PyPedal:
Software for pedigree analysis

Release 2.0.0a19

John B. Cole, PhD

August 11, 2005

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

Legal Notice

Disclaimer

The author of this software does not make any warranty, express or implied, or assume any liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represent that its use would not infringe privately-owned rights. Reference herein to any specific commercial products, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or the author. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government and shall not be used for advertising or product endorsement purposes.

CONTENTS

LIST OF TABLES

License

CHAPTER

TWO

Introduction

This chapter introduces the PyPedal module for Python 2.4 and outlines the rest of the document.

PyPedal (**P**ython **Ped**igree An**al**ysis) is a tool for analyzing 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. Tools for pedigree visualization and report generation are provided. 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 gene-dropping simulations to compute effective number of founder genomes, but no other measures of alleic diversity are currently supported.

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() as PyPedal() makes use of some version-specific features found only in 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.

This document is the "official" documentation for PyPedal. It includes a tutorial and is 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. The best way to learn is by doing — the aim of this tutorial is to guide you along this "doing."

This content of this manual is broken down as follows:

License Chapter 1 describes the license under which PyPedal is distributed. It is important that you review the license before using the program.

Installing PyPedal Chapter 3 provides information on testing Python and installing PyPedal.

High-Level Overview Chapter 4 gives a high-level overview of the components of the PyPedal system as a whole.

Applications Programming Interface Chapter 5 includes a complete reference, including useage notes, for all functions in all PyPedal. modules.

PyPedal Tutorial Chapter 6 provides a gentle introduction to PyPedal.

Glossary Chapter 7 provides a glossary of terms.

References and Indices are provided at the end of the manual.

2.1 Implemented Features

PyPedal is currently capable of doing the following things:

- Reading pedigree files in user-defined 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;
- 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 (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;
- Calculation of theoretical effective population size;
- Calculation of actual effective population size based on the change in population average inbreeding;
- Computation of effective founder number using the exact algorithm of Lacy (1989);
- Computation of effective founder number using the approximate algorithm of Boichard, Maignel, and Verrier (1997);
- Computation of effective ancestor number using the algorithm of Boichard, Maignel, and Verrier (1997);
- 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.

A full list of features, including notes on useage and computational details, is provided in Chapter 5. PyPedal has been used to perform calculations on pedigrees as large as 100,000 animals and has used in scientific research (Cole, Franke, and Leighton 2004).

2.2 Where to get information and code

PyPedal and its documentation are available at: http://pypedal.sourceforge.net/. The Numarray web site is: http://numpy.sourceforge.net/. The Python web site is http://www.python.org/.

2.3 Acknowledgments

PyPedal was initially written to support the author's dissertation research while at Louisiana State University, Baton Rouge (http://www.lsu.edu/). It lay fallow for some time but 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. Additional feedback in the form of bug reports, feature requests, and discussion of computing strategies was provided by Edward H. Hagen (Institute for Theoretical Biology, Humboldt-Universitï¿½zu Berlin), Kathy Hanford (University of Nebraska, Lincoln), Thomas von Hassell, and Gianluca Saba.

CHAPTER

THREE

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.

3.1 Overview

In addition to Python 2.4 (http://www.python.org/2.4.1/) PyPedal makes use of functionality from several other libraries. Some of them must be installed for you to use PyPedal, and others only ned to be installed if you would like to make use of certain features of PyPedal.

- **REQUIRED:** Numeric version 23.1 or later (http://numeric.scipy.org/) or numarray version 1.2.3 or later (http://www.stsci.edu/resources/software_hardware/numarray) are used for mathematical computations.
- Graphviz (http://www.graphviz.org/), pydot (http://dkbza.org/pydot.html), and pyparsing (http://pyparsing.sourceforge.net/) are used to visualize pedigrees as directed graphs.
- matplotlib (http://matplotlib.sourceforge.net/) is required by some functions in the `pyp_graph` module.
- The Python Imaging Library (http://www.pythonware.com/products/pil/) is used by some routines in the `pyp_graph` module.
- SQLite (http://sqlite.org/) and pysqlite (http://initd.org/tracker/pysqlite) provide relational database functionality that is used by the `pyp_db` and `pyp_reports` modules.
- TestOOB (http://testoob.sourceforge.net/) provides enhanced unit-testing functionality that is used by the `pyp_tests` module.

If you do not install one or more optional modules you will still be able to use PyPedal, although some features may not be available to you. Details on installing the libraries listed above can be found on the webpages listed. All of these extensions are available for Unix-type operating systems (e.g. Linux, Mac OS X) as well as for Microsoft Windows.

3.2 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 2.4! When installed, starting Python by typing python at the shell or double-clicking on the Python interpreter should give a prompt such as:

```
Python 2.4 (#1, Feb 25 2005, 12:30:11)
[GCC 3.3.3] on linux2
Type "help", "copyright", "credits" or "license" for more information.
```

If you have problems getting Python to work, contact your local support person or e-mail 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).

3.3 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__
'1.2.3'
```

indicating that numarray is installed. If it is installed, you can skip the next section and go ahead to section 3.4. 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.

3.4 Installing PyPedal

In order to get PyPedal, visit the official website at http://pypedal.sourceforge.net/. 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 **not** currently possible to get the latest sources from a CVS repository.

3.4.1 Installing on Unix, Linux, and Mac OSX

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

```
gunzip pypedal-2.0.0a19.tar.gz
tar xf pypedal-2.0.0a19.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. All of the code samples will assume that they have been preceded by a statement:

```
>>> from PyPedal import *
```

3.4.2 Installing on Windows

To install PyPedal, 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, please let me 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, please let me know.

3.5 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__
'2.0.0a19'
```

indicating that PyPedal is installed.

CHAPTER

FOUR

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.

4.1 Interacting with PyPedal

There are two ways to interact with PyPedal: interactively from a Python command line, and programmatically using a script that is run using the Python interpreter. The latter is preferred to the former for any but trivial examples, although it is useful to work with the command line while learning how to use PyPedal. A number of sample programs are included with the PyPedal distribution. Examples of both styles of interaction may be found in the tutorial (Chapter **??**).

4.2 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 a little about all of them. 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.

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 3.4.1.

4.3 Pedigree Files

Pedigree files consist of plain-text files (also known as ASCII or flatfiles) whose rows contain records on individual animals and whose columns contain different variables. The columns are delimited (separated from one another) by some character such as a space or a tab (
t). Pedigree files may also contain comments (notes) about the pedigree that are ignored by PyPedal; comments always begin with an octothorpe (#). For example, the following pedigree contains records for 13 animals, and each record contains three variables (animal ID, sire ID, and dam ID):

```
# This pedigree is taken from Boichard et al. (1997).
# Each records contains an animal ID, a sire ID, and
# a dam ID.
1 0 0
2 0 0
3 0 0
4 0 0
5 2 3
6 0 0
7 5 6
8 0 0
9 1 2
10 4 5
11 7 8
12 7 8
13 7 8
```

When this pedigree is processed by PyPedal the comments are ignored. If you need to change the default column separator, which is a space (' '), set the `sepchar` option to the desired value. For example, if your columns are

tab-delimited you would set the option as:

```
options['sepchar'] = '\t'
```

Options are discussed at length in section 4.3.2.

4.3.1 Pedigree Format Codes

Pedigree format codes consisting of a string of characters are used to describe the contents of a pedigree file. The simplest pedigree file that can be read by PyPedal is shown above; the pedigree format for this file is `asd`. A pedigree format is required for reading a pedigree; there is no default code used, and PyPedal wil halt with an error if you do not specify one. You specify the format using an option statement at the start of your program:

```
options['pedformat'] = 'asd'
```

Please note that the format codes are case-sensitive, which means that 'a' is considered to be a different character than 'A'. 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 = animal ID as a string (cannot contain sepchar)
- S = sire ID as a string (cannot contain sepchar)
- D = dam ID as a string (cannot contain sepchar)
- L = alleles (two alleles separated by a non-null character)

As noted, all pedigrees must contain columns corresponding to animals, sires, and dams. Pedigree codes should be entered in the same order in which the columns occur in the pedigee file. The character that separates alleles when the 'L' format code is used cannot be the same character used to separate columns in the pedigree file. If you do use the same character, PyPedal will write an error message to the log file and screen and halt.

If you used an earlier version of PyPedal you may have added a pedigree format string, e.g. `% asd`, to your pedigree file(s). You no longer need to include that string in your pedigrees, and if PyPedal sees one while reading a pedigree file it will ignore that line.

4.3.2 Options

Many aspects of PyPedal's operation can be controlled using a series of options. A complete list of these options, their defaults, and a brief desription of their purpose is presented in Table 4.1. Options are stored in a Python dictionary that you must create in your programs. You must specify values for the `pedfile` and `pedformat` options; all others are optional. `pedfile` is a string containing the name of the file from which your pedigree will be read. `pedformat` is a string containing a pedigree format code (see section 4.3.1) for each column in the datafile in the order in which those columns occur. The following code fragement demonstrates how options are specified.

```
options = {}
options['messages'] = 'verbose'
options['renumber'] = 0
options['counter'] = 5
options['pedfile'] = 'new_lacy.ped'
options['pedformat'] = 'asd'
options['pedname'] = 'Lacy Pedigree'
example = pyp_newclasses.NewPedigree(options)
```

First, a dictionary named 'options' is created; you may use any name you like as long as it is a valid Python variable name. Next, values are assigned to several options. Finally, 'options' is passed to pyp_newclasses.NewPedigree(), which requires that you pass it a dictionary of options. If you do not peovide any options, PyPedal will halt with an error.

A single PyPedal program may be used to read one or more pedigrees. Each pedigree that you read must be passed its own dictionary of options. The easiest way to do this is by creating a dictionary with global options. You can then customize the dictionary for each pedigree you want to read. Once you have created a PyPedal pedigree by calling pyp_newclasses.NewPedigree(options) you can change the options dictionary without affecting that pedigree because it has a separate copy of those options stored in its 'kw' attribute. The following code fragment demonstrates how to read two pedigree files using the same dictionary of options.

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

if __name__ == '__main__':
#   Read the first pedigree
    options['pedfile'] = 'new_lacy.ped'
    options['pedformat'] = 'asd'
    options['pedname'] = 'Lacy Pedigree'
    example1 = pyp_newclasses.NewPedigree(options)
    example1.load()
#   Read the second pedigree
    options['pedfile'] = 'new_boichard.ped'
    options['pedformat'] = 'asdg'
    options['pedname'] = 'Boichard Pedigree'
    example2 = pyp_newclasses.NewPedigree(options)
    example2.load()
```

Note that `pedformat` only needs to be changed if the two pedigrees have different formats. Only `pedfile` **has** to

Table 4.1: Options for controlling PyPedal.

Option	Default	Note(s)
alleles_sepchar	'/'	The character separating the two alleles in an animal's allelotype. 'alleles_sepchar' must NOT be the same as 'sepchar'!
counter	1000	How often should PyPedal write a note to the screen when reading large pedigree files.
database_name	'pypedal'	The name of the database to be used when using the pyp_reports nodule.
dbtable_name	filetag	The name of the database table to which the current pedigree will be written when using the pyp_reports module.
debug_messages	0	Indicates whether or not PyPedal should print debugging information.
messages	'verbose'	How many message should PyPedal provide; only 'verbose' is currently implemented.
file_io	1	When true, routines that can write results to output files will do so and put messages in the program log to that effect.
filetag	pedfile	A filetag is a descriptive label attached to output files created when processing a pedigree. By default the filetag is based on 'pedfile', minus its file extension.
form_nrm	0	Indicates whether or not to form a NRM and bind it to the pedigree as an instance of a NewAMatrix object.
logfile	filetag.log	The name of the file to which PyPedal should write messages about its progress.
messages	'verbose'	How chatty should be PyPedal be with respect to messages to the user. 'verbose' indicates that all status messages will be written to STDOUT, while 'quiet' suppresses all output to STDOUT.
missing_parent	'0'	Indicates what code is used to identify missing/unknown parentsin the pedigree file.
nrm_method	'nrm'	Specifies that an NRM formed from the current pedigree as an instance of a NewAMatrix object should ('frm') or should not ('nrm') be corrected for parental inbreeding.
pedfile	None	File from which pedigree is read; must provide.
pedformat	'asd'	See PEDIGREE_FORMAT_CODES for details.
pedname	'Untitled'	A name/title for your pedigree.
pedgree_is_renumbered	0	Indicates whether or not the pedigree has been renumbered.
renumber	0	Renumber the pedigree after reading from file (0/1).
sepchar	' '	The character separating columns of input in the pedfile.
set_ancestors	0	Iterate over the pedigree to assign ancestors lists to parents in the pedigree (0/1).
set_alleles	0	Assign alleles for use in gene-drop simulations (0/1).
set_generations	0	Iterate over the pedigree to infer generations (0/1).
slow_reorder	1	Option to override the slow, but more correct, reordering routine used by PyPedal by default (0/1). ONLY CHANGE THIS IF YOU REALLY UNDERSTAND WHAT IT DOES! Careless use of this option can lead to erroneous results.

be changed.

All pedigree options other than `pedfile` and `pedformat` have default values. If you provide a value that is invalid the option will revert to the default. In most cases, a message to that effect will also be placed in the log file.

4.4 Logging

PyPedal uses the `logging` module that is part of the Python standard library to record events during pedigree processing. Informative messages, as well as warnings and errors, are written to the logfile, which can be found in the directory from which you ran PyPedal. An example of a log from a successful (error-free) run of a program is presented below:

```
Fri, 06 May 2005 10:27:22 INFO     Logfile boichard2.log instantiated.
Fri, 06 May 2005 10:27:22 INFO     Preprocessing boichard2.ped
Fri, 06 May 2005 10:27:22 INFO     Opening pedigree file
Fri, 06 May 2005 10:27:22 INFO     Pedigree comment (line 1): # This pedigree is taken from Boi
Fri, 06 May 2005 10:27:22 INFO     Pedigree comment (line 2): # It contains two unrelated famil
Fri, 06 May 2005 10:27:22 WARNING  Encountered deprecated pedigree format string (% asdg
) on line 3 of the pedigree file.
Fri, 06 May 2005 10:27:22 WARNING  Reached end-of-line in boichard2.ped after reading 23 lines.
Fri, 06 May 2005 10:27:22 INFO     Closing pedigree file
Fri, 06 May 2005 10:27:22 INFO     Assigning offspring
Fri, 06 May 2005 10:27:22 INFO     Creating pedigree metadata object
Fri, 06 May 2005 10:27:22 INFO     Forming A-matrix from pedigree
Fri, 06 May 2005 10:27:22 INFO     Formed A-matrix from pedigree
```

The `WARNING`s let you know when something unexpected has happened, although you might argue that coming to the end of an input file is not unexpected. If you get unexpected results from your program make sure that you check the logfile for details – some subroutines return default values such as -999 when a problem occurs but do not halt the program. Note that comments found in the pedigree file were written to the log, as was an deprecated pedigree format string used by earlier versions of PyPedal. When an error from which PyPedal cannot recover occurs a message is written to both the screen and the logfile. We can see from the following log that the number of columns in the pedigree file did not match the number of columns in the pedigree format option.

```
Thu, 04 Aug 2005 15:36:18 INFO     Logfile hartlandclark.log instantiated.
Thu, 04 Aug 2005 15:36:18 INFO     Preprocessing hartlandclark.ped
Thu, 04 Aug 2005 15:36:18 INFO     Opening pedigree file
Thu, 04 Aug 2005 15:36:18 INFO     Pedigree comment (line 1): # Pedigree from van Noordwijck an
Thu, 04 Aug 2005 15:36:18 INFO     Pedigree comment (line 2): # in Hartl and Clark (1989), p. 2
Thu, 04 Aug 2005 15:36:18 ERROR    The record on line 3 of file hartlandclark.ped does not have
```

There is no sensible ”best guess” that PyPedal can make about handling this situation, so it halts. There are some cases where PyPedal does ”guess” how it should proceed in the face of ambiguity, which is why it is always a good idea to check for `WARNING`s in your logfiles.

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_db

pyp_db contains a set of procedures for ...

Module Contents

createPedigreeDatabase(dbname='pypedal') ⇒ **integer** createPedigreeDatabase() creates a new database in SQLite.

dbname The name of the database to create.

Returns: A 1 on successful database creation, a 0 otherwise.

createPedigreeTable(curs, tablename='example') ⇒ **integer** createPedigreeDatabase() creates a new pedigree table in a SQLite database.

tablename The name of the table to create.

Returns: A 1 on successful table creation, a 0 otherwise.

databaseQuery(sql, curs=0, dbname='pypedal') ⇒ **string** databaseQuery() executes an SQLite query. This is a wrapper function used by the reporting functions that need to fetch data from SQLite. I wrote it so that any changes that need to be made in the way PyPedal talks to SQLite will only need to be changed in one place.

sql A string containing an SQL query.

_curs An [optional] SQLite cursor.

dbname The database into which the pedigree will be loaded.

Returns: The results of the query, or 0 if no resultset.

getCursor(dbname='pypedal') ⇒ **cursor** getCursor() creates a database connection and returns a cursor on success or a 0 on failure. It isvery useful for non-trivial queries because it creates SQLite aggrefates before returning the cursor. The reporting routines in pyp_reports make heavy use of getCursor().

dbname The database into which the pedigree will be loaded.

Returns: An SQLite cursor if the database exists, a 0 otherwise.

loadPedigreeTable(pedobj) ⇒ **integer** loadPedigreeDatabase() takes a PyPedal pedigree object and loads the animal records in that pedigree into an SQLite table.

pedobj A PyPedal pedigree object.

dbname The database into which the pedigree will be loaded.

tablename The table into which the pedigree will be loaded.

Returns: A 1 on successful table load, a 0 otherwise.

tableCountRows(dbname='pypedal', tablename='example') ⇒ **integer** tableCountRows() returns the number of rows in a table.

dbname The database into which the pedigree will be loaded.

tablename The table into which the pedigree will be loaded.

Returns: The number of rows in the table 1 or 0.

tableDropRows(dbname='pypedal', tablename='example') ⇒ **integer** tableDropRows() drops all of the data from an existing table.

dbname The database into which the pedigree will be loaded.

tablename The table into which the pedigree will be loaded.

Returns: A 1 if the data were dropped, a 0 otherwise.

tableExists(dbname='pypedal', tablename='example') ⇒ **integer** tableExists() queries the sqlite_master view in an SQLite database to determine whether or not a table exists.

dbname The database into which the pedigree will be loaded.

tablename The table into which the pedigree will be loaded.

Returns: A 1 if the table exists, a 0 otherwise.

The PypMean Class

PypMean() (class) PypMean is a user-defined aggregate for SQLite for returning means from queries.

The PypSSD Class

PypSSD() (class) PypSSD is a user-defined aggregate for SQLite for returning sample standard deviations from queries.

The PypSum Class

PypSum() (class) PypSum is a user-defined aggregate for SQLite for returning sums from queries.

The PypSVar Class

PypSVar() (class) PypSVar is a user-defined aggregate for SQLite for returning sample variances from queries.

5.3 pyp_demog

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

Module Contents

age_distribution(pedobj, sex=1) ⇒ **None** 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 An instance of a PyPedal NewPedigree object.

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

founders_by_year(pedobj) ⇒ **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') ⇒ **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) ⇒ **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(pedobj) ⇒ **dictionary** sex_ratio() returns a dictionary containing the proportion of males and females in the population.

myped An instance of a PyPedal NewPedigree object.

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 routines 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(pedobj, gfilename='pedigree', gtitle='My_Pedigree', gformat='jpg', gsize='f', gdot='1') ⇒ **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.

pedobj 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.

pcolor_matrix_pylab(A, fname='pcolor_matrix_matplotlib') ⇒ **lists** pcolor_matrix_pylab() 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.

plot_founders_by_year(pedobj, gfilename='founders_by_year', gtitle='Founders by Birthyear') ⇒ **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') ⇒ **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) ⇒ **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) ⇒ **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) ⇒ **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.

spy_matrix_pylab(A, fname='spy_matrix_matplotlib') ⇒ **lists** spy_matrix_pylab() 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.

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.

a_inverse_from_file(inputfile) ⇒ **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(pedobj, ainv=") a_inverse_to_file() uses the Python pickle system for persistent objects to write the inverse of a relationship matrix to a file.

pedobj A PyPedal pedigree object.

filetag A descriptor prepended to output file names.

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

pedobj A PyPedal pedigree object.

dissertation_pedigree_to_pedig_format(pedobj) 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.

pedobj A PyPedal pedigree object.

dissertation_pedigree_to_pedig_format_mask(pedobj) 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.

pedobj A PyPedal pedigree object.

dissertation_pedigree_to_pedig_interest_format(pedobj) 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.

pedobj A PyPedal pedigree object.

pickle_pedigree(pedobj, filename=") ⇒ **integer** pickle_pedigree() pickles a pedigree.

pedobj An instance of a PyPedal pedigree object.

filename The name of the file to which the pedigree object should be pickled (optional).

Returns: A 1 on success, a 0 otherwise.

pyp_file_footer(ofhandle, caller="Unknown PyPedal routine") ⇒ **None** 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") ⇒ **integer** 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") ⇒ **integer** renderTitle() ... Produced HTML output by default.

unpickle_pedigree(filename="") ⇒ **object** unpickle_pedigree() reads a pickled pedigree in from a file and returns the unpacked pedigree object.

filename The name of the pickle file.

Returns: An instance of a NewPedigree object on success, a 0 otherwise.

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(pedobj, a="", method='nrm') ⇒ **dictionary** 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.

pedobj A PyPedal pedigree object.

a A numerator relationship matrix (optional).

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

Returns: A dictionary of non-zero individual inbreeding coefficients.

a_effective_ancestors_definite(pedobj, a="", gen="") ⇒ **float** a_effective_ancestors_definite() uses the algorithm in Appendix B of Boichard, Maignel, and Verrier (1997) 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.

pedobj A PyPedal pedigree object.

a A numerator relationship matrix (optional).

gen Generation of interest.

Returns: The effective founder number.

a_effective_ancestors_indefinite(pedobj, a="", gen="", n=25) ⇒ **float** a_effective_ancestors_indefinite() uses the approach outlined on pages 9 and 10 of Boichard et al. (Boichard, Maignel, and Verrier 1997) 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.

pedobj A PyPedal pedigree object.

a A numerator relationship matrix (optional).

gen Generation of interest.

Returns: The effective founder number.

a_effective_founders_boichard(pedobj, a=", gen=") ⇒ **float** a_effective_founders_boichard() uses the algorithm in Appendix A of Boichard, Maignel, and Verrier (1997) to compute the effective founder number for pedobj. 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.

pedobj A PyPedal pedigree object.

a A numerator relationship matrix (optional).

gen Generation of interest.

Returns: The effective founder number.

a_effective_founders_lacy(pedobj, a=") ⇒ **float** a_effective_founders_lacy() calculates the number of effective founders in a pedigree using the exact method of Lacy (1989).

pedobj A PyPedal pedigree object.

a A numerator relationship matrix (optional).

Returns: The effective founder number.

common_ancestors(anim_a, anim_b, pedobj) ⇒ **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.

pedobj A PyPedal pedigree object.

Returns: A list of animals related to anim_a AND anim_b

descendants(anid, pedobj, _desc) ⇒ **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

pedobj 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(pedobj, rounds=10) ⇒ **float** effective_founder_genomes() simulates the random segregation of founder alleles through a pedigree after the method of MacCluer, VandeBerg, Read, and Ryder (1986). At present only two alleles are simulated for each founder. Summary statistics are computed on the most recent generation.

pedobj A PyPedal pedigree object.

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

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

effective_founders_lacy(pedobj) ⇒ **float** effective_founders_lacy() calculates the number of effective founders in a pedigree using the exact method of Lacy (1989). 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 "population-wise" relationship matrix.

pedobj A PyPedal pedigree object.

Returns: The effective founder number.

fast_a_coefficients(pedobj, a='', method='nrm', debug=0) ⇒ **dictionary** a_fast_coefficients() writes population average coefficients of inbreeding and relationship to a file, as well as individual animal IDs and coefficients of inbreeding. It returns a list of non-zero individual CoI.

pedobj A PyPedal pedigree object.

a A numerator relationship matrix (optional).

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

Returns: A dictionary of non-zero individual inbreeding coefficients.

founder_descendants(pedobj) ⇒ **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(pedobj, units='y') ⇒ **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.

pedobj A PyPedal pedigree object.

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(pedobj, units='y') ⇒ **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 birthyear is provided a default value (1900) is assigned to all animals in the pedigree.

pedobj A PyPedal pedigree object.

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, pedobj) ⇒ **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.

pedobj A PyPedal pedigree object.

Returns: The coefficient of relationship of anim_a and anim_b

min_max_f(pedobj, a=", n=10) ⇒ **list** min_max_f() takes a pedigree and returns a list of the individuals with the n largest and n smallest coefficients of inbreeding. Individuals with CoI of zero are not included.

pedobj A PyPedal pedigree object.

a A numerator relationship matrix (optional).

n An integer (optional, default is 10).

Returns: Lists of the individuals with the n largest and the n smallest CoI in the pedigree as (ID, CoI) tuples.

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

pedobj A PyPedal pedigree object.

Returns: A dictionary containing the five average ages.

num_traced_gens(pedobj) ⇒ **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 at al. (Valera, Molina, Gutiérrez, Gómez, and Goyache 2005) for details.

pedobj A PyPedal pedigree object.

Returns: A dictionary containing the five average ages.

partial_inbreeding(pedobj) ⇒ **dictionary** partial_inbreeding() computes the number of equivalent generations as the sum of $\frac{1}{2}^n$, where n is the number of generations separating an individual and each of its known ancestors.

pedobj A PyPedal pedigree object.

Returns: A dictionary containing the five average ages.

pedigree_completeness(pedobj, gens=4) 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. This is similar to pedigree completeness as computed by Cassell, Adamec, and Pearson (2003), but with some of the modifications of VanRaden (2003) (http://www.aipl.arsusda.gov/reference/changes/eval0311.html).

pedobj A PyPedal pedigree object.

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

related_animals(anim_a, pedobj) ⇒ **list** related_animals() returns a list of the ancestors of an animal.

anim_a The renumbered ID of an animal, a.

pedobj A PyPedal pedigree object.

Returns: A list of animals related to anim_a

relationship(anim_a, anim_b, pedobj) ⇒ **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.

pedobj A PyPedal pedigree object.

Returns: The coefficient of relationship of anim_a and anim_b

theoretical_ne_from_metadata(pedobj) ⇒ None 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.

pedobj A PyPedal pedigree object.

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.

form_a_matrix(pedigree) ⇒ integer form_a_matrix() calls pyp_nrm/fast_a_matrix() or 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() ⇒ 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) ⇒ 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) ⇒ 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) ⇒ 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() ⇒ 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() ⇒ None printme() prints a summary of the data stored in the Animal() object.

self Reference to the current Animal() object

string_to_int(idstring) ⇒ None string_to_int() takes an Animal/Sire/Dam ID as a string and returns a string that can be represented as an integer by replacing each character in the string with its corresponding ASCII table value.

stringme() ⇒ None stringme() returns a summary of the data stored in the Animal() object as a string.

self Reference to the current Animal() object

trap() ⇒ None 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(pedsource='file') ⇒ 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() ⇒ **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() ⇒ **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') ⇒ **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() ⇒ **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) ⇒ **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() ⇒ **None** fileme() writes the metada stored in the Pedigree() object to disc.

self Reference to the current Pedigree() object

nud() ⇒ **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() ⇒ **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() ⇒ **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() ⇒ **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() ⇒ **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() ⇒ **None** printme() prints a summary of the metadata stored in the Pedigree() object.

self Reference to the current Pedigree() object

stringme() ⇒ **None** 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 (VanRaden 1992).

Module Contents

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

pedobj A PyPedal pedigree object.

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

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

pedobj A PyPedal pedigree object.

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

a_inverse_dnf(pedobj, filetag='_a_inverse_dnf_') ⇒ **matrix** Form the inverse of A directly using the method of Henderson (Henderson 1976) which does not account for inbreeding.

pedobj A PyPedal pedigree object.

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

a_matrix(pedobj, save=0) ⇒ **array** a_matrix() is used to form a numerator relationship matrix from a pedigree. DEPRECATED. use fast_a_matrix() instead.

pedobj A PyPedal pedigree object.

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(pedigree, pedopts, save=0) ⇒ **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 upper off-diagonal of A are populated. This is done to save n(n+1) / 2 matrix writes. For a 1000-element array, this saves 500,500 writes.

pedigree A PyPedal pedigree.

pedopts PyPedal options.

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(pedigree, pedopts, save=0) ⇒ **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.

pedobj A PyPedal pedigree object.

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

Returns: A relationship as Numarray matrix.

form_d_nof(pedobj) ⇒ **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.

pedobj A PyPedal pedigree object.

Returns: A diagonal matrix, D.

inbreeding(pedobj, method='tabular') ⇒ **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 (VanRaden 1992) is used.

pedobj A PyPedal pedigree object.

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(pedobj) ⇒ **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.

pedobj A PyPedal pedigree object.

Returns: A dictionary of CoI keyed to renumbered animal IDs

inbreeding_vanraden(pedobj, cleanmaps=1) ⇒ **dictionary** inbreeding_vanraden() uses VanRaden's (VanRaden 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 (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.

pedobj A PyPedal pedigree object.

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(pedobj, anid, _ped) ⇒ **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.

pedobj 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(pedobj, anid, _ped) ⇒ **list** recurse_pedigree_idonly() performs the recursion needed to build subpedigrees.

pedobj 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(pedobj, anid, _ped, depth=3) ⇒ **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.

pedobj 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(pedobj, anid, _ped, side) ⇒ **list** recurse_pedigree_onsided() recurses to build a subpedigree from either the sire or dam side of a pedigree.

pedobj 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_reports

pyp_reports contains a set of procedures for ...

Module Contents

meanMetricBy(pedobj, metric='fa', byvar='by') ⇒ **dictionary** meanMetricBy() returns a dictionary containing means for the metric variable keyed to levels of the byvar. This provides a quick-and-easy way of getting summary reports.

pedobj A PyPedal pedigree object.

metric The variable to summarize on a BY variable.

byvar The variable on which to group the metric.

Returns: A dictionary containing means for the metric variable keyed to levels of the byvar.

5.10 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(pedobj) ⇒ **integer** assign_offspring() assigns offspring to their parent(s)'s unknown sex offspring list (well, dictionary).

myped An instance of a NewPedigree object.

Returns: 0 for failure and 1 for success.

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

pedobj A renumbered and reordered PyPedal pedigree object.

Returns: 0 for failure and 1 for success.

delete_id_map(filetag='_renumbered_') ⇒ **integer** delete_id_map() checks to see if an ID map for the given file-tag 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) ⇒ **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.

load_id_map(filetag='_renumbered_') ⇒ **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 or an empty dictionary (on failure).

pedigree_range(pedobj, n) ⇒ **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 or an empty list (on failure).

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

None

Returns: A string containing the formatted date and time.

renumber(myped, filetag='_renumbered_', io='no', outformat='0', debug=0) ⇒ **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 to 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') ⇒ **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) ⇒ **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(pedobj) ⇒ **integer** 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.

pedobj A PyPedal pedigree object.

Returns: 0 for failure and 1 for success.

set_ancestor_flag(pedobj) ⇒ **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. set_ancestor_flag() expects a reordered and renumbered pedigree as input!

pedobj A PyPedal NewPedigree object.

Returns: 0 for failure and 1 for success.

set_generation(pedobj) ⇒ **integer** 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.

pedobj A PyPedal NewPedigree object.

Returns: 0 for failure and 1 for success.

set_species(pedobj, species='u') ⇒ **integer** set_species() assigns a specie to every animal in the pedigree.

pedobj A PyPedal pedigree object.

species A PyPedal string.

Returns: 0 for failure and 1 for success.

simple_histogram_dictionary(mydict, histchar='*', histstep=5) ⇒ **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 or an empty dictionary (on failure).

sort_dict_by_keys(mydict) ⇒ **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 or an empty dictionary (on failure).

sort_dict_by_values(first, second) ⇒ **list** sort_dict_by_values() returns a dictionary where the keys in the dictionary are sorted ascending value, first on value and then on key within value. The implementation was taken from John Hunter's contribution to a newsgroup thread: http://groups-beta.google.com/group/comp.lang.python/browse_thread/thread/bbc259f8454e4d3f/cc686f4cd795feb4?q=python+%

mydict A non-empty Python dictionary.

Returns: A list of tuples sorted in ascending order.

string_to_table_name(instring) ⇒ **string** string_to_table_name() takes an arbitrary string and returns a string that is safe to use as an SQLite table name.

instring A string that will be converted to an SQLite-safe table name.

Returns: A string that is safe to use as an SQLite table name.

trim_pedigree_to_year(pedobj, year) ⇒ 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 or an ampty list (on failure).

CHAPTER

SIX

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 5.

6.1 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!

6.2 A Gentle Introduction to PyPedal

For this tutorial we are going to use a sample pedigree from Hartl and Clark (Hartl and Clark 1989) (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**.

6.2.1 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 4.3.1. 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

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

enough to add the needed records automatically. Comment lines, which begin with '#', may appear anywhere in the file; they are ignored by the preprocessor.

```
# 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.

6.2.2 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.

```
001 print 'Starting pypedal.py at %s' % asctime(localtime(time()))
002 print '\tPreprocessing pedigree at %s' % asctime(localtime(time()))
003 example = preprocess('hartl.ped',sepchar=' ')
004 example = renumber(example,'example',io='yes')
005 print '\tCalling set_ancestor_flag at %s' % asctime(localtime(time()))
006 set_ancestor_flag(example,'example',io='yes')
007 print '\tCollecting pedigree metadata at %s' % asctime(localtime(time()))
008 example_meta = Pedigree(example,'example.ped','example_meta')
009 print '\tCalling a_effective_founders_lacy() at %s' % asctime(localtime(time()))
010 a_effective_founders_lacy(example,filetag='example')
011 print '\tCalling a_effective_founders_boichard() at %s' % asctime(localtime(time()))
012 a_effective_founders_boichard(example,filetag='example')
013 print '\tCalling a_effective_ancestors_definite() at %s' % asctime(localtime(time()))
014 a_effective_ancestors_definite(example,filetag='example')
015 print '\tCalling a_effective_ancestors_indefinite() at %s' % asctime(localtime(time()))
016 a_effective_ancestors_indefinite(example,filetag='example',n=10)
017 print '\tCalling related_animals() at %s' % asctime(localtime(time()))
018 list_a = related_animals(example[14].animalID,example)
019 print list_a
020 print '\tCalling related_animals() at %s' % asctime(localtime(time()))
021 list_b = related_animals(example[9].animalID,example)
022 print list_b
023 print '\tCalling common_ancestors() at %s' % asctime(localtime(time()))
024 list_r = common_ancestors(example[14].animalID,example[9].animalID,example)
025 print list_r
026 print 'Stopping pypedal.py at %s' % asctime(localtime(time()))
```

6.2.3 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

SEVEN

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.

BIBLIOGRAPHY

Boichard, D., L. Maignel, and E. Verrier (1997). The value of using probabilities of gene origin to measure genetic variability in a population. *Genetics Selection Evolution 29*, 5–23.

Cassell, B. G., V. Adamec, and R. E. Pearson (2003). Effect of incomplete pedigrees on estimates of inbreeding and inbreeding depression for days to first service and summit milk yield in Holsteins and Jerseys. *Journal of Dairy Science 86*, 2967–2976.

Cole, J. B., D. E. Franke, and E. A. Leighton (2004). Population structure of a colony of dog guides. *Journal of Animal Science 82*, 2906–2912.

Emik, L. O. and C. E. Terrill (1949). Systematic procedures for calculating inbreeding coefficients. *Journal of Heredity 40*, 51–55.

Hartl, D. L. and A. G. Clark (1989). *Principles of Population Genetics* (2nd ed.). Sinauer Associates, Inc.

Henderson, C. R. (1976). A simple method for computing the inverse of a numerator relationship matrix used in prediction of breeding values. *Biometrics 32*, 69–83. CRHenderson1976b.

Lacy, R. C. (1989). Analysis of founder representation in pedigrees: founder equivalents and founder genome equivalents. *Zoo Biology. 8*, 111–123.

MacCluer, J. W., J. L. VandeBerg, B. Read, and O. A. Ryder (1986). Pedigree analysis by computer simulation. *Zoo Biology. 5*, 147–160.

Mrode, R. A. (1996). *Linear Models for the Prediction of Animal Breeding Values*. CAB International. RAMrode1996.

Quaas, R. L. (1976). Computing the diagonal elements and inverse of a large numerator relationship matrix. *Biometrics 32*. RLQuaas1976a.

Valera, M., A. Molina, J. P. Gutiérrez, J. Gómez, and F. Goyache (2005). Pedigree analysis in the Andalusian horse: population structure, genetic variability and influence of the Carthusian strain. *Livestock Production Science 95*, 57–66.

VanRaden, P. M. (1992). Accounting for inbreeding and crossbreeding in genetic evaluation of large populations. *Journal of Dairy Science 75*, 3136–3144.

FUNCTION INDEX

INDEX