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

*Release 2.0.0rc2*

John B. Cole

November 29, 2005
Revised March 11, 2008

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. 2008. 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 twelve 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/`.

Mention of trade names or commercial products in this manual is solely for the purpose of providing specific information and does not imply recommendation or endorsement by the U.S. Department of Agriculture.

All programs and services of the U.S. Department of Agriculture are offered on a nondiscriminatory basis without regard to race, color, national origin, religion, sex, age, marital status, or handicap.

Revised March 11, 2008

# Legal Notice

> The only people who have anything to fear from free software are those whose products are worth even less. — David Emery

Copyright (c) 2002-2008. John B. Cole. All rights reserved.

Permission to use, copy, modify, and distribute this software for any purpose without fee is hereby granted, provided that this entire notice is included in all copies of any software which is or includes a copy or modification of this software and in all copies of the supporting documentation for such software.

## Disclaimer

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

# License

This is free software; you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation; either version 2.1 of the License, or (at your option) any later version.

This software is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.

You should have received a copy of the GNU Lesser General Public License along with this library; if not, write to the Free Software Foundation, Inc., 51 Franklin St, Fifth Floor, Boston, MA 02110-1301, USA.

# CONTENTS

# LIST OF TABLES

# LIST OF FIGURES

CHAPTER

**ONE**

# Introduction

> Any sufficiently disguised bug is indistinguishable from a feature. — Rich Kulawiec

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 2.

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 **??** describes the license under which PyPedal is distributed. It is important that you review the license before using the program.

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

**High-Level Overview** Chapter 3 gives a high-level overview of the components of the PyPedal system as a whole.

**Methodology** Chapter 7 provides a brief overview of the methodology used to calculate measures of genetic diversity.

**HOWTOs** Chapter 8 provides demonstrations of how to perform common tasks.

**Graphics** Chapter 9 provides details on producing graphics with PyPedal.

**Reports** Chapter 10 provides details about the report generation tools available in PyPedal.

**Implementing New Features** Chapter 11 introduces the idea of extensibility and walks the reader through the development of a new PyPedal routine.

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

**Glossary** Chapter 12 provides a glossary of terms.

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

## 1.1 Implemented Features

A full list of features, including notes on useage and computational details, is provided in Chapter **??**. 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;
- Calculation of coefficients of partial inbreeding using an iterative tabulat method (Lacy, Alaks, and Walsh 1996; Gulisija, Gianola, Weigel, and Toro 2006);
- Calculation of coefficients of ancestral inbreeding using the methods of Ballou (1997) or Suwanlee et al. (2007);
- 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;
- Calculation of the inbreeding of offspring from a prospective mating;

- Output to ASCII text files, including matrices, coefficients of inbreeding and relationship, and summary information;
- Simulation of pedigrees using an algorithm derived from that in Matvec 1.1a;

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; Cole 2007).

## 1.2 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`).

## 1.3 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 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 Kelly (Department of Animal and Poultry Science, University of Guelph), Thomas von Hassell, and Gianluca Saba. Gregor Gorjanc has written a blog entry describing how to install PyPedal on Debian Linux. Fernando Perez posted a LaTeXpreamble to the NumPy listserver that dramatically improved the PDF version of This Manual.

The Newfoundland pedigree presented in Figure 9.4 was taken from the NewFoundland Dog database (`http://www.newfoundlanddog-database.net/en/`) and is used with permission.

The pedigree of European royalty used in the GEDCOM discussion (Appendix B), ged3.ged, was taken from the Genealogy Forum website (`http://www.genealogyforum.com/`). It is believed to be in the public domain, and is used with the knowledge of the website administrators.

## 1.4 Disclaimer

Reference to any commercial product is made with the understanding that no discrimination is intended and no endorsement by USDA is implied.

CHAPTER

**TWO**

# Installing PyPedal

> As we acquire more knowledge, things do not become more comprehensible, but more mysterious. — Will Durant

This chapter explains how to install and test PyPedal under Posix-type operating systems and Microsoft Windows.

## 2.1 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 2.1. Note that some extensions need to be installed before others: `NumPy` 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.

Note that Graphviz, NetworkX, PyGraphviz, PythonDoc, ReportLab, and SQLite are not installed by the Enthought Python distribution for Windows.

Table 2.1: 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/` |
| NumPy | Array manipulation | `http://www.numpy.org/` |

*continued on next page*

| Extension | Function | URL |
|---|---|---|
| PIL | Image processing | http://effbot.org/zone/pil-index.htm |
| pydot | Interface to Graphviz | http://dkbza.org/pydot.html |
| PyGraphviz | Interface to Graphviz | https://networkx.lanl.gov/wiki/pygraphviz |
| 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/ |

## 2.2 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.

## 2.3 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.

### 2.3.1 Installing on Unix, Linux, and Mac OSX

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

```
gunzip pypedal-2.0.0a17.tar.gz
tar xf pypedal-2.0.0a17.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 **??**.

### 2.3.2 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 PyPedal has been lightly-tested on Windows XP, but I 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 listed above are satisfied. Enthought provides a Python distribution that is bundled with a number of extras, including most of the dependencies needed to install and run PyPedal in the Windows environment (`http://code.enthought.com/enthon/#download`). It is a large download ( 120 MB) but greatly simplifies installation. If you use the Enthought distribution you will still need to download and install Graphviz, ReportLab, and SQLite from their respective sites (Table 2.1). Graphviz and ReportLab are easy to install. Installation instructions for SQLite are provided below.

**FIX ►►► There is not yet an easy way to install the `PyGraphviz` library under Windows. `PyGraphviz` is used by some of the graphics routines for rendering pedigrees. Windows users will need to refer to Chapter 9 for details on routines which require this module. ◄◄◄**

It is possible that the installation may fail with the message

```
error: Download error: (10060, 'Operation timed out')
```

This means that the installer was trying to download a dependency and that operation was unsuccessful. This can usually be remedied by simply running the installer again.

In order to get your installation working correctly you will need to set some environment variables. Under Windows XP (and 2000) 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 *System 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 using an unzipping utility such as 7-Zip (`http://www.7-zip.org/`) that understands gzipped-and-tarred files. Once you've done that, open a shell by left-clicking on "Start", selecting "Run", and typing `cmd` into the box. Navigate to the directory into which you unpacked the PyPedal distributio(this is an example, your file location may vary):

```
C:\> cd C:\PyPedal
```

2. Build it using the distutils defaults:

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

This installs PyPedal in `C:\python24\site-packages`.

### Installation on Cygwin

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

## 2.4 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
'2.0.0b17'
```

indicating that PyPedal is installed.

CHAPTER

**THREE**

# High-Level Overview

By a small sample we may judge the whole piece. — Miguel de Cervantes Saavedra

## 3.1 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.

## 3.2 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.

PyPedal also provides `LightAnimal` and `SimAnimal` objects. `LightAnimal`s are intended for use with the graph theoretic routines provided in `pyp_network` and lack many of the attributes of `NewAnimal` objects, such as names, breeds, and alleleotypes. `SimAnimal`s are intended for internal use only by the pedigree simulation routines.

A detailed explanation of each class is provided in Chapter 6.

## 3.3 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 3.4). 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 either creating a Python dictionary and passing it as a parameter when `pyp_newclasses.loadPedigree()` is called or by specifying a configuration file name. For example, here is how you would create and populate an options dictionary:

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

The syntax used in a configuration file is similar. Consider the file 'options.ini', which contains the same options as set in the *options* dictionary in the previous example:

```
# options.ini
# This is an example of a PyPedal configuration file
messages = verbose
renumber = 0
pedfile = new_lacy.ped
pedformat = asd
pedname = Lacy (1989) Pedigree
```

More details on configuration files are provided in Section 3.4.1.

You may name your dictionary or configuration file 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. This is as simple as calling `pyp_newclasses.NewPedigree()`:

```
example = pyp_newclasses.loadPedigree(options)
```

If you would like to use a configuration file to set your pedigree options, supply the configuration file name using the *optionsfile* keyword:

```
example = pyp_newclasses.loadPedigree(optionsfile='options.ini')
```

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.

## 3.4 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 3.1. Options are stored in a Python dictionary that you must create in your programs. You must specify values for the *pedfile* and *pedformat* options; all others are optional. *pedfile* is a string containing the name of the file from which your pedigree will be read. *pedformat* is a string containing a pedigree format code (see section 3.5.1) 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.loadPedigree(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.loadPedigree()`, which requires that you pass it either a dictionary of options or a configuration file name. If you do not provide one of these, 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.loadPedigree(options)
#   Read the second pedigree
    options['pedfile'] = 'new_boichard.ped'
    options['pedformat'] = 'asdg'
    options['pedname'] = 'Boichard Pedigree'
    example2 = pyp_newclasses.loadPedigree(options)
```

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 3.1: 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*! |
| animal_type | 'new' | Indicates which animal class should be used to instantiate animal records, `NewAnimal` or `LightAnimal` ('new'—'light'). |
| 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_fontsize | 10 | Specifies the default font size used in `pyp _graphics`. If the font size cannot be cast to an integer, it is set to the default value of 10. Font sizes less than 1 are set to the default of 10. |
| 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()`. |

*continued on next page*

| Option | Default | Note(s) |
|---|---|---|
| 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_breed | 'Unknown' | Default breed name. |
| missing_byear | 1900 | Default birth year. |
| missing_name | 'Unknown' | Default animal name. |
| missing_parent | 0 | Indicates what code is used to identify missing/unknown parents in the pedigree file. |
| newanimal_caller | 'loader' | Internal parameter needed for addanimal() to work correctly with ASD pedigrees. |
| nrm_format | 'text' | Format to use when writing an NRM to a file ('text'—'binary'). Array elements in text files are separated by *sepchar*. |
| 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'). |
| pedcomp | 0 | When 1, calculate pedigree completeness using pedcomp_gens generations of pedigree. |
| pedcomp_gens | 3 | Number of generations of pedigree to use when calculating pedigree completeness. |
| pedfile | None | File from which pedigree is read; must provided unless you are simulating a pedigree. Defaults to 'simulated_pedigree' for simulated pedigrees. |
| pedformat | 'asd' | See 3.5.1 for details. |
| pedname | 'Untitled' | A name/title for your pedigree. |

*continued on next page*

| Option | Default | Note(s) |
|---|---|---|
| pedgree_is_renumbered | 0 | Indicates whether or not the pedigree has been renumbered. |
| pedigree_summary | 1 | Indicates whether or not the pedigree loading details and summary are printed to STDOUT. Output is only written if *message* is set to 'verbose'. |
| 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). |
| simulate_fs | 0 | Flag indicating whether or not full-sib matings are allowed. |
| simulate_g | 3 | Number of distinct generations in the simulated pedigree. |
| simulate_ir | 0.0 | Immigration rate, the rate at which new founders with unknown parents enter the population. |
| simulate_mp | 0 | Flag indicating whether or not simulated animals may have missing parents. |
| simulate_n | 15 | Total number of animals in simulated pedigree, including founders. |
| simulate_nd | 4 | Number of initial founder dams in pedigree. |
| simulate_ns | 4 | Number of initial founder sires in pedigree. |
| simulate_pedigree | 0 | Option to simulate a pedigree rather that load one from a file. All other simulation-related variables are ignored when this is not set to 1. |
| simulate_pmd | 100 | Maximum number of draws allowed when trying to sample parents that comply with all restrictions. |
| simulate_po | 0 | Flag indicating whether or not parent-offspring matings are allowed. |
| simulate_save | 0 | Flag indicating whether or not the simulated pedigree should be written to a file after it is created. |
| simulate_sr | 0.5 | Sex ratio in simulated pedigree; ¡ 0.5 gives more females, ¿ 0.5 gives more males. |
| 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. |

### 3.4.1 Configuration Files

The Dict4Ini module (http://cheeseshop.python.org/pypi/Dict4Ini/0.4) is used to process configuration files, and in included with the distribution so that you do not need to download and install it. Dict4Ini objects can be addressed as though they are standard Python dictionaries, which made it very easy to add configuration file support to PyPedal. Configuration files consist of simple *keyword = value* pairs on separate lines[1], and may include comments.

[1] Please note that the Dict4Ini documentation referrs to sections. Sections are very commonly used in configuration files, but PyPedal does not use them.

```
# new_options.ini
# This is an example of a PyPedal configuration file.
pedfile = new_lacy.ped
pedformat = asd
pedname = Lacy Pedigree
```

If neither an options dictionary nor a configuration file name is provided, `pyp_newclasses.loadPedigree()` will try and load the file named 'pypedal.ini'.

## 3.5 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 3.4. PyPedal also provides tools for pedigree simulation, which are discussed in section 3.8. More details about pedigree input may be found in Chapter 4.

### 3.5.1 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 3.2.

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.

Table 3.2: Pedigree format codes.

| Code | Description |
|---|---|
| a | animal ('a' or 'A' REQUIRED) |
| s | sire ('s' or 'S' REQUIRED) |
| d | dam ('d' or 'D' REQUIRED) |
| y | birthyear (YYYY) |
| e | age |
| f | coefficient of inbreeding |
| g | generation |
| h | herd |
| l | alive (1) or dead (0) |
| n | name |
| p | Pattie's (1965) generation coefficient |
| r | breed |
| u | user-defined field (string) |
| b | birthdate in "MMDDYYYY" format |
| x | sex |

*continued on next page*

| Code | Description |
|---|---|
| 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) |
| Z | indicates a column that should be skipped (one allowed per pedigree) |

## 3.6 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:

```
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 renumbered your pedigree.

### 3.6.1 Animal Identification

A detailed explanation of animal identification cross-references is provided in Section 7.2.

## 3.7 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.

## 3.8 Simulating Pedigrees

PyPedal is capable of simulating pedigrees using am algorithm based on the `Pedigree::sample` method in Matvec 1.1a (`http://statistics.unl.edu/faculty/steve/software/matvec/`), although the implementation in `NewPedigree` is all original code. A pedigree is simulated when the `simulate_pedigree` flag is set, and is the only case in which a `pedfile` does not need to be provided to PyPedal. All simulated pedigrees have the code 'asdxg' and are *not* renumbered. The options used to control pedigree simulation are presented in Table 3.1.

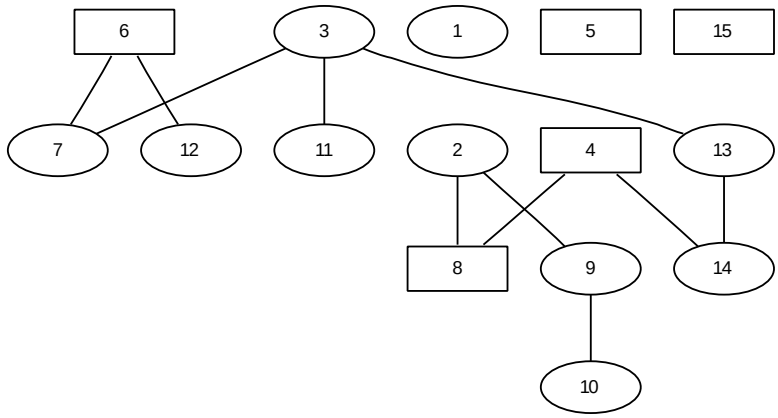


Figure 3.1: Simulated pedigree using default options

The basic structure of a simulated pedigree is determined by the total number of simulated animals (`simulate_n`), founder sires (`simulate_ns`) and dams (`simulate_nd`), and the number of distinct generations in the pedigree (`simulate_g`). Populations can be closed or open based on the value of `simulate_ir`; when the immigration rate is $> 0$ that proportion of new animals will be immigrants with unknown parents. The sex ratio can be altered by changing `simulate_sr`; values $< 0.5$ will result in more females than males, and values $> 0.5$ will result in more males than females. By default, `NewPedigree.simulate` produces a three-generation pedigree with 15 animals descended from 4 founder sires and 4 founder dams (Figure 3.1). *simulate_mp* is a flag indicating whether or not simulated animals may have missing parents. When missing parents are allowed, animals may have no, one, or both parents unknown. The related parameter, *simulate_pmd*, specifies the number of times parents should be sampled at random when trying to satisfy all of the simulation constraints. If parents are sampled *simulate_pmd* times without satisfying the rules in place, both parents are set to missing, even if missing parents are not permitted. Other constraints include allowing/forbidding parent-offspring (*simulate_po*) and/or full sib (*simulate_fs*) matings.

CHAPTER

**FOUR**

# Input and Output

The only legitimate use of a computer is to play games. — Eugene Jarvis

## 4.1 Overview

Getting data into and out of programs, while extremely important to end-users, is often challenging. PyPedal is able to load pedigrees from, and save them to, from a number of different sources. A list of supported input and output methods may be found in Table 4.1.

Table 4.1: PyPedal input and output methods.

| Direction | Source | Description |
|---|---|---|
| Input | db | Load an ASDx-formatted data from an SQLite database |
| | fromgraph | Load a pedigree from an instance of an XDiGraph object (not a file) |
| | gedcomfile | Load a pedigree from a GEDCOM 5.5 file |
| | graphfile | Load a pedigree from an adjacency list using the read_adjlist() function from the NetworkX module |
| | simulate | Simulate a pedigree rathen that loading it from a file |
| | textstream | Load a pedigree from a string contaiing animal records |
| Output | save | Save a pedigree to a user-specified file |
| | savedb | Save a pedigree to a database table |
| | savegedcom | Save a pedigree to a GEDCOM 5.5 file |
| | savegraph | Save a pedigree to a file as an adjacency list |

It has evolved over time that the input methods are implemented as cases of '`pedsource`' in the `NewPedigree::load()` method, while output methods are implemented as individual methods of `NewPedigree`. While it is quite straightforward to add new methods for pedigree output, it is trickier to add new input sources. The greater difficulty in adding new input sources is largely attributable to the mysterious workings of the `NewPedigree::preprocess()` method, which walks through input line-by-line to load the pedigree and perform a number of integrity checks. Once the data are loaded into a `NewPedigree` object it is easy to output them because there is no need to check the integrity of the pedigree and relationships among the records in the pedigree. If you want to implement a new input source the sanity-saving way to go is to get the data into a list that you can pass to `preprocess()`; `preprocess()` can then walk the list as it would walk through the lines of an input file.

## 4.2 Input

The process of data input hsa been made as simple as is reasonably possible, but it is ultimately the responsibility of the user to prepare their data. Most people will load their data from a simple text file, with one pedigree entry per line. A number of pedigree integrity checks are performed when the data are loaded. First, the pedigree format string (Section 3.5.1) is checked for validity and compared to a line of data to make sure that the specfied number of colums of data exist. Once that check is passed individual records are loaded and checked one at a time. As individual records are vvalidated they are added to the pedigree. Duplicate records are eliminated, parents that appear in the pedigree but do not have their own record in the inpur data are added to the pedigree, and sexes are checked (animals cannot appear as both sires and dams). If animal IDs are strings they are hashed to get numeric IDs for use in pedigree reordering and renumbering.

If you used an earlier version of PyPedal you may have added a pedigree format string to your pedigree file. You do not need to include that string in the current version of PyPedal, and if `preprocess()` sees one while reading a pedigree file it will ignore it. You may include comment lines in your file as well; they will also be ignored.

### 4.2.1 Databases

PyPedal can load ASDx-formatted pedigrees from an SQLite database using '`pedsource='db''`. The pedigree will be loaded from the database and table specified in the *database_name* and *dbtable_name* variables. Consider the following example:

```
test = pyp_newclasses.loadPedigree(options,pedsource='db')
test.metadata.printme()
```

This produces the output:

```
Metadata for  DB Stream ()
    Records:                7
    Unique Sires:           3
    Unique Dams:            3
    Unique Gens:            1
    Unique Years:           1
    Unique Founders:        4
    Unique Herds:           1
    Pedigree Code:          ASDx
```

Note that user-supplied values of the pedigree format string will be over-written by the `load()` method and do not affect database processing. Database importation is hard-coded to accept only pedigrees in that format.

### 4.2.2 Graph Objects

Pedigrees can be represented as a type of mathematical object called a directed graph (digraph; not to be confused with visualizations of data). The `NetworkX` module provides Python tools for working with digraphs, and PyPedal provides a convenient way for loading a pedigree from an instance of an `XDiGraph` object.

```
example = pyp_newclasses.loadPedigree(optionsfile='new_networkx.ini')
ng = pyp_network.ped_to_graph(example)
options = {}
options['pedfile'] = 'dummy'
options['pedformat'] = 'asd'
example2 = pyp_newclasses.loadPedigree(options,pedsource='graph',pedgraph=ng)
example2.metadata.printme()
```

You can see from the printout of the metadata from the new pedigree, '`example2`', that the graph '`ng`' was successfully converted to a `NewPedigree` object.

```
Metadata for  Testing fromgraph() (dummy)
Records:                13
Unique Sires:           3
Unique Dams:            4
Unique Gens:            1
Unique Years:           1
Unique Founders:        5
Unique Herds:           1
Pedigree Code:          asd
```

When considering the utility of such tools it might be helpful to recall that PyPedal was written by and for use in scientific research. Perhaps that will allow the author to be forgiven a multitude of sins.

### 4.2.3 GEDCOM Files

A thorough description of support for GEDCOM files may be found in Appendix B.

### 4.2.4 Text Files

Most users will load their pedigrees from simple text files. As an example, consider a large dog pedigree, an excerpt of which is presented below.

```
# dogID,fatherID,motherID,gender,born
64 66 67 2 1979
63 64 65 1 1982
62 191 195 2 1982
61 64 65 2 1982
...
```

This pedigree can be loaded using this short program:

```
options = {}
options['pedfile'] = 'dog.ped'
options['pedname'] = 'A Large Dog Pedigree'
options['pedformat'] = 'asdgb'
if __name__ == '__main__':
    test = pyp_newclasses.loadPedigree(options)
```

There are numerous examples of loading pedigrees from text files throughout this manual.

### 4.2.5 Text Streams

There are some use cases, such as web services, for which it may be desirable to load pedigrees from strings rather than from files. This is done by passing the `pedsource` keyword to `pyp_newclasses.loadPedigree` with a value of '`textstream`', along with a string named '`pedstream`':

```
options = {}
options['pedfile'] = ''
options['messages'] = 'verbose'
options['pedformat'] = 'ASD'

if __name__ == "__main__":
    pedstream = 'a1,s1,d1\na2,s2,d2\na3,a1,a2\n'
    test = pyp_newclasses.loadPedigree(options,pedsource='textstream',pedstream=pedstream)
```

Only ASD-formatted pedigrees can be loaded this way, individual IDs are separated with commas, and successive records are separated by newlines. These restrictions are hard-coded into the `NewPedigree::load()` method. All records must contain a newline, including the last record in the string! You must also set the '`pedfile`' option to some value, even if that value is just an empty string as in the example.

The expected use case is that the pedigree would be retrieved from a database, and the result set from the SQL query converted into a string. There is an upper bound on the size of pedigree that can be loaded from a stream based on the physical memory available on your platform, and extremely large pedigrees should be loaded from a text file.

## 4.3 Output

### 4.3.1 Databases

PyPedal can write `NewPedigree` pedigrees to ASDx-formatted SQLite tables using the `NewPedigree::savedb()` method. The following program will result in the creation of a table named '`test_save`' in the database '`test_pypedal_save`'.

```
test.kw['database_name'] = 'test_pypedal_save'
test.kw['dbtable_name'] = 'test_save'
test.savedb()
```

The pedigree is saved to the database and table specified in the *database_name* and *dbtable_name* variables. If the specified table already exists it will be dropped and a new table created. This can result in data loss! Please be careful.

### 4.3.2 GEDCOM Files

A thorough description of GEDCOM file exportation may be found in Appendix B.

### 4.3.3 Graph Objects

The `NewPedigree::savegraph()` save a pedigree to a file as an adjacency list, which is a commonly-used format for describing digraphs. If the pedigree has already been converted to a digraph pass it using the *pedgraph* argument; otherwise, the method will call the appropriate converter:

```
test.savegraph(pedoutfile='test.adj')
```

The file '`test.adj`' has the following comments:

```
# sqlite.py
# GMT Tue Mar  4 20:38:52 2008
# Text Stream
1 5
2 6
3 5
4 6
5 7
6 7
7
```

This example demonstrates that there can be a considerable loss of information when going from a PyPedal pedigree to some other way of representing the pedigree, such as a graph.

### 4.3.4 Text Files

The `NewPedigree::save()` method writes a PyPedal pedigree to a user-specified file with either the format specified in the pedigree format string (default) or a format including all variables in the pedigree ('`outformat='l''`). Either the original (default) or renumbered ('`idformat='r''`) IDs can be used for animal, sire, and dam IDs. In the following example all variables are written to a file named '`all_data.ped`' using the same IDs as in the original file:

```
test.save(filename='all_data.ped',outformat='l',idformat='o')
```

`NewPedigree::save()` tries never to overwrite your data. If you do not pass a filename argument a file whose name is derived from, but not the same as, the original pedigree filename will be used. The string '`_saved`' will be appended to the filename in order to distinguish it from the original pedigree file.

### 4.3.5 Text Streams

`NewPedigree::tostream()` returns a text stream from an instance of a `NewPedigree` object. The text stream contains an ASD-formatted pedigree as a string in which individual IDs are separated by commas and successive records are separated by newlines.

```
>>> pedstream = 'a1,s1,d1\na2,s2,d2\na3,a1,a2\n'
>>> test = pyp_newclasses.loadPedigree(options,pedsource='textstream',pedstream=pedstream)
>>> pedstream2 = test.tostream()
>>> print pedstream2
'd1,0,0\nd2,0,0\ns1,0,0\ns2,0,0\na2,s2,d2\na1,s1,d1\na3,a1,a2\n'
```

Note that in this example the input and output text streams differ because the input stream did not include pedigree entries for all animals in the pedigree (for example, sire '`s1`'). Recall that PyPedal adds missing entries for parents when the pedigree is loaded.

CHAPTER

**FIVE**

# Working with Pedigrees

> Are you quite sure that all those bells and whistles, all those wonderful facilities of your so called powerful programming languages, belong to the solution set rather than the problem set? — Edsger Dijkstra

## 5.1 Overview

In Chapter 4 tou learned how to get your pedigree data loaded into PyPedal. This chapter will show you what you can do with the pedigree once it is loaded. The examples in this chapter assumes that you are working with a reordered and renumbered pedigree (see Section 7.1 for additional details).

In order to get the most out of your pedigrees you need to understand the basic structure of a PyPedal pedigree, which consists of two components: the pedigree itself, which is composed of a list of `NewAnimal` objects, and metadata, which is data about the animals contained in the pedigree. Some calculations are performed on the animal records directly, while others use the metadata, or some combination of the two. The fundamental goal of PyPedal is to provide the user with tools for asking questions about their pedigrees.

The following discussion will use the following pedigree taken from Boichard et al. (1997):

```
# pedformat: asdg
1 0 0 1
2 0 0 1
3 0 0 1
4 0 0 1
5 1 2 2
6 3 4 2
7 5 6 3
8 5 6 3
9 5 6 3
10 5 6 3
11 5 6 3
12 5 6 3
13 5 6 3
14 5 6 3
```

Many of the subsequent code snippets are taken from the '`new_methods.py`' example program (see: Appendix A).

## 5.2 Inbreeding and Relationships

The

```
inbr = pyp_nrm.inbreeding(example)
print 'inbr: ',
>>> inbr:  {
    '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},
    'metadata': {
        'nonzero': {'f_max': 0.0, 'f_avg': 0.0, 'f_rng': 0.0,
            'f_sum': 0.0, 'f_min': 0.0, 'f_count': 0},
        'all': {'f_max': 0.0, 'f_avg': 0.0, 'f_rng': 0.0, 'f_sum': 0.0,
            'f_min': 0.0, 'f_count': 14}
        }
    }
```

dictionary returned by `inbreeding()` contains two dictionaries: '`fx`' contains coefficients of inbreeding keyes to animal IDs, and '`metadata`' contains summary information about the coefficients of inbreeding in the pedigree. '`metadata`' also contains two dictionaries: '`nonzero`' contains summary statostics only for animals with non-zero coefficients of inbreeding, and '`all`' contains statistics for all animals.

Relationship metadata, similar to the inbreeding metadata described above but for coefficients of relationship, are available but not calculated by default.

```
inbr,reln = pyp_nrm.inbreeding(example,rels=1)
print 'reln: ', reln
>>> reln:  {'r_nonzero_count': 10, 'r_nonzero_avg': 0.40000000000000002,
    'r_min': 0.25, 'r_sum': 4.0, 'r_avg': 0.19047619047619047, 'r_max': 0.5,
    'r_count': 21, 'r_rng': 0.25}
```

The dictionary of relationship metadata returned by `inbreeding()` also contains statistics for zero and non-zero coefficients of relationship. On the example presented above the pedigree contains

Relationship metadata are not guaranteed to be correct when '`method = 'vanraden'`' is used. This is because `inbreeding_vanraden()` uses a speed-up when there are full-sibs in the pedigree to avoid repeating calculations. The metadata should be reasonably accurate for pedigrees with few or no full-sibs. The summary statistics will not be very accurate in the case of pedigrees that contain lots of full-sibs.

The relationship metadata do not include individual pairwise relationships. In order to associate those with your pedigree you must create a `NewAMatrix` object, form the numerator relationship matrix (NRM), and attach it to the pedigree:

```
options = {}
...
example = pyp_newclasses.loadPedigree(options)
example.nrm = pyp_newclasses.NewAMatrix(example.kw)
example.nrm.form_a_matrix(example.pedigree)
```

If you know when you load the pedigree file that you want to calculate and store the NRM you can save a little typing by setting the '`form_nrm`' option:

```
options = {}
options['form_nrm'] = 1
...
example = pyp_newclasses.loadPedigree(options)
```

If you want to inspect the NRM you can use '`example.nrm.printme()`' to print the matrix to the screen, which is probably not a particularly good idea for large matrices.

```
[[ 1.   0.   0.   0.   0.5  0.   0.25 0.25 0.25 0.25 0.25 0.25 0.25 0.25]
 [ 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.   1.   0.   0.   0.5  0.25 0.25 0.25 0.25 0.25 0.25 0.25 0.25]
 [ 0.   0.   0.   1.   0.   0.5  0.25 0.25 0.25 0.25 0.25 0.25 0.25 0.25]
 [ 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.   0.5  0.5  0.   1.   0.5  0.5  0.5  0.5  0.5  0.5  0.5  0.5 ]
 [ 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 ]
 [ 0.25 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 ]
 [ 0.25 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 ]
 [ 0.25 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 ]
 [ 0.25 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 ]
 [ 0.25 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 ]
 [ 0.25 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 ]
 [ 0.25 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.  ]]
```

If you want to get the pairwise relationship between two animals you need to use their renumbered IDs and subtract 1 (because the array is zero-indexed). For example, if you wanted the coeffient of relationship between animals 2 and 5 (an individual and its sire) you would use the indices '`1`' and '`4`':

```
print example.nrm.nrm[1][4]
>>> 0.5
```

The NRM is symmetric, which means that '`nrm[1][4]`' and '`nrm[4][1]`' are identical.

```
print example.nrm.nrm[1][4]
>>> 0.5
print example.nrm.nrm[5][1]
>>> 0.5
```

You can also easily save the NRM to a file for future reference:

```
example.nrm.save('Amatrix.txt')
```

If you're from Missouri you can verify that the contents of '`Amatrix.txt`' are:

```
1.0  0.0  0.0  0.0  0.5 0.0 0.25 0.25 0.25 0.25 0.25 0.25 0.25 0.25
0.0  1.0  0.0  0.0  0.5 0.0 0.25 0.25 0.25 0.25 0.25 0.25 0.25 0.25
0.0  0.0  1.0  0.0  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  1.0  0.0 0.5 0.25 0.25 0.25 0.25 0.25 0.25 0.25 0.25
0.5  0.5  0.0  0.0  1.0 0.0 0.5  0.5  0.5  0.5  0.5  0.5  0.5  0.5
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
0.25 0.25 0.25 0.25 0.5 0.5 1.0  0.5  0.5  0.5  0.5  0.5  0.5  0.5
0.25 0.25 0.25 0.25 0.5 0.5 0.5  1.0  0.5  0.5  0.5  0.5  0.5  0.5
0.25 0.25 0.25 0.25 0.5 0.5 0.5  0.5  1.0  0.5  0.5  0.5  0.5  0.5
0.25 0.25 0.25 0.25 0.5 0.5 0.5  0.5  0.5  1.0  0.5  0.5  0.5  0.5
0.25 0.25 0.25 0.25 0.5 0.5 0.5  0.5  0.5  0.5  1.0  0.5  0.5  0.5
0.25 0.25 0.25 0.25 0.5 0.5 0.5  0.5  0.5  0.5  0.5  1.0  0.5  0.5
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  0.5
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
```

Information on the endianess and precision of the data in the array are lost, so this is not a good way to archive data or move data between machines with different endianess. The data are always written in C (row major) order. For better performance you can set the '`nrm_format`' option to '`binary`', which will use a binary file formt.

Finally, if you're going to work on the exact same pedigree later you can load '`Amatrix.txt`' and avoid having to recalculate the NRM entirely:

```
example.nrm2 = pyp_newclasses.NewAMatrix(example.kw)
example.nrm2.load('Amatrix.txt')
example.nrm2.printme()
```

Since we've loaded '`Amatrix.txt`' into a second NRM ('`nrm2`') attached to our pedigree it's straightforward, if tedious, to verify that the two NRM contain the same values.

## 5.3 Matings

There are a number of questions you might wish to ask that involve matings, such as, "What is the minimum-inbreeding mating among this set of individuals?". Several routines in `pyp_metrics`.

Suppose you were considering a mating between animals 5 (index 4) and 14 (index 13), which is a sire-daughter mating. How would you go about this? You simply call `pyp_metrics.mating_coi()`:

```
print '\tCalling mating_coi() at %s' % ( pyp_nice_time() )
f = pyp_metrics.mating_coi(example.pedigree[4].animalID,
    example.pedigree[13].animalID,example,1)
print f
```

which produces the output:

```
Calling mating_coi() at Wed Mar  5 11:31:30 2008
0.25
```

We don't need PyPedal to tell us the coefficient of inbreeding of such a simple mating, but the calculations can be complex for more complicated cases. While you can do all of the necessary computations "by hand", `pyp_-`

metrics.mating_coi() takes care of that for you by adding a new dummy animal to the pedigree with the proposed parents, calculating the coefficient of inbreeding of that mating, deleting the dummy animal from the pedigree, and returning the coefficient of inbreeding.

PyPedal takes things one step further, allowing you to work with groups of proposed matings at one time using pyp_metrics.mating_coi_group(). Internally it works similarly to pyp_metrics.mating_coi(), although instead of passing a pair of parents you pass a list of proposed matings. The matings are of the form '<parent1>_<parent2>'. Here's an example in which we are going to consider the following three matings: 1 with 5, 1 with 14, and 5 with 14. Note how the matings are formed and appended to the 'matings' list in one step.

```
matings = []
matings.append('%s_%s'%(example.pedigree[0].animalID, example.pedigree[4].animalID))
matings.append('%s_%s'%(example.pedigree[0].animalID, example.pedigree[13].animalID))
matings.append('%s_%s'%(example.pedigree[4].animalID, example.pedigree[13].animalID))
fgrp = pyp_metrics.mating_coi_group(matings,example)
print 'fgrp: ', fgrp['matings']
```

That code produces a list of the proposed matings with associated coefficients of inbreeding. In addition to the 'matings' dictionary, the 'fgrp' dictionary also contains a dictionary named 'metadata' which contains summary statistics about the proposed matings.

```
fgrp:  {'1_5': 0.25, '5_14': 0.25, '1_14': 0.125}
```

## 5.4 Relatives

PyPedal provides a number of tools for extracting information about relatives from a pedigree. Examples include: obtaining lists of the ancestors of an animal, getting lists of ancestors shared in common by a pair of animals, listing the descendants of an individual, the calculation of the additive genetic relationship between a given pair of animals, and the creation of "subpedigrees" containing only specified animals.

It is easy to obtain a list of an animal's relatives using pyp_metrics.related_animals():

```
list_a = pyp_metrics.related_animals(example.pedigree[6].animalID,example)
list_b = pyp_metrics.related_animals(example.pedigree[13].animalID,example)
```

produces a list of each animal's relatives:

```
[5, 1, 2]
[14, 5, 1, 2, 6, 3, 4]
```

pyp_metrics.common_ancestors() is used to get a list of the common ancestors of two animals, say animals '5' and '14' (remember that this pedigree is renumbered, and that Python lists are indexed from 0, so we need to offset the animal IDs by 1 to get the correct animal IDs).

```
list_r = pyp_metrics.common_ancestors(example.pedigree[4].animalID,example.pedigree[13].ani
malID,example)
print list_r
```

Results are returned as lists:

```
[1, 2, 5]
```

If you've already obtained ancestor lists for a given pair of animals in which you're interested you can also obtain a list of common ancestors using Python sets, which avoids performing calcualtions more than needed:

```
>>> set_a = set(list_a)
>>> set_b = set(list_b)
>>> set_c = set_a.intersection(set_b)
>>> set_c
set([1, 2, 5])
>>> list_c = list(set_c)
>>> list_c
[1, 2, 5]
```

A list of descendants can be obtained by calling `pyp_metrics.descendants()`, which returns a dictionary:

```
>>> pyp_metrics.descendants(5,example,{})
{7: 7, 8: 8, 9: 9, 10: 10, 11: 11, 12: 12, 13: 13, 14: 14}
```

However, it is improtant to note that you will not get the answer you expect unless you have either set the option '`set_offspring = 1`' or called `pyp_utils.assign_offspring()` after loading the pedigree. There is also a convenience function, `pyp_metrics.founder_descendants()`, for handling the special case of obtaining descendants of all of the founders in the pedigree:

```
>>> pyp_metrics.founder_descendants(example)
{1: {5: 5, 7: 7, 8: 8, 9: 9, 10: 10, 11: 11, 12: 12, 13: 13, 14: 14},
 2: {5: 5, 7: 7, 8: 8, 9: 9, 10: 10, 11: 11, 12: 12, 13: 13, 14: 14},
 3: {6: 6, 7: 7, 8: 8, 9: 9, 10: 10, 11: 11, 12: 12, 13: 13, 14: 14},
 4: {6: 6, 7: 7, 8: 8, 9: 9, 10: 10, 11: 11, 12: 12, 13: 13, 14: 14}}
```

CHAPTER

**SIX**

# Using PyPedal Objects

In every chaotic behaviour, there lies a pattern. — Jean Jacques Rousseau

In this chapter, a detailed explanation of each PyPedal class is presented, including attributes and methods.

## 6.1 Animal Objects

Three types of animal object are provided in PyPedal (6.1.1). Users will typically work with instances of `NewAnimal` objects, while `LightAnimal` (6.1.2) and `SimAnimal` (6.1.3) objects are of interest primarily to developers. Detailed descriptions of each class and class method may be found in the API Reference for the `pyp_newclasses` module (Section **??**).

### 6.1.1 The NewAnimal Class

Table 6.1: Attributes of `NewAnimal` objects.

| Attribute | Default | | Description |
|---|---|---|---|
| | Integral IDs (`'asd'`) | String IDs (`'ASD'`) | |
| age | -999 | -999 | The animal's age based on the global `BASE_DEMOGRAPHIC_YEAR` defined in `pyp_demog`. If the *by* is unknown, the inferred generation is used. If the inferred generation is unknown, the age is set to -999. |
| alive | `'0'` | `'0'` | Flag indicating whether or not the animal is alive: `'0'` = dead, `'1'` = alive. |
| alleles | [″,″] | [″,″] | Alleles used for gene dropping. |
| ancestor | `'0'` | `'0'` | Flag indicating whether or not the animal has offspring: `'0'` = has no offspring, `'1'` = has offspring. The flags are set by calling `pyp_utils.set_ancestor_flag()` and passing it a renumbred pedigree. |
| animalID | animal ID | animal ID ↦ *integer* | Animal's ID. Animal IDs change when a pedigree is renumbered. IDs must be provided for all animals in a pedigree file. When strings are provided for animal IDs using the `ASD` pedigree format code they are converted to integral animal IDs using the `string_to_int()` method. |
| bd | `missing_bdate` | `missing_bdate` | The animal's birthdate in *MMDDYY* format. |
| breed | `missing_breed` | `missing_breed` | The animal's breed as a string. |
| by | `missing_byear` | `missing_byear` | The animal's birthyear in *YYYY* format. Default values set in `this typeface` are PyPedal options which are described in detail in Section 3.4. |
| damID | dam ID | dam ID ↦ *integer* | Dam's ID. When strings are provided for animal IDs using the `ASD` pedigree format code they are converted to integral animal IDs using the `string_to_int()` method. |
| damName | dam ID | dam ID | The name of the animal's dam. |
| daus | {} | {} | Dictionary containing all known daughters of an animal. The keys and values are the animal IDs for each offspring. When the pedigree is renumbered, keys are updated to correspond to the renumbered IDs for each offspring. |

*continued on next page*

| Attribute | Default | | Description |
|---|---|---|---|
| | Integral IDs (`'asd'`) | String IDs (`'ASD'`) | |
| fa | 0.0 | 0.0 | The animal's coefficient of inbreeding. |
| founder | `'n'` | `'n'` | Character indicating whether or not the animal is a founder (had unknown parents): `'n'` = not a founder (one or both parents known), `'y'` = founder (parents unknown). |
| gen | -999 | -999 | Generation to which the animal belongs. |
| gencoeff | -999.0 | -999.0 | Pattie's (1965) generation coefficient. |
| herd | `missing_herd` ↦ *integer* | `missing_herd` ↦ *integer* | The ID of the herd to which the animal belongs. |
| igen | -999 | -999 | Generation inferred by `pyp_utils.set_generation()`. |
| name | animal ID | animal ID | The animal's name. This attribute is quite useful in `ASD` pedigrees. and less so in `asd` pedigrees. |
| originalHerd | herd | herd | The original herd ID to which an animal belonged before the herd was converted from a string to an integer; most useful with the `'H'` pedigree format code. |
| originalID | animalID | animal ID ↦ *integer* | Original ID assigned to an animal. It will not change when the pedigree is renumbered. |
| paddedID | `animalID` ↦ *integer* | `animalID` ↦ *integer* | The animal ID padded to fifteen digits, with the birthyear (or 1950 if the birth year is unknown) prepended. The order of elements is: birthyear, animalID,count of zeros, zeros. Used to create alleles for gene dropping. |
| pedcomp | -999.9 | -999.9 | Pedigree completeness as described in Section 7.4.8. |
| renumberedID | -999 | -999 | ID assigned to an animal when the pedigree is renumbered. The default value indicates that the pedigree has not been renumbered using PyPedal. |
| sex | `'u'` | `'u'` | The sex of the animal: `'m'` = male, `'f'` = female, `'u'` = unknown/not provided. |
| sireID | sire ID | sire ID ↦ *integer* | Sire's ID. When strings are provided for animal IDs using the `ASD` pedigree format code they are converted to integral animal IDs using the `string_to_int()` method. |
| sireName | sireID | sireID | The name of the animal's sire. |
| sons | {} | {} | Dictionary containing all known sons of an animal. The keys and values are the animal IDs for each offspring. When the pedigree is renumbered, keys are updated to correspond to the renumbered IDs for each offspring. |

*continued on next page*

| Attribute | Default | | Description |
|---|---|---|---|
| | Integral IDs (`'asd'`) | String IDs (`'ASD'`) | |
| unks | {} | {} | Dictionary containing all offspring of an animal with unknown sex. The keys and values are the animal IDs for each offspring. When the pedigree is renumbered, keys are updated to correspond to the renumbered IDs for each offspring. |

`NewAnimal` objects have the seven methods listed in Table 6.2. The methods focus on returning information about an instance of an object; calculations are left to functions in, e.g., the `pyp_metrics` and `pyp_nrm` modules.

Table 6.2: Methods of `NewAnimal` objects.

| Method | Description |
|---|---|
| `__init__` | Initializes a `NewAnimal` object and returns an instance of a `NewAnimal` object. |
| printme | Prints a summary of the data stored in a `NewAnimal` object. |
| stringme | Returns the data stored in a `NewtAnimal` object as a string. |
| dictme | Returns the data stored in a `NewtAnimal` object as a dictionary whose keys are attribute names and whose values are attribute values. |
| trap | Checks for common errors in `NewtAnimal` objects. |
| pad_id | Takes an animal ID, pads it to fifteen digits, and prepends the birthyear. The order of elements is: birthyear, animalID, count of zeros, zeros. The padded ID is used for reordering the pedigree with the `fast_reorder` routine. |
| string_to_int | Takes an animal ID as a string and returns a hash. The algorithm used is taken from "Character String Keys" in "Data Structures and Algorithms with Object-Oriented Design Patterns in Python" by Bruno R. Preiss: `http://www.brpreiss.com/books/opus7/html/page220.html#progstrnga`. |

### 6.1.2 The LightAnimal Class

The `LightAnimal` class holds animals records read from a pedigree file. It implements a much simpler object than the `NewAnimal` object and is intended for use with the graph theoretic routines in `pyp_network`. The only attributes of these objects are animal ID, sire ID, dam ID, original ID, renumbered ID, birth year, and sex (Table 6.3).

Table 6.3: Attributes of `LightAnimal` objects.

| Attribute | Default | Description |
|---|---|---|
| animalID | animal ID | Animal's ID. |
| by | `missing_byear` | The animal's birthyear in *YYYY* format. |
| damID | 0 | Dam's ID. |

*continued on next page*

| Attribute | Default | Description |
|---|---|---|
| originalID | animalID | Original ID assigned to an animal. It will not change when the pedigree is renumbered. |
| renumberedID | animalID | Renumbered ID assigned to an animal. It is assigned by the renumbering routine. |
| sex | `'u'` | The sex of the animal: `'m'` = male, `'f'` = female, `'u'` = unknown. |
| sireID | 0 | Sire's ID. |

`LightAnimal` objects have the same seven methods (Table 6.4) as `NewAnimal` objects (Table 6.2).

Table 6.4: Methods of `LightAnimal` objects.

| Method | Description |
|---|---|
| __init__ | Initializes a `LightAnimal` object and returns an instance of a `LightAnimal` object. |
| printme | Prints a summary of the data stored in a `LightAnimal` object. |
| stringme | Returns the data stored in a `LightAnimal` object as a string. |
| dictme | Returns the data stored in a `LightAnimal` object as a dictionary whose keys are attribute names and whose values are attribute values. |
| trap | Checks for common errors in `LightAnimal` objects. |
| pad_id | Takes an animal ID, pads it to fifteen digits, and prepends the birthyear. The order of elements is: birthyear, animalID, count of zeros, zeros. The padded ID is used for reordering the pedigree with the `fast_reorder` routine. |
| string_to_int | Takes an animal ID as a string and returns a hash. The algorithm used is taken from "Character String Keys" in "Data Structures and Algorithms with Object-Oriented Design Patterns in Python" by Bruno R. Preiss: http://www.brpreiss.com/books/opus7/html/page220.html#progstrnga. |

### 6.1.3 The SimAnimal Class

The `SimAnimal` class is used for pedigree simulation, which is described in Section 3.8. All simulated pedigrees have the format code `asdxg`, and those are the only class attributes (Table 6.5). This class is intended for use only by the pedigree simulation routines, so the lack of attributes and methods as compared to the `NewAnimal` class is a deliberate design decision.

Table 6.5: Attributes of `SimAnimal` objects.

| Attribute | Default | Description |
|---|---|---|
| animalID | animal ID | Animal's ID. |
| damID | 0 | Dam's ID. |
| gen | 0 | Generation to which the animal belongs. |
| sex | `'u'` | The sex of the animal: `'m'` = male, `'f'` = female, `'u'` = unknown. |

*continued on next page*

| Attribute | Default | Description |
|---|---|---|
| sireID | 0 | Sire's ID. |

`SimAnimal` objects have only three methods (Table 6.6).

Table 6.6: Methods of `SimAnimal` objects.

| Method | Description |
|---|---|
| __init__ | Initializes a `SimAnimal` object and returns an instance of a `SimAnimal` object. |
| printme | Prints a summary of the data stored in a `SimAnimal` object. |
| stringme | Returns the data stored in a `SimAnimal` object as a string. |

## 6.2 The NewPedigree Class

The `NewPedigree` class is the fundamental object in PyPedal.

Table 6.7: Attributes of `NewPedigree` objects.

| Attribute | Default | Description |
|---|---|---|
| kw | `kw` | Keyword dictionary. |
| pedigree | [] | A list of `NewAnimal` objects. |
| metadata | {} | A `PedigreeMetadata` object. |
| idmap | {} | Dictionary for mapping original IDs to renumbered IDs (7.2). |
| backmap | {} | Dictionary for mapping renumbered IDs to original IDs (7.2). |
| namemap | {} | Dictionary for mapping names to original IDs (7.2). |
| namebackmap | {} | Dictionary for mapping original IDs to names (7.2). |
| starline | `'*'*80` | Convenience string. |
| nrm | None | An instance of a `NewAMatrix` object. |

The methods of `NewPedigree` objects are listed in Table 6.8. !!!I need to put something in here about pedsources and make sure that it's in the index!!!

Table 6.8: Methods of `NewPedigree` objects.

| Method | Description |
|---|---|
| __init__ | Initializes and returns a `NewPedigree` object. |
| addanimal | Adds a new animal of class `NewAnimal` to the pedigree. **Note:** This function should be used by `NewPedigree` methods only, not userspace routines. Improper use of `addanimal` may result in data loss or corruption. You have been warned. |

*continued on next page*

| Method | Description |
|---|---|
| delanimal | Deletes an animal from the pedigree. Note that this method DOES not update the metadata attached to the pedigree and should only be used if that is not important. **Note:** This function should be used by `NewPedigree` methods only, not userspace routines. Improper use of `delanimal` may result in data loss or corruption. You have been warned. |
| fromgraph | Creates a `NewPedigree` from an `XDiGraph` objject. |
| 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 metadata, checking for common errors, setting ancestor and sex flags, and renumbering the pedigree. |
| preprocess | Processes the entries in a pedigree file, which includes reading each entry, checking it for common errors, and instantiating a `NewAnimal` object. |
| printoptions | Prints the contents of the options dictionary, which is useful for debugging. |
| renumber | Calls the proper reordering and renumbering routines; updates the ID map after a pedigree has been renumbered so that all references are to renumbered rather than original IDs. |
| 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. |
| simulate | Simulate simulates an arbitrary pedigree of size $n$ with $g$ generations starting from $n\_s$ base sires and $n\_d$ base dams. This method is based on the concepts and algorithms in the `Pedigree::sample` method from Matvec 1.1a (src/classes/pedigree.cpp; http://statistics.unl.edu/faculty/steve/software/matvec/), although all of the code in this implementation was written from scratch. |
| updateidmap | Updates the ID map after a pedigree has been renumbered so that all references are to renumbered rather than original IDs. |

See Section 3.8 for details on pedigree simulation.

## 6.3 The PedigreeMetadata Class

The `PedigreeMetadata` class stores metadata about pedigrees. This helps improve performance in some procedures, and also makes it easy to access useful summary data. Metadata are collected when the pedigree is loaded and accessed by many PyPedal routines.

Table 6.9: Attributes of `PedigreeMetadata` objects.

| Attribute | Default | Description |
|---|---|---|
| *continued on next page* | | |

| Attribute | Default | Description |
|---|---|---|
| name | `pedname` | Name assigned to the pedigree. |
| filename | `pedfile` | File from which the pedigree was loaded. |
| pedcode | `pedformat` | Pedigree format string. |
| num_records | 0 | Number of records in the pedigree. |
| num_unique_sires | 0 | Number of unique sires in the pedigree. |
| num_unique_dams | 0 | Number of unique dams in the pedigree. |
| num_unique_founders | 0 | Number of unique founders in the pedigree. |
| num_unique_gens | 0 | Number of unique generations in the pedigree. |
| num_unique_years | 0 | Number of unique birth years in the pedigree. |
| num_unique_herds | 0 | Number of unique herds in the pedigree. |
| unique_sire_list | [] | List of the unique sires in the pedigree. |
| unique_dam_list | [] | List of the unique dams in the pedigree. |
| unique_founder_list | [] | List of the unique founders in the pedigree. |
| unique_gen_list | [] | List of the unique generations in the pedigree. |
| unique_year_list | [] | List of the unique birth years in the pedigree. |
| unique_herd_list | [] | List of the unique herds in the pedigree. |

Metadata are gathered furing the pedigree loading process, but after load-time renumbering has occured (if requested). When a pedigree is renumbered after it has been loaded the unique sire, dam, and founders lists are not updated to contain the renumbere IDs. The metadata may be updated by instantiating a new `PedigreeMetadata` object and using it to replace the original metadata:

```
example.metadata = PedigreeMetadata(example.pedigree,example.kw)
```

Alternatively, ID maps (Section 7.2) may be used to produce expected lists of animals.

The methods of `PedigreeMetadata` objects are listed in Table 6.10. The couting methods (`nud`, `nuf`, etc.) return two values each, a count and a list, and new couting methods may easily be added.

Table 6.10: Methods of `PedigreeMetadata` objects.

| Method | Description |
|---|---|
| __init__ | Initializes and returns a `PedigreeMetadata` object. |
| fileme | Writes the metada stored in the `PedigreeMetadata` object to disc. |
| nud | Returns the number of unique dams in the pedigree along with a list of the dams. |
| nuf | Returns the number of unique founders in the pedigree along with a list of the founders. |
| nug | Returns the number of unique generations in the pedigree along with a list of the generations. |
| nuherds | Returns the number of unique herds in the pedigree along with a list of the herds. |

*continued on next page*

| Method | Description |
|---|---|
| nus | Returns the number of unique sires in the pedigree along with a list of the sires. |
| nuy | Returns the number of unique birth years in the pedigree along with a list of the birth years. |
| printme | Prints a summary of the pedigree metadata stored in the `PedigreeMetadata` object. |
| stringme | Returns a summary of the pedigree metadata stored in the `PedigreeMetadata` object as a string. |

## 6.4 The NewAMatrix Class

The `NewAMatrix` class provides an instance of a numerator relationship matrix as a NumPy 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.

Table 6.11: Attributes of `NewAMatrix` objects.

| Attribute | Default | Description |
|---|---|---|
| kw | `kw` | Keyword dictionary. |
| nrm | `None` | A numerator relationship matrix; exists only after the `form_a_matrix` method has been called. |

The methods of `NewAMatrix` objects are listed in Table 6.12.

Table 6.12: Methods of `NewAMatrix` objects.

| Method | Description |
|---|---|
| __init__ | Initializes and returns a `NewAMatrix` object. |
| form_a_matrix | Calls `pyp_nrm.fast_a_matrix()` or `pyp_nrm.fast_a_matrix_r()` to form a NRM from a pedigree. |
| info | Uses the NRM's `info()` method to dump some information about the NRM. This is useful for debugging. |
| load | Uses the NumPy function `fromfile` to load an array from a binary file. If the load is successful, self.nrm contains the matrix. |
| save | Uses the NRM's `tofile()` method to save an array to either a text or binary file. |
| printme | Prints the NRM to the screen. |

CHAPTER

**SEVEN**

# Methodology

To iterate is human, to recurse divine. — L. Peter Deutsch

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.

## 7.1 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. Founders are also grouped together at the beginning of the pedigree. 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. Founders are not guaranteed to be grouped at the beginning of the pedigree when `fast_reorder()` is used; if you are going to calculate coefficients of partial inbreeding (Section 7.4.3) then you should instead use `reorder()` to reorder your pedigree.

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.

## 7.2 Animal Identification and Cross-References

There are a number of identification attributes associated with animal objects in PyPedal pedigrees. A description of those fields, as well as their default values, is provided in Table 6.1. This section describes the data structures provided for mapping between various animal IDs. Table 7.1 lists the four structures provided for ID mapping and lists the default values for pedigrees with integral (`asd` formats) and string (`ASD` formats) IDs. Renumbered PyPedal pedigree objects store animal objects in a list that is typically indexed by renumbered ID. When animal IDs are strings (see 3.5.1) they are hashed to an integer and the original ID is stored in a name field. In renumbered pedigrees the original ID is stored and replaced by a renumbered ID.

Once your pedigree is renumbered it is quite easy to see how these maps can be used to convert between various IDs. The maps mean that you don't have to worry about renumbered IDs and can continue to think about your animals in terms of their original IDs, whether they be ID numbers or names. Consider the Newfoundland pedigree presented in Figure 9.4 – it is much more convenient to think about the dog named Kaptn Kvols von Widdersdorf, rather than the dog whose name was hashed to the ID 5523557808241831142 amd renumbered to 48. For example, suppose you wanted to determine his coefficient of inbreeding. It is simple to do using the maps:

```
>>> example = pyp_newclasses.loadPedigree(optionsfile='newfoundland.ini')
>>> newf_f = pyp_nrm.inbreeding(example)
>>> print newf_f['fx'][example.idmap[example.namemap['Kaptn Kvols von Widdersdorf']]]

0.0
```

`example.namemap['Kaptn Kvols von Widdersdorf']` returns the original ID assigned to the name, while `example.idmap[...]` converts from the original ID to the renumbered ID. This sort of ID/name mapping is sued in a number of places in PyPedal, such as in the three generation pedigree routine in the `pyp_reports` module.

Note that if an animal has its original ID as its name, which is the default when integral IDs and no animal names are provided, the name is changed to the renumbered ID when the pedigree is renumbered.

Table 7.1: Animal identification and cross-references.

| Map name | `asd` format | | `ASD` format | | Renumbered | |
|---|---|---|---|---|---|---|
| | Key | Value | Key | Value | Key | Value |
| idmap | animal ID | animal ID | name | name | original ID | renumbered ID |
| backmap | animal ID | animal ID | name | name | renumbered ID | original ID |
| namemap | name | animal ID | name | name | name | original ID |
| namebackmap | animal ID | name | name | name | original ID | name |

## 7.3 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.

## 7.4 Computational Details

### 7.4.1 Inbreeding and Related Measures

Coefficients of relationship ($r_{ij}$) and inbreeding ($f_i$) 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 $\Delta f$ is the change in population average inbreeding between generations *t* and *t+1*. Interpretation of $N_{e(t)}$ can be problematic when $\Delta f$ is calculated from incomplete or error-prone pedigrees.

### 7.4.2 Ancestral Inbreeding

Ballou (1997) described ancestral inbreeding, the probability that an individual inherited an allele that had undergone inbreeding in the past at least once, in a study of purging recessives and inbreeding in conservation genetics. This is a different idea than the usual coefficient of inbreeding as an individual that is not inbred may carry alleles that have been exposed to substantial inbreeding; recall that an individual may have inbred parents, but if the parents are not related to one another then the resulting offspring will not be themselves inbred. It has been proposed that animals deriving from highly inbred lines may be less susceptible to inbreeding depression because deleterious recessive alleles have been purged from the population, but literature reports on extant populations are inconsistent. Ancestral inbreeding is calculated using a recursion equation as: $f_a = [f_{a(s)} + (1 - f_{a(s)})f_s + f_{a(d)} + (1 - f_{a(d)})f_d]/2$ where $f_a$ is the ancestral inbreeding coefficient for an individual, $f$ is the usual coefficient of inbreeding, and subscripts $s$ and $d$ represent sires and dams, respectively. Calculations are from oldest to youngest in the population and require as inputs only coefficients of inbreeding.

Suwanlee et al. (2007) extended the concept by presenting a gene-dropping approach (MacCluer, VandeBerg, Read, and Ryder 1986) for calculating ancestral inbreeding, as well as modifying Ballou's equation to account for non-independence between individual inbreeding coefficients and ancestral inbreeding coefficients. Gene dropping is also used in PyPedal for calculating founder genome equivalents (see: Section 7.4.6), and the code used by the `pyp_metrics.dropped_ancestral_inbreeding()` and `pyp_metrics.effective_founder_genomes()` routines are similar, although the former drops an arbitrary number of unlinked biallelic loci through the pedigree while the latter drops only a single locus.

### 7.4.3 Partial Inbreeding

Partial inbreeding coefficients, $F_{ij}$, measure the probability that the alleles at an arbitrary locus in individual $i$ are identitical-by-descent and that the alleles were derived from an allele in founder $j$ (Lacy, Alaks, and Walsh 1996); Gulisija et al. (2006) provide an excellent description of the tabular method for calculating $F_{ij}$. Computational requirements may be high for large pedigrees with a large number of founders because partial kinship matrices are calculated for each founder in the pedigree. The usual coefficients of inbreeding may be obtained by summing the coefficients of partial inbreeding over all founders common to the parents of animal $i$, that is, $f_i = \sum_j F_{ij}$.

For example, consider the pedigree presented in Figure 2 of Gulisija and Crow (2007). The individual of interest, $I$, has an inbreeding coefficient of 0.375 and coefficients of partial inbreeding to founders J, K, and M of 0.21875,0.09375, and 0.0625, respectively. As asserted, $0.21875 + 0.09375 + 0.0625 = 0.375$.

### 7.4.4 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.

### 7.4.5 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 7.4.1 for details), is also used for calculating $f_e$.

### 7.4.6 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).

### 7.4.7 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.

### 7.4.8 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. Pedigree completeness and ancestor loss coefficients (`http://www.newfoundlanddog-database.net/en/ahnen.php?num=0000025330`), which are sometimes seen in dog breeding materials, are equivalent measures if the same number of generations were used in the calculations.

CHAPTER

**EIGHT**

# HOWTOs

Computers are good at following instructions, but not at reading your mind. — Donald E. Knuth

## 8.1 Basic Tasks

### 8.1.1 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.loadPedigree(options)
```

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.

### 8.1.2 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.loadPedigree(options)

#   Read the second pedigree
options['pedfile'] = 'new_boichard.ped'
options['pedformat'] = 'asdg'
options['pedname'] = 'Boichard Pedigree'
example2 = pyp_newclasses.loadPedigree(options)
```

Note that *pedformat* only needs to be changed if the two pedigrees have different formats. You do not even have to change *pedfile*.

### 8.1.3 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'
example1 = pyp_newclasses.loadPedigree(options)
```

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 3.6.

### 8.1.4 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.

### 8.1.5 How do I load a pedigree whose columns are tab-delimited?

The default column-delimiter used by PyPedal is a space (‘ ’). You can change the delimiter by setting the *sepchar* option:

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

Commas are also commonly used as delimiters, and comma-separated value (CSV) files can be read by setting *sepchar* to `','`. If you are using a configuration file, you *must* enclose any delimiter containing a backslash in double quotation marks (“”):

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

If you do not enclose the delimiter properly you will receive an error message such as:

```
[jcole@jcole2 examples]$ python new_ids.py
[INFO]: Logfile new_ids2.log instantiated.
[INFO]: Preprocessing new_ids2.ped
[INFO]: Opening pedigree file
[ERROR]: The record on line 2 of file new_ids2.ped does not have the same number
         of columns (1) as the pedigree format string (ASD) says that it should
         (3). Please check your pedigree file and the pedigree format string for
         errors.
[jcole@jcole2 examples]$
```

## 8.2 Calculating Measures of Genetic Variation

### 8.2.1 How do I calculate coefficients of inbreeding?

This requires that you have a renumbered pedigree (HOWTO 8.1.3).

```
options = {}
options['renumber'] = 1
options['pedfile'] = 'new_lacy.ped'
options['pedformat'] = 'asd'
example1 = pyp_newclasses.loadPedigree(options)
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'
```

## 8.3 Databases and Report Generation

### 8.3.1 How do I load a pedigree into a database?

The `pyp_reports` module (**??**) uses the `pyp_db` module (Section **??**) 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.loadPedigree(options)

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.

### 8.3.2 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.loadPedigree(options)

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!
```

## 8.4 Pedigrees as Graphs

PyPedal includes tools for working with pedigrees as algebraic structures known as directed graphs, or digraphs. Digraphs are not graphs in the sense of graphics for presentation or display. Rather, they are mathematical abstractions, the study of which can provide interesting information about the structure of a population. A digraph represents a pedigree as a set of vertices (also called nodes), which correspond to animals, and a collection of edges, which connect nodes to one another. In the context of a pedigree, edges indicate that a parent–offspring relationship exists between two animals. If a path can be constructed between two vertices (animals) in the graph then those animals are related. If no path can be constructed between teo nodes, then no relatinship exists between the two. Routines for working with graphs (also called networwks) are contained in the `pyp_network` module (**??**)

### 8.4.1 How do I convert a pedigree to a graph?

The function `pyp_network.ped_to_graph()` takes a PyPedal pedigree object as its argument and returns a NetworkX (`https://networkx.lanl.gov/`) XDiGraph object:

```
example = pyp_newclasses.loadPedigree(optionsfile='new_networkx.ini')
ng = pyp_network.ped_to_graph(example)
```

Once you've got a graph, you use the NetworkX API to operate on the graph. For example, the number of animals in the pedigree is simply the number of nodes in the graph:

```
print 'Number of animals in pedigree: %s' % ( ng.order() )
print ng.nodes()
```

### 8.4.2 How do I convert a graph to a pedigree?

It is possible to create a PyPedal pedigree from a NetworkX graph. This is useful, for example, when you'd like to create a pedigree representing a subset of the population in a `NewPedigree` object. `pyp_nrm.recurse_pedigree()` and related functions won't do the trick because they return only lists of animal IDs, not actual `NewPedigree` instances. To create a pedigree from a graph you simply build your options dictionary and call `pyp_newclasses.loadPedigree()`:

```
options = {}
options['pedfile'] = 'dummy'
options['messages'] = 'verbose'
options['renumber'] = 1
options['pedname'] = 'Testing fromgraph()'
options['pedformat'] = 'asd'
options['set_offspring'] = 1
options['set_ancestors'] = 1
options['set_sexes'] = 1
options['set_generations'] = 1
example2 = pyp_newclasses.loadPedigree(options,pedsource='graph',pedgraph=ng)
```

You must provide a non-null `pedfile` keyword in your options dictionary, as well as the `pedsource` and `pedgraph` arguments to `pyp_newclasses.loadPedigree()`.

There is a known bug with logfile creation when loading a pedigree from a digraph.

### 8.4.3 How do I load a pedigree from a file containing a graph stored as an adjacency list?

PyPedal can read graphs stored in text files as adjacency lists, which is one way of representing directed graphs:

```
options = {}
options['pedfile'] = 'pedigree.adjlist'
options['messages'] = 'verbose'
options['pedname'] = 'Testing graphfile'
options['pedformat'] = 'asd'
example = pyp_newclasses.loadPedigree(options,pedsource='graphfile')
```

### 8.4.4 How do I save a graph as an adjacency list?

PyPedal, using NetworkX, can save graphs as adjacency lists:

```
example.savegraph('pedigree.adjlist')
```

## 8.5 Miscellaneous

### 8.5.1 How do I export a numerator relationship matrix so that I can read it into Octave?

Numerator relationship matrices may be exported to text files for use with, e.g., Octave using the `NewAMatrix.save` method:

```
example = pyp_newclasses.loadPedigree(optionsfile='denny.ini')
amatrix = pyp_newclasses.NewAMatrix(example.kw)
amatrix.form_a_matrix(example.pedigree)
amatrix.tofile('Ainv.txt')
```

When matrices are written to text files array elements are separated by *sepchar* (Table 3.1).

Matrices may also be written to a binary format. The default value of the *nrm_format* pedigree option is `text`. To write files in binary format you must either specify the value of the *nrm_format* option as `binary` before you load your pedigree file or use the *nrm_format* keyword when you call `NewAMatrix.save`:

```
amatrix.tofile('Ainv.bin',nrm_format='binary')
```

Once you've saved the NRM to a file, say `'Ainv.txt'`, in text format you can easily read it into Octave:

```
octave:1> myfile = fopen ("Ainv.txt", "r");
octave:2> ainv = fscanf(myfile,'%f',[18,18])
```

This has been verified with Octave 2.1.57 under RedHat Enterprise Linux on small matrices.

### 8.5.2 How else can I export a NRM to a file?

Numerator relationship matrices may be exported to a text file in "ijk format", where each line is of the form "animal_A animal_B rAB" using the `pyp_io.save_ijk()` function. Diagonal entries are $1 + f_a$, where $f_a$ is the animal's coefficient of inbreedin.

```
example = pyp_newclasses.loadPedigree(options)
# Save the NRM to a file in ijk format.
# Don't forget to set the filename.
pyp_io.save_ijk(example,'nrm_ijk.txt')
```

Suppose that the example above produces the following file:

```
$ head nrm_ijk.txt
4627 4627 1.125
4627 0832 0.0
4627 5538 0.5
...
```

In order to get $f_a$ for animal 4627 you need to find the corresponding diagonal element and subtract 1 from it:

$$f_a = 1.125 - 1.0 = 0.125$$

The coefficient of relationship between 4627 and 5538 is $0.5$ (4627 is probably a parent of 5538). Note that the file *nrm_ijk.txt* will include only the diagonal and upper off-diagonal elements of the NRM, and should have $n+n(n+1)/2$ lines.

### 8.5.3 How do I load a pedigree from a GEDCOM file?

As of version 2 release candidate 1 PyPedal can load pedigrees from GEDCOM 5.5 files. This is done by passing the `pedsource` keyword to `pyp_newclasses.loadPedigree` with a value of '`gedcomfile`':

```
options['pedfile'] = 'example2.ged'
options['pedformat'] = 'ASD'
options['pedname'] = 'A GEDCOM pedigree'
example2 = pyp_newclasses.loadPedigree(options,pedsource='gedcomfile')
```

Note that only a limited subset of the GEDCOM format is supported, and it is possible to lose metadata when converting a pedigree from GEDCOM to PyPedal. More details on PyPedal's GEDCOM handling can be found in Appendix B.

### 8.5.4 How do I save a pedigree to a GEDCOM file?

As of version 2 release candidate 1 PyPedal can write pedigrees to GEDCOM 5.5-formatted files using the `savegedcom` method of `pyp_newclasses.NewPedigree` objects. The method takes an output file name as its argument:

```
test.savegedcom('ged3.pypedal.ged')
```

Note that not all attributes of `pyp_newclasses.NewAnimal` objects are written to the output file. More details on PyPedal's GEDCOM handling can be found in Appendix B.

### 8.5.5 How do I load a pedigree from a string?

There are some use cases for which it is desirable to load pedigrees from strings rather than from files. This is done by passing the `pedsource` keyword to `pyp_newclasses.loadPedigree` with a value of '`textstream`', along with a string named '`pedstream`' (Figure 8.1):

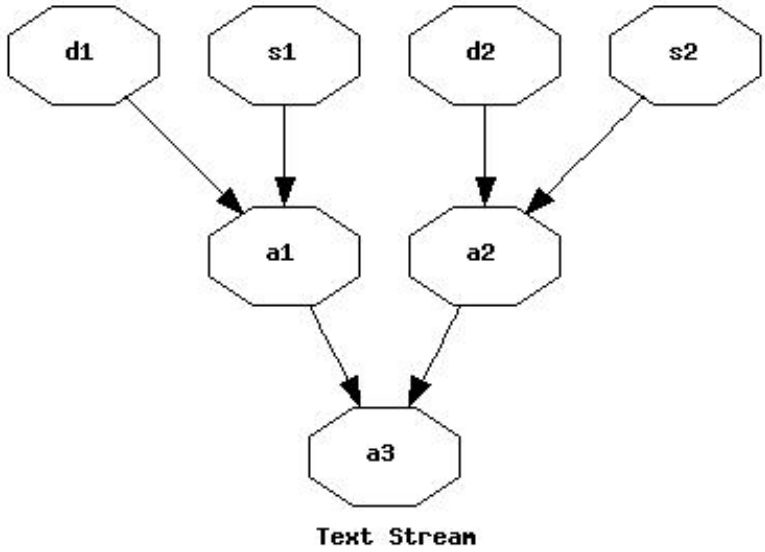


Figure 8.1: Pedigree loaded from a string

```
options = {}
options['pedfile'] = ''
options['renumber'] = 1
options['pedformat'] = 'ASD'
if __name__ == "__main__":
    pedstream = 'a1,s1,d1\na2,s2,d2\na3,a1,a2\n'
    test = pyp_newclasses.loadPedigree(options,pedsource='textstream',pedstream=pedstream)
    pyp_graphics.new_draw_pedigree(test, gfilename='partial', gtitle='Text Stream', gorient='p
```

Note that only ASD-formatted pedigrees can be loaded this way, individual IDs are separated with commas, and successive records are separated by newlines. All records must contain a newline, including the last record in the string! You must also set the '`pedfile`' option to a value, even if that value is just an empty string as in the example.

## 8.6 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

**NINE**

# Graphics

If I could say it in words there would be no reason to paint. — Edward Hopper

## 9.1 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 9.1.

Table 9.1: Default graphics formats.

| Routine | Default Format | Supported Formats |
|---|---|---|
| draw_pedigree | JPG | JPG, PNG, PS |
| new_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 |

### 9.1.1 Drawing Pedigrees

The pedigree from Figure 2 in Boichard et al. (1997) is shown in Figure 9.1, and shows males enclosed in rectangles and females in ovals. Figure 9.2 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 9.3; the code used to create this pedigree is:

```
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 9.1 that the default file format for `draw_-`

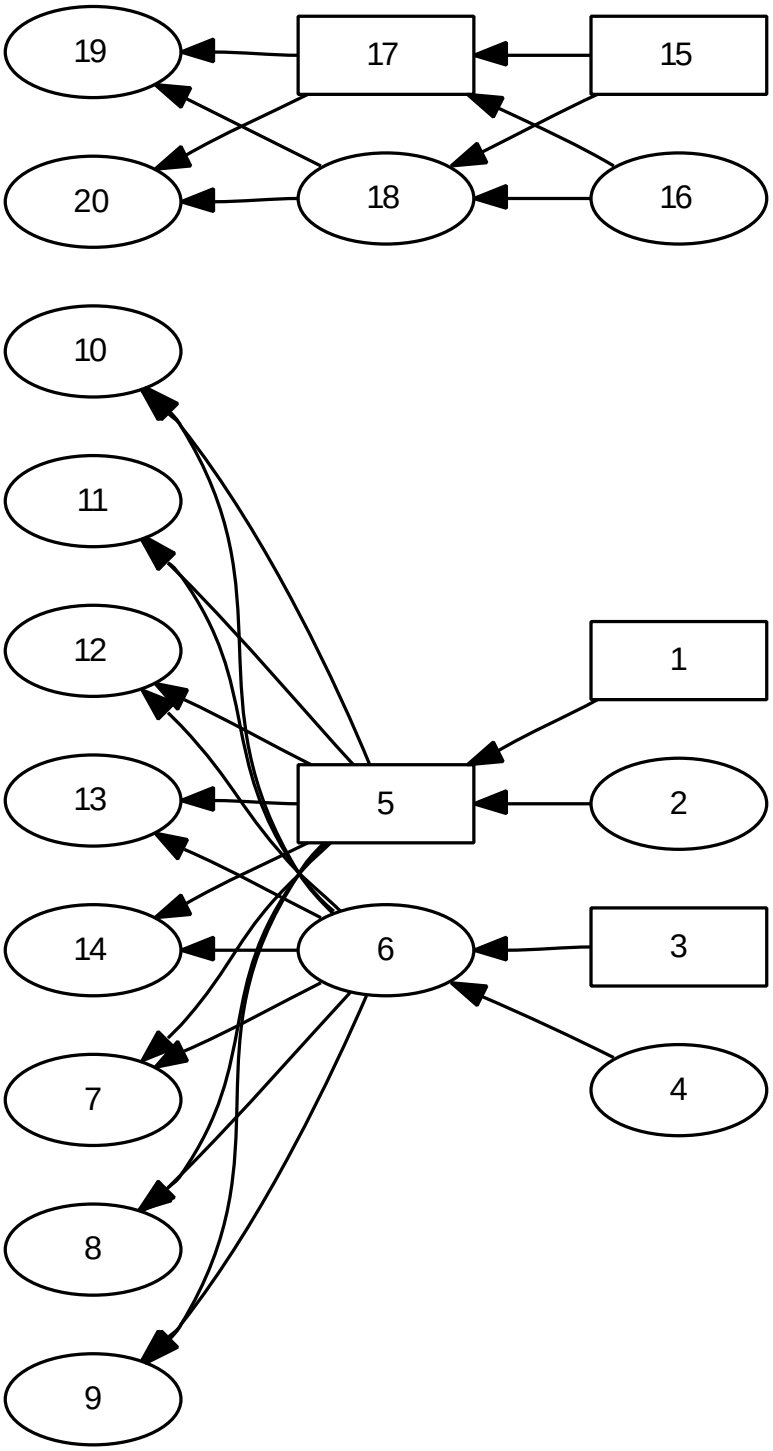


Figure 9.1: Pedigree 2 from Boichard et al. (1997)

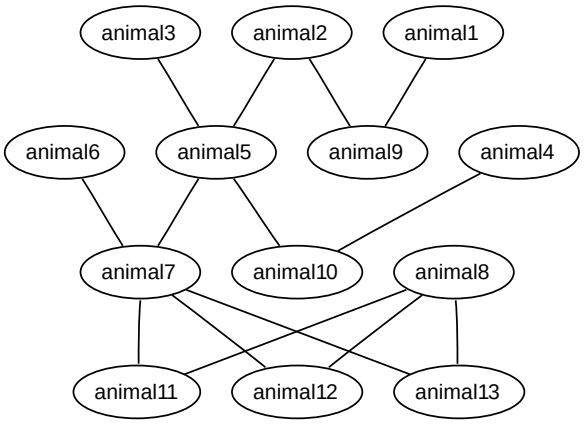


Figure 9.2: A pedigree with strings as animal IDs

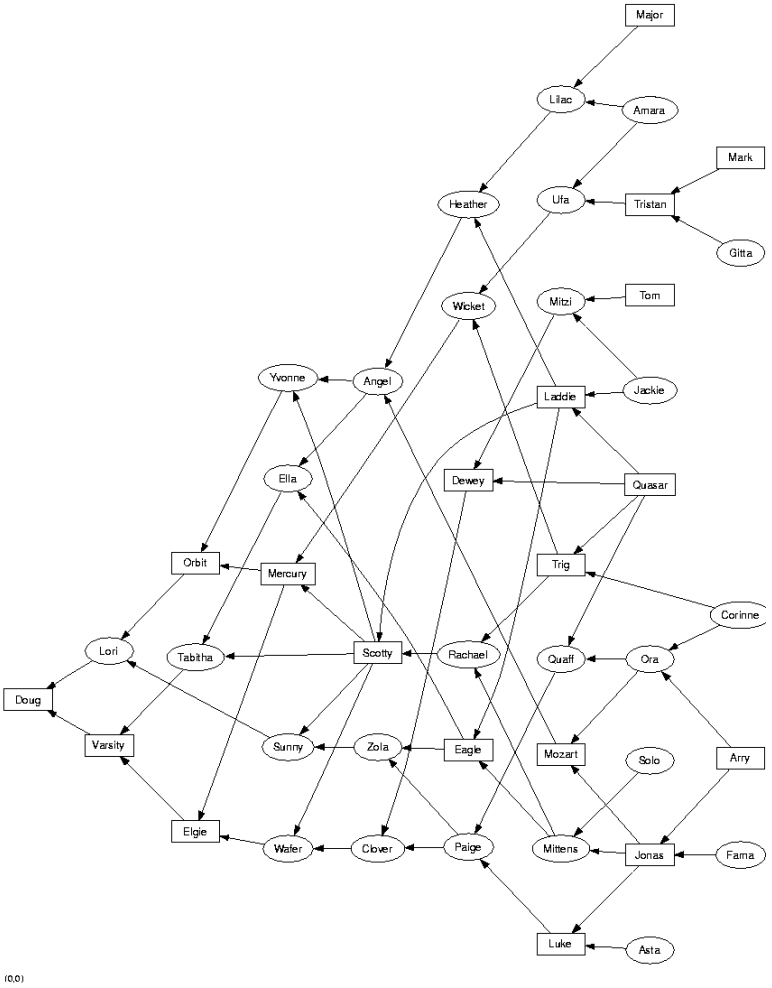


Figure 9.3: German Shepherd pedigree

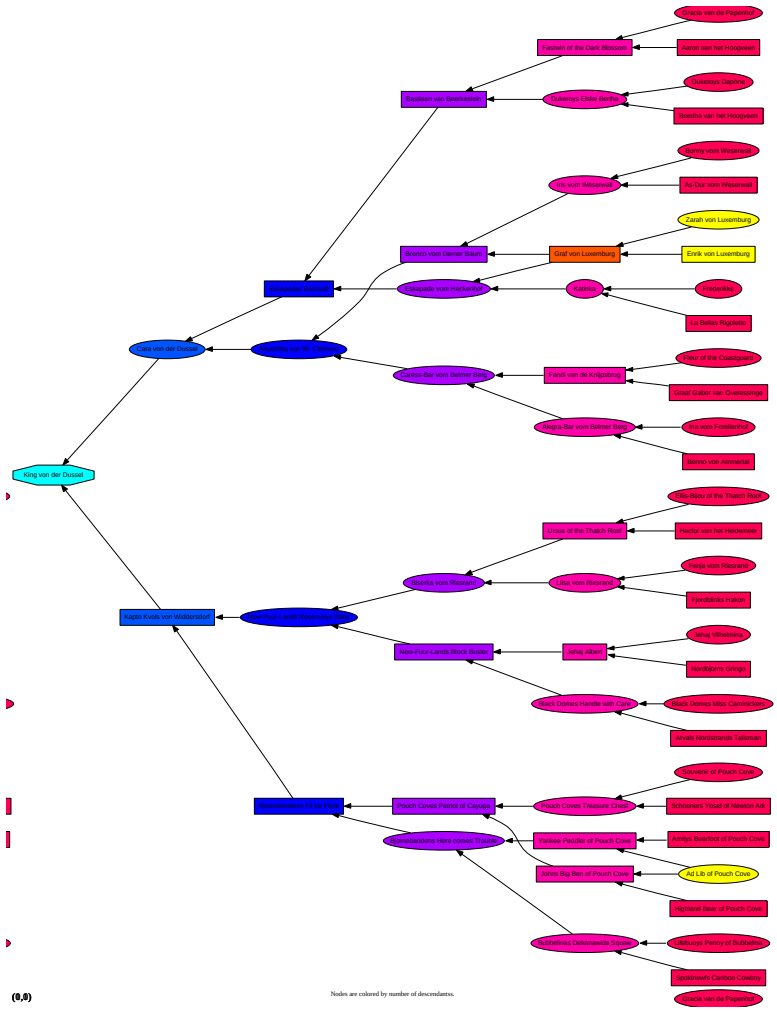


Figure 9.4: Newfoundland colored pedigree

`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 **??**). `draw_pedigree()` uses rectangles to indicates known males, circles to indicate known females, and octagons to indicate animals of unknown sex.

Pedigrees can also be colored using the `color_pedigree()` function in the `pyp_jbc` module. At present, animals are shaded either by the number of sons produced or by the total number of descendants. The five-generation pedigree of the Newfoundland dog King von der Düssel is presented in Figure 9.4 (`http://www.newfoundlanddog-database.net/en/ahnen.php?num=0000025330`, data used with permission), and the nodes are shaded based on number of descendants.

**FIX ►►► Windows users should set the *drawers* keyword to 'old' when calling `color_pedigree()`. This will call `draw_colored_pedigree()` rather than `new_draw_colored_pedigree()`. The latter requires that PyGraphviz library be installed and there is not yet an easy way to install it on Windows. ◄◄◄**

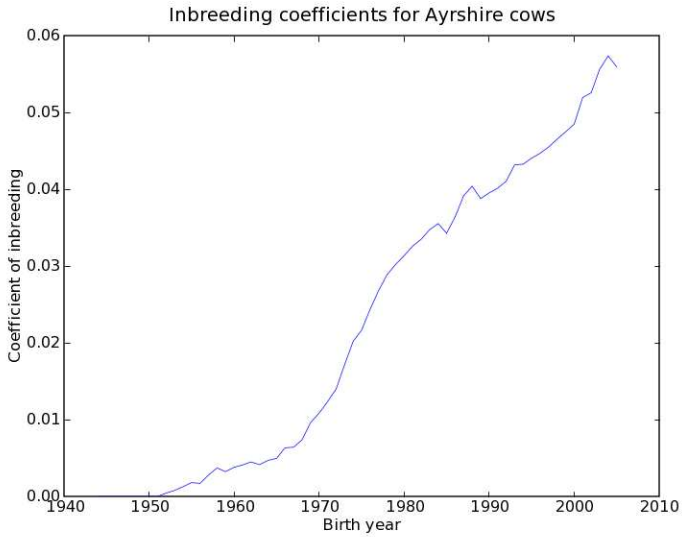


Figure 9.5: Average inbreeding by birth year for the US Ayrshire cattle population

### 9.1.2 Drawing Line Graphs

The `plot_line_xy()` routine provides a convenient tool for creating two-dimensional line graphs. Figure 9.5 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 **??**) 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 11) 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 (**??**).

### 9.1.3 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 9.1. The matrix is square and symmetric; the

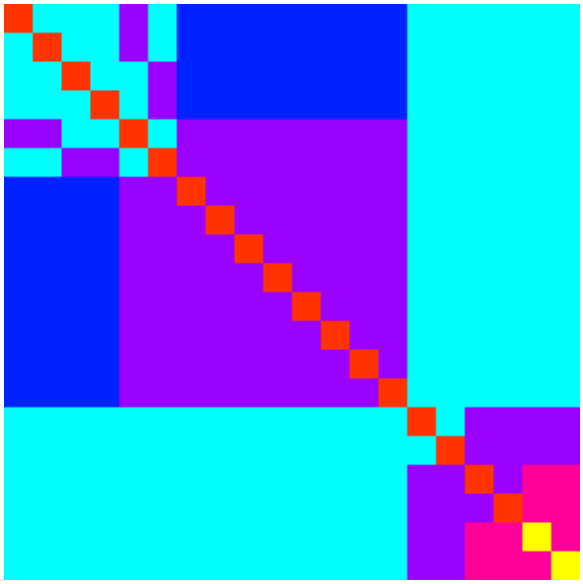

Figure 9.6: Pseudocolored NRM from the Boichard et al. (1997) pedigree

diagonal values correspond to $1 + f_a$, where $f_a$ is an animal's coefficient of inbreeding; animals with a diagonal 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 9.6). 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 9.7 it is easy to see the two separate families in the pedigree.

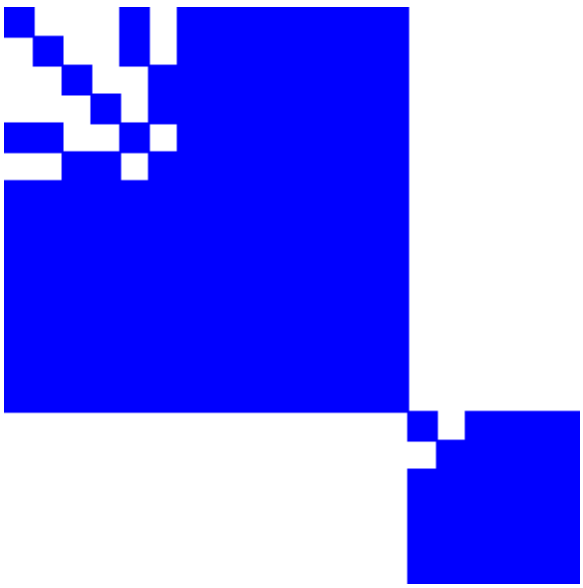

Figure 9.7: Sparsity of the NRM from the Boichard et al. (1997) pedigree

CHAPTER

**TEN**

# Report Generation

If we spoke a different language, we would perceive a somewhat different world. — Ludwig Wittgenstein

## 10.1 Overview

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

PyPedal has a framework in place to support basic report generation. This franework consists of two components: a database access module, `pyp_db` (Section **??**), and a reporting module, `pyp_reports` (Section **??**). 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 10.2 and 10.3 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 11.

Column names, data types, and descriptions of contents for pedigree tables are presented in Table 10.1. 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 10.1 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.

Table 10.1: Columns in pedigree database tables.

| Name | Type | Note(s) |
|---|---|---|
| age | real | Age of animal |
| alive | char(1) | Animal's mortality status |

*continued on next page*

| Name | Type | Note(s) |
|---|---|---|
| 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 |

### 10.1.1 Three Generation Pedigrees

A report for producing three-generation pedigrees, `pdf3GenPed()`, is included in the `pyp_reports` module. The sample output shown in Figure 10.1 contains output for one animal. However, if `pdf3GenPed()` is passed a list of animal IDs the resulting PDF will contain a pedigree for each animal that can be printed as a booklet. See Section **??** for usage details.

## 10.2 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 9.1.1). That approach is outlined in code below.


Figure 10.1: Example of a printed three generation pedigree.

```
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.

## 10.3 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 10.2. 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

**ELEVEN**

# Implementing New Features

First, solve the problem. Then, write the code. — John Johnson

## 11.1 Overview

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.

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.

### 11.1.1 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 11.2, I will present a detailed solution to this problem in Section 11.3.

## 11.2 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
```

## 11.3 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.loadPedigree(options)
    pyp_jbc.color_pedigree(example)
```

The resulting colorized pedigree can be seen in Figure 11.1. 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.

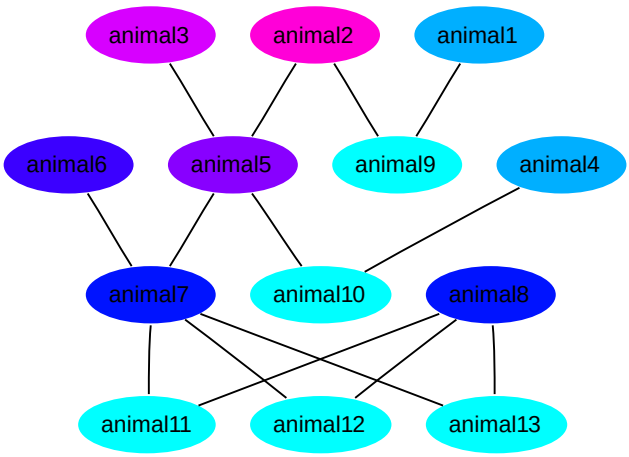


Figure 11.1: Colorized version of the pedigree in Figure 9.2

## 11.4 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

**TWELVE**

# Glossary

> Just as birds have wings, man has language. — George Henry Lewes

This chapter provides a glossary of terms.[1]

**ancestor loss coefficient** See: pedigree completeness.

**coefficient of ancestral inbreeding** The probability that an individual jas inherited an allele that has undergone inbreeding in the past at least once.

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

**coefficient of partial inbreeding** The probability that the alleles at an arbitrary locus in an individual are identitical-by-descent, and that the alleles were derived from an allele in a particular founder.

**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.

**pedigree completeness** The proportion of known pedigree information for an arbitrary number of generations.

**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.

**reordering** The process of arranging animals in a pedigree so that parents appear before their offspring; this is a necessary step in renumbering a pedigree.

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

APPENDIX

**A**

# Example Programs

Either I've invented a whole new logic or, ahem, I'm not playing with a full deck. — Philip K. Dick

A number of example programs are distributed with PyPedal. Table A.1 provides the name of each file, the configuration and pedigree files used by those programs, and a brief description of the concepts and techniques presented.

Table A.1: Example programs distributed with PyPedal.

| Program Name | Configuration File | Pedigree File | Description |
|---|---|---|---|
| new_amatrix.py | new_amatrix.ini | new_amatrix.ped | Create, save, load, and view information about NewAMatrix objects |
| new_classes.py | new_classes.ini | boichard2.ped | ??? |
| new_db.py | new_db.ini | hartlandclark.ped | Loading a pedigree into SQLite and creating a report of mean inbreeeding by birth year |
| new_doug.py | new_doug.ini | doug.ped | Reading a pedigree in which names are strings, drawing pedigrees |
| new_format.py | new_format.ini | boichard2a.ped | Reading a pedigree using the 'skip column' format code (Z), printing pedigree metadata |
| new_graphics.py | new_graphics.ini | boichard2.ped | Use of a number of routines from `pyp_graphics` |
| new_hartl.py | new_hartl.ini | hartlandclark.ped | Demonstrates use of `pyp_graphics.draw_pedigree()` |
| new_ids.py | new_ids.ini | new_ids2.ped | Demonstrates reading tab-delimited files, using strings as animal IDs, overriding the default missing parent code, printing animal records |
| new_inbreeding.py | new_inbreeding.ini | new_renumbering.ped | Calculating coefficients of inbreeding |
| new_inbreeding2.py | new_inbreeding2.ini, new_inbreeding2multiple.ini | new_renumbering.ped, horse.ped | Advanced .ini file techniques, computations on extremely inbred animals, calculation of summary statistics for coefficients of relationship |

*continued on next page*

| Program Name | Configuration File | Pedigree File | Description |
|---|---|---|---|
| new_jbc.py | new_jbc.ini | new_ids2.ped | Using `pyp_jbc.color_pedigree()` to produce a weighted, colored pedigree |
| new_lacy.py | new_lacy.ini, new_format.ini | new_lacy.ped, boichard2a.ped | Calculating effective ancestor and founder numbers |
| new_methods.py | new_format.ini | boichard2a.ped | Use of `pyp_metrics.related_animals()` and `pyp_metrics.common_ancestors()` |
| new_networkx.py | new_networkx.ini | generations.ped | Use of [algebraic] graph functions |
| new_options.py | new_options.ini | new_lacy.ped | Use of the configuration files |
| new_renumbering.py | new_renumbering.ini | new_renumbering.ped | Renumbering a pedigree, calculating inbreeding, pedigree drawing |
| new_reporting.py | new_reporting.ini | new_renumbering.ped | Use of reporting functions |
| new_simulate.py | new_simulate.ini | None | Demonstrates how to create a random pedigree and produce a drawing of that pedigree. |

APPENDIX

**B**

# GEDCOM File Handling

PyPedal is capable of importing from, and exporting to, GEDCOM 5.5 files using a subset of data record and tag types from the standard (Table **??**). Most of the information that can be exchanged in GEDCOM files has no direct use in PyPedal, so important information from PyPedal's point-of-view is not lost. However, it's important to note that **PyPedal's GEDCOM import and export is lossy**! This means that information in a GEDCOM file is lost when importing the file, and data from PyPedal pedigrees is lost when exporting. There are many free and commercial packages for doing human genealogy that take full advantage of GEDCOM, so you might want to look at one if you need more advanced GEDCOM support than PyPedal provides.

Table B.1: GEDCOM 5.5 data records and tags imported by PyPedal.

| Data Record Type | Supported Tags | Description[3] |
|---|---|---|
| Fam_Record | FAM | Alphanumeric with underscores; formed from parent IDs |
| | HUSB | Sire, if known |
| | WIFE | Dam, if known |
| | CHIL | Pointer to Individual_Record (one record per child) |
| Individual_Record | INDI | Individual ID |
| | SEX | M, F, or U (unknown) |
| | NAME | Individual's name, if known |
| | BIRT | Indicates that a birth date or year follows |
| | DATE | Birth date or birth year, if known |
| | FAMC | Pointer to family to which this individual belongs |
| | FAMS | Pointer to family in which this individual is a parent |

The list of recognized tags is hard-coded in a list named *known_tags* in `pyp_io.load_from_gedcom()`.

Table B.2: GEDCOM 5.5 data records and tags exported by PyPedal.

| Data Record Type | Supported Tags | Description[6] |
|---|---|---|
| Header | HEAD | — |
| | SOUR | PYPEDAL |

*continued on next page*

| Data Record Type | Supported Tags | Description[7] |
|---|---|---|
| | VERS | V2.0 |
| | CORP | USDA-ARS-BA-ANRI-AIPL |
| | DEST | PYPEDAL |
| | DATE | Timestamp from time of file creation |
| | FILE | Filename provided by user |
| | GEDC | — |
| | VERS | VERS 5.5 |
| | FORM | Lineage-Linked |
| | CHAR | ASCII |
| Fam_Record | FAM | Alphanumeric with underscores; formed from parent IDs |
| | HUSB | Sire, if known |
| | WIFE | Dam, if known |
| | CHIL | Pointer to Individual_Record (one record per child) |
| Individual_Record | INDI | Individual ID |
| | SEX | M, F, or U (unknown) |
| | NAME | Individual's name, if known |
| | BIRT | Indicates that a birth date or year follows |
| | DATE | Birth date or birth year, if known |
| | FAMC | Pointer to family to which this individual belongs |
| | FAMS | Pointer to family in which this individual is a parent |

Some tags have slightly different connotatios in PyPedal than in GEDCOM. For example, in human genealogy marriages are important events, but that is not the case in animal pedigrees. PyPedal creates family records only for unique mating pairs, and marriage information is lost when importing a GEDCOM file. Similarly, no marriage information is exported, and you will not see any family records containing only HUSB and WIFE tags. Founders (animals with both parents unknown) will have individual records but no family records. The default birth year used by PyPedal is 1900; if you do not override that value then individuals with birth years of 1900 will not have BIRT/DATE tags written to their individual record. The same is true of default birth dates (01011900).

Importation is done by reading the GEDCOM file, parsing out the supported tags into "family" and "individual", and using Python dictionaries (hash tables) to map everything down to individual records. Those individual records are then written to a file, the pedigree format string and pedfile variables are updated for the new file. That file is then loaded automatically. The downside is that you end up with two copies of each pedigree file, but disc space is cheap. I won't add an option for automatic deletion of the original GEDCOM file becuase of the lossiness of the import procedure.

The export process is uncoupled from the import process. You can export any pedigree that PyPedal can read as a GEDCOM file regardless of the original source. Perhaps some human types will be interested in some of the calculations that PyPedal can do, or perhaps a dog breeder will do something unexpected, such as exporting to GEDCOM so that they can use GRAMPS or something like that to manipulate their data. Who knows. Anyway, PyPedal supports two-way data flow.

# BIBLIOGRAPHY


Ballou, J. D. (1997). Ancestral inbreeding only minimally affects inbreeding depression in mammalian populations. *Journal of Heredity 88*, 169–178.

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. (2007). PyPedal: a computer program for pedigree analysis. *Computers and Electronics in Agriculture 57*, 107–113.

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.

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

Gulisija, D. and J. F. Crow (2007). Inferring purging from pedigree data. *Evolution 61*, 1043–1051.

Gulisija, D., D. Gianola, K. A. Weigel, and M. A. Toro (2006). Between-founder heterogeneity in inbreeding depression for production in Jersey cows. *Livestock Science 104*, 244–253.

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.

Lacy, R. C., G. Alaks, and A. Walsh (1996). Hierarchical analysis of inbreeding depression in *Peromyscus polionotus*. *Evolution 50*, 2187–2200.

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

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.

Suwanlee, S., R. Baumung, J. Sölkner, and I. Curik (2007). Evaluation of ancestral inbreeding coefficients: Ballou's formula versus gene dropping. *Conservation Genetics 8*, 489–495.

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*.

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