Fastest-lap

Release v0.5

Juan Manzanero

Aug 16, 2023

GETTING STARTED

Fastest-lap is a vehicle dynamics simulator. It can be used to understand vehicle dynamics, to learn about driving techniques, to design car prototypes, or just for fun!

CHAPTER

ONE

OVERVIEW

Fastest-lap an open source (MIT) written in C++, and it is usable from any scripting language such as Python and MATLAB. It is cross-platform, and it has been extensively tested on Windows 10, Linux (Ubuntu) and Mac.

Fastest-lap is characterized for its simplity and ease of use. With less than 10 lines of code you will be running and analyzing your first laptime simulation.

1.1 Installation

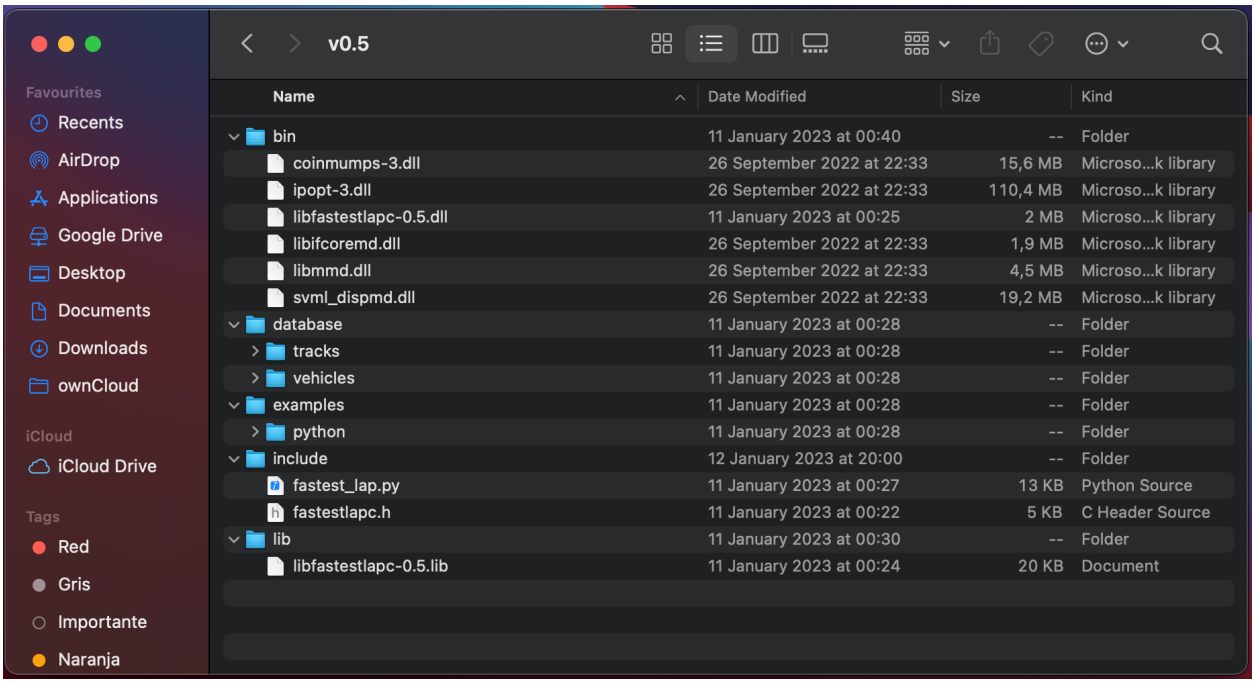
Fastest-lap is ready to use once you have the dynamic library `libfastestlapc-VERSION`, plus other libraries such as Ipopt and Mumps. The following steps describe how to get such library.

1.1.1 Windows 10 (Precompiled binaries)

Prerequisites: none :)

Download the latest release of Fastest-lap, and unzip its contents to your desired folder. If you don't want bad things to happen, keep all the files into their original directory relative to the root!

- **bin** contains the Fastest-lap C++ dynamic library: `libfastestlapc-VERSION.dll` plus other dynamic libraries fastest-lap depends on.
- **include** contains the Fastest-lap C header `fastestlapc.h` and the python module `fastest_lap.py`
- database includes track and vehicle models inputs
- examples contains python notebook examples



1.1.2 Linux and Mac

Prerequisites:

- C/C++ and Fortran compilers
- CMake
- (mac) Command line tools

Fastest-lap has been successfully compiled with:

Mac

- Apple clang version 13.0.0
- GNU Fortran (Homebrew GCC 11.2.0_3) 11.2.0

Linux

- g++ (GCC) 11.2.0
- GNU Fortran (GCC) 11.2.0

The steps to build this project using CMake are the usual:

1) Set a variable `FASTESTLAP` to the top level directory

```
$ export FASTESTLAP=/path/to/fastest-lap
```

2) Create a build folder.

```
$ mkdir ${FASTESTLAP}/build
```

3) From the build folder, run cmake

```
$ cd ${FASTESTLAP}/build && cmake ..
```

The following options can be set to customise the compilation:

- `CMAKE_BUILD_TYPE`: Debug/Release (defaults to RELEASE)
- `CMAKE_C_COMPILER`: /path/to/cc
- `CMAKE_CXX_COMPILER`: /path/to/cxx
- `CMAKE_Fortran_COMPILER`: /path/to/fc

At this stage, CMake will download and install all the thirdparty dependencies. This can take up to **30 minutes**

4) Compile

```
$ make
```

If make was successful, the dynamic library `libfastestlapc.dylib` (Mac) or `libfastestlap.so` (Linux) should be found in `${FASTESTLAP}/build/lib`

5) (Optional but recommended) Test

```
$ ctest --verbose
```

1.1.3 Troubleshooting

If you encounter any issue while repeating the steps, first take a look in the Issues section of the repository. Probably someone has been there before :). If still you cannot fix the issue, feel free to open a new issue.

1.2 Quickstart

Tutorial prerequisites

- A suitable python shell (e.g. Anaconda)
- Knowledge of XML (this video can serve as a nice introduction)

Fastest-lap can be very easily invoked from scripting languages such as Python and MATLAB. Let's do a super fast and simple system test: let's compute a lap around Circuit de Catalunya using Python.

This example is based on the python notebook 1-simple-lap that can be found in the repository.

The entry point to fastest-lap, is the file `fastest_lap.py`. It is located under `examples/python` for Mac and Linux, and under `include` for Windows. This file already knows how to find the C++ dynamic library, so you do not need to worry about it.

We start by including the `fastest_lap` module. In this tutorial, every string called as `"/path/to/whatever"` is a placeholder for you to introduce the real path in your system to the indicated file.

```
import sys,os,inspect
sys.path.append("/path/to/folder/where/fastest_lap.py/is/found/")
import fastest_lap
```

This command imports the Fastest-lap python API functions, and also loads the C++ library. The C++ library is a collection of functions responsible of the computations, plus its internal memory where cars, circuits, and results are stored. From this point, Fastest-lap is ready.

We can create a car model from an XML data file by calling `create_vehicle_from_xml()`

```
vehicle_name = "car"
fastest_lap.create_vehicle_from_xml(vehicle_name, "/path/to/database/vehicles/f1/
↪mercedes-2020-catalunya.xml");
```

This creates a variable of type 3dof F1 car in the Fastest-lap C++ internal memory by the name of `"car"`. If you try to create another variable with the same name, the application will throw an error.

Next, we can load a circuit from an XML file by calling `create_track_from_xml()`. This XML file contains a mesh of the track centerline, and precomputed values for the heading angle, curvature, and distance to the track limits.

```
track_name = "catalunya"
fastest_lap.create_track_from_xml(track_name, "/path/to/database/tracks/catalunya/
↪catalunya_adapted.xml");
```

This creates a variable of type `track` in the internal memory with the name `"catalunya"`. We can retrieve additional data from the track that is needed for future calculations, such as the arclength (traveled distance along the track centerline) mesh. We can do so by calling `track_download_data()`

```
s = fastest_lap.track_download_data(track_name,"arclength");
```

Car and circuit are ready. Let's compute the optimal laptime. We start by setting the options configure the computation. Options are passed through a string written in XML format. For now, we will just set two options:

- First, the simulation produces timeseries for the dynamic variables (think of the velocity, positions, forces, etc). These are stored as variables in the Fastest-lap internal memory that can be later retrieved. We just must specify a virtual folder where these variables will be stored, in this case we use `"run/"`. The velocity will be later accessed as `run/chassis.velocity.x`.
- Second, while the simulator runs we can have some screen output to see the progress. This output is redirected from Ipopt, and it has up to 12 print levels (print level 0 produces no output). We chose `print_level = 5`, as it gives enough representative data of how the solution is converging.

```
options  = "<options>"
options += "    <output_variables>"
options += "        <prefix>run/</prefix>"
options += "    </output_variables>"
options += "    <print_level> 5 </print_level>"
options += "</options>"
```

Check in this link the full list of options available.

To compute the laptime, we call `optimal_laptime()`. We pass to this funciton the vehicle, the track, the arclength mesh, and the options. Its output is the names of all the generated variables in the internal memory. We can redirect this data to `fastest_lap.download_variables()`, which downloads the variables to the python workspace. This variables are stored in a dictionary `run` and they can be accessed by name (e.g. `x = run["chassis.position.x"]`).

```
run = fastest_lap.download_variables(*fastest_lap.optimal_laptime(vehicle_name, track_
↪name, s, options));
```

```
This is Ipopt version 3.14.8, running with linear solver MUMPS 5.5.0.

Number of nonzeros in equality constraint Jacobian...:   154734
Number of nonzeros in inequality constraint Jacobian.:    18122
Number of nonzeros in Lagrangian Hessian.............:    85034

Total number of variables............................:    10455
                     variables with only lower bounds:        0
                variables with lower and upper bounds:    10455
                     variables with only upper bounds:        0
Total number of equality constraints.................:     9061
Total number of inequality constraints...............:     4182
        inequality constraints with only lower bounds:        0
inequality constraints with lower and upper bounds:     4182
        inequality constraints with only upper bounds:        0

iter    objective    inf_pr   inf_du lg(mu)  ||d||  lg(rg) alpha_du alpha_pr  ls
0  3.3056907e+02 2.69e-01 4.57e-02  -1.0 0.00e+00    -  0.00e+00 0.00e+00   0
1  3.2889671e+02 1.14e+00 8.46e-01  -1.0 1.18e+00    -  7.05e-01 1.00e+00f  1
2  3.1958071e+02 3.11e-01 1.13e-01  -1.0 4.22e-01    -  8.49e-01 1.00e+00f  1
3  1.6821222e+02 2.80e+00 7.32e-01  -1.0 1.94e+01    -  6.60e-01 1.00e+00f  1
4  1.2837223e+02 1.21e+00 6.27e-01  -1.0 2.09e+01    -  6.77e-01 1.00e+00f  1
5  9.9468468e+01 8.94e-01 7.43e-01  -1.0 2.36e+01    -  6.96e-01 1.00e+00f  1
6  9.2840692e+01 5.66e-01 3.26e-01  -1.7 9.36e+00    -  8.21e-01 1.00e+00h  1
7  8.5271100e+01 4.41e-01 1.41e-01  -1.7 1.11e+01    -  1.00e+00 1.00e+00f  1
8  8.5277352e+01 6.13e-02 5.65e-01  -1.7 5.96e-01   0.0 1.00e+00 1.00e+00h  1
9  8.5278537e+01 1.74e-03 2.30e-01  -1.7 8.39e-02   0.4 1.00e+00 1.00e+00h  1
iter    objective    inf_pr   inf_du lg(mu)  ||d||  lg(rg) alpha_du alpha_pr  ls
10  8.5276892e+01 1.53e-04 1.03e-02  -1.7 1.16e-02  -0.1 1.00e+00 1.00e+00h  1
11  8.0956232e+01 3.37e-01 1.06e-01  -3.8 8.10e+00    -  6.33e-01 6.48e-01f  1
12  7.8792474e+01 2.86e-01 1.13e-01  -3.8 1.02e+01    -  5.74e-01 3.43e-01h  1
13  7.6337073e+01 2.45e-01 1.30e-01  -3.8 1.14e+01    -  2.85e-01 4.48e-01h  1
14  7.6345553e+01 1.69e-01 1.58e+00  -3.8 3.10e-01  -0.5 2.83e-02 1.00e+00h  1
15  7.6346316e+01 5.11e-02 9.21e-01  -3.8 1.20e-01  -0.1 8.55e-01 1.00e+00h  1
16  7.6329460e+01 9.29e-03 2.23e-01  -3.8 5.64e-02  -0.6 1.00e+00 1.00e+00h  1
17  7.6287363e+01 8.06e-03 4.78e-02  -3.8 1.29e-01  -1.1 1.00e+00 1.00e+00h  1
18  7.6185877e+01 1.64e-02 3.11e-02  -3.8 2.66e-01  -1.5 1.00e+00 1.00e+00h  1
19  7.6127662e+01 8.19e-02 4.57e-02  -3.8 9.62e-01  -2.0 3.67e-01 2.42e-01h  1
iter    objective    inf_pr   inf_du lg(mu)  ||d||  lg(rg) alpha_du alpha_pr  ls
20  7.6118606e+01 3.31e-02 1.29e-01  -3.8 2.20e-01  -0.7 1.00e+00 1.00e+00h  1
21  7.6085306e+01 1.53e-03 5.87e-03  -3.8 8.50e-02  -1.2 1.00e+00 1.00e+00h  1
22  7.5992174e+01 1.20e-02 6.35e-03  -3.8 2.57e-01  -1.6 1.00e+00 1.00e+00h  1
23  7.5866448e+01 3.66e-02 2.11e-01  -3.8 5.46e-01  -2.1 2.75e-01 5.12e-01h  1
24  7.5782968e+01 3.18e-02 3.21e-01  -3.8 1.38e+00  -2.6 1.00e+00 1.34e-01h  1
25  7.5574503e+01 2.03e-02 2.85e-01  -3.8 3.08e-01  -2.2 3.78e-01 9.34e-01h  1
26  7.5372005e+01 1.57e-02 2.10e-01  -3.8 7.58e-01  -2.6 1.00e+00 3.63e-01h  1
27  7.5209407e+01 1.00e-02 2.79e-02  -3.8 2.86e-01  -2.2 1.00e+00 8.62e-01h  1
28  7.5140991e+01 2.26e-03 1.74e-03  -3.8 1.07e-01  -1.8 1.00e+00 1.00e+00h  1
...
```

(continues on next page)

(continued from previous page)

```
...
110  7.3367142e+01 1.39e-04 3.83e-04  -8.6 7.39e-01    -  9.87e-01 9.57e-01h  1
111  7.3367124e+01 8.70e-06 9.28e-07  -8.6 2.18e-01    -  1.00e+00 1.00e+00h  1
112  7.3367124e+01 1.91e-07 6.65e-08  -8.6 3.83e-02    -  1.00e+00 1.00e+00h  1
113  7.3367124e+01 4.79e-09 1.72e-09  -8.6 1.76e-03    -  1.00e+00 1.00e+00h  1
114  7.3367123e+01 4.94e-09 1.06e-09 -11.0 4.98e-03    -  1.00e+00 1.00e+00h  1
115  7.3367123e+01 5.14e-12 3.13e-13 -11.0 2.68e-05    -  1.00e+00 1.00e+00h  1

Number of Iterations....: 115

                                   (scaled)                 (unscaled)
Objective...............:   7.3367123134545238e+01    7.3367123134545238e+01
Dual infeasibility......:   3.1333330545970075e-13    3.1333330545970075e-13
Constraint violation....:   5.1400678418439538e-12    5.1400678418439538e-12
Variable bound violation:   9.3933291700487587e-11    9.3933291700487587e-11
Complementarity.........:   9.9178403300925627e-12    9.9178403300925627e-12
Overall NLP error.......:   9.9178403300925627e-12    9.9178403300925627e-12


Number of objective function evaluations             = 121
Number of objective gradient evaluations             = 116
Number of equality constraint evaluations            = 121
Number of inequality constraint evaluations          = 121
Number of equality constraint Jacobian evaluations   = 116
Number of inequality constraint Jacobian evaluations = 116
Number of Lagrangian Hessian evaluations             = 115
Total seconds in IPOPT                               = 82.715

EXIT: Optimal Solution Found.
```

After approximately 1 minute, the results should be ready. You can plot, analyze, and visualize all the data stored in `run`. You can list all the available variables calling `run.keys()`.

A quick way to plot the trajectory is calling `plot_optimal_laptime`

```
import numpy as np
fastest_lap.plot_optimal_laptime(s, run["chassis.position.x"], run["chassis.position.y"],
↪ track_name);
plt.gca().invert_xaxis()
```

And that's all folks, I hope you enjoy it!

1.3 Overview

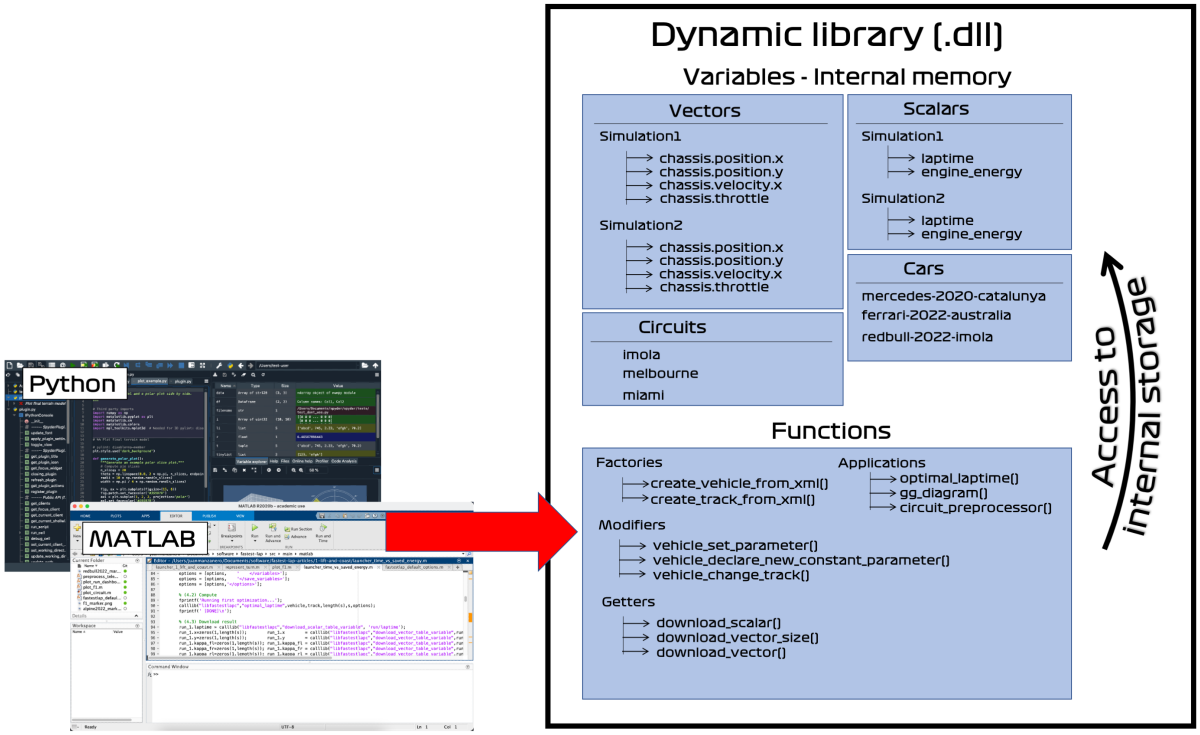
The philosophy of this software is to find the perfect balance between performance and versatility. Code written in C++ is super fast and efficient, and scripting languages such as MATLAB and Python are versatile and perfect for data processing and visualization, plus are well known to everybody.

What we came up with, it's a C++ world in the form of a dynamic library, that is controlled through Python/MATLAB. This library (the `.dll` file for Windows, `.dylib` for Mac, and `.so` for Linux) is responsible for all the heavy computations such as gg diagrams, laptime optimizations, and circuit preprocessors.

A dynamic library is a collection of **functions** plus **internal memory** to store data (scalars, vectors, cars, circuits,...). The functions are the entry port to the Fastest-lap library from Python/MATLAB. One can create/modify/delete variables in the internal memory, and run applications such as gg-diagrams, optimal laptimes, and circuit preprocessings. After an application has been run, the results will be stored in the internal memory, which can be later *downloaded* to Python/MATLAB.

Examples of the functions you can find are:

- **Factories**: to create variables in the internal memory (e.g. to create a car, a circuit, a scalar variable, a vector)
- **Modifiers**: to modify variables stored in the internal memory (e.g. to modify a setup parameter of a created car)
- **Applications**: to run simulations: circuit preprocessor, gg diagram, and optimal laptime
- **Getters**: to retrieve the results and data back to Python/MATLAB
- **Destructors**: to delete variables once they are not needed anymore



The variables are internally stored by name (as in every programming language), and there can only one and only variable with the same name. That is, if there's a `vehicle` called `"my_car"`, one cannot create another `vehicle` called `"my_car"` or a `vector` called `"my_car"`.

Also, one can use *paths* to group variables. For example, in the figure, there are the vector variables `Simulation1/chassis.position.x` and `Simulation2/chassis.position.x`

With the use of this methodology, it is very easy to perform computations. For example, what the *quickstart example* does is:

1. Create a car by the name of `"car"`. This will add a variable of type `"vehicle"` into the internal memory with the name `"car"`
2. Create a circuit by the name of `"catalunya"`. This will add a variable of type `"track"` into the internal memory with the name `"catalunya"`
3. Run optimal laptime. The output of this function is a collection of timehistories from the simulation. These timehistories are new variables of type `vector` stored in the internal memory under the prefix `run/`, for example, `"run/chassis.position.x"`, `"run/chassis.position.y"`, `"run/chassis.velocity.x"`, ...
4. Download results. The vector and scalar tables can be emptied afterwards by calling

```
fastest_lap.clear_tables_by_prefix("run/")
```

1.4 Variable types

This page provides a reference on the types of variables that are currently supported in Fastest-lap

Variable type	Description	Create	Modify	Get
Scalar	Store one real number	*create_scalar()*		download_scalar()
Vector	Store multiple real numbers	*create_vector()*		download_vector()
Vehicle	Store one vehicle	*create_vehicle_from_xml()*	vehicle_set_parameter()	vehicle_save_as_xml()
		create_vehicle_empty()	vehicle_declare_new_constant_parameter()	
			vehicle_change_track()	
Track	Store one circuit	*create_track_from_xml()*		track_get_data()

1.5 Functions

This the reference page for all the functions included in Fastest-lap API. Both the C and Python API versions are provided.

1.5.1 Configuration

set_print_level

Adjusts the level of detail of the screen output. (Default: 2)

C API

```c
void set_print_level(int print_level);
```

Python API

```python
def set_print_level(print_level)
```

`print_level` ranges from 0 to 2: `print_level=0` supresses all outputs, and `print_level=2` is the fully detailed output.

1.5.2 Factories

create_scalar

Creates a scalar variable in the internal memory and assigns it an initial value

C API

```
void create_scalar(const char* variable_name, double variable_name);
```

Python API

```
def create_scalar(variable_name, variable_value)
```

create_vector

Creates a vector in the internal memory and assigns it an initial value. In C you need to specify how many elements it should contain, whereas in python it is automatically inferred.

C API

```
void create_vector(const char* variable_name, const int vector_size, double*␣
↪values);
```

Python API

```
def create_vector(variable_name, variable_values)
```

create_vehicle_from_xml

Creates a vehicle from an XML database file.

C API

```
void create_vehicle_from_xml(const char* vehicle_name, const char* database_
↪xml_file);
```

Python API

```
def create_vehicle_from_xml(vehicle_name, database_xml_file)
```

This database file is passed as the argument `database_xml_file`. Its path can be absolute or relative but if you use relative paths remember to make sure the program is run from the proper folder!

After the creation, the vehicle is stored in the internal memory with the name provided in `vehicle_name`.

create_vehicle_empty

Creates an **empty** vehicle of a given type.

C API

```
void create_vehicle_empty(const char* vehicle_name, const char* vehicle_
↪type);
```

Python API

```
def create_vehicle_empty(vehicle_name, vehicle_type)
```

All the car parameters (for example, the vehicle mass) will be defaulted to *0.0* and they must be later set using *vehicle_set_parameter()*.

The type of the car model is specified through `vehicle_type`.

Two types are currently supported: `"f1-3dof"` and `kart-6dof`.

create_track_from_xml

Creates a circuit from an XML file. This XML file contains the geometrical description of the track: the centerline, heading angle, curvature, and track limits.

C API

```
void create_track_from_xml(const char* track_name, const char* track_xml_
↪file);
```

Python API

```
def create_track_from_xml(track_name, track_xml_file);
```

Examples of track XML files can be found in the database folder

copy_variable

Creates a new instance of a given existing variable under a new name

C API

```
void copy_variable(const char* source_name, const char* destination_name);
```

Python API

```
def copy_variable(source_name, destination_name)
```

move_variable

Renames an existing to a new name

C API

```
void move_variable(const char* old_name, const char* new_name);
```

Python API

```
def move_variable(old_name, new_name)
```

1.5.3 Destructors

delete_variable

Deletes a variable with name `variable_name` from the internal memory.

C API

```
void delete_variable(const char* variable_name);
```

Python API

```
def delete_variable(variable_name)
```

`delete_variable` accepts regular expressions. For example one can delete all the variables under the prefix `run/` by using `delete_variable("run/*")`.

1.5.4 Modifiers

vehicle_set_parameter

Sets a parameter from the physical model of an existing vehicle.

C API

```
void vehicle_set_parameter(const char* vehicle_name, const char* parameter_
↪name, const double parameter_value);
```

Python API

```
def vehicle_set_parameter(vehicle_name, parameter_name, parameter_value)
```

`vehicle_name` is the name of the vehicle to be modified, `parameter_name` is the path to the selected parameter, and `parameter_value` its new given value.

For example, to set the mass of a vehicle to 795.0, one can use `vehicle_set_parameter(vehicle_name, "vehicle/chassis/mass", 795.0)`.

The full list of model parameters can be found *here*.

vehicle_declare_new_constant_parameter

Among all the physical parameters of a model, selects a parameter to perform its sensitivity analysis after the computation of an optimal laptime.

C API

```
void vehicle_declare_new_constant_parameter(const char* vehicle_name, const␣
↪char* parameter_name,
                                            const char* parameter_alias,␣
↪const double parameter_value);
```

Python API

```
def vehicle_declare_new_constant_parameter(vehicle_name, parameter_name,␣
↪parameter_alias, parameter_value)
```

`parameter_name` is the physical parameter that will be studied (e.g. `vehicle/chassis/mass`). Parameter alias is the name by which the sensitivity analysis will be found in the internal memory (e.g. one can simply call it `mass`), and `parameter_value` is a new value for the parameter.

vehicle_declare_new_variable_parameter

Among all the physical parameters of a model, selects a parameter to perform its sensitivity analysis after the computation of an optimal laptime.

As opposed to constant parameters, variable parameters are allowed to vary along the circuit.

C API

```
void vehicle_declare_new_variable_parameter(const char* vehicle_name, const␣
↪char* parameter_name,
                                            const char* parameter_aliases,␣
↪const int number_of_values,
                                            const double* parameter_values,␣
↪const int number_of_mesh_points,
                                            const int* mesh_parameter_
↪indexes, const double* mesh_points);
```

`parameter_name` is the physical parameter that will be studied (e.g. `vehicle/chassis/aerodynamics/cd`). Parameter aliases are the names by which the sensitivity analysis will be found in the internal memory, separated by a semicolon (e.g. `"cd1;cd2"`). `number_of_values` is the number of different values that the parameter can take, and `parameter_values` the values.

`number_of_mesh_points` is the number of spatial points in which value breakpoints are specified, `mesh_parameter_indexes` selects which value will be used for each breakpoint, and `mesh_points` is the mesh arclength breakpoints.

The value of the parameter in an arbitrary position is computed using a linear interpolation using these breakpoints.

For example, this can be used to define a DRS. We can define two values of the drag coefficient `cd_drs_on` and `cd_drs_off`. If we have a DRS zone from `s=100` to `s=700`, then the arguments are

- `parameter_name = "vehicle/chassis/aerodynamics/cd"`
- `parameter_aliases = "cd_drs_on;cd_drs_off"`
- `number_of_values = 2`
- `parameter_values = {cd_drs_on, cd_drs_off}`
- `number_of_mesh_points = 6`
- `mesh_parameter_indexes = {1, 1, 0, 0, 1, 1}`
- `mesh_points = {0.0, 100.0, 101.0, 700.0, 701.0, track_length}`

1.5.5 Getters

1.6 Models

Fastest-lap implements two car physical models:

1.6.1 F1 3 degrees of freedom model

Work in progress…

1.6.2 Go-kart 6 degrees of freedom model

Work in progress…

1.7 Modules

1.7.1 Circuit preprocessor

Work in progress…

1.7.2 GG-diagram

Work in progress…

1.7.3 Optimal laptime

Work in progress…

1.8 Defining and exploring variables

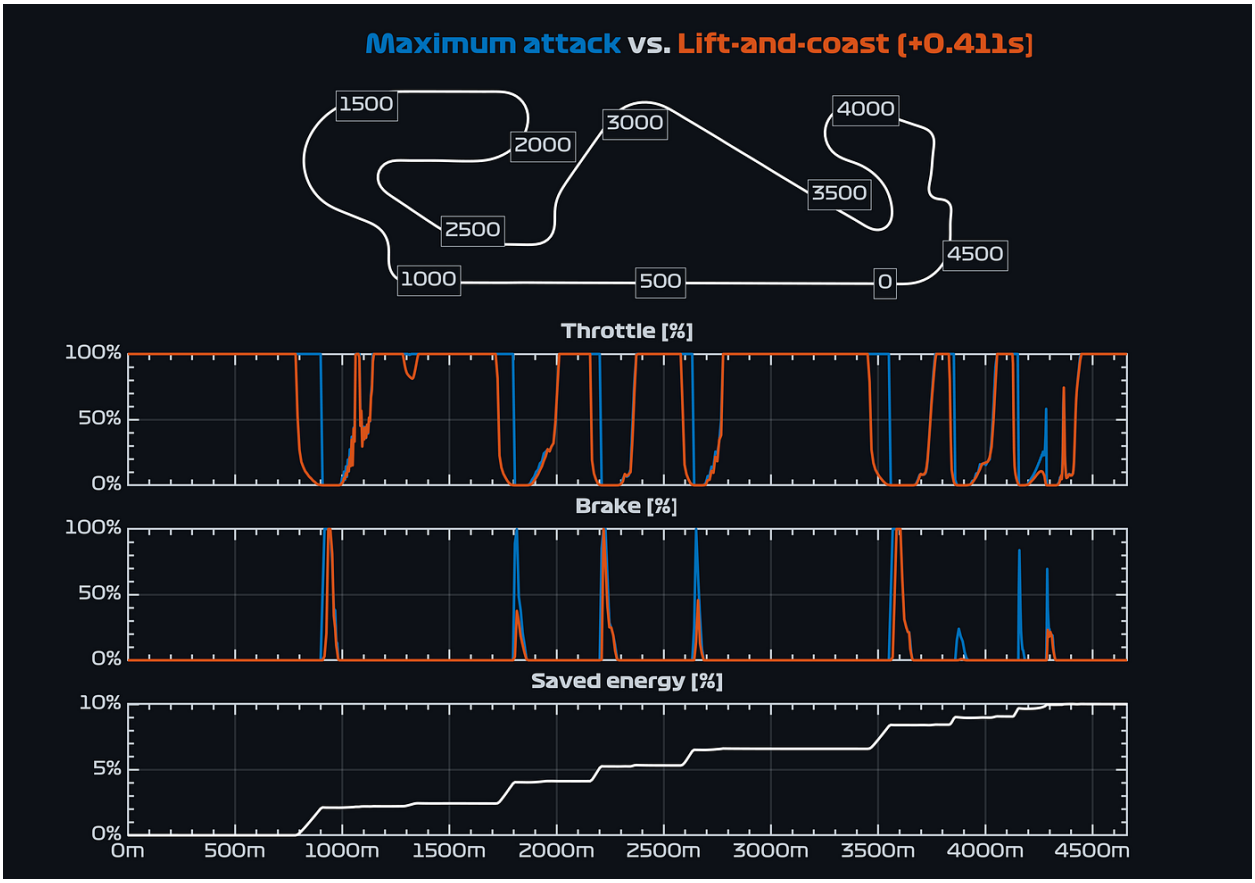
Work in progress…

1.9 How racing drivers save fuel efficiently: the lift-and-coast technique

In most motorsport disciplines, fuel management across the race distance is of paramount importance given the strong and restrictive regulations on the powertrains. A few years ago in Formula 1, when the hybrid V6s landed all immature, drivers used to spend a big part of their race saving fuel. We all remember Fernando Alonso's nightmarish team communications back in the McLaren-Honda days when he was asked to slow down "I don't want, I don't want. So I race and then I concentrate on the fuel".

The question is: how can drivers adapt their driving to save fuel? Probably advanced readers know that the most successful technique is called lift-and-coast. It's so called as drivers lift the throttle earlier before a heavy braking section, let the engine brake and aerodynamic drag take over for a few hundred meters, and then if needed mildly apply the brakes to coast into the corner. (Other fuel saving methods include short-shifting, left out of the scope of this post.)

Let's analyse in depth how this technique works. We show the throttle and brake used to save 10% fuel. The maximum attack (qualifying lap) throttle is shown in blue, whereas the lift-and-coast is colored in orange. The track Circuit de Catalunya, located in Spain, is used for this example.



We have three plots: throttle (check how the orange curve systematically backs off to zero some meters before the blue!), brake, and saved energy. All three are represented against the meters traveled by the car. The track map should help to relate distance and the corners in the track.

Overall, the driver loses +0.411s. This means, that to save fuel for an entire extra lap, the driver will lose +4.110s.

… continue reading in medium

1.10 Car and tire dynamics at the limits of handling (Part I)

Formula 1 cars can swift through corners at more than 5g, and after twenty years watching the sport, still fascinates me. Really. And it gets even more mind-boggling once you realise that this acceleration is entirely produced by the grip of tires. To push the tires into their operating window and extract their maximum performance is key in Formula 1. All the dynamics, the power of the engine, the energy of the brakes, ultimately is imprinted to the asphalt by the tires. So we need to talk about tires!

In this blog, you will read about how tires generate grip, about what parameters dictate the grip levels, how this grip can be modelled mathematically, how it is effectively controlled by the drivers, and how it affects the car dynamics. This blog is the first part out of two, which covers straight line acceleration and braking. Cornering and lateral forces will be covered in Part II.

There are many articles on internet about tire modelling, but what makes this one powerful, is that through simulations you will see them in action fitted in cars, which gives you a broader insight on how they really work.

… continue reading in medium

1.11 Car and tire dynamics at the limits of handling (Part II)

Right. Part II, here. we. go.

This blog is the continuation of Car and tire dynamics at the limits of handling (Part I), where we laid the foundation of tire grip generation and its relation to car acceleration in straight line. As surprising as it sounds, cars normally also need to take corners, which is the raison d'être for this second blog in the series.

Knowledge of how tires behave at the limits of handling is fundamental to grasp advanced concepts such as understeer and oversteer. Usually, these are presented as 'when the car does not want to turn' or 'when the car turns more than what the driver commands to'. Do you want to level up from these definitions and master the underlying science? then my friend, you will enjoy the rest of this blog.

First and foremost, you need to understand that cornering also needs acceleration. Acceleration is the rate of change in the velocity, and when a car corners, it changes its velocity, precisely, its direction. In particular, cars need lateral acceleration to corner, and it is no surprise, that this acceleration is again produced by the grip of the tires.

… continue reading in medium

1.12 Can 2022 F1 cars tame 130R with DRS open? — Suzuka tech bits

Formula 1 arrived at Suzuka, Japan. Its 5.8km length circuit comprises all corner types from slow ones like the Hairpin to thrilling high speed turns like the 130R. And about 130R, one of the talking points prior to the grand prix was whether the new cars could take 130R flat out with the DRS open. Mainly because due to the more stable rear part of the 2022 cars, the influence of the DRS in switching the balance frontward is more mitigated than in previous years.

In this blog, we look at the possibility of taking 130R with the DRS open with the new regulation F1 cars, and we will do so by using laptime simulation [4].

To start, we find a setup of the car that fits real telemetry data. For this example, Fernando Alonso's best qualifying lap in the 2022 event was selected.

With this simulation, we take a deeper look into the stress on the tires. In the diagram below, we see that the rear part of the car is more loaded than the front (the vertical 'energy' bars at the right of each tire), being the right tires the one putting the effort in this left handed corner.

ince the rear part of the car is more loaded, the front tires need more slippage to turn, hence, the car understeers into the corner. We see that the car can take this corner flat out without any issues since the tires are approximately at their 50% of the maximum grip. The velocity at which the cars hit the apex is 303km/h.

Now, we open the DRS in the backstraight…

… continue reading in medium

1.13 Formula 1 cars or sailing ships?– United States GP Tech bits

During the 2022 United States Formula 1 Grand Prix, the presence of heavy gusts of wind was one of the talking points (besides the superhuman race from Alonso!), altering the balance of the cars and even causing Valtteri Bottas to lose control of his car.

Wind gusts are specially dangerous since they concentrate on a specific location, and their strength doubles or triples the mean wind in the region. In the case of Austin, the average wind was 20km/h, but drivers could encounter sudden gusts of 50km/h in a particular section of the circuit.

Having seen this, I run a poll in Twitter after the race. The question what simple: there's a 50km/h wind gust at Turn 11. In what direction would it be more favorable for a driver? After more than 2000 votes, head-wind in the braking point was the clear winner (south-west direction).

Is it really the best direction performance-wise? Well, the answer will surprise you… unless you got it right :D

… continue reading in medium