PyPedal: Software for pedigree analysis

Release 2.0.0a16

John B. Cole, Ph.D.

May 03, 2005

USDA Animal Improvement Programs Laboratory, Bldg 005 Room 306 BARC-West, 10300 Baltimore Avenue, Beltsville, MD 20705-2350

Legal Notice

Disclaimer

CONTENTS

Part I

PyPedal

PyPedal ("PyPedal") provides pedigree analysis tools for Python. This part contains all you need to know about "PyPedal" pedigrees and the functions that operate upon them.

CHAPTER

ONE

Introduction

This chapter introduces the PyPedal extensionto Python and outlines the rest of the document.

PyPedal (**Py**thon **Ped**igree An**al**ysis) is a tool for analyzing animal pedigree files. It calculates several quantitative measures of allelic and genoytpic diversity from pedigrees, including average coefficients of inbreeding and relationship, effective number of founders, and effective number of ancestors. Some qualitative checks are performed in order to catch some common mistakes, such as parents with more recent birthdates or ID numbers than their offspring. Currently, PyPedal only makes use of information on pedigree structure. Allelotypes can be assigned to founders (or read from the pedigree file) for use in genedropping to compute effective number of founder genomes, but no other measures of alleic diversity are currently supported. Some tools for non-interactive pedigree visualization are also provided.

Routines are also provided for the decomposition of A and the direct formation of A^1 with and without accounting for inbreeding. These are of academic interest rather than practical interest, but if a simple script is needed for the inversion of a reasonably-sized pedigree PyPedal is up to the task.

PyPedal is a Python ((http://www.python.org/))language module that may be called by other Python programs or used interactively from the Python interpreter. You must have Python 2.4 installed in order to use PyPedal() because PyPedal() makes use of some features found only in version 2.4. The Numarray module must be installed in order for you to use PyPedal(), and may be found at http://www.stsci.edu/resources/software_hardware/numarray. In addition, PyPedal() will make use of the following modules if they are installed:

- Graphviz ((http://www.research.att.com/sw/tools/graphviz/)) using the PyDot ((http://dkbza.org/pydot.html)) module is needed by the draw_pedigree() routine.
- matplotlib ((http://matplotlib.sourceforge.net/)) is used to draw histograms and line graphs.
- The Python Imaging Library ((http://www.pythonware.com/products/pil/)) is used to visualize numerator relationship matrices.

This document is the "official" documentation for pypedal. It is both a tutorial and the most authoritative source of information about pypedal with the exception of the source code. The tutorial material will walk you through a set of manipulations of a simple pedigree. All users of PyPedal are encouraged to follow the tutorial with a working PyPedal installation, working the examples. The best way to learn is by doing — the aim of this tutorial is to guide you along this "doing."

This manual contains:

Installing PyPedal Chapter 2 provides information on testing Python and installing PyPedal.

PyPedal Tutorial Chapter 4 provides information on testing Python and installing PyPedal.

High-Level Overview Chapter **??** gives a high-level overview of the components of the PyPedal system as a whole.

Glossary Appendix 6 gives a glossary of terms.

1.1 Implemented Features

PyPedal is currently capable of doing the following things:

- Reading pedigree files in several formats;
- Checking pedigree integrity (duplicate IDs, parents younger than offspring, etc.);
- Generating summary information such as frequency of appearance in the pedigree file;
- Computation of the numerator relationship matrix (A) from a pedigree file using the tabular method;
- Inbreeding calculations for large pedigrees using VanRaden's (1992) recursive algorithm;
- Computation of average total and average individual coefficients of inbreeding and relationship;
- Decomposition of A into T and D such that $A = TDT'$;
- Computation of the direct inverse of A (not accounting for inbreeding) using the method of Henderson (1976);
- Computation of the direct inverse of A (accounting for inbreeding) using the method of Quaas (1976);
- Storage of A and its inverse between user sessions as persistent Python objects using the pickle module to avoid unnecessary calculations;
- Computation of effective founder number using the exact algorithm of Lacy (1989);
- Computation of effective founder number using the approximate algorithm of Boichard et al. (1996);
- Computation of effective ancestor number using the algorithm of Boichard et al. (1996);
- Selection of subpedigrees containing all ancestors of an animal;
- Identification of the common relatives of two animals;
- Output to ASCII text files, including matrices, coefficients of inbreeding and relationship, and summary information;
- Reordering and renumbering of pedigree files.

1.2 Planned Features

The following features are not yet implemented in PyPedal, but will probably be added in a future release:

- Direct calculation of the inverse of A accounting for inbreeding using the method of Luo and Meuwissen;
- Calculation of theoretical effective population size;
- Calculation of actual effective population size based on the change in population average inbreeding;
- Calculation of some measure of effective family number (inspired by a post of D. Gianola's to the Animal Geneticists Discussion Group email list on 30 January 2001);
- Representation of pedigrees as an algebraic structure (i.e. graphs);
- Identification of disconnected subgroups (if any) in a pedigree;
- Fast operations on graphs;

1.3 Where to get information and code

PyPedal and its documentation are available at the author's website (sourceforge.net). The Numarray web site is: http://numpy.sourceforge.net/. The Python web site is http://www.python.org/.

1.4 Acknowledgments

PyPedal was initially written to support the author's dissertation research while at Louisiana State University, Baton Rouge (http://www.lsu.edu/). It sat fallow for some time and has recently come under active development again. This is due in part to a request from colleagues at the University of Minnesota that led to the inclusion of new functionality in PyPedal. The author wishes to thank Dr. Paul VanRaden for very helpful suggestions for improving the ability of PyPedal to handle certain computations in very large pedigrees.

Some of the text in this manual is taken verbatim from the Numarray manual.

Installing PyPedal

This chapter explains how to install and test PyPedal from either the source distribution or from the binary distribution.

Before we can begin the tutorial, we need to make sure that you can install and test Python, the Numeric or Numarray extension, and the PyPedal extension.

2.1 Testing the Python installation

The first step is to install Python if you haven't already. Python is available from the Python project page at http://sourceforge.net/projects/python/. Click on the link corresponding to your platform, and follow the instructions described there. PyPedal requires version 2.3 as a minimum. When installed, starting Python by typing python at the shell or double-clicking on the Python interpreter should give a prompt such as:

```
Python 2.3.3 (#2, Feb 17 2004, 11:45:40)
[GCC 3.3.2 (Mandrake Linux 10.0 3.3.2-6mdk)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
```

If you have problems getting Python to work, consider contacting your local support person or e-mailing python-help@python.org for help. If neither solution works, consider posting on the comp.lang.python newsgroup (details on the newsgroup/mailing list are available at http://www.python.org/psa/MailingLists.html#clp).

2.2 Testing the Numarray Python Extension Installation

The standard Python distribution does not come, as of this writing, with the numarray Python extensions installed, but your system administrator may have installed them already. To find out if your Python interpreter has numarray installed, type '`import numarray`' at the Python prompt. You'll see one of two behaviors (throughout this document user input and python interpreter output will be emphasized as shown in the block below):

```
>>> import numarray
Traceback (innermost last):
File "<stdin>", line 1, in ?
ImportError: No module named numarray
```

indicating that you don't have numarray installed, or:

```
>>> import numarray
>>> numarray.__version__
'0.9'
```

indicating that numarray is installed. If it is installed, you can skip the next section and go ahead to section 2.3. If you don't, you have to get and install the numarray extensions as described on the Numarray website at http://www.stsci.edu/resources/software_hardware/numarray.

2.3 Installing PyPedal

In order to get PyPedal, visit the official website at http://sourceforge.net/projects/pypedal. Click on the "PyPedal" release and you will be presented with a list of the available files. The files whose names end in ".tar.gz" are source code releases. The other files are binaries for a given platform (if any are available).

It is possible to get the latest sources directly from our CVS repository using the facilities described at SourceForge. Note that while every effort is made to ensure that the repository is always "good", direct use of the repository is subject to more errors than using a standard release.

2.3.1 Installing on Unix, Linux, and Mac OSX

The source distribution should be uncompressed and unpacked as follows (for example):

```
gunzip pypedal-2.0.0a12.tar.gz
tar xf pypedal-2.0.0a12.tar.gz
```

Follow the instructions in the top-level directory for compilation and installation. Note that there are options you must consider before beginning. Installation is usually as simple as:

```
python setup.py install
```

or:

```
python setupall.py install
```

There are currently no extra packages for PyPedal.

Important Tip Just like all Python modules and packages, the PyPedal module can be invoked using either the '`import PyPedal`' form, or the '`from PyPedal import ...`' form. Because most of the functions we'll talk about are in the numarray module, in this document, all of the code samples will assume that they have been preceded by a statement:

```
>>> from numarray PyPedal *
```

2.3.2 Installing on Windows

To install numarray, you need to be in an account with Administrator privileges. As a general rule, always remove (or hide) any old version of PyPedal before installing the next version.

Please note that we have **NOT** tested PyPedal on any Win-32 platforms! However, PyPedal should install and run properly on Win-32 as long as the dependencies mentioned above are satisfied.

Installation from source

1. Unpack the distribution: (NOTE: You may have to download an ”unzipping” utility)

   ```
   C:\> unzip PyPedal.zip
   C:\> cd PyPedal
   ```

2. Build it using the distutils defaults:

   ```
   C:\pyPedal> python setup.py install
   ```

   This installs PyPedal in `C:\pythonXX` where XX is the version number of your python installation, e.g. 20, 21, etc.

Installation from self-installing executable

1. Click on the executable’s icon to run the installer.
2. Click ”next” several times. I have not experimented with customizing the installation directory and don’t recommend changing any of the installation defaults. If you do and have problems, let us know.
3. Assuming everything else goes smoothly, click ”finish”.

Installation on Cygwin

No information on installing PyPedal on Cygwin is available. If you manage to get it working, let us know.

2.4 Testing the PyPedal Python Extension Installation

To find out if you have correctly installed PyPedal, type ‘`import PyPedal`’ at the Python prompt. You’ll see one of two behaviors (throughout this document user input and Python interpreter output will be emphasized as shown in the block below):

```
>>> import PyPedal
Traceback (innermost last):
File "<stdin>", line 1, in ?
ImportError: No module named PyPedal
```

indicating that you don’t have PyPedal installed, or:

```
>>> import PyPedal
>>> PyPedal.__version__
'2.0.0a1'
```

indicating that PyPedal is installed.

2.5 At the SourceForge...

The SourceForge project page for numarray is at http://sourceforge.net/projects/pyedal. On this project page you will find links to:

The PyPedal Discussion List You can subscribe to a discussion list about PyPedal using the project page at SourceForge. The list is a good place to ask questions and get help. Send mail to pyedal-discussion@lists.sourceforge.net. There is also a pypedal-discussion group that you may join.

The Web Site Click on "home page" to get to the PyPedal Home Page, which has links to documentation and other resources.

Bugs and Patches Bug tracking and patch-management facilities is provided on the SourceForge project page.

FTP Site The FTP Site contains this documentation in several formats, plus maybe some other goodies we have lying around.

CHAPTER

THREE

High-Level Overview

In this chapter, a high-level overview of PyPedal is provided, giving the reader the definitions of the key components of the system. This section defines the concepts used by the remaining sections.

3.1 The PyPedal Object Model

At the heart of PyPedal are four different types of objects. These objects combine data and the code that operate on those data into one convenient package. Although most PyPedal users will only work directly with one or two of these objects it is worthwhile to know what they are. An instance of the **NewPedigree** class stores a pedigree read from an input file as well as metadata about that pedigree. The pedigree is a Python list of **NewAnimal** objects. Information about the pedigree, such as the number and identity of founders, is contained in an instance of the **PedigreeMetadata** class.

The fourth PyPedal class, **NewAMatrix**, is used to manipulate numerator relationship matrices (NRM). When working with large pedigrees it can take a long time to compute the elements of a NRM, and having an easy way to save and restore them is quite convenient.

‘u1’

Here is an example of Python code using the NewPedigree object (`examples/new_lacy.py`):

```
import pyp_newclasses, pyp_nrm. pyp_metrics
from pyp_utils import pyp_nice_time

options = {}
options['messages'] = 'verbose'
options['renumber'] = 0
options['counter'] = 5

if __name__ == '__main__':
    print 'Starting pypedal.py at %s' % (pyp_nice_time())

# Example taken from Lacy (1989), Appendix A.
    options['pedfile'] = 'new_lacy.ped'
    options['pedformat'] = 'asd'
    options['pedname'] = 'Lacy Pedigree'
    example = pyp_newclasses.NewPedigree(options)
    example.load()
    if example.kw['messages'] == 'verbose':
        print '[INFO]: Calling pyp_metrics.effective_founders_lacy at %s' % (pyp_nice_time())
    pyp_metrics.effective_founders_lacy(example)
```

See section 2.3.1.

3.2 Pedigree Format Codes

Pedigree format codes consisting of a string of characters are used to describe the contents of a pedigree file. The codes currently recognized by PyPedal are:

- a = animal (REQUIRED)
- s = sire (REQUIRED)
- d = dam (REQUIRED)
- g = generation
- x = sex
- b = birthyear (YYYY)
- f = inbreeding
- r = breed
- n = name
- y = birthdate in ”MMDDYYYY” format
- l = alive (1) or dead (0)
- e = age
- A = alleles (two alleles separated by a non-null character)

As noted, all pedigrees must contain columns corresponding to animals, sires, and dams.

Ufuncs are covered in detail in ”Ufuncs” on page **??**.

CHAPTER

FOUR

Tutorial

This chapter provides a tutorial for PyPedal. The sample pedigree files may be found in the directory in the distribution. [1]

We are going to start the actual tutorial in this chapter. First, however, we will describe some key concepts that will help you work successfully with PyPedal. You can find a more detailed explanation of PyPedal components in chapter **??**.

4.1 A Few Important Concepts

To make the most of PyPedal you, the user, need to have a solid understanding of your dataset as well as of the PyPedal API. While Python is an object-oriented programming language, PyPedal is at heart a procedural tool. One of the exceptions to this rule is what PyPedal terms a pedigree, which is a Python list containing Animal() objects. The first step in most PyPedal analyses is to read your pedigree into PyPedal from a textfile. After that, you will spend most of your time passing your pedigree from one procedure to another. But always remember that the elements in the pedigree are objects!

4.2 A Gentle Introduction to PyPedal

For this tutorial we are going to use a sample pedigree from Hartl and Clark'a (1989) ”Principles of Population Genetics (Second Edition)” (Figure 5, p. 242). The pedigree is provided as **hartl.ped** in the distribution in the `tutorial` subdirectory, There is also an accompanying Python program, **hartl.py**.

4.2.1 The Anatomy of a Pedigree File

Obviously you need a pedigree file in order to work with PyPedal. There are a couple of things that you need to know about pedigree files and at least one thing that is helpful to know. Pedigree files must contain a format code, and the format code **must** precede the first animal record. A complete list of pedigree codes appears in section 3.2. Each animal record must appear on a separate line in the pedigree file. An animal record consists of at least an animal ID, a sire ID, and a dam ID; the IDs are separated by a delimiter, usually a comma or a space. More information may be required on a linde depending on the pedigree format used. Missing parents should be coded as '0'. Parents do not need to have their own entry in the pedigree if THEIR parents are unknown; the `preprocess()` procedure is clever enough to add the needed records automatically. Comment lines, which begin with '#', may appear anywhere in the file; they are ignored by the preprocessor.

[1] Please let me know of any additions to this tutorial that you feel would be helpful.

```
# Great tit pedigree from Hartl and Clark (1989), figure 5, p. 242.
# Used in PyPedal tutorial.
% asd
1 0 0
2 0 0
3 0 0
4 1 2
5 1 2
6 3 4
7 3 4
8 5 0
9 0 6
10 7 0
11 8 0
12 9 11
13 12 7
14 10 11
15 13 14
```

This pedigree contains fifteen animals, including three founders (animals with neither parent known), in the familiar 'animal sire dam' format.

4.2.2 The Anatomy of a Program

The **hartl.ped** program is fairly simple, but it demonstrates some of the things that you can easily do with PyPedal. Please note that while I have placed these comamnds in a file, you can also walk through the steps using the Python command line. Most of the print statements are there to provide feedback while the program is running. It is not a big deal with a small pedigree, but it is nice to know that something is happening when you throw a large pedigree at PyPedal(). I have put in line numbers for ease os reference, but if you are working along with the tutorial at the command line you should not type in the line numbers.

```
print 'Starting pypedal.py at %s' % asctime(localtime(time()))
print '\tPreprocessing pedigree at %s' % asctime(localtime(time()))
example = preprocess('hartl.ped',sepchar=' ')
example = renumber(example,'example',io='yes')
print '\tCalling set_ancestor_flag at %s' % asctime(localtime(time()))
set_ancestor_flag(example,'example',io='yes')
print '\tCollecting pedigree metadata at %s' % asctime(localtime(time()))
example_meta = Pedigree(example,'example.ped','example_meta')
print '\tCalling a_effective_founders_lacy() at %s' % asctime(localtime(time()))
a_effective_founders_lacy(example,filetag='example')
print '\tCalling a_effective_founders_boichard() at %s' % asctime(localtime(time()))
a_effective_founders_boichard(example,filetag='example')
print '\tCalling a_effective_ancestors_definite() at %s' % asctime(localtime(time()))
a_effective_ancestors_definite(example,filetag='example')
print '\tCalling a_effective_ancestors_indefinite() at %s' % asctime(localtime(time()))
a_effective_ancestors_indefinite(example,filetag='example',n=10)
print '\tCalling related_animals() at %s' % asctime(localtime(time()))
list_a = related_animals(example[14].animalID,example)
print list_a
print '\tCalling related_animals() at %s' % asctime(localtime(time()))
list_b = related_animals(example[9].animalID,example)
print list_b
print '\tCalling common_ancestors() at %s' % asctime(localtime(time()))
list_r = common_ancestors(example[14].animalID,example[9].animalID,example)
print list_r
print 'Stopping pypedal.py at %s' % asctime(localtime(time()))
```

4.2.3 Reading PyPedal Output

```
Starting pypedal.py at Mon Apr 19 15:28:53 2004
        Preprocessing pedigree at Mon Apr 19 15:28:53 2004
        Calling set_ancestor_flag at Mon Apr 19 15:28:53 2004
        Collecting pedigree metadata at Mon Apr 19 15:28:53 2004
PEDIGREE example_meta (example.ped)
        Records:                15
        Unique Sires:           9
        Unique Dams:            7
        Unique Gens:            1
        Unique Years:           1
        Unique Founders:        3
        Pedigree Code:          asd
        Calling inbreeding() at Mon Apr 19 15:28:53 2004
{1: 0.0, 2: 0.0, 3: 0.0, 4: 0.0, 5: 0.0, 6: 0.0, 7: 0.0, 8: 0.0, 9: 0.0, 10: 0.0, 11: 0.0, 12: 0.0156
        Calling a_effective_founders_lacy() at Mon Apr 19 15:28:53 2004
============================================================
animals:        15
founders:       3
descendants:    12
f_e:            7.205
============================================================
        Calling a_effective_founders_boichard() at Mon Apr 19 15:28:53 2004
============================================================
animals:        15
founders:       3
descendants:    12
f_e:            5.856
============================================================
        Calling a_effective_ancestors_definite() at Mon Apr 19 15:28:53 2004
============================================================
animals:        15
founders:       0
descendants:    15
f_a:            0.000
============================================================
        Calling a_effective_ancestors_indefinite() at Mon Apr 19 15:28:53 2004
------------------------------------------------------------
WARNING: (pyp_metrics/a_effective_ancestors_indefinite()): Setting n (10) to be equal to the actual n
============================================================
animals:        15
founders:       0
descendants:    15
f_l:            0.000
f_u:            1.000
============================================================
        Calling related_animals() at Mon Apr 19 15:28:53 2004
[15, 13, 7, 3, 4, 1, 2, 12, 9, 6, 11, 8, 5, 14, 10]
        Calling related_animals() at Mon Apr 19 15:28:53 2004
[10, 7, 3, 4, 1, 2]
        Calling common_ancestors() at Mon Apr 19 15:28:53 2004
[1, 2, 3, 4, 7, 10]
Stopping pypedal.py at Mon Apr 19 15:28:53 2004
```

CHAPTER

FIVE

API

This chapter provides an overview of the PyPedal Application Programming Interface (API). More simply, it is a reference to the various classes, methods, and procedures that make up the PyPedal module.

5.1 Some Background

Erm...

5.2 pyp_classes

pyp_classes contains two base classes that are used by PyPedal, the Animal() class and the Pedigree() class. What most PyPedal routines recognize as a pedigree is actually just a Python list of Animal() objects. An instance of a Pedigree() object is a collection of METADATA about a list of Animals(). I know that this is confusing, and it is going to change by the time that PyPedal 2.0.0 final is released.

Module Contents

Animal(animalID, sireID, damID, gen='0', by=1900, sex='u', fa=0., name='u', alleles=['', ''], breed='u', age=-999, alive=-999) (c
]

The Animal() class is holds animals records read from a pedigree file.

For more information about this class, see *The Animal Class* .

Pedigree(myped, inputfile, name, pedcode='asd', reord=0, renum=0, debug=0) (class) [#]

The Pedigree() class stores metadata about pedigrees.

For more information about this class, see *The Pedigree Class* .

The Animal Class

Animal(animalID, sireID, damID, gen='0', by=1900, sex='u', fa=0., name='u', alleles=['', ''], breed='u', age=-999, alive=-999) (c
]

The Animal() class is holds animals records read from a pedigree file.

__init__(animalID, sireID, damID, gen='0', by=1900, sex='u', fa=0., name='u', alleles=['', ''], breed='u', age=-999, alive=-999)
]

__init__() initializes an Animal() object.

self Reference to the current Animal() object

animalID Animal ID number

sireID Sire ID number

damID Dam ID number

gen Generation to which the animal belongs

by Birthyear of the animal

sex Sex of the animal (m—f—u)

fa Coefficient of inbreeding of the animal

name Name of animal

alleles A two-element array of strings, which represent allelotypes.

breed Breed of animal

age Age of animal

alive Status of animal (alive or dead)

Returns: An instance of an Animal() object populated with data

pad_id() &rArr; integer [#]

pad_id() takes an Animal ID, pads it to fifteen digits, and prepends the birthyear (or 1950 if the birth year is unknown). The order of elements is: birthyear, animalID, count of zeros, zeros.

self Reference to the current Animal() object

Returns: A padded ID number that is supposed to be unique across animals

printme() [#]

printme() prints a summary of the data stored in the Animal() object.

self Reference to the current Animal() object

stringme() [#]

stringme() returns a summary of the data stored in the Animal() object as a string.

self Reference to the current Animal() object

trap() [#]

trap() checks for common errors in Animal() objects

self Reference to the current Animal() object

The Pedigree Class

Pedigree(myped, inputfile, name, pedcode='asd', reord=0, renum=0, debug=0) (class) [#]

The Pedigree() class stores metadata about pedigrees. Hopefully this will help improve performance in some procedures, as well as provide some useful summary data.

__init__(myped, inputfile, name, pedcode='asd', reord=0, renum=0, debug=0) &rArr; object [#]

__init__() initializes a Pedigree metata object.

self Reference to the current Pedigree() object

myped A PyPedal pedigree

inputfile The name of the file from which the pedigree was loaded

name The name assigned to the PyPedal pedigree

pedcode The format code for the PyPedal pedigree

reord Flag indicating whether or not the pedigree is reordered (0—1)

renum Flag indicating whether or not the pedigree is renumbered (0—1)

Returns: An instance of a Pedigree() object populated with data

fileme() [#]

fileme() writes the metada stored in the Pedigree() object to disc.

self Reference to the current Pedigree() object

nud() &rArr; integer-and-list [#]

nud() returns the number of unique dams in the pedigree along with a list of the dams

self Reference to the current Pedigree() object

Returns: The number of unique dams in the pedigree and a list of those dams

nuf() &rArr; integer-and-list [#]

nuf() returns the number of unique founders in the pedigree along with a list of the founders

self Reference to the current Pedigree() object

Returns: The number of unique founders in the pedigree and a list of those founders

nug() &rArr; integer-and-list [#]

nug() returns the number of unique generations in the pedigree along with a list of the generations

self Reference to the current Pedigree() object

Returns: The number of unique generations in the pedigree and a list of those generations

nus() &rArr; integer-and-list [#]

nus() returns the number of unique sires in the pedigree along with a list of the sires

self Reference to the current Pedigree() object

Returns: The number of unique sires in the pedigree and a list of those sires

nuy() &rArr; integer-and-list [#]

nuy() returns the number of unique birthyears in the pedigree along with a list of the birthyears

self Reference to the current Pedigree() object

Returns: The number of unique birthyears in the pedigree and a list of those birthyears

printme() [#]

printme() prints a summary of the metadata stored in the Pedigree() object.

self Reference to the current Pedigree() object

stringme() [#]

stringme() returns a summary of the metadata stored in the pedigree as a string.

self Reference to the current Pedigree() object

5.3 pyp_demog

pyp_demog contains a set of procedures for demographic calculations on the population describe in a pedigree.

age_distribution(myped, verbose=1, sex=1) [#]

age_distribution() computes histograms of the age distribution of males and females in the population. You can also stratify by sex to get individual histograms.

myped A PyPedal pedigree.

verbose Print or suppress output. (???)

sex A flag which determines whether or not to stratify by sex.

founders_by_year(pedobj) &rArr; dictionary [#]

founders_by_year() returns a dictionary containing the number of founders in each birthyear.

pedobj A PyPedal pedigree object.

Returns: dict A dictionary containing entries for each sex/gender code defined in the global SEX_CODE_MAP.

set_age_units(units='year') &rArr; None [#]

set_age_units() defines a global variable, BASE_DEMOGRAPHIC_UNIT.

units The base unit for age computations ('year'—'month'—'day').

Returns: None

set_base_year(year=1950) &rArr; None [#]

set_base_year() defines a global variable, BASE_DEMOGRAPHIC_YEAR.

year The year to be used as a base for computing ages.

Returns: None

sex_ratio(myped, verbose=1) &rArr; dictionary [#]

sex_ratio() returns a dictionary containing the proportion of males and females in the population.

year The year to be used as a base for computing ages.

Returns: dict A dictionary containing entries for each sex/gender code defined in the global SEX_CODE_MAP.

5.4 pyp_graphics

pyp_graphics contains for working with graphics in PyPedal, such as creating directed graphs from pedigrees using PyDot and visualizing relationship matrices using Rick Muller's spy and pcolor routines (http://aspn.activestate.com/ASPN/Cookbook/Python/). The Python Imaging Library (http://www.pythonware.com/products/pil/), matplotlib (http://matplotlib.sourceforge.net/), Graphviz (http://www.graphviz.org/), and pydot (http://dkbza.org/pydot.html) are required by one or more routines in this module. They ARE NOT distributed with PyPedal and must be installed by the end-user! Note that the matplotlib functionality in PyPedal requires only the Agg backend, which means that you do not have to install GTK/PyGTK or WxWidgets/PyWxWidgets just to use PyPedal. Please consult the sites above for licensing and installation information.

draw_pedigree(myped, gfilename='pedigree', gtitle='My_Pedigree', gformat='jpg', gsize='f', gdot='1') &rArr; integer [#]

draw_pedigree() uses the pydot bindings to the graphviz library – if they are available on your system – to produce a directed graph of your pedigree with paths of inheritance as edges and animals as nodes. If there is more than one generation in the pedigree as determind by the "gen" attributes of the anumals in the pedigree, draw_pedigree() will use subgraphs to try and group animals in the same generation together in the drawing.

myped A PyPedal pedigree object.

gfilename The name of the file to which the pedigree should be drawn

gtitle The title of the graph.

gsize The size of the graph: 'f': full-size, 'l': letter-sized page.

gdot Whether or not to write the dot code for the pedigree graph to a file (can produce large files).

Returns: A 1 for success and a 0 for failure.

plot_founders_by_year(pedobj, gfilename='founders_by_year', gtitle='Founders by Birthyear') &rArr; integer [#]

founders_by_year() uses matplotlib – if available on your system – to produce a bar graph of the number (count) of founders in each birthyear.

pedobj A PyPedal pedigree object.

gfilename The name of the file to which the pedigree should be drawn

gtitle The title of the graph.

Returns: A 1 for success and a 0 for failure.

plot_founders_pct_by_year(pedobj, gfilename='founders_pct_by_year', gtitle='Founders by Birthyear') &rArr; integer [#]

founders_pct_by_year() uses matplotlib – if available on your system – to produce a line graph of the frequency (percentage) of founders in each birthyear.

pedobj A PyPedal pedigree object.

gfilename The name of the file to which the pedigree should be drawn

gtitle The title of the graph.

Returns: A 1 for success and a 0 for failure.

rmuller_get_color(a, cmin, cmax) &rArr; integer [#]

rmuller_get_color() Converts a float value to one of a continuous range of colors using recipe 9.10 from the Python Cookbook.

a Float value to convert to a color.

cmin Minimum value in array (?).

cmax Maximum value in array (?).

Returns: An RGB triplet.

rmuller_pcolor_matrix_pil(A, fname='tmp.png', do_outline=0, height=300, width=300) &rArr; lists [#]

rmuller_pcolor_matrix_pil() implements a matlab-like 'pcolor' function to display the large elements of a matrix in pseudocolor using the Python Imaging Library.

A Input Numpy matrix (such as a numerator relationship matrix).

fname Output filename to which to dump the graphics (default 'tmp.png')

do_outline Whether or not to print an outline around the block (default 0)

height The height of the image (default 300)

width The width of the image (default 300)

Returns: A list of Animal() objects; a pedigree metadata object.

rmuller_spy_matrix_pil(A, fname='tmp.png', cutoff=0.1, do_outline=0, height=300, width=300) &rArr; lists [#]

rmuller_spy_matrix_pil() implements a matlab-like 'spy' function to display the sparsity of a matrix using the Python Imaging Library.

A Input Numpy matrix (such as a numerator relationship matrix).

fname Output filename to which to dump the graphics (default 'tmp.png')

cutoff Threshold value for printing an element (default 0.1)

do_outline Whether or not to print an outline around the block (default 0)

height The height of the image (default 300)

width The width of the image (default 300)

Returns: A list of Animal() objects; a pedigree metadata object.

5.5 pyp_io

pyp_io contains several procedures for writing structures to and reading them from disc (e.g. using pickle() to store and retrieve A and A-inverse). It also includes a set of functions used to render strings as HTML or plaintext for use in generating output files.

Module Contents

a_inverse_from_file(inputfile) &rArr; matrix [#]

a_inverse_from_file() uses the Python pickle system for persistent objects to read the inverse of a relationship matrix from a file.

inputfile The name of the input file.

Returns: The inverse of a numerator relationship matrix.

a_inverse_to_file(myped, filetag='_pickled_', ainv='') [#]

a_inverse_to_file() uses the Python pickle system for persistent objects to write the inverse of a relationship matrix to a file.

myped A PyPedal pedigree object.

filetag A descriptor prepended to output file names.

a_matrix_from_file(inputfile) &rArr; matrix [#]

a_matrix_from_file() uses the Python pickle system for persistent objects to read a relationship matrix from a file.

inputfile The name of the input file.

Returns: A numerator relationship matrix.

a_matrix_from_text_file() [#]

a_matrix_from_text_file() is a placeholder. The a_matrix() procedure currently writes A to a text file after A is formed and before the function returns. It would be handy to have a nice procedure to suck that back into an object.

a_matrix_to_file(myped, filetag='_pickled_', a='') [#]

a_matrix_to_file() uses the Python pickle system for persistent objects to write a relationship matrix to a file. This works well for small pedigrees, not so well for large pedigrees; large pedigrees will eat some serious disc space.

myped A PyPedal pedigree object.

filetag A descriptor prepended to output file names.

save Flag to indicate whether or not the relationship matrix is written to a file.

dissertation_pedigree_to_file(myped, filetag='_diss') [#]

dissertation_pedigree_to_file() takes a pedigree in 'asdxfg' format and writes is to a file.

myped A PyPedal pedigree object.

filetag A descriptor prepended to output file names.

dissertation_pedigree_to_pedig_format(myped, filetag='_diss') [#]

dissertation_pedigree_to_pedig_format() takes a pedigree in 'asdbxfg' format, formats it into the form used by Didier Boichard's 'pedig' suite of programs, and writes it to a file.

myped A PyPedal pedigree object.

filetag A descriptor prepended to output file names.

dissertation_pedigree_to_pedig_format_mask(myped, filetag='_diss_mask') [#]

dissertation_pedigree_to_pedig_format_mask() Takes a pedigree in 'asdbxfg' format, formats it into the form used by Didier Boichard's 'pedig' suite of programs, and writes it to a file. THIS FUNCTION MASKS THE GENERATION ID WITH A FAKE BIRTH YEAR AND WRITES THE FAKE BIRTH YEAR TO THE FILE INSTEAD OF THE TRUE BIRTH YEAR. THIS IS AN ATTEMPT TO FOOL PEDIG TO GET f_e, f_a et al. BY GENERATION.

myped A PyPedal pedigree object.

filetag A descriptor prepended to output file names.

dissertation_pedigree_to_pedig_interest_format(myped, filetag='_diss') [#]

dissertation_pedigree_to_pedig_interest_format() takes a pedigree in 'asdbxfg' format, formats it into the form used by Didier Boichard's parente program for the studied individuals file.

myped A PyPedal pedigree object.

filetag A descriptor prepended to output file names.

id_map_from_file(inputfile) &rArr; dictionary [#]

id_map_from_file() reads an ID map from the file generated by pyp_utils/renumber() into a dictionary. There is a VERY similar function, pyp_utils/load_id_map(), that is preferred because it is more robust that this procedure.

inputfile The name of the file from which the ID map should be read.

Returns: A dictionary whose keys are renumbered IDs and whose values are original IDs.

pyp_file_footer(ofhandle, caller="Unknown PyPedal routine") [#]

pyp_file_footer()

ofhandle A Python file handle.

caller A string indicating the name of the calling routine.

Returns: None

pyp_file_header(ofhandle, caller=”Unknown PyPedal routine”) [#]

pyp_file_header()

ofhandle A Python file handle.

caller A string indicating the name of the calling routine.

Returns: None

renderTitle(title_string, title_level=”1”) [#]

renderTitle() ... Produced HTML output by default.

5.6 pyp_metrics

pyp_metrics contains a set of procedures for calculating metrics on PyPedal pedigree objects. These metrics include coefficients of inbreeding and relationship as well as effective founder number, effective population size, and effective ancestor number.

Module Contents

a_coefficients(myped, filetag=’_coefficients_’, a=”, method=’nrm’) [#]

a_coefficients() writes population average coefficients of inbreeding and relationship to a file, as well as individual animal IDs and coefficients of inbreeding. Some pedigrees are too large for fast_a_matrix() or fast_a_matrix_r() – an array that large cannot be allocated due to memory restrictions – and will result in a value of -999.9 for all outputs.

myped A PyPedal pedigree object.

filetag A descriptor prepended to output file names.

a A numerator relationship matrix (optional).

method If no relationship matrix is passed, determines which procedure should be called to build one (nrm—frm).

a_effective_ancestors(myped, filetag=’_f_a_’, a=”, gen=”, n=25) &rArr; float [#]

a_effective_ancestors() calls either a_effective_ancestors_definite() or a_effective_ancestors_indefinite() based on pedigree size using an arbitrarily-assigned threshold of 1,000 animals. For small pedigrees (N <= 1,000) the exact computation is performed. For larger pedigrees, an approximate computation is carried out based on inexact lower and upper bounds of f_a (see Boichard et al. (1996) pp.9-10). If no number of ancestors is specified in the call to a_effective_ancestors() and the indef- inite routine is used, a default of 25 is used. Boichard’s algorithms require information about the GENERATION of animals. If you do not provide an input pedigree with generations things may not work. By default the most recent generation – the generation with the largest generation ID – will be used as the reference population.

myped A PyPedal pedigree object.

filetag A descriptor prepended to output file names.

a A numerator relationship matrix (optional).

gen Generation of interest.

gen Number of ancestors to use with the indefinite routine.

Returns: The effective founder number.

a_effective_ancestors_definite(myped, filetag='_f_a_definite_', a='', gen='') &rArr; float [#]

a_effective_ancestors_definite() uses the algorithm in Appendix B of Boichard et al. (1996) to compute the effective ancestor number for a myped pedigree. NOTE: One problem here is that if you pass a pedigree WITHOUT generations and error is not thrown. You simply end up wth a list of generations that contains the default value for Animal() objects, 0. Boichard's algorithm requires information about the GENERATION of animals. If you do not provide an input pedigree with generations things may not work. By default the most recent generation – the generation with the largest generation ID – will be used as the reference population.

myped A PyPedal pedigree object.

filetag A descriptor prepended to output file names.

a A numerator relationship matrix (optional).

gen Generation of interest.

Returns: The effective founder number.

a_effective_ancestors_indefinite(myped, filetag='_f_a_definite_', a='', gen='', n=25) &rArr; float [#]

a_effective_ancestors_indefinite() uses the approach outlined on pages 9 and 10 of Boichard et al. (1996) to compute approximate upper and lower bounds for f_a. This is much more tractable for large pedigrees than the exact computation provided in a_effective_ancestors_definite(). NOTE: One problem here is that if you pass a pedigree WITHOUT generations and error is not thrown. You simply end up wth a list of generations that contains the default value for Animal() objects, 0. NOTE: If you pass a value of n that is greater than the actual number of ancestors in the pedigree then strange things happen. As a stop-gap, a_effective_ancestors_indefinite() will detect that case and replace n with the number of founders - 1. Boichard's algorithm requires information about the GENERATION of animals. If you do not provide an input pedigree with generations things may not work. By default the most recent generation – the generation with the largest generation ID – will be used as the reference population.

myped A PyPedal pedigree object.

filetag A descriptor prepended to output file names.

a A numerator relationship matrix (optional).

gen Generation of interest.

Returns: The effective founder number.

a_effective_founders_boichard(myped, filetag='_f_e_boichard_', a='', gen='') &rArr; float [#]

a_effective_founders_boichard() uses the algorithm in Appendix A of Boichard et al. (1996) to compute the effective founder number for myped. Note that results from this function will not necessarily match those from a_effective_founders_lacy(). Boichard's algorithm requires information about the GENERATION of animals. If you do not provide an input pedigree with generations things may not work. By default the most recent generation – the generation with the largest generation ID – will be used as the reference population.

myped A PyPedal pedigree object.

filetag A descriptor prepended to output file names.

a A numerator relationship matrix (optional).

gen Generation of interest.

Returns: The effective founder number.

a_effective_founders_lacy(myped, filetag='_f_e_lacy_', a='') &rArr; float [#]

a_effective_founders_lacy() calculates the number of effective founders in a pedigree using the exact method of Lacy.

myped A PyPedal pedigree object.

filetag A descriptor prepended to output file names.

a A numerator relationship matrix (optional).

Returns: The effective founder number.

common_ancestors(anim_a, anim_b, myped, filetag='_steps_') &rArr; list [#]

common_ancestors() returns a list of the ancestors that two animals share in common.

anim_a The renumbered ID of the first animal, a.

anim_b The renumbered ID of the second animal, b.

myped A PyPedal pedigree object.

filetag A descriptor prepended to output file names.

Returns: A list of animals related to anim_a AND anim_b

descendants(anid, myped, _desc) &rArr; list [#]

descendants() uses pedigree metadata to walk a pedigree and return a list of all of the descendants of a given animal.

anid An animal ID

myped A Python list of PyPedal Animal() objects.

_desc A Python dictionary of descendants of animal anid.

Returns: A list of descendants of anid.

effective_founder_genomes(myped, filetag='_gene_drop_', rounds=10, verbose=0, quiet=1) &rArr; float [#]

effective_founder_genomes() simulates the random segregation of founder alleles through a pedigree. At present only two alleles are simulated for each founder. Summary statistics are computed on the most recent generation.

myped A PyPedal pedigree object.

filetag A descriptor prepended to output file names.

rounds The number of times to simulate segregation through the entire pedigree.

verbose A flag to indicate whether or not diagnostic/debugging information is printed.

Returns: The effective number of founder genomes over based on 'rounds' gene-drop simulations.

effective_founders_lacy(pedobj) &rArr; float [#]

effective_founders_lacy() calculates the number of effective founders in a pedigree using the exact method of Lacy. This version of the routine a_effective_founders_lacy() is designed to work with larger pedigrees as it forms "familywise" relationship matrices rather than a "populationwise" relationship matrix.

myped A PyPedal pedigree object.

filetag A descriptor prepended to output file names.

a A numerator relationship matrix (optional).

Returns: The effective founder number.

fast_a_coefficients(myped, filetag='_coefficients_', a=", method='nrm', debug=0) [#]

a_fast_coefficients() writes population average coefficients of inbreeding and relationship to a file, as well as individual animal IDs and coefficients of inbreeding.

myped A PyPedal pedigree object.

filetag A descriptor prepended to output file names.

a A numerator relationship matrix (optional).

method If no relationship matrix is passed, determines which procedure should be called to build one (nrm—frm).

founder_descendants(pedobj) &rArr; dictionary [#]

founder_descendants() returns a dictionary containing a list of descendants of each founder in the pedigree.

pedojb An instance of a PyPedal NewPedigree object.

generation_lengths(myped, filetag='_generation_lengths_', debug=0, quiet=0, units='y') &rArr; dictionary [#]

generation_lengths() computes the average age of parents at the time of birth of their first offspring. This is implies that selection decisions are made at the time of birth of of the first offspring. Average ages are computed for each of four paths: sire-son, sire-daughter, dam-son, and dam-daughter. An overall mean is computed, as well. IT IS IMPORTANT to note that if you DO NOT provide birthyears in your pedigree file that the returned dictionary will contain only zeroes! This is because when no birthyer is provided a default value (1900) is assigned to all animals in the pedigree.

myped A PyPedal pedigree object.

filetag A descriptor prepended to output file names.

rounds The number of times to simulate segregation through the entire pedigree.

debug A flag to indicate whether or not diagnostic/debugging information is printed.

units A character indicating the units in which the generation lengths should be returned.

Returns: A dictionary containing the five average ages.

generation_lengths_all(myped, filetag='_generation_lengths_', debug=0, quiet=0, units='y') &rArr; dictionary [#]

generation_lengths_all() computes the average age of parents at the time of birth of their offspring. The computation is made using birth years for all known offspring of sires and dams, which implies discrete generations. Average ages are computed for each of four paths: sire-son, sire-daughter, dam-son, and dam-daughter. An overall mean is computed, as well. IT IS IMPORTANT to note that if you DO NOT provide birthyears in your pedigree file that the returned dictionary will contain only zeroes! This is because when no birthyer is provided a default value (1900) is assigned to all animals in the pedigree.

myped A PyPedal pedigree object.

filetag A descriptor prepended to output file names.

rounds The number of times to simulate segregation through the entire pedigree.

debug A flag to indicate whether or not diagnostic/debugging information is printed.

units A character indicating the units in which the generation lengths should be returned.

Returns: A dictionary containing the five average ages.

mating_coi(anim_a, anim_b, myped, filetag='_mating_coi_') &rArr; float [#]

mating_coi() returns the coefficient of inbreeding of offspring of a mating between two animals, anim_a and anim_b.

anim_a The renumbered ID of an animal, a.

anim_b The renumbered ID of an animal, b.

myped A PyPedal pedigree object.

filetag A descriptor prepended to output file names.

Returns: The coefficient of relationship of anim_a and anim_b

min_max_f(myped, filetag='_min_max_f_', a='', n=10) &rArr; list [#]

min_max_f() takes a pedigree and returns a list of the individuals with the n largest and n smallest coefficients of inbreeding.

myped A PyPedal pedigree object.

filetag A descriptor prepended to output file names.

a A numerator relationship matrix (optional).

n An integer (optional, default is 10).

Returns: A list of the individuals with the n largest and the n smallest CoI in the pedigree.

num_equiv_gens(myped, filetag='_num_traced_gen_', debug=0, quiet=0) &rArr; dictionary [#]

num_equiv_gens() computes the number of equivalent generations as the sum of (1/2)ˆn, where n is the number of generations separating an individual and each of its known ancestors.

myped A PyPedal pedigree object.

filetag A descriptor prepended to output file names.

rounds The number of times to simulate segregation through the entire pedigree.

debug A flag to indicate whether or not diagnostic/debugging information is printed.

Returns: A dictionary containing the five average ages.

num_traced_gens(myped, filetag='_num_traced_gen_', debug=0, quiet=0) &rArr; dictionary [#]

num_traced_gens() is computed as the number of generations separating offspring from the oldest known ancestor in in each selection path. Ancestors with unknown parents are assigned to generation 0. See: Valera, M., Molina, A., Gutierrez, J. P., Gomez, J., and Goyache, F. 2004. Pedigree analysis in the Andalusian horse: population structure, genetic variability and the influence of the Carthusian strain. Livestock Production Science. (Article in Press).

myped A PyPedal pedigree object.

filetag A descriptor prepended to output file names.

rounds The number of times to simulate segregation through the entire pedigree.

debug A flag to indicate whether or not diagnostic/debugging information is printed.

Returns: A dictionary containing the five average ages.

partial_inbreeding(myped, filetag='_num_traced_gen_', debug=0, quiet=0) &rArr; dictionary [#]

partial_inbreeding() computes the number of equivalent generations as the sum of (1/2)ˆn, where n is the number of generations separating an individual and each of its known ancestors.

myped A PyPedal pedigree object.

filetag A descriptor prepended to output file names.

rounds The number of times to simulate segregation through the entire pedigree.

debug A flag to indicate whether or not diagnostic/debugging information is printed.

Returns: A dictionary containing the five average ages.

pedigree_completeness(myped, filetag='_pedigree_completeness_', gens=4, verbose=1, debug=0) [#]

pedigree_completeness() computes the proportion of known ancestors in the pedigree of each animal in the population for a user-determined number of generations. Also, the mean pedcomps for all animals and for all animals that are not founders are computed as summary statistics.

myped A PyPedal pedigree object.

filetag A descriptor prepended to output file names.

gens The number of generations the pedigree should be traced for completeness.

verbose Request (1) or suppress (0) output (1 is default).

related_animals(anim_a, myped, filetag='_related_') &rArr; list [#]

related_animals() returns a list of the ancestors of an animal.

anim_a The renumbered ID of an animal, a.

myped A PyPedal pedigree object.

filetag A descriptor prepended to output file names.

Returns: A list of animals related to anim_a

relationship(anim_a, anim_b, myped, filetag='_relationship_') &rArr; float [#]

relationship() returns the coefficient of relationship for two animals, anim_a and anim_b.

anim_a The renumbered ID of an animal, a.

anim_b The renumbered ID of an animal, b.

myped A PyPedal pedigree object.

filetag A descriptor prepended to output file names.

Returns: The coefficient of relationship of anim_a and anim_b

theoretical_ne_from_metadata(metaped, filetag='_ne_from_metadata_') [#]

theoretical_ne_from_metadata() computes the theoretical effective population size based on the number of sires and dams contained in a pedigree metadata object. Writes results to an output file.

metaped A PyPedal pedigree metadata object.

filetag A descriptor prepended to output file names.

5.7 pyp_newclasses

pyp_newclasses contains the new class structure that will be a part of PyPedal 2.0.0Final. It includes a master class to which most of the computational routines will be bound as methods, a NewAnimal() class, and a PedigreeMetadata() class.

Module Contents

NewAMatrix(kw) (class) [#]

NewAMatrix provides an instance of a numerator relationship matrix as a Numarray array of floats with some convenience methods.

For more information about this class, see *The NewAMatrix Class* .

NewAnimal(locations, data, mykw) (class) [#]

The NewAnimal() class is holds animals records read from a pedigree file.

For more information about this class, see *The NewAnimal Class* .

NewPedigree(kw) (class) [#]

The NewPedigree class is the main data structure for PyP 2.0.0Final.

For more information about this class, see *The NewPedigree Class* .

PedigreeMetadata(myped, kw) (class) [#]

The PedigreeMetadata() class stores metadata about pedigrees.

For more information about this class, see *The PedigreeMetadata Class* .

The NewAMatrix Class

NewAMatrix(kw) (class) [#]

NewAMatrix provides an instance of a numerator relationship matrix as a Numarray array of floats with some convenience methods. The idea here is to provide a wrapper around a NRM so that it is easier to work with. For large pedigrees it can take a long time to compute the elements of A, so there is real value in providing an easy way to save and retrieve a NRM once it has been formed.

fast_a_matrix(pedigree) &rArr; integer [#]

fast_a_matrix() calls pyp_nrm/fast_a_matrix() to form a NRM from a pedigree.

pedigree The pedigree used to form the NRM.

Returns: A NRM on success, 0 on failure.

fast_a_matrix_r(pedigree) &rArr; integer [#]

fast_a_matrix_r() calls pyp_nrm/fast_a_matrix_r() to form a NRM from a pedigree.

pedigree The pedigree used to form the NRM.

Returns: A NRM on success, 0 on failure.

info() &rArr; None [#]

info() uses the info() method of Numarray arrays to dump some information about the NRM. This is of use predominantly for debugging.

None

Returns: None

load(nrm_filename) &rArr; integer [#]

load() uses the Numarray Array Function “fromfile()” to load an array from a binary file. If the load is successful, self.nrm contains the matrix.

nrm_filename The file from which the matrix should be read.

Returns: A load status indicator (0: failed, 1: success).

save(nrm_filename) &rArr; integer [#]

save() uses the Numarray method “tofile()” to save an array to a binary file.

nrm_filename The file to which the matrix should be written.

Returns: A save status indicator (0: failed, 1: success).

The NewAnimal Class

NewAnimal(locations, data, mykw) (class) [#]

The NewAnimal() class is holds animals records read from a pedigree file.

__init__(locations, data, mykw) &rArr; object [#]

__init__() initializes a NewAnimal() object.

locations A dictionary containing the locations of variables in the input line.

data The line of input read from the pedigree file.

Returns: An instance of a NewAnimal() object populated with data

pad_id() &rArr; integer [#]

pad_id() takes an Animal ID, pads it to fifteen digits, and prepends the birthyear (or 1950 if the birth year is unknown). The order of elements is: birthyear, animalID, count of zeros, zeros.

self Reference to the current Animal() object

Returns: A padded ID number that is supposed to be unique across animals

printme() [#]

printme() prints a summary of the data stored in the Animal() object.

self Reference to the current Animal() object

stringme() [#]

stringme() returns a summary of the data stored in the Animal() object as a string.

self Reference to the current Animal() object

trap() [#]

trap() checks for common errors in Animal() objects

self Reference to the current Animal() object

The NewPedigree Class

NewPedigree(kw) (class) [#]

The NewPedigree class is the main data structure for PyP 2.0.0Final.

load() &rArr; None [#]

load() wraps several processes useful for loading and preparing a pedigree for use in an analysis, including reading the animals into a list of animal objects, forming lists of sires and dams, checking for common errors, setting ancestor flags, and renumbering the pedigree.

renum Flag to indicate whether or not the pedigree is to be renumbered.

alleles Flag to indicate whether or not pyp_metrics/effective_founder_genomes() should be called for a single round to assign alleles.

Returns: None

preprocess() &rArr; None [#]

preprocess() processes a pedigree file, which includes reading the animals into a list of animal objects, forming lists of sires and dams, and checking for common errors.

None

Returns: None

renumber() &rArr; None [#]

renumber() updates the ID map after a pedigree has been renumbered so that all references are to renumbered rather than original IDs.

None

Returns: None

save(filename=”, outformat=’o’, idformat=’o’) &rArr; integer [#]

save() writes a PyPedal pedigree to a user-specified file. The saved pedigree includes all fields recognized by PyPedal, not just the original fields read from the input pedigree file.

filename The file to which the pedigree should be written.

outformat The format in which the pedigree should be written: 'o' for original (as read) and 'l' for long version (all available variables).

idformat Write 'o' (original) or 'r' (renumbered) animal, sire, and dam IDs.

Returns: A save status indicator (0: failed, 1: success)

updateidmap() &rArr; None [#]

updateidmap() updates the ID map after a pedigree has been renumbered so that all references are to renumbered rather than original IDs.

None

Returns: None

The PedigreeMetadata Class

PedigreeMetadata(myped, kw) (class) [#]

The PedigreeMetadata() class stores metadata about pedigrees. Hopefully this will help improve performance in some procedures, as well as provide some useful summary data.

__init__(myped, kw) &rArr; object [#]

__init__() initializes a PedigreeMetadata object.

self Reference to the current Pedigree() object

myped A PyPedal pedigree.

kw A dictionary of options.

Returns: An instance of a Pedigree() object populated with data

fileme() [#]

fileme() writes the metada stored in the Pedigree() object to disc.

self Reference to the current Pedigree() object

nud() &rArr; integer-and-list [#]

nud() returns the number of unique dams in the pedigree along with a list of the dams

self Reference to the current Pedigree() object

Returns: The number of unique dams in the pedigree and a list of those dams

nuf() &rArr; integer-and-list [#]

nuf() returns the number of unique founders in the pedigree along with a list of the founders

self Reference to the current Pedigree() object

Returns: The number of unique founders in the pedigree and a list of those founders

nug() &rArr; integer-and-list [#]

nug() returns the number of unique generations in the pedigree along with a list of the generations

self Reference to the current Pedigree() object

Returns: The number of unique generations in the pedigree and a list of those generations

nus() &rArr; integer-and-list [#]

nus() returns the number of unique sires in the pedigree along with a list of the sires

self Reference to the current Pedigree() object

Returns: The number of unique sires in the pedigree and a list of those sires

nuy() &rArr; integer-and-list [#]

nuy() returns the number of unique birthyears in the pedigree along with a list of the birthyears

self Reference to the current Pedigree() object

Returns: The number of unique birthyears in the pedigree and a list of those birthyears

printme() [#]

printme() prints a summary of the metadata stored in the Pedigree() object.

self Reference to the current Pedigree() object

stringme() [#]

stringme() returns a summary of the metadata stored in the pedigree as a string.

self Reference to the current Pedigree() object

5.8 pyp_nrm

pyp_nrm contains several procedures for computing numerator relationship matrices and for performing operations on those matrices. It also contains routines for computing CoI on large pedigrees using the recursive method of VanRaden (1992).

Module Contents

a_decompose(myped, filetag='_a_decompose_') &rArr; matrices [#]

Form the decomposed form of A, TDT', directly from a pedigree (after Henderson, 1976; Thompson, 1977; Mrode, 1996). Return D, a diagonal matrix, and T, a lower triagular matrix such that A = TDT'.

myped A PyPedal pedigree object.

filetag A descriptor prepended to output file names.

Returns: A diagonal matrix, D, and a lower triangular matrix, T.

a_inverse_df(myped, filetag='_a_inverse_df_') &rArr; matrix [#]

Directly form the inverse of A from the pedigree file - accounts for inbreeding - using the method of Quaas (1976).

myped A PyPedal pedigree object.

filetag A descriptor prepended to output file names.

Returns: The inverse of the NRM, A, accounting for inbreeding.

a_inverse_dnf(myped, filetag='_a_inverse_dnf_') &rArr; matrix [#]

Form the inverse of A directly using the method of Henderson (1976) which does not account for inbreeding.

myped A PyPedal pedigree object.

filetag A descriptor prepended to output file names.

Returns: The inverse of the NRM, A, not accounting for inbreeding.

a_matrix(myped, filetag='_a_matrix_', save=0) &rArr; array [#]

a_matrix() is used to form a numerator relationship matrix from a pedigree. DEPRECATED. use fast_a_matrix() instead.

myped A PyPedal pedigree object.

filetag A descriptor prepended to output file names.

save Flag to indicate whether or not the relationship matrix is written to a file.

Returns: The NRM as a numarray matrix.

fast_a_matrix(myped, filetag='_new_a_matrix_', save=0, debug=0) &rArr; matrix [#]

Form a numerator relationship matrix from a pedigree. fast_a_matrix() is a hacked version of a_matrix() modified to try and improve performance. Lists of animal, sire, and dam IDs are formed and accessed rather than myped as it is much faster to access a member of a simple list rather than an attribute of an object in a list. Further note that only the diagonal and uppef off diagonal of A are populated. This is done to save n(n+1) / 2 matix writes. For a 1000-element array, this saves 500,500 writes.

myped A PyPedal pedigree object.

filetag A descriptor prepended to output file names.

save Flag to indicate whether or not the relationship matrix is written to a file.

Returns: The NRM as Numarray matrix.

fast_a_matrix_r(myped, filetag='_a_matrix_r_', save=0) &rArr; matrix [#]

Form a relationship matrix from a pedigree. fast_a_matrix_r() differs from fast_a_matrix() in that the coefficients of relationship are corrected for the inbreeding of the parents.

myped A PyPedal pedigree object.

filetag A descriptor prepended to output file names.

save Flag to indicate whether or not the relationship matrix is written to a file.

Returns: A relationship as Numarray matrix.

form_d_nof(myped) &rArr; matrix [#]

Form the diagonal matrix, D, used in decomposing A and forming the direct inverse of A. This function does not write output to a file - if you need D in a file, use the a_decompose() function. form_d() is a convenience function used by other functions. Note that inbreeding is not considered in the formation of D.

myped A PyPedal pedigree object.

Returns: A diagonal matrix, D.

inbreeding(myped, filetag='inbreeding', method='tabular') &rArr; dictionary [#]

inbreeding() is a proxy function used to dispatch pedigrees to the appropriate function for computing CoI. By default, small pedigrees $<$ 10,000 animals) are processed with the tabular method directly. For larger pedigrees, or if requested, the recursive method of VanRaden (1992) is used.

myped A PyPedal pedigree object.

filetag A descriptor prepended to output file names.

method Keyword indicating which method of computing CoI should be used (tabular—vanraden).

Returns: A dictionary of CoI keyed to renumbered animal IDs.

inbreeding_tabular(myped, filetag='_inbreeding_') &rArr; dictionary [#]

inbreeding_tabular() computes CoI using the tabular method by calling fast_a_matrix() to form the NRM directly. In order for this routine to return successfully requires that you are able to allocate a matrix of floats of dimension len(myped)**2.

myped A PyPedal pedigree object.

filetag A descriptor prepended to output file names.

Returns: A dictionary of CoI keyed to renumbered animal IDs

inbreeding_vanraden(myped, filetag='_inbreeding_', debug=0, cleanmaps=1) &rArr; dictionary [#]

inbreeding_vanraden() uses VanRaden's (1992) method for computing coefficients of inbreeding in a large pedigree. The method works as follows: 1. Take a large pedigree and order it from youngest animal to oldest (n, n-1, ..., 1); 2. Recurse through the pedigree to find all of the ancestors of that animal n; 3. Reorder and renumber that "subpedigree"; 4. Compute coefficients of inbreeding for that "subpedigree" using the tabular method (Emik and Terrill, 1949); 5. Put the coefficients of inbreeding in a dictionary; 6. Repeat 2 - 5 for animals n-1 through 1; the process is slowest for the early pedigrees and fastest for the later pedigrees.

myped A PyPedal pedigree object.

filetag A descriptor prepended to output file names.

debug A flag passed to pyp_nrm/renumber() which indicates whether or not progress reports should be printed to stdout.

cleanmaps Flag to denote whether or not subpedigree ID maps should be delete after they are used (0—1)

Returns: A dictionary of CoI keyed to renumbered animal IDs

recurse_pedigree(myped, anid, _ped) &rArr; list [#]

recurse_pedigree() performs the recursion needed to build the subpedigrees used by inbreeding_vanraden(). For the animal with animalID anid recurse_pedigree() will recurse through the pedigree myped and add references to the relatives of anid to the temporary pedigree, _ped.

myped A PyPedal pedigree.

anid The ID of the animal whose relatives are being located.

_ped A temporary PyPedal pedigree that stores references to relatives of anid.

Returns: A list of references to the relatives of anid contained in myped.

recurse_pedigree_idonly(myped, anid, _ped) &rArr; list [#]

recurse_pedigree_idonly() performs the recursion needed to build subpedigrees.

myped A PyPedal pedigree.

anid The ID of the animal whose relatives are being located.

_ped A PyPedal list that stores the animalIDs of relatives of anid.

Returns: A list of animalIDs of the relatives of anid contained in myped.

recurse_pedigree_n(myped, anid, _ped, depth=3) &rArr; list [#]

recurse_pedigree_n() recurses to build a pedigree of depth n. A depth less than 1 returns the animal whose relatives were to be identified.

myped A PyPedal pedigree.

anid The ID of the animal whose relatives are being located.

_ped A temporary PyPedal pedigree that stores references to relatives of anid.

depth The depth of the pedigree to return.

Returns: A list of references to the relatives of anid contained in myped.

recurse_pedigree_onesided(myped, anid, _ped, side) &rArr; list [#]

recurse_pedigree_onsided() recurses to build a subpedigree from either the sire or dam side of a pedigree.

myped A PyPedal pedigree.

side The side to build: ’s’ for sire and ’d’ for dam.

anid The ID of the animal whose relatives are being located.

_ped A temporary PyPedal pedigree that stores references to relatives of anid.

Returns: A list of references to the relatives of anid contained in myped.

5.9 pyp_utils

pyp_utils contains a set of procedures for creating and operating on PyPedal pedigrees. This includes routines for reordering and renumbering pedigrees, as well as for modifying pedigrees.

Module Contents

assign_offspring(myped, debug=0) [#]

assign_offspring() assigns offspring to their parent(s)’s unknown sex offspring list (well, dictionary).

myped A renumbered and reordered PyPedal pedigree object.

debug Flag to indicate whether or not progress messages are written to stdout.

assign_sexes(myped, debug=0) [#]

assign_sexes() assigns a sex to every animal in the pedigree using sire and daughter lists for improved accuracy.

myped A renumbered and reordered PyPedal pedigree object.

debug Flag to indicate whether or not progress messages are written to stdout.

delete_id_map(filetag=’_renumbered_’) &rArr; integer [#]

delete_id_map() checks to see if an ID map for the given filetag exists. If the file exists, it is deleted.

filetag A descriptor prepended to output file names that is used to determine name of the file to delete.

Returns: A flag indicating whether or not the file was successfully deleted (0—1)

fast_reorder(myped, filetag=’_new_reordered_’, io=’no’, debug=0) &rArr; list [#]

fast_reorder() renumbers a pedigree such that parents precede their offspring in the pedigree. In order to minimize overhead as much as is reasonably possible, a list of animal IDs that have already been seen is kept. Whenever a parent that is not in the seen list is encountered, the offspring of that parent is moved to the end of the pedigree. This should ensure that the pedigree is properly sorted such that all parents precede their offspring. myped is reordered in place. fast_reorder() uses dictionaries to renumber the pedigree based on paddedIDs.

myped A PyPedal pedigree object.

filetag A descriptor prepended to output file names.

io Indicates whether or not to write the reordered pedigree to a file (yes—no).

debug Flag to indicate whether or not debugging messages are written to STDOUT.

Returns: A reordered PyPedal pedigree.

id_map_new_to_old(id_map, new_id) &rArr; integer [#]

id_map_new_to_old() takes an ID from a renumbered pedigree and an ID map, and returns the original ID number.

id_map A dictionary mapping renumbered animalIDs to original animalIDs.

new_id A renumbered animalID.

Returns: A dictionary whose keys are renumbered IDs and whose values are original IDs.

load_id_map(filetag='_renumbered_') &rArr; dictionary [#]

load_id_map() reads an ID map from the file generated by pyp_utils/renumber() into a dictionary. There is a VERY similar function, pyp_io/id_map_from_file(), that is deprecated because it is much more fragile that this procedure.

filetag A descriptor prepended to output file names that is used to determine the input file name.

Returns: A dictionary whose keys are renumbered IDs and whose values are original IDs.

load_pedigree(inputfile, filetag='_load_pedigree_', sepchar=',', debug=0, io='no', renum=1, outformat='0', name='Pedigree M]

load_pedigree() wraps several processes useful for loading and preparing a pedigree for use in an analysis, including reading the animals into a list of animal objects, forming lists of sires and dams, checking for common errors, setting ancestor flags, and renumbering the pedigree.

inputfile Name of the file from which the pedigree is to be read.

filetag A descriptor prepended to output file names.

sepchar Indicates which character is used to separate entries in the pedigree file (default is CSV).

debug Flag to indicate whether or not progress messages are written to stdout.

io Indicates whether or not to write an ancestor list to a file..

renum Flag to indicate whether or not the pedigree is to be renumbered.

outformat Flag to indicate whether or not to write an asd pedigree (0) or a full pedigree (1).

name The name of the pedigree (descriptive).

alleles Flag to indicate whether or not pyp_metrics/effective_founder_genomes() should be called for a single round to assign alleles.

progress Flag to indicate whether or not to print progress messages to STDOUT when loading very large pedigrees.

Returns: A list of Animal() objects; a pedigree metadata object.

new_preprocess(kw) &rArr; list [#]**

new_preprocess() processes a pedigree file, which includes reading the animals into a list of animal objects, forming lists of sires and dams, and checking for common errors.

inputfile Name of the file from which the pedigree is to be read.

sepchar What character separates entries in the pedigree file (default is CSV).

debug Flag to indicate whether or not progress messages are written to stdout.

Returns: A list of Animal() objects; this is what PyPedal calls a pedigree.

pedigree_range(myped, n) &rArr; list [#]

pedigree_range() takes a renumbered pedigree and removes all individuals with a renumbered ID $>$ n. The reduced pedigree is returned. Assumes that the input pedigree is sorted on animal key in ascending order.

myped A PyPedal pedigree object.

n A renumbered animalID.

Returns: A pedigree containing only animals born in the given birthyear.

preprocess(inputfile, sepchar=',', debug=0, progress=0) &rArr; list [#]

preprocess() processes a pedigree file, which includes reading the animals into a list of animal objects, forming lists of sires and dams, and checking for common errors.

inputfile Name of the file from which the pedigree is to be read.

sepchar Indicates which character is used to separate entries in the pedigree file (default is CSV).

debug Flag to indicate whether or not progress messages are written to stdout.

progress Flag to indicate whether or not to print progress messages to STDOUT when loading very large pedigrees.

Returns: A list of Animal() objects; this is what PyPedal calls a pedigree.

pyp_nice_time() &rArr; string [#]

pyp_nice_time() returns the current date and time formatted as, e.g., Wed Mar 30 10:26:31 2005.

Returns: A string containg the formatted date and time.

renumber(myped, filetag='_renumbered_', io='no', outformat='0', debug=0) &rArr; list [#]

renumber() takes a pedigree as input and renumbers it such that the oldest animal in the pedigree has an ID of '1' and the n-th animal has an ID of 'n'. If the pedigree is not ordered from oldest to youngest such that all offspring precede their offspring, the pedigree will be reordered. The renumbered pedigree is written to disc in 'asd' format and a map file that associates sequential IDs with original IDs is also written.

myped A PyPedal pedigree object.

filetag A descriptor prepended to output file names.

io Indicates whether or not to write the renumbered pedigree to a file (yes—no).

outformat Flag to indicate whether or not ro write an asd pedigree (0) or a full pedigree (1).

debug Flag to indicate whether or not progress messages are written to stdout.

Returns: A reordered PyPedal pedigree.

reorder(myped, filetag='_reordered_', io='no') &rArr; list [#]

reorder() renumbers a pedigree such that parents precede their offspring in the pedigree. In order to minimize overhead as much as is reasonably possible, a list of animal IDs that have already been seen is kept. Whenever a parent that is not in the seen list is encountered, the offspring of that parent is moved to the end of the pedigree. This should ensure that the pedigree is properly sorted such that all parents precede their offspring. myped is reordered in place. reorder() is VERY slow, but I am pretty sure that it works correctly.

myped A PyPedal pedigree object.

filetag A descriptor prepended to output file names.

io Indicates whether or not to write the reordered pedigree to a file (yes—no).

Returns: A reordered PyPedal pedigree.

reverse_string(mystring) &rArr; string [#]

reverse_string() reverses the input string and returns the reversed version.

mystring A non-empty Python string.

Returns: The input string with the order of its characters reversed.

set_age(myped) [#]

set_age() Computes ages for all animals in a pedigree based on the global BASE_DEMOGRAPHIC_YEAR defined in pyp_demog.py. If the by is unknown, the inferred generation is used. If the inferred generation is unknown, the age is set to -999.

myped A PyPedal pedigree object.

set_ancestor_flag(myped, filetag='_ancestor_', io='no', debug=0) &rArr; integer [#]

set_ancestor_flag() loops through a pedigree to build a dictionary of all of the parents in the pedigree. It then sets the ancestor flags for the parents. It assumes that the pedigree is reordered and renumbered. NOTE: set_ancestor_flag() expects a reordered and renumbered pedigree as input!

myped A PyPedal pedigree object.

filetag A descriptor prepended to output file names.

io Indicates whether or not to write an ancestor list to a file.

Returns: 0 for failure and 1 for success.

set_generation(myped) [#]

set_generation() Works through a pedigree to infer the generation to which an animal belongs based on founders belonging to generation 1. The igen assigned to an animal as the larger of sire.igen+1 and dam.igen+1. This routine assumes that myped is reordered and renumbered.

myped A PyPedal pedigree object.

set_species(myped, species='u') [#]

set_species() assigns a specie to every animal in the pedigree.

myped A PyPedal pedigree object.

species A PyPedal string.

simple_histogram_dictionary(mydict, histchar='*', histstep=5) &rArr; dictionary [#]

simple_histogram_dictionary() returns a dictionary containing a simple, text histogram. The input dictionary is assumed to contain keys which are distinct levels and values that are counts.

mydict A non-empty Python dictionary.

histchar The character used to draw the histogram (default is '*').

histstep Used to determine the number of bins (stars) in the diagram.

Returns: A dictionary containing the histogram by level.

sort_dict_by_keys(mydict) &rArr; dictionary [#]

sort_dict_by_keys() returns a dictionary where the values in the dictionary in the order obtained by sorting the keys. Taken from the routine sortedDictValues3 in the "Python Cookbook", p. 39.

mydict A non-empty Python dictionary.

Returns: The input dictionary with keys sorted in ascending order.

trim_pedigree_to_year(myped, year) &rArr; list [#]

trim_pedigree_to_year() takes pedigrees and removes all individuals who were not born in birthyear 'year'.

myped A PyPedal pedigree object.

year A birthyear.

Returns: A pedigree containing only animals born in the given birthyear.

Glossary

This chapter provides a glossary of terms.[1]

coefficient of inbreeding ...

coefficient of relationship ...

effective ancestor number ...

effective founder number ...

effective population size ...

founder ...

numerator relationship matrix ...

pedigree A PyPedal pedigree consists of a Python list containing instances of PyPedal Animal objects.

[1]Please let me know of any additions to this list which you feel would be helpful.