifier of the node to check for outgoing 'weft' edges.
- **data** (*bool, optional*) – If `True`, will also return the edges associated data attribute dictionary.

  Defaults to `False`.

Returns **weft_edges** (`list`) – List of outgoing 'weft' edges.

verify_dual_form()

Verifies this network to have the correct form of a dual as needed for representing this network as a 2d knitting pattern.

Returns *bool* – `True` on success, `False` otherwise.

2.2.5 cockatoo.KnitMappingNetwork

class cockatoo.**KnitMappingNetwork**(*data=None*, ***attr*)

Bases: `networkx.classes.multigraph.MultiGraph`, `cockatoo._knitnetworkbase.KnitNetworkBase`

Datastructure representing a mapping between connected chains of 'weft' edges in a KnitNetwork for final creation of 'weft' and 'warp' edges.

Inherits from `networkx.MultiGraph`, *KnitNetworkBase* For more info, see *NetworkX*[13].

Notes

Not intended to be instantiated separately. Should only be instantiated by the KnitNetwork.create_mapping_network method!

The implemented algorithms are strongly based on the paper *Automated Generation of Knit Patterns for Non-developable Surfaces*[1]. Also see *KnitCrete - Stay-in-place knitted formworks for complex concrete structures*[2].

The implementation was further influenced by concepts and ideas presented in the papers *Automatic Machine Knitting of 3D Meshes*[3], *Visual Knitting Machine Programming*[4] and *A Compiler for 3D Machine Knitting*[5].

ToString()
Return a textual description of the network.

Returns **description** (*str*) – A textual description of the network.

Notes

Used for overloading the Grasshopper display in data parameters.

build_chains(*source_as_dict=False*, *target_as_dict=False*)
Method for building source and target chains from segment contour edges.

Parameters

- **source_as_dict** (*bool*) – If `True`, will return the source chains as a dictionary indexed by their chain value.
- **target_as_dict** (*bool*) – If `True`, will return the target chains as a dictionary indexed by their chain value.

Returns **chains** (`tuple` of `list`) – 2-tuple in the form of (source_chains, target_chains).

traverse_segments_until_warp(*way_segments*, *down=False*, *by_end=False*)
Method for traversing a path of 'segment' edges until a 'warp' edge is discovered which points to the previous or the next segment. Returns the ids of the segment array.

Parameters

- **way_segments** (`list`) – List of segments that is filled during method execution. The list should contain the start segment when calling this method!
- **down** (*bool, optional*) – If `True`, will traverse until a downwards 'warp' edge is discovered, otherwise will traverse antil an upwards 'warp' edge is discovered.

  Defaults to `False`
- **by_end** (*bool, optional*) – If `True`, will traverse the 'segment' edges in the opposite direction.

  Defaults to `False`.

Returns **segments** (`list`) – List of segments representing a chain.

Raises **ValueError:** – If `way_segments` is empty at call.

CHAPTER

THREE

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

C

INDEX

K

L

M

N

P

R

S

T

V

W