A Manual for use of PyPedal
A software package for pedigree analysis

Release 2.0.0b4

John B. Cole

November 29, 2005
Revised December 6, 2005

Animal Improvement Programs Laboratory, Agricultural Research Service, United States Department of Agriculture, Room 306 Bldg 005 BARC-West, 10300 Baltimore Avenue, Beltsville, MD 20705-2350

Abstract

Cole, J.B. 2005. A Manual for use of PyPedal: A software package for pedigree analysis. Animal Improvement Programs Laboratory, Agricultural Research Service, United States Department of Agriculture.

This manual in eleven chapters describes PyPedal (v 2.0), a software package for pedigree analysis, report generation, and data visualization. Metrics include coefficients of inbreeding and relationship, effective founder and ancestor numbers, and founder genome equivalents. Tools are provided for identifying ancestors and descendants, computing coefficients of inbreeding from potential matings, quantifying pedigree completeness, and visualizing pedigrees. Scripting support is provided by the Python programming language; this language may be used to easily automate analyses and implement new features. Input and output files utilize plain-text formats. The program has been used for the analysis of dairy cattle and working dog pedigrees. PyPedal runs on the GNU/Linux and Microsoft Windows operating systems. The program, documentation, and examples of usage are available at `http://pypedal.sourceforge.net/`.

Revised December 6, 2005

Legal Notice

Disclaimer

CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER

ONE

License

CHAPTER

TWO

Introduction

This chapter introduces the PyPedal module for Python 2.4, provides an overview of key features of the software, and describes the contents of this manual.

PyPedal (**P**ython **Ped**igree An**al**ysis) is a tool for analyzing pedigree files. It calculates several quantitative measures of genetic diversity from pedigrees, including average coefficients of inbreeding and relationship, effective founder numbers, and effective ancestor numbers. Checks are performed catch common mistakes in pedigree files, such as parents with more recent birthdates or smaller ID numbers than their offspring and animals appearing as both sires and dams in the pedigree. Tools for pedigree visualization and report generation are also provided. PyPedal only makes use of information on pedigree structure, not individual genotypes. Allelotypes can be assigned to founders for use in gene-dropping simulations to calculate the 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 programs or used interactively from the interpreter. You must have Python 2.4 (or later) installed in order to use PyPedal as PyPedal makes use of features found only in that version. The Numarray module must also be installed in order for you to use PyPedal, and may be found at `http://www.stsci.edu/resources/software_hardware/numarray`. In addition, there are a number of third-party packages used by PyPedal; they are discussed in Chapter 3.

This manual 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.

Methodology Chapter 5 provides a brief overview of the methodology used to calculate measures of genetic diversity.

HOWTOs Chapter 6 provides demonstrations of how to perform common tasks.

Graphics Chapter 7 provides details on producing graphics with PyPedal.

Reports Chapter 8 provides details about the report generation tools available in PyPedal.

Implementing New Features Chapter 9 introduces the idea of extensibility and walks the reader through the development of a new PyPedal routine.

Applications Programming Interface Chapter 10 includes a complete reference, including useage notes, for all functions in all PyPedal modules.

Glossary Chapter 11 provides a glossary of terms.

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

2.1 Implemented Features

A full list of features, including notes on useage and computational details, is provided in Chapter 10. Some of the notable features of PyPedal include:

- 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;
- Reordering and renumbering of pedigree files.
- 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 (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 and actual effective population sizes;
- 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. (1997);
- Computation of effective ancestor number using the algorithms of Boichard et al. (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;

PyPedal has been used to perform calculations on pedigrees as large as 600,000 animals and has been 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 Sourceforge site, `http://sourceforge.net/projects/pypedal/`, provides tools for reporting bugs (`https://sourceforge.net/tracker/?func=add&group_id=106679&atid=645233`, making feature requests (`https://sourceforge.net/tracker/?func=add&group_id=106679&atid=645236`), and discussing PyPedal (`https://sourceforge.net/forum/?group_id=106679`).

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/`). The initial development was supported in part by a grant from The Seeing Eye, Inc., Morristown, NJ, USA. 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 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 Bradley J. Heins (University of Minnesota-Twin Cities), Edward H. Hagen (Institute for Theoretical Biology, Humboldt-Universität zu 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 under Posix-type operating systems and Microsoft Windows.

3.1 Overview of installation

Before we can begin the tutorial, you need install and test Python, Numarray and some other Python extensions, and PyPedal itself. The extensions that you need to install in order to use all of the features of PyPedal are listed in Table 3.1. Note that some extensions need to be installed before others: `Numarray` should be installed first, SQLite must be installed before `pysqlite`, and `pyparsing` and Graphviz must be installed before `pydot`.

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 extensions listed above can be found on their respective websites. All of these extensions are available for Unix-type operating systems (e.g. Linux, Mac OS X) as well as for Microsoft Windows; most sites also provide binary installers for Windows. Python extensions can usually be installed by unzipping/untaring the archives, entering the folder, and issuing the command '`python setup.py install`' as a root/administrative user.

3.2 Testing the Python installation

The first step is to install Python 2.4 (or later) if you haven't already done so. Python is available at `http://sourceforge.net/projects/python/`. Click on the link corresponding to your platform, and follow the instructions presented there. Python can usually be started by typing '`python`' at the shell (Posix) or double-clicking on the Python interpreter (Windows). When you start Python you should see a message 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.

Table 3.1: Third-party extensions used by PyPedal.

Extension	Function	URL
elementtree	Lightweight XML processing	`http://effbot.org/zone/element-index.htm`
Graphviz	Draw directed graphs	`http://www.research.att.com/sw/tools/graphviz/`
matplotlib	Plotting, matrix visualization	`http://matplotlib.sourceforge.net/`
NetworkX	Network analysis	`https://networkx.lanl.gov/`
Numarray	Array manipulation	`http://www.stsci.edu/resources/software_hardware/numarray`
PIL	Image processing	`http://effbot.org/zone/pil-index.htm`
pydot	Interface to Graphviz	`http://dkbza.org/pydot.html`
pyparsing	Text parsing	`http://pyparsing.sourceforge.net/`
pysqlite	Interface to SQLite	`http://initd.org/tracker/pysqlite`
PythonDoc	Generate API documentation	`http://effbot.org/zone/pythondoc.htm`
ReportLab	Generate PDF documents	`http://www.reportlab.org/`
SQLite	Lightweight SQL database	`http://www.sqlite.org/`
testoob	Advanced unit testing	`http://testoob.sourceforge.net/`

3.3 Installing PyPedal

In order to get PyPedal, visit the official website at `http://pypedal.sourceforge.net/`. Click on the ”Sourceforge Page” link, click on the ”Download PyPedal” button, and select the latest file release. Files whose names end in ”.tar.gz” are source code releases. The other files are binaries for a given platform (if any are available).

The CVS repository on the Sourceforge site is not in synch with the development tree; to get the latest version you should download the source code release.

3.3.1 Installing on Unix, Linux, and Mac OSX

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

```
gunzip pypedal-2.0.0a20.tar.gz
tar xf pypedal-2.0.0a20.tar.gz
```

Follow the instructions in the top-level directory for compilation and installation. Installation is usually as simple as:

```
python setup.py install
```

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 statements such as:

```
>>> from PyPedal import <module-name>
```

A complete list of modules is provided in Chapter 10.

3.3.2 Installing on Windows

To install PyPedal, you need to be logged into an account with Administrator privileges. As a general rule, always remove any old version of PyPedal before installing the next version.

Please note that we have lightly tested PyPedal on Windows XP, but cannot guarantee that it runs without problems on Win-32 platforms! PyPedal should install and run properly on Win-32 as long as the dependencies mentioned above are satisfied.

In order to get your installation working correctly you will need to set some environment variables. Under Windows XP you access those variables by right-clicking on the *My Computer* icon on your desktop, selecting *Properties*, selecting the *Advanced* tab, and clicking the *Environment Variables* button. First, add `;C:\Python24` to the PATH by selecting it in the *User Variables* list and clicking *Edit*. Next, create a PYTHONPATH environment variable by clicking the `New` button under `User Variables`, entering the path to the PyPedal directory in the `Variable value` field.

The documentation for SQLite for Windows is kind of vague. I got it to work by downloading the files 'sqlite-3_2_7.zip' and 'sqlitedll-3_2_7.zip' and extracting their contents into `C:\Windows`. Your mileage may vary.

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:\python24\site-packages`.

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.

3.4 Testing the PyPedal 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__.version
’2.0.0b4’
```

indicating that PyPedal is installed.

CHAPTER

FOUR

High-Level Overview

In this chapter, a high-level overview of PyPedal is presented, including key components of the module and concepts used throughout this manual.

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 convenient packages. Although most PyPedal users will only work directly with one or two of these objects it is worthwhile to know a little about each 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.

4.3 Program Structure

PyPedal programs load pedigrees from files and operate on those pedigrees. A program consists of four basic parts: a header, an options section, pedigree creation, and pedigree operations. The program header is used to import modules used in that program, and may include any Python module available on your system. You must import a module before you can use it:

```
# Program header -- load modules used by a program
from PyPedal import pyp_newclasses
from PyPedal import  pyp_metrics
```

You should only import modules that you are going to use in your program; you do not need to import every PyPedal module in every program you write.

PyPedal recognizes a number of diffferent options that are used to control its behavior (Section 4.4). Before you can load your pedigree into a PyPedal object you must provide a pedigree file name (‘`pedname`’) and a pedigree format string (‘`pedformat`’). This is done by creating a Python dictionary and passing it as a parameter when `pyp_newclasses.NewPedigree()` is called.

```
options = {}
options['messages'] = 'verbose'
options['renumber'] = 0
options['pedfile'] = 'new_lacy.ped'
options['pedformat'] = 'asd'
options['pedname'] = 'Lacy (1989) Pedigree'
```

You may name your dictionary whatever you like; the examples in this manual, as well as those distributed with PyPedal, use the name ‘`options`’.

Once you have defined your options to is time to load your pedigree file. This is a two-step process that involves creating an instance of a `NewPedigree` object and then loading the pedigree file into that object:

```
example = pyp_newclasses.NewPedigree(options)
example.load()
```

If you really despise having to wrte two lines of code to load your pedigree file, you can easily create a simple procedure that does it for you in a single step:

```
def customLoadPedigree(options):
    try:
        _pedigree = pyp_newclasses.NewPedigree(options)
        _pedigree.load()
        return  _pedigree
    except:
        return 0
```

Once you have loaded your pedigree file into a `NewPedigree` object you can unleash the awesome power of a fully-functional PyPedal installation on it. For example, calculating the effective number of founders in your pedigree using Lacy’s (1989) exact method is as simple as:

```
pyp_metrics.effective_founders_lacy(example)
```

Example programs that demonstrate how to use many of the features of PyPedal are included in the ‘`examples`’ directory of the distribution.

4.4 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 4.4. 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 4.5.1) 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 provide 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 be changed at all.

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.

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* CANNOT 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.
default_report	filetag	Default report name for use by `pyp_reports`.
default_unit	'inch'	The default unit of measurement for report generation ('cm'—'inch').
debug_messages	0	Indicates whether or not PyPedal should print debugging information.
f_computed	0	Indicates whether or not coefficients of inbreeding have been computed for animals in the current pedigree. If the pedigree format string includes '`f`' this will be set to 1; it is also set to 1 on a successful return from `pyp_nrm/inbreeding()`.
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	*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.
gen_coeff	0	When nonzero, calculate generation coefficients using the method of Pattie (1965) and store them in the `gen_coeff` attribute of a `NewAnimal` object. The inferred generation stored in the `igen` attribute will be the `gen_coeff` rounded to the nearest 0.5. When zero, the `gen_coeff` is -999.
log_long_filenames	0	When nonzero, long logfile names will be used, which means that log file names will include datestamps.
log_ped_lines	0	When $>$ 0 indicates how many lines read from the pedigree file should be printed in the log file for debugging purposes.
logfile	filetag.log	The name of the file to which PyPedal should write messages about its progress.
messages	'verbose'	How chatty PyPedal should 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_bdate	'01011900'	Default birth date.
missing_byear	1900	Default birth year.
missing_parent	'0'	Indicates what code is used to identify missing/unknown parents in 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.

paper_size	'letter'	Default paper size for printed reports ('A4'—'letter').
pedfile	None	File from which pedigree is read; must provide.
pedformat	'asd'	See 4.5.1 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).
set_offspring	0	Assigns offspring to their parent(s)'s unknown sex offspring list.
set_sexes	0	Iterate over the pedigree to assign sexes to all animals in the pedigree (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.

4.5 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 delimiter , 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 4.4.

4.5.1 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 code than '`A`'. The codes currently recognized by PyPedal are listed in Table 4.2.

As noted, all pedigrees must contain columns corresponding to animals, sires, and dams, either in the 'asd' or 'ASD' formats (it is not recommended that you mix them such as in 'AsD'). 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. The herd column type simply refers to a management group identifier, and can correspond to a herd, flock, litter, etc.

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 it.

Note that if your pedigree file uses strings for animal, sire, and dam IDs (the ASD pedigree format codes) you may need to override the *missing_parent* option, which is '`0`' by default. For example, the pedigree file shown in Figure **??** uses *animal0* to denote unknown parents. If '`options['missing_parent'] = 'animal0'`' is not set before the pedigree file is loaded missing parents will be treated as animals with unknown parents, rather than as unknown parents.

4.6 Renumbering a Pedigree

Whenever you load a pedigree into PyPedal a list of offspring is attached to the record for each animal in the pedigree file. If you renumber the pedigree at the time it is loaded, there is no problem. However, if you do not renumber a pedigree at load time and choose to renumber it later in your session you must be careful. The API documentation may lead you to believe that

```
example.pedigree = pyp_utils.renumber()
```

is the correct way to renumber the pedigree, but that is not correct. The pedigree should always be numbered as:

Table 4.2: Pedigree format codes.

Code	Description
a	animal ('a' or 'A' REQUIRED)
s	sire ('s' or 'S' REQUIRED)
d	dam ('d' or 'D' REQUIRED)
b	birthyear (YYYY)
e	age
f	coefficient of inbreeding
g	generation
h	herd
l	alive (1) or dead (0)
n	name
p	Pattie's (Pattie 1965) generation coefficient
r	breed
y	birthdate in "MMDDYYYY" format
x	sex
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`')
H	herd as a string (cannot contain '`sepchar`')
L	alleles (two alleles separated by a non-null character)

```
example.kw['renumber'] = 1
example.renumber()
```

If you are seeing strange results when trying to cross-reference offspring to their parents check to make sure that you have not incorrectly your pedigree.

4.7 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 Boicherd et al. (1997).
Fri, 06 May 2005 10:27:22 INFO     Pedigree comment (line 2): # It contains two
                                   unrelated families.
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 or unusual has happened, although you might argue that coming to the end of an input file is neither. 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 are written to the log, as are deprecated pedigree format strings 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 string.

```
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 and Scharloo (1981) as presented
Thu, 04 Aug 2005 15:36:18 INFO     Pedigree comment (line 2): # in Hartl and Clark
                                   (1989), p. 242.
Thu, 04 Aug 2005 15:36:18 ERROR    The record on line 3 of file hartlandclark.ped
                                   does not have the same number of columns (4) as
                                   the pedigree format string (asd) says that it
                                   should (3). Please check your pedigree file and
                                   the pedigree format string for errors.
```

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.

CHAPTER

FIVE

Methodology

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.

5.1 Reordering and Renumbering

Many computations on pedigrees require that the pedigree be renumbered such that animal IDs are consecutive from 1 to ‘`n`’, where ‘`n`’ is the total number of animalsin the pedigree. The renumbering process requires that the pedigree be reordered such that parents always precede their offspring in the list of animal IDs. The actual ID assigned to an animal is of no particular importance, and it is even possible for parents to have larger IDs than their ofspring. PyPedal can reorder any pedigree unless there is an error in it that would prevent unambiguously placing parents before offspring. For example, a pedigree containing a keypunch error such that an animal is one of its own grandparents cannot be reordered because there is no way to unambiguously order the animals. The `pyp_utils` module provides two routines for pedigree reordering, `reorder()` and `fast_reorder()`. By default, `reorder()` is used to reorder pedigrees in place. It does this by maintaining a list of animal IDs that have been processed; whenever a parent that is not in the list of encountered animals the offspring of that parent are moved to the end of the pedigree. This ensures the pedigree is properly sorted such that all parents precede their offspring. This procedure will always correctly reorder a pedigree but it can be quite inefficient as it is similar to an insertion sort, which has a worst-case runtime proportional to n^2 (Cormen, Leiserson, Rivest, and Stein 2003).

`fast_reorder()` provides a much faster means of reordering a pedigree, but can incorrectly reorder a pedigree in some cases. When an instance of a `NewAnimal` object is created the `pad_id()` method is called. `pad_id()` uses the animal ID and birth year to form an ID used by `by pyp_utils/fast_reorder()` for quick sorting; if your pedigree file is numbered such that offspring always have larger IDs than their parents and your birth years (if provided) are correct (that is, parents always born BEFORE offspring) then `pyp_utils.fast_reorder()` works as expected. If you do not provide birth years in your pedigree file but your parent IDs are always smaller than your animal IDs, the reordering will be correct. If you do not provide birth years, all animals in the pedigree will be assigned a default value of ‘1900’. In that case, if parents have IDs larger than that of one or more of their offspring, the pedigree will be incorrecrly reordered by `fast_reorder()`. If your pedigree file contains birth years, or you know that parents always have smaller IDs than their offspring, then `fast_reorder()` will correctly reorder your pedigree in linear time.

The performance difference between the two reordering routines is not very noticeable on pedigrees of a few hundred to a few thousand animals, but is quite dramatic for very large pedigrees. If your pedigree file is already reordered then there is essentially no performance difference between the two. When creating a pedigree file from data stored in a relational database, let the database perform the sort for you by using an ‘`ORDER BY`’ statement.

5.2 Measures of Genetic Variation

Coefficients of inbreeding and relationship (Wright 1922) have been commonly used to describe the genetic diversity in livestock populations (Young and Seykora 1996). Inbreeding coefficients represent an individual's expected genetic homozygosity due to the relatedness of its parents. Coefficients of relationship describe the expected proportion of genes two individuals share due to their relatedness. These are relative measures that depend on such factors as the completeness and depth of pedigrees. Over time, these coefficients change in response to breeding and culling decisions, and they may be used as indicators of the genetic variability of a population. Rapid methods for calculating coefficients of inbreeding and relationship for large populations have been implemented (Wiggans, Van Raden, and Zuurbier 1995).

Populations under study rarely conform to the theory established for the use of coefficients of inbreeding (Wright 1931). Lacy (1989) and Boichard et al. (1997) proposed measures of genetic variation based on ideas from conservation genetics. Lacy (1989) proposed the idea of the number of founder equivalents in assessing populations. A founder is an ancestor whose parents are unknown. If all founders contribute to the population equally, then the founder equivalent is equal to the number of founders. When founders contribute unequally to the population, the number of founder equivalents decreases. Boichard et al. (1997) developed the idea of founder ancestor equivalents, which is the minimum number of ancestors necessary to explain the genetic diversity of the current population. Founder ancestor equivalents account for bottlenecks, unlike founder equivalents, and are more accurate in populations undergoing intense selection. Caballero and Toro (2000) discussed the relationships among these and other measures of diversity in small populations, and demonstrate their use (Toro, Rodriganez, Silio., and Rodriguez 2000).

Roughsedge et al. (1999) used average coefficients of inbreeding, average coefficients of relationship, founder equivalent numbers, and founder ancestor numbers to document the decrease in genetic diversity in the British dairy cattle population over the last 25 years. Changes in founder equivalent number and founder ancestor number reflected the use of a small number of influential individuals to improve the genetic merit of that population. Accompanying changes in average inbreeding and relationship did not accurately reflect that loss of diversity. Such results highlight the need for additional tools when assessing complex populations.

5.3 Computational Details

5.3.1 Inbreeding and Related Measures

Coefficients of relationship and inbreeding are calculated using the method of Wiggans et al. (1995). An empty dictionary is created to store animal IDs and coefficients of inbreeding. For each animal in the pedigree, working from youngest to oldest, the dictionary is queried for the animal ID. If the animal does not have an entry in the dictonary, a subpedigree containing only relatives of that animal is extracted and the coefficients of inbreeding are calculated and stored in the dictionary. A second dictionary keeps track of sire-dam combinations seen in the pedigree. If a full-sib of an animal whose pedigree has already been processed is encountered the full-sib receives a COI identical to that of the animal already processed. This approach allows for computation of COI for arbitrarily large populations because it does not require allocation of a single NRM of order n^2, where n is the size of the pedigreed population. In most cases, the NRM for a subpedigree is on the order of 200, although this can vary with species and population data structure.

Average and maximum coefficients of inbreeding are computed for the entire population and for all individuals with non-zero inbreeding. The average relationship among all individuals is also computed. Theoretical and realized effective population sizes, $N_{e(t)}$, and $N_{e(r)}$, were estimated as (Falconer and MacKay 1996):

$$N_{e(t)} = \frac{4N_m N_f}{N_m + N_f}$$

and

$$N_{e(t)} = \frac{1}{2\Delta f}$$

where N_m and N_f are the number of sires and dams in the population, respectively, and Δf is the change in population average inbreeding between generations *t* and *t+1*. Interpretation of $N_{e(t)}$ can be problematic when Δf is calculated from incomplete or error-prone pedigrees.

5.3.2 Generation Coefficients

Generation coefficients are assigned using the method of (Pattie 1965). Founders, defined as individuals with unknown parents, are assigned generation codes of 0. All other animals are assigned generation codes as:

$$GC_o = \frac{(GC_s + GC_d)}{2} + 1$$

where GC_o, GC_s, GC_d represent offspring, sire, and dam codes, respectively.

5.3.3 Effective Founder Number

The effective founder number (f_e) was calculated as:

$$f_e = \frac{1}{\sum p_i^2}$$

where p_i is the proportion of genes contributed by ancestor i to the current population (Lacy 1989). If all founders had contributed equally to the population, then f_e would be the same as the actual number of founders. When founders contribute to the population unequally, f_e is smaller than the actual number of founders. The greater the inequity in founder contributions, the smaller the effective founder number.

A subpedigree approach, similar to that used for calculation of inbreeding (see 5.3.1 for details), is also used for calculating f_e.

5.3.4 Founder Genome Equivalents

Lacy (1989) also defined the number of founder genome equivalents (f_g) as a measure of genetic diversity. A founder genome equivalent is the number of founders that would produce a population with the same diversity of founder alleles as the pedigree population assuming all founders contributed equally to each generation of descendants. Founder genome equivalents are calculated as:

$$f_g = \frac{1}{\sum \frac{p_i}{r_i}}$$

where p_i is the proportion of genes contributed by ancestor *i* to the current population and r_i is the proportion of founder *i*'s genes that are retained in the current population. Like f_e, f_g accounts for unequal founder contributions. Unlike f_e, f_g also accounts for the fraction of founder genomes lost from the pedigree through drift during bottlenecks. Although f_g is the more accurate description of the amount of founder variation present in a population, it can only be calculated directly for simple pedigrees. For large or complex pedigrees, the number of founder genome equivalents must be approximated based on computer simulation of a large number of segregations through the pedigree. This is done by assigning each founder a unique pair of alleles and randomly transmitting those alleles through the pedigree (MacCluer, VandeBerg, Read, and Ryder 1986). The number of founder genome equivalents is similar to the effective founder number, but the former has been devalued based on the proportion of its genome that has probably been lost to drift (Lacy 1989).

5.3.5 Effective Ancestor Number

In populations that have undergone a bottleneck the effective number of founders computed using Lacy's (1989) approach is overestimated. Large contributions made by recent ancestors are more important to the population with respect to the loss of genetic diversity than equal contributions made long ago. Boichard et al. (1997) proposed a second measure of diversity to deal with such situations, the effective number of ancestors (f_a), which considers the genetic contribution of all ancestors in the population, not just founders. The effective number of ancestors treats all ancestors in the population the same way, and is computed as:

$$f_a = \frac{1}{\sum q_i^2}$$

where q_i is the genetic contribution of the *i*th ancestor not explained by the previous *i*-1 ancestors. The ancestors with the greatest contributions are selected iteratively. The number of ancestors with a positive genetic contribution is less than or equal to the actual number of founders.

5.3.6 Pedigree Completeness

Pedigree completeness (Cassell, Adamec, and Pearson 2003), the proportion of known pedigree information for an arbitrary number of generations, is computed as:

$$c_p = \frac{a_k}{\sum_{i=1}^{g} 2^i}$$

where c_p is pedigree completeness and a_k is the number of known ancestors in *g* generations. The default (which may be overridden) is to compute four-generation pedigree completeness. Low c_p indicates that there is little pedigree information available for an individual, which may result in biased estimates of inbreeding and other measures of diversity.

CHAPTER

SIX

HOWTOs

In this chapter, examples of common operations are presented.

6.1 Basic Tasks

6.1.1 How do I load a pedigree from a file?

Each pedigree that you read must be passed its own dictionary of options that must have at least a pedigree file name (*pedfile*) and a pedigree format string (*pedformat*). You then call `pyp_newclasses.NewPedigree()` and pass the options dictionary as an argument. The following code fragment demonstrates how to read a pedigree file:

```
options = {}
options['pedfile'] = 'new_lacy.ped'
options['pedformat'] = 'asd'

example1 = pyp_newclasses.NewPedigree(options)
example1.load()
```

The options dictionary may be named anything you like. In this manual, and in the example programs distributed with PyPedal, *options* is the name used.

6.1.2 How do I load multiple pedigrees in one program?

A PyPedal program can load more than one pedigree at a time. Each pedigree must be passed its own options dictionary, and the pedigrees must have different names. This is easily done by creating a dictionary with global options and customizing it for each pedigree. Once you have created a pedigree by calling `pyp_-newclasses.NewPedigree('options')` you can change the options dictionary without affecting that pedigree (a pedigree stores a copy of the options dictionary in its `kw` attribute). The following code fragment demonstrates how to read two pedigree files in a single program:

```
#   Create the empty options dictionary
options = {}

#   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. You do not even have to change *pedfile*.

6.1.3 How do I renumber a pedigree?

Set the `renumber` option to '`1`' before you load the pedigree.

```
options = {}
options['renumber'] = 1
options['pedfile'] = 'new_lacy.ped'
options['pedformat'] = 'asd'
if __name__ == '__main__':
    example1 = pyp_newclasses.NewPedigree(options)
    example1.load()
```

If you do not renumber a pedigree at load time and choose to renumber it later you must set the `renumber` option and call the pedigree's `renumber()` method:

```
example.kw['renumber'] = 1
example.renumber()
```

For more details on pedigree renumbering see Section 4.6.

6.1.4 How do I turn off output messages?

You may want to suppress the output that is normally written to STDOUT by scripts. You do this by setting the `messages` option:

```
options['messages'] = 'quiet'
```

The default setting for `messages` is '`verbose`', which produces lots of messages.

6.2 Calculating Measures of Genetic Variation

6.2.1 How do I calculate coefficients of inbreeding?

This requires that you have a renumbered pedigree (HOWTO 6.1.3).

```
options = {}
options['renumber'] = 1
options['pedfile'] = 'new_lacy.ped'
options['pedformat'] = 'asd'
example1 = pyp_newclasses.NewPedigree(options)
example1.load()
example_inbreeding = pyp_nrm.inbreeding(example)
print example_inbreeding
```

The dictionary returned by `pyp_nrm.inbreeding(example)`, *example_inbreeding*, contains two dictionaries: *fx* contains coefficients of inbreeding (COI) keyed to renumbered animal IDs and *metadata* contains summary statistics. *metadata* also contains two dictionaries: *all* contains summary statistics for all animals, while *nonzero* contains summary statistics for only animals with non-zero coefficients of inbreeding. If you print *example_inbreeding* you'll get the following:

```
{'fx': {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: 0.0, 13: 0.0, 14: 0.0, 15: 0.0, 16: 0.0, 17: 0.0, 18: 0.0,
19: 0.0, 20: 0.0, 21: 0.0, 22: 0.0, 23: 0.0, 24: 0.0, 25: 0.0, 26: 0.0, 27: 0.0,
28: 0.25, 29: 0.0, 30: 0.0, 31: 0.25, 32: 0.0, 33: 0.0, 34: 0.0, 35: 0.0, 36: 0.0,
37: 0.0, 38: 0.21875, 39: 0.0, 40: 0.0625, 41: 0.0, 42: 0.0, 43: 0.03125, 44: 0.0,
45: 0.0, 46: 0.0, 47: 0.0},
'metadata': {'nonzero': {'f_max': 0.25, 'f_avg': 0.16250000000000001,
'f_rng': 0.21875, 'f_sum': 0.8125, 'f_min': 0.03125, 'f_count': 5},
'all': {'f_max': 0.25, 'f_avg': 0.017287234042553192, 'f_rng': 0.25,
'f_sum': 0.8125, 'f_min': 0.0, 'f_count': 47}}}
```

Obtaining the COI for a given animal, say 28, is simple:

```
>>> print example_inbreeding['fx'][28]
'0.25'
```

To print the mean COI for the pedigree:

```
>>> print example_inbreeding['metadata']['all']['f_avg']
'0.017287234042553192'
```

6.3 Databases and Report Generation

6.3.1 How do I load a pedigree into a database?

The `pyp_reports` module (10.9) uses the `pyp_db` module (Section 10.2) to store and manipulate a pedigree in an SQLite database. In order to use these tools you must first load your pedigree into the database. This is done with a call to `pyp_db.loadPedigreeTable()`:

```
options = {}
options['pedfile'] = 'hartlandclark.ped'
options['pedname'] = 'Pedigree from van Noordwijck and Scharloo (1981)'
options['pedformat'] = 'asdb'

example = pyp_newclasses.NewPedigree(options)
example.load()

pyp_nrm.inbreeding(example)
pyp_db.loadPedigreeTable(example)
```

The routines in `pyp_reports` will check to see if your pedigree has already been loaded; if it has not, a table will be created and populated for you.

6.3.2 How do I update a pedigree in the database?

Changes to a PyPedal pedigree object are not automatically saved to the database. If you have changed your pedigree, such as by calculating coefficients of inbreeding, and you want those changes visible to the database you have to call `pyp_db.loadPedigreeTable()` again. **IMPORTANT NOTE:** If you call `pyp_-db.loadPedigreeTable()` after you have already loaded your pedigree into the database it will drop the existing table and reload it; all data in the existing table will be lost! In the following example, the pedigree is written to table **hartlandclark** in the database **pypedal**:

```
options = {}
options['pedfile'] = 'hartlandclark.ped'
options['pedname'] = 'Pedigree from van Noordwijck and Scharloo (1981)'
options['pedformat'] = 'asdb'

example = pyp_newclasses.NewPedigree(options)
example.load()

pyp_db.loadPedigreeTable(example)
```

`pypedal` is the default database name used by PyPedal, and can be changed using a pedigree's `database_-name` option. By default, table names are formed from the pedigree file name. A table name can be specified using a pedigree's `dbtable_name` option. Continuing the above example, suppose that I calculated coefficients of inbreeding on my pedigree and want to store the resulting pedigree in a new table named *noordwijck_and_scharloo_-inbreeding*:

```
options['dbtable_name'] = 'noordwijck_and_scharloo_inbreeding'
pyp_nrm.inbreeding(example)
pyp_db.loadPedigreeTable(example)
```

You should see messages in the log telling you that the table has been created and populated:

```
Tue, 29 Nov 2005 11:24:22 WARNING  Table noordwijck_and_scharloo_inbreeding does
                                   not exist in database pypedal!
Tue, 29 Nov 2005 11:24:22 INFO     Table noordwijck_and_scharloo_inbreeding
                                   created in database pypedal!
```

6.4 Contribute a HOWTO

Users are invited to contribute HOWTOs demonstrating how to solve problems they've found interesting. In order for such HOWTOs to be considered for inclusion in this manual they must be licensed under the GNU Free Documentation License version 1.2 or later (`http://www.gnu.org/copyleft/fdl.html`). Authorship will be acknowledged, and copyright will remain with the author of the HOWTO.

CHAPTER

SEVEN

Graphics

This chapter presents an overview of using the graphics routines PyPedal.

7.1 PyPedal Graphics

PyPedal is capable of producing graphics from information contained in a pedigree, including pedigree drawings, line graphs of changes in genetic diversity over time, and visualizations of numerator relationship matrices. These graphics are non-interactive: output images are created and written to output files. A separate program must be used to view and/or print the image; web browsers make reasonably good viewers for a small number of images. If you are creating and viewing large numbers of images you may want to obtain an image management package for your platform. Default and supported file formats for each of the graphics routines are presented in Table 7.1.

7.1.1 Drawing Pedigrees

The pedigree from Figure 2 in Boichard et al. (1997) is shown in Figure 7.1, and shows males enclosed in rectangles and females in ovals. Figure 7.2 shows a pedigree in which strings are used for animal IDs; animal are enclosed in ovals because sexes were not specified in the pedigree file and the `set_sexes` option was not specified. A more complex German Shepherd pedigree is presented in Figure 7.3; the code used to create this pedigree is:

Table 7.1: Default graphics formats.

Routine	Default Format	Supported Formats
draw_pedigree	JPG	JPG, PNG, PS
pcolor_matrix_pylab	PNG	PNG only
plot_founders_by_year	PNG	PNG only
plot_founders_pct_by_year	PNG	PNG only
plot_line_xy	PNG	PNG only
rmuller_pcolor_matrix_pil	PNG	PNG only
rmuller_spy_matrix_pil	PNG	PNG only
spy_matrix_pylab	PNG	PNG only

```
pyp_graphics.draw_pedigree(example, gfilename='doug_p_rl_notitle', gname=1,
    gdirec='RL', gfontsize=12)
```

The resulting graphic is written to doug_p_rl_notitle.jpg; note from Table 7.1 that the default file format for `draw_pedigree()` is **JPG** rather than **PNG**, as is the case for the other graphics routines. To get a PNG simply pass the argument *gformat='png'* to `draw_pedigree()`. For details on the options taken by `draw_pedigree()` please refer to the API documentation (Section 10.4).

7.1.2 Drawing Line Graphs

The `plot_line_xy()` routine provides a convenient tool for creating two-dimensional line graphs. Figure 7.4 shows the plot of inbreeding by birth year for the US Ayrshire cattle population. The plot is produced by the call:

```
pyp_db.loadPedigreeTable(ay)
coi_by_year = pyp_reports.meanMetricBy(ay,metric='fa',byvar='by')
cby = coi_by_year
del(cby[1900])
pyp_graphics.plot_line_xy(coi_by_year, gfilename='ay_coi_by_year',
    gtitle='Inbreeding coefficients for Ayrshire cows', gxlabel='Birth year',
    gylabel='Coefficient of inbreeding')
```

The code above uses `pyp_reports.meanMetricBy()` (see 10.9) to populate *coi_by_year*; the keys in *coi_by_year* are plotted in the x-axis, and the values are plotted on the y-axis. The default birth year, 1900, was deleted from the dictionary before the plot was drawn because leaving the default birthyear in the plot was distracting and somewhat misleading. The only restriction that you have to observe is that the value plotted on the y-ais has to be a numeric quantity.

If you need more complicated plots than are produced by `plot_line_xy()` you can write a new plotting function (Chapter 9) that uses the tools in matplotlib (`http://matplotlib.sourceforge.net/`). For complete details on the options taken by `plot_line_xy` please refer to the API documentation (10.4).

7.1.3 Visualizing Numerator Relationship Matrices

Two routines are provided for visualization of numerator relationship matrices (NRM), `rmuller_pcolor_matrix_pil()` and `rmuller_spy_matrix_pil()`.

As an example, we will consider the NRM for the pedigree in Figure 7.1. The matrix is square and symmetric; the diagonal values correspond to $1 + f_a$, where f_a is an animal's coefficient of inbreeding; animals with a diagonal

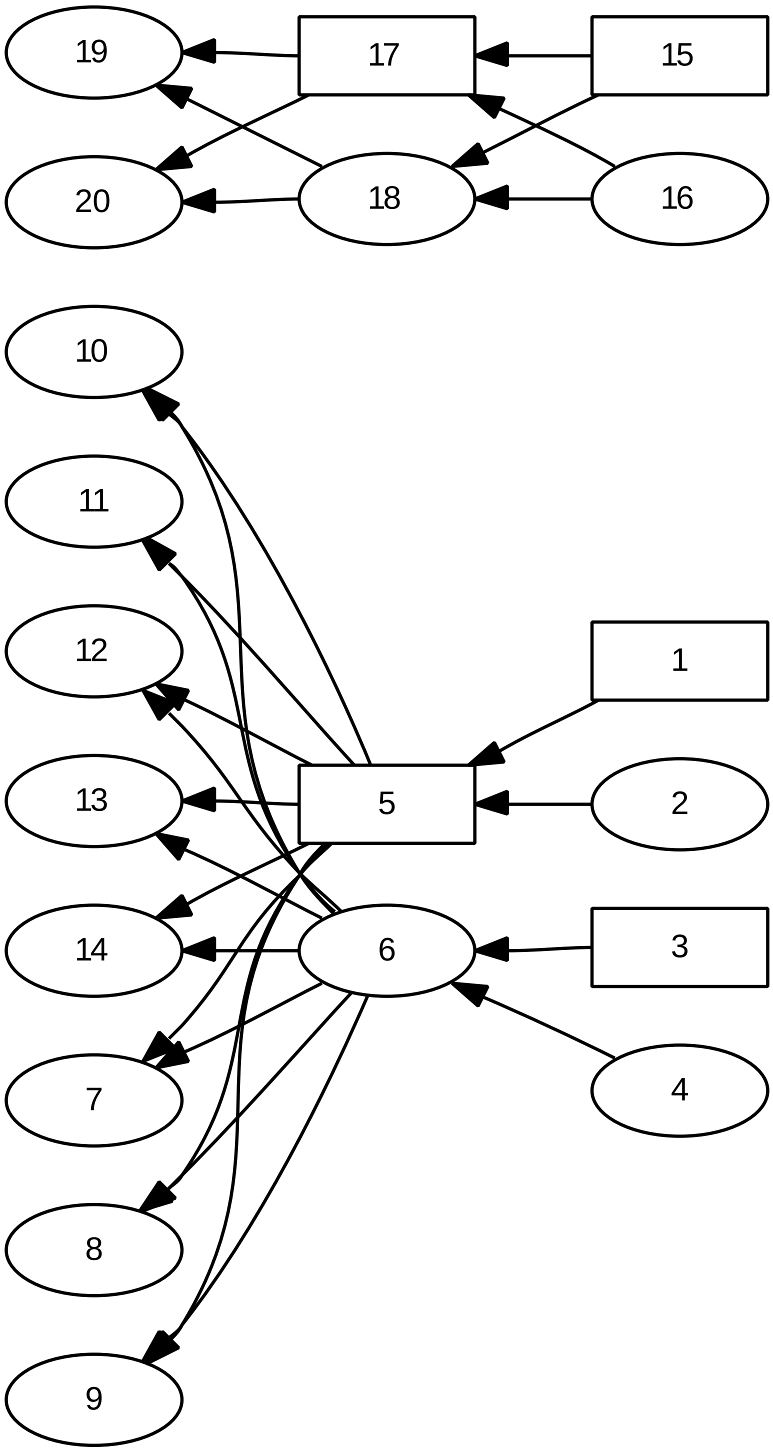

Figure 7.1: Pedigree 2 from Boichard et al. (1997)

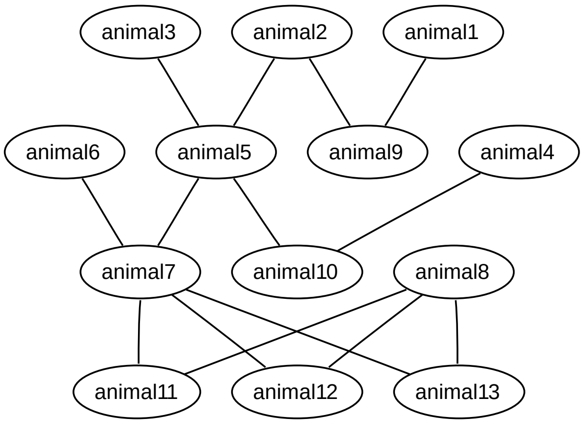

Figure 7.2: A pedigree with strings as animal IDs

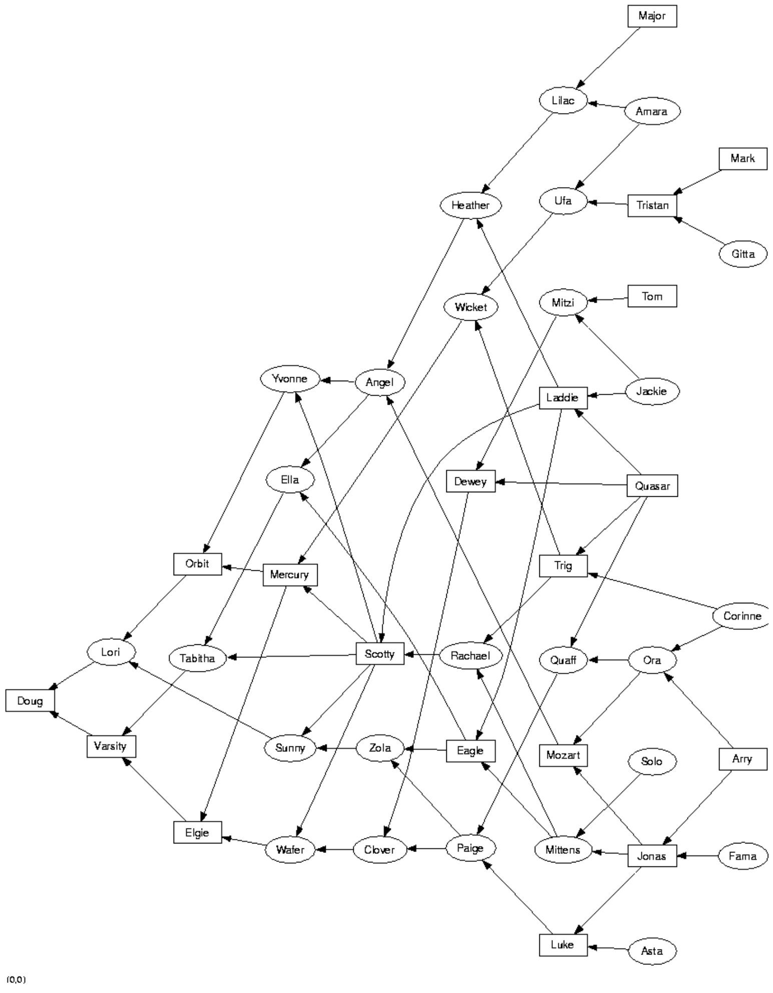

Figure 7.3: German Shepherd pedigree

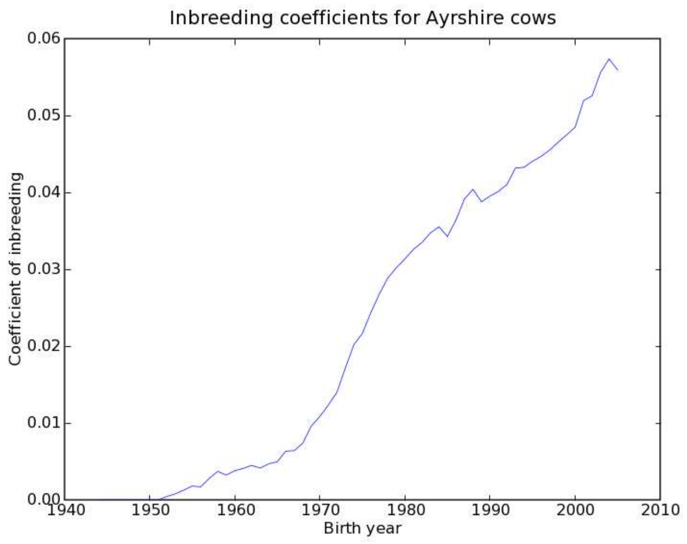

Figure 7.4: Average inbreeding by birth year for the US Ayrshire cattle population

element > 1 are inbred.

$$\begin{bmatrix}
1. & 0. & 0. & 0. & 0.5 & 0. & 0.25 & 0.25 & 0.25 & 0.25 & 0.25 & 0.25 & 0.25 & 0.25 & 0. & 0. & 0. & 0. & 0. & 0. \\
0. & 1. & 0. & 0. & 0.5 & 0. & 0.25 & 0.25 & 0.25 & 0.25 & 0.25 & 0.25 & 0.25 & 0.25 & 0. & 0. & 0. & 0. & 0. & 0. \\
0. & 0. & 1. & 0. & 0. & 0.5 & 0.25 & 0.25 & 0.25 & 0.25 & 0.25 & 0.25 & 0.25 & 0.25 & 0. & 0. & 0. & 0. & 0. & 0. \\
0. & 0. & 0. & 1. & 0. & 0.5 & 0.25 & 0.25 & 0.25 & 0.25 & 0.25 & 0.25 & 0.25 & 0.25 & 0. & 0. & 0. & 0. & 0. & 0. \\
0.5 & 0.5 & 0. & 0. & 1. & 0. & 0.5 & 0.5 & 0.5 & 0.5 & 0.5 & 0.5 & 0.5 & 0.5 & 0. & 0. & 0. & 0. & 0. & 0. \\
0. & 0. & 0.5 & 0.5 & 0. & 1. & 0.5 & 0.5 & 0.5 & 0.5 & 0.5 & 0.5 & 0.5 & 0.5 & 0. & 0. & 0. & 0. & 0. & 0. \\
0.25 & 0.25 & 0.25 & 0.25 & 0.5 & 0.5 & 1. & 0.5 & 0.5 & 0.5 & 0.5 & 0.5 & 0.5 & 0.5 & 0. & 0. & 0. & 0. & 0. & 0. \\
0.25 & 0.25 & 0.25 & 0.25 & 0.5 & 0.5 & 0.5 & 1. & 0.5 & 0.5 & 0.5 & 0.5 & 0.5 & 0.5 & 0. & 0. & 0. & 0. & 0. & 0. \\
0.25 & 0.25 & 0.25 & 0.25 & 0.5 & 0.5 & 0.5 & 0.5 & 1. & 0.5 & 0.5 & 0.5 & 0.5 & 0.5 & 0. & 0. & 0. & 0. & 0. & 0. \\
0.25 & 0.25 & 0.25 & 0.25 & 0.5 & 0.5 & 0.5 & 0.5 & 0.5 & 1. & 0.5 & 0.5 & 0.5 & 0.5 & 0. & 0. & 0. & 0. & 0. & 0. \\
0.25 & 0.25 & 0.25 & 0.25 & 0.5 & 0.5 & 0.5 & 0.5 & 0.5 & 0.5 & 1. & 0.5 & 0.5 & 0.5 & 0. & 0. & 0. & 0. & 0. & 0. \\
0.25 & 0.25 & 0.25 & 0.25 & 0.5 & 0.5 & 0.5 & 0.5 & 0.5 & 0.5 & 0.5 & 1. & 0.5 & 0.5 & 0. & 0. & 0. & 0. & 0. & 0. \\
0.25 & 0.25 & 0.25 & 0.25 & 0.5 & 0.5 & 0.5 & 0.5 & 0.5 & 0.5 & 0.5 & 0.5 & 1. & 0.5 & 0. & 0. & 0. & 0. & 0. & 0. \\
0.25 & 0.25 & 0.25 & 0.25 & 0.5 & 0.5 & 0.5 & 0.5 & 0.5 & 0.5 & 0.5 & 0.5 & 0.5 & 1. & 0. & 0. & 0. & 0. & 0. & 0. \\
0. & 0. & 0. & 0. & 0. & 0. & 0. & 0. & 0. & 0. & 0. & 0. & 0. & 0. & 1. & 0. & 0.5 & 0.5 & 0.5 & 0.5 \\
0. & 0. & 0. & 0. & 0. & 0. & 0. & 0. & 0. & 0. & 0. & 0. & 0. & 0. & 0. & 1. & 0.5 & 0.5 & 0.5 & 0.5 \\
0. & 0. & 0. & 0. & 0. & 0. & 0. & 0. & 0. & 0. & 0. & 0. & 0. & 0. & 0.5 & 0.5 & 1. & 0.5 & 0.75 & 0.75 \\
0. & 0. & 0. & 0. & 0. & 0. & 0. & 0. & 0. & 0. & 0. & 0. & 0. & 0. & 0.5 & 0.5 & 0.5 & 1. & 0.75 & 0.75 \\
0. & 0. & 0. & 0. & 0. & 0. & 0. & 0. & 0. & 0. & 0. & 0. & 0. & 0. & 0.5 & 0.5 & 0.75 & 0.75 & 1.25 & 0.75 \\
0. & 0. & 0. & 0. & 0. & 0. & 0. & 0. & 0. & 0. & 0. & 0. & 0. & 0. & 0.5 & 0.5 & 0.75 & 0.75 & 0.75 & 1.25
\end{bmatrix}$$

Note that the array only contains six distinct values: 0., 0.25, 0.5, 0.75, 1.0, and 1.25. These six values will be used to create the color map used by `rmuller_pcolor_matrix_pil()`.

`rmuller_pcolor_matrix_pil()` produces pseudocolor plots from NRM. A pseudocolor plot is an array of cells that are colored based on the values the corresponding cells in the NRM. The minimum and maximum values in the NRM are assigned the first and last colors in the colormap; other cells are colored by mapping their values to colormap elements. In the example above, the minimum value is 0.0 and the maximum value is 1.0 (Figure 7.5). The two inbred animals in the population are easily identified as the yellow diagonal elements in the bottom-left corner of the matrix. `rmuller_spy_matrix_pil()` is similar to `rmuller_pcolor_matrix_pil()`, but it is used to visualize the sparsity of a matrix. Cells are either filled, indicating that the value is non-zero, or not filled, indicating that the cell's value is zero. In Figure 7.6 it is easy to see the two separate families in the pedigree.

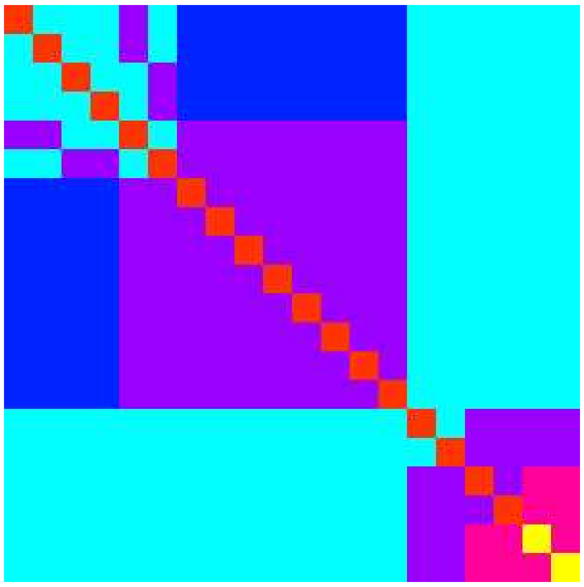
Figure 7.5: Pseudocolored NRM from the Boichard et al. (1997) pedigree

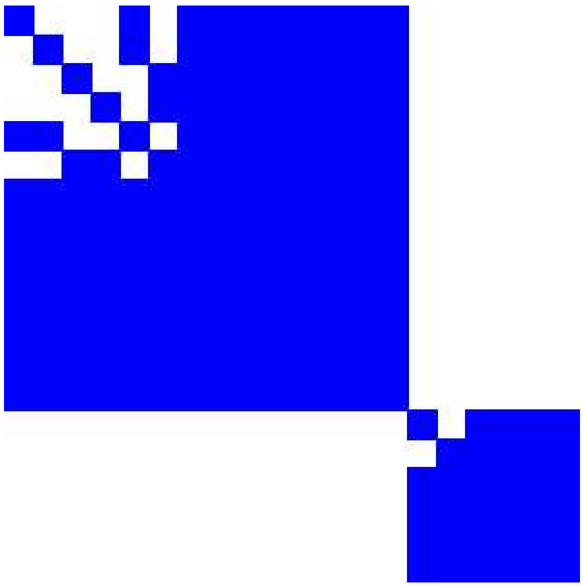
Figure 7.6: Sparsity of the NRM from the Boichard et al. (1997) pedigree

CHAPTER

EIGHT

Report Generation

An overview of the report generation tools in PyPedal is provided in this chapter. The creation of a new, custom report is demonstrated.

8.1 Overview

PyPedal has a framework in place to support basic report generation. This franework consists of two components: a database access module, `pyp_db` (Section 10.2), and a reporting module, `pyp_reports` (Section 10.9). The SQLite 3 database engine (`http://www.sqlite.org/`) is used to store data and generate reports. The ReportLab extension to Python (`http://www.reportlab.org/`) allows users to create reports in the Adobe Portable Document Format (PDF). As a result, there are two types of reports that can be produced: internal summaries that can be fed to other PyPedal routines (e.g. the report produced by `pyp_reports.meanMetricBy()` can be passed to `pyp_graphics.plot_line_xy()` to produce a plot) and printed reports in PDF format. When referencing the `pyp_reports` API note that the convention used in PyPedal is that procedures which produce PDFs are prepended with 'pdf'. Sections 8.2 and 8.3 demonstrate how to create new or custom reports. `pyp_reports` was added to PyPedal with the intention that end-users develop their own custom reports using `pyp_reports.meanMetricBy()` as a template. More material on adding new functionality to PyPedal can be found in Chapter 9.

Column names, data types, and descriptions of contents for pedigree tables are presented in Table 8.1. The `metric_to_column` and `byvar_to_column` dictionaries in `pyp_db` are used to convert between convenient mnemonics and database column names. You may need to refer to Table 8.1 for unmapped column names when writing custom reports. If you happen to view a table scheme using the **sqlite3** command-line utility you will notice that the columns are ordered differently in the database than they are in the table; the table has been alphabetized for easy reference.

8.2 Creating a Custom Internal Report

Internal reports typically aggregate data such that the result can be handed off to another PyPedal routine for further processing. To do this, the pedigree is loaded into a table in an SQLite database against which queries are made. This is faster and more flexible than writing reporting routines that loop over the pedigree to construct reports, but it does require some knowledge of the Structured Query Language (SQL; `http://www.sql.org/`). The canonical example of this kind of report is the passing of the dictionary returned by `pyp_reports.meanMetricBy()` to `pyp_graphics.plot_line_xy()` (see 7.1.1). That approach is outlined in code below.

Table 8.1: Columns in pedigree database tables.

Name	Type	Note(s)
age	real	Age of animal
alive	char(1)	Animal's mortality status
ancestor	char(1)	Ancestor status
animalID	integer	**Must be unique!**
animalName	varchar(128)	Animal name
birthyear	integer	Birth year
breed	text	Breed
coi	real	Coefficient of inbreeding
damID	integer	Dam's ID
founder	char(1)	Founder status
gencoeff	real	Pattie's generation coefficient
generation	real	Generation
herd	integer	Herd ID
infGeneration	real	Inferred generation
num_daus	integer	Number of daughters
num_sons	integer	Number of sons
num_unk	integer	Offspring of unknown sex
originalHerd	varchar(128)	Original herd ID
originalID	text	Animal's original ID
pedgreeComp	real	Pedigree completeness
renumberedID	integer	Animal's renumbered ID
sex	char(1)	Sex of animal
sireID	integer	Sire's ID

```
def inbreedingByYear(pedobj):
    curs = pyp_db.getCursor(pedobj.kw['database_name'])

    # Check and see if the pedigree has already been loaded.  If not, do it.
    if not pyp_db.tableExists(pedobj.kw['database_name'], pedobj.kw['dbtable_name']):
        pyp_db.loadPedigreeTable(pedobj)

    MYQUERY = "SELECT birthyear, pyp_mean(coi) FROM %s GROUP BY birthyear \
        ORDER BY birthyear ASC" % (pedobj.kw['dbtable_name'])
    curs.execute(MYQUERY)
    myresult = curs.fetchall()
    result_dict = {}
    for _mr in myresult:
        _level, _mean = _mr
        result_dict[_level] = _mean
    return result_dict
```

You should always check to see if your pedigree has been loaded into the database before you try and make queries against the pedigree table or your program may crash. `inbreedingByYear()` returns a dictionary containing average coefficients of inbreeding keyed to birth years. The query result, *myresult*, is a list of tuples; each tuple in the list corresponds to one row in an SQL resultset. The tuples in *myresult* are unpacked into temporary variables that are then stored in the dictionary, *result_dict* (for information on tuples see the Python Tutorial (`http://www.python.org/doc/tut/node7.html#SECTION007300000000000000000`). If the resultset is empty, *result_dict* will also be empty. As long as you can write a valid SQL query for the report you'd like to assemble, there is no limitation on the reports that can be prepared by PyPedal.

8.3 Creating a Custom Printed Report

If you are interested in custom printed reports you should begin by opening the file `pyp_reports.py` and reading through the code for the `pdfPedigreeMetadata()` report. It has been heavily commented so that it can be used as a template for developing other reports. ReportLab provides fairly low-level tools that you can use to assemble documents. The basic idea is that you create a canvas on which your image will be drawn. You then create text objects and draw them on the canvas. When your report is assembled you save the canvas on which it's drawn to a file. PyPedal provides a few convenience functions for such commonly-used layouts as title pages and page "frames". In the following sections of code I will discuss the creation of a `pdfInbreedingByYear()` printed report to accompany the `inbreedingByYear()` internal report written in Section 8.2. First, we import ReportLab and check to see if the user provided an output file name. If they didn't, revert to a default.

```
def pdfInbreedingByYear(pedobj,results,titlepage=0,reporttitle='',reportauthor='', \
    reportfile=''):
    import reportlab
    if reportfile == '':
        _pdfOutfile = '%s_inbreeding_by_year.pdf' % ( pedobj.kw['default_report'] )
    else:
        _pdfOutfile = reportfile
```

Next call `_pdfInitialize()`, which returns a dictionary of settings, mostly related to page size and margin locations, that is used throughout the routine. `_pdfInitialize()` uses the `paper_size` keyword in the pedigree's options dictionary, which is either 'letter' or 'A4', and the `default_unit`, which is either 'inch' or 'cm' to populate the returned structure. This should allow users to move between paper sizes without little or no work. Once the PDF settings have been computed we instantiate a canvas object on which to draw.

```
_pdfSettings = _pdfInitialize(pedobj)
canv = canvas.Canvas(_pdfOutfile, invariant=1)
canv.setPageCompression(1)
```

There is a hook in the code to toggle cover pages on and off. It is arguably rather pointless to put a cover page on a one-page document, but all TPS reports require new coversheets. The call to `_pdfDrawPageFrame()` frames the page with a header and footer that includes the pedigree name, date and time the report was created, and the page number.

```
if titlepage:
    if reporttitle == '':
        reporttitle = 'meanMetricBy Report for Pedigree\n%s' \
            % (pedobj.kw['pedname'])
    _pdfCreateTitlePage(canv, _pdfSettings, reporttitle, reportauthor)
_pdfDrawPageFrame(canv, _pdfSettings)
```

The largest chunk of code in `pdfInbreedingByYear()` is dedicated to looping over the input dictionary, *results*, and writing its contents to text objects. If you want to change the typeface for the rendered text, you need to make the appropriate changes to all calls to `canv.setFont("Times-Bold", 12)`. The ReportLab documentation includes a discussion of available typefaces.

```
canv.setFont("Times-Bold", 12)
tx = canv.beginText( _pdfSettings['_pdfCalcs']['_left_margin'],
    _pdfSettings['_pdfCalcs']['_top_margin'] - 0.5 * \
        _pdfSettings['_pdfCalcs']['_unit'] )
```

Every printed report will have a section of code in which the input is processed and written to text objects. In this case, the code loops over the key-and-value pairs in *results*, determines the width of the key, and creates a string with the proper spacing between the key and its value. That string is then written to a `tx.textLine()` object.

```
# This is where the actual content is written to a text object that
# will be displayed on a canvas.
for _k, _v in results.iteritems():
    if len(str(_k)) <= 14:
        _line = '\t%s:\t\t%s' % (_k, _v)
    else:
        _line = '\t%s:\t%s' % (_k, _v)
    tx.textLine(_line)
```

ReportLab's text objects do not automatically paginate themselves. If you write, say, ten pages of material to a text object and render it without manually paginating the object you're going to get a single page of chopped-off text. The following section of code is where the actual pagination occurs, so careful cutting-and-pasting should make pagination seamless.

```
    # Paginate the document if the contents of a textLine are longer than one page.
    if tx.getY() < _pdfSettings['_pdfCalcs']['_bottom_margin'] + \
        0.5 * _pdfSettings['_pdfCalcs']['_unit']:
        canv.drawText(tx)
        canv.showPage()
        _pdfDrawPageFrame(canv, _pdfSettings)
        canv.setFont('Times-Roman', 12)
        tx = canv.beginText( _pdfSettings['_pdfCalcs']['_left_margin'],
            _pdfSettings['_pdfCalcs']['_top_margin'] -
            0.5 * _pdfSettings['_pdfCalcs']['_unit'] )
```

Once we're done writing our text to text objects we need to draw the text object on the canvas and make the canvas visible. If you omit this step, perhaps because of the kind of horrible cutting-and-pasting accident to which I am prone, your PDF will not be written to a file.

```
if tx:
    canv.drawText(tx)
    canv.showPage()
canv.save()
```

While PyPedal does not yet have any standard reports that include graphics, ReportLab does support adding graphics, such as a pedigree drawing, to a canvas. Interested readers should refer to the ReportLab documentation.

CHAPTER

NINE

Implementing New Features

In this chapter, an example of wil be provided of how to extend PyPedal by creating a user-defined routine. New routines may implement a new measure of genetic diversity, extend the graphics module, add a new report, or group a series of actions into a single convenient routine.

9.1 Overview

One of the appealing features of PyPedal is its easy extensibility. In this section, we will demonstrate how to add a user-written module to PyPedal. The file `pyp_template.py` that is distributed with PyPedal is a skeleton that can be used to help you get started writing your custom module(s). You should also look at the source code of the standard modules, particularly if there is already a routine that does something similar to what you would like to do, to see if you can jump-start your project by reusing code.

9.1.1 Defining the Problem

Before you open your editor and begin writing code you need to clearly define your problem. Answering a few questions can help you do this:

- What output do I want from my routine?
- What calculations do I need to perform?
- What input do I need to give my routine in order to perform those calculations?
- Are there any PyPedal routines that already do something similar?

The last question is as important as the others — if there is already a PyPedal routine that does similar calculations you can use it as a starting point. Code reuse is a great idea.

The problem that will motivate the rest of this section sounds very tricky, but is not really so bad because we are going to reuse a lot of code. I want to create a routine for drawing pedigrees that color nodes (animals) based on their importance as measured by their connectedness to other animals in the pedigree. After a brief review of the contents of the Module Template in Section 9.2, I will present a detailed solution to this problem in Section 9.3.

9.2 Module Template

The file ‘pyp_template.py’ is a skeleton that can be used to get started writing a custom module. The first thing you should do is save a copy of ‘pyp_template.py’ with your working module name; we will use the filename ‘pyp_jbc.py’ for the following example. You should also fill-in the module header so that it contains your name, e-mail address, etc. The version number of your module does not have to match that of the main PyPedal distribution, and is only used as an aid to the programmer.

```
###############################################################################
# NAME: pyp_jbc.py
# VERSION: 1.0.0 (16NOVEMBER2005)
# AUTHOR: John B. Cole, PhD (jcole@aipl.arsusda.gov)
# LICENSE: LGPL
###############################################################################
# FUNCTIONS:
#     get_color_32()
#     color_pedigree()
#     draw_colored_pedigree()
###############################################################################
```

The imports section of the template includes `import` statements for all of the standard PyPedal modules. There’s no harm in including all of them in your module, but it’s good practice to include only the modules you need. You should always include the `logging` module because it’s needed for communicating with the log file. For `pyp_jbc` I am including only the `pyp_graphics`, `pyp_network`, and `pyp_utils` modules.

```
##
# pyp_jbc provides tools for enhanced pedigree drawing.
##
import logging
from PyPedal import pyp_graphics
from PyPedal import pyp_network
from PyPedal import pyp_utils
```

There is a very sketchy function prototype included in the template. It is probably enough for you to get started if you have a little experience programming in Python. If you don’t have any experience programming in Python you should be able to get up-and-running with a little trial-and-error and some study of PyPedal source. You should always write a comment block similar to that attached to `yourFunctionName()` for each of your functions. This comment block is recognized by PythonDoc, a tool for automatically generating program documentation. Parameters are the inputs that you send to a function, return is a description of the function’s output, and defreturn is the type of output that is returned, such as a list, dictionary, integer, or tuple.

```
##
# yourFunctionName() <description of what function does>
# @param <parameter_name> <parameter description>
# @return <description of returned value(s)
# @defreturn <type of returned data, e.g., ’dictionary’ or ’list’>
def yourFunctionName(pedobj):
    try:
        # Do something here
        logging.info(’pyp_template/yourFunctionName() did something.’)
        # return a value/dictionary/etc.
    except:
        logging.error(’pyp_template/yourFunctionName() encountered a problem.’)
        return 0
```

9.3 Solving the Problem

The measure of connectedness I am going to use for coloring the pedigree is the proportion of animals in the pedigree that are descended from each animal in the pedigree. In order to do this we need to do the following:

1. Compute the proportion of animals in the pedigree that are descended from each animal in the pedigree; the values will be stored in a dictionary keyed by animal IDs.
2. Map the proportion of descendants from decimal values on the interval (0,1) to RGB triples.
3. Use the RGB triples to set the fill color for nodes.

There is not an existing function for the first item, but there is a function in the `pyp_network` module, `find_-descendants()`, for identifying all of the descendants of an animal. We can use the length of the list of descendants and the number of animals in the pedigree to calculate the proportion of animals in the pedigree descended from that animal. The `color_pedigree()` function creates a dictionary and loops over the pedigree to compute the proporions. It also calls `draw_colored_pedigree()`, which is a modified version of `pyp_graphics.draw_-pedigree()`, to draw the pedigree with colored nodes.

```
##
# color_pedigree() forms a graph object from a pedigree object and
# determines the proportion of animals in a pedigree that are
# descendants of each animal in the pedigree.  The results are used
# to feed draw_colored_pedigree().
# @param pedobj A PyPedal pedigree object.
# @return A 1 for success and a 0 for failure.
# @defreturn integer
def color_pedigree(pedobj):
    _pedgraph = pyp_network.ped_to_graph(pedobj)
    _dprop = {}
    # Walk the pedigree and compute proportion of animals in the
    # pedigree that are descended from each animal.
    for _p in pedobj.pedigree:
        _dcount = pyp_network.find_descendants(_pedgraph,_p.animalID,[])
        if len(_dcount) < 1:
            _dprop[_p.animalID] = 0.0
        else:
            _dprop[_p.animalID] = float(len(_dcount)) / \
                float(pedobj.metadata.num_records)
    del(_pedgraph)
    _gfilename = '%s_colored' % \
        (pyp_utils.string_to_table_name(pedobj.metadata.name))
    draw_colored_pedigree(pedobj, _dprop, gfilename=_gfilename,
        gtitle='Colored Pedigree', gorient='p', gname=1, gdirec='',
        gfontsize=12, garrow=0, gtitloc='b')
```

`pyp_graphics.draw_pedigree()` was copied into `pyp_jbc`, renamed to `draw_colored_pedigree()`, and modified to draw colored nodes. Two basic changes were made to accomplish that: the function was altered to accept a dictionary of weights to be used for coloring, and code for actually coloring the nodes was written. The first change was simply the addition of a new required parameter, *shading*, to the function header. The second step required a little more work. For each animal in the pedigree, the descendant proportion is looked-up in the shading dictionary, the proportion is passed to `get_color_32()` and converted into an RGB triple, and the `filled` and `color` attributes for the node representing that animal are set. The hardest part of creating this routine was determining where changes should be made when modifying `pyp_graphics.draw_pedigree()`.

```
##
# draw_colored_pedigree() uses the pydot bindings to the graphviz library
# 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 animals in the
# pedigree, draw_pedigree() will use subgraphs to try and group animals in
# the same generation together in the drawing.  Nodes will be colored
# based on the number of outgoing connections (number of offspring).
# @param pedobj A PyPedal pedigree object.
# @param shading A dictionary mapping animal IDs to levels that will be
#                used to color nodes.
# ...
# @return A 1 for success and a 0 for failure.
# @defreturn integer
def draw_colored_pedigree(pedobj, shading, gfilename=’pedigree’, \
    gtitle=’My_Pedigree’, gformat=’jpg’, gsize=’f’, gdot=’1’, gorient=’l’, \
    gdirec=’’, gname=0, gfontsize=10, garrow=1, gtitloc=’b’, gtitjust=’c’):

    from pyp_utils import string_to_table_name
    _gtitle = string_to_table_name(gtitle)
    ...
    # If we do not have any generations, we have to draw a less-nice graph.
    if len(gens) <= 1:
        for _m in pedobj.pedigree:
            ...
            _an_node = pydot.Node(_node_name)
            ...
            _color = get_color_32(shading[_m.animalID],0.0,1.0)
            _an_node.set_style(’filled’)
            _an_node.set_color(_color)
            ...
    # Otherwise we can draw a nice graph.
    ...
        ...
            for _m in pedobj.pedigree:
                ...
                _an_node = pydot.Node(_node_name)
                ...
                _color = get_color_32(shading[_m.animalID])
                _an_node.set_style(’filled’)
                _an_node.set_color(_color)
                ...
```

The `get_color_32()` function is a modified version of `pyp_graphics.rmuller_get_color()` that returns RGB triplets of the form ‘`#1a2b3c`’, which are required by the program that renders the graphs. This is another example of how code reuse can reduce development time.

```
##
# get_color_32() Converts a float value to one of a continuous range of colors
# using recipe 9.10 from the Python Cookbook.
# @param a Float value to convert to a color.
# @param cmin Minimum value in array (0.0 by default).
# @param cmax Maximum value in array (1.0 by default).
# @return An RGB triplet.
# @defreturn integer
def get_color_32(a,cmin=0.0,cmax=1.0):
    try:
        a = float(a-cmin)/(cmax-cmin)
    except ZeroDivisionError:
        a=0.5 # cmax == cmin
    blue = min((max((4*(0.75-a),0.)),1.))
    red = min((max((4*(a-0.25),0.)),1.))
    green = min((max((4*math.fabs(a-0.5)-1.,0)),1.))
    _r = '%2x' % int(255*red)
    if _r[0] == ' ':
        _r = '0%s' % _r[1]
    _g = '%2x' % int(255*green)
    if _g[0] == ' ':
        _g = '0%s' % _g[1]
    _b = '%2x' % int(255*blue)
    if _b[0] == ' ':
        _b = '0%s' % _b[1]
    _triple = '#%s%s%s' % (_r,_g,_b)
    return _triple
```

This change will probably be to rolled into `rmuller_get_color()` so that the form of the return triplet is user-selectable.

The program 'new_jbc.py' demonstrates use of the new `pyp_jbc.color_pedigree()` routine:

```
options = {}
options['renumber'] = 1
options['sepchar'] = '\t'
options['missing_parent'] = 'animal0'

if __name__=='__main__':
    options['pedfile'] = 'new_ids2.ped'
    options['pedformat'] = 'ASD'
    options['pedname'] = 'Boichard Pedigree'

    example = pyp_newclasses.NewPedigree(options)
    example.load()
    pyp_jbc.color_pedigree(example)
```

The resulting colorized pedigree can be seen in Figure 9.1. Each of the nodes is colored according to the proportion of animals in the complete pedigree descended from a given animal. Clearly there is still room for improvement; for example, there is no key provided in the image so that you can see how colors map to proportions. Implementation of a key is left as an exercise for the reader.

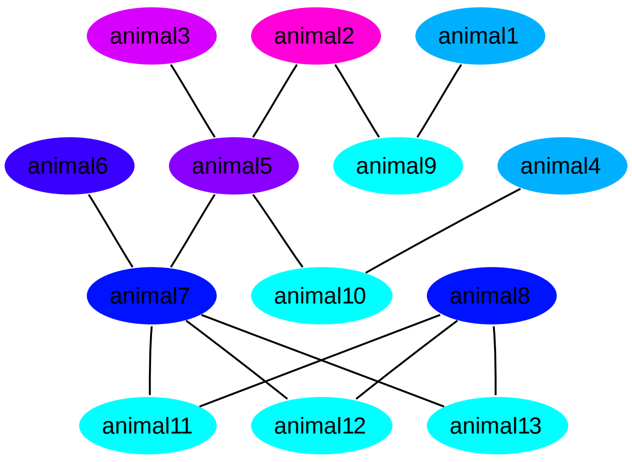

Figure 9.1: Colorized version of the pedigree in Figure 7.2

9.4 Contributing Code to PyPedal

If you would like to contribute your code back to PyPedal please note that it must be licensed under version 2.1 or any later version of the GNU Lesser General Public License. The GNU LGPL has all of the restrictions of the GPL except that you may use the code at compile time without the derivative work becoming a GPL work. This allows the use of the code in proprietary works. You must also complete and return the joint copyright assignment form distributed as `pypedal_copyright_assignment.pdf` before any contributions can be accepted and merged into the development tree.

Contributors are asked to document their code using the documentation comments recognized by PythonDoc 2.0 or later (`http://effbot.org/zone/pythondoc.htm`). PythonDoc is used to generates API documentation in HTML and other formats based on descriptions in Python source files. You are also strongly encouraged to provide example programs abd datasets with any code submissions.

CHAPTER

TEN

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.

10.1 Some Background

A complete list of the core PyPedal() modules is presented in Table 10.1. Using the PyPedal API is quite simple. The following discussion assumes that you have imported each of the Python modules using, e.g., '`from PyPedal import pyp_utils`' rather than '`from PyPedal.pyp_utils import *`'. The latter is poor style and can result in namespace pollution; this is not known to be a problem with PyPedal, but I offer no guarantees that this will remain the case. In order to access a function in the `pyp_utils` module, such as `pyp_nice_time()`, you use a dotted notation with a '.' separating the module name and the function name. For example:

```
[jcole@aipl440 jcole]$ python
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.
>>> from PyPedal import pyp_utils
>>> pyp_utils.pyp_nice_time()
'Mon Aug 15 16:27:38 2005'
```

10.2 pyp_db

pyp_db contains a set of procedures for accessing SQLite 3 (`http://www.sqlite.org/`) databases.

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.

Table 10.1: PyPedal modules.

Module Name	Description
pyp_db	Working with SQLite relational databases: create databases, add/drop tables, load PyPedal pedigrees into tables.
pyp_demog	Generate demographic reports, age distributions, for the pedigreed population.
pyp_graphics	Visualize pedigrees and numerator relationship matrices (NRM).
pyp_io	Save and load NRM and inverses of NRM; write pedigrees to formats used by other packages.
pyp_metrics	Compute metrics on pedigrees: effective founder and ancestor numbers, effective number of founder genomes, pedigree completeness. Tools for identifying related animals, calculating coefficients of inbreeding and relationship, and computing expected offspring inbreeding from matings.
pyp_network	Convert pedigrees directed graphs.
pyp_newclasses	Pedigree, animal, and metadata classes used by PyPedal.
pyp_nrm	Creating, decompose, and inverting NRM, and recurse through pedigrees.
pyp_reports	Create reports from pedigree database (loaded in pyp_db).
pyp_template	Template for developers to use when adding new features to PyPedal (Chapter 9).
pyp_utils	Load, reorder and renumber pedigrees; set flags in individual animal records; string and date-time tools.

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 from which data will be dropped.

tablename The table from which data will be dropped.

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

tableDropTable(dbname='pypedal', tablename='example') ⇒ **integer** tableDropTable() drops an existing table from the database.

dbname The database from which the table will be dropped.

tablename The table which will be dropped.

Returns: 1.

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.

10.3 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.

10.4 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 details on licensing and installation.

Module Contents

draw_pedigree(pedobj, gfilename=’pedigree’, gtitle=’My_Pedigree’, gformat=’jpg’, gsize=’f’,gdot=’1’, gorient=’l’, gdirec=”,
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 animals 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.

gformat The format in which the output file should be written (JPG—PNG—PS).

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).

gorient The orientation of the graph: ’p’: portrait, ’l’: landscape.

gdirec Direction of flow from parents to offspring: ’TB’: top-bottom, ’LR’: left-right, ’RL’: right-left.

gname Flag indicating whether ID numbers (0) or names (1) should be used to label nodes.

gfontsize Integer indicating the typeface size to be used in labelling nodes.

garrow Flag indicating whether or not arrowheads should be drawn.

gtitloc Indicates if the title be drawn or above (’t’) or below (’b’) the graph.

gtitjust Indicates if the title should be center- (’c’), left- (’l’), or right-justified (’r’).

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.

plot_line_xy(xydict, gfilename='plot_line_xy', gtitle='Value by key', gxlabel='X', gylabel='Y', gformat='png') ⇒ integer plot_line_xy() uses matplotlib – if available on your system – to produce a line graph of the values in a dictionary for each level of key.

dictionary A Python dictionary

gfilename The name of the file to which the figure should be written

gtitle The title of the graph.

gxlabel The label for the x-axis.

gylabel The label for the y-axis.

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.

10.5 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) ⇒ **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.

10.6 pyp_metrics

pyp_metrics contains a set of procedures for calculating measures of genetic variation 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 "populationwise" 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)ˆn, 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 **?**), 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.

10.7 pyp_newclasses

pyp_newclasses contains the new class structure that is used by PyPedal 2.0.0. It includes a master pedigree class, a NewAnimal() class, a PedigreeMetadata() class, and a NewAMatrix 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.

__init__(self, kw) ⇒ **object** __init__() initializes a NewAMatrix() object.

kw A dictionary of options.

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

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).

printme() ⇒ **None** printme() prints the NRM to STDOUT.

self Reference to the current NewAMatrix() object

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.

__init__(self, kw) ⇒ object __init__() initializes a NewPedigree() object.

> ***kw*** A dictionary of options.
>
> **Returns:** An instance of a NewPedigree() object populated with data

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

printoptions() ⇒ None printoptions() prints the contents of a pedigree's `kw` dictionary.

> ***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

nuherds() ⇒ **integer-and-list** nuherds() returns the number of unique herds in the pedigree along with a list of the herdss

> ***self*** Reference to the current Pedigree() object
>
> **Returns:** The number of unique herds in the pedigree and a list of those herds

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

10.8 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.

10.9 pyp_reports

pyp_reports contains a set of procedures for generating reports

Module Contents

_pdfCreateTitlePage(canv, _pdfSettings, reporttitle=”, reportauthor=”) ⇒ **None** _pdfCreateTitlePage() adds a title page to a ReportLab canvas object.

canv An instance of a ReportLab Canvas object.

_pdfSettings An options dictionary created by _pdfInitialize().

Returns: None

_pdfDrawPageFrame(canv, _pdfSettings) ⇒ **None** _pdfDrawPageFrame() nicely frames page contents and includes the document title in a header and the page number in a footer.

canv An instance of a ReportLab Canvas object.

_pdfSettings An options dictionary created by _pdfInitialize().

Returns: None

_pdfInitialize(pedobj) ⇒ **dictionary** _pdfInitialize() returns a dictionary of metadata that is used for report generation.

pedobj A PyPedal pedigree object.

Returns: A dictionary of metadata that is used for report generation.

meanMetricBy(pedobj, metric=’fa’, byvar=’by’) ⇒ **dictionary** meanMetricBy() returns a dictionary of means keyed by levels of the ’byvar’ that can be used to draw graphs or prepare reports of summary statistics.

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.

pdfPedigreeMetadata(pedobj, titlepage=0, reporttitle=”, reportauthor=”, reportfile=”) ⇒ integer
pdfPedigreeMetadata() produces a report, in PDF format, of the metadata from the input pedigree. It is intended for use as a template for custom printed reports.

pedobj A PyPedal pedigree object.

titlepage Show (1) or hide (0) the title page.

reporttitle Title of report; if ”, _pdfTitle is used.

reportauthor Author/preparer of report.

reportfile Optional name of file to which the report should be written.

Returns: A 1 on success, 0 otherwise.

10.10 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 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) ⇒ 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+%22sorted+dictionary%22=1=en#cc686f4cd795feb4

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

ELEVEN

Glossary

This chapter provides a glossary of terms.[1]

coefficient of inbreeding Probability that two alleles selected at random are identical by descent.

coefficient of relationship Proportion of genes that two individuals share on average.

effective ancestor number The number of equally-contributing ancestors, not necessarily founders, needed to produce a population with the heterozygosity of the studied population (Boichard, Maignel, and Verrier 1997).

effective founder number The number of equally-contributing founders needed to produce a population with the heterozygosity of the studied population (Lacy 1989).

effective population size The effective population size is the size of an ideal population that would lose heterozygosity at a rate equal to that of the studied population (Falconer and MacKay 1996).

founder An animal with unknown parents that is assumed to be unrelated to all other founders.

internal report A PyPedal() report that is intended for use by other PyPedal() procedures, such as plotting routines, and not for printing.

numerator relationship matrix Matrix of additive genetic covariances among the animals in a population.

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

renumbering Many calculations require that the animals in a pedigree be ordered from oldest to youngest, with sires and dams preceding offspring, and renumbered starting with 1. This is a computational necessity, and results in an animal's ID (`animalID`) being changed to reflect that animal's order in the pedigree. All animals have their original IDs stored in their `originalName` attribute.

[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.

Caballero, A. and M. A. Toro (2000). Interrelations between effective population size and other pedigree tools for the management of conserved populations. *Genetical Research (Cambridge) 75*, 331–343.

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.

Cormen, T. H., C. E. Leiserson, R. L. Rivest, and C. Stein (2003). *Introduction to Algorithms, Second Edition*. Englewood Cliffs, NJ: Prentice-Hall.

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

Falconer, D. S. and F. C. MacKay (1996). *Introduction to Quantitative Genetics* (4th ed.). Longman.

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.

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*. Wallingford, UK: CAB International.

Pattie, W. (1965). Selection for weaning weight in merino sheep. *J. Agric. Exp. Animl. Husb. 5*, 353–360.

Quaas, R. L. (1976). Computing the diagonal elements and inverse of a large numerator relationship matrix. *Biometrics 32*, 949–953.

Roughsedge, T., S. Brotherstone, and P. M. Visscher (1999). Quantifying genetic contributions to a dairy cattle population using pedigree analysis. *Livestock Production Science 60*, 359–369.

Toro, M. A., J. Rodriganez, L. Silio., and C. Rodriguez (2000). Genealogical analysis of a closed herd of black hairless iberian pigs. *Conservation Biology 14*.

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.

Wiggans, G. R., P. M. Van Raden, and J. Zuurbier (1995). Calculation and use of inbreeding coefficients for genetic evaluation of united states dairy cattle. *Journal of Dairy Science 78*, 1584–1590.

Wright, S. (1922). Coefficients of inbreeding and relationship. *Amer. Nat. 56*, 330–338.

Wright, S. (1931). Evolution in mendelian populations. *Genetics 16*, 97–159.

Young, C. W. and A. J. Seykora (1996). Estimates of inbreeding and relationship among registered holstein females in the united states. *Journal of Dairy Science 79*, 502–505.

INDEX

FUNCTION INDEX