cadnano Documentation

Release 2.5.0

NJ Conway, SM Douglas

July 26, 2016

cadnano is a computer-aided design tool for creating DNA nanostructures.

Warning: These docs are under development as part of the upcoming cadnano 2.5 release. Some information, such as installation steps, may not be useful prior to the official release.

CHAPTER 1

Quickstart

Installation

Install Python 3.5 , followed by cadnano.

Usage

To launch the cadnano GUI from the command line:

```
python3 cadnano
```

To import cadnano as a Python module and work without a GUI:

```
import cadnano
```

CHAPTER 2

Installation

2.1 Install Python 3.5

There are many ways to get Python on your system.

- If you prefer the complete "batteries included" option, Anaconda Python is available for Mac, Windows or Linux.
- If you just want a clean Python install and nothing else, installers from python.org work great too.
- On Mac, [homebrew](http://brew.sh/) is another great way to install Python.

Cadnano will run on Python 3.3, 3.4 and 3.5, but we only support 3.5. We do not support Python 2.X.

2.2 Install cadnano

1. Install SIP and PyQt5 as the dependencies.

Currently (6/27/2016), `PyQt5` v 5.6 there is a bug with `QGraphicsItem.itemChange` that prevents us from being able to use the wheels on pypi. We've built our own versions for Windows and Mac OS X for Python 3.5 64 bit that don't have this problem:

- Mac Python 3.5 SIP 64-bit
- Mac Python 3.5 PyQt5 64-bit

And install with:

```
pip install sip-4.16.9-cp35-none-macosx_10_10_x86_64.whl
pip install PyQt5-5.5.1-cp35-none-macosx_10_10_x86_64.whl
```

- Windows Python 3.5 SIP 64-bit
- Windows Python 3.5 PyQt5 64-bit

And install with:

```
pip install sip-4.18.1-cp35-none-win_amd64.whl
pip install PyQt5-5.6.1-cp35-none-win_amd64.whl
```

Ultimately, we plan to maintain `PyQt5` wheel builds that work on Linux, Windows and OS X for Python 3.4+ so you can rely on a single version of Python 3.X. This of course is subject to change, and will change this notice once the official wheels are rebuilt with the bug fix.

2. Then clone the cadnano source from github and run:

```
python setup.py install
```

Once things are stable, we will distribute a cadnano wheel and start bundling cross platform installers. For now have fun running radnano.

2.3 Advanced: Building from scratch

Should the above not work for you:

1. Let us know
2. The requirements `PyQt5` and `SIP` are available from Riverbank Computing Limited at:

- Qt5
- PyQt5 downloads
- SIP downloads

2.3.1 Windows

Instructions to come.

2.3.2 Mac and Linux

These instructions can work 10.10 Yosemite and 10.11 El Capitan under Xcode 7.0.1 and 6.5. It has also been tested on Debian 7.9 Wheezy Please provide feedback if you have problems running this in issues.

You can run the included `pyqt5_check.py` which will grab, build and install `Qt5`, `SIP` and `PyQt5` in your python environment. It is cleanest using `virtualenv` and `virtualenvwrapper` creating a virtualenv with:

```
mkvirtualenv --always-copy <myvenv>
python pyqttools/install_pyqt_from_src.py
```

and then running the script, but you can definitely install in your system python if you run:

```
sudo python pyqttools/install_pyqt_from_src.py
```

This script only builds required parts of `Qt5` and `PyQt5` in the interest of time.

Manual installation of `PyQt5` is fine too, but you'll need to trouble shoot on your own

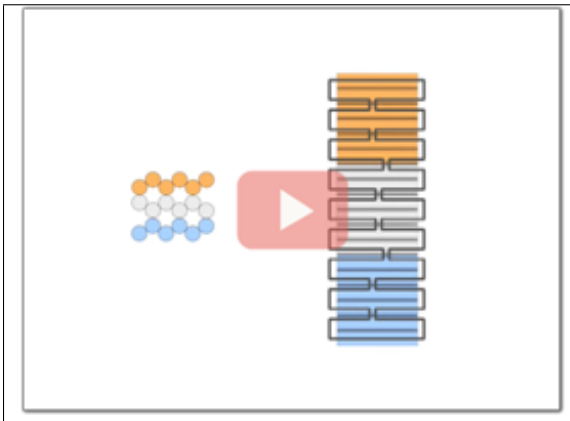
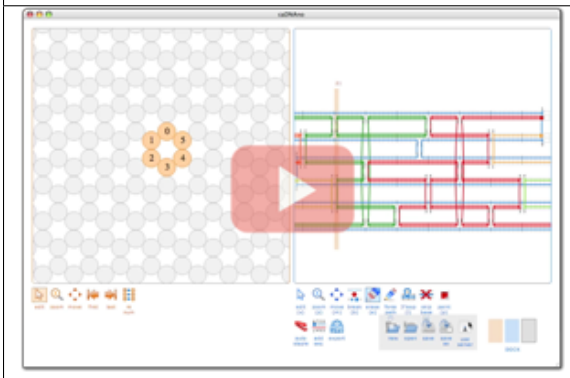
1. Install `Qt5`. download the online installer
2. Build sip and PyQt5 against this `Qt5`

Of course there are many ways to accomplish this feat, but needless to say OS X and Linux installs of `PyQt5` can be painful for some people.

CHAPTER 3

Tutorial

We have planned to create some new tutorials to add here. In the meantime, you can start by checking out the original tutorials that were released for the original version of cadnano.

	Tutorial 1: How DNA geometry relates to building 3D shapes with a honeycomb lattice
	Tutorial 2: How to create a basic shape in cadnano export the DNA staple sequences

3.1 Understanding the interface

The basic knowledge of DNA geometry necessary to understand the interface is covered by Tutorial 1. The new tutorial planned for this section will explain the latest version of the GUI interface in that context, and walk through what each of the tools do.

3.2 Creating a new design

We are also planning an updated version of Tutorial 2, which will be a step-by-step design, start to finish.

CHAPTER 4

API

4.1 cadnano.color module

class cadnano.color.**Color**(**args*)
Bases: object

hex()

name()

setAlpha(*a*)

setRgb(*r*, *g*, *b*, *a=255*)

cadnano.color.**intToColor**(*color_number*)
legacy color support

cadnano.color.**intToColorHex**(*color_number*)

4.2 cadnano.cnobject module

class cadnano.cnobject.**CNObject**(*parent*)
Bases: *cadnano.cnproxy.ProxyObject*

undoStack()

4.3 cadnano.cnproxy module

cadnano.cnproxy.**BaseObject**
alias of *ProxyObject*

class cadnano.cnproxy.**DummySignal**(**args*, ***kwargs*)
Bases: object

connect(*target*)

disconnect(*target*)

emit(**args*)

class cadnano.cnproxy.**ProxyObject**(*parent*)
Bases: object

connect(*sender*, *bsignal*, *method*)

deleteLater()

disconnect(*sender*, *bsignal*, *method*)

parent()

setParent(*parent*)

signals()

cadnano.cnproxy.**ProxySignal**
alias of *DummySignal*

class cadnano.cnproxy.**TempApp**
Bases: object

documentWasCreatedSignal = <cadnano.cnproxy.DummySignal object>

4.4 cadnano.decorators package

4.4.1 Submodules

cadnano.decorators.insertion module

class cadnano.decorators.insertion.**Insertion**(*index*, *length*)
Bases: object

Insertions do affect an applied sequence and do not store a sequence themselves. They are a skip if the length is less than 0

Parameters

- **index** (*int*) – the index into the *StrandSet* the *Insertion* occurs at
- **length** (*int*) – length of *Insertion*

idx()

Returns the index into the *StrandSet* the *Insertion* occurs at

Return type int

isSkip()

Returns True is is a skip, False otherwise

Return type bool

length()
This is the length of a sequence that is immutable by the strand

Returns length of *Insertion*

Return type int

setLength(*length*)
Setter for the length

Parameters **length** (*int*) –

updateIdx(*delta*)
Increment the index by delta

Parameters **delta** (*int*) – can be negative

4.4.2 Module contents

4.5 cadnano.document module

4.6 cadnano.enum module

class cadnano.enum.**BreakType**
Bases: object

LEFT3PRIME = 1

LEFT5PRIME = 0

RIGHT3PRIME = 3

RIGHT5PRIME = 2

class cadnano.enum.**EndType**
Bases: object

FIVE_PRIME = 0

THREE_PRIME = 1

class cadnano.enum.**Enum**
Bases: enum.Enum

bwv

class cadnano.enum.**EnumMask**(*enum*, *value*)
Bases: object

class cadnano.enum.**GridType**
Bases: object

HONEYCOMB = 2

NONE = 0

SQUARE = 1

class cadnano.enum.**HandleOrient**
Bases: object

LEFT_DOWN = 2

LEFT_UP = 0

RIGHT_DOWN = 3

RIGHT_UP = 1

class cadnano.enum.**ItemType**
Bases: enum.Enum

class cadnano.enum.**LatticeType**
Bases: object

HONEYCOMB = 1

SQUARE = 0

class cadnano.enum.**ModType**
Bases: object

END_3PRIME = 1

END_5PRIME = 0

INTERNAL = 2

class cadnano.enum.**Parity**
Bases: object

EVEN = 0

ODD = 1

class cadnano.enum.**PartEdges**
Bases: *cadnano.enum.Enum*

class cadnano.enum.**PartType**
Bases: enum.Enum

class cadnano.enum.**PointType**
Bases: object

ARBITRARY = 1

Z_ONLY = 0

class cadnano.enum.**StrandType**
Bases: object

FWD = 0

REV = 1

SCAFFOLD = 0

STAPLE = 1

cadnano.enum.**enumNames**(*cls*)

4.7 cadnano.fileio package

4.7.1 Submodules

cadnano.fileio.c25decode module

cadnano.fileio.lattice module

class cadnano.fileio.lattice.**HoneycombDnaPart**
Bases: object

SCAF_LOW = [[1, 11], [8, 18], [4, 15]] SCAF_HIGH = [[2, 12], [9, 19], [5, 16]] STAP_LOW = [[6, 16], [3, 13], [10, 20]] STAP_HIGH = [[7, 17], [4, 14], [0, 11]]

from 0: DR U DL aka 210 90 330 SCAF_LOW = [[1, 12], [8, 19], [5, 15]] SCAF_HIGH = [[2, 13], [9, 20], [6, 16]] STAP_LOW = [[17], [3], [10]] STAP_HIGH = [[18], [4], [11]]

HELICAL_PITCH = 10.5

STEP = 21

SUB_STEP_SIZE = 7.0

TURNS_PER_STEP = 2.0

TWIST_OFFSET = -34.285714285714285

TWIST_PER_BASE = 34.285714285714285

static isEvenParity(*row*, *column*)

static isOddParity(*row*, *column*)

static latticeCoordToPositionXY(*radius*, *row*, *column*, *scale_factor=1.0*)
make sure radius is a float

static positionToLatticeCoord(*radius*, *x*, *y*, *scale_factor=1.0*)

static positionToLatticeCoordRound(*radius*, *x*, *y*, *round_up_row*, *round_up_col*, *scale_factor=1.0*)

class cadnano.fileio.lattice.**SquareDnaPart**
Bases: object

SCAF_LOW = [[4, 26, 15], [18, 28, 7], [10, 20, 31], [2, 12, 23]] SCAF_HIGH = [[5, 27, 16], [19, 29, 8], [11, 21, 0], [3, 13, 24]] STAP_LOW = [[31], [23], [15], [7]] STAP_HIGH = [[0], [24], [16], [8]]

HELICAL_PITCH = 10.666666666666666

STEP = 32

SUB_STEP_SIZE = 8.0

TURNS_PER_STEP = 3.0

TWIST_OFFSET = 196.875

TWIST_PER_BASE = 33.75

static isEvenParity(*row*, *column*)

static isOddParity(*row*, *column*)

static latticeCoordToPositionXY(*radius*, *row*, *column*, *scale_factor=1.0*)

static positionToLatticeCoord(*radius*, *x*, *y*, *scale_factor=1.0*)

static positionToLatticeCoordRound(*radius*, *x*, *y*, *scale_factor=1.0*)

cadnano.fileio.nnodecode module

cadnano.fileio.nnoencode module

cadnano.fileio.v2decode module

cadnano.fileio.v3decode module

cadnano.fileio.v3encode module

cadnano.fileio.v3encode.**encodeDocument**(*document*)

Parameters **document** (*Document*) –

Returns

Return type dict

cadnano.fileio.v3encode.**encodePart**(*part*)

Parameters **part** (*Part*) –

Returns dict

cadnano.fileio.v3encode.**encodePartList**(*part*, *vh_group_list*)

Used for copying and pasting TODO: unify encodePart and encodePartList

Parameters

- **part** (*Part*) –
- **vh_group_list** (*List[int]*) – list of virtual_helices to encode to be used with copy and paste serialization

Returns dict

4.7.2 Module contents

4.8 cadnano.math package

4.8.1 Submodules

cadnano.math.box module

class cadnano.math.box.**Box**(*min_point*, *max_point*)

Bases: object

Cube box object

For doing an oct tree type thing

Parameters

- **min_point** (*Tuple*) – length 3 lower left corner
- **max_point** (*Tuple*) – length 3 diagonal opposite top corner

center()

Return the center of this Box

Returns the center point of this box.

Return type *Vector3*

clone()

Clone this Box

Returns a copy of this box.

Return type *Box*

containsBox(*box*)

Does this object contain the Box box?

Parameters **Box** –

Returns True if *box* is in *self* otherwise False

Return type bool

containsPoint(*point*)
Is the *point* within this Box?

Parameters **point** (Vector3) – to check for inclusion.

Returns True if the specified point lies within the boundaries of this box False otherwise

Return type bool

doesBoxSpan(*box*)
doe this object contain the Box *box*? :param Box:

Returns True if *box* spans *self* otherwise False

Return type bool

set(*min_point*, *max_point*)
Set the Tuples

Setter for setting the bounding points of the Box

Parameters

- **min_point** (*Tuple*) – length 3 lower left corner
- **max_point** (*Tuple*) – length 3 diagonal opposite top corner

size()
Find the dimensions of the Box

Returns the width, height, and depth of this box.

Return type Tuple

cadnano.math.face module

class cadnano.math.face.**Face**(*normal*, *v1*, *v2*, *v3*)
Bases: tuple

namedtuple of tuple: 4 x 3 tuple of tuples corresponding to the normal and the three points comprising a *Face*

__getnewargs__()
Return self as a plain tuple. Used by copy and pickle.

__getstate__()
Exclude the OrderedDict from pickling

static __new__(*_cls*, *normal*, *v1*, *v2*, *v3*)
Create new instance of Face(normal, v1, v2, v3)

__repr__()
Return a nicely formatted representation string

normal
Alias for field number 0

v1
Alias for field number 1

v2
Alias for field number 2

v3
Alias for field number 3

cadnano.math.matrix3 module

class cadnano.math.matrix3.**Matrix3**(*n11*, *n12*, *n13*, *n21*, *n22*, *n23*, *n31*, *n32*, *n33*)
Bases: tuple

namedtuple: 3 x 3 matrix

__getnewargs__()
Return self as a plain tuple. Used by copy and pickle.

__getstate__()
Exclude the OrderedDict from pickling

static __new__(*_cls*, *n11*, *n12*, *n13*, *n21*, *n22*, *n23*, *n31*, *n32*, *n33*)
Create new instance of Matrix3(n11, n12, n13, n21, n22, n23, n31, n32, n33)

__repr__()
Return a nicely formatted representation string

n11
Alias for field number 0

n12
Alias for field number 1

n13
Alias for field number 2

n21
Alias for field number 3

n22
Alias for field number 4

n23
Alias for field number 5

n31
Alias for field number 6

n32
Alias for field number 7

n33
Alias for field number 8

cadnano.math.matrix3.**getInverse**(*m4*)

Parameters **m4** (Matrix4) –

Returns

Return type *Matrix3*

cadnano.math.matrix3.**getNormalMatrix**(*m*)
Normalize the matrix *m* :param m:

Returns Matrix3

cadnano.math.matrix3.**transpose**(*m*)
Compute the inverse of *m*

Parameters **m** (Matrix3) –

Returns the inverse

Return type *Matrix3*

cadnano.math.matrix4 module

class cadnano.math.matrix4.**Matrix4**(*n11*, *n12*, *n13*, *n14*, *n21*, *n22*, *n23*, *n24*, *n31*, *n32*, *n33*, *n34*, *n41*, *n42*, *n43*, *n44*)

Bases: tuple

namedtuple: 4 x 4 matrix

__getnewargs__()
Return self as a plain tuple. Used by copy and pickle.

__getstate__()
Exclude the OrderedDict from pickling

static __new__(*_cls*, *n11*, *n12*, *n13*, *n14*, *n21*, *n22*, *n23*, *n24*, *n31*, *n32*, *n33*, *n34*, *n41*, *n42*, *n43*, *n44*)
Create new instance of Matrix4(n11, n12, n13, n14, n21, n22, n23, n24, n31, n32, n33, n34, n41, n42, n43, n44)

__repr__()
Return a nicely formatted representation string

n11
Alias for field number 0

n12
Alias for field number 1

n13
Alias for field number 2

n14
Alias for field number 3

n21
Alias for field number 4

n22
Alias for field number 5

n23
Alias for field number 6

n24
Alias for field number 7

n31
Alias for field number 8

n32
Alias for field number 9

n33
Alias for field number 10

n34
Alias for field number 11

n41
Alias for field number 12

n42
Alias for field number 13

n43
Alias for field number 14

n44
Alias for field number 15

cadnano.math.matrix4.**makeRotationZ**(*theta*)
Create a rotation matrix of angle *theta*

Parameters **theta** (*float*) – Angle in radians

Returns rotation matrix about the Z axis

Return type *Matrix4*

cadnano.math.matrix4.**makeTranslation**(*x*, *y*, *z*)
create a translation matrix given a displacement x, y, z

Parameters

- **x** (*float*) –
- **y** (*float*) –
- **z** (*float*) –

Returns translation matrix

Return type *Matrix4*

cadnano.math.solid module

class cadnano.math.solid.**Solid**(*name*)
Bases: object

addFace(*v1*, *v2*, *v3*, *normal=None*)
List vertices using right hand rule so that unit normal will point out of the surface

vertices are given by index into vertices list

Parameters

- **v1** (Vector3) –
- **v2** (Vector3) –
- **v3** (Vector3) –
- **normal** (Vector3) – face normal

addVertex(*vertex*)
Add a vertex to the Solid :param vertex:

applyMatrix(*matrix4*)

computeFaceNormals()

cadnano.math.vector module

class cadnano.math.vector.**Vector2**(*x*, *y*)
Bases: tuple

__getnewargs__()
Return self as a plain tuple. Used by copy and pickle.

__getstate__()
Exclude the OrderedDict from pickling

static __new__(*_cls*, *x*, *y*)
Create new instance of Vector2(x, y)

__repr__()
Return a nicely formatted representation string

x
Alias for field number 0

y
Alias for field number 1

class cadnano.math.vector.**Vector3**(*x*, *y*, *z*)
Bases: tuple

__getnewargs__()
Return self as a plain tuple. Used by copy and pickle.

__getstate__()
Exclude the OrderedDict from pickling

static __new__(*_cls*, *x*, *y*, *z*)
Create new instance of Vector3(x, y, z)

__repr__()
Return a nicely formatted representation string

x
Alias for field number 0

y
Alias for field number 1

z
Alias for field number 2

cadnano.math.vector.**addVectors**(*v1*, *v2*)

cadnano.math.vector.**applyMatrix3**(*m*, *v*)

cadnano.math.vector.**applyMatrix4**(*m*, *v*)

cadnano.math.vector.**crossProduct**(*a*, *b*)
return normalized cross product

cadnano.math.vector.**multiplyScalar**(*v*, *s*)
return v1*s

cadnano.math.vector.**normalToPlane**(*v1*, *v2*, *v3*)
Calculate unit normal to the normal to the plane defined by vertices v1, v2, and v3

cadnano.math.vector.**normalizeV2**(*v*)

cadnano.math.vector.**normalizeV3**(*v*)

cadnano.math.vector.**subVectors**(*v1*, *v2*)
return v1 - v2

cadnano.math.vector.**v2AngleBetween**(*a*, *b*)

cadnano.math.vector.**v2DistanceAndAngle**(*a*, *b*)

cadnano.math.vector.**v2dot**(*a*, *b*)

cadnano.math.vector.**v3SetX**(*v*, *x*)

cadnano.math.vector.**v3SetY**(*v*, *y*)

cadnano.math.vector.**v3SetZ**(*v*, *z*)

4.8.2 Module contents

4.9 cadnano.objectinstance module

class cadnano.objectinstance.**ObjectInstance**(*reference_object*, *parent=None*)
Bases: *cadnano.cnobject.CNObject*

deepCopy(*reference_object*, *parent*)

destroy()

instanceDestroyedSignal = <cadnano.cnproxy.DummySignal object>

instanceMovedSignal = <cadnano.cnproxy.DummySignal object>

instanceParentChangedSignal = <cadnano.cnproxy.DummySignal object>

parent()

position()

reference()

shallowCopy()

unwipe(*doc*)
For Removing ObjectInstances from a Document

wipe(*doc*)
For adding ObjectInstances to a Document

4.10 cadnano.oligo package

4.10.1 Submodules

cadnano.oligo.applycolorcmd module

class cadnano.oligo.applycolorcmd.**ApplyColorCommand**(*oligo*, *color*)
Bases: *cadnano.undocommand.UndoCommand*

redo()

undo()

cadnano.oligo.applysequencecmd module

class cadnano.oligo.applysequencecmd.**ApplySequenceCommand**(*oligo*, *sequence*)
Bases: *cadnano.undocommand.UndoCommand*

redo()

undo()

cadnano.oligo.oligo module

class cadnano.oligo.oligo.**Oligo**(*part*, *color=None*)
Bases: *cadnano.cnobject.CNObject*

Oligo is a group of Strands that are connected via 5' and/or 3' connections. It corresponds to the physical DNA strand, and is thus used tracking and storing properties that are common to a single strand, such as its color.

Commands that affect Strands (e.g. create, remove, merge, split) are also responsible for updating the affected Oligos.

addToPart(*part*)

applyAbstractSequences()

applyColor(*color*, *use_undostack=True*)

applySequence(*sequence*, *use_undostack=True*)

applySequenceCMD(*sequence*, *use_undostack=True*)

clearAbstractSequences()

decrementLength(*delta*)

destroy()

displayAbstractSequences()

dump()
Return tuple of this oligo and its properties. It's expected that caller will copy the properties if mutating

editable_properties = ['name', 'color']

getColor()

getName()

getOutlineProperties()

getProperty(*key*)

getPropertyDict()

incrementLength(*delta*)

isLoop()

length()

locString()

oligoIdentityChangedSignal = <cadnano.cnproxy.DummySignal object>

oligoPropertyChangedSignal = <cadnano.cnproxy.DummySignal object>

oligoRemovedSignal = <cadnano.cnproxy.DummySignal object>

oligoSequenceAddedSignal = <cadnano.cnproxy.DummySignal object>

oligoSequenceClearedSignal = <cadnano.cnproxy.DummySignal object>

part()

refreshLength()

remove(*use_undostack=True*)

removeFromPart()
This method merely disconnects the object from the model. It still lives on in the undoStack until clobbered

Note: don't set self._part = None because we need to continue passing the same reference around.

sequence()

sequenceExport()

setColor(*color*)

setLength(*length*)

setLoop(*bool*)

setPart(*part*)

setProperty(*key*, *value*)

setStrand5p(*strand*)

shallowCopy()

shouldHighlight()

strand3p()

strand5p()

strandMergeUpdate(*old_strand_low*, *old_strand_high*, *new_strand*)
This method sets the isLoop status of the oligo and the oligo's 5' strand.

strandResized(*delta*)
Called by a strand after resize. Delta is used to update the length, which may case an appearance change.

strandSplitUpdate(*new_strand5p*, *new_strand3p*, *oligo3p*, *old_merged_strand*)
If the oligo is a loop, splitting the strand does nothing. If the oligo isn't a loop, a new oligo must be created and assigned to the new_strand and everything connected to it downstream.

undoStack()

cadnano.oligo.removeoligocmd module

class cadnano.oligo.removeoligocmd.**RemoveOligoCommand**(*oligo*)
Bases: *cadnano.undocommand.UndoCommand*

redo()

undo()

4.10.2 Module contents

4.11 cadnano.part package

4.11.1 Submodules

cadnano.part.changeviewpropertycmd module

cadnano.part.createvhelixcmd module

cadnano.part.nucleicacidpart module

cadnano.part.part module

cadnano.part.refresholigoscmd module

cadnano.part.refreshsegmentscmd module

cadnano.part.removepartcmd module

cadnano.part.removevhelixcmd module

cadnano.part.renumbercmd module

cadnano.part.resizevirtualhelixcmd module

cadnano.part.translatevhelixcmd module

cadnano.part.virtualhelixgroup module

cadnano.part.xovercmds module

4.11.2 Module contents

4.12 cadnano.preferences module

4.13 cadnano.proxyconfigure module

cadnano.proxyconfigure.**proxyConfigure**(*signal_type=None*)
call once per application at the start of the import chain

4.14 cadnano.strand package

4.14.1 Submodules

cadnano.strand.insertioncmd module

class cadnano.strand.insertioncmd.**AddInsertionCommand**(*strand*, *idx*, *length*)
Bases: *cadnano.undocommand.UndoCommand*

redo()

undo()

class cadnano.strand.insertioncmd.**ChangeInsertionCommand**(*strand*, *idx*, *new_length*)
Bases: *cadnano.undocommand.UndoCommand*

Changes the length of an insertion to a non-zero value the caller of this needs to handle the case where a zero length is required and call RemoveInsertionCommand

redo()

undo()

class cadnano.strand.insertioncmd.**RemoveInsertionCommand**(*strand*, *idx*)
Bases: *cadnano.undocommand.UndoCommand*

redo()

undo()

cadnano.strand.modscmd module

class cadnano.strand.modscmd.**AddModsCommand**(*document*, *strand*, *idx*, *mod_id*)
Bases: *cadnano.undocommand.UndoCommand*

redo()

undo()

class cadnano.strand.modscmd.**RemoveModsCommand**(*document*, *strand*, *idx*, *mod_id*)
Bases: *cadnano.undocommand.UndoCommand*

redo()

undo()

cadnano.strand.resizecmd module

class cadnano.strand.resizecmd.**ResizeCommand**(*strand*, *new_idxs*, *update_segments=True*)
Bases: *cadnano.undocommand.UndoCommand*

redo()

undo()

cadnano.strand.strand module

class cadnano.strand.strand.**Strand**(*strandset*, *base_idx_low*, *base_idx_high*, *oligo=None*)
Bases: *cadnano.cnobject.CNObject*

A Strand is a continuous stretch of bases that are all in the same StrandSet (recall: a VirtualHelix is made up of two StrandSets).

Every Strand has two endpoints. The naming convention for keeping track of these endpoints is based on the relative numeric value of those endpoints (low and high). Thus, Strand has a '_base_idx_low', which is its index with the lower numeric value (typically positioned on the left), and a '_base_idx_high' which is the higher-value index (typically positioned on the right)

Strands can be linked to other strands by "connections". References to connected strands are named "_strand5p" and "_strand3p", which correspond to the 5' and 3' phosphate linkages in the physical DNA strand, respectively. Since Strands can point 5'-to-3' in either the low-to-high or high-to-low directions, connection accessor methods (connectionLow and connectionHigh) are bound during the init for convenience.

addDecorators(*additionalDecorators*)
Used to add decorators during a merge operation.

addInsertion(*idx*, *length*, *use_undostack=True*)
Adds an insertion or skip at idx. length should be

>0 for an insertion -1 for a skip

addMods(*document*, *mod_id*, *idx*, *use_undostack=True*)
Used to add mods during a merge operation.

applyAbstractSequence()
Assigns virtual index from 5' to 3' on strand and it's complement location.

canInstallXoverAt(*idx*, *from_strand*, *from_idx*)
Assumes idx is: self.lowIdx() <= idx <= self.highIdx()

canResizeTo(*new_low*, *new_high*)
Checks to see if a resize is allowed. Similar to getResizeBounds but works for two bounds at once.

changeInsertion(*idx*, *length*, *use_undostack=True*)

clearAbstractSequence()

clearDecoratorCommands()

clearInsertionsCommands(*insertions*, *idxL*, *idxH*)
clear out insertions in this range

connection3p()

connection5p()

copyAbstractSequenceToSequence()

decorators()

deepCopy(*strandset*, *oligo*)
can't use python module 'copy' as the dictionary _decorators needs to be shallow copied as well, but wouldn't be if copy.copy() is used, and copy.deepcopy is undesired

TODO: consider renaming this method

destroy()

document()

dump5p()

generator3pStrand()
Iterate from self to the final _strand3p is None 5prime to 3prime Includes originalCount to check for circular linked list

generator5pStrand()
Iterate from self to the final _strand5p is None 3prime to 5prime Includes originalCount to check for circular linked list

getColor()

getComplementStrands()
return the list of complement strands that overlap with this strand

getPreDecoratorIdxList()
Return positions where predecorators should be displayed. This is just a very simple check for the presence of xovers on the strand.

Will refine later by checking for lattice neighbors in 3D.

getRemoveInsertionCommands(*new_idxs*)
Removes Insertions, Decorators, and Modifiers that have fallen out of range of new_idxs.

For insertions, it finds the ones that have neither Staple nor Scaffold strands at the insertion idx as a result of the change of this strand to new_idxs

getResizeBounds(*idx*)
Determines (inclusive) low and high drag boundaries resizing from an endpoint located at idx.

When resizing from _base_idx_low: low bound is determined by checking for lower neighbor strands. high bound is the index of this strand's high cap, minus 1.

When resizing from _base_idx_high: low bound is the index of this strand's low cap, plus 1. high bound is determined by checking for higher neighbor strands.

When a neighbor is not present, just use the Part boundary.

getSequenceList()
return the list of sequences strings comprising the sequence and the inserts as a tuple with the index of the insertion [(idx, (strandItemString, insertionItemString), ...]

This takes advantage of the fact the python iterates a dictionary by keys in order so if keys are indices, the insertions will iterate out from low index to high index

hasInsertion()
Iterate through dict of insertions for this strand's virtualhelix and return True of any of the indices overlap with the strand.

hasInsertionAt(*idx*)

hasXoverAt(*idx*)
An xover is necessarily at an enpoint of a strand

highIdx()

idNum()

idx3Prime()
Returns the absolute base_idx of the 3' end of the strand. overloaded in __init__

idx5Prime()
Returns the absolute base_idx of the 5' end of the strand. overloaded in __init__

idxs()

insertionLengthBetweenIdxs(*idxL*, *idxH*)
includes the length of insertions in addition to the bases

insertionsOnStrand(*idxL=None*, *idxH=None*)
if passed indices it will use those as a bounds

isForward()

length()

lowIdx()

merge(*idx*)
Check for neighbor, then merge if possible.

modifersOnStrand()

oligo()

part()

reapplySequence()

removeInsertion(*idx*, *use_undostack=True*)

removeMods(*document*, *mod_id*, *idx*, *use_undostack=True*)
Used to add mods during a merge operation.

resize(*new_idxs*, *use_undostack=True*, *update_segments=True*)

sequence(*for_export=False*)

setComplementSequence(*sequence_string*, *strand*)
This version takes anothers strand and only sets the indices that align with the given complimentary strand

return the used portion of the sequence_string

As it depends which direction this is going, and strings are stored in memory left to right, we need to test for is_forward to map the reverse compliment appropriately, as we traverse overlapping strands.

We reverse the sequence ahead of time if we are applying it 5' to 3', otherwise we reverse the sequence post parsing if it's 3' to 5'

Again, sequences are stored as strings in memory 5' to 3' so we need to jump through these hoops to iterate 5' to 3' through them correctly

Perhaps it's wiser to merely store them left to right and reverse them at draw time, or export time

setConnection3p(*strand*)

setConnection5p(*strand*)

setIdxs(*idxs*)

setOligo(*new_oligo*, *emit_signal=True*)

setSequence(*sequence_string*)
Applies sequence string from 5' to 3' return the tuple (used, unused) portion of the sequence_string

setStrandSet(*strandset*)

shallowCopy()
can't use python module 'copy' as the dictionary _decorators needs to be shallow copied as well, but wouldn't be if copy.copy() is used, and copy.deepcopy is undesired

split(*idx*, *update_sequence=True*)
Called by view items to split this strand at idx.

strandFilter()

strandHasNewOligoSignal = <cadnano.cnproxy.DummySignal object>

strandInsertionAddedSignal = <cadnano.cnproxy.DummySignal object>

strandInsertionChangedSignal = <cadnano.cnproxy.DummySignal object>

strandInsertionRemovedSignal = <cadnano.cnproxy.DummySignal object>

strandModifierAddedSignal = <cadnano.cnproxy.DummySignal object>

strandModifierChangedSignal = <cadnano.cnproxy.DummySignal object>

strandModifierRemovedSignal = <cadnano.cnproxy.DummySignal object>

strandModsAddedSignal = <cadnano.cnproxy.DummySignal object>

strandModsChangedSignal = <cadnano.cnproxy.DummySignal object>

strandModsRemovedSignal = <cadnano.cnproxy.DummySignal object>

strandRemovedSignal = <cadnano.cnproxy.DummySignal object>

strandResizedSignal = <cadnano.cnproxy.DummySignal object>

strandSelectedChangedSignal = <cadnano.cnproxy.DummySignal object>

strandSet()

strandType()

strandUpdateSignal = <cadnano.cnproxy.DummySignal object>

strandXover5pChangedSignal = <cadnano.cnproxy.DummySignal object>

strandXover5pRemovedSignal = <cadnano.cnproxy.DummySignal object>

totalLength()
includes the length of insertions in addition to the bases

updateIdxs(*delta*)

cadnano.strand.strand.**sixb**(*x*)

cadnano.strand.strand.**tostring**(*x*)

4.14.2 Module contents

4.15 cadnano.strandset package

4.15.1 Submodules

cadnano.strandset.createstrandcmd module

class cadnano.strandset.createstrandcmd.**CreateStrandCommand**(*strandset*, *base_idx_low*, *base_idx_high*, *color*, *update_segments=True*)

Bases: *cadnano.undocommand.UndoCommand*

Create a new Strand based with bounds (base_idx_low, base_idx_high), and insert it into the strandset at position strandset_idx. Also, create a new Oligo, add it to the Part, and point the new Strand at the oligo.

redo()

undo()

cadnano.strandset.mergecmd module

class cadnano.strandset.mergecmd.**MergeCommand**(*strand_low*, *strand_high*, *priority_strand*)
Bases: *cadnano.undocommand.UndoCommand*

This class takes two Strands and merges them. This Class should be private to StrandSet as knowledge of a strandsetIndex outside of this of the StrandSet class implies knowledge of the StrandSet implementation

Must pass this two different strands, and nominally one of the strands again which is the priority_strand. The resulting "merged" strand has the properties of the priority_strand's oligo. Decorators are preserved

the strand_low and strand_high must be presorted such that strand_low has a lower range than strand_high

low_strandset_idx should be known ahead of time as a result of selection

redo()

undo()

cadnano.strandset.removestrandcmd module

class cadnano.strandset.removestrandcmd.**RemoveStrandCommand**(*strandset*, *strand*, *solo=True*)
Bases: *cadnano.undocommand.UndoCommand*

RemoveStrandCommand deletes a strand. It should only be called on strands with no connections to other strands.

Parameters

- **strandset** (StrandSet) –
- **strand** (Strand) –
- **solo** (bool, optional) – set to True if only one strand is being removed to minimize signaling

redo()

undo()

cadnano.strandset.splitcmd module

class cadnano.strandset.splitcmd.**SplitCommand**(*strand*, *base_idx*, *update_sequence=True*)
Bases: *cadnano.undocommand.UndoCommand*

The SplitCommand takes as input a strand and "splits" the strand in two, such that one new strand 3' end is at base_idx, and the other new strand 5' end is at base_idx +/- 1 (depending on the direction of the strands).

Under the hood: On redo, this command actually is creates two new copies of the original strand, resizes each and modifies their connections. On undo, the new copies are removed and the original is restored.

redo()

undo()

cadnano.strandset.strandset module

class cadnano.strandset.strandset.**StrandSet**(*is_fwd*, *id_num*, *part*, *initial_size*)
Bases: *cadnano.cnobject.CNObject*

StrandSet is a container class for Strands, and provides the several publicly accessible methods for editing strands, including operations for creation, destruction, resizing, splitting, and merging strands.

Views may also query StrandSet for information that is useful in determining if edits can be made, such as the bounds of empty space in which a strand can be created or resized.

__iter__()
Iterate over each strand in the strands list.

addToStrandList(*strand*, *update_segments=True*)
Inserts strand into the strand_array at idx.

complementStrandSet()
Returns the complementary strandset. Used for insertions and sequence application.

createDeserializedStrand(*base_idx_low*, *base_idx_high*, *color*, *use_undostack=False*)
Passes a strand to AddStrandCommand that was read in from file input. Omits the step of checking _couldStrandInsertAtLastIndex, since we assume that deserialized strands will not cause collisions.

createStrand(*base_idx_low*, *base_idx_high*, *color=None*, *use_undostack=True*)
Assumes a strand is being created at a valid set of indices.

deepCopy(*virtual_helix*)
docstring for deepCopy

destroy()

document()

dump(*xover_list*)
Serialize a StrandSet, and append to a xover_list of xovers adding a xover if the 3 prime end of it is founds TODO update this to support strand properties :param xover_list: A list to append xovers to :type xover_list: List

Returns indices low and high of each strand in the Strandset

Return type List[Tuple]

getBoundsOfEmptyRegionContaining(*base_idx*)
Return the bounds of the empty region containing base index <base_idx>.

getLegacyArray()
docstring for getLegacyArray

getNeighbors(*strand*)

getOverlappingStrands(*idx_low*, *idx_high*)

getStrand(*base_idx*)
Returns the strand that overlaps with base_idx.

getStrandIndex(*strand*)

hasNoStrandAtOrNoXover(*idx*)

hasStrandAt(*idx_low*, *idx_high*)
Return True if strandset has a strand in the region between idx_low and idx_high (both included).

hasStrandAtAndNoXover(*idx*)

idNum()

indexOfRightmostNonemptyBase()
Returns the high base_idx of the last strand, or 0.

isDrawn5to3()

isForward()

isReverse()

isStrandInSet(*strand*)

length()

mergeStrands(*priority_strand*, *other_strand*, *use_undostack=True*)
Merge the priority_strand and other_strand into a single new strand. The oligo of priority should be propagated to the other and all of its connections.

oligoStrandRemover(*strand*, *cmds*, *solo=True*)

part()

partMaxBaseIdx()
Return the bounds of the StrandSet as defined in the part.

remove(*use_undostack=True*)
Removes a VirtualHelix from the model. Accepts a reference to the VirtualHelix, or a (row,col) lattice coordinate to perform a lookup.

removeAllStrands(*use_undostack=True*)

removeFromStrandList(*strand*, *update_segments=True*)
Remove strand from strand_array.

removeStrand(*strand*, *use_undostack=True*, *solo=True*)
solo is an argument to enable limiting signals emiting from the command in the case the command is instantiated part of a larger command

reset(*initial_size*)

resize(*delta_low*, *delta_high*)

simpleCopy(*part*)
Create an empty copy (no strands) of this strandset with the only a new virtual_helix_group parent

TODO: consider renaming this method

splitStrand(*strand*, *base_idx*, *update_sequence=True*, *use_undostack=True*)
Break strand into two strands. Reapply sequence by default.

strandCanBeSplit(*strand*, *base_idx*)
Make sure the base index is within the strand Don't split right next to a 3Prime end Don't split on endpoint (AKA a crossover)

strandCount()

strandFilter()

strandType()

strand_array

strand_heap

strands()

strandsCanBeMerged(*strandA*, *strandB*)
returns None if the strands can't be merged, otherwise if the strands can be merge it returns the strand with the lower index

only checks that the strands are of the same StrandSet and that the end points differ by 1. DOES NOT check if the Strands overlap, that should be handled by addStrand

strandsetStrandAddedSignal = <cadnano.cnproxy.DummySignal object>

updateStrandIdxs(*strand*, *old_idxs*, *new_idxs*)
update indices in the strand array/list of an existing strand

4.15.2 Module contents

4.16 cadnano.undocommand module

class cadnano.undocommand.**UndoCommand**(*name=None*)
Bases: object

addCommand(*cmd*)

redo()

undo()

4.17 cadnano.undostack module

class cadnano.undostack.**UndoStack**(*limit=10*)
Bases: object

appendUndoStack(*undocommand*)

beginMacro(*message*)

canRedo()

canUndo()

endMacro()

push(*undocommand*)

redo()

undo()

4.18 cadnano.util module

util.py

cadnano.util.**beginSuperMacro**(*model_object*, *desc=None*)
SuperMacros can be used to nest multiple command lists.

Normally execCommandList macros all the commands in a list. In some cases, multiple command lists need to be executed separately because of dependency issues. (e.g. in part.autoStaple, strands must be completely 1. created and 2. split before 3. xover installation.)

cadnano.util.**clamp**(*x*, *min_x*, *max_x*)

cadnano.util.**comp**(*seqStr*)
Returns the complement of the sequence in seqStr.

cadnano.util.**endSuperMacro**(*model_object*)
Ends a SuperMacro. Should be called after beginSuperMacro.

cadnano.util.**execCommandList**(*model_object*, *commands*, *desc=None*, *use_undostack=True*)
This is a wrapper for performing QUndoCommands, meant to ensure uniform handling of the undoStack and macro descriptions.

When using the undoStack, commands are pushed onto self.undoStack() as part of a macro with description desc. Otherwise, command redo methods are called directly.

cadnano.util.**finalizeCommands**(*model_object*, *commands*, *desc=None*)
used to enable interaction with the model but not push commands to the undostack. In practice:

1. **Call a bunch of commands and don't push them to the undostack AKA:** cmd.redo()

2. call finalizeCommands() to push the cummulative change to the stack

this assumes that the UndoCommands provided this function are respresent a transition from the initial state to the final state UndoCommands need to implement specialUndo which could just call normal undo

cadnano.util.**findChild**(*self*)
When called when self is a QGraphicsItem, iterates through self's childItems(), placing a red rectangle (a sibling of self) around each item in sequence (press return to move between items). Since the index of each child item is displayed as it is highlighted, one can use findChild() to quickly get a reference to one of self's children. At each step, one can type a command letter before hitting return. The command will apply to the current child. Command Letter: Action: <return> Advance to next child s<return> Show current child S<return> Show current child, hide siblings h<return> Hide current child r<return> return current child

cadnano.util.**init_logging**(*args=None*, *logdir=None*)
Set up standard logging system based on parameters in args, e.g. loglevel and testing.

cadnano.util.**isLinux**()

cadnano.util.**isMac**()

cadnano.util.**isWindows**()

cadnano.util.**loadAllPlugins**()

cadnano.util.**loadPlugin**(*f*)

cadnano.util.**markwhite**(*seqStr*)

cadnano.util.**methodName**()
Returns string containing name of the calling method.

cadnano.util.**nearest**(*a*, *l*)

cadnano.util.**nowhite**(*seqStr*)
Gets rid of whitespace in a string.

cadnano.util.**overlap**(*x*, *y*, *a*, *b*)
finds the overlap of the range x to y in a to b assumes an overlap exists, i.e.

y >= a and b >= x

cadnano.util.**parse_args**(*argv=None*, *gui=None*)
Uses argparse to parse commandline args. Returns a NameSpace object. This can easily be converted to a regular dict through:

argns.__dict__

This also presents a nice command line help to the user, exposed with –help flag: python main.py –help

If gui is set to "qt", then the parser will use parse_known_args. Unlike parse_args(), parse_known_args() will not cause abort by show the help message and exit, if it finds any unrecognized command-line arguments. Alternatively, you can initialize your app via

app = QApplication(sys.argv) parse_args(app.arguments())

QApplication.arguments() returns a list of arguments with all Qt arguments stripped away. Qt command line args include:

-style=<style> -stylesheet=<stylesheet> -widgetcount -reverse -qmljsdebugger -session=<session>

cadnano.util.**rcomp**(*seqStr*)
Returns the reverse complement of the sequence in seqStr.

cadnano.util.**read_fasta**(*fp*)

cadnano.util.**starmapExec**(*f*, *tupleIter*)
takes a function f and * starmaps the list but drops the results

cadnano.util.**strToDna**(*seqStr*)
Returns str having been reduced to capital ACTG.

cadnano.util.**this_path**()

cadnano.util.**trace**(*n*)
Returns a stack trace n frames deep

cadnano.util.**unloadedPlugins**()
Returns a list of plugin paths that have yet to be loaded but are in the top level of one of the search directories specified in pluginDirs

CHAPTER 5

Changelog

5.1 Unreleased

- GUI: 3D view
- I/O: Export to PDB (experimental)

5.2 v2.5

Major changes and new features since cadnano 2:

- Installer: Distribution as a Python package
- Design/GUI: Helices no longer constrained to lattices
- Design/GUI: Added support for "abstract" sequences
- Design/GUI: Added support for parallel crossovers
- Code: Updated from Python 2 $\rightarrow$ 3
- Code: Updated from PyQt4 $\rightarrow$ PyQt5
- Code: Rewrote underlying data model
- Code: Better stability
- GUI: Improved hinting across views
- GUI/Installer: Removed Maya plugin code
- I/O: New file format
- I/O: Easier scripting via command-line mode
- I/O: Export to STL (experimental)

CHAPTER 6

Authors

6.1 Development Leads

- Nick Conway <nick.conway@wyss.harvard.edu>
- Shawn Douglas <shawn.douglas@ucsf.edu>

6.2 Patches and Suggestions

- Rasmus Schøler Sørensen <rasmusscholer@gmail.com>

CHAPTER 7

License

CHAPTER 8

How to Cite

If you use cadnano in your work, please cite its associated paper. Cadnano is free software, and we rely on grant funding to support its continued development.

CHAPTER 9

Indices and tables

- genindex
- modindex
- search

C

Symbols

A

B

C

D

E

F

G

H

I

L

M

N

O

P

R

S

T

U

V

W

X

Y

Z