Parasect

George Zogopoulos

Jan 06, 2024

CONTENTS

CHAPTER

ONE

USAGE

1.1 Menu Creation

To fully utilize the capabilities of *Parasect*, you will need to specify certain information about your desired autopilot configuration. This comes down to creating a folder structure that represents your *Meals Menu*.

Note: It is highly recommended that you read the Concepts section first and get acquainted with the *Restaurant Analogy* that is used as nomenclature.

See also:

You can find a generic example Meals Menu as well as an PX4 example Meals Menu, that are part of the unit tests of *Parasect*. Refer to it while following this guide or use them as a starting point for your own use cases.

First, create your main *Parasect* input folder, which we'll call `menu` for now (the actual folder name isn't important). You can place this folder anywhere in your system. You will point *Parasect* to it later.

By the end of this guide, its tree will look something like this:

```
menu/
├── custom_dishes/
│   ├── bread.yaml
│   ├── coffee.yaml
│   ├── dairy.yaml
│   ├── desert.yaml
│   ├── extra_quantities.yaml
│   ├── fruit.yaml
│   ├── meat_and_potatoes.yaml
│   └── salad.yaml
├── meals.yaml
└── staple_dishes/
    ├── calibration.yaml
    ├── footer.yaml
    ├── header.yaml
    └── operator.yaml
```

Make two subfolders in it: `custom_dishes` and `staple_dishes`.

1.1.1 Filling in Staple Dishes

Each file contains all necessary information to create a *Dish*. A Dish is supposed to reflect one coherent set of parameters, for example related to PID gains, peripherals configuration, Remote Controller configuration or the installed battery.

There are currently four Staple Dishes in *Parasect*: `calibration.yaml`, `operator.yaml`, `header.yaml` and `footer.yaml`. They are reserved dishes that are used by *Parasect* in either the comparison or build functions.

`calibration.yaml`: This is a Dish that contains the names of the calibration parameters of your autopilot. List those parameters here to: a) not take them into account when comparing parameter sets and b) to remove them from generated parameter sets. An example Calibration Dish follows:

```
common:
  ingredients:
    - [SALT, ~, Each has his own taste]
```

Notice how each list item is a triplet of items, representing a parameter. The first value is the parameter name. The second is the parameter value. For the calibration Dish you do not need to specify a value, you can leave it as None (~). Finally, the third item is a string, where you can document why you chose to include this parameter here. You can also set it to None (as ~).

Tip: Python regular expressions can be used to capture more than one parameter name per line, e.g.

```
common:
   ingredients:
      - [RC\d+_TRIM, ~, Disregard channel trim values]
```

`operator.yaml`: This is a Dish (with the usual Dish syntax) that contains the names of the operator parameters of your autopilot. List those parameters here to: a) not take them into account when comparing parameter sets and b) to remove them from generated parameter sets. You do not need to specify a value for a operator parameter either, leave it as None (~).

Its syntax is the same as `calibration.yaml`.

Note: `calibration.yaml` and `operator.yaml` are the only Dish files that are used by the `compare` command. If you only plan to use that and not `build` any parameter sets, then you can ignore the rest of this guide.

`header.yaml` and `footer.yaml`. These are files that don't follow the usual Dish syntax. They contain the boilerplate text that your autopilot may require of the generated parameter files. The top-level dictionary does contain a Common section, that applies to all exported formats, but then alongside is a `formats` section.

This reflects the available export *Formats* that *Parasect* offers (e.g. *px4* parameters and *px4afv1* and *px4afv2* for PX4 airframe files.)

Each named format then contains a `common` and `variants` section as usual, which you can refer to in your Meal.

An example `header.yaml` is:

```
common:
  - "# Parameter file exported by Parasect."

formats:
  csv:
```

(continues on next page)

(continued from previous page)

```
    common:
      - "# Parameter name, Parameter value"
  px4:
    common:
      - "PX4-format header line 1\n"
      - "PX4-format header line 2\n"
  px4afv1:
    common:
      - "PX4-airframe-format header line 1\n"
      - "PX4-airframe-format header line 2\n"
```

1.1.2 Creating Custom Dishes

Next, you can create the Dishes that describe your parameter sets. Within the `custom_dishes` folder, create as many Dish `.yaml` files as you want and name them as you like.

In each Dish file, start by defining your Common Ingredients as a list of triplets, just as in `calibration.yaml` and `operator.yaml`. Each triplet refers to a single parameter and its contents are:

1. The parameter name.
2. The parameter value.
3. A justification string, to remind you why you chose to fix this parameter and why selected this particular value. Can be set to None (~).

If you plan to generate only one Meal (a single parameter set), then put all your parameter definitions here. Otherwise, put in the Common Ingredients **only** those parameters who are common across all of your Meals.

Then, define your Allergen Ingredients. These are parameters that you don't want to exist in your generated parameter set. They will be removed from the default parameters list, if you point to one as a basis of your Meal. Some autopilot software group their parameters in *parameter groups* (e.g. PX4). This makes it easier to mark them as allergens as a whole, by placing the group name in the Allergen Groups section of the Dish.

If you plan to use slightly different versions of your Dish across the different Meals, then add a Variants section in your Dish. This is a dictionary where you can specialize your Dish with different Ingredients and Allergens for each Variant.

The contents of an example dish are:

```
common:
  ingredients:
    - [BEEF, 1, ~]
    - [POTATOES, 2, ~]
    - [THYME, 0.001, ~]
    - [GRAVY, 0.01, ~]
  allergens:
    substances:
      - [MUSHROOMS, ~, My stomach really can't handle them]
    groups:

variants:
  spicy:
    common:
      ingredients:
        - [CHILLI, 1, ~]
```

(continues on next page)

(continued from previous page)

```
    variants:
      extra:
        common:
          ingredients:
            - [JALLAPENOS, 2, ~]
```

Note that it is possible to nest the Variants more than one level deep.

1.1.3 Creating your Menu

Now that your Dishes and their Variants are specified, you can bring it all together by designating Meals in your Menu. Create a `meals.yaml` file in the top-level directory of your Menu folder. This is a dictionary from strings to dictionaries.

Each section represents a unique autopilot configuration and it starts with an arbitrary name.

Then, in each row you add Dishes to your Meal. The key is the Dish name and the value is the Dish Variant. Set the value to None (~) to use only the Dish Common section. Refer to the nested Variants using a slash (/).

Example Meals Menu:

```
snack:
  fruit: single
  header: ~
  footer: ~

breakfast:
  dairy: cereal
  header: ~
  footer: ~

light_meal:
  salad: ~
  meat_and_potatoes: ~
  remove_calibration: True
  remove_operator: True
  header: ~
  footer: ~

spicy_meal:
  salad: ~
  meat_and_potatoes: spicy/extra
  remove_calibration: True
  remove_operator: True
  header: ~
  footer: ~

full_meal:
  salad: ~
  meat_and_potatoes: ~
  bread: baguette
  desert: ~
  header: ~
  footer: full_meal
```

(continues on next page)

(continued from previous page)

```
christmass_at_grandmas:
  parent: full_meal
  salad: village
  extra_quantities: ~
  add_new: True

dangerous_combinations:
  parent: breakfast
  coffee: filter
  add_new: True
```

There are also some reserved, optional keywords for the Meal dictionary:

- `defaults`: Specifies a filepath where the default parameters file is found. If it is not absolute, then it is relative to your menu folder. This option overrides the *Parsect* default parameters filepath configuration.
- `sitl`: Marks the Meal for Simulation-In-The-Loop (SITL) purposes. You can select to build SITL or non-SITL Meals with the corresponding argument of the *`build`* API function.
- `hitl`: Marks the Meal for Hardware-In-The-Loop (HITL). In some output formats (e.g. PX4), this affects the output filename.
- `parent`: Adds a reference to another Meal, to be used as a parent for the current Meal. For more information see section *Parent Meals*.
- `remove_calibration`: Removes the Calibration Dish Ingredients from this Meal. **Strongly** recommended to be set to `true`.
- `remove_operator`: Removes the Operator Dish Ingredients from this Meal. **Strongly** recommended to be set to `true`.
- `add_new`: Allows the addition of Ingredients that are **not stated** in the default parameter set (or the parent, if one is specified).
- `frame_id`: This is applicable to PX4 output formats. It sets the `SYS_AUTOSTART` parameter.

1.1.4 Parent Meals

Sometimes you want to specify a base Meal (parameter set) and then a bunch more that are basically identical to that first one, except for very few changes. One such example is wanting a basic parameter set and then additional vehicles with adjustments to the values of certain parameters. Another example is wanting additional Meal variations configuring a camera mount or a secondary radio.

Instead of re-specifying an identical Meal with the additional Dishes, you can specify the base Meal as the `parent` of the new Meal and specify **only the additional** Dishes.

The Default Parameter set is not used for the child-Meals. The parameter set of the parent is used instead. Editing the parent Ingredients is thus always possible, but to add new parameters that don't exist in the parent Meal you will need to also set `add_new` to `true`.

See also:

Now that you have created your Meals Menu, find out how to point *Parasect* to it in Setting Paths.

1.2 Setting Paths

Parasect needs to know where certain files are to function:

1. To ignore *calibration* or *operator* parameters in `compare`, it needs to know where the Meals Menu is, to find the corresponding staple Dishes.
2. To `build` the Meals Menu, it needs to know its folder path.
3. To `build` a Meals Menu using the default parameters file of an autopilot, it also needs to know the path of the default parameters file.

In sum, there are two resources that the user might need to specify:

1. The Meals Menu folder path.
2. The filepath of the autopilot default parameters.

1.2.1 Pointing to the Meals Menu folder

By priority, pass the Meals Menu path in the `--input_folder` (`-i`) option of `compare` or `build`.

Alternatively, with lower priority, you can set the environment variable `PARASECT_PATH`.

1.2.2 Pointing to the default parameters file

By priority, use the `defaults` reserved keyword in a Meal description, as described *here*.

Alternatively, with lower priority, pass the default parameters filepath in the `--default_parameters` (`-d`) option of `build`.

Alternatively, with lower priority, you can set the environment variable `PARASECT_DEFAULTS`.

1.3 Usage Example with Ardupilot

This *tutorial* presents a use case of *Parasect* supporting the Ardupilot ecosystem. *Parasect* can be effective in producing a parameter file that configures an Unmanned Vehicle (UA) completely.

This tutorial mainly targets:

- UA manufacturers, wishing to manage the parameter configuration of their fleet.
- Advanced users, looking for more fine-grained control of their parameter set.

1.3.1 Initial Tuning

Let's imagine that we have built a custom multicopter of our own design and now we wish to configure the parameters of Ardupilot. To that goal, let's first obtain the default parameter set of the autopilot. We connect via our favourite GCS, flash the autopilot with the appropriate airframe and then save the default parameter set, i.e. the parameters and their values as they where on the first boot. It doesn't matter much if this is done before or after the initial sensor calibration. For the purposes of this tutorial, let's say we saved the file after calibration.

Let's create a folder `parasect_files` and place those parameters in a file inside that, named `defaults.params`. From now on our working folder will be `parasect_files`.

The file is going to look something like

```
ACRO_BAL_PITCH   1.000000
ACRO_BAL_ROLL    1.000000
ACRO_OPTIONS     0
ACRO_RP_EXPO     0.300000
ACRO_RP_RATE     360.000000
...
WPNAV_TER_MARGIN 10.000000
WP_NAVALT_MIN    0.000000
WP_YAW_BEHAVIOR  2
WVANE_ENABLE     0
ZIGZ_AUTO_ENABLE 0
```

Let's start tuning our UAV. We know that our UAV carries a 5000mAh battery (`BATT_CAPACITY=5000`) and we have calculated that we need about 1000mAh to land at our own pace (`BATT_LOW_MAH=1000`).

Then we fly a bit and tune the PID loops. We come up with the following changes:

```
LOIT_SPEED=1500
ATC_ANG_PIT_P=5.0
ATC_ANG_RLL_P=5.0
ATC_RATE_R_MAX=60
ATC_RATE_P_MAX=60
```

Finally, we choose that we purposely want the default behaviour regarding what happens mission after a reboot (`MIS_OPTIONS=0`).

Let's save the parameter file anew, naming it `my_copter_post_tuning.params`. Parasect can compare the two parameter sets and highlight the differences:

```
> parasect compare defaults.param my_copter_post_tuning.param
File comparison : defaults.param | my_copter_post_tuning.param
================================================================================
--------------------------------------------------------------------------------
Component 1-1:
--------------------------------------------------------------------------------
ATC_ANG_PIT_P   : 4.5            | 5
ATC_ANG_RLL_P   : 4.5            | 5
ATC_RATE_R_MAX  : 0              | 60
BARO1_GND_PRESS : 94502.5        | 101079.8
BARO2_GND_PRESS : 94502.5        | 101080.9
BATT_CAPACITY   : 3300           | 5000
BATT_LOW_MAH    : 0              | 1000
LOIT_SPEED      : 1250           | 1500
STAT_RUNTIME    : 0              | 123
================================================================================
9 parameters differ
```

The *compare* command of *Parasect* lists the differences between two parameter files. The tuning changes we made are in there, so we can inspect and make sure that they ended up in our saved file. But so are some that we didn't actively change. `STAT_RUNTIME` is a parameter that constantly changes value so we don't care seeing that in the differences list. Additionally, calibration values such as `BARO1_GND_PRESS` and `BARO2_GND_PRESS` are also bound to change automatically and we don't want to manually set them.

1.3.2 Our First Meal

Parasect can help us reproduce this parameter set systematically. For that, we set up a *Menu* folder with the following contents:

```
menu
├── custom_dishes
│   ├── battery.yaml
│   ├── mission.yaml
│   └── tuning.yaml
├── defaults.param
├── meals.yaml
└── staple_dishes
    ├── calibration.yaml
    └── header.yaml
```

First off, we copied our `defaults.param` file in the `menu`. The rest of the menu is made up of `.yaml` files. Let's see what each file contains. In `custom_dishes` we can name our files as we want, grouping parameters as we like. For example:

Listing 1: battery.yaml

```yaml
common:
    ingredients:
        - [BATT_CAPACITY, 5000, ~]
        - [BATT_LOW_MAH, 1000, Enough juice to RTL]
```

`battery.yaml` contains those parameters and their values related to the battery configuration. Each line is a triplet describing a) the parameter name, b) the parameter value and c) an optional reasoning of why this value was selected (reminder: the ~ symbol means `None` in `yaml` syntax).

Note: The `common` and `ingredients` keys are significant, but for now they will not be explained. You can read more in Menu Creation.

The contents of `mission.yaml` and `tuning.yaml` is similar:

Listing 2: mission.yaml

```yaml
common:
    ingredients:
        - [MIS_OPTIONS, 0, The default behaviour is what I want]
```

Listing 3: tuning.yaml

```yaml
common:
    ingredients:
        - [LOIT_SPEED, 1500, I like things a bit fast]
        - [ATC_ANG_PIT_P, 5, It can use a bit more oomph here]
        - [ATC_ANG_RLL_P, 5, It can use a bit more oomph here]
        - [ATC_RATE_R_MAX, 60, "It's a big bird, so let's take it slow"]
        - [ATC_RATE_P_MAX, 60, "It's a big bird, so let's take it slow"]
```

Let's now take a look at the `staple_dishes` folder. Its contents can only be specific `yaml` files.

In the `header.yaml` file we can put custom headers that will always be prepended in our files. In this example, the header adds two comment lines.

Listing 4: header.yaml

```yaml
formats:
    apm:
        common:
            - "# Maintainer: George Zogopoulos"
        variants:
            my_copter_1:
                common:
                - "# Parameter set for my_copter_1"
```

Finally, remember how calibration parameters would appear earlier in the parameter files comparison, making the results harder to read? `calibration.yaml` gives us a chance to fix that. Any parameter placed here will be ignored by *compare*. Additionally, a regular expression can be used here to capture more than one parameter name per line.

Listing 5: calibration.yaml

```yaml
common:
    ingredients:
        - [BARO._GND_PRESS, ~, ~]
        - [COMPASS_DIA_., ~, ~]
        - [RC\d+_MAX, ~, ~]
        - [RC\d+_MIN, ~, ~]
        - [RC\d+_TRIM, ~, ~]
        - [STAT_RUNTIME, ~, ~]
```

Finally, let's define the *meals.yaml* file, that brings everything together:

Listing 6: meals.yaml

```yaml
my_copter_1:
    defaults: defaults.param
    battery: ~
    tuning: ~
    mission: ~
    header: my_copter_1
    remove_calibration: true
```

In this file we ask *Parasect* to build a parameter file titled `my_copter_1.param`, using the `common` sections of `battery.yaml`, `tuning.yaml` and `mission.yaml` and the `my_copter_1` section of `header.yaml`. Additionally, we ask it to use `calibration.yaml` to remove the calibration parameters from the parameter set. All of these parameter changes will be done on top of `defaults.param`. The path we passed to the `defaults` keyword is relative to the `menu` folder.

Let's now use the *compare* command to build the file.

```
> parasect build -i menu -f apm -o my_parameters
```

The command points to the `menu` folder for build information. The output format is of `apm` type and the file will be placed in a folder named `my_parameters`.

Let's see the contents of `my_parameters/my_copter_1.param`.

```
# Maintainer: George Zogopoulos
# Parameter set for my_copter_1
ACRO_BAL_PITCH      1
ACRO_BAL_ROLL       1
ACRO_OPTIONS        0
...
WPNAV_TER_MARGIN    10
WP_NAVALT_MIN       0
WP_YAW_BEHAVIOR     2
WVANE_ENABLE        0
ZIGZ_AUTO_ENABLE    0
```

Let's compare the produced file with the intended result.

```
 parasect compare my_parameters/my_copter_1.param my_copter_post_tuning.param
File comparison : my_copter_1.param | my_copter_post_tuning.param
================================================================================
--------------------------------------------------------------------------------
Component 1-1:
--------------------------------------------------------------------------------
BARO1_GND_PRESS : X                 < 101079.8
BARO2_GND_PRESS : X                 < 101080.9
BARO3_GND_PRESS : X                 < 0
COMPASS_DIA_X   : X                 < 1
COMPASS_DIA_Y   : X                 < 1
COMPASS_DIA_Z   : X                 < 1
RC10_MAX        : X                 < 1900
RC10_MIN        : X                 < 1100
...
RC9_MIN         : X                 < 1100
RC9_TRIM        : X                 < 1500
STAT_RUNTIME    : X                 < 123
================================================================================
55 parameters differ
```

55 Parameters are different! But all of them are calibration parameters, that don't exist in `my_copter_1.param`, as we asked. Still, they clutter the comparison. Let's use the `-s` option to ignore them.

Note: We still need to point to the `menu` folder to let *Parasect* know where `calibration.yaml` is, but that can be circumvented by permanently setting the *Parasect* path, as described in Setting Paths.

```
 parasect compare -i menu -s my_parameters/my_copter_1.param my_copter_post_tuning.param
File comparison : my_copter_1.param | my_copter_post_tuning.param
================================================================================
================================================================================
0 parameters differ
```

Great! The produced parameter file is exactly as we wanted it! We can write it in our UAV as many times as we want to reset the parameters to their intended values, without fear of overwritign the calibration!

1.3.3 Another UAV Variant

We now decide to build another, slightly different airframe, named `my_copter_2`. This one will be identical to the previous one, but it will carry a smaller battery. We adapt `battery.yaml` and `header.yaml` accordingly.

Since `my_copter_2` has a different `BATT_CAPACITY` than `my_copter_1` but the same `BATT_LOW_MAH`, we split the battery definition into a `common` part and individual `variants`.

Listing 7: battery.yaml

```yaml
common:
    ingredients:
        - [BATT_LOW_MAH, 1000, Enough juice to RTL]

variants:
    my_copter_1:
        common:
            ingredients:
                - [BATT_CAPACITY, 5000, ~]
    my_copter_2:
        common:
            ingredients:
                - [BATT_CAPACITY, 3000, ~]
```

While we are at it, we also want to define some parameters as *operator* parameters. They will be treated the same as *calibration* parameters, by being removed from the parameter file and we can ignore them in comparisons. This will allow our friend to change them at any time as he pleases to suit his operation better.

To that goat, we add an `operator.yaml` file.

Listing 8: operator.yaml

```yaml
common:
    ingredients:
        - [RTL_ALT, ~, ~]
        - [RTL_CONE_SLOPE, ~, ~]
        - [RTL_LOIT_TIME, ~, ~]
        - [FLTMODE., ~, ~]
```

Finally, we edit the `meals.yaml` file to strip the *operator* parameters too.

Listing 9: meals.yaml

```yaml
my_copter_1:
    defaults: defaults.param
    battery: my_copter_1
    tuning: ~
    mission: ~
    header: my_copter_1
    remove_calibration: true
    remove_operator: true

my_copter_2:
    defaults: defaults.param
    battery: my_copter_2
```

(continues on next page)

(continued from previous page)

```
    tuning: ~
    mission: ~
    header: my_copter_2
    remove_calibration: true
    remove_operator: true
```

Let's build the files anew and compare them.

```
> parasect build -i menu -f apm -o my_parameters
> parasect compare my_parameters/my_copter_1.param my_parameters/my_copter_2.param
File comparison : my_copter_1.param | my_copter_2.param
================================================================================
--------------------------------------------------------------------------------
Component 1-1:
--------------------------------------------------------------------------------
BATT_CAPACITY : 5000              | 3000
================================================================================
1 parameters differ
```

Excellent! That's just what we wanted!

Let's give this new airframe to a friend! He needs a platform to brush up his flyig skills.

1.3.4 Read-Only Parameters

Oh no! Our friend came back saying that his drone crashed! He says that suddenly, as the battery got low it fell out of the sky. First things first, let's compare the ideal parameter file from the actual one, as our friend gave it to us (called `friend_dump.param`).

```
> parasect compare -i menu -s my_parameters/my_copter_2.param friend_dump.param
File comparison : my_copter_2.param | friend_dump.param
================================================================================
--------------------------------------------------------------------------------
Component 1-1:
--------------------------------------------------------------------------------
BATT_FS_CRT_ACT : 0                | 5
================================================================================
1 parameters differ
```

Oh dear… he had set the critical battery failsafe action to *Terminate*, inadvertently causing the crash. We will repair his UAV, but let's make sure that doesn't happen again, by making the `BATT_FS_CRT_ACT` parameter *read-only* and use the appropriate workflow involving the apj-tool to bake in its default read-only value.

We add it in the `mission.yaml` file and mark it accordingly.

Listing 10: mission.yaml

```
common:
    ingredients:
        - [MIS_OPTIONS, 0, The default behaviour is what I want]
        - [BATT_FS_CRT_ACT, 4, Do the best thing possible apart from crashing @READONLY]
```

Parasect can scan the *reasoning* section for the keyword `@READONLY` and add it in the parameter file. But `.param` files containing the `@READONLY` flag cause errors when they are loaded in normal GCSs, like MAVProxy. Thus, the `.param`

file to be used by the `apj_tool.py` will be exported as a different format: *apj*.

We can build only that meal with the desired format

```
> parasect build -i menu -f apj -c my_copter_2 -o my_parameters
```

and then inspect the resulting file.

Listing 11: my_copter_2.param

```
...
BATT_CRT_VOLT       0
BATT_CURR_PIN       12
BATT_FS_CRT_ACT     4       @READONLY
BATT_FS_LOW_ACT     0
BATT_FS_VOLTSRC     0
...
```

That's just what we need.

Now, to bake the *read-only* status in the firmware, we need to use the apj_tool. Unfortunately, *apj_tool* can fit only 8 kilobytes of parameters in the `.apj` file, whilst our file is a lot larger.

```
> du -h my_parameters/my_copter_2.param
24K my_parameters/my_copter_2.param
```

We have to make a concession and strip our parameter file from the default parameters. The downside is that we can no longer use the same parameter file with our GCS to reset all the parameters to the intended value. But we can easily circumvent this issue by simply creating a parameter file exclusively for this use, and explicity setting defaults to `None`.

Listing 12: meals.yaml

```
...

my_copter_2:
    defaults: defaults.param
    battery: my_copter_2
    tuning: ~
    mission: ~
    header: my_copter_2
    remove_calibration: true
    remove_operator: true

my_copter_2_apj:
    defaults: ~
    battery: my_copter_2
    tuning: ~
    mission: ~
    header: my_copter_2
    remove_calibration: true
    remove_operator: true
    add_new: true
```

Note how we have added the `add_new: true` entry in the new meal. This is necessary, because *Parasect* by default does not allow creating new parameters in a set, to prevent typographical errors. However, in this case we indeed want to start on an empty slate, without a default parameter set, so we have to explicitly allow new parameter names.

The resulting parameter set is, as expected:

Listing 13: my_copter_2_apj.param

```
# Maintainer: George Zogopoulos
# Parameter set for my_copter_2
ATC_ANGLE_BOOST     1
ATC_ANG_PIT_P       5
ATC_ANG_RLL_P       5
ATC_RATE_P_MAX      60
ATC_RATE_R_MAX      60
BATT_CAPACITY       3000
BATT_FS_CRT_ACT     4       @READONLY
BATT_LOW_MAH        1000
LOIT_SPEED  1500
MIS_OPTIONS 0
```

We can now bake in the parameters in our `.apj` file with the apj_tool. We assume that `arducopter.apj` and `apj_tool.py` has been copied into `parasect_files`.

```
> python3 apj_tool.py --set-file my_parameters/my_copter_2_apj.param arducopter.apj
Loaded apj file of length 1809920
Found param defaults max_length=8192 length=282
Setting defaults from my_parameters/my_copter_2_apj.param
Saved apj of length 1809920
```

Success!

1.3.5 Conclusion

This tutorial taught you how to use *Parasect* to compare and create your own parameter sets. Now go forth and don't ever let mixed parameters ruin your day ever again!

See also:

A full reference of the CLI and the API is available here.

1.4 CLI Usage

The primary way of using *Parasect* is as a command-line tool. It provides two functions, `compare` and `build`.

It is strongly recommended that you first read the Concepts that *Parasect* employs, if you plan to make full use of it. A long-form tutorial in using `compare` and `build` with the Ardupilot ecosystem can be found here.

1.4.1 Compare

At its most basic, a comparison between two parameter files can be invoked by:

```
parasect compare <FILE_1> <FILE_2>
```

A comparison table will be printed. Following the MAVLink conventions, all parameters are assumed to fall under a Vehicle ID and a Component ID.

Example:

```
 parasect compare 6fcfa754-186b-41ae-90a4-8de386f712c3.params 607f3c36-a9f8-
↪428e-bc16-2c2615b84291.params
File comparison  : 6fcfa754-186b-41ae-90a4-8de386f712c3.params | 607f3c36-a9f8-
↪428e-bc16-2c2615b84291.params
================================================================================
--------------------------------------------------------------------------------
↪-
Component 1-1:
--------------------------------------------------------------------------------
↪-
ASPD_SCALE       : X                                           < 1
BAT1_A_PER_V     : 36.4                                        | 34
BAT1_CAPACITY    : 4000                                        | 10000
BAT1_V_DIV       : 18.2                                        | 19.7
BAT1_V_EMPTY     : 3.5                                         | 3.55
[...]
```

- Parameters whose values are different in each file are printed in one line each.
- Parameters that don't exist in one of the two files will be marked with an *X*
- Parameters that have practically the same value in both files are not shown.

Typically it is not desirable to show the differences in calibration or operation-specific parameters. `compare` offers additional flags to filter out such parameters.

First create your Meals Menu, filling in at least your *calibration* and *operator Dishes*. Then learn how to point Parasect to it. Finally, you can filter out the calibration parameters by

```
parasect compare -s <FILE_1> <FILE_2>
```

or filter out the operator parameters by

```
parasect compare -u <FILE_1> <FILE_2>
```

The two flags can be combined.

1.4.2 Build

Parasect can generate parameter sets for your autopilot or fleet of autopilots.

First create your Meals Menu. Then, you can generate the parameter sets for all your vehicles with a call similar to:

```
parasect build -o <output_folder> -f <output_format> -i <meals_menu_folder> -d <path_to_
↪default_parameters_file>
```

Here is a partial output of our example Meal Menu, on .csv format:

```
# Parameter file exported by Parasect.
# Parameter name, Parameter value
BAGUETTE,1
BEEF,1
CUCUMBER,2
EGG,1
FLOUR,2
GRAVY,0.01
OIL,0.5
POTATOES,2
SALT,0.01
STRAWBERRY,10
THYME,0.001
```

1.5 API Usage

Parasect also exposes an API for the `compare` and `build` commands, that can be useful in project automation. Their arguments are identical to their CLI counterparts.

See the *API Reference* for the full documentation.

CHAPTER

TWO

CONCEPTS

2.1 The Restaurant Analogy

Parasect employs a **restaurant analogy** to encode its functionality.

When you interact with *Parasect*, you define a **Menu of Meals** that *Parasect* can prepare for you.

The Menu contains multiple **Meals**. Each Meal represents a distinct parameter set, a configuration that should be unique and reflect a unique unmanned vehicle. The Menu is defined in the `meals.yaml` file where each Meal is named and represented by a dictionary.

Example `meals.yaml` *file:*

```
snack:
  fruit: single
  header: ~
  footer: ~

breakfast:
  dairy: cereal
  header: ~
  footer: ~

light_meal:
  salad: ~
  meat_and_potatoes: ~
  remove_calibration: True
  remove_operator: True
  header: ~
  footer: ~

spicy_meal:
  salad: ~
  meat_and_potatoes: spicy/extra
  remove_calibration: True
  remove_operator: True
  header: ~
  footer: ~

full_meal:
  salad: ~
  meat_and_potatoes: ~
```

(continues on next page)

(continued from previous page)

```
  bread: baguette
  desert: ~
  header: ~
  footer: full_meal

christmass_at_grandmas:
  parent: full_meal
  salad: village
  extra_quantities: ~
  add_new: True

dangerous_combinations:
  parent: breakfast
  coffee: filter
  add_new: True
```

The Meal is made up of **Dishes**. Each Dish represents a notional subset of the autopilot's parameters, which makes it easier to define and manage. In this case `light_meal` contains the Dishes `salad`, `meat_and_potatoes`, `header` and `footer`. The rest of the key-value pairs will be explained later.

Each Dish is defined in a separate file and contains a **common Recipe**, which specifies which **Ingredients** make up the Dish, as well as a list of **Allergens** that should not exist in the Dish.

Furthermore, one or more Dish **Variants** can be specified in the Dish file, which are specializations of the common Recipe. They define more Ingredients and more Allergens. The Variants can be infinitely nested.

In the Meal dictionary, the Dish names are defined in the dictionary key, while the dictionary value specifies the Dish **Variant** that should be used for that meal. If no Variant is specified (the value is set to ~, i.e. `None`) then the `common` Recipe is used. On the contrary, a value of `variant_1/subvariant_a` signifies that the Meals needs to be specialized with Variant Recipes.

Example: salad Dish file:

```
common:
  ingredients:
    - [CUCUMBER, 2, ~]
    - [OIL, 0.5, ~]
    - [SALT, 0.01, ~]
  allergens:
    substances:
      - [VINEGAR, ~, Mom doesn't like it]

variants:
  village:
    common:
      ingredients:
        - [TOMATO, 3, ~]
        - [WHITE_CHEESE, 0.5, ~]
```

The user is free to create as many Custom Dishes he wants under the `custom_dishes` folder and then refer to them in `meals.yaml`. Not all Meals need to refer to all available Dishes. However, there are Staple Dishes that serve special purposes and are defined in the `staple_dishes` folder; their filenames are significant and should not be altered. Still the user is expected to fill in their content.

Such Dishes are:

- `calibration.yaml`: The calibration parameters can be defined there.
- `operator.yaml`: The operator parameters can be defined there.
- `header.yaml`: The header text of the generated parameter files can be defined there.
- `footer.yaml`: The footer text of the generated parameter files can be defined there.

2.2 Menu Folder Structure

In the end, your meals menu folder will have the following structure:

```
menu/
├── custom_dishes/
│   ├── dish_1.yaml
│   ├── dish_2.yaml
├── staple_dishes/
│   ├── calibration.yaml
│   ├── header.yaml
│   ├── footer.yaml
│   ├── operator.yaml
└── meals.yaml
```

2.3 Functional Requirements

Parasect builds parameter sets while satisfying and enforcing these requirements:

1. If a *Default Parameters* set is provided, it will form the basis of the Meal Ingredients. These default parameters are often autogenerated by the autopilot toolchain.

   Why?: It is highly recommended to build upon a default parameters set, because you can thus enforce the default value in all of the parameters you don't wish to alter.

2. If you provide a default parameters set, it is an illegal operation to define and set an Ingredient that doesn't exist in it.

   Why?: This is to catch spelling mistakes in the parameter names. However, this functionality can be disabled to enable specific workflows.

3. The Variant Recipes should not re-define an Ingredient that their parent Common Recipe already defined.

   Why?: This is to make it easier to trace which part of the Dish file ends up in the Meal. If you need a Dish Variant to alter the value of an Ingredient found in the Common Dish, you can create another Variant and define your Ingredients there.

4. Each Ingredient should be set at most once across all Dishes:

   Why?: Remember that in the end, the whole Meal is exported as a flat list of parameters. You don't want the same parameter being defined more than once.

These requirements allow the parameter set generated by *Parasect* to be idempotent:

1. You can apply it on your autopilot and be sure that it is configured exactly to your intent.
2. You can reapply it as many times as you wish, without affecting the existing calibration or the operator configuration.

You can also compare it to the current autopilot parameter set. The calibration and operator parameters can be ignored so only meaningful changes will surface.

It is recommended, but not enforced to:

1. Define calibration parameters as Ingredients in the *calibration.yaml* Dish and remove them from the Meal, using the *remove_calibration* flag.

   Why?: Each vehicle has its own unique calibration. Overwriting this calibration values is not appropriate.

2. Define operation-related parameters as Ingredients in the *operator.yaml* Dish and remove them from the Meal, using the *remove_operator* flag.

   Why?: Each operator should have the freedom to choose the values of certain parameters. Overwriting them is not appropriate.

CHAPTER

THREE

REFERENCE

3.1 CLI Reference

3.1.1 parasect

Main CLI entry point.

```
parasect [OPTIONS] COMMAND [ARGS]...
```

Options

--debug

Generate log file.

--version

Show the version and exit.

build

Build command.

```
parasect build [OPTIONS]
```

Options

-o, --output_folder <output_folder>

Specify the output folder.

-c, --configuration <config>

Specify a single Meal to build.

-f, --format <format>

Select autopilot format. Read the documentation of *Formats* for more information.

Options

csv | px4 | px4afv1 | px4afv2 | apm | apj

-i, --input_folder <input_folder>

Specify the folder from which to read configurations and parameters.

-d, --default_parameters <default_parameters>
Specify the default parameters file to apply to all Meals.

compare

Compare command.

```
parasect compare [OPTIONS] FILE_1 FILE_2
```

Options

-i, --input_folder <input_folder>
The directory where the Meals Menu is created, containing at least the *calibration* and *operator* staple dishes. Necessary when the *nocal* and *noop* options are set.

-s, --suppress-calibration
Don't compare calibration parameters.

-u, --suppress-operator-defined
Don't compare operator-selectable parameters.

-c, --component <component>
Compare the parameters of a specific component. Refers to MAVLink-type Component IDs.

Arguments

FILE_1
Required argument

FILE_2
Required argument

3.2 API Reference

Parasect package.

class parasect.**Formats**(*value*)
Supported output formats.

apj = 'apj'
File compatible with Ardupilot's apj tool.

apm = 'apm'
Ardupilot-compatible file.

csv = 'csv'
Simple parameter name, value .csv file.

px4 = 'px4'
QGroundControl-style parameter file.

px4afv1 = 'px4afv1'

Legacy PX4 airframe file, prior to version 1.11.

px4afv2 = 'px4afv2'

New PX4 airframe file, version 1.11 and later.

parasect.**build**(*meal_ordered*, *format*, *input_folder*, *default_params*, *output_folder=None*, *sitl=False*)

Build parameter sets.

Parameters

- **meal_ordered** (*str | None*) – Specify which meal should be built. All meals specified in the menu will be built if left None.
- **format** (Formats) – The autopilot format to export as.
- **input_folder** (*str | None*) – The directory where the Meals Menu is created.
- **default_params** (*str | None*) – If set, this file will provide the full parameters set for all the Meals.
- **output_folder** (*str | None*) – The directory where the meals will be exported.
- **sitl** (*bool*) – Filter for Meals marked as "sitl". None: Disregard this meal keyword. True: Only build "sitl" meals. False: Don't build "sitl" meals.

Raises

ValueError – If *meal_ordered* is None (hence all the meals will be exported) but no *output_folder* is specified.

Return type

None

parasect.**compare**(*file_1*, *file_2*, *input_folder*, *nocal*, *noop*, *component*)

Compare two parameter files.

Parameters

- **file_1** (*str*) – The path to the first parameter file to compare.
- **file_2** (*str*) – The path to the second parameter file to compare.
- **input_folder** (*str | None*) – Necessary when the *nocal* and *noop* options are set. The directory where the Meals Menu is created, containing at least the *calibration* and *operator* staple dishes.
- **nocal** (*bool*) – Don't compare the calibration parameters.
- **noop** (*bool*) – Don't compare the operator parameters.
- **component** (*int | None*) – Compare the parameters of a specific component. Refers to MAVLink-type Component IDs.

Returns

A string with the comparison table contents.

Return type

str

CHAPTER

FOUR

CONTRIBUTOR GUIDE

Thank you for your interest in improving this project. This project is open-source under the MIT license and welcomes contributions in the form of bug reports, feature requests, and pull requests.

Here is a list of important resources for contributors:

- Source Code
- Documentation
- Issue Tracker
- Code of Conduct

4.1 How to report a bug

Report bugs on the Issue Tracker.

When filing an issue, make sure to answer these questions:

- Which operating system and Python version are you using?
- Which version of this project are you using?
- What did you do?
- What did you expect to see?
- What did you see instead?

The best way to get your bug fixed is to provide a test case, and/or steps to reproduce the issue.

4.2 How to request a feature

Request features on the Issue Tracker.

4.3 How to set up your development environment

First of all, clone the *Parasect* repository.

Then, you need Python 3.8+ and the following tools:

- Poetry
- Nox
- nox-poetry

Install Poetry by downloading and running the following script:

```
$ curl -sSL https://install.python-poetry.org | python3 -
```

Install Nox and nox-poetry:

```
$ pipx install nox
$ pipx inject nox nox-poetry
```

Navigate into the location where you cloned *Parasect* and install the package with development requirements:

```
$ poetry install
```

You can now run an interactive Poetry shell, giving you access to the virtual environment.

```
$ poetry shell
```

4.4 How to test the project

The *Parasect* CLI offers debug output in the form of a `parasect.log` file. The file can be created by issuing the `--debug` flag when calling *Parasect*.

```
$ parasect --debug <rest_of_the_arguments>
```

Additionally, you can run the full test suite:

```
$ nox
```

List the available Nox sessions:

```
$ nox --list-sessions
```

You can also run a specific Nox session. For example, invoke the unit test suite like this:

```
$ nox --session=tests
```

Unit tests are located in the `tests` directory, and are written using the pytest testing framework.

4.5 How to submit changes

Open a pull request to submit changes to this project.

Your pull request needs to meet the following guidelines for acceptance:

- The Nox test suite must pass without errors and warnings.
- Include unit tests. You must provide tests covering 100% (or as close as possible) of your code changes and additions.
- If your changes add functionality, update the documentation accordingly.

Feel free to submit early. Mark your PR as WIP (Work in Progress) in the PR title, to signal that it is not in its final form yet.

To run linting and code formatting checks before committing your change, you can install pre-commit as a Git hook by running the following command:

```
$ nox --session=pre-commit -- install
```

If you are unsure how your contribution would fit in *Parasect*, feel free to raise an issue for discussion. It is always preferable to spend a little time discussing your approach, instead of spending a lot of effort on a large chunk of code that might be rejected.

CHAPTER

FIVE

CONTRIBUTOR COVENANT CODE OF CONDUCT

5.1 Our Pledge

We as members, contributors, and leaders pledge to make participation in our community a harassment-free experience for everyone, regardless of age, body size, visible or invisible disability, ethnicity, sex characteristics, gender identity and expression, level of experience, education, socio-economic status, nationality, personal appearance, race, religion, or sexual identity and orientation.

We pledge to act and interact in ways that contribute to an open, welcoming, diverse, inclusive, and healthy community.

5.2 Our Standards

Examples of behavior that contributes to a positive environment for our community include:

- Demonstrating empathy and kindness toward other people.
- Being respectful of differing opinions, viewpoints, and experiences.
- Giving and gracefully accepting constructive feedback.
- Accepting responsibility and apologizing to those affected by our mistakes, and learning from the experience.
- Focusing on what is best not just for us as individuals, but for the overall community.

Examples of unacceptable behavior include:

- The use of sexualized language or imagery, and sexual attention or advances of any kind.
- Trolling, insulting or derogatory comments, and personal or political attacks.
- Public or private harassment.
- Publishing others' private information, such as a physical or email address, without their explicit permission.
- Other conduct which could reasonably be considered inappropriate in a professional setting.

5.3 Enforcement Responsibilities

Community leaders are responsible for clarifying and enforcing our standards of acceptable behavior and will take appropriate and fair corrective action in response to any behavior that they deem inappropriate, threatening, offensive, or harmful.

Community leaders have the right and responsibility to remove, edit, or reject comments, commits, code, wiki edits, issues, and other contributions that are not aligned to this Code of Conduct, and will communicate reasons for moderation decisions when appropriate.

5.4 Scope

This Code of Conduct applies within all community spaces, and also applies when an individual is officially representing the community in public spaces. Examples of representing our community include using an official e-mail address, posting via an official social media account, or acting as an appointed representative at an online or offline event.

5.5 Enforcement

Instances of abusive, harassing, or otherwise unacceptable behavior may be reported to the community leaders responsible for enforcement at geo.zogop.papal@gmail.com. All complaints will be reviewed and investigated promptly and fairly.

All community leaders are obligated to respect the privacy and security of the reporter of any incident.

5.6 Enforcement Guidelines

Community leaders will follow these Community Impact Guidelines in determining the consequences for any action they deem in violation of this Code of Conduct:

5.6.1 1. Correction

Community Impact: Use of inappropriate language or other behavior deemed unprofessional or unwelcome in the community.

Consequence: A private, written warning from community leaders, providing clarity around the nature of the violation and an explanation of why the behavior was inappropriate. A public apology may be requested.

5.6.2 2. Warning

Community Impact: A violation through a single incident or series of actions.

Consequence: A warning with consequences for continued behavior. No interaction with the people involved, including unsolicited interaction with those enforcing the Code of Conduct, for a specified period of time. This includes avoiding interactions in community spaces as well as external channels like social media. Violating these terms may lead to a temporary or permanent ban.

5.6.3 3. Temporary Ban

Community Impact: A serious violation of community standards, including sustained inappropriate behavior.

Consequence: A temporary ban from any sort of interaction or public communication with the community for a specified period of time. No public or private interaction with the people involved, including unsolicited interaction with those enforcing the Code of Conduct, is allowed during this period. Violating these terms may lead to a permanent ban.

5.6.4 4. Permanent Ban

Community Impact: Demonstrating a pattern of violation of community standards, including sustained inappropriate behavior, harassment of an individual, or aggression toward or disparagement of classes of individuals.

Consequence: A permanent ban from any sort of public interaction within the community.

5.7 Attribution

This Code of Conduct is adapted from the Contributor Covenant, version 2.0, available at https://www.contributor-covenant.org/version/2/0/code_of_conduct.html.

Community Impact Guidelines were inspired by Mozilla's code of conduct enforcement ladder.

For answers to common questions about this code of conduct, see the FAQ at https://www.contributor-covenant.org/faq. Translations are available at https://www.contributor-covenant.org/translations.

CHAPTER

SIX

MIT LICENSE

Welcome to *Parasect*, a utility for manipulating parameter sets for autopilots!

CHAPTER

SEVEN

FEATURES

Parasect has two-fold capabilities:

1. Compare two parameter sets and highlighting their differences.
2. Parsing from user-defined content and generating new parameter sets, ready for loading into an autopilot.

List of currently supported autopilots:

- Ardupilot
- PX4

CHAPTER

EIGHT

REQUIREMENTS

Parasect is a pure-Python project. Its requirements are managed by the Poetry dependency manager. When you install *Parasect* via pip its requirements will also be installed automatically.

Currently *Parasect* has been tested:

- in Continuous Integration servers for **Ubuntu Linux**, **Windows**
- manually in **Ubuntu Linux**.

CHAPTER

NINE

INSTALLATION

You can install *Parasect* via pip from PyPI:

```
$ pip install parasect
```

If you don't care about using *Parasect* as a library and are only interested in command-line use, you can also use pipx, that provides better isolation from the rest of the system:

```
$ pipx install parasect
```

CHAPTER

TEN

USAGE

Parasect is primarily used as a command-line program. In its simplest form, two parameter files can be compared via:

```
$ parasect compare <FILE_1> <FILE_2>
```

The usage for building parameter sets is more involved. Please see the Command-line Reference for details.

Additionally, it exposes a minimal API, enabling automated operations. This is described in the API Reference.

It is strongly recommended that you read the Concepts that *Parasect* employs, if you plan to make full use of it.

CHAPTER

ELEVEN

CONTRIBUTING

Contributions are very welcome. To learn more, see the Contributor Guide.

CHAPTER

TWELVE

LICENSE

Distributed under the terms of the MIT license, *Parasect* is free and open source software.

CHAPTER

THIRTEEN

ISSUES

If you encounter any problems, please file an issue along with a detailed description.

CHAPTER

FOURTEEN

CREDITS

This project was sponsored by Avy B.V., a UAV company in Amsterdam.

This project was generated from @cjolowicz's Hypermodern Python Cookiecutter template.

The project logo was created by Cynthia de Vries.

PYTHON MODULE INDEX

INDEX

Symbols

A

B

C

E

F

M

P