

Python-csa tutorial v0.2

Mikael Djurfeldt

2011-01-17

Contents

1	Purpose of this document	4
2	Introduction	5
3	Tutorial	6
3.1	Basic concepts	6
3.2	Random connectivity	9
3.3	The block operator	9
3.4	Geometry	10
3.5	A network with gaussian connectivity	12
4	Reference	13
4.1	Classes	13
4.1.1	ConnectionSet	13
4.1.2	Mask	14
4.1.3	ValueSet	14
4.1.4	IntervalSet	14
4.2	Constructor and selectors	14
4.2.1	cset	14
4.2.2	mask	15
4.2.3	value	15
4.2.4	arity	15
4.2.5	vset	15
4.3	Integer sets	15
4.3.1	ival	16
4.3.2	N	16
4.3.3	cross	16
4.4	Utilities	16
4.5	Elementary masks	17
4.5.1	empty	17
4.5.2	full	17
4.5.3	oneToOne	17
4.5.4	random	18
4.6	Set operators	18
4.7	Arithmetic operators	18
4.8	Operator application	18
4.9	Miscellaneous connection-set operators	19
4.9.1	random	19

4.9.2	disc	19
4.9.3	gaussian	19
4.9.4	block	20
4.9.5	block1	20
4.9.6	transpose	20
4.9.7	shift	20
4.9.8	fix	20
4.10	Geometry	20
4.10.1	grid2d	20
4.10.2	random2d	21
4.10.3	euclidMetric2d	21
4.10.4	ProjectionOperator	21
4.11	Plotting	21
4.11.1	show	21
4.11.2	gplotsel2d	21

Chapter 1

Purpose of this document

This is a preliminary documentation and tutorial for the python-csa demonstration implementation in Python of the Connection-Set Algebra (Djurfeldt, 2011, submitted)

The CSA library provides elementary connection-sets and operators for combining them. It also provides an iteration interface to such connection-sets enabling efficient iteration over existing connections with a small memory footprint also for very large networks. The CSA can be used as a component of neuronal network simulators or other tools.

Section 2 introduces some basic concepts while section 3 provides some *hands-on* material for getting started. Section 4 contain a preliminary reference documentation.

Chapter 2

Introduction

When building a neuronal network model, we often want to connect one set of neurons—the *source* set—with another set—the *target* set. When applying the Connection-Set Algebra (hereafter denoted *CSA*), we start by *enumerating* the source and target sets, i.e. we assign arbitrary integer indices to the neurons of each set. This allows us to represent a connection between source neuron number 3 and target neuron number 17 as a pair of integers (3, 17). More generally, the source and target sets do not need to be neurons. For example, the target set might be a set of synaptic sites. Also, source and target sets can be (and is often) the same set. This is the case when using CSA to describe connectivity within a neuronal population.

- A *mask* contains information about which connections exist. It is a set of (source, target) pairs, one pair for each existing connection. It can also be regarded as a function mapping a pair of arbitrary non-negative integers to a boolean value—*true* for each existing connection.
- A *value-set* is a function mapping each existing connection to a value, such as a synaptic weight.
- A *connection-set* is a tuple of a mask and zero or more value sets.

CSA connection sets are usually infinite. This is a simplification compared to the common situation of finite source and target sets in that the sizes of these sets do not need to be considered. Connection sets can have arbitrary values associated with connections. Pure connection sets without any values associated are called masks.

Chapter 3

Tutorial

3.1 Basic concepts

To get access to the CSA in Python, type:

```
from csa import *
```

The mask representing all possible connections between an infinite source and target set is:

```
full
```

To display a finite portion of the corresponding connectivity matrix, type:

```
show (full)
```

One-to-one connectivity (where source node 0 is connected to target node 0, source 1 to target 1 etc) is represented by the mask `oneToOne` (Figure 3.1):

```
show (oneToOne)
```

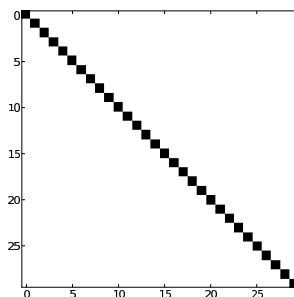


Figure 3.1: `oneToOne`

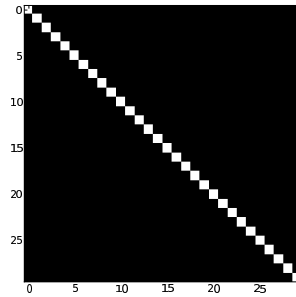


Figure 3.2: `full - oneToOne`

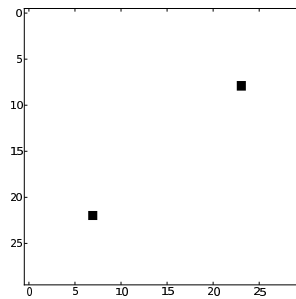


Figure 3.3: `[(22, 7), (8, 23)]`

The default portion displayed by "show" is $(0, 29) \times (0, 29)$. $(0, 99) \times (0, 99)$ can be displayed using:

```
show (oneToOne, 100, 100)
```

If source and target set is the same, `oneToOne` describes self-connections. We can use CSA to compute the set of connections consisting of all possible connections except for self-connections using the set difference operator "-" (Figure 3.2):

```
show (full - oneToOne)
```

Finite connection sets can be represented using either lists of connections, with connections represented as tuples (Figure 3.3):

```
show([(22, 7), (8, 23)])
```

or using the Cartesian product of intervals (Figure 3.4):

```
show(cross(xrange(10), xrange(20)))
```

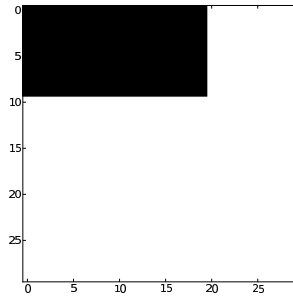


Figure 3.4: `xrange (10), xrange (20)`

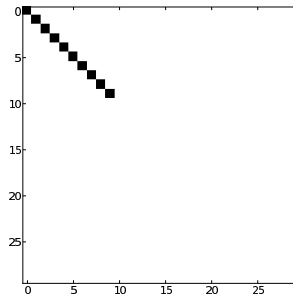


Figure 3.5: `cross (xrange (10), xrange (10)) * oneToOne`

We can form a finite version of the infinite `oneToOne` by taking the intersection `"*"` with a finite connection set (Figure 3.5):

```
c = cross (xrange (10), xrange (10)) * oneToOne
show (c)
```

Finite connection sets can be tabulated:

```
>>> tabulate(c)
0      0
1      1
2      2
3      3
4      4
5      5
6      6
7      7
8      8
9      9
```

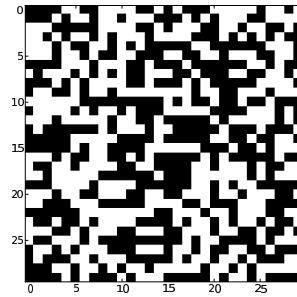



Figure 3.6: `random (0.5)`

In Python, finite connection sets provide an iterator interface:

```
>>> for x in cross (xrange (4), xrange (4)) * oneToOne:
...     print x
...
(0, 0)
(1, 1)
(2, 2)
(3, 3)
```

3.2 Random connectivity

Connectivity where the existence of each possible connection is determined by a Bernoulli trial with probability p is expressed with the random mask `random (p)`, e.g. (Figure 3.6):

```
show (random (0.5))
```

3.3 The block operator

The block operator expands each connection in the operand into a rectangular block in the resulting connection matrix, e.g. (Figure 3.7):

```
show (block (5,3) * random (0.5))
```

Note that `""` here means operator application (see section 4.8). There is also a quadratic version of the operator:

```
show (block (10) * random (0.7))
```

Using intersection and set difference, we can now formulate a more complex mask:

```
show (block (10) * random (0.7) * random (0.5) - oneToOne)
```

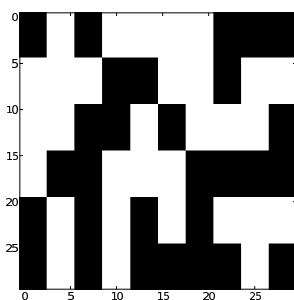


Figure 3.7: `block (5,3) * random (0.5)`

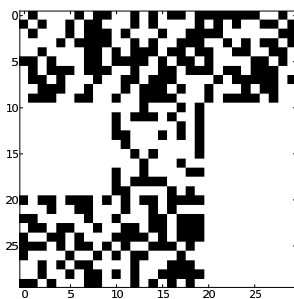


Figure 3.8: `block (10) * random (0.7) * random (0.5) - oneToOne`

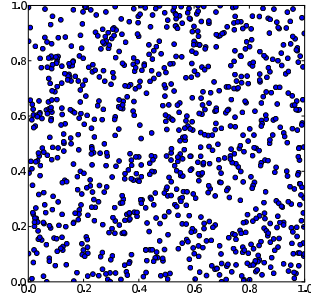


Figure 3.9: `gplot2d (random2d (900), 900)`

The block operator is especially useful when creating connectivity with hierarchical substructure, such as a set of cortical columns.

3.4 Geometry

In CSA, the basic tool to handle distance dependent connectivity is metrics. Metrics are value sets $d(i, j)$. Metrics can be defined through geometry functions. A geometry function maps an index to a position. We can, for example, assign a random position in the unit square to each index:

```
g = random2d (900)
```

The positions of the grid described by g have indices from 0 to 899 and can be displayed like this:

```
gplot2d (g, 900)
```

Alternatively, we can arrange indices in a 30 x 30 grid within the unit square:

```
g = grid2d (30)
```

We can now define the euclidean metric on this grid:

```
d = euclidMetric2d (g)
```

An example of a distance dependent connection set is the disc mask $\text{Disc}(r) * d$ which connects each index i to all indices j within a distance $d(i, j) \leq r$:

```
c = disc (r) * d
```

To examine the result we can employ the function `gplotsel2d (g, c, i)` which displays the targets $g(j)$ of i in the connection set c (Figure 3.11):

```
gplotsel2d (g, c, 434)
```

[A known bug in the current implementation makes the above expression crash. This only happens for infinite sets like c and can be amended by intersecting it with a finite set: `cross (xrange (900), xrange (900)) * c`.]

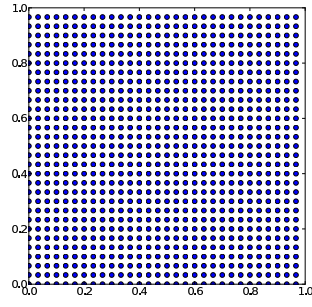


Figure 3.10: `gplot2d (grid2d (30), 900)`

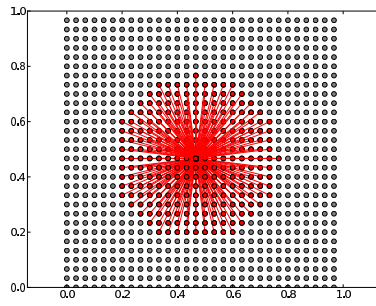


Figure 3.11: Projection from source neuron #434 in `disc (0.3) * d`.

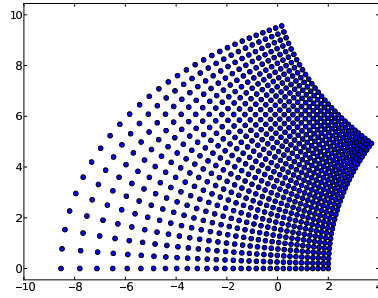


Figure 3.12: `gplot2d (GvspaceToCx * g1, 900)`

In the case where the connection set represents a projection between two different coordinate systems, we define one geometry function for each. In the following example `g1` is direction in visual space in arc minutes while `g2` is position in the cortical representation of the Macaque fovea in mm:

```
g1 = grid2d (30)
g2 = grid2d (30, x0 = -7.0, xScale = 8.0, yScale = 8.0)
```

We now define a projection operator which takes visual coordinates into cortical (Dow et al. 1985):

```
import cmath

@ProjectionOperator
def GvspaceToCx (p):
    w = 7.7 * cmath.log (complex (p[0] + 0.33, p[1]))
    return (w.real, w.imag)
```

To see how the grid `g1` is transformed into cortical space, we type:

```
gplot2d (GvspaceToCx * g1, 900)
```

The inverse projection is defined:

```
@ProjectionOperator
def GcxToVspace (p):
    c = cmath.exp (complex (p[0], p[1]) / 7.7) - 0.33
    return (c.real, c.imag)
```

Real receptive field sizes vary with eccentricity. Assume, for now, that we want to connect each target index to sources within a disc of constant radius. We then need to project back into visual space and use the disc operator:

```
c = disc (0.1) * euclidMetric2d (g1, GcxToVspace * g2)
```

Again, we use `gplotsel2d` to check the result (Figure 3.13):

```
gplotsel2d (g2, c, 282)
```

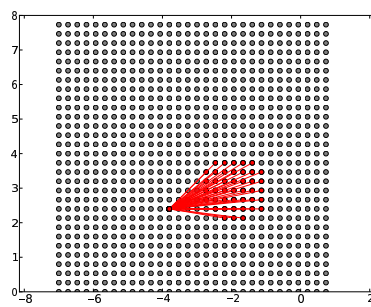


Figure 3.13: `disc (0.1) * euclidMetric2d (g1, GcxToVspace * g2)`

3.5 A network with gaussian connectivity

In the following example we represent the connectivity of a network with excitatory and inhibitory neurons and gaussian connectivity in a random geometry using a single connection-set (Figure 3.14).

Example 3.1: Network with gaussian geometry-dependent connectivity

```
from csa import *

# Create index intervals for excitatory, inhibitory
# and all cells
e = ival (0, 599)
i = ival (600, 899)
a = e + i

# Create geometry function g and metric d
g = random2d (900)
d = euclidMetric2d (g)

# Excitatory and inhibitory conductances, computed as
# gaussian value sets (provides the gaussian of the
# distance for every index pair)
g_e = gaussian (0.1, 0.3) * d
g_i = gaussian (0.2, 0.3) * d

# Create connection-sets with gaussian dependent random
# masks, gaussian dependent conductance and distance
# dependent delay: (mask, conductance, delay)
c_e = cset (random * g_e, g_e, d)
c_i = cset (random * g_i, -g_i, d)

# Combine excitatory and inhibitory connectivity into one
# network using intersection (*) and multiset sum (+)
# operators
c = cross (e, a) * c_e + cross (i, a) * c_i

# We may also plot the outgoing connections from one
# excitatory neuron around coordinate (0.33, 0.5) and one
# inhibitory neuron around coordinate (0.67, 0.5)
sources = [g.inverse(0.33,0.5,e), g.inverse(0.67,0.5,i)]
gplotsel2d (g, c, sources, value=0, range=[-1,1])
```

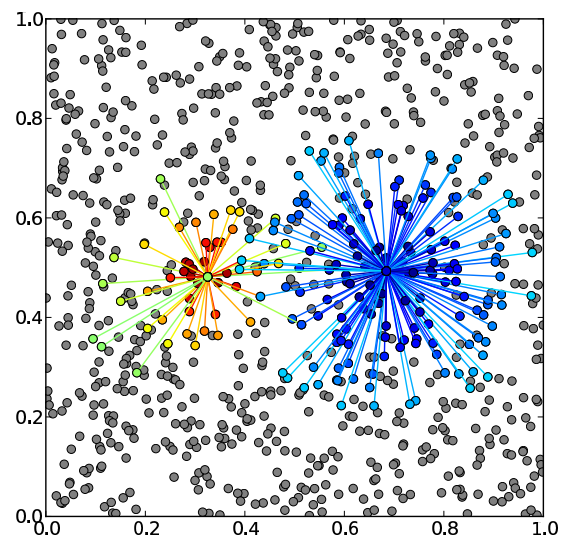


Figure 3.14: Projections of an excitatory (warm colors) and an inhibitory (cold colors).

Chapter 4

Reference

This section documents how to use existing python-csa classes.

4.1 Classes

This section briefly documents some important classes in the python-csa implementation and their public API. The examples use many elements which are defined in later sections. It is suggested to use the index on page 22 to find the reference documentation for these elements.

4.1.1 ConnectionSet

A connection-set can be regarded as a set of connections, represented by their source and target indices, with zero or more associated values. In the CSA, a connection-set with no associated values is a mask. Thus, in the python-csa implementation, in all cases where an instance of the class `ConnectionSet` is expected, it is OK to pass an instance of `Mask`.

`--len--` (`self`)

return value the number of connections in this connection-set

This method returns the number of connections in this connection-set. An error is reported if this connection-set isn't finite.

Example 4.1: Obtaining the number of connections in a connection-set

```
>>> len (cross ((0, 1), (0, 1)))
4
```

`--iter--` (`self`)

return value iterator over the connections represented by this
instance

This method returns an iterator over the connections represented by this instance. Each item generated by the iterator is a tuple

$$(i, j, v_0, \dots, v_{n-1})$$

Example 4.2: Iterating over a connection-set

```
>>> m = cross ((0, 1), (2, 3))
>>> v = vset (lambda i, j: i + j)
>>> c = cset (m, v, v * v)
>>> for x in c:
...     print x
...
(0, 2, 2, 4)
(1, 2, 3, 9)
(0, 3, 3, 9)
(1, 3, 4, 16)
```

4.1.2 Mask

A mask gives information about which connections exist. It can be regarded as a set of connections, represented by their source and target indices. In the CSA, a connection-set with no associated values is a mask. In the python-csa implementation, an attempt to construct a connection-set with zero associated values, yields an instance of the class `Mask`. In cases where a mask is expected, a python list of (source, target) tuples can also be passed.

The class `Mask` has the same public methods (`__len__`, `__iter__`) as the class `ConnectionSet`.

4.1.3 ValueSet

To be documented.

4.1.4 IntervalSet

To be documented.

4.2 Constructor and selectors

4.2.1 cset

```
cset (mask, valueSet, ...)
```

mask	a <code>Mask</code>
valueSet	zero or more <code>ValueSet</code> :s

This function constructs and returns a connection-set from a `Mask` and zero or more `ValueSet`:s. [Note: In the current implementation, `mask` is returned if no value-sets are given. This should probably change so that a new object is returned.]

4.2.2 mask

mask (cset)

cset	a <code>ConnectionSet</code>
<i>return value</i>	the <code>Mask</code> of cset

This function returns the `Mask` of the `ConnectionSet` cset.

4.2.3 value

value (cset, k)

cset	a <code>ConnectionSet</code>
k	index of the value-set to return
<i>return value</i>	the k:th <code>ValueSet</code> of cset

This function returns the k:th `ValueSet` of the `ConnectionSet` cset.

4.2.4 arity

arity (cset)

cset	a <code>ConnectionSet</code>
<i>return value</i>	the <i>arity</i> of cset

This function returns the *arity* of the `ConnectionSet` cset. The *arity* of a connection-set is the number of value-sets of the connection-set.

4.2.5 vset

vset (x)

vset (callable)

x	a value
callable	a callable taking two arguments

This function constructs and returns a value-set. In the first form, the number x is taken as the value of each of all existing connections. In the second form, the value of each existing connection is the one returned by applying `callable` to the source and target indices of the connection.

4.3 Integer sets

In the current python-csa implementation, integer sets are usually represented using the class `IntervalSet` (see section 4.1.4). Functions that take integer sets as arguments generally coerce `tuple`s of two non-negative integers into `IntervalSet`s:

$(1, 2) \longrightarrow \text{IntervalSet } ([(1, 2)])$

4.3.1 ival

ival (*beginning*, *end*)

beginning	start of interval
end	end of interval (inclusive)
<i>return value</i>	the interval (beginning , end)

This function returns the interval (**beginning**, **end**) represented as a set of non-negative integers. The underlying representation is space-efficient.

4.3.2 N

N

This constant represents the set of all non-negative integers.

4.3.3 cross

cross (*set0*, *set1*)

set0	a set of non-negative integers
set1	a set of non-negative integers
<i>return value</i>	the Cartesian cross product of set0 and set1

This function returns the Cartesian cross product of **set0** and **set1** represented as a **Mask**.

Example 4.3: The Cartesian product of (1,2) and (3,4)

```
>>> tabulate (cross ((1,2), (3,4)))
1      3
2      3
1      4
2      4
```

4.4 Utilities

tabulate (*cset*)

cset	a ConnectionSet
-------------	------------------------

This procedure tabulates the connection-set `cset`. An iteration over the connections in `cset` is performed. The source and target indices are tabulated in the first and second columns with value-sets tabulated in columns three and upwards.

Tabulate can be used to print connection-sets during development.

4.5 Elementary masks

4.5.1 empty

empty

This constant `Mask` represents the set of no connection. Iterating results in nothing, no matter how hard you try.

4.5.2 full

full

This constant `Mask` represents the (infinite) set of all connections.

Example 4.4: Finite portion of the `full` mask

```
>>> tabulate (cross ((0, 1), (0, 1)) * full)
0      0
1      0
0      1
1      1
```

4.5.3 oneToOne

oneToOne

This constant `Mask` represents the (infinite) set of one-to-one connections. It resembles Kronecker's delta or an infinite identity matrix.

Example 4.5: Finite portion of the `oneToOne` mask

```
>>> tabulate (cross ((0, 3), (0, 3)) * oneToOne)
0      0
1      1
2      2
3      3
```

4.5.4 random

random (*p*)

<i>p</i>	the probability for a potential connection to exist
<i>return value</i>	an infinite Mask where the existence of each connection is determined by a Bernoulli trial with probability <i>p</i> .

This function returns a random mask where a connection between given source and target indices exists with probability *p*.

See also section 4.9.1 for the set of functions returning random *operators*. These support sampling a given number of connections from a finite mask or random sampling with constraints on **fanIn** or **fanOut**.

4.6 Set operators

The following binary operators can be applied to integer sets, masks and connection-sets:

$A + B$	the <i>multiset sum</i> of A and B
$A - B$	the <i>set difference</i> between A and B
$A * B$	the <i>intersection</i> of A and B

In addition, the following unary operator applies to integer sets and masks:

$\sim A$	the <i>complement</i> of A
----------	----------------------------

4.7 Arithmetic operators

The arithmetic operators on connection-sets which are defined in the connection-set algebra are not yet implemented in the python-csa demo implementation.

4.8 Operator application

The operator application operator is used to apply unary connection-set algebra operators to their operand:

operator * operand	apply operator to operand
---------------------------	---------------------------

The operator application operator is overloaded with the arithmetic multiplication and set intersection operators.

4.9 Miscellaneous connection-set operators

4.9.1 random

random (*N* = *n*) * *cset*

n the number of connections to sample (keyword arg named *N*)
cset any *finite* connection-set
return value a connection-set containing *n* randomly sampled connections from *cset*

random (*fanIn* = *n*) * *cset*

n the number of sources sampled for each target (keyword arg named *fanIn*)
cset any *finite* connection-set
return value a connection-set randomly sampled from *cset* with *fanIn* *n*

random (*fanOut* = *n*) * *cset*

n the number of targets sampled for each source (keyword arg named *fanOut*)
cset any *finite* connection-set
return value a connection-set randomly sampled from *cset* with *fanOut* *n*

4.9.2 disc

disc (*r*) * *metric*

r radius
return value a mask of all connections for which *metric* (*source*, *target*) < *r*

4.9.3 gaussian

gaussian (*sigma*, *cutoff*) * *metric*

sigma
cutoff
return value a value set associating the result of applying the normalized gaussian function with standard deviation *sigma* and cutoff *cutoff* to *metric* (*source*, *target*) to each connection

4.9.4 block

```
block (M, N)
block (M)

M
N
```

4.9.5 block1

```
block1 (N)
```

4.9.6 transpose

```
transpose
```

4.9.7 shift

```
shift (M, N)

M
N
```

4.9.8 fix

```
fix
```

4.10 Geometry

4.10.1 grid2d

```
grid2d (width, xScale = 1.0, yScale = 1.0, x0 = 0.0, y0 = 0.0)

width
xScale
return value
```

4.10.2 random2d

random2d (N, xScale = 1.0, yScale = 1.0)

return value

4.10.3 euclidMetric2d

euclidMetric2d (g1, [g2])

g1
g2 optional
return value

4.10.4 ProjectionOperator

```
@ProjectionOperator
def fname (p):
    ...
    return q

fname
p
```

4.11 Plotting

4.11.1 show

show (cset, N0 = 30, [N1])

cset
N0

4.11.2 gplotsel2d

gplotsel2d (g, cset, source = N, target = N,
N0 = 900, [N1], [value], range=[], lines = True)

gplot2d (g, N, [color], show = True)
