

Lezione 11

Bioinformatica

Mauro Ceccanti[‡] e Alberto Paoluzzi[†]

[†] Dip. Informatica e Automazione – Università “Roma Tre”

[‡] Dip. Medicina Clinica – Università “La Sapienza”



Lezione 10: Gene's 3D structure assembly

Introduction to PyPlasm

PyPlasm: some primitives

New version: double DNA helix

DNA base pairing

3D positioning of a base pair

3D structure of a generic DNA strand



Introduzione

in questa lezione calcoleremo la trasformazione di coordinate che deve essere applicata ad un componente di una struttura biologica per posizionarlo correttamente in una struttura più complessa

- ▶ L'argomento è di carattere generale, e si può adattare a molti contesti differenti. Lo introdurremo per posizionare una coppia di basi all'interno della doppia elica del DNA



Introduzione

in questa lezione calcoleremo la trasformazione di coordinate che deve essere applicata ad un componente di una struttura biologica per posizionarlo correttamente in una struttura più complessa

- ▶ L'argomento è di carattere generale, e si può adattare a molti contesti differenti. Lo introdurremo per posizionare una coppia di basi all'interno della doppia elica del DNA
- ▶ Per essere certi della correttezza delle operazioni effettuate, posizioneremo sui “diametri” dell'elica delle stringhe di testo vettoriale che rappresentano i codici (a,t,g,c) delle basi coinvolte. La trasformazione così calcolata potrà essere applicata anche al modello molecolare relativo



Introduzione

in questa lezione calcoleremo la trasformazione di coordinate che deve essere applicata ad un componente di una struttura biologica per posizionarlo correttamente in una struttura più complessa

- ▶ L'argomento è di carattere generale, e si può adattare a molti contesti differenti. Lo introdurremo per posizionare una coppia di basi all'interno della doppia elica del DNA
- ▶ Per essere certi della correttezza delle operazioni effettuate, posizioneremo sui “diametri” dell'elica delle stringhe di testo vettoriale che rappresentano i codici (a,t,g,c) delle basi coinvolte. La trasformazione così calcolata potrà essere applicata anche al modello molecolare relativo
- ▶ Nella lezione si introduce anche il linguaggio di programmazione geometrica PLaSM, sviluppato alla Sapienza e a Roma Tre, che stiamo portando in Python, e che utilizzeremo per le operazioni geometrico-grafiche



Sommario

Lezione 10: Gene's 3D structure assembly

Introduction to PyPlasm

PyPlasm: some primitives

New version: double DNA helix

DNA base pairing

3D positioning of a base pair

3D structure of a generic DNA strand



Introduction to PyPlasm

The new Plasm version ported to Python — a component of BioPlasm

► use `from trxge import *`



Introduction to PyPlasm

The new Plasm version ported to Python — a component of BioPlasm

- ▶ use `from trsxge import *`
- ▶ `every` PyPlasm primitive is **UPPER**-case (to distinguish it)



Introduction to PyPlasm

The new Plasm version ported to Python — a component of BioPlasm

- ▶ use `from trxge import *`
- ▶ `every` PyPlasm primitive is `UPPER`-case (to distinguish it)
- ▶ `every` PyPlasm primitive has a `single`-argument—very often a `list`



Introduction to PyPlasm

The new Plasm version ported to Python — a component of BioPlasm

- ▶ use `from trxge import *`
- ▶ `every` PyPlasm primitive is `UPPER`-case (to distinguish it)
- ▶ `every` PyPlasm primitive has a `single`-argument—very often a `list`
- ▶ `most` PyPlasm primitives are `higher-level` functions.



Introduction to PyPlasm

The new Plasm version ported to Python — a component of BioPlasm

- ▶ use `from trxge import *`
- ▶ `every` PyPlasm primitive is `UPPER`-case (to distinguish it)
- ▶ `every` PyPlasm primitive has a `single`-argument—very often a `list`
- ▶ `most` PyPlasm primitives are `higher-level` functions.



Introduction to PyPlasm

The new Plasm version ported to Python — a component of BioPlasm

- ▶ use `from trsxge import *`
- ▶ `every` PyPlasm primitive is **UPPER**-case (to distinguish it)
- ▶ `every` PyPlasm primitive has a **single**-argument—very often a `list`
- ▶ `most` PyPlasm primitives are **higher-level** functions.

EXAMPLE: $F(\text{arg}_1)(\text{arg}_2)(\text{arg}_3)$ is the value in Dom_4 produced by the map

$$F : Dom_1 \rightarrow Dom_2 \rightarrow Dom_3 \rightarrow Dom_4$$

where

- ▶ F returns a **function** $Dom_1 \rightarrow (Dom_2 \rightarrow Dom_3 \rightarrow Dom_4)$



Introduction to PyPlasm

The new Plasm version ported to Python — a component of BioPlasm

- ▶ use **from** `trxge` **import** *
- ▶ **every** PyPlasm primitive is **UPPER**-case (to distinguish it)
- ▶ **every** PyPlasm primitive has a **single**-argument—very often a **list**
- ▶ **most** PyPlasm primitives are **higher-level** functions.

EXAMPLE: $F(\text{arg}_1)(\text{arg}_2)(\text{arg}_3)$ is the value in Dom_4 produced by the map

$$F : Dom_1 \rightarrow Dom_2 \rightarrow Dom_3 \rightarrow Dom_4$$

where

- ▶ F returns a **function** $Dom_1 \rightarrow (Dom_2 \rightarrow Dom_3 \rightarrow Dom_4)$
- ▶ $F(\text{arg}_1)$ returns a **function** $Dom_2 \rightarrow (Dom_3 \rightarrow Dom_4)$



Introduction to PyPlasm

The new Plasm version ported to Python — a component of BioPlasm

- ▶ use **from** `trxge` **import** *
- ▶ **every** PyPlasm primitive is **UPPER**-case (to distinguish it)
- ▶ **every** PyPlasm primitive has a **single**-argument—very often a **list**
- ▶ **most** PyPlasm primitives are **higher-level** functions.

EXAMPLE: $F(\text{arg}_1)(\text{arg}_2)(\text{arg}_3)$ is the value in Dom_4 produced by the map

$$F : Dom_1 \rightarrow Dom_2 \rightarrow Dom_3 \rightarrow Dom_4$$

where

- ▶ F returns a **function** $Dom_1 \rightarrow (Dom_2 \rightarrow Dom_3 \rightarrow Dom_4)$
- ▶ $F(\text{arg}_1)$ returns a **function** $Dom_2 \rightarrow (Dom_3 \rightarrow Dom_4)$
- ▶ $F(\text{arg}_1)(\text{arg}_2)$ returns a **function** $Dom_3 \rightarrow Dom_4$



Introduction to PyPlasm

The new Plasm version ported to Python — a component of BioPlasm

- ▶ use **from** `trxge` **import** *
- ▶ **every** PyPlasm primitive is **UPPER**-case (to distinguish it)
- ▶ **every** PyPlasm primitive has a **single**-argument—very often a **list**
- ▶ **most** PyPlasm primitives are **higher-level** functions.

EXAMPLE: $F(\text{arg}_1)(\text{arg}_2)(\text{arg}_3)$ is the value in Dom_4 produced by the map

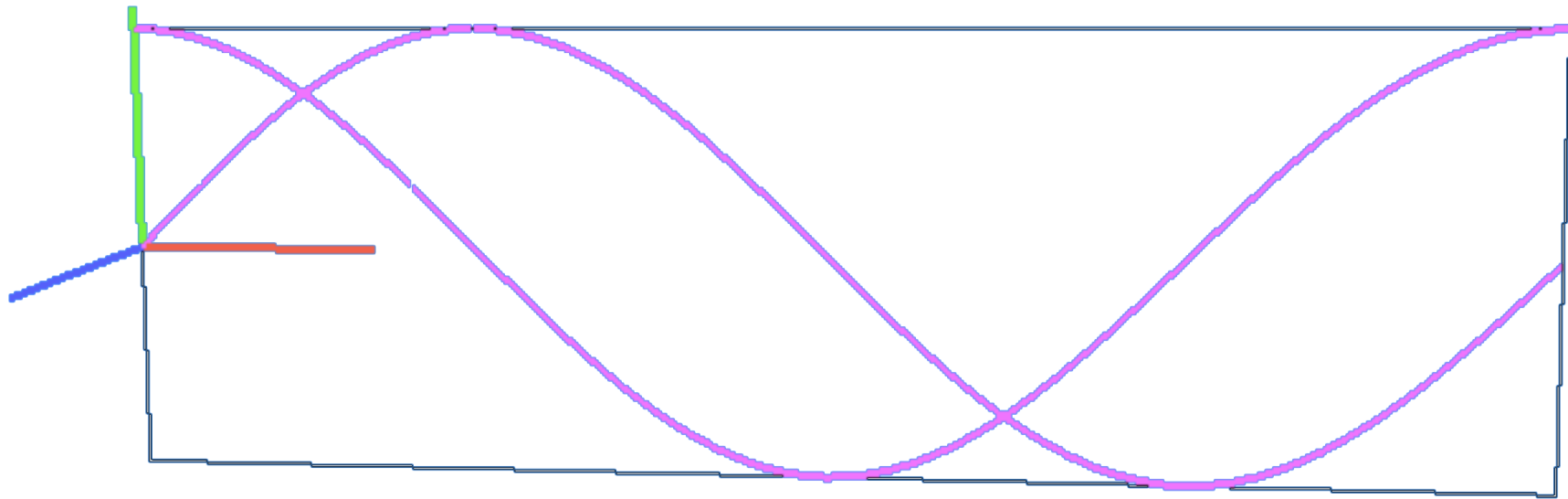
$$F : Dom_1 \rightarrow Dom_2 \rightarrow Dom_3 \rightarrow Dom_4$$

where

- ▶ F returns a **function** $Dom_1 \rightarrow (Dom_2 \rightarrow Dom_3 \rightarrow Dom_4)$
- ▶ $F(\text{arg}_1)$ returns a **function** $Dom_2 \rightarrow (Dom_3 \rightarrow Dom_4)$
- ▶ $F(\text{arg}_1)(\text{arg}_2)$ returns a **function** $Dom_3 \rightarrow Dom_4$
- ▶ $F(\text{arg}_1)(\text{arg}_2)(\text{arg}_3)$ returns an **object** in Dom_4



Introduction to PyPlasm

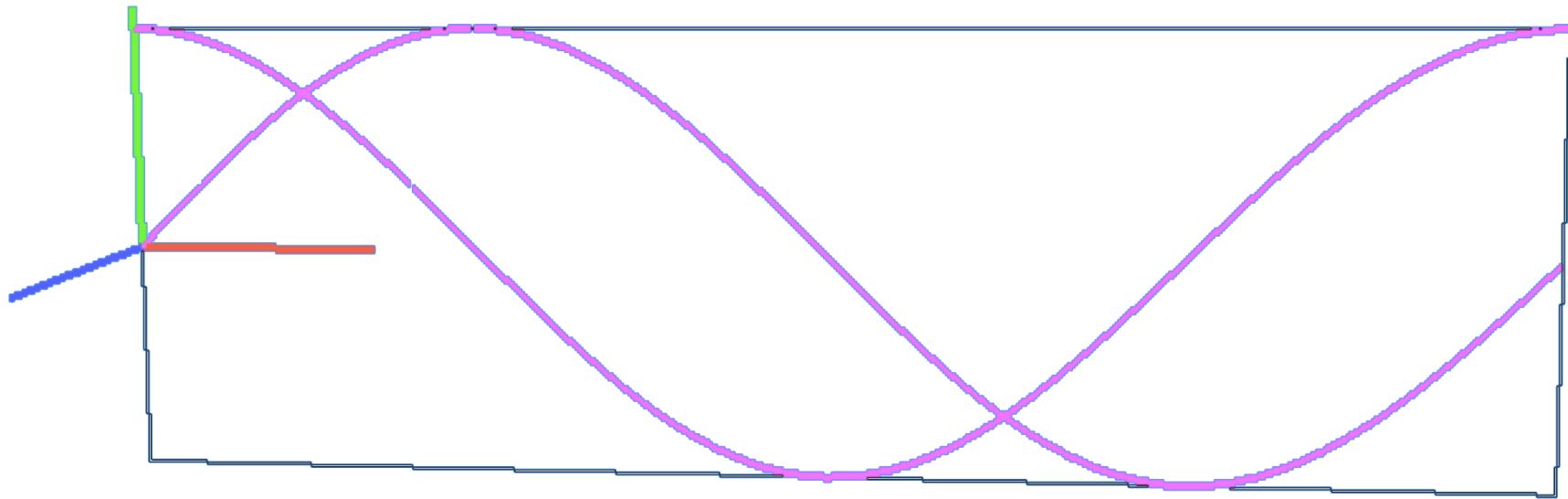


- $f(x)$ is the **application** of function f to argument x

```
1 def drawgraph (f):  
2     domain = INTERVALS (2*PI) (64)  
3     return MAP ([ S1, COMP ([f, S1]) ]) (domain)  
4  
5 VIEW (drawgraph (SIN))  
6 VIEW (STRUCT ([ drawgraph (SIN), drawgraph (COS) ]))
```



Introduction to PyPlasm

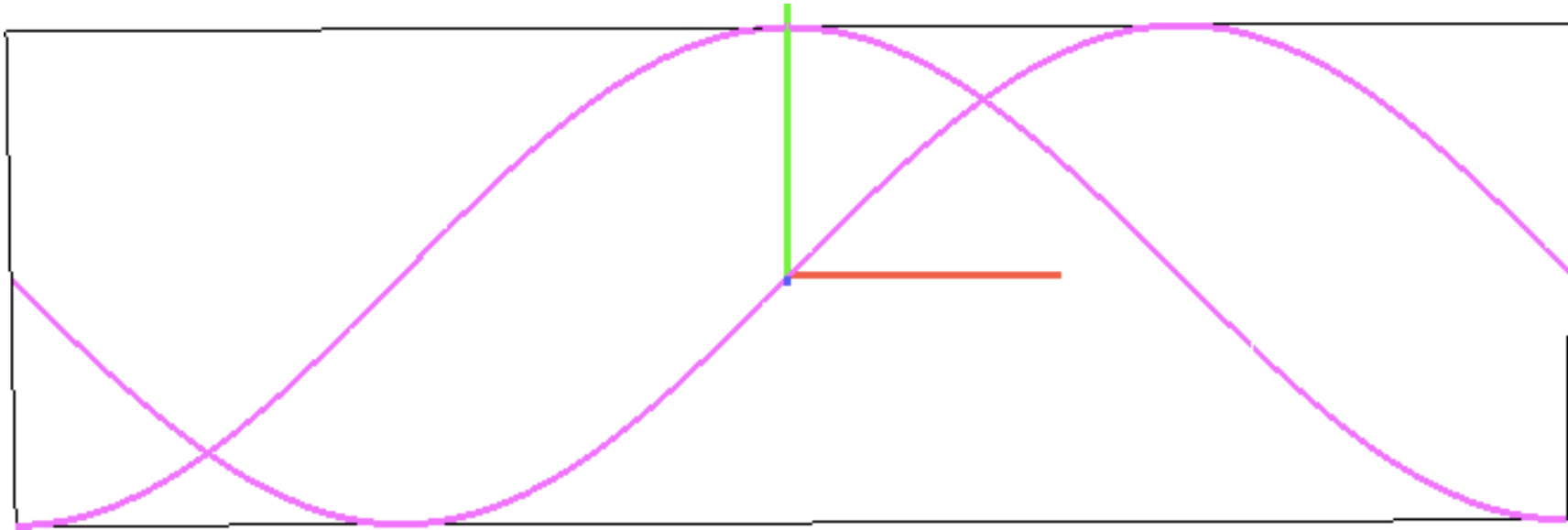


- ▶ $f(x)$ is the **application** of function f to argument x
- ▶ **STRUCT** returns a **single** geometric value from its list of arguments

```
1 def drawgraph (f):  
2     domain = INTERVALS (2*PI) (64)  
3     return MAP ([ S1, COMP ([f, S1]) ]) (domain)  
4  
5 VIEW (drawgraph (SIN) )  
6 VIEW (STRUCT ([ drawgraph (SIN), drawgraph (COS) ]))
```



Introduction to PyPlasm



- domain is translated of $-\pi$ on the first coordinate

```
1 def drawgraph (f):  
2     domain = T(1)(-PI)(INTERVALS(2*PI)(64))  
3     return MAP([ S1, COMP([f, S1]) ])(domain)  
4  
5 VIEW(STRUCT([ drawgraph(SIN), drawgraph(COS) ]))
```



Sommario

Lezione 10: Gene's 3D structure assembly

Introduction to PyPlasm

PyPlasm: some primitives

New version: double DNA helix

DNA base pairing

3D positioning of a base pair

3D structure of a generic DNA strand



PyPlasm: (some) geometric primitives

CUBOID n -dim rectangle: `VIEW(CUBOID([1,4,9]))`



PyPlasm: (some) geometric primitives

CUBOID n -dim rectangle: `VIEW(CUBOID([1,4,9]))`

SIMPLEX n -dim simplex: `VIEW(SIMPLEX(3))`



PyPlasm: (some) geometric primitives

CUBOID n -dim rectangle: `VIEW(CUBOID([1,4,9]))`

SIMPLEX n -dim simplex: `VIEW(SIMPLEX(3))`

QUOTE linspace generator: `D = QUOTE(10*[0.1, -0.1])`



PyPlasm: (some) geometric primitives

CUBOID *n*-dim rectangle: `VIEW(CUBOID([1,4,9]))`

SIMPLEX *n*-dim simplex: `VIEW(SIMPLEX(3))`

QUOTE linspace generator: `D = QUOTE(10*[0.1, -0.1])`

PROD Cartesian product: `VIEW(PROD([D,D,D]))`



PyPlasm: (some) geometric primitives

CUBOID n -dim rectangle: `VIEW(CUBOID([1,4,9]))`

SIMPLEX n -dim simplex: `VIEW(SIMPLEX(3))`

QUOTE linspace generator: `D = QUOTE(10*[0.1, -0.1])`

PROD Cartesian product: `VIEW(PROD([D,D,D]))`

MAP coordinate transformation: `MAP([f_1, \dots, f_d])(dom)`



PyPlasm: (some) geometric primitives

CUBOID n -dim rectangle: `VIEW(CUBOID([1,4,9]))`

SIMPLEX n -dim simplex: `VIEW(SIMPLEX(3))`

QUOTE linspace generator: `D = QUOTE(10*[0.1, -0.1])`

PROD Cartesian product: `VIEW(PROD([D,D,D]))`

MAP coordinate transformation: `MAP([f_1, \dots, f_d])(dom)`

POLYLINE polygonal lines: `POLYLINE([p_1, \dots, p_m])`



PyPlasm: (some) geometric primitives

CUBOID n -dim rectangle: `VIEW(CUBOID([1,4,9]))`

SIMPLEX n -dim simplex: `VIEW(SIMPLEX(3))`

QUOTE linspace generator: `D = QUOTE(10*[0.1, -0.1])`

PROD Cartesian product: `VIEW(PROD([D,D,D]))`

MAP coordinate transformation: `MAP([f_1, \dots, f_d])(dom)`

POLYLINE polygonal lines: `POLYLINE([p_1, \dots, p_m])`

TEXT Testi vettoriali 3D: `VIEW(TEXT('PLaSM'))`



PyPlasm: (some) geometric primitives

CUBOID n -dim rectangle: `VIEW(CUBOID([1,4,9]))`

SIMPLEX n -dim simplex: `VIEW(SIMPLEX(3))`

QUOTE linspace generator: `D = QUOTE(10*[0.1, -0.1])`

PROD Cartesian product: `VIEW(PROD([D,D,D]))`

MAP coordinate transformation: `MAP([f_1, \dots, f_d])(dom)`

POLYLINE polygonal lines: `POLYLINE([p_1, \dots, p_m])`

TEXT Testi vettoriali 3D: `VIEW(TEXT('PLaSM'))`

STRUCT Strutture grafiche: `STRUCT([obj_1, \dots, obj_m])`



PyPlasm: (some) geometric primitives

CUBOID n -dim rectangle: `VIEW(CUBOID([1,4,9]))`

SIMPLEX n -dim simplex: `VIEW(SIMPLEX(3))`

QUOTE linspace generator: `D = QUOTE(10*[0.1, -0.1])`

PROD Cartesian product: `VIEW(PROD([D,D,D]))`

MAP coordinate transformation: `MAP([f_1, \dots, f_d])(dom)`

POLYLINE polygonal lines: `POLYLINE([p_1, \dots, p_m])`

TEXT Testi vettoriali 3D: `VIEW(TEXT('PLaSM'))`

STRUCT Strutture grafiche: `STRUCT([obj_1, \dots, obj_m])`

T,R,S,MAT affine transformations: `T([i_1, \dots, i_n])([t_{i_1}, \dots, t_{i_n}])(obj)`



PyPlasm: (some) geometric primitives

CUBOID n -dim rectangle: `VIEW(CUBOID([1,4,9]))`

SIMPLEX n -dim simplex: `VIEW(SIMPLEX(3))`

QUOTE linspace generator: `D = QUOTE(10*[0.1, -0.1])`

PROD Cartesian product: `VIEW(PROD([D,D,D]))`

MAP coordinate transformation: `MAP([f_1, \dots, f_d])(dom)`

POLYLINE polygonal lines: `POLYLINE([p_1, \dots, p_m])`

TEXT Testi vettoriali 3D: `VIEW(TEXT('PLaSM'))`

STRUCT Strutture grafiche: `STRUCT([obj_1, \dots, obj_m])`

T,R,S,MAT affine transformations: `T([i_1, \dots, i_n])([t_{i_1}, \dots, t_{i_n}])(obj)`

SOLIDIFY Boundary \rightarrow solid transform



PyPlasm: (some) geometric primitives

CUBOID n -dim rectangle: `VIEW(CUBOID([1,4,9]))`

SIMPLEX n -dim simplex: `VIEW(SIMPLEX(3))`

QUOTE linspace generator: `D = QUOTE(10*[0.1, -0.1])`

PROD Cartesian product: `VIEW(PROD([D,D,D]))`

MAP coordinate transformation: `MAP([f_1, \dots, f_d])(dom)`

POLYLINE polygonal lines: `POLYLINE([p_1, \dots, p_m])`

TEXT Testi vettoriali 3D: `VIEW(TEXT('PLaSM'))`

STRUCT Strutture grafiche: `STRUCT([obj_1, \dots, obj_m])`

T,R,S,MAT affine transformations: `T([i_1, \dots, i_n])([t_1, \dots, t_n])(obj)`

SOLIDIFY Boundary \rightarrow solid transform

Boolean Ops `INTERSECTION, DIFFERENCE, XOR, UNION`



PyPlasm: (some) geometric primitives

CUBOID n -dim rectangle: `VIEW(CUBOID([1,4,9]))`

SIMPLEX n -dim simplex: `VIEW(SIMPLEX(3))`

QUOTE linspace generator: `D = QUOTE(10*[0.1, -0.1])`

PROD Cartesian product: `VIEW(PROD([D,D,D]))`

MAP coordinate transformation: `MAP([f_1, \dots, f_d])(dom)`

POLYLINE polygonal lines: `POLYLINE([p_1, \dots, p_m])`

TEXT Testi vettoriali 3D: `VIEW(TEXT('PLaSM'))`

STRUCT Strutture grafiche: `STRUCT([obj_1, \dots, obj_m])`

T,R,S,MAT affine transformations: `T([i_1, \dots, i_n])([t_{i_1}, \dots, t_{i_n}])(obj)`

SOLIDIFY Boundary \rightarrow solid transform

Boolean Ops `INTERSECTION, DIFFERENCE, XOR, UNION`

BOX Containment box: `BOX(STRUCT([...]))`



PyPlasm: (some) geometric primitives

CUBOID n -dim rectangle: `VIEW(CUBOID([1,4,9]))`

SIMPLEX n -dim simplex: `VIEW(SIMPLEX(3))`

QUOTE linspace generator: `D = QUOTE(10*[0.1, -0.1])`

PROD Cartesian product: `VIEW(PROD([D,D,D]))`

MAP coordinate transformation: `MAP([f_1, \dots, f_d])(dom)`

POLYLINE polygonal lines: `POLYLINE([p_1, \dots, p_m])`

TEXT Testi vettoriali 3D: `VIEW(TEXT('PLaSM'))`

STRUCT Strutture grafiche: `STRUCT([obj_1, \dots, obj_m])`

T,R,S,MAT affine transformations: `T([i_1, \dots, i_n])([t_{i_1}, \dots, t_{i_n}])(obj)`

SOLIDIFY Boundary \rightarrow solid transform

Boolean Ops `INTERSECTION, DIFFERENCE, XOR, UNION`

BOX Containment box: `BOX(STRUCT([...]))`

JOIN Convex-comb of point-sets `VIEW(JOIN([SIMPLEX(3),Q(10)]))`



PyPlasm: (some) geometric primitives

CUBOID n -dim rectangle: `VIEW(CUBOID([1,4,9]))`

SIMPLEX n -dim simplex: `VIEW(SIMPLEX(3))`

QUOTE linspace generator: `D = QUOTE(10*[0.1, -0.1])`

PROD Cartesian product: `VIEW(PROD([D,D,D]))`

MAP coordinate transformation: `MAP([f_1, \dots, f_d])(dom)`

POLYLINE polygonal lines: `POLYLINE([p_1, \dots, p_m])`

TEXT Testi vettoriali 3D: `VIEW(TEXT('PLaSM'))`

STRUCT Strutture grafiche: `STRUCT([obj_1, \dots, obj_m])`

T,R,S,MAT affine transformations: `T([i_1, \dots, i_n])([t_1, \dots, t_n])(obj)`

SOLIDIFY Boundary \rightarrow solid transform

Boolean Ops `INTERSECTION, DIFFERENCE, XOR, UNION`

BOX Containment box: `BOX(STRUCT([...]))`

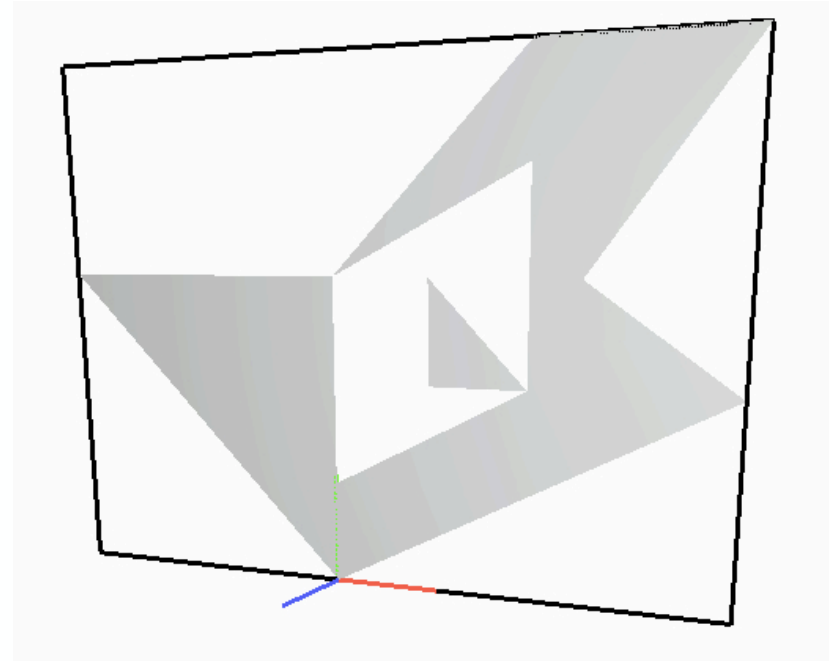
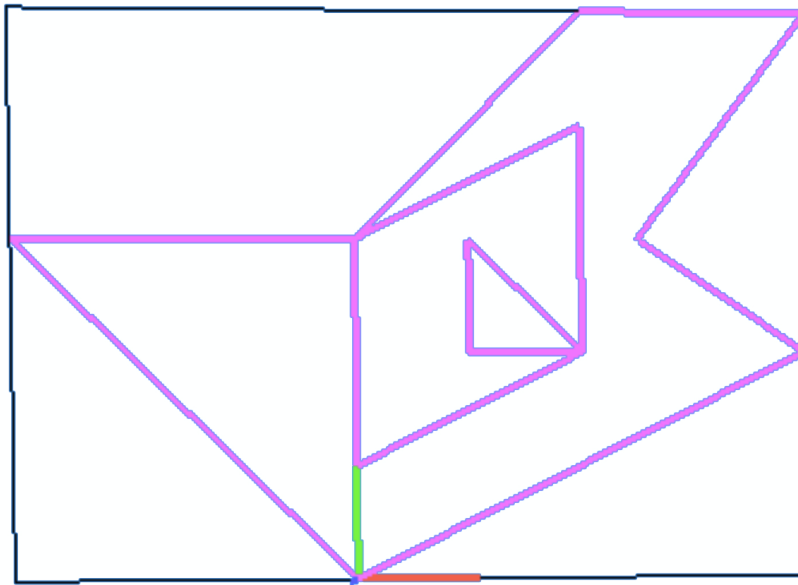
JOIN Convex-comb of point-sets `VIEW(JOIN([SIMPLEX(3),Q(10)]))`

Example: `VIEW(XOR([CIRCLE(1)([4,1]), CIRCLE(1)([3,1])]))`



PyPlasm: some primitives

Some **closed** polylines (where $p_1 == p_m$), and the **solid** polygon they are boundary of

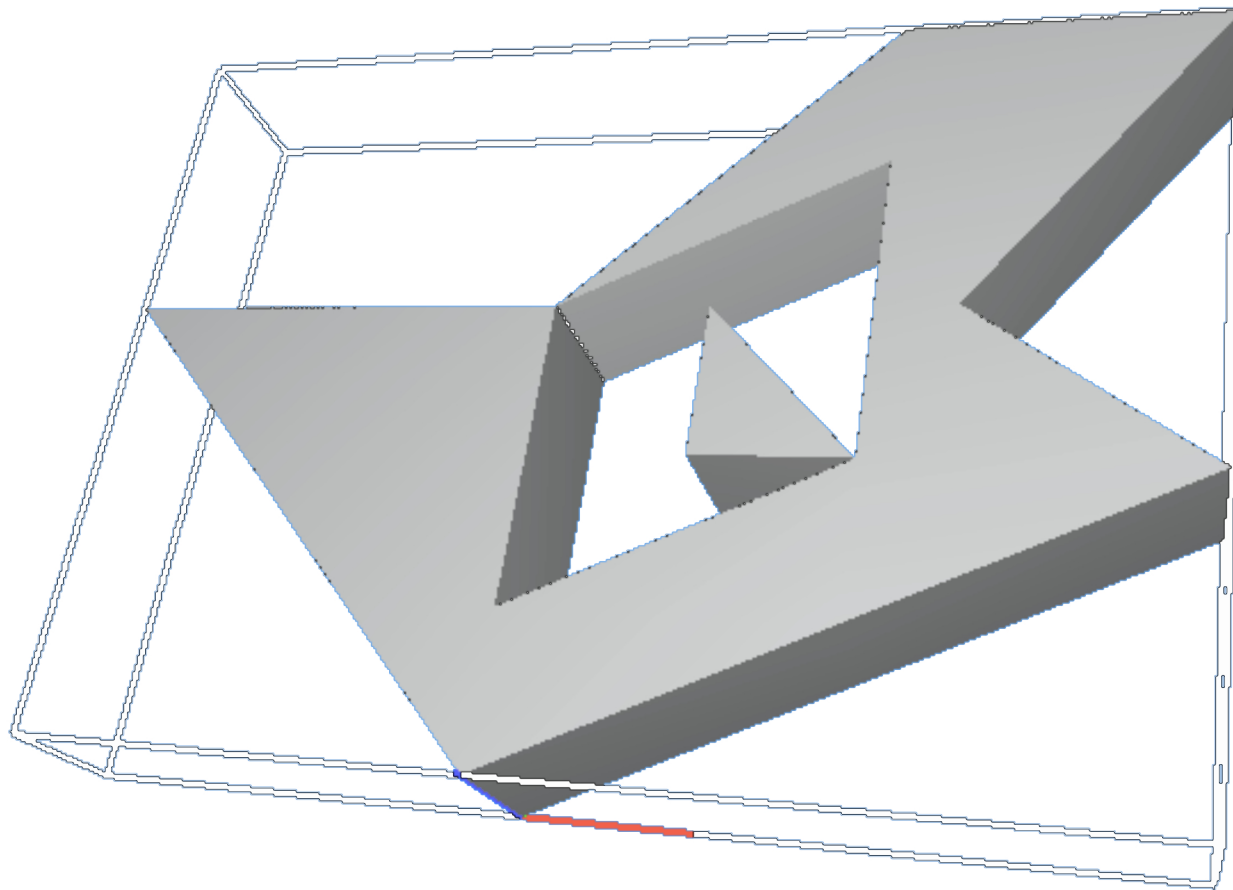


```
1 out = STRUCT( AA(POLYLINE) ([  
    [[0,0],[4,2],[2.5,3],[4,5],[2,5],[0,3],[-3,3],[0,0]],  
    [[0,3],[0,1],[2,2],[2,4],[0,3]],  
    [[2,2],[1,3],[1,2],[2,2]] ]) )  
2  
3 VIEW(out)  
4 VIEW(SOLIDIFY(out))
```



PyPlasm: some primitives

The Cartesian product of a 2D shape for a 1D interval—in this case $Q(1)$ —produces a 3D shape



```
1 VIEW (PROD ([ SOLIDIFY(out), Q(1) ]))
```



Sommario

Lezione 10: Gene's 3D structure assembly

Introduction to PyPlasm

PyPlasm: some primitives

New version: double DNA helix

DNA base pairing

3D positioning of a base pair

3D structure of a generic DNA strand



New version: double DNA helix

- ▶ Riprendiamo il modello della doppia elica parametrizzata rispetto a raggio, passo e numero di giri, e lo modifichiamo leggermente per adattarlo all'uso delle primitive Plasm



New version: double DNA helix

- ▶ Riprendiamo il modello della doppia elica parametrizzata rispetto a raggio, passo e numero di giri, e lo modifichiamo leggermente per adattarlo all'uso delle primitive Plasm
- ▶ l'attuale implementazione plasm tratta l'insieme dei vertici di un oggetto geometrico come una lista di liste (di coordinate), e non come un array a due indici di coordinate, come fatto in precedenza



New version: double DNA helix

- ▶ Riprendiamo il modello della doppia elica parametrizzata rispetto a raggio, passo e numero di giri, e lo modifichiamo leggermente per adattarlo all'uso delle primitive Plasm
- ▶ l'attuale implementazione plasm tratta l'insieme dei vertici di un oggetto geometrico come una lista di liste (di coordinate), e non come un array a due indici di coordinate, come fatto in precedenza
- ▶ eviteremo dunque di effettuare continue trasformazioni di tipo, riferendoci ovunque possibile alla lista di liste di numeri (le coordinate dei vertici)



New version: double DNA helix

The code is adapted to fit the high-level geometric and graphics primitives of PyPlasm

```
1  from numpy import *
2  from trsxge import *
3
4  def helixPoints(radius, pitch, nturns):
5      c = linspace(0, 2*pi*nturns, 12*nturns)
6      return map(list, zip( cos(c), sin(c), c*(pitch/(2*pi))
7                          ) ))
8
9  def helix(radius, pitch, nturns):
10     return POLYLINE(helixPoints(radius, pitch, nturns))
11
12
13 VIEW(helix(1, 1.5, 6))
```

Note: (a) `helixPoints` returns a list of `list` ; (b) `POLYLINE` is UPPER-case



New version: double DNA helix

The code is adapted to fit the high-level geometric and graphics primitives of PyPlasm

`doubleHelix` returns a **STRUCT** of 2 `p` instances

```
1  def doubleHelix(radius, pitch, nturns):
2      p = POLYLINE(helixPoints(radius, pitch, nturns))
3      return STRUCT([p, R([1, 2])(PI)(p)])
4
5
6  VIEW(doubleHelix(1, 2, 4))
```

- The second `p` instance is rotated of π around the z-axis



New version: double DNA helix

The code is adapted to fit the high-level geometric and graphics primitives of PyPlasm

`doubleHelix` returns a **STRUCT** of 2 `p` instances

```
1 def doubleHelix(radius, pitch, nturns):  
2     p = POLYLINE(helixPoints(radius, pitch, nturns))  
3     return STRUCT([p, R([1, 2])(PI)(p)])  
4  
5  
6 VIEW(doubleHelix(1, 2, 4))
```

- ▶ The second `p` instance is rotated of π around the z-axis
- ▶ The rotation tensor `R([1,2])(PI)` is a higher-level function (is applied more than one time to different arguments; every application returns a function)



New version: double DNA helix

The code is adapted to fit the high-level geometric and graphics primitives of PyPlasm

```
1 def dnaStructure(radius, pitch, nturns):
2     p = helixPoints(radius, pitch, nturns)
3     q = (matrix(p) * matrix([[[-1, 0, 0], [0, -1, 0],
4                               [0, 0, 1]]])).tolist()
5     diameters = TRANS([p, q])
6     return STRUCT(AA(POLYLINE)(diameters) + [POLYLINE(p
7     ), POLYLINE(q)])
8
9 VIEW(dnaStructure(1, 2, 4))
```

AA (second-order function) stands for **Apply—to—All**:

$$AA(f)([x_1, x_2, \dots, x_m]) \equiv [f(x_1), f(x_2), \dots, f(x_m)]$$

It has the same semantics of the Python's `map(f, [x1, x2, ..., xm])`



Sommario

Lezione 10: Gene's 3D structure assembly

Introduction to PyPlasm

PyPlasm: some primitives

New version: double DNA helix

DNA base pairing

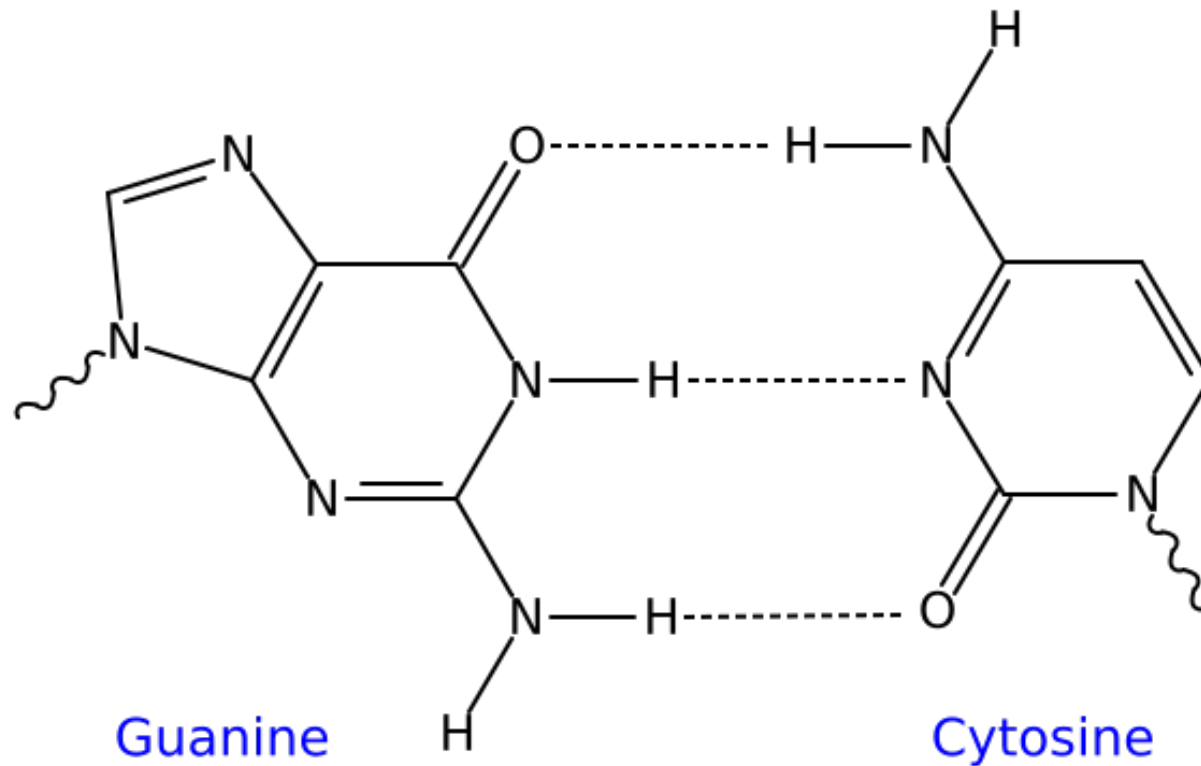
3D positioning of a base pair

3D structure of a generic DNA strand



DNA base pairing

We know that two nucleotides on opposite complementary DNA or RNA strands that are connected via hydrogen bonds are called a **base pair**



So, let define a set of allowed pairs

```
basepairs = [ ['a', 't'], ['t', 'a'], ['g', 'c'], ['c', 'g'] ]
```



DNA base pairing

We write a simple filter to produce the sequence of base pairs associated to a DNA strand

```
1 strand = 'atgaaaatgaataaaaagtctcatcggtcc'
2
3 def doublestrand (strand):
4     bases = ['t','a','g','c']
5     basepairs = [['a','t'], ['t','a'], ['g','c'], ['c','g']]
6     return [[a,b] for a in strand for b in bases if [a,b]
7         in basepairs]
8
9 doublestrand(strand) ≡
10 [['a','t'], ['t','a'], ['g','c'], ['a','t'], ['a','t'], ['a','t'],
11     ['t'], ['a','t'], ['t','a'], ['g','c'], ['a','t'], ['a','t'],
12     ], ['t','a'], ['a','t'], ['a','t'], ['a','t'], ['a','t'], [
13     'g','c'], ['t','a'], ['c','g'], ['t','a'], ['c','g'], ['a',
14     't'], ['t','a'], ['c','g'], ['g','c'], ['t','a'], ['c','g',
15     ''], ['c','g']]
```

REMARK: We use a [List Comprehension](#) construct, with a doubly-nested cycle to select the only allowed base pairs



Sommario

Lezione 10: Gene's 3D structure assembly

Introduction to PyPlasm

PyPlasm: some primitives

New version: double DNA helix

DNA base pairing

3D positioning of a base pair

3D structure of a generic DNA strand



3D positioning of a base pair

A vector text string is generated in the $z = 0$ plane

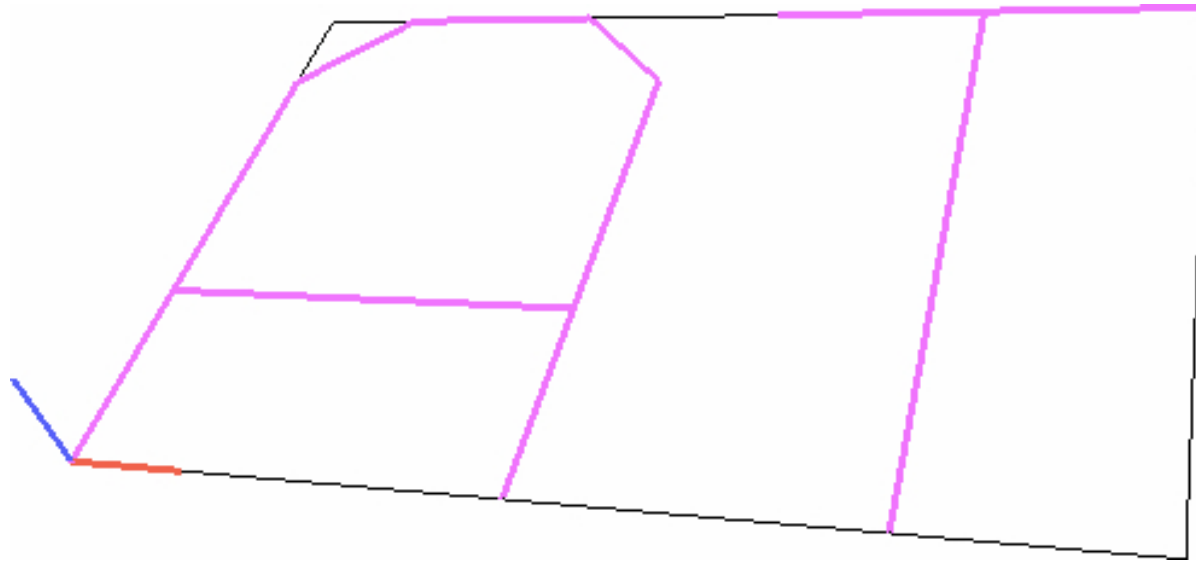
```
1 def TEXT: ... return hpc
2 VIEW(TEXT('AT'))
3
4 def TEXTWITHATTRIBUTES (TEXTALIGNMENT, TEXTANGLE,
5     TEXTWIDTH, TEXTHEIGHT, TEXTSPACING):
6     ... return hpc
7 VIEW(TEXTWITHATTRIBUTES('centre', 0, 0.5, 1.0, 0.125)('AT'))
8 VIEW(R([2,3])(PI/2)(TEXTWITHATTRIBUTES('centre', 0, 0.5,
9     1.0, 0.125)('AT'))))
```

In (8) the resulting geometric object is rotated in the $y = 0$ plane via the rotation tensor $R([2,3])(PI/2)$



3D positioning of a base pair

A vector-text string is generated in the $z = 0$ plane. The native dimensions of characters of the vector font are pretty big

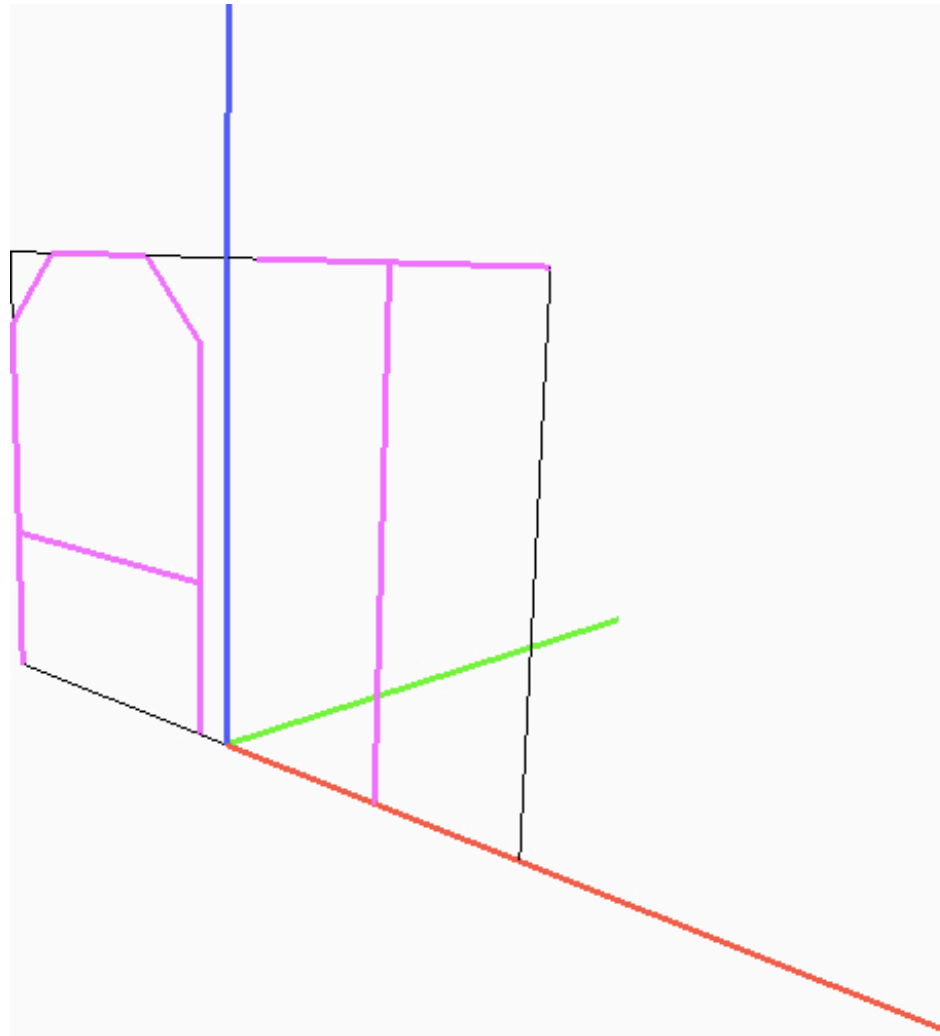


`VIEW(TEXT('AT'))`



3D positioning of a base pair

A vector-text string is generated in the $z = 0$ plane and rotated in the $y = 0$ plane



```
R ([2,3]) (PI/2) (TEXTWITHATTRIBUTES('centre',0.0,0.5,1.0,0.125)('AT'))
```



3D positioning of a base pair

The function **return** the text string transformed and translated in its proper 3D space position

In particular, we compute the affine transformation from the standard position seen above, to the proper space position of each base pair

- input data: $p_1, q_1 \in strand_1, p_2 \in strand_2$



3D positioning of a base pair

The function **return** the text string transformed and translated in its proper 3D space position

In particular, we compute the affine transformation from the standard position seen above, to the proper space position of each base pair

- ▶ input data: $p_1, q_1 \in strand_1, p_2 \in strand_2$
- ▶ Steps: linear transformation followed translation



3D positioning of a base pair

The function **return** the text string transformed and translated in its proper 3D space position

In particular, we compute the affine transformation from the standard position seen above, to the proper space position of each base pair

- ▶ input data: $p_1, q_1 \in strand_1, p_2 \in strand_2$
- ▶ Steps: linear transformation followed translation

TRANSF to bring \mathbf{e}_1 to \mathbf{r}_1 , \mathbf{e}_2 to \mathbf{r}_2 , and \mathbf{e}_3 to \mathbf{r}_3 ,
where $\mathbf{e}_1, \mathbf{e}_2, \mathbf{e}_3$ are the standard basis of \mathbb{E}^3 , and

$$\mathbf{r}_1 = \mathbf{p}_2 - \mathbf{p}_1, \quad \mathbf{r}_2 = \mathbf{q}_1 - \mathbf{p}_1, \quad \mathbf{r}_3 = \mathbf{r}_1 \times \mathbf{r}_2$$



3D positioning of a base pair

The function **return** the text string transformed and translated in its proper 3D space position

In particular, we compute the affine transformation from the standard position seen above, to the proper space position of each base pair

- ▶ input data: $p_1, q_1 \in strand_1, p_2 \in strand_2$
- ▶ Steps: linear transformation followed translation

TRANSF to bring \mathbf{e}_1 to \mathbf{r}_1 , \mathbf{e}_2 to \mathbf{r}_2 , and \mathbf{e}_3 to \mathbf{r}_3 ,
where $\mathbf{e}_1, \mathbf{e}_2, \mathbf{e}_3$ are the standard basis of \mathbb{E}^3 , and

$$\mathbf{r}_1 = \mathbf{p}_2 - \mathbf{p}_1, \quad \mathbf{r}_2 = \mathbf{q}_1 - \mathbf{p}_1, \quad \mathbf{r}_3 = \mathbf{r}_1 \times \mathbf{r}_2$$

TRANSL to bring the origin to point $\mathbf{p}_m = (\mathbf{p}_1 + \mathbf{p}_2)/2$ of diameter



3D positioning of a base pair

The function **return** the text string transformed and translated in its proper 3D space position

clearly we must have

$$\mathbf{Q} \begin{bmatrix} \mathbf{e}_1 & \mathbf{e}_2 & \mathbf{e}_3 \end{bmatrix} = \begin{bmatrix} \mathbf{r}_1 & \mathbf{r}_2 & \mathbf{r}_3 \end{bmatrix}$$

so that, in homogeneous coordinates, we have:

$$\mathbf{Q} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ \mathbf{0} & \mathbf{r}_1 & \mathbf{r}_2 & \mathbf{r}_3 \end{bmatrix}.$$

If we include the translation, the desired transformation becomes:

$$\bar{\mathbf{Q}} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ \mathbf{p}_m & \mathbf{r}_1 & \mathbf{r}_2 & \mathbf{r}_3 \end{bmatrix},$$

where $\mathbf{p}_m = (\mathbf{p}_1 + \mathbf{p}_2)/2$.



3D positioning of a base pair

The function **return** the text string scaled, rotated and translated in its proper 3D space position

```
1  def basepair (args):
2      """ p1,p2,q1 are 3D points, pair a two-letter string
3      returns a geometric value """
4      p1,p2,q1,pair = args
5      pm = SCALARVECTPROD([ SUM([p1,p2]), 0.5 ])
6      t = T([1,2,3])(pm)
7      q2 = UNITVECT(DIFF([p1,pm]))
8      q3 = [q2[1], -q2[0], 0]
9      w = [0,0,1]
10     sizeX, sizeZ = VECTNORM(q2)/4, VECTNORM(q3)/2,
11     obj = TEXTWITHATTRIBUTES('centre',0.0,sizeX,
12         sizeZ,sizeX/16)(''.join(['__',pair[0],'__',
13         pair[1],'__']))
13     return (t(MAT(INV(MATHOM([q2,w,q3])))(obj)))
```



Sommario

Lezione 10: Gene's 3D structure assembly

Introduction to PyPlasm

PyPlasm: some primitives

New version: double DNA helix

DNA base pairing

3D positioning of a base pair

3D structure of a generic DNA strand



3D structure of a generic DNA strand

Here we give yet another version of the `dnaStruct` function, where the number of turns of each helix is determined by the length of the DNA strand, and where each basepair symbol is properly located in 3D space

In the following `dnaStruct(radius,pitch)(strand)` implementation, we introduce some variations w.r.t. the previous version:

- ▶ `n = len(strand)` is the number of basepairs in the input strand



3D structure of a generic DNA strand

Here we give yet another version of the `dnaStruct` function, where the number of turns of each helix is determined by the length of the DNA strand, and where each basepair symbol is properly located in 3D space

In the following `dnaStruct(radius,pitch)(strand)` implementation, we introduce some variations w.r.t. the previous version:

- ▶ `n = len(strand)` is the number of basepairs in the input strand
- ▶ `basepairs` is the list of pairs of strings to be transformed into vector text



3D structure of a generic DNA strand

Here we give yet another version of the `dnaStruct` function, where the number of turns of each helix is determined by the length of the DNA strand, and where each basepair symbol is properly located in 3D space

In the following `dnaStruct(radius, pitch)(strand)` implementation, we introduce some variations w.r.t. the previous version:

- ▶ `n = len(strand)` is the number of basepairs in the input strand
- ▶ `basepairs` is the list of pairs of strings to be transformed into vector text
- ▶ `map(basepair, zip(p[:-1], q[:-1], p[1:], basepairs))` apply the proper *string* \rightarrow *text* transformation to each tuple of properly aligned points and symbols from the two helices `p` and `q` and from the `basepairs` list of symbols



3D structure of a generic DNA strand

We give here a function of second order, that must be first applied to the 2 intrinsic scaling factors of the structure (radius, pitch), and then to the DNA strand to be modeled, given as a string. From the programming viewpoint, notice how a higher-level function is structured: it returns either **partial functions** or the **final value**

```
1  def dnaStruct (radius, pitch) :  
2  
3      def dna (strand) :  
4          n = len(strand)  
5          p = helixPoints (radius, pitch, n/12)  
6          q = (matrix(p) * matrix([[-1, 0, 0], [0, -1, 0],  
7              [0, 0, 1]])) .tolist()  
8          basepairs = doublestrand(strand)  
9          basepairs3D = map(basepair, zip(p[:-1], q[:-1], p  
10              [1:], basepairs))  
11          return STRUCT(basepairs3D + [POLYLINE(p),  
12              POLYLINE(q)])  
13  
14      return dna  
  
VIEW(dnaStruct(1, 4)(strand))
```



3D structure of a generic DNA strand

```
1 strand = 'acagtaacacactctgttaacctt'
2
3 doublestrand(strand) =
4 [['a', 't'], ['c', 'g'], ['a', 't'],
   ['g', 'c'], ['t', 'a'], ['a', 't'],
   ['a', 't'], ['c', 'g'], ['a', 't'],
   ['c', 'g'], ['t', 'a'], ['c', 'g'],
   ['t', 'a'], ['g', 'c'], ['t', 'a'],
   ['t', 'a'], ['a', 't'], ['a', 't'],
   ['a', 't'], ['c', 'g'], ['c', 'g'],
   ['t', 'a'], ['t', 'a']]
5
6 VIEW(dnaStruct(1, 4)(strand))
```

