CMCpy Documentation

Release 0.1

David H. Ardell, Peter Becich and Brian Stark

October 26, 2012

CONTENTS

Contents:

CHAPTER

ONE

CMCPY PACKAGE

1.1 cmcpy Package

1.2 __main__ Module

1.3 amino_acid_spaces Module

class amino_acid_spaces.**RegionAminoAcidSpace**(*num_aas=None*, *coords=None*, *num_dims=1*, *seed=42*, *labels=None*)

Bases: amino_acid_spaces._AminoAcidSpace

Region amino acid spaces model amino acid (dis)similarities in bounded regions of a finite number of dimensions.

```
>>> aa = RegionAminoAcidSpace(num_aas = 5,num_dims = 2)
>>> map(lambda x:x.round(2),aa.coords)
[array([[ 0.16,  0.71]]), array([[ 0.37,  0.16]]), array([[ 0.6,  0.6]]), array([[ 0.73,  0.87]]
>>> dm =  aa.get_distance_matrix()
>>> dm.round(3)
array([[ 0.   ,  0.594,  0.455,  0.597,  1.027],
       [ 0.594,  0.   ,  0.498,  0.795,  0.584],
       [ 0.455,  0.498,  0.   ,  0.297,  0.647],
       [ 0.597,  0.795,  0.297,  0.   ,  0.837],
       [ 1.027,  0.584,  0.647,  0.837,  0.   ]])
```

class amino_acid_spaces.**RingAminoAcidSpace**(*num_aas=None*, *seed=42*, *coords=None*, *labels=None*)

Bases: amino_acid_spaces._AminoAcidSpace

Ring amino acid spaces model amino acid (dis)similarities in a one-dimensional circular physicochemical amino acid space

```
>>> aa = RingAminoAcidSpace(num_aas = 5)
>>> map(lambda x: x.round(3),aa.coords)
[array([[ 0.156]]), array([[ 0.375]]), array([[ 0.599]]), array([[ 0.732]]), array([[ 0.951]])]
>>> dm = aa.get_distance_matrix()
>>> dm.round(3)
array([[ 0.   ,  0.219,  0.443,  0.424,  0.205],
       [ 0.219,  0.   ,  0.224,  0.357,  0.424],
       [ 0.443,  0.224,  0.   ,  0.133,  0.352],
       [ 0.424,  0.357,  0.133,  0.   ,  0.219],
       [ 0.205,  0.424,  0.352,  0.219,  0.   ]])
```

1.4 codon_spaces Module

class codon_spaces.**RingCodonSpace**(*num_codons*, *mu*)
Bases: codon_spaces._CodonSpace

Ring codon spaces are wrapped linear mutation spaces where codons mutate only to their two immediate neighbors. For ring codon models, mu defines the probability of change to one of two codon neighbors. The probability of no change is [1 - (2*mu)]

```
>>> codons = RingCodonSpace(num_codons = 5,mu = 0.1)
>>> mm = codons.get_mutation_matrix()
>>> mm.round(3)
array([[ 0.9 ,  0.05,  0.  ,  0.  ,  0.05],
       [ 0.05,  0.9 ,  0.05,  0.  ,  0.  ],
       [ 0.  ,  0.05,  0.9 ,  0.05,  0.  ],
       [ 0.  ,  0.  ,  0.05,  0.9 ,  0.05],
       [ 0.05,  0.  ,  0.  ,  0.05,  0.9 ]])
```

get_derivative_matrix()

post_process_perturbative_solution(*lpert*, *vpert*)

class codon_spaces.**WordCodonSpace**(*num_bases*, *num_positions*, *mu*, *kappa=1.0*)
Bases: codon_spaces._CodonSpace

Word codon spaces model natural codons with a finite number of bases and a finite word-length.

For word codon models, mu defines the total probability of change of a base to any neighbor. The probability of no change of a single base is defined as (1 - mu).

If kappa is not equal to 1.0, then num_bases must be even.

```
>>> codons = WordCodonSpace(num_bases = 2,num_positions = 2, mu = 0.1)
>>> mm = codons.get_mutation_matrix()
>>> mm.round(3)
array([[ 0.81 ,  0.045,  0.045,  0.003],
       [ 0.045,  0.81 ,  0.003,  0.045],
       [ 0.045,  0.003,  0.81 ,  0.045],
       [ 0.003,  0.045,  0.045,  0.81 ]])
>>> codons = WordCodonSpace(num_bases = 4,num_positions = 2, mu = 0.2,kappa = 2)
>>> mm = codons.get_mutation_matrix()
>>> mm.round(3)
```

get_derivative_matrix()
This function exists to serve the perturbative solution in evolvers.py

post_process_perturbative_solution(*lpert*, *vpert*)

1.5 evolvers Module

1.5.1 evolvers – cmcpy module for abstract base class of Ardell Sella Evolvers

class evolvers.**ArdellSellaEvolverAbstractBase**(*initial_code*, *site_types*, *delta*, *epsilon*, *observables*)
Bases: object

Abstract Base Class for ArdellSellaEvolvers for Code-Message Coevolution corresponding to models published in Ardell and Sella (2001, 2002) and Sella and Ardell (2002, 2006). Concrete Implementations subclass from this for different implementations of Eigenvalue solutions.

compute_code_fitness_given_messages(*equilibrated_messages*, *effective_code_matrix*)
Implement eg eqns. 2-7 from Sella and Ardell(2006)

compute_max_fitness_code_mutation()

equilibrate_messages()
Compute eigensystems in site-types for an established genetic code.

This finds eigensystems (codon frequencies and growth rates) for different site-types given the genetic code.

Abstract method: subclasses must:

1) store their results by setting self.eigenvalues and self.eigenmatrix

2.set self.equilibrated to True

3) call super(<<SubClass>>, self).equilibrate_messages() to print observables at end of subclass method where <<SubClass>> is the subclass name to print observables.

evolve_code_unless_frozen()
Mutate genetic code.

Unless genetic code is frozen, mutate it according to the Ardell and Sella models, and update code to most fit mutant if it exists. If no more fit mutant code exists, set the "frozen" attribute to True.

evolve_one_step()
Mutate genetic code and equilibrate messages.

Unless genetic code is frozen, mutate it according to the Ardell and Sella models, update code to most fit mutant if it exists, and equilibrate messages to the new mutant genetic code. If no more fit mutant code exists, set the "frozen" attribute to True.

evolve_until_frozen()
Iteratively evlove genetic code and messages.

Until genetic code is frozen, mutate it according to the Ardell and Sella models, update code to most fit mutant if it exists, and equilibrate messages to the new mutant genetic code. Once no more fit mutant code exists, set the "frozen" attribute to True.

get_eigenvalue(*msm*, *eigenvec*)

get_mutation_selection_matrix(*alpha*)

get_selection_matrix(*alpha*)

growth_rate()

growth_rate_from_lambda()

initial_equilibrate_messages()
Compute eigensystems in site-types for an established genetic code.

This finds eigensystems (codon frequencies and growth rates) for different site-types given the genetic code.

messages()

print_initial_observables()

print_observables()

print_observables_header()

class evolvers.**ArdellSellaEvolverNumpy**(*initial_code*, *site_types*, *num_processes*, *observables*, *delta=1e-32*, *epsilon=1e-12*)
Bases: evolvers.ArdellSellaEvolverAbstractBase

compute_eigensystem(*alpha*, *max_time=60*, *numpy_type=<type 'numpy.float64'>*)

equilibrate_messages()

class evolvers.**ArdellSellaEvolverNumpyMulticore**(*initial_code*, *site_types*, *num_processes*, *observables*, *delta=1e-32*, *epsilon=1e-12*)
Bases: evolvers.ArdellSellaEvolverAbstractBase

equilibrate_messages()

class evolvers.**ArdellSellaEvolverNumpyProcessChild**(*in_queue*, *out_queue*)
Bases: multiprocessing.process.Process

Finds the eigensystem (message equilibrium and growth rate) for an established genetic code

run()

class evolvers.**ArdellSellaEvolverPerturbative**(*initial_code*, *site_types*, *num_processes*, *observables*, *delta=1e-32*, *epsilon=1e-12*)
Bases: evolvers.ArdellSellaEvolverAbstractBase

choose(*n*, *k*)
A fast way to calculate binomial coefficients by Andrew Dalke (contrib).

compute_eigensystem(*alpha*, *max_time=60*, *numpy_type=<type 'numpy.float64'>*)

equilibrate_messages()

class evolvers.**ArdellSellaEvolverPowerCUDA**(*initial_code*, *site_types*, *num_processes*, *observables*, *delta=1e-32*, *epsilon=1e-12*)
Bases: evolvers.ArdellSellaEvolverAbstractBase

equilibrate_messages()

class evolvers.**ArdellSellaEvolverPowerMethod**(*initial_code*, *site_types*, *num_processes*, *observables*, *delta=1e-32*, *epsilon=1e-12*)
Bases: evolvers.ArdellSellaEvolverAbstractBase

compute_eigensystem(*alpha*, *max_time=60*, *numpy_type=<type 'numpy.float64'>*)

equilibrate_messages()

class evolvers.**ArdellSellaEvolverPowerMethodProcessChild**(*in_queue*, *out_queue*, *num_codons*, *delta*, *max_time=60*)
Bases: multiprocessing.process.Process

Finds the eigensystem (message equilibrium and growth rate) for an established genetic code

run()

class evolvers.**ArdellSellaEvolverPowerMulticore**(*initial_code*, *site_types*, *num_processes*, *observables*, *delta=1e-32*, *epsilon=1e-12*)
Bases: evolvers.ArdellSellaEvolverAbstractBase

equilibrate_messages()

in_queue = <multiprocessing.queues.JoinableQueue object at 0x104eb6290>

out_queue = <multiprocessing.queues.JoinableQueue object at 0x104e42510>

class evolvers.**eigensystem_CUDA_implementation**(*parent*, *max_time=60*, *delta=1e-32*)

calculate()

done()

error_check()

get_eigenmatrix()

get_eigenvalue(*alpha*)

get_eigenvalues()

1.6 genetic_codes Module

class genetic_codes.**GeneticCodeMutation**(*code*, *codon*, *aa*)

get_effective_code_matrix()

class genetic_codes.**InitiallyAmbiguousGeneticCode**(*codons*, *amino_acids*, *misreading=None*)

Bases: genetic_codes._GeneticCode

class genetic_codes.**UserInitializedGeneticCode**(*codons*, *amino_acids*, *code_matrix=None*, *code_dict=None*, *misreading=None*)

Bases: genetic_codes._GeneticCode

User-Initialized Genetic Codes are initialized with a numpy.ndarray code matrix or a dict of codons mapping to indices (not labels) of amino acids

```
>>> codons = codon_spaces.WordCodonSpace(num_bases = 4,num_positions = 2, mu = 0.2,kappa = 2)
>>> aas    = amino_acid_spaces.RegionAminoAcidSpace(num_aas = 20, seed = 40)
>>> cm = numpy.eye(16)
>>> cm = numpy.hstack((cm,numpy.zeros((16,4))))
>>> cm.shape
(16, 20)
>>> cm[0][1] = 1
>>> cm /= cm.sum(axis = 1).reshape(16,1)
>>> gc = UserInitializedGeneticCode(codons,aas,code_matrix = cm)
>>> gc.num_codons
16
>>> gc.num_amino_acids
20
>>> gc.ambiguous_codons()
set([0])
>>> gc.encoded_aas
set([1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15])
>>> print gc
|* b c d|
|e f g h|
|i j k l|
|m n o p|
>>> gc.as_labelled_dict()
{0: '*', 1: 'b', 2: 'c', 3: 'd', 4: 'e', 5: 'f', 6: 'g', 7: 'h', 8: 'i', 9: 'j', 10: 'k', 11: 'l
>>> gc.as_dict()
{0: '*', 1: 1, 2: 2, 3: 3, 4: 4, 5: 5, 6: 6, 7: 7, 8: 8, 9: 9, 10: 10, 11: 11, 12: 12, 13: 13, 1
>>> gc2 = UserInitializedGeneticCode(codons,aas,code_dict = {0: '*', 1: 1, 2: 2, 3: 3, 4: 4, 5:
```

```
>>> print gc2
|* b c d|
|e f g h|
|i j k l|
|m n o p|
>>> print gc2.code_matrix[0]
[ 0.05  0.05  0.05  0.05  0.05  0.05  0.05  0.05  0.05  0.05  0.05  0.05
  0.05  0.05  0.05  0.05  0.05  0.05  0.05  0.05]
```

1.7 misreading Module

class misreading.**PositionalMisreading**(*codons*, *misreading*)

Bases: misreading._Misreading

Positional misreading models misreading on word codon spaces which model natural codons with a finite number of bases and a finite word-length.

For positional misreading, the misreading parameter is a list of positional misreading parameters mr_i which define the total misreading probability of a base at position i to any neighbor. The probability of no misreading of a single base at position i is defined as (1 - mr_i).

```
>>> codons = codon_spaces.WordCodonSpace(num_bases = 2,num_positions = 2, mu = 0.1)
>>> misreading = PositionalMisreading(codons,[0.1,0.01])
>>> mr = misreading.get_misreading_matrix()
>>> mr.round(3)
array([[ 0.891,  0.099,  0.009,  0.001],
       [ 0.099,  0.891,  0.001,  0.009],
       [ 0.009,  0.001,  0.891,  0.099],
       [ 0.001,  0.009,  0.099,  0.891]])
```

class misreading.**RingMisreading**(*codons*, *misreading*)

Bases: misreading._Misreading

Ring misreading is one-dimensional misreading uniform over all other codons. For ring misreading, (mr/(nc - 1)) is the probability of misreading as a specific codon. The probability of no misreading is (1 - (mr)).

The misreading parameter is a list with one element, mr.

```
>>> codons = codon_spaces.RingCodonSpace(num_codons = 5,mu = 0.1)
>>> misreading = RingMisreading(codons,[0.1])
>>> mr = misreading.get_misreading_matrix()
>>> mr.round(3)
array([[ 0.9  ,  0.025,  0.025,  0.025,  0.025],
       [ 0.025,  0.9  ,  0.025,  0.025,  0.025],
       [ 0.025,  0.025,  0.9  ,  0.025,  0.025],
       [ 0.025,  0.025,  0.025,  0.9  ,  0.025],
       [ 0.025,  0.025,  0.025,  0.025,  0.9  ]])
```

1.8 observables Module

Control and select output from CMCpy simulations

class observables.**Observables**(*show_codes=True*, *show_messages=False*, *show_initial_parameters=True*, *show_matrix_parameters=False*, *show_fitness_statistics=False*, *show_code_evolution_statistics=False*, *show_frozen_results_only=False*, *print_precision=6*, *show_all=False*)

1.9 site_type_spaces Module

Site-type fitness matrices are intended as site-types over rows and amino acids over columns

class site_type_spaces.**MirroringSiteTypeSpace**(*amino_acids*, *phi*, *weights=None*)

This class models site-types in one-to-one correspondence with amino acids as according to the published models of Ardell and Sella.

```
>>> aa = amino_acid_spaces.RingAminoAcidSpace(num_aas = 5)
>>> dm = aa.get_distance_matrix()
>>> dm.round(3)
array([[ 0.   ,  0.219,  0.443,  0.424,  0.205],
       [ 0.219,  0.   ,  0.224,  0.357,  0.424],
       [ 0.443,  0.224,  0.   ,  0.133,  0.352],
       [ 0.424,  0.357,  0.133,  0.   ,  0.219],
       [ 0.205,  0.424,  0.352,  0.219,  0.   ]])
>>> st = MirroringSiteTypeSpace(aa,phi = 0.96)
>>> fm = st.get_fitness_matrix()
>>> fm.round(3)
array([[ 1.   ,  0.991,  0.982,  0.983,  0.992],
       [ 0.991,  1.   ,  0.991,  0.986,  0.983],
       [ 0.982,  0.991,  1.   ,  0.995,  0.986],
       [ 0.983,  0.986,  0.995,  1.   ,  0.991],
       [ 0.992,  0.983,  0.986,  0.991,  1.   ]])
```

get_fitness_matrix()

get_site_type_weights()

CHAPTER

TWO

CMCPY – CODE-MESSAGE COEVOLUTION MODELS IN PYTHON

CMCpy provides an object-oriented python API, together with command-line interface executables, that implement "Code-Message Coevolution" models. These published evolutionary models pertain to the evolution by natural selection of a genetic code in coevolution with a population of protein-coding genes.

Formally, CMC models are sets of quasispecies coupled together for their fitness through a genetic code. The system alternates between quasispecies equilibration and adaptive hill-climbing through codon assignments and reassignment by code mutation.

CMCpy can reproduce the statistics and results of [Ardell_and_Sella_2001], [Sella_and_Ardell_2002], [Ardell_and_Sella_2002] and [Sella_and_Ardell_2006]. CMCpy additionally implements additional extensions that have not yet been studied in published work. It is easliy feasible to extend the present code-base to implement the model studied by [Vetsigian_et_al_2006].

CMC evolutionary trajectories are partly a sequence of eigensystem solutions. Qualitative differences in results on different platforms can originate from differences in convergence criteria when power method-based eigensystem solvers are used, or from differences in floating point representations. Python defers to the platform C library for float representation. The default eigensystem solver is the eig() function in Numpy.

CHAPTER

THREE

DEPENDENCIES

CMCpy relies heavily on, and absolutely requires, numpy as a prerequisite. You should install numpy with the easy_install framework to be detected as installed when installing this package.

If you wish to play with an experimental CUDA-based power-method eigensystem solver, you must install pycuda. This implementation is not faster than the NumPy default solver for many systems.

CHAPTER

FOUR

INSTALLATION

This installer requires setuptools, the most recent python packaging framework. If you do not already have this installed, this package will install it for you, so long as you have network access. Otherwise preinstall the correct version of setuptools using the EasyInstall installation instructions at http://peak.telecommunity.com/DevCenter/EasyInstall#installation-instructions

If you need to install this package somewhere other than the main site-packages directory, install setuptools using the instructions for Custom Installation Locations before installing this package. The instructions are here: http://peak.telecommunity.com/DevCenter/EasyInstall#custom-installation-locations

Since you have downloaded the source-code package, the easiest way to install the package is to execute (from within the source root directory):

```
easy_install .
```

Mac users may need to run this command with "sudo" prepended.

CHAPTER

FIVE

USAGE

CMCpy comes with an executable inside the bin subdirectory to the installation source package, a UNIX-compatible script called “cmc”.

Additionally, a platform-specific executable may be automatically generated on installation.

Published results with CMC models may be (at least qualitatively) reproduced through the –demo option to the executables.

Also try running the –help option to the executables after installation and for a command-line example.

Programmers may use the executable in bin as a guide and template for how to program against the cmcpy API.

CHAPTER

SIX

DOCUMENTATION

Some documentation of the cmcpy API is available within the “doc” subdirectory of the source distribution. HTML, pdf and texinfo alternative formats are provided.

CHAPTER

SEVEN

LICENSING AND ATTRIBUTION

The CMCpy project is distributed under the terms of the Apache License 2.0 as described in the file LICENSE.txt

Please cite Becich et al. (2012) in all scientific works that use this code.

CHAPTER

EIGHT

RELEASE NOTES

The most recent version is 0.1 released October 2012.

See CHANGES.txt for version-related changes to the CMCpy code-base.

CHAPTER

NINE

REFERENCES

CHAPTER

TEN

INDICES AND TABLES

- *genindex*
- *modindex*
- *search*

BIBLIOGRAPHY

[Ardell$_a nd_S ell a_2$001] D.H. Ardell and G. Sella (2001). On the evolution of redundancy in genetic codes. Journal of Molecular Evolution 53(4/5):269-281.

[Ardell$_a nd_S ell a_2$002] D.H. Ardell and G. Sella (2002). No accident: genetic codes freeze in error-correcting patterns of the standard genetic code. Philosophical Transactions of the Royal Society of London B 357:1625-1642.

[Sella$_a nd_A r dell_2$002] 7. Sella and D.H. Ardell (2002). The impact of message mutation on the fitness of a genetic code. Journal of Molecular Evolution 54(5):638-651.

[Sella$_a nd_A r dell_2$006] 7. Sella and D.H. Ardell (2006). The coevolution of genes and genetic codes: Crick's frozen accident revisited. J. Mol. Evol. 63(3):297-313.

[Vetsigian$_e t_a l_2$006] Vetsigian K., Woese C. R., Goldenfeld N. (2006). Collective evolution and the genetic code. Proc. Natl. Acad. Sci. U.S.A. 103, 10696-10701.

PYTHON MODULE INDEX

INDEX

I

M

O

P

R

S

U

W