## Contents:

1 Installing PyORBIT 3
   1.1 Requirements ................................................................. 3
   1.2 Instructions .................................................................. 4
   1.3 Additional codes .............................................................. 5
   1.4 Cythonizing your code ...................................................... 7
   1.5 Troubleshooting .............................................................. 7

2 Prepare the dataset 13
   2.1 Generic dataset .............................................................. 13
   2.2 Central transit times ......................................................... 14
   2.3 Jitter, offset and linear flags ............................................. 14

3 Prepare a configuration file 17
   3.1 File structure ................................................................. 17
   3.2 Adding a dataset ............................................................. 17
   3.3 Include common parameters ........................................... 18
   3.4 Include the models .......................................................... 20
   3.5 Additional parameters .................................................... 21
   3.6 Sampler parameters ....................................................... 21

4 API 23
   4.1 Abstract Common Model ............................................... 23
   4.2 Abstract Model .............................................................. 23
   4.3 Planets Common Model ................................................... 24

5 Indices and tables 27

Python Module Index 29

Index 31
PyORBIT is a robust, versatile framework for the characterization of planetary systems. With PyORBIT you can model light curves, radial velocities, activity indexes and transit time variations. In addition to the exoplanet signature, you can add to your model instrumental systematics and stellar activity, including Gaussian processes regression, among other things. Different parametrization can be used for orbital parameters, and orbits can be computed using dynamical integration or just non-interacting Keplers. For every parameter in the model, it is possible to set a prior, explore it in linear or logarithmic space, or keep it fixed if necessary. Thanks to abstraction, it is virtually possible to implement any physical model you can think of.

Setting-up a configuration file and running the analysis is super easy, barely an inconvenience, even if you have never wrote a line of Python (or any language at all) in your life, thanks to the versatility of YAML. Alternatively, for easy automatization, PyORBIT can be called as a Python function by passing a dictionary instead of a configuration file.

PyORBIT started in 2015 as an exercise to learn Python (2) and because at the time the only publicly available code for the Bayesian analysis of RVs was written in IDL, for which I didn’t have a license. Since then I’ve been added new options every time I needed so, and I kept updating the code while improving my Python skills.

PyORBIT has been now converted and tested for Python 3, but it should still be back-compatible with Python 2 (at your own risk).

If you are wondering what PyORBIT stands for: I am bad at acronym creation so I decided to write it with capitoul ORBIT just because I liked how it looked. Feel free to submit your retrofitting acronym!
CHAPTER 1

Installing PyORBIT

Before proceeding with the installation, I suggest to create an environment dedicated to PyORBIT. With conda/anaconda:

```
conda create --name pyorbit python=3
```

To list the available environment do:

```
conda env list
```

The active environment will be marked with a *

To activate the pyorbit environment:

- WINDOWS: `activate pyorbit`
- LINUX, macOS: `source activate pyorbit`

To download PyORBIT from the repository:

```
git clone https://github.com/LucaMalavolta/PyORBIT.git
```

Right now the code is not installable, so `python setup.py install` will not work. Software requirements can be satisfied after downloading the code from the repository.

1.1 Requirements

This is the list of packages required by PyORBIT to work out-of-the-box:

- `numpy, scipy, matplotlib`: pretty standard
- `argparse`: required to pass terminal keywords
- `pyyaml`: YAML is the language used for the configuration file
- `corner`: to make corner plots (corner.py home page)
Basic analysis can be performed using the `scipy.optimize` package, however to fully unwind the power of PyORBIT I strongly recommend these two packages: - `pyDE`: global optimization package ([PyDE home page]) - `emcee`: ensemble sampling toolkit for affine-invariant MCMC ([emcee home page])

Simply speaking, PyDE searches for the best global solution and passes it to `emcee`, ensuring that the MCMC will not be stuck around a local minimum of the chi-square. The `PyDE + emcee` combination is the easiest to install and set up, but it is possible to specify the starting point of `emcee` instead of using the outcome of `PyDE`. It is possible to use other samplers as well, such as:

- `MultiNest` ([MultiNest home page](https://github.com/MultiNest/MultiNest) and [PyMultiNest home page](https://github.com/MultiNest/PyMultiNest))
- `PolyChordLite`, previously known as just `PolyChord` ([PolyChordLite home page](https://github.com/MultiNest/PyMultiNest))
- `dynesty` ([dynesty home page](https://github.com/MultiNest/dynesty))

Additional packages may be required to perform certain types of analysis:

- `batman`: Bad-Ass Transit Model cAlculatioN ([BATMAN home page](https://github.com/BATMAN-project/BATMAN))
- `george`: Fast and flexible Gaussian Process regression in Python ([george home page](https://github.com/dfm/george))
- `celerite`: scalable 1D Gaussian Processes ([celerite home page](https://github.com/dfm/celerite))
- `TRADES`: dynamical simulation of exoplanetary systems ([TRADES home page](https://github.com/dfm/TRADES))
- `TTVfast`: transit times for and radial velocities for n-planet systems ([TTVfast home page](https://github.com/dfm/TTVfast))
- `cython`: C extension for Python ([Cython home page](https://cython.org/))
- `getdist`: For the analysis of MCMC chains ([getdist home page](https://github.com/dfm/getdist))

If you are using any of those packages listed above, please be sure to cite the proper references, as stated in their web page.

### 1.2 Instructions

#### 1.2.1 pyDE

Installing from pip will results in an error, so you have to install the most up-to-date version from source using the following commands:

```bash
git clone https://github.com/hpparvi/PyDE.git
cd PyDE
python setup.py install
```

From the `pyDE` source repository.

#### 1.2.2 emcee

I’m currently using the latest version of `emcee` (Version 3.0 at the moment of writing), which can be installed with pip:

```bash
pip install emcee
```

If you want to install from the source repository, there is a bug concerning the version numbering of `emcee` that must be fixed before installation:
git clone https://github.com/dfm/emcee.git

cd emcee

From here got to the directory ./src/emcee/ and open the file __init__.py with your favourite text editor, then edit line 22 from this:

to this:

```python
#from .emcee_version import __version__ # isort:skip
__version__ = '3.0.0'
```

Save the file and got back to the main emcee directory (i.e., by doing cd ../../) and then execute the usual script to install the file:


In principle PyORBIT_GetResults should be able to recognize if the output files have been produced by version 2.x or 3.x. To save you some trouble, however, I suggest you to check that you have actually installed version 3.x:

```python
import emcee
emcee.__version__
```

### 1.3 Additional codes

When running PyORBIT you may get one of the following warnings:

```
WARNING! Imported dummy celerite, models relying on this package will not work
WARNING: Could not preload libmpi.so. If you are running with MPI, this may cause segfaults
WARNING! Imported dummy PyPolyChord, models relying on this package will not work
WARNING! Imported dummy TRADES, models relying on this package will not work
WARNING! Imported dummy TTVFAST, models relying on this package will not work
WARNING! Imported dummy george, models relying on this package will not work
WARNING! Imported pyorbit.classes.dummy batman, models relying on this package will not work
```

*Simple* RV fit and analysis will still work, but if you want to use one of these packages and you are getting one of these errors, the code will fail miserably. You will still have some of these warnings because PyORBIT will try to load the relative module even if you are not actually using it. So be worried only if you want to do some of the things listed here Requirements and the appropriate package is not installed (the code will crash anyway).

The following codes may be required to do some specific kind of analysis.

#### 1.3.1 george

Please refer to the [george installation page](http://emcee.readthedocs.io/en/stable/user/install.html#from-source) for detailed instruction on your preferred method of installation. At the time of writing this guide, using conda installed version 0.3.1 of the package.

```bash
conda install -c conda-forge george
```

Please check that your installed version is equal or higher than 0.3:

```python
import george
george.__version__
```
1.3.2 celerite

On some systems, importing george and celerite during the same Python session may cause a segmentation fault. The only workaround I found is to install celerite using conda-forge instead of pip. If you are not planning to use celerite, you may proceed with the standard installation through conda-forge:

```
conda install -c conda-forge celerite
```

If you plan to use celerite, you may be interested in compiling from source in order to enable improved automatic differentiation. Please refer to the celerite installation page to check the best option for your installation.

1.3.3 MultiNest

This Nested Sampling algorithm is available at MultiNest home page. To work with PyORBIT, the Python interface `'_PyMultiNest'`_ is required.

First of all, make sure cmake is installed on your computer, then download the MultiNest repository and install compile the code following this procedure:

```
git clone https://github.com/farhanferoz/MultiNest.git
cd MultiNest/MultiNest_v3.11_CMake/multinest/
cmake .
make
make install
```

You’ll get an error complaining the lack of administrative privileges to create a directory, you can ignore it the installation procedure has gone beyond -- Installing: /usr/local/lib/libmultinest.a `'' (or something similar) without problems. Alternatively, you can skip the last passage and add the path of the multinest library to your `.bashrc` code in this way:

```
export LD_LIBRARY_PATH=/path/to/MultiNest_v3.11_CMake/multinest/lib/:$LD_LIBRARY_PATH
```

After installing MultiNest, you can proceed with PyMultiNest

```
git clone https://github.com/JohannesBuchner/PyMultiNest.git
cd PyMultiNest
python setup.py install
```

Please check PyMultiNest documentation for any doubt

1.3.4 PolyChordLite

This Nested Sampling algorithm is available at PolyChordLite home page. pypolychord, the Python interface of PolyChord, has been revamped starting from version 1.12 and then renamed after its transformation to PolyChordLite. Earlier versions will likely not work with PyORBIT.

```
git clone https://github.com/PolyChord/PolyChordLite.git
cd PolyChordLite/
```

Change the Makefile appropriately if you are using weird C/Fortran compilers or Linux distributions. With anaconda on Ubuntu 16.04 LTS and Ubuntu 18.04 LTS I didn’t have to change any setting. In the past, MPI was disabled by default when installing on macOS. I didn’t manage to make MPI and PolyChord work together on my laptop so decided to leave it that way. Right now I’m not sure what is the situation.
When you have finished modifying the Makefile, to build the code run

```
make pypolychord
python setup.py install --user
```

The next step is to configure your `LD_LIBRARY_PATH` to point to your PolyChord installation, and your `LD_PRELOAD` to point to your mpi installation. PolyChord will tell you the exact line to be added to your `~\bashrc` file by executing:

```
python run_pypolychord.py
```

Remember to load the modified `~\bashrc` file by running `source ~\bashrc` in a terminal.

Finally, to use the MPI functionalities, prepend the MPI command before the python one, specifying the number of processor you want to use after `--np` (20 in the example).

```
mpirun --np 20 python run_pypolychord.py
```

If you already ran the command without the MPI instruction or with a different number of CPU, remember to delete the `chains` directory or the execution will fail.

**NOTE:** I have encountered too many problems when trying to use PolyChord with MPI on a Mac, so I decided to remove the help pages and use MPI only on Linux machines.

### 1.4 Cythonizing your code

You can improve the performance of the code by compiling it with Cython and distutils. To compile the code, just execute

```
./compile.sh
```

in the main directory of the source code of PyORBIT. Note that you have to run the command every time you change a file in the code, otherwise the compiled version will stay behind.

```
./compile.sh
```

To clean the repository from the compiled version, i.e. if frequent changes are made to the code and you want to avoid recompiling each time, simply run:

```
./clean_compile.sh
```

Note that in order to allow cythonization, the `.py` files in the `pyorbit/classes` and `pyorbit/models` directory are actually symbolic links to the `.pyx` files in the same directory.

More information on Cython and distutils can be found at their respective web pages.

### 1.5 Troubleshooting

I run my code on a Linux Box, but if I need to do a quick test or debug and I’m not in the office I do it on my Mac. Unfortunately some things are not as straightforward as they should be. Below you can find a collection of errors I found along the way and how I fix them.

In the following, I assume you have installed the Command Line Tools with the command `xcode-select --install` and the package manager for macOS `brew`. If you are using macOS 10.14 (Mojave), follow this additional instructions here: Fixing missing headers for homebrew in Mac OS X Mojave (from The caffeinated engineer).
Note that I had to download again the Command Line Tools from the Apple Developer website in order to have the macOS SDK headers appearing in the correct folder.

I’m not a IT expert, use these advices at your own risk!

1.5.1 MPI (all systems) - Crash after a few iterations

If you have an error similar to this one:

```
Primary job terminated normally, but 1 process returned
a non-zero exit code. Per user-direction, the job has been aborted.

mpirun noticed that process rank 0 with PID 0 on node ghoul exited on signal 11
 -> (Segmentation fault).
```

You are experiencing a problem already reported in the README file of the PolyChord source:

Try increasing the stack size: Linux: `ulimit -s unlimited` OSX: `ulimit -s hard` and resume your job. The slice sampling & clustering steps use a recursive procedure. The default memory allocated to recursive procedures is embarrassingly small (to guard against memory leaks).

1.5.2 MPI (all systems) - No available slots

The solution to this error:

```
mpirun -np 8 python run_PyPolyChord.py

There are not enough slots available in the system to satisfy the 8 slots
that were requested by the application:
/usr/bin/python

Either request fewer slots for your application, or make more slots available
for use.
```

Is quite simple: use a lower number after `-np`. If `HyperThreading` is activated, the number of cores you see in your favorite task manager (or just `htop`) is the number of _logical_ processor, while MPI cannot go further than the real number of cores in your machine.

1.5.3 PolyChord (Mac) - check gcc/gfortran/g++ versions

I have `gfortran` installed through brew on my macOS 10.14, but when I run `make pypolychord` it keeps asking for gfortran-8 when installing PolyChord 1.16. The offending lines are from 11 to 13 of the Makefile_gnu file, in the main directory:

```
FC = gfortran-8
CC = gcc-8
CXX = g++-8
```

To fix this, first check the version of your fortran compiler with `gfortran -v`:
From the last line I can see that my `gfortran` is part of version 9 of the `gcc` compiler provided by `brew`. However, a version check of `gcc` gives a different answer:

```
$ gcc -v
```

```
Configured with: ...
Apple clang version 11.0.0 (clang-1100.0.20.17)
Target: x86_64-apple-darwin18.6.0
Thread model: posix
InstalledDir: /Library/Developer/CommandLineTools/usr/bin
```

In other words, the command `gcc` will call the version provided by Apple, while `gfortran` comes with the `brew` version of `gcc` (and apparently it’s not provided by Apple at all). To avoid conflicts with libraries, be sure to use to identify the correct commands to call `gcc`, `gfortran` and `g++` from the same installation. Most of the time, you just have to append the version number at the end, i.e. `gcc-9`, `gfortran-9`, and `g++-9`.

Finally, modify the `Makefile_gnu` accordingly:

```
FC = gfortran-9
CC = gcc-9
CXX = g++-9
```

Run `make pypolychord`, ignore the warnings, and then execute the command suggested at the end (if compilation was successful), in my case `CC=gcc-9 CXX=g++-9 python setup.py install --user`

### 1.5.4 Magically fixed problems

Here I list some problems that I encountered in the past while installing some code, but that didn’t appear anymore when a tried a new installation on more recent computers.

**ldd: command not found**

This error seems to be fixed in `PolyChord` v1.14, but I’ll leave it here for reference.

```
/bin/sh: ldd: command not found
```

Open the `Makefile` in the main directory and substitute `ldd` with `otool -L`. In version 1.12 this is the only line you have to change, from this:

```
$(shell touch PyPolyChord/.ld_preload.sh; ldd $(LIB_DIR)/libchord.so | grep -o '/. 
˓→*libmpi.so[^/]* ' | awk '{print "export LD_PRELOAD="$$1:"$$LD_PRELOAD"}'} >
˓→PyPolyChord/.ld_preload.sh)
```

to this:

```
$(shell touch PyPolyChord/.ld_preload.sh; otool -L $(LIB_DIR)/libchord.so | grep -o '/ 
˓→*libmpi.so[^/]* ' | awk '{print "export LD_PRELOAD="$$1:"$$LD_PRELOAD"}'} >
˓→PyPolyChord/.ld_preload.sh)
```
Executing `make clean` will not delete the library files created in the `lib` folder, so you have to delete them manually:

```
make clean
rm lib/polychord*.*
make
```

### 1.5.5 Magically fixed MPI problems

Here I report errors I encountered so far when I try to install or run PolyChord in MPI mode. I had all these problems using PolyChord 1.12 on Ubuntu 16.04 LTS. Installing and running PolyChord 1.14 on Ubuntu 18.04 LTS didn’t result in any of these errors. MAGIC! For other errors, please refer to the README that comes with the source code.

**Broken MPI**

If you get the following errors when executing `run_pyploychord.py`, your MPI/OpenMPI installation is likely broken and you have to re-install it. You need to have a working MPI installation even when you are using PolyChord in single-CPU mode!

```
[[INVALID],INVALID] ORTE_ERROR_LOG: A system-required executable either could not be found or was not executable by this user in file ess_singleton_module.c at line 231
[[INVALID],INVALID] ORTE_ERROR_LOG: A system-required executable either could not be found or was not executable by this user in file ess_singleton_module.c at line 140
[[INVALID],INVALID] ORTE_ERROR_LOG: A system-required executable either could not be found or was not executable by this user in file runtime/orte_init.c at line 128
```

In my case, I decided to re-build OpenMPI by following these instructions. Be sure to modify the `LD_PRELOAD` in your `~/.bashrc` accordingly. If you are not able to fix the problem, you can still run PolyChord without using the MPI/OpenMPI support (but be ready to wait a lot of time when executing a program...). Open the Makefile file end switch the MPI flag to zero:

```
# Whether to use MPI
MPI=1
```

then run:

```
make veryclean
make
```

**MPI non starting**

If you get the following error when executing `mpirun -np 20 python run_PyPolyChord.py`:

```
---------------------------------------------------------------------------
It seems that there is no lamd running on the host.
This indicates that the LAM/MPI runtime environment is not operating.
The LAM/MPI runtime environment is necessary for the "mpirun" command.
Please run the "lamboot" command the start the LAM/MPI runtime environment. See the LAM/MPI documentation for how to invoke "lamboot" across multiple machines.
---------------------------------------------------------------------------
```

Chapter 1. Installing PyORBIT
Then check if the mpirun executable belongs to the same installation of the library that have been used to compile PolyChord. For example, in my case I re-installed OpenMPI in the directory /home/malavolta/CODE/others/openmpi_dir. This is how `LD_PRELOAD` is configured in my ~/.bashrc file:

```
export LD_PRELOAD=/home/malavolta/CODE/others/openmpi_dir/lib/libmpi.so:$LD_PRELOAD
export LD_LIBRARY_PATH=/home/malavolta/CODE/others/PolyChord/lib:$LD_LIBRARY_PATH
```

I have to add the path of the binaries of my OpenMPI installation. The correct mpirun is:

```
$ which mpirun
/home/malavolta/CODE/others/openmpi_dir/bin/mpirun
```

If your mpirun is not coming from the same installation directory of your MPI libraries, add to the PATH environment variable the bin directory of the MPI distribution you are currently using, at the end of your ~/.bashrc file:

```
export PATH=/home/malavolta/CODE/others/openmpi_dir/bin:$PATH
```
Prepare the dataset

Sometimes your instrument may be affected by some weird systematics (e.g., offsets between measurements taken at different epochs, trends with time), these can be easily encoded in the dataset file without burdening the configuration file with extra parameters, or even worse by splitting the data into several files.

2.1 Generic dataset

Generic input data file must have this structure:

- 1st column: epoch of the observation
- 2nd column: independent measurement
- 3rd column: error associated to the measurement
- 4th column: flag to activate the jitter parameter(s)
- 5th column: flag to activate the offset parameter(s)
- 6th column: flag to activate the linear trend parameter(s)

In the case of central time of transits, the first column identifies the number of the transit (to keep into account missing T0s), skip to Central transit times for more information.

The flags of the last three columns must be expressed as integers. Jitter, offset, and linear parameters cannot be shared between datasets. For linear trends of physical origin, a model that can be shared between datasets is available. Check the Jitter, offset and linear flags subsection for more information.

This is an extract from the file TestCase01_RV.dat, in the example folder, with epochs expressed as BJD_{TDB}-2450000.0 Julian days:

<table>
<thead>
<tr>
<th>Epoch</th>
<th>RV</th>
<th>Flag 1</th>
<th>Flag 2</th>
<th>Flag 3</th>
<th>Flag 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>6000.010493</td>
<td>4764.73</td>
<td>1.00</td>
<td>0</td>
<td>0</td>
<td>-1</td>
</tr>
<tr>
<td>6000.719975</td>
<td>4766.58</td>
<td>1.00</td>
<td>0</td>
<td>0</td>
<td>-1</td>
</tr>
<tr>
<td>6001.967132</td>
<td>4779.52</td>
<td>1.00</td>
<td>0</td>
<td>0</td>
<td>-1</td>
</tr>
<tr>
<td>..........</td>
<td>..........</td>
<td>......</td>
<td>......</td>
<td>......</td>
<td>......</td>
</tr>
</tbody>
</table>
The choice of the time standard is arbitrary, just pay attention to always be self-consistent. When provided to the `pyorbit_emcee()` subroutine instead of being read from a file, the input_dataset must be a dictionary where each keyword corresponds to the label of the dataset. For each keyword, a `[n,6]` numpy array must be provided, where `n` is the number of observations. If the keyword for a dataset is present, it will have priority on the dataset file unless the keyword is empty.

For example:

```python
input_dataset['RV'] = np.zeros([n,6])
input_dataset['RV'][:,0] = epochs of the observations
input_dataset['RV'][:,1] = values of the observations
input_dataset['RV'][:,2] = associated errors
input_dataset['RV'][:,3] = jitter flag
input_dataset['RV'][:,4] = offset flag
input_dataset['RV'][:,5] = linear trend flag
```

### 2.2 Central transit times

For central time of transit (`Tcent`) file data, the structure is slightly different, since the first column denote the number of the transit. This number will help in identifying missing transits (but honestly I don’t remember right now what happens if you start from a random number...)

```
0 4959.70736 0.00145 0 -1 -1
1 4968.99347 0.00225 0 -1 -1
2 4978.28014 0.00202 0 -1 -1
. . . . . . . . . . . . . .
```

**IMPORTANT** always set the 5th (offset) and 6th column to -1 (or don’t include them at all) to avoid unphysical solution (drift and jumps in time are not allowed)

```python
input_dataset['Tcent_b'] = np.zeros([n,6])
input_dataset['Tcent_b'][:,0] = number of transit (e.g. if some transit is missing)
input_dataset['Tcent_b'][:,1] = transit time
input_dataset['Tcent_b'][:,2] = associated error
input_dataset['Tcent_b'][:,3] = jitter flag
input_dataset['Tcent_b'][:,4] = should be set to -1 to avoid unphysical solution
input_dataset['Tcent_b'][:,5] = should be set to -1 to avoid unphysical solution
```

### 2.3 Jitter, offset and linear flags

Jitter, offset and linear parameters cannot be shared between datasets. For linear trends of physical origin, a model that can be shared between datasets is available.

Activating the jitter flag will introduce a new parameter which will be added in quadrature to the error bars of the measurement for which the flag has been activated. The offset flag will add a constant offset (or zero-point) to all the measurement for which the flag has been activated. The linear flag will include a linear trend in the dataset. Note that this flag will add only the slope as additional parameter, while the intercept of the linear trend must be set using the offset flat. Only a linear trend is supported, higher order terms have not been implemented simply because I never encountered such an extreme case, but on request it may be included as additional columns in the file.

The flags of the last three columns must be expressed as integers. The value of a flag must be set -1 if you don’t want to include the corresponding parameter in the model, otherwise to increasing number starting from 0. The flags can be used to divide the dataset in groups where different parameters are used for a specific model. For example, it is
possible to use different offset parameters for data taken before and after a given epoch (for example, if the instrument has been modified in some way). To do so, set to 0 the offset flag of the data taken before the chosen epoch, and to 1 the data taken after that epoch. Just increase by another unit if you want to add an additional offset parameter.

The code will assume that the number of parameters is equal to the maximum value of the flag plus one, so pay attention in increasing the flag sequentially and without jumps.

For a given kind of flag:

- Flags must be given in consecutive order starting from zero (Python notation).
- Inactive flag must be set to -1.
- All the parameters that share the same flag value will have that parameter in common.
- Different parameters will be used for measurements with different value of flag.
- Flags in different columns are independent.

Let's look at the following example:

<table>
<thead>
<tr>
<th>epoch</th>
<th>meas</th>
<th>err</th>
<th>offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>-1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>-1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>-1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>-1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>-1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>-1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>-1</td>
</tr>
</tbody>
</table>

- `epoch_00` and `epoch_01` share the same jitter term, so they do (epoch_02, epoch_03, epoch_04) and (epoch_05, epoch_06), for a total of 3 jitter parameters.
- `epoch_00, epoch_01, epoch_02` and `epoch_06` share the same offset. `epoch_03, epoch_04, epoch_05` share a different offset parameter.
- `epoch_04, epoch_05, epoch_06` are modeled using a linear trend. `epoch_00, epoch_01, epoch_02` and `epoch_03` are not.

What’s the point of using the flags instead of creating different datasets? Here a few examples:

- Suppose your instrument undergoes some slight modifications, and the zero point of the RV is shifted but the overall instrument is the same: you can account for this zero-point difference while sharing the same jitter parameter.
- Again your instrument undergoes major changes and both the zero-point and jitter are affected. However, observational parameters that depend on the characteristics of the instrument will be the same (e.g. the amplitude of stellar activity signals observed at optical wavelength), so you want to use only one parameter for this dataset and a different one for another dataset (e.g. observations gathered in the infrared).

Shortly, the flags represent a way to separate instrumental issues from the physical problems.
CHAPTER 3

Prepare a configuration file

The configuration file is the most important part of your analysis, because it will contain the specifics of your model, the priors you want to assign to the parameters, and the way you want to analyze your data. Thanks to the readability of the YAML language, it will also serve as a useful memento at a glance for the set-up of your model.

3.1 File structure

The file is divided in several sections, each one with a clear purpose:

- **inputs:** here you list the files containing the datasets
- **common:** the physical parameters of your planetary system that are going to be shared between the models (and are independent from them): orbital parameters of the planets, mass and radius of the star, rotational period and decay timescale of the active regions…
- **models:** this is where you specify how the physical elements of the system in the common section are going to influence your data: which planets include in the radial velocity computation, what kind of model/kernel you want to use to model the activity…
- **parameters:** values that are not system parameters per se, but are required for the correctness of the analysis
- **solver:** the parameters required by PyDE, emcee, or MultiNest are all listed here

To have a glance at how a configuration file looks like, check the .. _documentation_example.yaml: http://cnn.com/:

3.2 Adding a dataset

Datasets are grouped under the input section:

```yaml
inputs:
  RVdata:
    file: K2-141_dataset/K2-141_RV_HARPN.dat
    kind: RV
```

(continues on next page)
models:  
  - rv_model  
  - gp_regression

- **RVdata**: the label to identify the dataset. The same label must be used later in the file if we need to specify a property of the dataset.
- **file**: the file including the dataset.
- **kind**: the kind of dataset provided. This label is used only by specific models that requires a special treatment with respect to standard datasets, i.e. central times of transits must be tagged with the Tcent data type.
- **models**: list of models’ labels to be used to analyze the dataset. If only one model is specified, it can be written on the same line without the – sign.

### 3.2.1 Dataset kinds

The following datasets are recognized by the code. In the majority of the cases, the way the dataset is treated depends on the specified models. A list of aliases is defined to circumvent the most common typos (and to avoid checking the documentation every time...).

- **RV**: radial velocities. Aliases: RV, RVs, rv, rvs.
- **Phot**: photometry. Aliases: P, Ph, p, PHOT, Phot, phot, Photometry, photometry.
- **Tcent**: central times of transits. Aliases: Tcent, TCent, Tc, TC, T0, TT.
- **BIS**: Bisector Inverse Span of the CCF. Aliases: BIS, bis.
- **S_index**: (uncalibrated) measurements of the emission in the core of the CA H&K lines. Aliases: S, S_index.
- **Ca_HK**: as for the S index, but with the photometric contribution removed. Aliases: Ca_HK, logR.

### 3.3 Include common parameters

The physical objects of your system must be included in the common section. The name of the section derives from the fact each specific model required to model a given dataset will rely on physical parameters of planets and star (and more...) that are shared with all the other models eventually employed. For example, a simultaneous fit of a radial velocity curve and a transit light curve will be characterized by a single period for the planet of interest.

Let’s have a look at this example where we have included a transiting planet in a circular orbit (b), a non-transiting planet (c), some properties for the stellar activity, and of of course the stellar parameters.

```plaintext
common:
  planets:
    b:
      orbit: circular
      parametrization: Eastman2013_Tcent
      boundaries:
        P: [0.29, 0.31]
    c:
      ... other properties ...
```
Quite a lot to process, right? Let's start with the main sections. planets and star are kind special because the section names are also the reference name of the objects, i.e., these names are hard coded and if you try to put planet or star parameters in sections with different names you will break everything. The reason is that planets and star are actually containers for the true objects, which are b, c (see the relative documentation for more details). stellar_activity instead is a label for the object with reference names activity, e.g., if you want to know more you have to look for the object named activity in the documentation and for the file activity.py in the source code. Note that if you are including just a single object of a kind, you can use its reference name as label and omit the type keyword, as in actual example file in the repository.

```python
    K: [0.01, 100.0]
    Tc: [7744.06, 7744.08]

    priors:
    P: ['Gaussian', 0.2803244, 0.00000015]
    Tc: ['Gaussian', 7744.07160, 0.00022]

    spaces:
    P: Linear

    fixed:
    c: [86.3, 3.00]

    orbit: keplerian

    boundaries:
    P: [1.0, 1000.0]
    K: [0.01, 100.0]
    e: [0.00, 0.95]

    stellar_activity:

    kind: activity

    boundaries:
    Prot: [5.00, 16.00]
    Pdec: [10.0, 2000.00]
    Oamp: [0.01, 0.60]

    priors:
    Prot: ['Gaussian', 13.941613, 0.66]
    Pdec: ['Gaussian', 12.203273, 3.21]
    Oamp: ['Gaussian', 0.342221, 0.054]

    star:

    star_parameters:

    priors:
    radius: ['Gaussian', 0.681, 0.018]
    mass: ['Gaussian', 0.708, 0.028]
    rho: ['Gaussian', 2.244, 0.161]
```

Note that also b and c are labels, i.e., you can give the planets any random name, but in their case it's not necessary to specify what kind of object they are because it's implicitly assumed for being inside the planet section (Note: this is true only for the planets!). The use of labels will be clear when dealing with multi-band photometry.

3.3. Include common parameters
Let’s focus on planet b as an example of the properties of an object. There are some keywords that are specific of the planet model, such as orbit and parametrization. Check the documentation of each model to know more about the specific keywords. Then there are four sections that every model must possess:

- **boundaries**: all the Bayesian samplers implemented in PyORBIT require the definition of lower and upper boundaries. If not specified, very broad boundaries will be set by default according to the source code.
- **priors**: is it a Bayesian analysis if you don’t specify at least one prior? When not included, the prior is assumed to be Uniform over the range specified in boundaries (or the one assumed by default).
- **spaces**: the choice here is between Linear and Logarithmic. Note that the logarithmic space is in base 2. Once again, the default choice is listed in the source code.
- **fixed**: You listed here every parameter you want to keep fixed. The value must be accompanied by his errors, because in some cases it will be used in the computation of the derived parameters, e.g., the real mass of a transiting planet.

Under this section you need to list only the parameters that will be actually used by the models in models section. For example, a circular orbit does not require boundaries or priors for the eccentricity.

The default choices for each possible parameter and for each section listed above are declared in the source file of the object.

### 3.4 Include the models

In this section of the configuration file, called models, we specify the properties of the models that we want to employ to analyze the data.

```python
models:
  rv_model:
    kind: radial_velocities
    planets:
      - b
  gp_regression:
    kind: gp_quasiperiodic
    common: stellar_activity
    RVdata:
      boundaries:
        Hamp: [0.01, 100.00]
```

In this example, our complete model comprises a radial_velocities model to model the orbital motion of the star due to the presence of planets b and c, and a gp_quasiperiodic to model the stellar activity with Gaussian process regression and a quasi-periodic kernel. Note that rv_model and gp_regression are the labels assigned to the two models, and they are the string that has to be referenced in the models section under each dataset.

There are two main sections:

- **kind**: the model you want to employ, e.g., how the physical parameters are converted into theoretical predictions for the observations.
- **common**: the list of labels referring to the common objects you want to be used in the model. For RVs and TTVs the keyword planet can be used as well.

In the following sections, e.g. RVdata in this example, the properties of parameters that depend specifically on the dataset are listed. The properties are boundaries, priors, spaces, and fixed, similarly as in Include common parameters. Here for example we are specifying the boundaries of the amplitude of the covariance matrix in the GP regression when applied to the radial velocity data.
Additional keywords may be present depending on the model, see the documentation for more details.

**Note:** the `star_parameters` object is included by default whenever needed, so you don’t need to list it in the common section.

### 3.5 Additional parameters

System-wide parameters that did not find place in any other section below are included in the `parameters` section.

```yaml
parameters:
  Tref: 7800.0
```

In this example, `Tref` is the epoch of reference, one of the most neglected orbital elements ever. For non-transiting planets, the argument of periapsis and the mean anomaly will be referred to this value. When not explicitly stated, it will be computed internally as the average of all the observational epochs.

### 3.6 Sampler parameters

Each sampler comes with its set of parameters, which fine-tuning depends on both the size of the datasets and the complexity of the model, among other things. These parameters can be specified in the configuration file under the `solver` section.

```yaml
solver:
  pyde:
    ngen: 4000
    npop_mult: 8
  emcee:
    npop_mult: 8
    nsteps: 20000
    nburn: 5000
    thin: 100
    nsave: 10000
  nested_sampling:
    nlive: 1000
    num_repeats_mult: 5
    sampling_efficiency: 0.30
    shutdown_jitter: True
    recenter_bounds: True
```

This is a brief explanation of the parameters associated to each keyword, please refer to the sampler documentation for their proper usage. - **pyde:** parameters for the global optimization code PyDE.

- `ngen`: number of generations.
- `npop_mult`: the size of the parameter vector population is given by the dimensionality of the problem multiplied by this number
- `emcee`: parameters for the ensemble sampling toolkit for affine-invariant MCMC (emcee). - `npop_mult`: the number of walkers in the ensemble is given by the dimensionality of the problem multiplied by this number. If PyDE and emcee are used sequentially, this keyword must have the same value in both sections (they are named in the same way as a reminder). - `nsteps`: number of steps of each chain. - `nburn`: number of ‘burn-in’ steps. - `thin`: thinning factor, should be at least equal to the autocorrelation time (before thinning). **Note:** the chains will be saved with the thinning factor already applied - `nsave`: results are saved every (unthinned) `nsave` steps, so that it is possible to perform a preliminary analysis while the code is still running.
• `nested_sampling`: these parameters are shared between all the implemented nested sampling algorithms, which are MultiNest, PolyChordLite, and dynesty. - `nlive`: total number of live points. - `num_repeats_mult`: The number of slice slice-sampling steps to generate a new point (PolyChord only). - `sampling_efficiency`: sampling efficiency (MultiNest only) - `shutdown_jitter`: if True (default value), the jitter parameters are removed from the model (even if the flag in the dataset is active).

• `recenter_bounds`: after the first run with (global) optimization, the boundaries of circular parameters (e.g. angles) will be recenter around the most likely value, in order to avoid border effects associated with some samplers.
4.1 Abstract Common Model

This is the common model class

Some stuff here

```python
class pyorbit.models.abstract_common.AbstractCommon(common_ref)
    Comments to be updated

define_variable_properties(ndim, output_lists, variable_list)
    Bounds are defined in this class, where all the Planet-related variables are stored. Bounds and parameter
    index CANNOT be defined in the Common class: we don’t know a priori which parameters will be actually
    used in the complete model.

return_priors(theta)
    Compute the prior probability for a given set of input parameters
    return_priors is defined in the common models because, differently from other functions that can be exe-
    cuted more than once on the same variable, the prior for a given parameter should be computed and added
    to the log_chi2 only once
    Args: theta: the set of parameters created by the solver
    Returns: prior_out: prior probability, to be added to the posterior prob.
```

4.2 Abstract Model

This is the common model class

Add explanation

```python
class pyorbit.models.abstract_model.AbstractModel(model_name, common_ref)
    Comments to be updated
```
define_variable_properties (ndim, output_lists, dataset_name)

Bounds are defined in this class, where all the Planet-related variables are stored. Bounds and parameter index CANNOT be defined in the Common class: we don’t know a priori which parameters will be actually used in the complete model.

4.3 Planets Common Model

Can I haz planetz?

Some planet stuff here

class pyorbit.models.planets.CommonPlanets(*args, **kwargs)

Inherited class from AbstractCommon

For computational reason it is better to fit $\sqrt{e}\sin\omega$ and $\sqrt{e}\cos\omega$. define_special_variable_properties() and define_special_starting_point() must be redefined

Attributes:

- model_class (string) identify the kind of class
- list_pams all the possible parameters that can be assigned to a planet are listed here
- default_bounds these default boundaries are used when the user does not define them in the yaml file
- recenter_pams circular parameters that may need a recentering around the most likely value after the global optimization run
- period_average variable used only by TRADES

default_bounds = {'K': [0.5, 2000.0], 'M': [0.5, 1000.0], 'P': [0.4, 100000.0], 'R': [1e-05, 0.5], 'Tc': [0.0, ... 'lN': [0.0, 6.283185307179586], 'o': [0.0, 6.283185307179586], 'sre_coso': [-1.0, 1.0], 'sre_sino': [-1.0, 1.0]}

Must be the same parameters as in list_pams, because priors are applied only to _physical_ parameters

define_special_starting_point (starting_point, var_sampler)

Eccentricity and argument of pericenter require a special treatment

since they can be provided as fixed individual values or may need to be combined in $\sqrt{e}\sin\omega$ and $\sqrt{e}\cos\omega$ if are both free variables

Args:

- starting_point
- var_sampler

Returns:

- bool

define_special_variable_properties (ndim, output_lists, var)

Boundaries definition for eccentricity $e$ and argument of pericenter $\omega$

The internal variable to be fitted are $\sqrt{e}\sin\omega$ and $\sqrt{e}\cos\omega$. With this parametrization it is not possible to naturally put a boundary to $e$ without affecting the $\omega$. Additionally the subroutine will check if either $e$ or $\omega$ have been provided as fixed values. If true, the parametrization will consist of $e$ or $\omega$ instead of $\sqrt{e}\sin\omega$ and $\sqrt{e}\cos\omega$

Args:

- ndim number of parameters already processed by other models
var  input variable, either $e$ or $\omega$  

Returns:

ndim  updated dimensionality of the problem

bounds_list  additional boundaries to be added to the original list

model_class $=$ 'planet'

choice to parametrize the eccentricity and argument of pericenter: Standard: $e$ and $\omega$ Ford2006: $e \cos{\omega}$ and $e \sin{\omega}$ Eastman2013: $\sqrt{e} \cos{\omega}$ and $\sqrt{e} \sin{\omega}$

Following PEP 8 Style Guide for Python Code, PEP 257 Docstring Conventions and Google Python Style Guide
CHAPTER 5

Indices and tables

- genindex
- modindex
- search
Python Module Index

\textbf{p}

\begin{itemize}
\item pyorbit.models.abstract_common, 23
\item pyorbit.models.abstract_model, 23
\item pyorbit.models.planets, 24
\end{itemize}
Index

A
AbstractCommon (class in pyorbit.models.abstract_common), 23
AbstractModel (class in pyorbit.models.abstract_model), 23

C
CommonPlanets (class in pyorbit.models.planets), 24

D
default_bounds (pyorbit.models.planets.CommonPlanets attribute), 24
define_special_starting_point() (pyorbit.models.planets.CommonPlanets method), 24
define_special_variable_properties() (pyorbit.models.planets.CommonPlanets method), 24
define_variable_properties() (pyorbit.models.abstract_common.AbstractCommon method), 23
define_variable_properties() (pyorbit.models.abstract_model.AbstractModel method), 23

M
model_class (pyorbit.models.planets.CommonPlanets attribute), 25

P
pyorbit.models.abstract_common (module), 23
pyorbit.models.abstract_model (module), 23
pyorbit.models.planets (module), 24

R
return_priors() (pyorbit.models.abstract_common.AbstractCommon method), 23