

Geometric and Graphics Programming Laboratory: Lecture 7b

Alberto Paoluzzi

October 27, 2017

Outline: Algebra3

1 Introduction to pyplasm

2 Matrices

Introduction to pyplasm

PLaSM Basics

PLaSM = Geometric extension of the FL language by Backus (developed at IBM Research)

A. Paoluzzi, V. Pascucci and M. Vicentino: Geometric Programming: A Programming Approach to Geometric Design. ACM Transactions on Graphics 14(3): 266-306 (1995)

- 1 geometric calculus in FL-style

PLaSM Basics

PLaSM = Geometric extension of the FL language by Backus (developed at IBM Research)

A. Paoluzzi, V. Pascucci and M. Vicentino: Geometric Programming: A Programming Approach to Geometric Design. ACM Transactions on Graphics 14(3): 266-306 (1995)

- ① geometric calculus in FL-style
- ② dimension independence

PLaSM Basics

PLaSM = Geometric extension of the **FL** language by **Backus** (developed at IBM Research)

*A. Paoluzzi, V. Pascucci and M. Vicentino: **Geometric Programming: A Programming Approach to Geometric Design**. **ACM Transactions on Graphics** 14(3): 266-306 (1995)*

- ① geometric calculus in FL-style
- ② dimension independence
- ③ dynamic typing

PLaSM Basics

PLaSM = Geometric extension of the **FL** language by **Backus** (developed at IBM Research)

*A. Paoluzzi, V. Pascucci and M. Vicentino: **Geometric Programming: A Programming Approach to Geometric Design**. **ACM Transactions on Graphics** 14(3): 266-306 (1995)*

- ① geometric calculus in FL-style
- ② dimension independence
- ③ dynamic typing
- ④ higher-level operators

PLaSM Basics

PLaSM = Geometric extension of the **FL** language by **Backus** (developed at IBM Research)

*A. Paoluzzi, V. Pascucci and M. Vicentino: **Geometric Programming: A Programming Approach to Geometric Design**. **ACM Transactions on Graphics** 14(3): 266-306 (1995)*

- ① geometric calculus in FL-style
- ② dimension independence
- ③ dynamic typing
- ④ higher-level operators
- ⑤ arity: always 1 (number of arguments of functions)

PLaSM Basics

PLaSM = Geometric extension of the **FL** language by **Backus** (developed at IBM Research)

*A. Paoluzzi, V. Pascucci and M. Vicentino: **Geometric Programming: A Programming Approach to Geometric Design**. **ACM Transactions on Graphics** 14(3): 266-306 (1995)*

- ① geometric calculus in FL-style
- ② dimension independence
- ③ dynamic typing
- ④ higher-level operators
- ⑤ arity: always 1 (number of arguments of functions)
- ⑥ small set of predefined functionals

PLaSM Basics

PLaSM = Geometric extension of the **FL** language by **Backus** (developed at IBM Research)

*A. Paoluzzi, V. Pascucci and M. Vicentino: **Geometric Programming: A Programming Approach to Geometric Design**. **ACM Transactions on Graphics** 14(3): 266-306 (1995)*

- ① geometric calculus in FL-style
- ② dimension independence
- ③ dynamic typing
- ④ higher-level operators
- ⑤ arity: always 1 (number of arguments of functions)
- ⑥ small set of predefined functionals
- ⑦ names of functions: all-caps

PLaSM Basics (AA: Apply-to-All)

```
>>> AA(SUM)([[1,2,3],[4,5,6]])  
[6,15]
```

PLaSM Basics (AA: Apply-to-All)

```
>>> AA(SUM)([[1,2,3],[4,5,6]])  
[6,15]
```

```
>>> mat = [[1,2,3],[4,5,6]]  
>>> [sum(v) for v in mat]  
[6,15]
```

PLaSM Basics (DISTL: DISTribute-Left)

```
>>> DISTL([2, [1, 2, 3]])  
[[2, 1], [2, 2], [2, 3]]
```

PLaSM Basics (DISTL: DISTribute-Left)

```
>>> DISTL([2, [1, 2, 3]])  
[[2, 1], [2, 2], [2, 3]]
```

```
>>> DISTL([2, []])  
[]
```

PLaSM Basics (TRANS: TRANSpose)

```
>>> TRANS([[1,2,3],[10,20,30],[100,200,300]])  
[[1,10,100],[2,20,200],[3,30,300]]
```

PLaSM Basics (TRANS: TRANSpose)

```
>>> TRANS([[1,2,3],[10,20,30],[100,200,300]])  
[[1,10,100],[2,20,200],[3,30,300]]
```

```
>>> TRANS([[1,2,3,4,5],[10,20,30,40,50]])  
[[1,10],[2,20],[3,30],[4,40],[5,50]]
```


PLaSM Basics (TRANS: TRANSpose)

```
>>> TRANS([[1,2,3],[10,20,30],[100,200,300]])  
[[1,10,100],[2,20,200],[3,30,300]]
```

```
>>> TRANS([[1,2,3,4,5],[10,20,30,40,50]])  
[[1,10],[2,20],[3,30],[4,40],[5,50]]
```

```
>>> TRANS([[],[]])  
[]
```

PLaSM Basics (arithmetic ops)

```
>>> PROD([3,4])  
12
```

PLaSM Basics (arithmetic ops)

```
>>> PROD([3,4])
```

```
12
```

```
>>> PROD([[1,2,3],[4,5,6]])
```

```
32.0
```

PLaSM Basics (arithmetic ops)

```
>>> PROD([3,4])
```

```
12
```

```
>>> PROD([[1,2,3],[4,5,6]])
```

```
32.0
```

```
>>> SUM([3,4])
```

```
7
```

PLaSM Basics (arithmetic ops)

```
>>> PROD([3,4])  
12
```

```
>>> PROD([[1,2,3],[4,5,6]])  
32.0
```

```
>>> SUM([3,4])  
7
```

```
>>> SUM([[1,2,3],[4,5,6]])  
[5, 7, 9]
```

PLaSM Basics (product scalar by vector)

```
>>> SCALARVECTPROD([3, [1, 2, 3]])  
[3, 6, 9]
```

PLaSM Basics (product scalar by vector)

```
>>> SCALARVECTPROD([3, [1, 2, 3]])  
[3, 6, 9]
```

```
>>> SCALARVECTPROD([4, [10, 20, 30]])  
[40, 80, 120]
```

Pyplasm: Exercise 1 (INNERPROD)

The inner (or scalar) product of $a, b \in \mathbb{R}^m$ is a number

$$\text{INNERPROD} : \mathbb{R}^m \times \mathbb{R}^m \rightarrow \mathbb{R} : (u, v) \mapsto \sum_{i=1}^m u_i v_i$$

Pyplasm: Exercise 1 (INNERPROD)

The **inner (or scalar) product** of $a, b \in \mathbb{R}^m$ is a number

$$\text{INNERPROD} : \mathbb{R}^m \times \mathbb{R}^m \rightarrow \mathbb{R} : (u, v) \mapsto \sum_{i=1}^m u_i v_i$$

```
>>> u = [1,2,3]
>>> v = [10,20,30]
>>> INNERPROD([u, v])
140
```

Pyplasm: Exercise 2 (VECTNORM)

The **norm** of a vector $a \in \mathbb{R}^m$ is a number.

$$\text{VECTNORM} : \mathbb{R}^m \rightarrow \mathbb{R} : v \mapsto \sqrt{\sum_{i=1}^m v_i^2}$$

Pyplasm: Exercise 2 (VECTNORM)

The **norm** of a vector $a \in \mathbb{R}^m$ is a number.

$$\text{VECTNORM} : \mathbb{R}^m \rightarrow \mathbb{R} : v \mapsto \sqrt{\sum_{i=1}^m v_i^2}$$

```
>>> a = [1,2,3]
>>> VECTNORM(a)
3.7416574954986572
```

Pyplasm: Exercise 3 (UNITVECT)

The **unit vector** is a function

$$\text{UNITVECT} : \mathbb{R}^m \rightarrow \mathbb{R}^m : v \mapsto \frac{v}{|v|}$$

Pyplasm: Exercise 3 (UNITVECT)

The `unit vector` is a function

$$\text{UNITVECT} : \mathbb{R}^m \rightarrow \mathbb{R}^m : v \mapsto \frac{v}{|v|}$$

```
>>> v = [1,2,3]
>>> UNITVECT(v)
[0.26726123690605164, 0.5345224738121033, 0.8017836809158325]
```

Pyplasm: Exercise 3 (UNITVECT)

The `unit vector` is a function

$$\text{UNITVECT} : \mathbb{R}^m \rightarrow \mathbb{R}^m : v \mapsto \frac{v}{|v|}$$

```
>>> v = [1,2,3]
>>> UNITVECT(v)
[0.26726123690605164, 0.5345224738121033, 0.8017836809158325]

>>> VECTNORM(UNITVECT(v))
0.9999999403953552  1
```

Pyplasm: Exercise 4 (SUM)

SUM adds m vectors in \mathbb{R}^n , i.e. the rows of a matrix in \mathbb{R}_n^m :

```
>>> a = [1,2,3]
```

```
>>> a
```

```
[1, 2, 3]
```

```
>>> b = [10,20,30]
```

```
>>> b
```

```
[10, 20, 30]
```

```
>>> SUM([a,b])
```

```
[11, 22, 33]
```

Pyplasm: Exercise 5 (SUM)

```
>>> a = range(10)
>>> a
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

>>> b = [10*k for k in range(10)]
>>> b
[0, 10, 20, 30, 40, 50, 60, 70, 80, 90]

>>> SUM([a,b])
[0, 11, 22, 33, 44, 55, 66, 77, 88, 99]

>>> c = [100*k for k in range(10)]
>>> c
[0, 100, 200, 300, 400, 500, 600, 700, 800, 900]

>>> SUM([a,b,c])
[0, 111, 222, 333, 444, 555, 666, 777, 888, 999]
```


Pyplasm: Exercise 6 (MATSUM)

Assignment

Write a function that adds any two matrices $[A], [B]$ (compatible by sum). Both $[A], [B]$ must belong to the same linear space \mathbb{R}_n^m

Pyplasm: Exercise 6 (MATSUM)

Assignment

Write a function that adds any two matrices $[A], [B]$ (compatible by sum). Both $[A], [B]$ must belong to the same linear space \mathbb{R}_n^m

```
>>> def MATSUM(args):  
...     return AA(AA(SUM)) (AA(TRANS)(TRANS(args)))  
  
>>> A = [ [1,2,3], [4,5,6], [7,8,9] ]  
>>> B = [ [10,20,30], [40,50,60], [70,80,90] ]
```

Pyplasm: Exercise 6 (MATSUM)

Assignment

Write a function that adds any two matrices $[A], [B]$ (compatible by sum). Both $[A], [B]$ must belong to the same linear space \mathbb{R}_n^m

```
>>> def MATSUM(args):
...     return AA(AA(SUM)) (AA(TRANS)(TRANS(args)))

>>> A = [ [1,2,3], [4,5,6], [7,8,9] ]
>>> B = [ [10,20,30], [40,50,60], [70,80,90] ]

>>> MATSUM([A,B])
[ [11,22,33], [44,55,66], [77,88,99] ]

>>> MATSUM([A,B,A])
[ [12,24,36], [48,60,72], [84,96,108] ]

>>> MATSUM([A,B,B,A])
[ [22,44,66], [88,110,132], [154,176,198] ]
```

Pyplasm: Exercise 7 (MATPROD)

Write a function that multiplies two matrices (compatible by product)

Remember that

$$A \in \mathbb{R}_n^m, \quad B \in \mathbb{R}_p^n, \quad \text{and} \quad C = AB \in \mathbb{R}_p^m,$$

with

$$C = \left(c_j^i \right) = \left(\mathbf{A}^i \mathbf{B}_j \right), \quad 1 \leq i \leq m, 1 \leq j \leq p,$$

where \mathbf{A}^i is the i -th row of \mathbf{A} , and \mathbf{B}_j is the j -th column of \mathbf{B} .

Pyplasm: Exercise 7 (MATPROD) – Solution

Write a function that multiplies two compatible matrices

```
>>> def MATPROD(args):  
...     A,B = args  
...     return AA(AA(INNERPROD)) (AA(DISTL) (DISTR ([A, TRANS (B)])))
```

Pyplasm: Exercise 7 (MATPROD) – Solution

Write a function that multiplies two compatible matrices

```
>>> def MATPROD(args):
...     A,B = args
...     return AA(AA(INNERPROD)) (AA(DISTL) (DISTR ([A, TRANS (B)])))

>>> A = [[1,2,3],[4,5,6],[7,8,9]]
>>> B = [[1,2,3],[4,5,6],[7,8,9]]
>>> MATPROD ([A,B])
[ [30, 36, 42], [66,81,96], [102,126,150] ]
```

Pyplasm: Exercise 7 (MATPROD) – Solution

Write a function that multiplies two compatible matrices

```
>>> def MATPROD(args):
...     A,B = args
...     return AA(AA(INNERPROD)) (AA(DISTL) (DISTR ([A, TRANS (B)])))

>>> A = [[1,2,3],[4,5,6],[7,8,9]]
>>> B = [[1,2,3],[4,5,6],[7,8,9]]
>>> MATPROD ([A,B])
[ [30, 36, 42], [66,81,96], [102,126,150] ]

>>> C = [[1,2,3],[4,5,6]]
>>> D = [[1,2],[4,5],[7,8]]
>>> MATPROD ([C,D])
[ [30,36], [66,81] ]
```

Pyplasm: Exercise 8 (some array operators)

Look at some PLaSM operators on [arrays](#)

```
>>> N(3) (0)    # REPEAT  
[0,0,0]  
>>> N(3) ([0,1])  
[ [0,1], [0,1], [0,1] ]
```


Pyplasm: Exercise 8 (some array operators)

Look at some PLaSM operators on [arrays](#)

```
>>> N(3) (0)    # REPEAT
```

```
[0,0,0]
```

```
>>> N(3) ([0,1])
```

```
[ [0,1], [0,1], [0,1] ]
```

```
>>> NN(3) ([0,1])  # REPEAT List & Catenate -- REPLICA
```

```
[ 0,1, 0,1, 0,1 ]
```

Pyplasm: Exercise 8 (some array operators)

Look at some PLaSM operators on [arrays](#)

```
>>> N(3) (0)    # REPEAT
```

```
[0,0,0]
```

```
>>> N(3) ([0,1])
```

```
[ [0,1], [0,1], [0,1] ]
```

```
>>> NN(3) ([0,1])  # REPEAT List & Catenate -- REPLICA
```

```
[ 0,1, 0,1, 0,1 ]
```

```
>>> AR ([ [0,0,0], 1 ])  # Append Right
```

```
[0,0,0,1]
```

Pyplasm: Exercise 8 (some array operators)

Look at some PLaSM operators on [arrays](#)

```
>>> N(3) (0)    # REPEAT
```

```
[0,0,0]
```

```
>>> N(3) ([0,1])
```

```
[ [0,1], [0,1], [0,1] ]
```

```
>>> NN(3) ([0,1]) # REPEAT List & CATenate -- REPLICA
```

```
[ 0,1, 0,1, 0,1 ]
```

```
>>> AR ([ [0,0,0], 1 ]) # Append Right
```

```
[0,0,0,1]
```

```
>>> AL ([ 1, [0,0,0] ]) # Append Left
```

```
[1,0,0,0]
```

Pyplasm: Exercise 9 (VECTPROD)

the vector product \mathbf{w} of vectors in \mathbb{R}^3 is defined as the function

$$\mathbb{R}^3 \times \mathbb{R}^3 \rightarrow \mathbb{R}^3 : (\mathbf{u}, \mathbf{v}) \mapsto \det \begin{pmatrix} \mathbf{e}_0 & \mathbf{e}_1 & \mathbf{e}_2 \\ u_x & u_y & u_z \\ v_x & v_y & v_z \end{pmatrix}$$

Therefore we can write, for the vector product of two 3D vector:

```
>>> def VECTPROD(args):
...     u,v = args
...     w = [0,0,0]
...     w[0] = u[1]*v[2] - u[2]*v[1]
...     w[1] = u[2]*v[0] - u[0]*v[2]
...     w[2] = u[0]*v[1] - u[1]*v[0]
...     return w
```

Pyplasm: Exercise 9 (VECTPROD)

the vector product \mathbf{w} of vectors in \mathbb{R}^3 is defined as the function

$$\mathbb{R}^3 \times \mathbb{R}^3 \rightarrow \mathbb{R}^3 : (\mathbf{u}, \mathbf{v}) \mapsto \det \begin{pmatrix} \mathbf{e}_0 & \mathbf{e}_1 & \mathbf{e}_2 \\ u_x & u_y & u_z \\ v_x & v_y & v_z \end{pmatrix}$$

Therefore we can write, for the vector product of two 3D vector:

```
>>> def VECTPROD(args):
...     u,v = args
...     w = [0,0,0]
...     w[0] = u[1]*v[2] - u[2]*v[1]
...     w[1] = u[2]*v[0] - u[0]*v[2]
...     w[2] = u[0]*v[1] - u[1]*v[0]
...     return w
```

```
>>> VECTPROD([[1,0,0], [0,1,0]])
[0,0,1]
>>> VECTPROD([[1,1,0], [0,1,0]])
[0,0,1]
```

Pyplasm: Exercise 10

```
>>> from random import random

>>> def randomPoints(m, sx=1, sy=1):
...     def point():
...         return [random() * sx, random() * sy]
...     return [point() for k in range(m)]
```

Pyplasm: Exercise 10

```
>>> from random import random

>>> def randomPoints(m, sx=1, sy=1):
...     def point():
...         return [random() * sx, random() * sy]
...     return [point() for k in range(m)]

>>> verts = randomPoints(200, 2*PI, 2)
>>> obj = MKPOL([verts, AA(LIST)(range(200)), None])
```

Pyplasm: Exercise 10

```
>>> from random import random

>>> def randomPoints(m, sx=1, sy=1):
...     def point():
...         return [random() * sx, random() * sy]
...     return [point() for k in range(m)]

>>> verts = randomPoints(200, 2*PI, 2)
>>> obj = MKPOL([verts, AA(LIST)(range(200)), None])

>>> VIEW(obj)
```


Pyplasm: Exercise 11

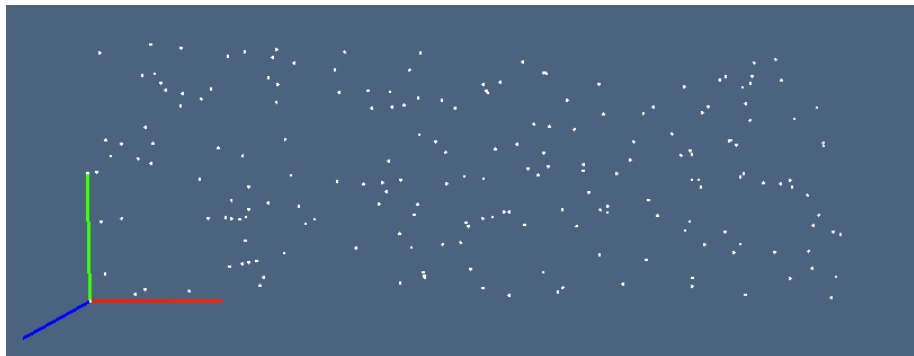


Figure 1: 200 random points in $[0, 2\pi] \times [0, 2] \subset \mathbb{E}^2$

Pyplasm: Exercise 12

coordinate functions

```
>>> def x (p):  
...     u,v = p  
...     return v * COS(u)
```

```
>>> def y (p):  
...     u,v = p  
...     return v * SIN(u)
```

Pyplasm: Exercise 12

coordinate functions

```
>>> def x (p):  
...     u,v = p  
...     return v * COS(u)  
  
>>> def y (p):  
...     u,v = p  
...     return v * SIN(u)  
  
>>> obj = MAP([ x,y ])(obj)
```

Pyplasm: Exercise 12

coordinate functions

```
>>> def x (p):  
...     u,v = p  
...     return v * COS(u)  
  
>>> def y (p):  
...     u,v = p  
...     return v * SIN(u)  
  
>>> obj = MAP([ x,y ])(obj)  
  
>>> VIEW(obj)
```

Pyplasm: Exercise 12 (4/4)

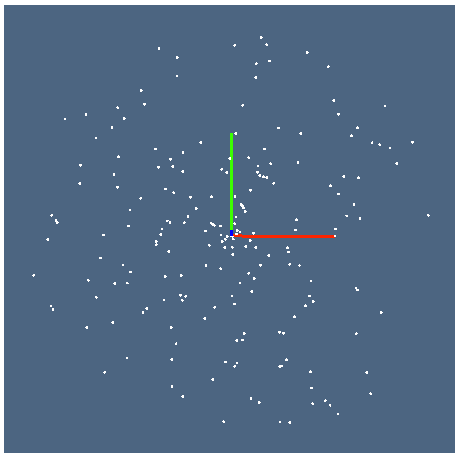


Figure 2: 200 random points within the 2D "ball" of radius 2

From PLaSM to Pyplasm

application (binary infix operator $:$) to (\dots)

$$f : x \rightarrow f(x)$$

composition (binary infix operator \sim) to COMP

$$f \sim g \rightarrow COMP([f, g])$$

construction (of a vector function) to CONS

$$[f, g] : x \rightarrow CONS([f, g])(x)$$

sequence (arrow parentheses $\langle \dots \rangle$) to list

$$f : \langle x_1, x_2, \dots, x_n \rangle \rightarrow f([x_1, x_2, \dots, x_n])$$

From PLaSM to Pyplasm

The original FL syntax

```
hpc = MAP:f:dom
WHERE
    f = [COS~S1, SIN~S1],
    dom = INTERVALS: (2*PI): 24
END;

DRAW:hpc
```

Ported syntactically to python

```
>>> f = CONS([ COMP([COS,S1]), COMP([SIN,S1]) ])
>>> dom = INTERVALS(2*PI)(24)
>>> hpc = MAP(f)(dom)
>>> VIEW(hpc)
```

Using properly the Python syntax

The **function** to be mapped is from **d -points** to lists of **coordinate functions** $\mathbb{R}^d \rightarrow \mathbb{R}$

```
>>> def circle(p):  
...     alpha = p[0]  
...     return [COS(alpha), SIN(alpha)]
```


Using properly the Python syntax

The **function** to be mapped is from **d -points** to lists of **coordinate functions** $\mathbb{R}^d \rightarrow \mathbb{R}$

```
>>> def circle(p):
...     alpha = p[0]
...     return [COS(alpha), SIN(alpha)]

>>> obj = MAP(circle)(INTERVALS(2*PI)(32))
>>> VIEW(obj)
```

In case of a **curve**, $d = 1$

Current plasm.js Library

a subset of pyplasm: see [fenvs.py](#))

AA	EMBED	LIST	SET
AL	EXPLODE	MAP	SIMPLEX
APPLY	EXTRUDE	MAT	SIMPLEXGRID
AR	FIRST	MATPROD	SimplicialComplex
BIGGER	FREE	MATSUM	SKELETON
BIGGEST	Graph	MUL	SMALLER
BOUNDARY	GRAPH	PointSet	SMALLEST
BUTLAST	HELIX	POLYLINE	SORTED
CART	ID	POLYMARKER	SUB
CAT	IDNT	PRECISION	SUM
CENTROID	IDNT	PRINT	T
CIRCLE	INNERPROD	PROD	TAIL
CLONE	INSL	PROGRESSIVE_SUM	Topology
CODE	INSR	QUADMESH	TORUSSOLID
COMP	INTERVALS	R	TORUSSURFACE
CONS	INV	REPEAT	TRANS
CUBE	ISFUN	REPLICA	TREE
CUBOID	ISNUM	REVERSE	TRIANGLEARRAY
CYLSOLID	K	S	TRIANGLEFAN
CYLSURFACE	LAST	S0	TRIANGLESTRIP
DISK	LEN	S1	UNITVECT
DISTL	Linspace1D	S2	VECTNORM
DISTR	Linspace2D	S3	VECTPROD
DIV	Linspace3D	S4	

Matrices

Efficient matrix calculus with NumPy and SciPy

docs.scipy.org

NumPy for Matlab Users

'array' or 'matrix'? Which should I use?

Use arrays

- They are the standard vector/matrix/tensor type of numpy. Many numpy function return arrays, not matrices.

The only disadvantage of using the array type is that you will have to use `dot` instead of `*` to multiply (reduce) two tensors (scalar product, matrix vector multiplication etc.).

'array' or 'matrix'? Which should I use?

Use arrays

- They are the standard vector/matrix/tensor type of numpy. Many numpy function return arrays, not matrices.
- There is a clear distinction between element-wise operations and linear algebra operations.

The only disadvantage of using the array type is that you will have to use `dot` instead of `*` to multiply (reduce) two tensors (scalar product, matrix vector multiplication etc.).

'array' or 'matrix'? Which should I use?

Use arrays

- They are the standard vector/matrix/tensor type of numpy. Many numpy function return arrays, not matrices.
- There is a clear distinction between element-wise operations and linear algebra operations.
- You can have standard vectors or row/column vectors if you like.

The only disadvantage of using the array type is that you will have to use `dot` instead of `*` to multiply (reduce) two tensors (scalar product, matrix vector multiplication etc.).

Assignments

READ:

NumPy: The Numpy array object

References

Python Scientific Lecture Notes