FAST-OAD

Release unknown

unknown

Mar 21, 2024

CONTENTS

For a **quick overview** of the way FAST-OAD works, please go *here*.

Installation instructions are *here*.

For a detailed description of the **input files and the Command Line Interface**, check out the *usage section*.

If you prefer to discover the **Application Programming Interface with Python notebooks**, you may go directly to the section *Using FAST-OAD through Python*.

For a description of **models used in FAST-OAD**, you may see the *model documentations*.

If you want to **add your own models**, please check out *How to add custom OpenMDAO modules to FAST-OAD*.

Note: Since version 1.3, FAST-OAD has its core features in FAST-OAD-core package, while the legacy models are in FAST-OAD-CS25 package.

Yet, installing FAST-OAD 1.x will keep installing both FAST-OAD-core and FAST-OAD-CS25.

Models in FAST-OAD-CS25 are still a work in progress.

CHAPTER

ONE

CONTENTS

1.1 License

GNU GENERAL PUBLIC LICENSE Version 3, 29 June 2007

Copyright (C) 2007 Free Software Foundation, Inc. <http://fsf.org/> Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

Preamble

The GNU General Public License is a free, copyleft license for

software and other kinds of works.

The licenses for most software and other practical works are designed

to take away your freedom to share and change the works. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change all versions of a program–to make sure it remains free software for all its users. We, the Free Software Foundation, use the GNU General Public License for most of our software; it applies also to any other work released this way by its authors. You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not

price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for them if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs, and that you know you can do these things.

To protect your rights, we need to prevent others from denying you

these rights or asking you to surrender the rights. Therefore, you have certain responsibilities if you distribute copies of the software, or if you modify it: responsibilities to respect the freedom of others.

For example, if you distribute copies of such a program, whether

gratis or for a fee, you must pass on to the recipients the same freedoms that you received. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

Developers that use the GNU GPL protect your rights with two steps:

(1) assert copyright on the software, and (2) offer you this License giving you legal permission to copy, distribute and/or modify it.

For the developers' and authors' protection, the GPL clearly explains

that there is no warranty for this free software. For both users' and authors' sake, the GPL requires that modified versions be marked as changed, so that their problems will not be attributed erroneously to authors of previous versions.

Some devices are designed to deny users access to install or run

modified versions of the software inside them, although the manufacturer can do so. This is fundamentally incompatible with the aim of protecting users' freedom to change the software. The systematic pattern of such abuse occurs in the area of products for individuals to use, which is precisely where it is most unacceptable. Therefore, we have designed this version of the GPL to prohibit the practice for those products. If such problems arise substantially in other domains, we stand ready to extend this provision to those domains in future versions of the GPL, as needed to protect the freedom of users.

Finally, every program is threatened constantly by software patents.

States should not allow patents to restrict development and use of software on general-purpose computers, but in those that do, we wish to avoid the special danger that patents applied to a free program could make it effectively proprietary. To prevent this, the GPL assures that patents cannot be used to render the program non-free.

The precise terms and conditions for copying, distribution and

modification follow.

TERMS AND CONDITIONS

0. Definitions.

"This License" refers to version 3 of the GNU General Public License.

"Copyright" also means copyright-like laws that apply to other kinds of

works, such as semiconductor masks.

"The Program" refers to any copyrightable work licensed under this

License. Each licensee is addressed as "you". "Licensees" and "recipients" may be individuals or organizations.

To "modify" a work means to copy from or adapt all or part of the work

in a fashion requiring copyright permission, other than the making of an exact copy. The resulting work is called a "modified version" of the earlier work or a work "based on" the earlier work.

A "covered work" means either the unmodified Program or a work based

on the Program.

To "propagate" a work means to do anything with it that, without

permission, would make you directly or secondarily liable for infringement under applicable copyright law, except executing it on a computer or modifying a private copy. Propagation includes copying, distribution (with or without modification), making available to the public, and in some countries other activities as well.

To "convey" a work means any kind of propagation that enables other

parties to make or receive copies. Mere interaction with a user through a computer network, with no transfer of a copy, is not conveying.

An interactive user interface displays "Appropriate Legal Notices"

to the extent that it includes a convenient and prominently visible feature that (1) displays an appropriate copyright notice, and (2) tells the user that there is no warranty for the work (except to the extent that warranties are provided), that licensees may convey the work under this License, and how to view a copy of this License. If the interface presents a list of user commands or options, such as a menu, a prominent item in the list meets this criterion.

1. Source Code.

The "source code" for a work means the preferred form of the work

for making modifications to it. "Object code" means any non-source form of a work.

A "Standard Interface" means an interface that either is an official

standard defined by a recognized standards body, or, in the case of interfaces specified for a particular programming language, one that is widely used among developers working in that language.

The "System Libraries" of an executable work include anything, other

than the work as a whole, that (a) is included in the normal form of packaging a Major Component, but which is not part of that Major Component, and (b) serves only to enable use of the work with that Major Component, or to implement a Standard Interface for which an implementation is available to the public in source code form. A "Major Component", in this context, means a major essential component (kernel, window system, and so on) of the specific operating system (if any) on which the executable work runs, or a compiler used to produce the work, or an object code interpreter used to run it.

The "Corresponding Source" for a work in object code form means all

the source code needed to generate, install, and (for an executable work) run the object code and to modify the work, including scripts to control those activities. However, it does not include the work's System Libraries, or general-purpose tools or generally available free programs which are used unmodified in performing those activities but which are not part of the work. For example, Corresponding Source includes interface definition files associated with source files for the work, and the source code for shared libraries and dynamically linked subprograms that the work is specifically designed to require, such as by intimate data communication or control flow between those subprograms and other parts of the work.

The Corresponding Source need not include anything that users

can regenerate automatically from other parts of the Corresponding Source.

The Corresponding Source for a work in source code form is that

same work.

2. Basic Permissions.

All rights granted under this License are granted for the term of

copyright on the Program, and are irrevocable provided the stated conditions are met. This License explicitly affirms your unlimited permission to run the unmodified Program. The output from running a covered work is covered by this License only if the output, given its content, constitutes a covered work. This License acknowledges your rights of fair use or other equivalent, as provided by copyright law.

You may make, run and propagate covered works that you do not

convey, without conditions so long as your license otherwise remains in force. You may convey covered works to others for the sole purpose of having them make modifications exclusively for you, or provide you with facilities for running those works, provided that you comply with the terms of this License in conveying all material for which you do not control copyright. Those thus making or running the covered works for you must do so exclusively on your behalf, under your direction and control, on terms that prohibit them from making any copies of your copyrighted material outside their relationship with you.

Conveying under any other circumstances is permitted solely under

the conditions stated below. Sublicensing is not allowed; section 10 makes it unnecessary.

3. Protecting Users' Legal Rights From Anti-Circumvention Law.

No covered work shall be deemed part of an effective technological

measure under any applicable law fulfilling obligations under article 11 of the WIPO copyright treaty adopted on 20 December 1996, or similar laws prohibiting or restricting circumvention of such measures.

When you convey a covered work, you waive any legal power to forbid

circumvention of technological measures to the extent such circumvention is effected by exercising rights under this License with respect to the covered work, and you disclaim any intention to limit operation or modification of the

work as a means of enforcing, against the work's users, your or third parties' legal rights to forbid circumvention of technological measures.

4. Conveying Verbatim Copies.

You may convey verbatim copies of the Program's source code as you

receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice; keep intact all notices stating that this License and any non-permissive terms added in accord with section 7 apply to the code; keep intact all notices of the absence of any warranty; and give all recipients a copy of this License along with the Program.

You may charge any price or no price for each copy that you convey,

and you may offer support or warranty protection for a fee.

5. Conveying Modified Source Versions.

You may convey a work based on the Program, or the modifications to

produce it from the Program, in the form of source code under the terms of section 4, provided that you also meet all of these conditions:

> a) The work must carry prominent notices stating that you modified it, and giving a relevant date.
>
> b) The work must carry prominent notices stating that it is released under this License and any conditions added under section 7. This requirement modifies the requirement in section 4 to "keep intact all notices".
>
> c) You must license the entire work, as a whole, under this License to anyone who comes into possession of a copy. This License will therefore apply, along with any applicable section 7 additional terms, to the whole of the work, and all its parts, regardless of how they are packaged. This License gives no permission to license the work in any other way, but it does not invalidate such permission if you have separately received it.
>
> d) If the work has interactive user interfaces, each must display Appropriate Legal Notices; however, if the Program has interactive interfaces that do not display Appropriate Legal Notices, your work need not make them do so.

A compilation of a covered work with other separate and independent

works, which are not by their nature extensions of the covered work, and which are not combined with it such as to form a larger program, in or on a volume of a storage or distribution medium, is called an "aggregate" if the compilation and its resulting copyright are not used to limit the access or legal rights of the compilation's users beyond what the individual works permit. Inclusion of a covered work in an aggregate does not cause this License to apply to the other parts of the aggregate.

6. Conveying Non-Source Forms.

You may convey a covered work in object code form under the terms

of sections 4 and 5, provided that you also convey the machine-readable Corresponding Source under the terms of this License, in one of these ways:

> a) Convey the object code in, or embodied in, a physical product (including a physical distribution medium), accompanied by the Corresponding Source fixed on a durable physical medium customarily used for software interchange.
>
> b) Convey the object code in, or embodied in, a physical product (including a physical distribution medium), accompanied by a written offer, valid for at least three years and valid for as long as you offer spare parts or customer support for that product model, to give anyone who possesses the object code either (1) a copy of the Corresponding Source for all the software in

the product that is covered by this License, on a durable physical medium customarily used for software interchange, for a price no more than your reasonable cost of physically performing this conveying of source, or (2) access to copy the Corresponding Source from a network server at no charge.

c) Convey individual copies of the object code with a copy of the written offer to provide the Corresponding Source. This alternative is allowed only occasionally and noncommercially, and only if you received the object code with such an offer, in accord with subsection 6b.

d) Convey the object code by offering access from a designated place (gratis or for a charge), and offer equivalent access to the Corresponding Source in the same way through the same place at no further charge. You need not require recipients to copy the Corresponding Source along with the object code. If the place to copy the object code is a network server, the Corresponding Source may be on a different server (operated by you or a third party) that supports equivalent copying facilities, provided you maintain clear directions next to the object code saying where to find the Corresponding Source. Regardless of what server hosts the Corresponding Source, you remain obligated to ensure that it is available for as long as needed to satisfy these requirements.

e) Convey the object code using peer-to-peer transmission, provided you inform other peers where the object code and Corresponding Source of the work are being offered to the general public at no charge under subsection 6d.

A separable portion of the object code, whose source code is excluded

from the Corresponding Source as a System Library, need not be included in conveying the object code work.

A "User Product" is either (1) a "consumer product", which means any

tangible personal property which is normally used for personal, family, or household purposes, or (2) anything designed or sold for incorporation into a dwelling. In determining whether a product is a consumer product, doubtful cases shall be resolved in favor of coverage. For a particular product received by a particular user, "normally used" refers to a typical or common use of that class of product, regardless of the status of the particular user or of the way in which the particular user actually uses, or expects or is expected to use, the product. A product is a consumer product regardless of whether the product has substantial commercial, industrial or non-consumer uses, unless such uses represent the only significant mode of use of the product.

"Installation Information" for a User Product means any methods,

procedures, authorization keys, or other information required to install and execute modified versions of a covered work in that User Product from a modified version of its Corresponding Source. The information must suffice to ensure that the continued functioning of the modified object code is in no case prevented or interfered with solely because modification has been made.

If you convey an object code work under this section in, or with, or

specifically for use in, a User Product, and the conveying occurs as part of a transaction in which the right of possession and use of the User Product is transferred to the recipient in perpetuity or for a fixed term (regardless of how the transaction is characterized), the Corresponding Source conveyed under this section must be accompanied by the Installation Information. But this requirement does not apply if neither you nor any third party retains the ability to install modified object code on the User Product (for example, the work has been installed in ROM).

The requirement to provide Installation Information does not include a

requirement to continue to provide support service, warranty, or updates for a work that has been modified or installed by the recipient, or for the User Product in which it has been modified or installed. Access to a network may be denied when the modification itself materially and adversely affects the operation of the network or violates the rules and protocols for communication across the network.

Corresponding Source conveyed, and Installation Information provided,

in accord with this section must be in a format that is publicly documented (and with an implementation available to the public in source code form), and must require no special password or key for unpacking, reading or copying.

7. Additional Terms.

"Additional permissions" are terms that supplement the terms of this

License by making exceptions from one or more of its conditions. Additional permissions that are applicable to the entire Program shall be treated as though they were included in this License, to the extent that they are valid under applicable law. If additional permissions apply only to part of the Program, that part may be used separately under those permissions, but the entire Program remains governed by this License without regard to the additional permissions.

When you convey a copy of a covered work, you may at your option

remove any additional permissions from that copy, or from any part of it. (Additional permissions may be written to require their own removal in certain cases when you modify the work.) You may place additional permissions on material, added by you to a covered work, for which you have or can give appropriate copyright permission.

Notwithstanding any other provision of this License, for material you

add to a covered work, you may (if authorized by the copyright holders of that material) supplement the terms of this License with terms:

> a) Disclaiming warranty or limiting liability differently from the terms of sections 15 and 16 of this License; or
>
> b) Requiring preservation of specified reasonable legal notices or author attributions in that material or in the Appropriate Legal Notices displayed by works containing it; or
>
> c) Prohibiting misrepresentation of the origin of that material, or requiring that modified versions of such material be marked in reasonable ways as different from the original version; or
>
> d) Limiting the use for publicity purposes of names of licensors or authors of the material; or
>
> e) Declining to grant rights under trademark law for use of some trade names, trademarks, or service marks; or
>
> f) Requiring indemnification of licensors and authors of that material by anyone who conveys the material (or modified versions of it) with contractual assumptions of liability to the recipient, for any liability that these contractual assumptions directly impose on those licensors and authors.

All other non-permissive additional terms are considered "further

restrictions" within the meaning of section 10. If the Program as you received it, or any part of it, contains a notice stating that it is governed by this License along with a term that is a further restriction, you may remove that term. If a license document contains a further restriction but permits relicensing or conveying under this License, you may add to a covered work material governed by the terms of that license document, provided that the further restriction does not survive such relicensing or conveying.

If you add terms to a covered work in accord with this section, you

must place, in the relevant source files, a statement of the additional terms that apply to those files, or a notice indicating where to find the applicable terms.

Additional terms, permissive or non-permissive, may be stated in the

form of a separately written license, or stated as exceptions; the above requirements apply either way.

8. Termination.

You may not propagate or modify a covered work except as expressly

provided under this License. Any attempt otherwise to propagate or modify it is void, and will automatically terminate your rights under this License (including any patent licenses granted under the third paragraph of section 11).

However, if you cease all violation of this License, then your

license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

Moreover, your license from a particular copyright holder is

reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

Termination of your rights under this section does not terminate the

licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, you do not qualify to receive new licenses for the same material under section 10.

9. Acceptance Not Required for Having Copies.

You are not required to accept this License in order to receive or

run a copy of the Program. Ancillary propagation of a covered work occurring solely as a consequence of using peer-to-peer transmission to receive a copy likewise does not require acceptance. However, nothing other than this License grants you permission to propagate or modify any covered work. These actions infringe copyright if you do not accept this License. Therefore, by modifying or propagating a covered work, you indicate your acceptance of this License to do so.

10. Automatic Licensing of Downstream Recipients.

Each time you convey a covered work, the recipient automatically

receives a license from the original licensors, to run, modify and propagate that work, subject to this License. You are not responsible for enforcing compliance by third parties with this License.

An “entity transaction” is a transaction transferring control of an

organization, or substantially all assets of one, or subdividing an organization, or merging organizations. If propagation of a covered work results from an entity transaction, each party to that transaction who receives a copy of the work also receives whatever licenses to the work the party’s predecessor in interest had or could give under the previous paragraph, plus a right to possession of the Corresponding Source of the work from the predecessor in interest, if the predecessor has it or can get it with reasonable efforts.

You may not impose any further restrictions on the exercise of the

rights granted or affirmed under this License. For example, you may not impose a license fee, royalty, or other charge for exercise of rights granted under this License, and you may not initiate litigation (including a cross-claim or counterclaim in a lawsuit) alleging that any patent claim is infringed by making, using, selling, offering for sale, or importing the Program or any portion of it.

11. Patents.

A “contributor” is a copyright holder who authorizes use under this

License of the Program or a work on which the Program is based. The work thus licensed is called the contributor’s “contributor version”.

A contributor’s “essential patent claims” are all patent claims

owned or controlled by the contributor, whether already acquired or hereafter acquired, that would be infringed by some manner, permitted by this License, of making, using, or selling its contributor version, but do not include claims that would be infringed only as a consequence of further modification of the contributor version. For purposes of this definition, “control” includes the right to grant patent sublicenses in a manner consistent with the requirements of this License.

Each contributor grants you a non-exclusive, worldwide, royalty-free

patent license under the contributor's essential patent claims, to make, use, sell, offer for sale, import and otherwise run, modify and propagate the contents of its contributor version.

In the following three paragraphs, a "patent license" is any express

agreement or commitment, however denominated, not to enforce a patent (such as an express permission to practice a patent or covenant not to sue for patent infringement). To "grant" such a patent license to a party means to make such an agreement or commitment not to enforce a patent against the party.

If you convey a covered work, knowingly relying on a patent license,

and the Corresponding Source of the work is not available for anyone to copy, free of charge and under the terms of this License, through a publicly available network server or other readily accessible means, then you must either (1) cause the Corresponding Source to be so available, or (2) arrange to deprive yourself of the benefit of the patent license for this particular work, or (3) arrange, in a manner consistent with the requirements of this License, to extend the patent license to downstream recipients. "Knowingly relying" means you have actual knowledge that, but for the patent license, your conveying the covered work in a country, or your recipient's use of the covered work in a country, would infringe one or more identifiable patents in that country that you have reason to believe are valid.

If, pursuant to or in connection with a single transaction or

arrangement, you convey, or propagate by procuring conveyance of, a covered work, and grant a patent license to some of the parties receiving the covered work authorizing them to use, propagate, modify or convey a specific copy of the covered work, then the patent license you grant is automatically extended to all recipients of the covered work and works based on it.

A patent license is "discriminatory" if it does not include within

the scope of its coverage, prohibits the exercise of, or is conditioned on the non-exercise of one or more of the rights that are specifically granted under this License. You may not convey a covered work if you are a party to an arrangement with a third party that is in the business of distributing software, under which you make payment to the third party based on the extent of your activity of conveying the work, and under which the third party grants, to any of the parties who would receive the covered work from you, a discriminatory patent license (a) in connection with copies of the covered work conveyed by you (or copies made from those copies), or (b) primarily for and in connection with specific products or compilations that contain the covered work, unless you entered into that arrangement, or that patent license was granted, prior to 28 March 2007.

Nothing in this License shall be construed as excluding or limiting

any implied license or other defenses to infringement that may otherwise be available to you under applicable patent law.

12. No Surrender of Others' Freedom.

If conditions are imposed on you (whether by court order, agreement or

otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot convey a covered work so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not convey it at all. For example, if you agree to terms that obligate you to collect a royalty for further conveying from those to whom you convey the Program, the only way you could satisfy both those terms and this License would be to refrain entirely from conveying the Program.

13. Use with the GNU Affero General Public License.

Notwithstanding any other provision of this License, you have

permission to link or combine any covered work with a work licensed under version 3 of the GNU Affero General Public License into a single combined work, and to convey the resulting work. The terms of this License will continue to apply to the part which is the covered work, but the special requirements of the GNU Affero General Public License, section 13, concerning interaction through a network will apply to the combination as such.

14. Revised Versions of this License.

The Free Software Foundation may publish revised and/or new versions of

the GNU General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the

Program specifies that a certain numbered version of the GNU General Public License "or any later version" applies to it, you have the option of following the terms and conditions either of that numbered version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of the GNU General Public License, you may choose any version ever published by the Free Software Foundation.

If the Program specifies that a proxy can decide which future

versions of the GNU General Public License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Program.

Later license versions may give you additional or different

permissions. However, no additional obligations are imposed on any author or copyright holder as a result of your choosing to follow a later version.

15. Disclaimer of Warranty.

THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY

APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

16. Limitation of Liability.

IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING

WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MODIFIES AND/OR CONVEYS THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

17. Interpretation of Sections 15 and 16.

If the disclaimer of warranty and limitation of liability provided

above cannot be given local legal effect according to their terms, reviewing courts shall apply local law that most closely approximates an absolute waiver of all civil liability in connection with the Program, unless a warranty or assumption of liability accompanies a copy of the Program in return for a fee.

END OF TERMS AND CONDITIONS

1.2 Contributors

- Christophe DAVID <christophe.david@onera.fr>
- Scott DELBECQ <scott.delbecq@isae-supaero.fr>
- Martin DELAVENNE <martin.delavenne@isae-supaero.fr>

1.3 How to cite us

Please cite this article when using FAST-OAD in your research works:

C. David, S. Delbecq, S. Defoort, P. Schmollgruber, E. Benard and V. Pommier-Budinger: *"From FAST to FAST-OAD: An open source framework for rapid Overall Aircraft Design"*, IOP Conference Series: Materials Science and Engineering, vol. 1024, n. 1, DOI: 10.1088/1757-899x/1024/1/012062

```
@article{David2021,
    doi = {10.1088/1757-899x/1024/1/012062},
    url = {https://doi.org/10.1088/1757-899x/1024/1/012062},
    year = 2021,
    month = {jan},
    publisher = {{IOP} Publishing},
    volume = {1024},
    number = {1},
    pages = {012062},
    author = {Christophe David and Scott Delbecq and Sebastien Defoort and Peter␣
↪Schmollgruber and Emmanuel Benard and Valerie Pommier-Budinger},
    title = {From {FAST} to {FAST}-{OAD}: An open source framework for rapid Overall␣
↪Aircraft Design},
    journal = {{IOP} Conference Series: Materials Science and Engineering}}
```

1.4 Changelog

1.4.1 Version 1.7.0

- **Added:**
  - Centralized way to set options from configuration file. (#510)
- **Fixed:**
  - Fix for validity domain checker. (#511)

1.4.2 Version 1.6.0

- **Added:**
  - FAST-OAD is now officially compatible with Python 3.10. Support of Python 3.7 has been abandoned. (#496)
  - OpenMDAO group options can now be set from configuration file. (#502)
  - **Mission computation:**
    * A value for maximum lift coefficient can now be set for climb and cruise segments. (#504)
    * Added the field consumed_fuel, computed for each time step and present in CSV output file. (#505)
- **Fixed:**
  - Decreased execution time by avoiding unnecessary setup operations. (#503)

1.4.3 Version 1.5.2

- **Added:**
  - Added sphinx documentation for source data file generation. (#500)
- **Fixed:**
  - Fix for climb segment going far too high when asked for optimal altitude in some cases. (#497 and #498)
  - Now accepting upper case distribution names for FAST-OAD plugins. (#499)
  - Now DataFile.from_problem() returns a DataFile instance, and not a VariableList instance. (#494)

1.4.4 Version 1.5.1

- **Fixed:**
  - Some warning were issued by pandas when using mission module. (#492)

1.4.5 Version 1.5.0

- **Added:**
  - Computation of payload-range data. (#471 and #482)
  - Payload-range plot. (#480)
  - Time-step simulation of takeoff in mission module (#481, #484, #487, #490)
  - Introduced concept of macro-segment, for proposing assembly of several segments as one usable segment. (#488)
  - Segment implementations can now be registered using decorators. (#485)
  - Mission definition can now define a global target fuel consumption. (#467)
  - A FAST-OAD plugin can now come with its own source data files, obtainable using *fastoad gen_source_data_file* command. (#477)
- **Changed:**

  - fast-oad (not fast-oad-core) now requires at least fast-oad-cs25 0.1.4. (#475)
  - fast-oad (and fast-oad-core) now requires at least OpenMDAO 3.18. (#483)
  - Variable viewer can now display discrete outputs of type string. (#479)
- **Fixed:**
  - MissionViewer was not able to show several missions. (#477)
  - Fixed compatibility with OpenMDAO 3.26 (#486)

1.4.6 Version 1.4.2

- **Fixed:**
  - Fixed compatibility with Openmdao 3.22. (#464)
  - Now a warning is issued when a nan value is in generated input file from a given data source. (#468)
  - Now FAST-OAD_CS25 0.1.4 is explicitly required. (#475)

1.4.7 Version 1.4.1

- **Fixed:**
  - Fixed backward compatibility of bundled missions. (#466)

1.4.8 Version 1.4.0

- **Changed:**
  - Added a new series of tutorials. (#426)
  - **Enhancements in mission module (#430 and #462), mainly:**
    * a parameter with a variable as value can now be associated to a unit and a default value that will be used in the OpenMDAO input declaration (and be in generated input data file).
    * a target parameter can be declared as relative to the start point of the segment by prefixing the parameter name with "**delta_**" when setting a parameter, a minus sign can be put before a variable name to get the opposite value (can be useful with relative values)
    * a parameter can now be set at route or mission level.
    * dISA can now be set in mission definition file with isa_offset.
    * a mission phase can now contain other phases.
    * if a segment parameter (dataclass field) is an array or a list, the associated variable in mission file will be declared with shape_by_conn=True.
    * **taxi-out and takeoff are no more automatically set outside of the mission definition file:**
      · mission starting point (altitude, speed, mass) can now be set using the "start" segment.
      · the mass input of the mission can be set using the "mass_input" segment. This segment can be anywhere in the mission, though it is expected that fuel consumption in previous segments is mass-independent.

        - if none of the two above solution is used to define a mass input variable, the mission module falls back to behaviour of earlier releases, i.e. the automatic addition of taxi-out and takeoff at beginning of the mission.

  - Upgrade to wop 2.x API. (#453)

- **Fixed:**
  - Variable viewer was showing only one variable at a time if variable names contained no colon. (#456)
  - Optimization viewer was handling incorrectly bounds with value 0. (#461)

1.4.9 Version 1.3.5

- **Fixed:**
  - Deactivated automatic reports from OpenMDAO 3.17+ (can still be driven by environment variable OPENMDAO_REPORTS). (#449)
  - Mass breakdown bar plot now accepts more than 5 datasets. The used color map is now consistent with othe FAST-OAD plots. (#451)

1.4.10 Version 1.3.4

- **Fixed:**
  - FAST-OAD was quickly crashing in multiprocessing environment. (#442)
  - Memory consumption could increase considerably when numerous computations were done in the same Python session. (#443)
  - Deactivated sub-models kept being deactivated in following computations done in the same Python session. (#444)

1.4.11 Version 1.3.3

- **Fixed:**
  - Fixed crash when using Newton solver or case recorders. (#434)
  - **DataFile class enhancement (#435) :**
    - Instantiating DataFile with an non-existent file now triggers an error.
    - DataClass.from_*() methods now return a DataClass instance instead of VariableList.
    - A dedicated section has been added in Sphinx documentation (General Documentation > Process variables > Serialization > FAST-OAD API).
  - A component input could be in FAST-OAD-generated input file though it was explicitly connected to an IndepVarComp output in configuration file. (#437)

1.4.12 Version 1.3.2

- **Fixed:**
  - Compatibility with OpenMDAO 3.17.0. (#428)

1.4.13 Version 1.3.1

- **Fixed:**
  - Version requirements for StdAtm and FAST-OAD-CS25 were unwillingly pinned to 0.1.x. (#422)
  - *fastoad -v* was producing *unknown* when only FAST-OAD-core was installed. (#422)
  - Fixed some deprecation warnings. (#423)

1.4.14 Version 1.3.0.post0

- Modified package organization. (#420)

1.4.15 Version 1.3.0

- **Changes:**
  - **Rework of plugin system. (#409 - #417)**
    - Plugin group identifier is now *fastoad.plugins* (usage of *fastoad_model* is deprecated)
    - A plugin can now provide, besides models, notebooks and sample configuration files.
    - CLI and API have been updated to allow choosing the source when generating a configuration file, and to provide the needed information about installed plugin (*fastoad plugin_info*)
    - Models are loaded only when needed (speeds up some basic operations like *fastoad -h*)
  - CS25-related models are now in separate package [FAST-OAD-CS25](https://pypi.org/project/fast-oad-cs25/). This package is still installed along with FAST-OAD to preserve backward-compatibility. Also, package [FAST-OAD-core](https://pypi.org/project/fast-oad-core/) is now available, which does NOT install FAST-OAD-CS25 (thus contains only the mission model). (#414)
  - IndepVarComp variables in FAST-OAD models are now correctly handled and included in input data file. (#408)
  - Changes in mission module. Most noticeable change is that the number of engines is no more an input of the mission module, but should be handled by the propulsion model. No impact when using the base CS-25 process, since the variable name has not changed.(#411)
- **Bug fixes:**
  - FAST-OAD is now able to manage dynamically shaped problem inputs. (#416 - #418)

1.4.16 Version 1.2.1

- Changes: - Updated dependency requirements. All used libraries are now compatible with Jupyter lab 3 without need for building extensions. (#392) - Now Atmosphere class is part of the [stdatm](https://pypi.org/project/stdatm/) package (#398) - For *list_variables* command, the output format can now be chosen, with the addition of the format of variables_description.txt (for custom modules now generate a variable descriptions. (#399)
- Bug fixes: - Minor fixes in Atmosphere class. (#386)

1.4.17 Version 1.1.2

- **Bug fixes:**
  - Engine setting could be ignored for cruise segments. (#397)

1.4.18 Version 1.1.1

- **Bug fixes:**
  - Fixed usage of list_modules with CLI. (#395)

1.4.19 Version 1.1.0

- **Changes:**
  - Added new submodel feature to enable a more modular approach. (#379)
  - Implemented the submodel feature in the aerodynamic module. (#388)
  - Implemented the submodel feature in the geometry module. (#387)
  - Implemented the submodel feature in the weight module. (#385)
  - Added the possibility to list custom modules. (#369)
  - Updated high lift aerodynamics and rubber engine models. (#352)
  - Added custom modules tutorial notebook. (#317)
- **Bug fixes:**
  - Fixed incompatible versions of jupyter-client. (#390)
  - Fixed the naming and description of the virtual taper ratio used in the wing geometry. (#383)
  - Fixed some wrong file links and typos in CeRAS notebook. (#380)
  - Fixed issues with variable descriptions in xml file. (#364)

1.4.20 Version 1.0.5

- **Changes:**
  - Now using the new WhatsOpt feature that allows to generate XDSM files without being registered on server. (#361)
  - Optimization viewer does no allow anymore to modify output values. (#372)
- **Bug fixes:**
  - Compatibility with OpenMDAO 3.10 (which becomes the minimal required version). (#375)
  - Variable descriptions can now be read from comment of XML data files, which fixes the missing descriptions in variable viewer. (#359)
  - Performance model: the computed taxi-in distance was irrelevant. (#368)

1.4.21 Version 1.0.4

- **Changes:**
  - Enum classes in FAST-OAD models are now extensible by using *aenum* instead of *enum*. (#345)
- **Bug fixes:**
  - Incompatibility with *ruamel.yaml* 0.17.5 and above has been fixed. (#344)
  - Computation of partial derivatives for OpenMDAO was incorrectly declared in some components. MDA, or MDO with COBYLA solver, were not affected. (#347)
  - Errors in custom modules are no more hidden. (#348)

1.4.22 Version 1.0.3

- **Changes:**
  - Configuration files can now contain unknown sections (at root level) to allow these files to be used by other tools. (#333)
- **Bug fixes:**
  - Importing, in a *__init__.py*, some classes that were registered as FAST-OAD modules could make that the register process fails. (#331)
  - When generating an input file using a data source, the whole data source was copied instead of just keeping the needed variables. (#332)
  - Instead of overwriting an existing input files, variables of previous file were kept. (#330)
  - A variable that was connected to an output could be incorrectly labelled as input when listing problem variables. (#341)
  - Fixed broken links in Sphinx documentation, including docstrings. (#315)

1.4.23 Version 1.0.2

- FAST-OAD now requires a lower version of *ruamel.yaml*. It should prevent Anaconda to try and fail to update its "clone" of *ruamel.yaml*. (#308)

1.4.24 Version 1.0.1

- **Bug fixes:**
  - In a jupyter notebook, each use of a filter in variable viewer caused the display of a new variable viewer. (#301)
  - Wrong warning message was displayed when an incorrect path was provided for *module_folders* in the configuration file. (#303)

1.4.25 Version 1.0.0

- **Core software:**
  - **Changes:**
    * FAST-OAD configuration file is now in YAML format. (#277)
    * Module declaration are now done using Python decorators directly on registered classes. (#259)
    * FAST-OAD now supports custom modules as plugins. (#266)
    * Added "fastoad.loop.wing_position" module for computing wing position from target static margin in MDA. (#268)
    * NaN values in input data are now detected at computation start. (#273)
    * Now api.generate_inputs() returns the path of generated file. (#254)
    * *fastoad list_systems* is now *fastoad list_modules* and shows documentation for OpenMDAO options. (#287)
    * Connection of OpenMDAO variables can now be done in configuration file. (#263)
    * More generic code for mass breakdown plots to ease usage for custom weight models. (#250)
    * DataFile class has been added for convenient interaction with FAST-OAD data files. (#293)
    * Moved some part of code to private API. What is still public will be kept and maintained. (#295)
  - **Bug fixes:**
    * FAST-OAD was crashing when mpi4py was installed. (#272)
    * Output of *fastoad list_variables* can now be redirected in a file. (#284)
    * Activation of time-step mission computation in tutorial notebook is now functional. (#285)
    * Variable viewer toolbar now works correctly in JupyterLab. (#288)
    * N2 diagrams caused a 404 error in notebooks since OpenMDAO 3.7. (#289)
- **Models:**
  - **Changes:**
    * A notebook has been added that shows how to compute CeRAS-01 aircraft. (#275)

    * **Unification of performance module. (#251)**
        - Breguet computations are now defined using the mission input file.
        - A computed mission can now be integrated or not to the sizing process.
    * Better management of speed parameters in Atmosphere class. (#281)
    * More robust airfoil profile processing. (#256)
    * Added tuner parameter in computation of compressibility. (#258)

1.4.26 Version 0.5.4-beta

- Bug fix: An infinite loop could occur if custom modules were declaring the same variable several times with different units or default values.

1.4.27 Version 0.5.3-beta

- Added compatibility with OpenMDAO 3.4, which is now the minimum required version of OpenMDAO. (#231)
- Simplified call to VariableViewer. (#221)
- Bug fix: model for compressibility drag now takes into account sweep angle and thickness ratio. (#237)
- Bug fix: at installation, minimum version of Scipy is forced to 1.2. (#219)
- Bug fix: SpeedChangeSegment class now accepts Mach number as possible target. (#234)
- Bug fix: variable "data:weight:aircraft_empty:mass has now "kg" as unit. (#236)

1.4.28 Version 0.5.2-beta

- Added compatibility with OpenMDAO 3.3. (#210)
- Added computation time in log info. (#211)
- Fixed bug in XFOIL input file. (#208)
- Fixed bug in copy_resource_folder(). (#212)

1.4.29 Version 0.5.1-beta

- Now avoids apparition of numerous deprecation warnings from OpenMDAO.

1.4.30 Version 0.5.0-beta

- Added compatibility with OpenMDAO 3.2.
- Added the mission performance module (currently computes a fixed standard mission).
- Propulsion models are now declared in a specific way so that another module can do a direct call to the needed propulsion model.

1.4.31 Version 0.4.2-beta

- Prevents installation of OpenMDAO 3.2 and above for incompatibility reasons.
- In Breguet module, output values for climb and descent distances were 1000 times too large (computation was correct, though).

1.4.32 Version 0.4.0-beta

Some changes in mass and performances components:

- The Breguet performance model can now be adjusted through input variables in the "settings" section.
- The mass-performance loop is now done through the "fastoad.loop.mtow" component.

1.4.33 Version 0.3.1-beta

- Adapted the FAST-OAD code to handle OpenMDAO version 3.1.1.

1.4.34 Version 0.3.0-beta

- In Jupyter notebooks, VariableViewer now has a column for input/output type.
- Changed base OAD process so that propulsion model can now be directly called by the performance module instead of being a separate OpenMDAO component (which is still possible, though). It prepares the import of FAST legacy mission-based performance model.

1.4.35 Version 0.2.2-beta

- Changed dependency requirement to have OpenMDAO version at most 3.1.0 (FAST-OAD is not yet compatible with 3.1.1)

1.4.36 Version 0.2.1-beta

- Fixed compatibility with wop 1.9 for XDSM generation

1.4.37 Version 0.2.0b

- First beta release

1.4.38 Version 0.1.0a

- First alpha release

1.5 General documentation

Here you will find the first things to know about FAST-OAD.

1.5.1 Installation procedure

Prerequisite:FAST-OAD needs at least **Python 3.7.0**.

It is recommended (but not required) to install FAST-OAD in a virtual environment (conda, venv…)

Once Python is installed, FAST-OAD can be installed using pip.

> **Note**: If your network uses a proxy, you may have to do some settings for pip to work correctly

You can install the latest version with this command:

```
$ pip install --upgrade fast-oad
```

1.5.2 FAST-OAD overview

FAST-OAD is a framework for performing rapid Overall Aircraft Design.

It proposes multi-disciplinary analysis and optimisation by relying on the OpenMDAO framework.

FAST-OAD allows easy switching between models for a same discipline, and also adding/removing disciplines to match the need of your study.

Currently, FAST-OAD is bundled with models for commercial transport aircraft of years 1990-2000. Other models will come and you may create your own models and use them instead of bundled ones.

How it works

A FAST-OAD run wraps up an OpenMDAO problem, which is, in a nutshell, the assembly of components that each have input and output variables. Of course, the outputs of some component can be the inputs of some other ones, so that the whole system can be solved.

FAST-OAD allows to define the problem to solve (or to optimize) through a configuration file that makes easy to add/remove/replace any component. By doing that, the input data of the problem can be very different from one problem to the other, but FAST-OAD comes with facilities to build the needed input data files.

A FAST-OAD problem can be fully run from *command line interface* or from the Python API.

Usage of Python API, including pre-processing and post-processing utilities are currently provided through *Python notebooks*.

Overview of FAST-OAD files

A typical run of FAST-OAD uses two types of user files:

configuration file (.yml)

This file defines the OpenMDAO problem by defining :

- what components will be in the problem
- the files for input and output data
- the problem settings
- the definition of the optimization problem if needed

A detailed description of this file can be found *here*.

The input and output data files (.xml)

These files contain the information of the variables involved in the system model:

1. The input file contains the global inputs values required to run all the model. The user is free to modify the values of the variables in order that these new values are considered during a model run.
2. The output file contains all the variables (inputs + outputs) values obtained after a model run.

The content of these files and the way variables are named and serialized is described *here*.

1.5.3 Usage

FAST-OAD uses a configuration file for defining your OAD problem. You can interact with this problem using command line or Python directly.

You may also use some lower-level features of FAST-OAD to interact with OpenMDAO systems. This part is addressed in the *API documentation*.

Contents

FAST-OAD configuration file

FAST-OAD configuration files are in YAML format. A quick tutorial for YAML (among many ones) is available here

```
title: Sample OAD Process

# List of folder paths where user added custom registered OpenMDAO components
module_folders:

# Input and output files
input_file: ./problem_inputs.xml
output_file: ./problem_outputs.xml

# Definition of problem driver assuming the OpenMDAO convention "import openmdao.api as␣
↪om"
driver: om.ScipyOptimizeDriver(tol=1e-2, optimizer='COBYLA')

# Definition of OpenMDAO model
# Although "model" is a mandatory name for the top level of the model, its
# sub-components can be freely named by user
model:

  #  Solvers are defined assuming the OpenMDAO convention "import openmdao.api as om"
  nonlinear_solver: om.NonlinearBlockGS(maxiter=100, atol=1e-2)
  linear_solver: om.DirectSolver()


  # Components can be put in sub-groups
  subgroup:

    # A group can be set with its own solvers.
```

(continues on next page)

(continued from previous page)

```
    nonlinear_solver: om.NonlinearBlockGS(maxiter=100, atol=1e-2, iprint=0)
    linear_solver: om.DirectSolver()

    geometry:
      # An OpenMDAO component is identified by its "id"
      id: fastoad.geometry.legacy
    weight:
      id: fastoad.weight.legacy
    mtow:
      id: fastoad.mass_performances.compute_MTOW
    hq_tail_sizing:
      id: fastoad.handling_qualities.tail_sizing
    hq_static_margin:
      id: fastoad.handling_qualities.static_margin
    wing_position:
      id: fastoad.loop.wing_position
    aerodynamics_highspeed:
      id: fastoad.aerodynamics.highspeed.legacy
    aerodynamics_lowspeed:
      id: fastoad.aerodynamics.lowspeed.legacy
    aerodynamics_takeoff:
      id: fastoad.aerodynamics.takeoff.legacy
    aerodynamics_landing:
      id: fastoad.aerodynamics.landing.legacy
      use_xfoil: false
  performance:
    id: fastoad.performances.mission
    propulsion_id: fastoad.wrapper.propulsion.rubber_engine
    # mission_file_path: ::sizing_breguet
    mission_file_path: ::sizing_mission
    out_file: ./flight_points.csv
    adjust_fuel: true
    is_sizing: true
  wing_area:
    id: fastoad.loop.wing_area

optimization: # This section is needed only if optimization process is run
  design_variables:
    - name: data:geometry:wing:aspect_ratio
      lower: 9.0
      upper: 18.0
  constraints:
    - name: data:geometry:wing:span
      upper: 60.0
  objective:
    - name: data:mission:sizing:needed_block_fuel
      scaler: 1.e-4
```

Now in details:

Custom module path

```
module_folders:
```

Provides the path where user can have his custom OpenMDAO modules. See section *How to add custom OpenMDAO modules to FAST-OAD*.

Input and output files

```
input_file: ./problem_inputs.xml
output_file: ./problem_outputs.xml
```

Specifies the input and output files of the problem. They are defined in the configuration file and DO NOT APPEAR in the command line interface.

Problem driver

```
driver: om.ScipyOptimizeDriver(tol=1e-2, optimizer='COBYLA')
```

This belongs the domain of the OpenMDAO framework and its utilization. This setting is needed for optimization problems. It is defined as in Python when assuming the OpenMDAO convention `import openmdao.api as om`.

For more details, please see the OpenMDAO documentation on drivers.

Solvers

```
model:
  nonlinear_solver: om.NonlinearBlockGS(maxiter=100, atol=1e-2)
  linear_solver: om.DirectSolver()
```

This is the starting point for defining the model of the problem. The model is a group of components. If the model involves cycles, which happens for instance when some outputs of A are inputs of B, and vice-versa, it is necessary to specify solvers as done above.

For more details, please see the OpenMDAO documentation on nonlinear solvers and linear solvers.

Problem definition

```
model:
  nonlinear_solver: om.NonlinearBlockGS(maxiter=100, atol=1e-2)
  linear_solver: om.DirectSolver()

  # Components can be put in sub-groups
  subgroup:

    # A group can be set with its own solvers.

    nonlinear_solver: om.NonlinearBlockGS(maxiter=100, atol=1e-2, iprint=0)
```

(continues on next page)

(continued from previous page)

```
    linear_solver: om.DirectSolver()

    geometry:
      # An OpenMDAO component is identified by its "id"
      id: fastoad.geometry.legacy
    weight:
      id: fastoad.weight.legacy
    mtow:
      id: fastoad.mass_performances.compute_MTOW
    hq_tail_sizing:
      id: fastoad.handling_qualities.tail_sizing
    hq_static_margin:
      id: fastoad.handling_qualities.static_margin
    wing_position:
      id: fastoad.loop.wing_position
    aerodynamics_highspeed:
      id: fastoad.aerodynamics.highspeed.legacy
    aerodynamics_lowspeed:
      id: fastoad.aerodynamics.lowspeed.legacy
    aerodynamics_takeoff:
      id: fastoad.aerodynamics.takeoff.legacy
    aerodynamics_landing:
      id: fastoad.aerodynamics.landing.legacy
      use_xfoil: false
  performance:
    id: fastoad.performances.mission
    propulsion_id: fastoad.wrapper.propulsion.rubber_engine
    # mission_file_path: ::sizing_breguet
    mission_file_path: ::sizing_mission
    out_file: ./flight_points.csv
    adjust_fuel: true
    is_sizing: true
  wing_area:
    id: fastoad.loop.wing_area
```

Components of the model can be modules, or sub-groups. They are defined as a sub-section of `model:`. Sub-sections and sub-components can be freely named by user.

A sub-group gathers several modules and/or other sub-groups and can be set with its own solvers to resolve cycles it may contains, using keys `linear_solver` and `nonlinear_solver`, like `model` (that is merely the root group).

Here above, a sub-group with geometric, weight, handling-qualities and aerodynamic modules is defined and internal solvers are activated. Performance and wing area computation modules are set apart.

A module is defined by its `id:` key that refers to the module registered name.

Additional keys can be used in `model`, sub-groups and modules. They are interpreted as option settings:

- For `model` and sub-groups, the OpenMDAO options for Group class apply.
- For FAST-OAD modules, the list of available options is available through the `list_modules` sub-command (see *How to get list of registered modules*).

Model options

OpenMDAO 3.27 introduced a new way to set options for any component in the problem, using the `model_options` attribute of the `Problem` object (see OpenMDAO documentation here).

This can be controlled from the configuration file, using for instance:

```
model_options:
  "*":
    propulsion_id: fastoad.wrapper.propulsion.rubber_engine
  "aerodynamics.*":
    use_xfoil: true
```

With above lines, we set the `"propulsion_id"` option for all concerned components in the problem, and we set the `"use_xfoil"` option for all components inside the `aerodynamics` module (please see OpenMDAO documentation for more examples using wildcards).

Note:

- Please note that the wildcards have to be (double) quoted.
- This feature is especially convenient to set options for sub-components of the declared models, since these options are not directly accessible from the configuration file.

Optimization settings

This settings are used only when using optimization (see *Run Multi-Disciplinary Optimization*). They are ignored when doing analysis (see *Run Multi-Disciplinary Analysis*).

The section is identified by:

```
optimization:
```

Design variables

```
design_var:
  - name: data:geometry:wing:MAC:at25percent:x
    lower: 16.0
    upper: 18.0
```

Here are defined design variables (relevant only for optimization). Keys of this section are named after parameters of the OpenMDAO System.add_design_var() method

Several design variables can be defined.

Also, see *How to get list of variables*.

Objective function

```
objective:
  - name: data:mission:sizing:fuel
```

Here is defined the objective function (relevant only for optimization). Keys of this section are named after parameters of the OpenMDAO System.add_objective() method

Only one objective variable can be defined.

Also, see *How to get list of variables*.

Constraints

```
constraint:
  - name: data:handling_qualities:static_margin
    lower: 0.05
    upper: 0.1
```

Here are defined constraint variables (relevant only for optimization). Keys of this section are named after parameters of the OpenMDAO System.add_constraint() method

Several constraint variables can be defined.

Also, see *How to get list of variables*.

Using FAST-OAD through Command line

FAST-OAD can be used through shell command line or Python. This section deals with the shell command line, but if you prefer using Python, you can skip this part and go to *Using FAST-OAD through Python*.

The FAST-OAD command is `fastoad`. Inline help is available with:

```
$ fastoad -h
```

fastoad works through sub-commands. Each sub-command provides its own inline help using

```
$ fastoad <sub-command> -h
```

How to get information about available plugins

FAST-OAD is built on a plugin architecture where each plugin can provide FAST-OAD modules, Jupyter notebooks and sample configuration files (see *plugin addition*),

A list of installed plugins can be obtained with:

```
$ fastoad plugin_info
```

How to generate a configuration file

FAST-OAD can provide a ready-to use configuration.

```
$ fastoad gen_conf my_conf.yml --from_package my_plugin_package --source sample_
↪configuration_1.yml
```

This copies the file `sample_configuration_1.yml`provided by installed package :code:`my_plugin_package` to file `my_conf.yml`.

See *how to get plugin information* for listing the values you can put for options `--from_package` and `--source`.

If only one package is available, option `--from_package` may be omitted. If the selected package provides only one configuration file, option `--source` may be omitted.

Hence with FAST-OAD installed (version below 2.0) without additional plugin, the command can be:

```
$ fastoad gen_conf my_conf.yml
```

How to get list of registered modules

If you want to change the list of components in the model in the configuration file, you need the list of available modules.

List of FAST-OAD modules can be obtained with:

```
$ fastoad list_modules
```

If you added custom modules in your configuration file `my_conf.yml` (see *how to add custom OpenMDAO modules to FAST-OAD*), they can be listed along FAST-OAD modules with:

```
$ fastoad list_modules my_conf.yml
```

You may also use the `--verbose` option to get detailed information on each module, including the available options, if any.

How to get list of variables

Once your problem is defined in *my_conf.yml*, you can get a list of the variables of your problem with:

```
$ fastoad list_variables my_conf.yml
```

How to generate an input file

The name of the input file is defined in your configuration file *my_conf.yml*. This input file can be generated with:

```
$ fastoad gen_inputs my_conf.yml
```

The generated file will be an XML file that contains needed inputs for your problem. Values will be the default values from module definitions, which means several ones will be "nan". Actual value must be filled before the process is run.

If you already have a file that contains these values, you can use it to populate your new input files with:

```
$ fastoad gen_inputs my_conf.yml my_ref_values.xml
```

If you are using the configuration file provided by the gen_conf sub-command (see *How to generate a configuration file*), you may download our CeRAS01_baseline.xml and use it as source for generating your input file. You may also generate a source data file using the appropriate command (see *How to generate a source data file*)

How to generate a source data file

As for the configuration file, FAST-OAD can provide a source data file usable for the generation of your input file.

```
$ fastoad gen_source_data_file my_source_data_file.xml --from_package my_plugin_package -
↪-source sample_source_data_file_1.xml
```

This copies the file `sample_source_data_file_1.xml` provided by installed package `my_plugin_package` to file `my_source_data_file.xml`.

The remarks made in section *how to generate a configuration file* on options `--from_package` and `--source` remain valid when generating a source data file.

How to view the problem process

FAST-OAD proposes two graphical ways to look at the problem defined in configuration file. This is especially useful to see how models and variables are connected.

N2 diagram

FAST-OAD can use OpenMDAO to create a N2 diagram. It provides in-depth information about the whole process.

You can create a `n2.html` file with:

```
$ fastoad n2 my_conf.yml
```

XDSM

Using WhatsOpt as web service, FAST-OAD can provide a XDSM.

XDSM offers a more synthetic view than N2 diagram.

As it uses a web service, you need an internet access for this command, but you do not need to be a registered user on the WhatsOpt server.

You can create a `xdsm.html` file with:

```
$ fastoad xdsm my_conf.yml
```

Note: It may take a couple of minutes

Also, you may see WhatsOpt developer documentation to run your own server. In such case, you will address your server by using the `--server` option:

```
$ fastoad xdsm my_conf.yml --server https://the/address/of/my/WhatsOpt/server
```

How to run the problem

Run Multi-Disciplinary Analysis

Once your problem is defined in *my_conf.yml*, you can simply run it with:

```
$ fastoad eval my_conf.yml
```

Note: This is equivalent to OpenMDAO's run_model()

Run Multi-Disciplinary Optimization

You can also run the defined optimization with:

```
$ fastoad optim my_conf.yml
```

Note: This is equivalent to OpenMDAO's run_driver()

Using FAST-OAD through Python

The command line interface can generate Jupyter notebooks that show how to use the high-level interface of FAST-OAD.

To do so, type this command **in your terminal**:

```
$ fastoad notebooks
```

Then run the Jupyter server as indicated in the obtained message.

1.5.4 Problem variables

FAST-OAD process relies on OpenMDAO, and process variables are OpenMDAO variables.

For any component, variables are declared as inputs or outputs as described here.

FAST-OAD uses the promotion system of OpenMDAO, which means that all variables that are exchanged between FAST-OAD registered systems[1] have a unique name and are available for the whole process.

The list of variable names and descriptions for a given problem can be obtained from command line (see *How to get list of variables*).

- *Variable naming*

[1] see *Register your system(s)*

Variable naming

Variables are named with a path-like pattern where path separator is `:`, e.g.:

- `data:geometry:wing:area`
- `data:weight:airframe:fuselage:mass`
- `data:weight:airframe:fuselage:CG:x`

The first path element distributes variables among three categories:

- `data`: variables that define the aircraft and its behaviour. This is the main category
- `settings`: model settings. Generally coefficients for advanced users
- `tuning`: coefficients that allow to do some assumptions (e.g.: "what if wing mass could be reduced of 20%?")

The second path element tells about the nature of the variable (geometry, aerodynamics, weight, ...).

The other path elements depend of the variable. The number of path elements is not fixed.

Serialization

File format

For writing input and output files, FAST-OAD relies on the path in the variable names.

For instance, the variable name `data:geometry:wing:area` will be split according to colons `:` and each part of the name will become a level in the XML hierarchy:

```
<data>
    <geometry>
        <wing>
            <area units="m**2">
                150.0
            </area>
        </wing>
    </geometry>
</data>
```

A complete file that would contain the three above-mentioned variables will be as following:

```
<FASTOAD_model>
    <data>
        <geometry>
            <wing>
                <area units="m**2">150.0</area>
            </wing>
        </geometry>
        <weight>
```

(continues on next page)

(continued from previous page)

```
            <fuselage>
                <mass units="kg">10000.0</mass>
                <CG>
                    <x units="m">20.0</x>
                </CG>
            </fuselage>
        </weight>
    </data>
</FASTOAD_model>
```

Note: Units are given as a string according to OpenMDAO units definitions

Note: XML requires a unique root element for containing all other ones. Its name can be freely chose, but it is *FASTOAD_model* in files written by FAST-OAD

FAST-OAD API

FAST-OAD proposes a convenient way to read/write such files in Python, through the *DataFile* class.

Provided that above file is named `data.xml`, following commands apply:

```
>>> import fastoad.api as oad
>>> # ---------------------------------
>>> datafile = oad.DataFile("./data.xml")
>>> # Getting information
>>> datafile.names()
['data:geometry:wing:area', 'data:weight:fuselage:mass', 'data:weight:fuselage:CG:x']
>>> len(datafile)
3
>>> datafile["data:geometry:wing:area"].value
[150.0]
>>> datafile["data:geometry:wing:area"].units
'm**2'
>>> # ---------------------------------
>>> # Writing data
>>> datafile.save()
>>> # ---------------------------------
>>> # Modifying data
>>> datafile["data:geometry:wing:area"].value = 120.0  # no need to provide list or␣
↪numpy array for scalar values.
>>> datafile["data:geometry:wing:area"].value
120.0
>>> # ---------------------------------
>>> # Adding data
>>> fuselage_length = oad.Variable("data:geometry:fuselage:length", val=35.0, units="m")
>>> datafile.append(fuselage_length)
>>> # or ...
>>> datafile["data:geometry:wing:mass"] = dict(val=10500.0, units="kg") # will replace␣
↪previous definition
```

(continues on next page)

(continued from previous page)

```
>>> datafile.names()
['data:geometry:wing:area', 'data:weight:fuselage:mass', 'data:weight:fuselage:CG:x',
↪'data:geometry:fuselage:length', 'data:geometry:wing:mass']
>>> # ---------------------------------
>>> # Removing data
>>> del datafile["data:weight:fuselage:CG:x"]
>>> datafile.names()
['data:geometry:wing:area', 'data:weight:fuselage:mass', 'data:geometry:fuselage:length',
↪ 'data:geometry:wing:mass']
>>> # ---------------------------------
>>> # Writing to another file
>>> datafile.save_as("./new_data.xml", overwrite=True)
>>> datafile.file_path  # The object is now associated to the new path
'./new_data.xml'
```

After running these lines of code, the generated file `new_data.xml` contains:

```
<FASTOAD_model>
    <data>
        <geometry>
            <fuselage>
                <length units="m">35.0</length>
            </fuselage>
            <wing>
                <area units="m**2">120.0</area>
                <mass units="kg">10500.0</mass>
            </wing>
        </geometry>
        <weight>
            <fuselage>
                <mass units="kg">10000.0</mass>
            </fuselage>
        </weight>
    </data>
</FASTOAD_model>
```

1.5.5 Mission module

Here you will find information about the performance module in FAST-OAD.

Defining missions

Here you will find information about the mission definition files for the FAST-OAD performance module.

Mission file

General description

A mission file describes precisely one or several missions that could be computed by the performance model `fastoad.performances.mission` of FAST-OAD.

The file format of mission files is the YAML format. A quick tutorial for YAML (among many ones) is available here.

The mission definition relies on 4 concepts that are, from lowest level to the highest one: segments, phases, routes and missions. They are summarized in this table:

Table 1: Mission elements

Type	Parts	Description
segment	N/A	The basic bricks that are provided by FAST-OAD. They are described in this *specific page*.
phase	segment(s) and/or phase(s)	A free assembly of one or more segments and/or other phases.
route	zero or more phase(s) **one cruise segment** zero or more phase(s)	A route is a climb/cruise/descent sequence with a fixed range. The range is achieved by adjusting the distance covered during the cruise part.
mission	routes and/or phases and/or segments	A mission is what is computed by `fastoad.performances.mission`. Generally, it begins when engine starts and ends when engine stops.

Important: Starting with version 1.4.0 of FAST-OAD, any mission has to use a variable for mass input. This variable can be defined using the *start segment*, if it provides the mass at mission start (typically a ramp-up weight), or using the *mass_input segment* otherwise (typically a takeoff weight, achieved after the taxi-out).

In the case no variable is defined for input mass, FAST-OAD will automatically add, at the beginning of the mission, a taxi-out and a very simple takeoff phase (*transition segment*) with a *mass_input segment*. In that case, the input mass is given by the `data:mission:<mission_name>:TOW` variable, which represents the aircraft mass just **after** takeoff.

This addition of taxi-out, takeoff and mass input allows to keep compatibility with mission definitions for FAST-OAD versions earlier than 1.4.

(Please note that takeoff weight should be actually considered as the mass just **before** takeoff, but this way of doing is kept for maximum backward-compatibility)

File sections

The organization of a mission definition file is organized in sections according to above-defined concepts.

Phase definition section

This section, identified by the `phases` keyword, defines flight phases. A flight phase is defined as an assembly of one or more *flight segment(s)*.

Basically, a phase has a name, and a `parts` attribute that contains a list of segment definitions.

Nevertheless, it is also possible to set, at phase level, the parameters that are common to several segments of the phase.

The phase section only defines flight phases, but not their usage, that is defined in *route* and *mission* sections. Therefore, the definition order of flight phases has no importance.

Note: Some parameters may be more conveniently set at an upper level than segment-level. See section *Factorizing parameters* to see how.

Example:

```
phases:
  initial_climb:                                # Phase name
    parts:                                          # Definition of segment list
      - segment: altitude_change                    # 1st segment (climb)
        polar: data:aerodynamics:aircraft:takeoff
        thrust_rate: 1.0
        target:
          altitude:
```

(continues on next page)

(continued from previous page)

```
          value: 400.
          unit: ft
        equivalent_airspeed: constant
    - segment: speed_change                          # 2nd segment (acceleration)
      polar: data:aerodynamics:aircraft:takeoff
      thrust_rate: 1.0
      target:
        equivalent_airspeed:
          value: 250
          unit: kn
    - segment: altitude_change                       # 3rd segment (climb)
      polar: data:aerodynamics:aircraft:takeoff
      thrust_rate: 0.95
      target:
        altitude:
          value: 1500.
          unit: ft
        equivalent_airspeed: constant
  climb:                                        # Phase name
    ...                                              # Definition of the phase...
```

Specific takeoff phase definition section

The takeoff and associated manoeuvres may be simulated by assembling the specific segments. An exemple of takeoff phase definition, as well as start-stop phase are given here:

Example:

```
takeoff:
    engine_setting: takeoff
    polar:
      CL: data:aerodynamics:aircraft:takeoff:CL
      CD: data:aerodynamics:aircraft:takeoff:CD
      ground_effect: None # Ground effect model selection
      CL0_clean: data:aerodynamics:aircraft:takeoff:CL0_clean
      CL_alpha: data:aerodynamics:aircraft:takeoff:CL_alpha
      CL_high_lift: data:aerodynamics:high_lift_devices:takeoff:CL
    thrust_rate: 1.0
    isa_offset: data:mission:operational:ISA_offset
    parts:
      - segment: ground_speed_change
        target:
          equivalent_airspeed:
            value: data:mission:operational:takeoff:Vr
      - segment: rotation
        target:
          delta_altitude:
            value: 35
            unit: ft
      - segment: end_of_takeoff
        time_step: 0.05
```

(continues on next page)

(continued from previous page)

```
        target:
          delta_altitude:
            value: 35
            unit: ft
start_stop: # start - stop manoeuvre with only brakes on
    engine_setting: takeoff
    polar:
      CL: data:aerodynamics:aircraft:takeoff:CL
      CD: data:aerodynamics:aircraft:takeoff:CD
      ground_effect: Raymer # Ground effect model selection
      CL0_clean: data:aerodynamics:aircraft:takeoff:CL0_clean
      CL_alpha: data:aerodynamics:aircraft:takeoff:CL_alpha
      CL_high_lift: data:aerodynamics:high_lift_devices:takeoff:CL
    thrust_rate: 1.0
    isa_offset: data:mission:operational:ISA_offset
    parts:
      - segment: ground_speed_change
        wheels_friction: 0.03
        time_step: 0.05
        target:
          equivalent_airspeed:
            value: data:mission:operational:takeoff:V1
      - segment: ground_speed_change
        engine_setting: idle
        thrust_rate: 0.07
        wheels_friction: 0.5
        time_step: 0.05
        target:
          true_airspeed:
            value: 0
            unit: m/s
```

Route definition section

This section, identified by the `routes` keyword, defines flight routes. A flight route is defined as climb/cruise/descent sequence with a fixed range. The range is achieved by adjusting the distance covered during the cruise part. Climb and descent phases are computed normally.

A route is identified by its name and has 4 attributes:

- `range`: the distance to be covered by the whole route
- `climb_parts`: a list of items like `phase : <phase_name>`
- `cruise_part`: a *segment* definition, except that it does not need any target distance.
- `descent_parts`: a list of items like `phase : <phase_name>`

Example:

```
routes:
  main_route:
    range:
      value: 3000.
```

(continues on next page)

(continued from previous page)

```
      unit: NM
    climb_parts:
      - phase: initial_climb
      - phase: climb
    cruise_part:
      segment: cruise
      engine_setting: cruise
      polar: data:aerodynamics:aircraft:cruise
      target:
        altitude: optimal_flight_level
      maximum_flight_level: 340
    descent_parts:
      - phase: descent
  diversion:
    range: distance
    climb_parts:
      - phase: diversion_climb
    cruise_part:
      segment: breguet
      engine_setting: cruise
      polar: data:aerodynamics:aircraft:cruise
    descent_parts:
      - phase: descent
```

Mission definition section

This is the main section. It allows to define one or several missions, that will be computed by the mission module.

A mission is identified by its name and has 3 attributes:

- `parts`: list of the *phase* and/or *route* names that compose the mission, with optionally a last item that is the `reserve` (see below).
- `use_all_block_fuel`: if True, the range of the main *route* of the mission will be adjusted so that all block fuel (provided as input *data:mission:<mission_name>:block_fuel*) will be consumed for the mission, excepted the reserve, if defined. The provided range for first route is overridden but used as a first guess to initiate the iterative process.

The mission name is used when configuring the mission module in the FAST-OAD configuration file. **If there is only one mission defined in the file, naming it in the configuration file is optional.**

Note: About reserve

The `reserve` keyword is typically designed to define fuel reserve as stated in EU-OPS 1.255.

It defines the amount of fuel that is expected to be still in tanks once the mission is complete. It takes as reference one of the route that composes the mission (`ref` attribute). The reserve is defined as the amount of fuel consumed during the referenced route, multiplied by the coefficient provided as the `multiplier` attribute.

Example:

```
missions:
  sizing:
```

(continues on next page)

(continued from previous page)

```
    parts:
      - phase: taxi_out
      - phase: takeoff
      - route: main_route
      - route: diversion
      - phase: holding
      - phase: landing
      - phase: taxi_in
      - reserve:
          ref: main_route
          multiplier: 0.03
  operational:
    parts:
      - phase: taxi_out
      - phase: takeoff
      - route: main_route
      - phase: landing
      - phase: taxi_in
  fuel_driven:
    parts:
      - phase: taxi_out
      - phase: takeoff
      - route: main_route
      - phase: landing
      - phase: taxi_in
    use_all_block_fuel: true
```

Factorizing parameters

Some parameters may be common to several segments and have same value across all of them. In such case, it is possible to define them at higher level (i.e. phase, route or mission) to avoid repeating them.

For example, to specify a temperature increment at mission level, the mission section could be:

```
missions:
  operational:
    isa_offset: 15.0            # It will apply to the whole mission
    parts:
      - route: main_route
      - phase: landing
      - phase: taxi_in
```

A high-level parameter definition will be overloaded by a lower-level definition, as illustrated in this example of phase definition:

```
phases:
  initial_climb:                                # Phase name
    engine_setting: takeoff                     # ---------------
    polar: data:aerodynamics:aircraft:takeoff   #   Common segment
    thrust_rate: 1.0                            #   parameters
    time_step: 0.2                              # ---------------
```

(continues on next page)

(continued from previous page)

```yaml
  parts:                                          # Definition of segment list
    - segment: altitude_change                      # 1st segment (climb)
      target:
        altitude:
          value: 400.
          unit: ft
        equivalent_airspeed: constant
    - segment: speed_change                         # 2nd segment (acceleration)
      target:
        equivalent_airspeed:
          value: 250
          unit: kn
    - segment: altitude_change                      # 3rd segment (climb)
      thrust_rate: 0.95        # --> PHASE THRUST RATE VALUE IS OVERWRITTEN
      target:
        altitude:
          value: 1500.
          unit: ft
        equivalent_airspeed: constant
```

Flight segments

Flight segments are the Python-implemented, base building blocks for the mission definition.

They can be used as parts in *phase* definition.

A segment simulation starts at the flight parameters (altitude, speed, mass…) reached at the end of the previous simulated segment. The segment simulation ends when its **target** is reached (or if it cannot be reached).

Sections:

Segment types

In the following, the description of each segment type links to the documentation of the Python implementation. All parameters of the Python constructor can be set in the mission file (except for `propulsion` and `reference_area` that are set within the mission module). Most of these parameters are scalars and can be set as described *here*. The segment target is a special parameter, detailed in *further section*. Other special parameters are detailed in *last section*.

Available segments are:

- *altitude_change segment*
- *cruise segment*
- *optimal_cruise segment*
- *holding segment*
- *taxi segment*
- *transition segment*
- *ground_speed_change segment*
- *rotation segment*
- *end_of_takeoff segment*
- *takeoff segment*

`start` segment

New in version 1.4.0.

`start` is a special segment to be used at the beginning of the mission definition to specify the starting point of the mission, preferably by defining variables so it can be controlled from FAST-OAD input file.

With no `start` specified, the mission is assumed to start at altitude 0.0, speed 0.0.

Note: The `start` segment allows to define the aircraft mass at the beginning of the mission. Yet it is possible to define aircraft mass at some intermediate phase (e.g. takeoff) using the *mass_input segment*.

Important: In any case, a variable for input mass has to be defined once and only once in the whole mission.

Example:

```
phases:
  start_phase:
    - segment: start
      target:
        true_airspeed: 0.0                # hard-coded value
        altitude:
          value: my:altitude:variable     # variable definition WITH associated default␣
↪value
          unit: ft
          default: 100.0
        mass:
          value: my:mass:variable         # variable definition WITHOUT associated␣
↪default value
          unit: kg                        # (will have to be set by another module or by␣
↪FAST-OAD
                                        #  input file)

...
```

(continues on next page)

(continued from previous page)

```
missions:
  main_mission:
    parts:
      - phase: start_phase
      - ...
```

See also:

Python documentation: *Start*

mass_input segment

New in version 1.4.0.

The *start segment* allows to define aircraft mass at the beginning of the mission, but it is sometimes needed to define the aircraft mass at some point in the mission. The typical example would be the need to specify a takeoff weight that is expected to be achieved after the taxi-out phase.

The `mass_input` segment is designed to address this need. It will ensure this mass is achieved at the specify instant in the mission by setting the start mass input accordingly.

Example:

```
# For setting mass at the end of taxi-out:
phases:
  taxi-out:
    parts:
      - segment: taxi
        ...
      - segment: mass_input
        target:
          mass:
            value: my:MTOW:variable
            unit: kg
```

Warning: Currently, FAST-OAD assumes the fuel consumption before the `mass_input` segment is independent of aircraft mass, which is considered true in a phase such as taxi. Assuming otherwise would require to solve an additional inner loop. Since it does not correspond to any use case we currently know of, it has been decided to stick to the simple case.

See also:

Python documentation: *MassTargetSegment*

`speed_change` segment

A `speed_change` segment simulates an acceleration or deceleration flight part, at constant altitude and thrust rate. It ends when the target speed (mach, true_airspeed or equivalent_airspeed) is reached.

Example:

```yaml
segment: speed_change
polar: data:aerodynamics:aircraft:takeoff   # High-lift devices are ON
engine_setting: takeoff
thrust_rate: 1.0                            # Full throttle
target:
  # altitude: constant                      # Assumed by default
  equivalent_airspeed:                      # Acceleration up to EAS = 250 knots
    value: 250
    unit: kn
```

See also:

Python documentation: *SpeedChangeSegment*

`altitude_change` segment

An `altitude_change` segment simulates a climb or descent flight part at constant thrust rate. Typically, it ends when the target altitude is reached.

But also, a target speed can be set, while keeping another speed constant (e.g. climbing up to Mach 0.8 while keeping equivalent_airspeed constant).

Examples:

```yaml
segment: altitude_change
polar: data:aerodynamics:aircraft:cruise    # High speed aerodynamic polar
engine_setting: idle
thrust_rate: 0.15                           # Idle throttle
target:                                     # Descent down to 10000. feet at constant EAS
  altitude:
    value: 10000.
    unit: ft
  equivalent_airspeed: constant
```

```yaml
segment: altitude_change
polar: data:aerodynamics:aircraft:cruise    # High speed aerodynamic polar
engine_setting: climb
thrust_rate: 0.93                           # Climb throttle
target:                                     # Climb up to Mach 0.78 at constant EAS
  equivalent_airspeed: constant
  mach: 0.78
```

```yaml
segment: altitude_change
polar: data:aerodynamics:aircraft:cruise    # High speed aerodynamic polar
engine_setting: climb
thrust_rate: 0.93                           # Climb throttle
target:                                     # Climb at constant Mach up to the flight
```

(continues on next page)

(continued from previous page)

```
  mach: constant                                #  level that provides maximum lift/drag
  altitude:                                     #  at current mass.
    value: optimal_flight_level
maximum_CL: 0.6                                 # Limitation on lift coefficient.
                                                # The altitude will be limited to the closest
                                                # flight level within the CL limitation.
```

See also:

Python documentation: *AltitudeChangeSegment*

`cruise` segment

A `cruise` segment simulates a flight part at constant speed and altitude, and regulated thrust rate (drag is compensated).

Optionally, target altitude can be set to `optimal_flight_level`. In such case, cruise will be preceded by a climb segment that will put the aircraft at the altitude that will minimize the fuel consumption for the whole segment (including the prepending climb). This option is available because the *altitude_change segment* segment can reach an altitude that will optimize the lift/drag ratio at current mass, but the obtained altitude will not guaranty an optimal fuel consumption for the whole cruise.

It ends when the target ground distance is covered (including the distance covered during prepending climb, if any).

Examples:

```
segment: cruise
polar: data:aerodynamics:aircraft:cruise    # High speed aerodynamic polar
engine_setting: cruise
target:
  # altitude: constant                      # Not needed, because assumed by default
  ground_distance:                          # Cruise for 2000 nautical miles
    value: 2000
    unit: NM
```

```
segment: cruise
polar: data:aerodynamics:aircraft:cruise    # High speed aerodynamic polar
engine_setting: cruise
target:
  altitude: optimal_flight_level            # Commands a prepending climb, id needed
  ground_distance:                          # Cruise for 2000 nautical miles
    value: 2000
    unit: NM
```

See also:

Python documentation: *ClimbAndCruiseSegment*

`optimal_cruise` segment

An `optimal_cruise` segment simulates a cruise climb, i.e. a cruise where the aircraft climbs gradually to keep being at altitude of maximum lift/drag ratio.

The `optimal_cruise` segment can be realised at a constant lift coefficient using the parameter `maximum_CL`.

It assumed the segment actually starts at altitude of maximum lift/drag ratio or the altitude given by `maximum_CL`, which can be achieved with an *altitude_change segment* segment with `optimal_altitude` as target altitude and `maximum_CL` as parameter.

The common way to optimize the fuel consumption for commercial aircraft is a step climb cruise. Such segment will be implemented in the future.

Example:

```
segment: optimal_cruise
polar: data:aerodynamics:aircraft:cruise    # High speed aerodynamic polar
engine_setting: cruise
maximum_CL: 0.6
target:
  ground_distance:                          # Cruise for 2000 nautical miles
    value: 2000
    unit: NM
```

See also:

Python documentation: *OptimalCruiseSegment*

`holding` segment

A `holding` segment simulates a flight part at constant speed and altitude, and regulated thrust rate (drag is compensated). It ends when the target time is covered.

Example:

```
segment: holding
polar: data:aerodynamics:aircraft:cruise    # High speed aerodynamic polar
target:
  # altitude: constant                      # Not needed, because assumed by default
  time:
    value: 20                               # 20 minutes holding
    unit: min
```

See also:

Python documentation: *HoldSegment*

`taxi` segment

A `taxi` segment simulates the mission parts between gate and takeoff or landing, at constant thrust rate. It ends when the target time is covered.

Example:

```
segment: taxi
thrust_rate: 0.3
target:
  time:
    value: 300              # taxi for 300 seconds (5 minutes)
```

See also:

Python documentation: *TaxiSegment*

`transition` segment

A `transition` segment is intended to "fill the gaps" when some flight part is not available for computation or is needed to be assessed without spending CPU time.

It can be used in various ways:

- *Target definition*
- *Usage of a mass ratio*
- *Reserve mass ratio*

Target definition

The most simple way is specifying a target with absolute and/or relative parameters. The second and last point of the flight segment will simply uses these values.

Example:

```
segment: transition # Rough simulation of a takeoff
target:
  delta_time: 60            # 60 seconds after start point
  delta_altitude:           # 35 ft above start point
    value: 35
    unit: ft
  delta_mass: -80.0         # 80kg lost from start point
  true_airspeed: 85         # 85m/s at end of segment.
```

Usage of a mass ratio

As seen above, it is possible to force a mass evolution of a certain amount by specifying `delta_mass`.

It is also possible to specify a mass ratio. This can be done outside the target, as a segment parameter.

Example:

```
segment: transition # Rough climb simulation
mass_ratio: 0.97            # Aircraft end mass will be 97% of total start mass
target:
  altitude: 10000.
  mach: 0.78
  delta_ground_distance:    # 250 km after start point.
    value: 250
    unit: km
```

Reserve mass ratio

Another segment parameter is `reserve_mass_ratio`. When using this parameter, another flight point is added to computed segment, where the aircraft mass is decreased by a fraction of the mass that remains at the end of the segment (including this reserve consumption).

Typically, it will be used as last segment to compute a reserve based on the Zero-Fuel-Weight mass.

Example:

```
segment: transition # Rough reserve simulation
reserve_mass_ratio: 0.06
target:
  altitude: 0.
  mach: 0.
```

See also:

Python documentation: *DummyTransitionSegment*

`ground_speed_change` segment

New in version 1.5.0.

This segment is used specifically during accelerating or decelerating parts while still on the ground. The friction force with the ground is accounted in the equation of movements. Whilst on the ground, the key `wheels_friction` is used to define the friction coefficient. The segment ends when the target velocity is reached.

Example:

```
segment: ground_speed_change
wheels_friction: 0.03
target:
  equivalent_airspeed:
    value: data:mission:operational:takeoff:Vr
```

See also:

Python documentation: *GroundSpeedChangeSegment*

`rotation` segment

New in version 1.5.0.

This segment is used to represent a rotation while still on the ground. This segment is specifically used for takeoff. The specific keys are (in addition to wheel friction coefficient):

`rotation_rate` in (rad/s) is the rotation speed used to realise the manoeuvre (by default 3deg/s, compliant with CS-25)

`alpha_limit` (in rad) is the maximum angle of attack possible before tail strike (by default 13.5deg).

The segment ends when lift equals weight. Therefore, no target needs to be set.

Example:

```
segment: rotation
wheels_friction: 0.03
rotation_rate:
  value: 0.0523
alpha_limit:
  value: 0.3489
```

See also:

Python documentation: *RotationSegment*

`end_of_takeoff` segment

New in version 1.5.0.

This segment is used at the end of the takeoff phase, between lift off and before reaching the safety altitude. The target sets the safety altitude. Because this phase is quite dynamic, it is a good practice to lower the time step at least to 0.05s for a good accuracy on takeoff distance.

Example:

```
segment: end_of_takeoff
time_step: 0.05
target:
  delta_altitude:
    value: 35
    unit: ft
```

See also:

Python documentation: *EndOfTakeoffSegment*

`takeoff` segment

New in version 1.5.0.

This segment is the sequence of *ground_speed_change segment*, *rotation segment* and *end_of_takeoff segment*.

The parameters for this segment are the same as for its 3 components, except that:

- `time_step` is used only for *ground_speed_change segment* and *rotation segment*.
- time step for *end_of_takeoff segment* is driven by the additional parameter `end_time_step`
- target speed at end of *ground_speed_change segment* is driven by the additional parameter `rotation_equivalent_airspeed`
- the target of `takeoff` segment is the target of *end_of_takeoff segment*, meaning it sets the safety altitude.

Example:

```
segment: takeoff
wheels_friction: 0.03
rotation_equivalent_airspeed:
  value: data:mission:operational:takeoff:Vr
rotation_rate:
  value: 0.0523
  units: rad/s
rotation_alpha_limit:
  value: 0.3489
  units: rad
end_time_step: 0.05
target:
  delta_altitude:
    value: 35
    unit: ft
```

See also:

Python documentation: *TakeOffSequence*

Segment target

The target of a flight segment is a set of parameters that drives the end of the segment simulation.

Possible target parameters are the available fields of *FlightPoint*. The actually useful parameters depend on the segment.

Each parameter can be set the *usual way*, generally with a numeric value or a variable name, but it can also be a string. The most common string value is `constant` that tells the parameter value should be kept constant and equal to the start value. In any case, please refer to the documentation of the flight segment.

Absolute and relative values

Amost all target parameters are considered as absolute values, i.e. the target is considered reached if the named parameter gets equal to the provided value.

They can also be specified as relative values, meaning that the target is considered reached if the named parameter gets equal to the provided value **added** to start value. To do so, the parameter name will be preceded by `delta_`.

Examples:

```
target:
  altitude: # Target will be reached at 35000 ft.
    value: 35000
    unit: ft
```

```
target:
  delta_altitude: # Target will be 5000 ft above the start altitude of the segment.
    value: 5000
    unit: ft
```

Important: There are 2 exceptions : `ground_distance` and `time` are always considered as relative values. Therefore, `delta_ground_distance` and `delta_time` will have the same effect.

Special segment parameters

Most of segment parameters must be set with a unique value, which can be done in several ways, as described *here*.

There are some special parameters that are detailed below.

`engine_setting` parameter

Expected value for `engine_setting` are `takeoff`, `climb` , `cruise` or `idle`

This setting is used by the "rubber engine" propulsion model (see class RubberEngine). It roughly links the "turbine inlet temperature" (a.k.a. T4) to the flight conditions.

If another propulsion model is used, this parameter may become irrelevant, and then can be omitted.

`polar` parameter(s)

The aerodynamic polar defines the relation between lift and drag coefficients (respectively CL and CD). This parameter is composed of two vectors of same size, one for CL, and one for CD.

The `polar` parameter has 2 sub-keys that are `CL` and `CD`.

A basic example would be:

```
segment: cruise
polar:
  CL: [0.0, 0.5, 1.0]
  CD: [0.01, 0.03, 0.12]
```

But generally, polar values will be obtained through variable names, because they will be computed during the process, or provided in the input file. This should give:

```
segment: cruise
polar:
  CL: data:aerodynamics:aircraft:cruise:CL
  CD: data:aerodynamics:aircraft:cruise:CD
```

Additionally, a convenience feature is proposes, which assumes CL and CD are provided by variables with same names, except one ends with `:CL` and the other one by `:CD`. In such case, providing only the common prefix is enough.

Therefore, the next example is equivalent to the previous one:

```
segment: cruise
polar: data:aerodynamics:aircraft:cruise
```

Setting values in mission file

Any parameter value in the mission file can be provided in several ways:

- *hard-coded value and unit*
- *hard-coded value with no unit*
- *OpenMDAO variable*
- *Contextual OpenMDAO variable*

hard-coded value and unit

The standard way is to set the parameter as value, with or without unit.

Note: If no unit is provided while parameter needs one, SI units will be assumed.

Provided units have to match OpenMDAO convention.

Examples:

```
altitude:
    value: 10.
    unit: km
altitude:
    value: 10000.   # equivalent to previous one (10km), because SI units are assumed
mach:
    value: 0.8
engine_setting:
    value: takeoff   # some parameters expect a string value
```

hard-coded value with no unit

When no unit is provided, the value can be "inlined". As for *hard-coded value and unit*, if the concerned parameter is not dimensionless, SI units will be assumed.

Example:

```
mach: 0.8                   # no unit
altitude: 10000.            # == 10 km
engine_setting: takeoff     # string value
```

OpenMDAO variable

It is possible to provide a variable name instead of a hard-coded value. This variable will be declared as input of the OpenMDAO component.

Unit can be specified. In that case, it will be the unit for OpenMDAO declaration and usage. In any case, the unit for computation will be the internal unit of the segments (SI units). Conversion will be done when needed.

Also, a default value can be specified, which will be the declared default value for OpenMDAO. It has to be consistent with declared unit. If no default value is specified, numpy.nan will be used in OpenMDAO declaration.

Example:

```
altitude:
    value: data:dummy_category:some_altitude
    unit: ft
    default: 35000.0
```

As for numeric values, the definition can be inlined if no unit or default value has to be declared:

```
altitude: data:dummy_category:some_altitude
```

Using opposite value

Sometimes, it can be convenient to use the opposite value of a variable. It can be done by simply putting the minus sign "-" just before the variable:

```
delta_mass:
    value: -data:dummy_category:consumed_fuel
    unit: kg
    default: 125.0
```

Important: The specified default value is for the declared variable, even when the minus sign is used. Therefore, if default value is set as negative and the variable is preceded by a minus sign, the actually used value (if the default value is kept) will be positive.

Contextual OpenMDAO variable

By using the tilda (~) in the variable name, it is also possible to make the variable name contextual according to the hierarchy the defined parameter belongs to.

When a parameter value is defined as `prefix~suffix`, the actual variable name will be `prefix:<mission_name>:<route_name>:<phase_name>:suffix`.

It is useful when defining a route or a phase that will be used in several missions (see *Mission file*).

Note:

- `<route_name>` and `<phase_name>` will be used only when applicable (see examples below).
- A contextual variable can be defined in a segment, but the variable will still be "associated" only to the phase.

If no prefix is provided (`~suffix`), the default prefix will be `data:mission:`.

If no suffix is provided (`prefix~`), the default suffix will be the parameter name.

It is also possible to have no prefix nor suffix (~). Then the 2 rules above apply.

Example

```
routes:
  route_A:
    range: ~distance                # Example #1: here the suffix is customized.
    parts:
      - phase_a
      - ...

phases:
  phase_a:
```

(continues on next page)

(continued from previous page)

```
    thrust_rate: ~                  # Example #2: default prefix and suffix will be used
    time_step: settings:mission~    # Example #3: Here the prefix is customized

missions:
  mission_1:
    parts:
      - ...
      - route: route_A
      - ...
  mission_2:
    parts:
      - ...
      - phase: phase_a
      - ...
```

Example 1

`route_A` contains the parameter `range` where a contextual variable name is affected, that will use the default prefix (`data:mission:`) and the customized suffix (`distance`).

`route_A` is used as a step by both `mission_1` and `mission_2`.

Then the mission computation has among its inputs:

Table 2: Variable names

#	Prefix	Hierarchy	Suffix	Full name
1	data:mission	mission_1:route_A	distance	data:mission:mission_1:route_A:distance
1	data:mission	mission_2:route_A	distance	data:mission:mission_2:route_A:distance

Examples 2 & 3

`phase_a` contains the parameters `thrust_rate` and `time_step` where contextual variable names are affected. For `thrust_rate`, default prefix (`data:mission:`) and suffix (`thrust_rate`) will be used. For `time_step`, prefix is customized (`settings:mission`) and default suffix (`time_step`) will be used.

`phase_a` is used as a step by `route_A`, that is used as a step by `mission_1`. `phase_a` is also used as a step directly by `mission_2`.

Then the mission computation has among its inputs:

Table 3: Variable names

#	Prefix	Hierarchy	Suffix	Full name
2	data:mission	mission_1:route_A:phase_a	thrust_rate	data:mission:mission_1:route_A:phase_a:thrust_rate
2	data:mission	mission_2:phase_a	thrust_rate	data:mission:mission_2:phase_a:thrust_rate
3	data:settings	mission_1:route_A:phase_a	time_step	data:settings:mission_1:route_A:phase_a:time_step
3	data:settings	mission_2:phase_a	time_step	data:settings:mission_2:phase_a:time_step

Mission module

The FAST-OAD mission module allows to simulate missions and to estimate their fuel burn, which is an essential part of the sizing process.

The module aims at versatility, by:

- providing a way to define missions from *custom files*
- linking mission inputs and outputs to the FAST-OAD data model
- linking or not a mission to the sizing process

Inputs and outputs of the module

The performance module, as any FAST-OAD module, is linked to the MDA process by the connection of its input and output variables. But unlike other modules, the list of inputs and outputs is not fixed, and widely depends on the mission definition.

The input variables are defined in the mission file, as described *here*.

Most outputs variables are automatically decided by the structure of the mission. Distance, duration and fuel burn are provided as outputs for each part of the mission.

Outputs for the whole mission:

- `data:mission:<mission_name>:distance`
- `data:mission:<mission_name>:duration`
- `data:mission:<mission_name>:fuel`

Outputs for each part of the mission (*flight route* or *flight phase*):

- `data:mission:<mission_name>:<part_name>:distance`
- `data:mission:<mission_name>:<part_name>:duration`
- `data:mission:<mission_name>:<part_name>:fuel`

Outputs for each *flight phase* of a route:

- `data:mission:<mission_name>:<route_name>:<phase_name>:distance`
- `data:mission:<mission_name>:<route_name>:<phase_name>:duration`
- `data:mission:<mission_name>:<route_name>:<phase_name>:fuel`

Other mission-related variables are:

- `data:mission:<mission_name>:TOW`: TakeOff Weight. Input or output, depending on options below.
- `data:mission:<mission_name>:needed_block_fuel`: Burned fuel during mission. Output.
- `data:mission:<mission_name>:block_fuel`: Actual block fuel. Input or output, depending on options below.

Usage in FAST-OAD configuration file

The mission module can be used with the identifier :code`fastoad.performances.mission`.

The available parameters for this module are:

Detailed description of parameters

propulsion_id

- **Mandatory**

It is the identifier of a registered propulsion wrapper (see *How to add a custom propulsion model to FAST-OAD*).

FAST-OAD comes with a parametric propulsion model adapted to engine of the 1990s, with "`fastoad.wrapper.propulsion.rubber_engine`" as identifier.

mission_file_path

- Optional (Default = "`::sizing_mission`")

It is the path to the file that defines the mission. As any file path in the configuration file, it can be absolute or relative. If relative, the path of configuration file will be used as basis.

FAST-OAD comes with two embedded missions, usable with special values:

- "`::sizing_mission`": a time-step simulation of a classical commercial mission with diversion and holding phases
- "`::sizing_breguet`": a very quick simulation based on Breguet formula, with rough assessment of fuel consumption during climb, descent, diversion and holding phases.

out_file

- Optional

If provided, a CSV file will be written at provided path with all computed flight points.

If relative, the path of configuration file will be used as basis.

mission_name

- Mandatory if the used mission file defines several missions. Optional otherwise.

Sets the mission to be computed.

use_initializer_iteration

Optional (Default = `true`)

During first solver loop, a complete mission computation can fail or consume useless CPU-time. When activated, this option ensures the first iteration is done using a simple, dummy, formula instead of the specified mission.

Warning: Set this option to `false` if you do expect this model to be computed only once. Otherwise, the performance computation will be done only by the initializer.

adjust_fuel

- Optional (Default = `true`)

If `true`, block fuel will be adjusted to fuel consumption during mission. If `false`, the input block fuel will be used.

compute_TOW

- Optional (Default = `false`)
- Not used (actually forced to `true`) if `adjust_fuel` is `true`.

If `true`, TakeOff Weight will be computed from mission block fuel and ZFW.

If `false`, block fuel will be computed from TOW and ZFW.

add_solver

- Optional (Default = `false`)
- Not used (actually forced to `false`) if `compute_TOW` is `false`.

Setting this option to False will deactivate the local solver of the component. Useful if a global solver is used for the MDA problem.

is_sizing

- Optional (Default = `false`)

If `true`, TOW for the mission will be considered equal to MTOW and mission payload will be considered equal to design payload (variable `data:weight:aircraft:payload`). Therefore, mission computation will be linked to the sizing process.

Extensibility

In FAST-OAD mission module, *segments* are the base building blocks used in the *mission definition file*. They are implemented in Python and FAST-OAD offers a set of segment types that allows defining typical aircraft mission profiles.

Yet, the need for some other segment types may occur. This is why FAST-OAD mission module is designed so that any user can develop new segment types and use them in a custom mission file.

Adding segment types

In FAST-OAD mission module, *segments* are the base building blocks used in the *mission definition file*. They are implemented in Python and FAST-OAD offers a set of segment types that allows defining typical aircraft mission profiles.

Yet, the need for some other segment types may occur. This is why FAST-OAD mission module is designed so that any user can develop new segment types and use them in a custom mission file.

First of all, be aware that segment implementation relies on Python dataclasses. This chapter will assume you already know how it works.

Links between Python implementation and mission definition file

Flight segment classes must all derive from *AbstractFlightSegment*.

Segment keyword

When subclassing, a keyword is associated to the class:

```
import fastoad.api as oad
from dataclasses import dataclass

@dataclass
class NewSegment(oad.AbstractFlightSegment, mission_file_keyword="new_segment"):
    ...
```

As soon as your code is interpreted, the `mission_file_keyword` will be usable in mission definition file when specifying segments:

```
phases:
  some_phase:
    parts:
    - segment: taxi
      ...
    - segment: new_segment
      ...
```

Note: Where to put the code for a new segment implementations?

Having your class in any imported Python module will do.

If you use FAST-OAD through Python, you are free to put your new segment classes where it suits you.

Also, know that FAST-OAD will make Python interpret any Python module in the *module folders you declare in the configuration file*. This works also for *declared plugins*. In both cases, it is not mandatory to add custom FAST-OAD modules.

Segment parameters

The other strong link between segment implementation and the mission definition file is that any dataclass field of the defined segment class will be available as parameter in the mission definition file.

Given this implementation:

```
import fastoad.api as oad
from dataclasses import dataclass, field
from typing import List

@dataclass
class NewSegment(oad.AbstractFlightSegment, mission_file_keyword="new_segment"):
    my_float: float = 0.0
    my_bool: bool = True
```

(continues on next page)

(continued from previous page)

```
    my_array: List[float] = field(default_factory=list)
    ...
```

... the mission definition file will accept the following implementation:

```
phases:
  some_phase:
    parts:
    - segment: new_segment
      my_float: 50.0
      my_bool: false
      my_array: [10.0, 20.0, 30.0]
      target:
        ...
```

Note: Defining mandatory parameters

It is possible to declare a segment parameter as mandatory (i.e. without associated default value) by using `fastoad.api.MANDATORY_FIELD`:

```
import fastoad.api as oad
from dataclasses import dataclass

@dataclass
class NewSegment(oad.AbstractFlightSegment, mission_file_keyword="new_segment"):
    my_mandatory_float: float = oad.MANDATORY_FIELD
    ...
```

This is a way to work around the fact that if a dataclass defines a field with a default value, inheritor classes will not be allowed to define fields without default value, because then the non-default fields will follow a default field, which is forbidden.

Implementation of a segment class

The AbstractFlightSegment class

As *previously said*, a segment class has to inherit from *`AbstractFlightSegment`* (and specify the *mission_file_keyword* if its usage is intended in mission definition files) and will be implemented like this:

```
import fastoad.api as oad
from dataclasses import dataclass, field
from typing import List

@dataclass
class NewSegment(oad.AbstractFlightSegment, mission_file_keyword="new_segment"):
    my_float: float = 0.0
    ...
```

The main field of the class will be *`target`*, provided as a *FlightPoint instance*, which will contain the flight point parameters set as target in the mission definition file.

The instantiation in FAST-OAD will be like this:

```
import fastoad.api as oad

segment = NewSegment( target=oad.FlightPoint(altitude=5000.0, true_airspeed=200.0),
                      my_float=4.2,
                     ...
                    )
```

Note: Instantiation arguments will always be passed as keyword arguments (this behavior can be enforced only for Python 3.10+).

The new class will have to implement the method *compute_from_start_to_target()* that will be in charge of computing the flight points between a provided *start* and a provided *target* (providing the result as a pandas DataFrame)

Note: The mission computation will actually call the method *compute_from()*, that will do the computation between provided *start* and the target defined at instantiation (i.e. in the mission definition file).

This method does some generic pre-processing of start and target before calling *compute_from_start_to_target()*. Therefore, in the vast majority of cases, implementing the latter will be the correct thing to do.

The AbstractTimeStepFlightSegment class

AbstractTimeStepFlightSegment is a base class for segments that do time step computations.

This class has 4 main additional fields:

- *propulsion*, that is expected to be an *IPropulsion* instance.
- *polar*, that is expected to be a *Polar* instance.
- *reference_area*, that provides the reference surface area consistently with provided aerodynamic polar.
- *time_step*, that sets the time step for resolution. It is set with a low enough default value.

An inheritor class will have to provide the implementations for 3 methods that are used at each computed time step: *get_distance_to_target()*, *compute_propulsion()* and *get_gamma_and_acceleration()*. (see each method documentation for more information)

There are some specialized base classes that provide a partial implementation of *AbstractTimeStepFlightSegment*:

- *AbstractManualThrustSegment* implements *compute_propulsion()*. It has its own field, *thrust_rate*, that is used to compute thrust.
- *AbstractRegulatedThrustSegment* also implements *compute_propulsion()*, but it adjusts the thrust rate to have aircraft thrust equal to its drag.
- *AbstractFixedDurationSegment* implements *get_distance_to_target()*. It allows to compute a segment with a time duration set by the target.

The FlightPoint class

The *FlightPoint* class is designed to store flight parameters for one flight point of any computed mission.

FlightPoint class is meant for:

- **storing all needed parameters** that are needed for performance modelling, including propulsion parameters.
- easily exchanging data with **pandas DataFrame**.
- **being extensible** for new parameters.

Note: All parameters in FlightPoint instances are expected to be in SI units.

Available flight parameters

The documentation of *FlightPoint* provides the list of available flight parameters, available as attributes. As FlightPoint is a dataclass, this list is available through Python using:

```
>>> import fastoad.api as oad
>>> from dataclasses import fields

>>> [f.name for f in fields(oad.FlightPoint)]
```

Exchanges with pandas DataFrame

A pandas DataFrame can be generated from a list of FlightPoint instances:

```
>>> import pandas as pd
>>> import fastoad.api as oad

>>> fp1 = oad.FlightPoint(mass=70000., altitude=0.)
>>> fp2 = oad.FlightPoint(mass=60000., altitude=10000.)
>>> df = pd.DataFrame([fp1, fp2])
```

And FlightPoint instances can be created from DataFrame rows:

```
# Get one FlightPoint instance from a DataFrame row
>>> fp1bis = oad.FlightPoint.create(df.iloc[0])

# Get a list of FlightPoint instances from the whole DataFrame
>>> flight_points = oad.FlightPoint.create_list(df)
```

Extensibility

FlightPoint class is bundled with several fields that are commonly used in trajectory assessment, but one might need additional fields.

Python allows to add attributes to any instance at runtime, but for FlightPoint to run smoothly, especially when exchanging data with pandas, you have to work at class level. This can be done using *add_field()*, preferably outside of any class or function:

```
# Adds a float field with None as default value
>>> FlightPoint.add_field("ion_drive_power")

# Adds a field and define its type and default value
>>> FlightPoint.add_field("warp", annotation_type=int, default_value=9)

# Now these fields can be used at instantiation
>>> fp = FlightPoint(ion_drive_power=110.0, warp=12)

# Removes a field, even an original one (useful only to avoid having it in outputs)
>>> FlightPoint.remove_field("sfc")
```

1.5.6 Adding modules to FAST-OAD

Here you will find information about custom modules in FAST-OAD.

How to add custom OpenMDAO modules to FAST-OAD

With FAST-OAD, you can register any OpenMDAO system of your own so it can be used through the configuration file.

It is therefore strongly advised to have at least a basic knowledge of OpenMDAO to develop a module for FAST-OAD.

To have your OpenMDAO system available as a FAST-OAD module, you should follow these steps:

- *Create your OpenMDAO system*
- *Register your system(s)*
- *Modify the configuration file*

Create your OpenMDAO system

It can be a Group or a Component-like class (generally an ExplicitComponent).

You can create the Python file at the location of your choice. You will just have to provide later the folder path in FAST-OAD configuration file (see *Modify the configuration file*).

Variable naming

You have to pay attention to the naming of your input and output variables. As FAST-OAD uses the promotion system of OpenMDAO, which means that variables you want to link to the rest of the process must have the name that is given in the global process.

Nevertheless, you can create new variables for your system:

- Outputs of your system will be available in output file and will be usable as any other variable.
- Unconnected inputs will simply have to be in the input file of the process. They will be automatically included in the input file generated by FAST-OAD (see *How to generate an input file*).
- And if you add more than one system to the FAST-OAD process, outputs created by one of your system can of course be used as inputs by other systems.

Also keep in mind that the naming of your variable will decide of its location in the input and output files. Therefore, the way you name your new variables should be consistent with FAST-OAD convention, as explained in *Problem variables*.

Defining options

You may use the OpenMDAO way for adding options to your system. The options you add will be accessible from the FAST-OAD configuration file (see *Problem definition*).

When declaring an option, the usage of the `desc` field if strongly advised, as any description you provide will be printed along with module information with the `list_modules` sub-command (see *How to get list of registered modules*).

Definition of partial derivatives

Your OpenMDAO system is expected to provide partial derivatives for all its outputs in analytic or approximate way.

At the very least, for most Component classes, the `setup()` method of your class should contain:

```
self.declare_partials("*", "*", method='fd')
```

or for a Group class:

```
self.approx_totals()
```

The two lines above are the most generic and the least CPU-efficient ways of declaring partial derivatives. For better efficiency, see how to work with derivatives in OpenMDAO.

About ImplicitComponent classes

In some cases, you may have to use ImplicitComponent classes.

Just remember, as told in this tutorial, that the loop that will allow to solve it needs usage of the NewtonSolver.

A good way to ensure it is to build a Group class that will solve the ImplicitComponent with NewtonSolver. This Group should be the system you will register in FAST-OAD.

Checking validity domains

Generally, models are valid only when variable values are in given ranges.

OpenMDAO provides a way to specify lower and upper bounds of an output variable and to enforce them when using a Newton solver by using backtracking line searches.

FAST-OAD proposes a way to set lower and upper bounds for input and output variables, but only for checking and giving feedback of variables that would be out of bounds.

If you want your OpenMDAO class to do this checking, simply use the decorator ValidityDomainChecker:

```
@ValidityDomainChecker
class MyComponent(om.ExplicitComponent):
    def setup(self):
        self.add_input("length", 1., units="km" )
        self.add_input("time", 1., units="h" )
        self.add_output("speed", 1., units="km/h", lower=0., upper=130.)
```

The above code make that FAST-OAD will issue a warning if at the end of the computation, "speed" variable is not between lower and upper bound.

But it is possible to set your own bounds outside of OpenMDAO by following this example:

```
@ValidityDomainChecker(
    {
        "length": (0.1, None),  # Defines only a lower bound
        "time": (0., 1.),  # Defines lower and upper bounds
        "speed": (None, 150.0),  # Ignores original bounds and sets only upper bound
    }
)
class MyComponent(om.ExplicitComponent):
    def setup(self):
        self.add_input("length", 1., units="km" )
        self.add_input("time", 1., units="h" )
        # Bounds that are set here will still apply if backtracking line search is used,␣
↪but
        # will not be used for validity domain checking because it has been replaced␣
↪above
        self.add_output("speed", 1., units="km/h", lower=0., upper=130.)
```

Register your system(s)

Once your OpenMDAO system is ready, you have to register it to make it known as a FAST-OAD module.

To do that, you just have to add the *RegisterOpenMDAOSystem* decorator to your OpenMDAO class like this:

```
import fastoad.api as oad
import openmdao.api as om

@oad.RegisterOpenMDAOSystem("my.custom.name")
class MyOMClass(om.ExplicitComponent):
    [ ... ]
```

Note: If you work with Jupyter notebook, remember that any change in your Python files will require the kernel to be restarted.

Modify the configuration file

The folders that contain your Python files must be listed in `module_folders` in the *FAST-OAD configuration file*:

```
title: OAD Process with custom component

# List of folder paths where user added custom registered OpenMDAO components
module_folders:
  - /path/to/my/custom/module/folder
  - /another/path/

[ ... ]
```

Once this is done, (assuming your configuration file is named `my_custom_conf.yml`) your custom, registered, module should appear in the list provided by the command line:

```
$ fastoad list_modules my_custom_conf.yml
```

Then your component can be used like any other using the id you have given.

```
# Definition of OpenMDAO model
model:
  [ ... ]

  my_custom_model:
    id: "my.custom.name"

  [ ... ]
```

Note: FAST-OAD will inspect all sub-folders in a specified module folder, **as long as they are Python packages**, i.e. if they contain a `__init__.py` file.

How to add a custom propulsion model to FAST-OAD

Propulsion models have a specific status because they are directly called by the performance models, so the connection is not done through OpenMDAO.

By following instructions in this page, you should ensure your propulsion model will run smoothly with the existing performance models. You will also be able to access your engine parameters through FAST-OAD process.

The IPropulsion interface

When developing your propulsion model, to ensure that it will work smoothly with current performances models, you have to do it in a class that implements the *IPropulsion* interface, meaning your class must have at least the 2 methods *compute_flight_points()* and *get_consumed_mass()*.

Computation of propulsion data

compute_flight_points() will modify the provided flight point(s) by adding propulsion-related parameters. A conventional fuel engine will rely on parameters like `mach`, `altitude` and will provide parameters like `sfc` (Specific Fuel Consumption).

Propulsion model inputs

For your model to work with current performance models, your model is expected to rely on known flight parameters, i.e. the original parameters of *FlightPoint*.

See *The FlightPoint class* for more details.

Note: Special attention has to be paid to the **thrust parameters**. Depending on the flight phase, the aircraft can fly in **manual** mode, with an imposed thrust rate, or in **regulated** mode, where propulsion has to give an imposed thrust. Your model has to provide these two modes, and to use them as intended.

The `thrust_is_regulated` parameter tells what mode is on. If it is True, the model has to rely on the `thrust` parameter. If it False, the model has to rely on the `thrust_rate` parameter.

Propulsion model outputs

If you work with the Breguet module, your model has to compute the `sfc` parameter.

But if you use the mission module, you have total freedom about the output of your model. If you want to use a parameter that is not available, you can add it to the FlightPoint class as described *above*.

The only requirement is that you have to implement *get_consumed_mass()* accordingly for the mission module to have a correct assessment of mass evolution.

Computation of consumed mass

The *get_consumed_mass()* simply provides the mass consumption over the provided time. It is meant to use the parameters computed in *compute_flight_points()*.

The OpenMDAO wrapper

Once your propulsion model is ready, you have to make a wrapper around it for:

- having the possibility to choose it in the FAST-OAD configuration file
- having its parameters available in FAST-OAD data files

Defining the wrapper

Your wrapper class has to implement the *IOMPropulsionWrapper* interface, meaning it should implement the 2 methods *get_model()* and *setup()*.

get_model() has to provide an instance of your model. If the constructor of your propulsion model class needs parameters, you may get them from `inputs`, that will be the `inputs` parameter that OpenMDAO will provide to the performance module when calling `compute()` method.

Therefore, the performance module will have to define the inputs that your propulsion model needs in its `setup` method, as required by OpenMDAO. To do this, the `setup` method ot the performance module calls the *setup()* of your wrapper, that is expected to define the needed input variables.

For an example, please see the source code of `OMRubberEngineWrapper`.

Registering the wrapper

Registering is needed for being able to choose your propulsion wrapper in FAST-OAD configuration file. Due to the specific status of propulsion models, the registering process is a bit different that *the one for classic OpenMDAO modules*.

The registering is done using the *RegisterPropulsion* decorator:

```
import fastoad.api as oad


@oad.RegisterPropulsion("star.trek.propulsion")
class WarpDriveWrapper(oad.IOMPropulsionWrapper):

    [ ... ]
```

Using the wrapper in the configuration file

As for *other custom modules*, the folder that contains your Python module(s) must be listed in the `module_folders` of the configuration file.

The association of the propulsion model to the performance module is done with the *propulsion_id* keyword, as in following example:

```
title: OAD Process with custom propulsion model

# List of folder paths where user added custom registered OpenMDAO components
module_folders:
  - /path/to/my/propulsion/wrapper/
```

(continues on next page)

(continued from previous page)

```
[ ... ]

# Definition of OpenMDAO model
model:
  [ ... ]
  performance:
    id: fastoad.performances.mission
    propulsion_id: star.trek.propulsion
```

How to document your variables

FAST-OAD can associate a description to each variable. Such description will be put as comment in datafiles, or displayed along with other variable information, like in command line (see *How to get list of variables*).

The description of a variable can be defined in two ways:

- *Defining variable description in your OpenMDAO component*
- *Defining variable description in dedicated files*

Defining variable description in your OpenMDAO component

OpenMDAO natively allows to define the description of a variable when declaring it.

FAST-OAD will retrieve this information (the description has to be defined once, even if the variable is declared at several locations).

Defining variable description in dedicated files

If you want to add description to your variables in a more centralized way, FAST-OAD will look for files named `variable_descriptions.txt` that are dedicated to that.

The file content is expected to process one variable per line, containing the variable name and its description, separated by `||`, as in following example:

```
my:variable||The description of my:variable, as long as needed, but on one line.
# Comments are allowed
my:other:variable || Another description (surrounding spaces are ignored)
```

FAST-OAD will search such files:

- in the root package of plugin modules (see *How to add custom OpenMDAO modules to FAST-OAD as a plugin*)
- in the root folder of module folders as declared in configuration file (see *Modify the configuration file*)
- in the same package as any class which is declared as FAST-OAD module (see *Register your system(s)*)

In practice, here you can see what description files will be consider, depending on their location:

```
my_modules/
├── __init__.py
├── subpackage1
```

(continues on next page)

(continued from previous page)

```
    ├── __init__.py
    ├── model.py                        <- contains a class decorated with
    │                                      RegisterOpenMDAOSystem
    └── variable_descriptions.txt       <- this file will be loaded
├── subpackage2
    ├── __init__.py
    ├── propulsion_model.py             <- contains a class decorated with
    │                                      RegisterOpenPropulsion
    └── variable_descriptions.txt       <- this file will be loaded
├── util
    ├── __init__.py
    ├── utility_module.py               <- no registering done here
    └── variable_descriptions.txt       <- this file will NOT be loaded
└── variable_descriptions.txt       <- this file will be loaded because it is in root␣
↪folder/package
```

How to add custom OpenMDAO modules to FAST-OAD as a plugin

Once you have *created your custom modules* for FAST-OAD, you may want to share them with other users, which can be done in two ways:

- Providing your code so they can copy it on their computer and have them set their `custom_modules` field accordingly in their *FAST-OAD configuration file*.
- Packaging your code as a FAST-OAD plugin and have them install it through `pip` or equivalent. This is the subject of current chapter.

A FAST-OAD plugin can provide additional FAST-OAD modules, Jupyter notebooks, configuration files and source data files:

- plugin-provided FAST-OAD modules are usable in *configuration files*, and can be *listed* and *used* in the same way as native modules.
- Command line can be used by users to retrieve *notebooks*, *configuration files* and *source data files*.

Plugin structure

In your source folder, a typical plugin structure would be like this:

```
my_package/
├── __init__.py
├── configurations/
    ├── __init__.py
    ├── configuration_1.yaml
    └── configuration_2.yaml
├── models/
    ├── __init__.py
    ├── my_model.py
    └── some_subpackage/
        ├── __init__.py
        └── some_more_code.py
├── notebooks/
```

(continues on next page)

(continued from previous page)

```
│   ├── __init__.py
│   ├── any_data/
│   │   ├── __init__.py
│   │   └── some_data.xml
│   ├── awesome_notebook.ipynb
│   └── good_notebook.ipynb
└── source_data_files
    ├── __init__.py
    ├── source_data_file_1.xml
    ├── source_data_file_2.xml
    └── source_data_file_3.xml
```

As shown above, the expected structure is composed of Python **packages**, i.e. every folder should contain a `__init__.py` file, **even if it contains only non-Python files** (e.g. data for notebooks).

The root folder can be anywhere in your project structure, since plugin declaration will point to its location.

Expected folders in a plugin package are:

- `models`: contains Python code where FAST-OAD modules are *registered*.
- `configurations`: contains only configuration files in YAML format. No sub-folder is allowed. These configuration files will be usable through *command line* or API method *`generate_configuration_file()`*.
- `notebooks`: contains any number of Jupyter notebooks and associated data, that will be made available to users through *command line*.
- `source_data_files`: contains only source data files in XML format. As for the `configurations` package, no sub-folder is allowed. These source data files will be usable through *command line* or API method *`generate_source_data_file()`*.

Any of these folders is optional. Any other folder will be ignored.

Plugin packaging

To make your custom modules usable as a FAST-OAD plugin, you have to package them and declare your package as a plugin with `fastoad.plugins` as plugin group name.

Here under is a brief tutorial about these operations using Poetry.

Note: If you are not familiar with Python packaging, it is recommended to look at this tutorial first. It presents the important steps and notions of the packaging process, and the "classic" way using setuptools. And if you want to stick to setuptools, check this page for details about plugin declaration.

Plugin declaration

For the example, let's consider that your project contains the package `star_trek.drives`, and that your project structure contains:

```
src/
├── star_trek/
│   ├── __init__.py
│   ├── drives/
│   │   ├── __init__.py
│   │   ├── configurations/
│   │   ├── models/
│   │   └── notebooks/
│   └── ...
└── ...
```

As previously stated, your folder `src/star_trek/drives` does not have to contain all of the folders `models`, `configurations`, `notebooks` nor `source_data_files`.

Assuming you project contains the package `star_trek.drives` that contains models you want to share, you can declare your plugin in your `pyproject.toml` file with:

```
...

[tool.poetry]
# Tells location of sources
packages = [
    { include = "star_trek", from = "src" },
]

...

# Plugin declaration
[tool.poetry.plugins."fastoad.plugins"]
"ST_plugin" = "star_trek.drives"

...
```

Note: It is discouraged to declare several FAST-OAD plugins for a same project.

Once your `pyproject.toml` is set, you can do `poetry install`. Besides installing your project dependencies, it will make your models **locally** available (i.e. you could use their identifiers in your FAST-OAD configuration file without setting the `custom_modules` field)

Building

You can build your package with the command line `poetry build`. Let's assume your `pyproject.toml` file is configured so that your project name is `ST_drive_models`, as below:

```
...

[tool.poetry]
name = "ST_drive_models"
version = "1.0.0"

# Tells location of sources
packages = [
    { include = "star_trek", from = "src" },
]

...

# Specify that Poetry is used for building the package
[build-system]
requires = ["poetry-core>=1.0.0"]
build-backend = "poetry.core.masonry.api"

...

# Plugin declaration
[tool.poetry.plugins."fastoad.plugins"]
"ST_plugin" = "star_trek.drives"
...
```

The command `poetry build` will create a `dist` folder with two files:

`ST_drive_models-1.0.0.tar.gz` and `ST_drive_models-1.0.0-py3-none-any.whl` (or something like this).

You may then have sent any of those two files to another user, who may then install your models using `pip` with:

```
$ pip install ST_drive_models-1.0.0-py3-none-any.whl  # or ST_drive_models-1.0.0.tar.gz
```

Publishing

Once you have built your package, you may publish it on a a package repository. `poetry publish` will publish your package on PyPI, provided that you have correctly set your account.

Note: Publishing on PyPI requires a valid account, and also that the chosen package name (defined by *name* field in the *pyproject.toml* file) is unused, or already associated to your account.

Poetry can also publish to another destination.

Please see here for detailed information.

Submodels in FAST-OAD

> **Warning:** Submodel feature is still considered as experimental.
>
> It as a feature for advanced users that want to replace a specific part of an existing FAST-OAD modules. At the very minimum, it needs a good understanding of the existing module because the developer is left with the responsibility to define a submodel that will work correctly in place of the original one.

Why submodels ?

FAST-OAD modules are generally associated to a discipline, and do all the related computations. For example, the native weight module computes the masses and the centers of gravity of each aircraft part and of the whole aircraft.

Now, let's say we want to modify the computation of wing mass. Then, we could add a new weight module where the only difference will be in the wing mass computation. This is not satisfactory because it would makes us copy all the code that is not related to wing mass.

To solve this problem, one solution would be to make smaller, more specific modules, and have them assembled in the configuration file. But it would result in very complex configuration files, and we do not want that.

There comes the principle of submodels. By using the *RegisterSubmodel* class in a FAST-OAD module, it is possible to allow some parts of the model to be changed later by a declared submodel.

How to use submodels in a custom module ?

Let's consider you want to build a custom module that will compute the number of atoms in the fuselage and the wing (don't ask me why you would do that, it is just an assumption).

You would begin by creating two `om.ExplicitComponent` classes: `CountWingAtoms` and `CountFuselageAtoms`. Then you would create the `om.Group` class that will be the registered FAST-OAD module. The Python code would look like:

```
import openmdao.api as om
import fastoad.api as oad

class CountWingAtoms(om.ExplicitComponent):
    """Put any implementation here"""

class CountFuselageAtoms(om.ExplicitComponent):
    """Put any implementation here"""

class CountEmpennageAtoms(om.ExplicitComponent):
    """Put any implementation here"""

@oad.RegisterOpenMDAOSystem("count.atoms")
class CountAtoms(om.Group):
    def setup(self):
        wing_component = CountWingAtoms()
        fuselage_component = CountFuselageAtoms()
        empennage_component = CountEmpennageAtoms()
        self.add_subsystem("wing", wing_component, promotes=["*"])
        self.add_subsystem("fuselage", fuselage_component, promotes=["*"])
        self.add_subsystem("empennage", empennage_component, promotes=["*"])
```

In the above implementation, someone that would want to provide an alternate method to count atoms in the wing, while keeping your method for fuselage, would have to provide its own FAST-OAD module, ideally by reusing your `CountFuselageAtoms` class, but possibly by needlessly copying it in its own code.

To allow a simpler replacement of your submodels, you will need to use the `RegisterSubmodel` class like this:

```
import openmdao.api as om
import fastoad.api as oad

WING_ATOM_COUNTER = "atom_counter.wing"
FUSELAGE_ATOM_COUNTER = "atom_counter.fuselage"
EMPENNAGE_ATOM_COUNTER = "atom_counter.empennage"

@oad.RegisterSubmodel(WING_ATOM_COUNTER, "original.counter.wing)
class CountWingAtoms(om.ExplicitComponent):
    """Put any implementation here"""

@oad.RegisterSubmodel(FUSELAGE_ATOM_COUNTER, "original.counter.fuselage)
class CountFuselageAtoms(om.ExplicitComponent):
    """Put any implementation here"""

@oad.RegisterSubmodel(EMPENNAGE_ATOM_COUNTER, "original.counter.empennage)
class CountEmpennageAtoms(om.ExplicitComponent):
    """Put any implementation here"""

@oad.RegisterOpenMDAOSystem("count.atoms")
class CountAtoms(om.Group):
    def setup(self):
        wing_component = oad.RegisterSubmodel.get_submodel(WING_ATOM_COUNTER)
        fuselage_component = oad.RegisterSubmodel.get_submodel(FUSELAGE_ATOM_COUNTER)
        empennage_component = oad.RegisterSubmodel.get_submodel(EMPENNAGE_ATOM_COUNTER)
        self.add_subsystem("wing", wing_component, promotes=["*"])
        self.add_subsystem("fuselage", fuselage_component, promotes=["*"])
        self.add_subsystem("empennage", empennage_component, promotes=["*"])
```

This has the same behavior as the previous one, but the second one will allow substitution of submodels, as shown in next part.

In details, `CountWingAtoms` is declared as a submodel that fulfills the role of "wing atom counter", identified by the `"atom_counter.wing"` (that is put in constant :code:`WING_ATOM_COUNTER`to avoid typos, as it is used several times). The same applies to the roles of "fuselage atom counter" and "empennage atom counter".

In the `CountAtoms` class, the line `oad.RegisterSubmodel.get_submodel(WING_ATOM_COUNTER)` expresses the **requirement** of getting a submodel that counts wing atoms.

Important: As long as only one declared submodel fulfills a requirement, the above instruction will be enough to provide it.

See below how to manage several "concurrent" submodels.

How to declare a custom submodel ?

As you have seen, we have already declared submodels in our previous custom module. The process for providing an alternate submodel is identical:

```
import openmdao.api as om
import fastoad.api as oad


@oad.RegisterSubmodel("atom_counter.wing", "alternate.counter.wing")
class CountWingAtoms(om.ExplicitComponent):
    """Put another implementation here"""
```

At this point, there are now 2 available submodels for the "atom_counter.wing" requirement. If we do nothing else, the command `oad.RegisterSubmodel.get_submodel("atom_counter.wing")` will raise an error because FAST-OAD needs to be instructed which submodel to use.

How to select submodels

There are two ways to specify which submodel has to be used when several ones fulfill a given requirement:

- *Using configuration file (recommended)*
- *Using Python*

Using configuration file (recommended)

The recommended way to select submodels is to use FAST-OAD configuration files.

Note: When it comes to the specification of selected submodels, the configuration file will have the priority over *Python instructions*.

The configuration file can be populated with a specific section that will state the submodels that should be chosen.

```
submodels:
    atom_counter.wing: alternate.counter.wing
    atom_counter.fuselage: original.counter.fuselage
```

In the above example, an alternate submodel is chosen for the "atom_counter.wing" requirement, whereas the original submodel is chosen for the "original.counter.fuselage" requirement (whether there is another one defined or not). No submodel is defined for the "atom_counter.empennage" requirement. It will be OK if only one submodel is available for this requirement. Otherwise, an error will be raised, unless the submodel choice is done through Python (see below).

Using Python

The second way to select submodels is to use Python.

You may insert the following line at module level (i.e. **NOT in any class or function**):

```
import fastoad.api as oad

oad.RegisterSubmodel.active_models["atom_counter.wing"] = "alternate.counter.wing"
```

> **Warning:** In case several Python modules define their own chosen submodel for the same requirement, the last interpreted line will preempt, which is not a reliable way to do.
>
> Therefore, this should be reserved to your tests.
>
> If you plan to provide your submodels to other people, it is recommended to avoid specifying the used submodel through Python and let them manage that through their configuration file.

Deactivating a submodel

It is also possible to deactivate a submodel:

From the configuration file, it can be done with:

```
submodels:
    atom_counter.wing: null  # The empty string "" is also possible
```

From Python, it can be done with:

```
import fastoad.api as oad

oad.RegisterSubmodel.active_models["atom_counter.wing"] = None  # The empty string "" is␣
↪also possible
```

Then nothing will be done when the `"atom_counter.wing"` submodel will be called. Of course, one has to correctly know which variables will be missing with such setting and what consequences it will have on the whole problem.

1.6 fastoad

1.6.1 fastoad package

Subpackages

fastoad.cmd package

Subpackages

Submodules

fastoad.cmd.api module

API

class fastoad.cmd.api.**UserFileType**(*value*)

Bases: enum.Enum

An enumeration.

CONFIGURATION = 'configuration'

SOURCE_DATA = 'source_data'

fastoad.cmd.api.**get_plugin_information**(*print_data=False*) → Dict[str, fastoad.module_management._plugins.DistributionPluginDefinition]

Provides information about available FAST-OAD plugins.

Parameters **print_data** – if True, plugin data are displayed.

Returns a dict with installed package names as keys and matching FAST-OAD plugin definitions as values.

fastoad.cmd.api.**generate_notebooks**(*destination_path: str*, *overwrite: bool = False*, *distribution_name=None*)

Copies notebook folder(s) from available plugin(s).

Parameters

- **destination_path** – the inner structure of the folders will depend on the number of installed package and the number of plugins they contain.
- **overwrite** – if True and *destination_path* exists, it will be removed before writing.
- **distribution_name** – the name of an installed package that provides notebooks

fastoad.cmd.api.**generate_configuration_file**(*configuration_file_path: str*, *overwrite: bool = False*, *distribution_name=None*, *sample_file_name=None*)

Copies a sample configuration file from an available plugin.

Parameters

- **configuration_file_path** – the path of file to be written
- **overwrite** – if True, the file will be written, even if it already exists
- **distribution_name** – the name of the installed package that provides the sample configuration file (can be omitted if only one plugin is available)
- **sample_file_name** – the name of the sample configuration file (can be omitted if the plugin provides only one configuration file)

Returns path of generated file

Raises *FastPathExistsError* – if overwrite==False and configuration_file_path already exists

fastoad.cmd.api.**generate_source_data_file**(*source_data_file_path: str*, *overwrite: bool = False*, *distribution_name=None*, *sample_file_name=None*)

Copies a sample source data file from an available plugin.

Parameters

- **source_data_file_path** – the path of file to be written
- **overwrite** – if True, the file will be written, even if it already exists

- **distribution_name** – the name of the installed package that provides the sample source data file (can be omitted if only one plugin is available)
- **sample_file_name** – the name of the sample source data file (can be omitted if the plugin provides only one source data file)

Returns path of generated file

Raises *FastPathExistsError* – if overwrite==False and source_file_path already exists

fastoad.cmd.api.**generate_inputs**(*configuration_file_path: str*, *source_data_path: Optional[str] = None*, *source_data_path_schema='native'*, *overwrite: bool = False*) → str

Generates input file for the problem specified in configuration_file_path.

Parameters

- **configuration_file_path** – where the path of input file to write is set
- **source_data_path** – path of source data file data will be taken from
- **source_data_path_schema** – set to 'legacy' if the source file come from legacy FAST
- **overwrite** – if True, file will be written even if one already exists

Returns path of generated file

Raises *FastPathExistsError* – if overwrite==False and configuration_file_path already exists

fastoad.cmd.api.**list_variables**(*configuration_file_path: str*, *out: Optional[Union[IO, str]] = None*, *overwrite: bool = False*, *force_text_output: bool = False*, *tablefmt: str = 'grid'*)

Writes list of variables for the problem specified in configuration_file_path.

List is generally written as text. It can be displayed as a scrollable table view if: - function is used in an interactive IPython shell - out == sys.stdout - force_text_output == False

Parameters

- **configuration_file_path** –
- **out** – the output stream or a path for the output file (None means sys.stdout)
- **overwrite** – if True and out parameter is a file path, the file will be written even if one already exists
- **force_text_output** – if True, list will be written as text, even if command is used in an interactive IPython shell (Jupyter notebook). Has no effect in other shells or if out parameter is not sys.stdout
- **tablefmt** – The formatting of the requested table. Options are the same as those available to the tabulate package. See tabulate.tabulate_formats for a complete list. If "var_desc" the file will use the variable_descriptions.txt format.

Returns path of generated file, or None if no file was generated.

Raises *FastPathExistsError* – if *overwrite==False* and *out* is a file path and the file exists

fastoad.cmd.api.**list_modules**(*source_path: Optional[Union[List[str], str]] = None*, *out: Optional[Union[IO, str]] = None*, *overwrite: bool = False*, *verbose: bool = False*, *force_text_output: bool = False*)

Writes list of available systems. If source_path is given and if it defines paths where there are registered systems, they will be listed too.

Parameters

- **source_path** – either a configuration file path, folder path, or list of folder path

- **out** – the output stream or a path for the output file (None means sys.stdout)
- **overwrite** – if True and out is a file path, the file will be written even if one already exists
- **verbose** – if True, shows detailed information for each system if False, shows only identifier and path of each system
- **force_text_output** – if True, list will be written as text, even if command is used in an interactive IPython shell (Jupyter notebook). Has no effect in other shells or if out parameter is not sys.stdout

Returns path of generated file, or None if no file was generated.

Raises ***FastPathExistsError*** – if *overwrite*==*False* and *out* is a file path and the file exists

fastoad.cmd.api.**write_n2**(*configuration_file_path: str*, *n2_file_path: Optional[str] = None*, *overwrite: bool = False*)

Write the N2 diagram of the problem in file n2.html

Parameters

- **configuration_file_path** –
- **n2_file_path** – if None, will default to *n2.html*
- **overwrite** –

Returns path of generated file.

Raises ***FastPathExistsError*** – if overwrite==False and n2_file_path already exists

fastoad.cmd.api.**write_xdsm**(*configuration_file_path: str*, *xdsm_file_path: Optional[str] = None*, *overwrite: bool = False*, *depth: int = 2*, *wop_server_url: Optional[str] = None*, *dry_run: bool = False*)

Parameters

- **configuration_file_path** –
- **xdsm_file_path** – the path for HTML file to be written (will overwrite if needed)
- **overwrite** – if False, will raise an error if file already exists.
- **depth** – the depth analysis for WhatsOpt
- **wop_server_url** – URL of WhatsOpt server (if None, ether.onera.fr/whatsopt will be used)
- **dry_run** – if True, will run wop without sending any request to the server. Generated XDSM will be empty. (for test purpose only)

Returns path of generated file.

Raises ***FastPathExistsError*** – if overwrite==False and xdsm_file_path already exists

fastoad.cmd.api.**evaluate_problem**(*configuration_file_path: str*, *overwrite: bool = False*) → *fastoad.openmdao.problem.FASTOADProblem*

Runs model according to provided problem file

Parameters

- **configuration_file_path** – problem definition
- **overwrite** – if True, output file will be overwritten

Returns the OpenMDAO problem after run

Raises ***FastPathExistsError*** – if overwrite==False and output data file of problem already exists

`fastoad.cmd.api.`**`optimize_problem`**(*configuration_file_path: str*, *overwrite: bool = False*, *auto_scaling: bool = False*) → *fastoad.openmdao.problem.FASTOADProblem*

Runs driver according to provided problem file

Parameters

- **configuration_file_path** – problem definition
- **overwrite** – if True, output file will be overwritten
- **auto_scaling** – if True, automatic scaling is performed for design variables and constraints

Returns the OpenMDAO problem after run

Raises ***FastPathExistsError*** – if overwrite==False and output data file of problem already exists

`fastoad.cmd.api.`**`optimization_viewer`**(*configuration_file_path: str*)

Displays optimization information and enables its editing

Parameters **configuration_file_path** – problem definition

Returns display of the OptimizationViewer

`fastoad.cmd.api.`**`variable_viewer`**(*file_path: str*, *file_formatter: Optional[*fastoad.io.formatter.IVariableIOFormatter*] = None*, *editable=True*)

Displays a widget that enables to visualize variables information and edit their values.

Parameters

- **file_path** – the path of file to interact with
- **file_formatter** – the formatter that defines file format. If not provided, default format will be assumed.
- **editable** – if True, an editable table with variable filters will be displayed. If False, the table will not be editable nor searchable, but can be stored in an HTML file.

Returns display handle of the VariableViewer

fastoad.cmd.cli module

Command Line Interface.

fastoad.cmd.cli_utils module

Utility functions for CLI interface.

`fastoad.cmd.cli_utils.`**`overwrite_option`**(*func*)

Decorator for adding the option for overwriting existing file.

Use *force* as argument of the function.

fastoad.cmd.cli_utils.**out_file_option**(*func*)
Decorator for writing command output in a file.

Use *out_file* and *force* as argument of the function.

fastoad.cmd.cli_utils.**manage_overwrite**(*func: Callable*, *filename_func: Optional[Callable] = None*, ***kwargs*)
Runs *func*, that is expected to write a file, with provided keyword arguments *args*.

If the run throws FastPathExistsError, a question is displayed and user is asked for a yes/no answer. If *yes* is given, arg["overwrite"] is set to True and *func* is run again.

Parameters

- **func** – callable that will do the operation and is expected to return the path of written element.
- **filename_func** – a function that provides the name of written file, given the value returned by func
- **kwargs** – keyword arguments for func

Returns True if the file has been written,

fastoad.cmd.exceptions module

Exception for cmd package

exception fastoad.cmd.exceptions.**FastPathExistsError**(**args*)
Bases: *fastoad.exceptions.FastError*

Raised when asked for writing a file/folder that already exists.

exception fastoad.cmd.exceptions.**FastNoAvailableNotebookError**(*distribution_name=None*)
Bases: *fastoad.exceptions.FastError*

Raised when no notebook is available for creation.

Module contents

fastoad.configurations package

Module contents

fastoad.gui package

Subpackages

Submodules

fastoad.gui.analysis_and_plots module

Defines the analysis and plotting functions for postprocessing

fastoad.gui.analysis_and_plots.**wing_geometry_plot**(*aircraft_file_path: str*, *name=None*, *fig=None*, ***, *file_formatter=None*) → plotly.graph_objs._figurewidget.FigureWidget

Returns a figure plot of the top view of the wing. Different designs can be superposed by providing an existing fig. Each design can be provided a name.

Parameters

- **aircraft_file_path** – path of data file
- **name** – name to give to the trace added to the figure
- **fig** – existing figure to which add the plot
- **file_formatter** – the formatter that defines the format of data file. If not provided, default format will be assumed.

Returns wing plot figure

fastoad.gui.analysis_and_plots.**aircraft_geometry_plot**(*aircraft_file_path: str*, *name=None*, *fig=None*, ***, *file_formatter=None*) → plotly.graph_objs._figurewidget.FigureWidget

Returns a figure plot of the top view of the wing. Different designs can be superposed by providing an existing fig. Each design can be provided a name.

Parameters

- **aircraft_file_path** – path of data file
- **name** – name to give to the trace added to the figure
- **fig** – existing figure to which add the plot
- **file_formatter** – the formatter that defines the format of data file. If not provided, default format will be assumed.

Returns wing plot figure

fastoad.gui.analysis_and_plots.**drag_polar_plot**(*aircraft_file_path: str*, *name=None*, *fig=None*, ***, *file_formatter=None*) → plotly.graph_objs._figurewidget.FigureWidget

Returns a figure plot of the aircraft drag polar. Different designs can be superposed by providing an existing fig. Each design can be provided a name.

Parameters

- **aircraft_file_path** – path of data file
- **name** – name to give to the trace added to the figure
- **fig** – existing figure to which add the plot
- **file_formatter** – the formatter that defines the format of data file. If not provided, default format will be assumed.

Returns wing plot figure

fastoad.gui.analysis_and_plots.**mass_breakdown_bar_plot**(*aircraft_file_path: str*, *name=None*, *fig=None*, ***, *file_formatter=None*, *input_mass_name='data:weight:aircraft:MTOW'*) → plotly.graph_objs._figurewidget.FigureWidget

Returns a figure plot of the aircraft mass breakdown using bar plots. Different designs can be superposed by providing an existing fig. Each design can be provided a name.

Parameters

- **aircraft_file_path** – path of data file
- **name** – name to give to the trace added to the figure
- **fig** – existing figure to which add the plot
- **file_formatter** – the formatter that defines the format of data file. If not provided, default format will be assumed.
- **input_mass_name** – the variable name for the mass input as defined in the mission definition file.

Returns bar plot figure

fastoad.gui.analysis_and_plots.**mass_breakdown_sun_plot**(*aircraft_file_path: str*, ***, *file_formatter=None*, *input_mass_name='data:weight:aircraft:MTOW'*)

Returns a figure sunburst plot of the mass breakdown. On the left a MTOW sunburst and on the right a OWE sunburst. Different designs can be superposed by providing an existing fig. Each design can be provided a name.

Parameters

- **aircraft_file_path** – path of data file
- **file_formatter** – the formatter that defines the format of data file. If not provided, default format will be assumed.
- **input_mass_name** – the variable name for the mass input as defined in the mission definition file.

Returns sunburst plot figure

fastoad.gui.analysis_and_plots.**payload_range_plot**(*aircraft_file_path: str*, *name='Payload-Range'*, *mission_name='operational'*, *variable_of_interest: Optional[str] = None*, *variable_of_interest_legend: Optional[str] = None*)

Returns a figure of the payload-range diagram. The diagram contains by default only the contour but can also provide a heatmap of the grid points, if variable_of_interest is not None. Please note that the data for the contour are expected in the variables *data:payload_range:{mission_name}:range* and *data:payload_range:{mission_name}:payload*. Similarly, the data for the heatmap are expected in the variables *data:payload_range:{mission_name}:grid:range*, *data:payload_range:{mission_name}:grid:payload* and *data:payload_range:{mission_name}:grid:{variable_of_interest}*.

Parameters

- **aircraft_file_path** – path of data file
- **name** – name to give to the trace added to the figure
- **mission_name** – name of the mission present in the data file to be plotted.
- **variable_of_interest** – variable of interest for the heatmap.
- **variable_of_interest_legend** – name to give to variable of interest in plot legend.

Returns payload-range plot figure

fastoad.gui.exceptions module

Exception for GUI

exception fastoad.gui.exceptions.**FastMissingFile**
Bases: *fastoad.exceptions.FastError*

Raised when a file does not exist

fastoad.gui.mission_viewer module

Defines the analysis and plotting functions for postprocessing regarding the mission

class fastoad.gui.mission_viewer.**MissionViewer**
Bases: object

A class for facilitating the post-processing of mission and trajectories

add_mission(*mission_data: Union[str, pandas.core.frame.DataFrame]*, *name=None*)
Adds the mission to the mission database (self.missions) :param mission_data: path of the mission file or Dataframe containing the mission data :param name: name to give to the mission

display()
Display the user interface :return the display object

fastoad.gui.optimization_viewer module

Defines the variable viewer for postprocessing

class fastoad.gui.optimization_viewer.**OptimizationViewer**
Bases: object

A class for interacting with FAST-OAD Problem optimization information.

problem_configuration:
fastoad.io.configuration.configuration.FASTOADProblemConfigurator
Instance of the FAST-OAD problem configuration

dataframe
The dataframe which is the mirror of self.file

load(*problem_configuration:* fastoad.io.configuration.configuration.FASTOADProblemConfigurator)
Loads the FAST-OAD problem and stores its data.

Parameters **problem_configuration** – the FASTOADProblem instance.

save()
Save the optimization to the files. Possible files modified are:

- the .yml configuration file
- the input file (initial values)
- the output file (values)

display()
Displays the datasheet. load() must be ran before.

Returns display of the user interface:

load_variables(*variables:* fastoad.openmdao.variables.variable_list.VariableList, *attribute_to_column: Optional[Dict[str, str]] = None*)
Loads provided variable list and replace current data set.

Parameters

- **variables** – the variables to load
- **attribute_to_column** – dictionary keys tell what variable attributes are kept and the values tell what name will be displayed. If not provided, default translation will apply.

get_variables(*column_to_attribute: Optional[Dict[str, str]] = None*) → *fastoad.openmdao.variables.variable_list.VariableList*

Parameters **column_to_attribute** – dictionary keys tell what columns are kept and the values tell whatvariable attribute it corresponds to. If not provided, default translation will apply.

Returns a variable list from current data set

fastoad.gui.variable_viewer module

Defines the variable viewer for postprocessing

class fastoad.gui.variable_viewer.**VariableViewer**
Bases: object

A class for interacting with FAST-OAD files. The file data is stored in a pandas DataFrame. The class built so that a modification of the DataFrame is instantly replicated on the file file. The interaction is achieved using a user interface built with widgets from ipywidgets and Sheets from ipysheet.

A classical usage of this class will be:

```
df = VariableViewer()  # instantiation of dataframe
file = AbstractOMFileIO('problem_outputs.file') #  instantiation of file io
df.load(file)  # load the file
df.display()  # renders a ui for reading/modifying the file
```

file
The path of the data file that will be viewed/edited

dataframe
The dataframe which is the mirror of self.file

load(*file_path: str*, *file_formatter: Optional[*fastoad.io.formatter.IVariableIOFormatter*] = None*)
Loads the file and stores its data.

Parameters

- **file_path** – the path of file to interact with
- **file_formatter** – the formatter that defines file format. If not provided, default format will be assumed.

save(*file_path: Optional[str] = None*, *file_formatter: Optional[*fastoad.io.formatter.IVariableIOFormatter*] = None*)
Save the dataframe to the file.

Parameters

- **file_path** – the path of file to save. If not given, the initially read file will be overwritten.
- **file_formatter** – the formatter that defines file format. If not provided, default format will be assumed.

display()
Displays the datasheet :return display of the user interface:

load_variables(*variables:* fastoad.openmdao.variables.variable_list.VariableList, *attribute_to_column: Optional[Dict[str, str]] = None*)
Loads provided variable list and replace current data set.

Parameters

- **variables** – the variables to load
- **attribute_to_column** – dictionary keys tell what variable attributes are kept and the values tell what name will be displayed. If not provided, default translation will apply.

get_variables(*column_to_attribute: Optional[Dict[str, str]] = None*) → *fastoad.openmdao.variables.variable_list.VariableList*

Parameters **column_to_attribute** – dictionary keys tell what columns are kept and the values tell what variable attribute it corresponds to. If not provided, default translation will apply.

Returns a variable list from current data set

Module contents

fastoad.io package

Subpackages

fastoad.io.configuration package

Subpackages

Submodules

fastoad.io.configuration.configuration module

Module for building OpenMDAO problem from configuration file

class fastoad.io.configuration.configuration.**FASTOADProblemConfigurator**(*conf_file_path=None*)
Bases: object

class for configuring an OpenMDAO problem from a configuration file

See *description of configuration file*.

Parameters **conf_file_path** – if provided, configuration will be read directly from it

property input_file_path
path of file with input variables of the problem

property output_file_path
path of file where output variables will be written

get_problem(*read_inputs: bool = False*, *auto_scaling: bool = False*) → *fastoad.openmdao.problem.FASTOADProblem*
Builds the OpenMDAO problem from current configuration.

Parameters

- **read_inputs** – if True, the created problem will already be fed with variables from the input file
- **auto_scaling** – if True, automatic scaling is performed for design variables and constraints

Returns the problem instance

load(*conf_file*)
Reads the problem definition

Parameters **conf_file** – Path to the file to open or a file descriptor

save(*filename: Optional[str] = None*)
Saves the current configuration If no filename is provided, the initially read file is used.

Parameters **filename** – file where to save configuration

write_needed_inputs(*source_file_path: Optional[str] = None*, *source_formatter: Optional[*fastoad.io.formatter.IVariableIOFormatter*] = None*)
Writes the input file of the problem with unconnected inputs of the configured problem.

Written value of each variable will be taken:

1. from input_data if it contains the variable
2. from defined default values in component definitions

Parameters

- **source_file_path** – if provided, variable values will be read from it
- **source_formatter** – the class that defines format of input file. if not provided, expected format will be the default one.

get_optimization_definition() → Dict

Returns information related to the optimization problem:

- Design Variables
- Constraints
- Objectives

Returns dict containing optimization settings for current problem

set_optimization_definition(*optimization_definition: Dict*)
Updates configuration with the list of design variables, constraints, objectives contained in the optimization_definition dictionary.

Keys of the dictionary are: “design_var”, “constraint”, “objective”.

Configuration file will not be modified until *save()* is used.

Parameters **optimization_definition** – dict containing the optimization problem definition

fastoad.io.configuration.exceptions module

Exceptions for package configuration

exception fastoad.io.configuration.exceptions.**FASTConfigurationBaseKeyBuildingError**(*original_exception: Exception*, *key: str*, *value=None*)

Bases: *fastoad.exceptions.FastError*

Class for being raised from bottom to top of TOML dict so that in the end, the message provides the full qualified name of the problematic key.

using *new_err = FASTConfigurationBaseKeyBuildingError(err, 'new_err_key', <value>)*:

- **if err is a FASTConfigurationBaseKeyBuildingError instance with err.key=='err_key':**
  - new_err.key will be 'new_err_key.err_key'
  - new_err.value will be err.value (no need to provide a value here)
  - new_err.original_exception will be err.original_exception
- **otherwise, new_err.key will be 'new_err_key' and new_err.value will be <value>**
  - new_err.key will be 'new_err_key'
  - new_err.value will be <value>
  - new_err.original_exception will be err

Parameters

- **original_exception** – the error that happened for raising this one
- **key** – the current key
- **value** – the current value

Constructor

key
the "qualified key" (like "problem.group.component1") related to error, build through raising up the error

value
the value related to error

original_exception
the original error, when eval failed

exception fastoad.io.configuration.exceptions.**FASTConfigurationBadOpenMDAOInstructionError**(*original_exception: Exception*, *key: str*, *value=None*)

Bases: *fastoad.io.configuration.exceptions.FASTConfigurationBaseKeyBuildingError*

Class for managing errors that result from trying to set an attribute by eval.

Constructor

Module contents

Package for building OpenMDAO problem from configuration file

fastoad.io.xml package

Subpackages

Submodules

fastoad.io.xml.constants module

Constants for the XML module

fastoad.io.xml.constants.**DEFAULT_UNIT_ATTRIBUTE = 'units'**
label of tag attribute for providing units as a string

fastoad.io.xml.constants.**DEFAULT_IO_ATTRIBUTE = 'is_input'**
label of tag attribute for providing io variable type as boolean

fastoad.io.xml.constants.**ROOT_TAG = 'FASTOAD_model'**
name of root element for XML files

fastoad.io.xml.exceptions module

Exceptions for io.xml module

exception fastoad.io.xml.exceptions.**FastXPathEvalError**
Bases: *fastoad.exceptions.FastError*

Raised when some xpath could not be resolved

exception fastoad.io.xml.exceptions.**FastXpathTranslatorInconsistentLists**
Bases: *fastoad.exceptions.FastError*

Raised when list of variable names and list of XPaths have not the same length

exception fastoad.io.xml.exceptions.**FastXpathTranslatorDuplicates**
Bases: *fastoad.exceptions.FastError*

Raised when list of variable names or list of XPaths have duplicate entries

exception fastoad.io.xml.exceptions.**FastXpathTranslatorVariableError**(*variable*)
Bases: *fastoad.exceptions.FastError*

Raised when a variable does not match any xpath in the translator file.

exception fastoad.io.xml.exceptions.**FastXpathTranslatorXPathError**(*xpath*)
Bases: *fastoad.exceptions.FastError*

Raised when a xpath does not match any variable in the translator file.

exception fastoad.io.xml.exceptions.**FastXmlFormatterDuplicateVariableError**
Bases: *fastoad.exceptions.FastError*

Raised a variable is defined more than once in a XML file

fastoad.io.xml.translator module

Conversion from OpenMDAO variables to XPath

class fastoad.io.xml.translator.**VarXpathTranslator**(*, *variable_names: Optional[Sequence[str]] = None*, *xpaths: Optional[Sequence[str]] = None*, *source: Optional[Union[IO, str]] = None*)
Bases: object

Allows to convert OpenMDAO variable names from and to XPath, using a provided conversion table.

At instantiation, user can provide (as keyword arguments only):

- variable_names and xpaths (see *set()*)
- translation file (see *read_translation_table()*)

set(*variable_names: Sequence[str]*, *xpaths: Sequence[str]*)
Sets the "conversion table", i.e. two lists where each element matches the other with same index. Provided lists must have the same length.

Parameters

- **variable_names** – List of OpenMDAO variable names
- **xpaths** – List of XML Paths

read_translation_table(*source: Union[str, IO]*)
Reads a file that sets how OpenMDAO variable are matched to XML Path. Provided file should have 2 comma-separated columns:

- first one with OpenMDAO names
- second one with their matching XPath

Parameters **source** –

property variable_names: Sequence[str]
List of variable names as set in *set()*

property xpaths: Sequence[str]
List of XPaths as set in *set()*

get_xpath(*var_name: str*) → str

Parameters **var_name** – OpenMDAO variable name

Returns XPath that matches var_name

Raises *FastXpathTranslatorVariableError* – if var_name is unknown

get_variable_name(*xpath: str*) → str

Parameters **xpath** – XML Path

Returns OpenMDAO variable name that matches xpath

Raises *FastXpathTranslatorXPathError* – if xpath is unknown

fastoad.io.xml.variable_io_base module

Defines how OpenMDAO variables are serialized to XML using a conversion table

class fastoad.io.xml.variable_io_base.**VariableXmlBaseFormatter**(*translator:* fastoad.io.xml.translator.VarXpathTranslator)

Bases: *fastoad.io.formatter.IVariableIOFormatter*

Customizable formatter for variables

User must provide at instantiation a VarXpathTranslator instance that tells how variable names should be converted from/to XPath.

Note: XPath are always considered relatively to the root. Therefore, "foo/bar" should be provided to match following XML structure:

```
<root>
    <foo>
        <bar>
            "some value"
        </bar>
    </foo>
</root>
```

Parameters **translator** – the VarXpathTranslator instance

xml_unit_attribute
The XML attribute key for specifying units

xml_io_attribute
The XML attribute key for specifying I/O status

read_variables(*data_source: Union[str, IO]*) → *fastoad.openmdao.variables.variable_list.VariableList*
Reads variables from provided data source file.

Parameters **data_source** –

Returns a list of Variable instance

write_variables(*data_source: Union[str, IO]*, *variables:* fastoad.openmdao.variables.variable_list.VariableList)
Writes variables to defined data source file.

Parameters

- **data_source** –
- **variables** –

fastoad.io.xml.variable_io_legacy module

Readers for legacy XML format

class fastoad.io.xml.variable_io_legacy.**VariableLegacy1XmlFormatter**
Bases: *fastoad.io.xml.variable_io_base.VariableXmlBaseFormatter*

Formatter for legacy XML format (version "1")

fastoad.io.xml.variable_io_standard module

Defines how OpenMDAO variables are serialized to XML

class fastoad.io.xml.variable_io_standard.**VariableXmlStandardFormatter**
Bases: *fastoad.io.xml.variable_io_base.VariableXmlBaseFormatter*

Standard XML formatter for variables

Assuming self.path_separator is defined as `:` (default), a variable named like `foo:bar` with units `m/s` will be read and written as:

```
<aircraft>
    <foo>
        <bar units="m/s" >`42.0</bar>
    </foo>
<aircraft>
```

When writing outputs of a model, OpenMDAO component hierarchy may be used by defining

```
self.path_separator = '.'  # Discouraged for reading !
self.use_promoted_names = False
```

This way, a variable like `componentA.subcomponent2.my_var` will be written as:

```
<aircraft>
    <componentA>
        <subcomponent2>
            <my_var units="m/s" >72.0</my_var>
        </subcomponent2>
    <componentA>
<aircraft>
```

property path_separator
The separator that will be used in OpenMDAO variable names to match XML path. Warning: The dot "." can be used when writing, but not when reading.

read_variables(*data_source: Union[str, IO]*) → *fastoad.openmdao.variables.variable_list.VariableList*
Reads variables from provided data source file.

Parameters **data_source** –

Returns a list of Variable instance

write_variables(*data_source: Union[str, IO]*, *variables:* fastoad.openmdao.variables.variable_list.VariableList)
Writes variables to defined data source file.

Parameters

- **data_source** –
- **variables** –

class fastoad.io.xml.variable_io_standard.**BasicVarXpathTranslator**(*path_separator*)
Bases: *fastoad.io.xml.translator.VarXpathTranslator*

Dedicated VarXpathTranslator that builds variable names by simply converting the '/' separator of XPaths into the desired separator.

get_variable_name(*xpath: str*) → str

Parameters **xpath** – XML Path

Returns OpenMDAO variable name that matches xpath

Raises ***FastXpathTranslatorXPathError*** – if xpath is unknown

get_xpath(*var_name: str*) → str

Parameters **var_name** – OpenMDAO variable name

Returns XPath that matches var_name

Raises ***FastXpathTranslatorVariableError*** – if var_name is unknown

Module contents

Package for handling XML files

Submodules

fastoad.io.formatter module

class fastoad.io.formatter.**IVariableIOFormatter**
Bases: abc.ABC

Interface for formatter classes to be used in VariableIO class.

The file format is defined by the implementation of this interface.

abstract read_variables(*data_source: Union[str, IO]*) → *fastoad.openmdao.variables.variable_list.VariableList*

Reads variables from provided data source file.

Parameters **data_source** –

Returns a list of Variable instance

abstract write_variables(*data_source: Union[str, IO]*, *variables:* fastoad.openmdao.variables.variable_list.VariableList)

Writes variables to defined data source file.

Parameters

- **data_source** –
- **variables** –

fastoad.io.variable_io module

class fastoad.io.variable_io.**VariableIO**(*data_source: Union[str, IO]*, *formatter: Optional[fastoad.io.formatter.IVariableIOFormatter] = None*)

Bases: object

Class for reading and writing variable values from/to file.

The file format is defined by the class provided as *formatter* argument.

Parameters

- **data_source** – the I/O stream, or a file path, used for reading or writing data
- **formatter** – a class that determines the file format to be used. Defaults to a VariableBasicXmlFormatter instance.

read(*only: Optional[List[str]] = None*, *ignore: Optional[List[str]] = None*) → *fastoad.openmdao.variables.variable_list.VariableList*

Reads variables from provided data source.

Elements of *only* and *ignore* can be real variable names or Unix-shell-style patterns. In any case, comparison is case-sensitive.

Parameters

- **only** – List of variable names that should be read. Other names will be ignored. If None, all variables will be read.
- **ignore** – List of variable names that should be ignored when reading.

Returns an VariableList instance where outputs have been defined using provided source

write(*variables:* fastoad.openmdao.variables.variable_list.VariableList, *only: Optional[List[str]] = None*, *ignore: Optional[List[str]] = None*)

Writes variables from provided VariableList instance.

Elements of *only* and *ignore* can be real variable names or Unix-shell-style patterns. In any case, comparison is case-sensitive.

Parameters

- **variables** – a VariableList instance
- **only** – List of variable names that should be written. Other names will be ignored. If None, all variables will be written.
- **ignore** – List of variable names that should be ignored when writing

class fastoad.io.variable_io.**DataFile**(*data_source: Optional[Union[str, IO, list]] = None*, *formatter: Optional[fastoad.io.formatter.IVariableIOFormatter] = None*, *load_data=True*)

Bases: *fastoad.openmdao.variables.variable_list.VariableList*

Class for managing FAST-OAD data files.

Behaves like VariableList class but has *load()* and *save()* methods.

If variable list is specified for data_source, *file_path* will have to be set before using **:method:`save`**.

Parameters

- **data_source** – Can be the file path where data will be loaded and saved, or a list of Variable instances that will be used for initialization (or a VariableList instance).

- **formatter** – (ignored if data_source is not an I/O stream nor a file path) a class that determines the file format to be used. Defaults to FAST-OAD native format. See *VariableIO* for more information.
- **load_data** – (ignored if data_source is not an I/O stream nor a file path) if True, file is expected to exist and its content will be loaded at instantiation.

property file_path: str
Path of data file.

property formatter: *fastoad.io.formatter.IVariableIOFormatter*
Class that defines the file format.

load()
Loads file content.

save()
Saves current state of variables in file.

save_as(*file_path: str*, *overwrite=False*, *formatter: Optional[*fastoad.io.formatter.IVariableIOFormatter*] = None*)
Sets the associated file path as specified and saves current state of variables.

Parameters

- **file_path** –
- **overwrite** – if specified file already exists and overwrite is False, an error is triggered.
- **formatter** – a class that determines the file format to be used. Defaults to FAST-OAD native format. See *VariableIO* for more information.

Module contents

Package for handling input/output streams

fastoad.model_base package

Subpackages

Submodules

fastoad.model_base.atmosphere module

Simple implementation of International Standard Atmosphere.

class fastoad.model_base.atmosphere.**Atmosphere**(**args*, ***kwargs*)
Bases: object

Simple implementation of International Standard Atmosphere for troposphere and stratosphere.

Atmosphere properties are provided in the same "shape" as provided altitude:

- if altitude is given as a float, returned values will be floats
- if altitude is given as a sequence (list, 1D numpy array, ...), returned values will be 1D numpy arrays
- if altitude is given as nD numpy array, returned values will be nD numpy arrays

Usage:

```
>>> pressure = Atmosphere(30000).pressure # pressure at 30,000 feet, dISA = 0 K
>>> density = Atmosphere(5000, 10).density # density at 5,000 feet, dISA = 10 K


>>> atm = Atmosphere(np.arange(0,10001,1000, 15)) # init for alt. 0 to 10,000, dISA␣
↪= 15K
>>> temperatures = atm.pressure # pressures for all defined altitudes
>>> viscosities = atm.kinematic_viscosity # viscosities for all defined altitudes
```

Parameters

- **altitude** – altitude (units decided by altitude_in_feet)
- **delta_t** – temperature increment (°C) applied to whole temperature profile
- **altitude_in_feet** – if True, altitude should be provided in feet. Otherwise, it should be provided in meters.

get_altitude(*altitude_in_feet: bool = True*) → Union[float, Sequence[float]]

Parameters **altitude_in_feet** – if True, altitude is returned in feet. Otherwise, it is returned in meters

Returns altitude provided at instantiation

property delta_t: Union[float, Sequence[float]]
Temperature increment applied to whole temperature profile.

property temperature: Union[float, Sequence[float]]
Temperature in K.

property pressure: Union[float, Sequence[float]]
Pressure in Pa.

property density: Union[float, Sequence[float]]
Density in kg/m3.

property speed_of_sound: Union[float, Sequence[float]]
Speed of sound in m/s.

property kinematic_viscosity: Union[float, Sequence[float]]
Kinematic viscosity in m2/s.

property mach: Union[float, Sequence[float]]
Mach number.

property true_airspeed: Union[float, Sequence[float]]
True airspeed (TAS) in m/s.

property equivalent_airspeed: Union[float, Sequence[float]]
Equivalent airspeed (EAS) in m/s.

property unitary_reynolds: Union[float, Sequence[float]]
Unitary Reynolds number in 1/m.

class fastoad.model_base.atmosphere.**AtmosphereSI**(**args*, ***kwargs*)
Bases: *fastoad.model_base.atmosphere.Atmosphere*

Same as *Atmosphere* except that altitudes are always in meters.

Parameters

- **altitude** – altitude in meters
- **delta_t** – temperature increment (°C) applied to whole temperature profile

property altitude
Altitude in meters.

fastoad.model_base.datacls module

Dataclass utilities.

fastoad.model_base.datacls.**MANDATORY_FIELD = <object object>**
To be put as default value for dataclass fields that should not have a default value. See *BaseDataClass* for further information.

class fastoad.model_base.datacls.**BaseDataClass**
Bases: object

This class is a workaround for the following dataclass problem:

> If a dataclass defines a field with a default value, inheritor classes will not be allowed to define fields without default value, because then the non-default fields will follow a default field, which is forbidden.

The chosen solution (from https://stackoverflow.com/a/53085935/16488238) is to always define default values, but mandatory fields will have the *MANDATORY_FIELD* object as default.

After initialization, `__post_init__()` will process fields and raise an error if a field has *MANDATORY_FIELD* as value.

fastoad.model_base.flight_point module

Structure for managing flight point data.

class fastoad.model_base.flight_point.**FlightPoint**(*time: float = 0.0*, *altitude: Optional[float] = None*, *isa_offset: float = 0.0*, *ground_distance: float = 0.0*, *mass: Optional[float] = None*, *consumed_fuel: float = 0.0*, *true_airspeed: Optional[float] = None*, *equivalent_airspeed: Optional[float] = None*, *mach: Optional[float] = None*, *engine_setting: Optional[fastoad.constants.EngineSetting] = None*, *CL: Optional[float] = None*, *CD: Optional[float] = None*, *lift: Optional[float] = None*, *drag: Optional[float] = None*, *thrust: Optional[float] = None*, *thrust_rate: Optional[float] = None*, *thrust_is_regulated: Optional[bool] = None*, *sfc: Optional[float] = None*, *slope_angle: Optional[float] = None*, *acceleration: Optional[float] = None*, *alpha: float = 0.0*, *slope_angle_derivative: Optional[float] = None*, *name: Optional[str] = None*)
Bases: object

Dataclass for storing data for one flight point.

This class is meant for:

- pandas friendliness: data exchange with pandas DataFrames is simple
- extensibility: any user might add fields to the **class** using *add_field()*

Exchanges with pandas DataFrame

A pandas DataFrame can be generated from a list of FlightPoint instances:

```
>>> import pandas as pd
>>> from fastoad.model_base import FlightPoint

>>> fp1 = FlightPoint(mass=70000., altitude=0.)
>>> fp2 = FlightPoint(mass=60000., altitude=10000.)
>>> df = pd.DataFrame([fp1, fp2])
```

And FlightPoint instances can be created from DataFrame rows:

```
# Get one FlightPoint instance from a DataFrame row
>>> fp1bis = FlightPoint.create(df.iloc[0])

# Get a list of FlightPoint instances from the whole DataFrame
>>> flight_points = FlightPoint.create_list(df)
```

Extensibility

FlightPoint class is bundled with several fields that are commonly used in trajectory assessment, but one might need additional fields.

Python allows to add attributes to any instance at runtime, but for FlightPoint to run smoothly, especially when exchanging data with pandas, you have to work at class level. This can be done using *add_field()*, preferably outside of any class or function:

```
# Adds a float field with None as default value
>>> FlightPoint.add_field("ion_drive_power")

# Adds a field and define its type and default value
>>> FlightPoint.add_field("warp", annotation_type=int, default_value=9)

# Now these fields can be used at instantiation
>>> fp = FlightPoint(ion_drive_power=110.0, warp=12)

# Removes a field, even an original one (useful only to avoid having it in␣
↪outputs)
>>> FlightPoint.remove_field("sfc")
```

Note: All parameters in FlightPoint instances are expected to be in SI units.

time: *float* = **0.0**
Time in seconds.

altitude: *float* = **None**
Altitude in meters.

isa_offset: *float* = **0.0**
temperature deviation from Standard Atmosphere

ground_distance: *float* = **0.0**
Covered ground distance in meters.

mass: *float* = **None**
Mass in kg.

consumed_fuel: *float* = **0.0**
Consumed fuel since mission start, in kg.

true_airspeed: *float* = **None**
True airspeed (TAS) in m/s.

equivalent_airspeed: *float* = **None**
Equivalent airspeed (EAS) in m/s.

mach: *float* = **None**
Mach number.

engine_setting: *fastoad.constants.EngineSetting* = **None**
Engine setting.

CL: *float* = **None**
Lift coefficient.

CD: *float* = **None**
Drag coefficient.

lift: *float* = **None**
Aircraft lift in Newtons

drag: *float* = **None**
Aircraft drag in Newtons.

thrust: *float* = **None**
Thrust in Newtons.

thrust_rate: *float* = **None**
Thrust rate (between 0. and 1.)

thrust_is_regulated: *bool* = **None**
If True, propulsion should match the thrust value. If False, propulsion should match thrust rate.

sfc: *float* = **None**
Specific Fuel Consumption in kg/N/s.

slope_angle: *float* = **None**
Slope angle in radians.

acceleration: *float* = **None**
Acceleration value in m/s**2.

alpha: *float* = **0.0**
angle of attack in radians

slope_angle_derivative: *float* = **None**
slope angle derivative in rad/s

name: *str* = **None**
Name of current phase.

set_as_relative(*field_names: Union[Sequence[str], str]*)
Makes that values for given field_names will be considered as relative when calling *make_absolute()*.

Parameters **field_names** –

set_as_absolute(*field_names: Union[Sequence[str], str]*)
Makes that values for given field_names will be considered as absolute when calling *make_absolute()*.

Parameters **field_names** –

is_relative(*field_name*) → bool
Tells if given field is considered as relative or absolut

Parameters **field_name** –

Returns True if it is relative

make_absolute(*reference_point:* fastoad.model_base.flight_point.FlightPoint) → *fastoad.model_base.flight_point.FlightPoint*
Computes a copy flight point where no field is relative.

Parameters **reference_point** – relative fields will be made absolute using this point.

Returns the copied flight point with no relative field.

classmethod get_field_names()

Returns names of all fields of the flight point.

classmethod get_units() → dict
Returns (field name, unit) dict for any field that has a defined unit.

A dimensionless physical quantity will have "-" as unit.

classmethod create(*data: Mapping*) → *fastoad.model_base.flight_point.FlightPoint*
Instantiate FlightPoint from provided data.

data can typically be a dict or a pandas DataFrame row.

Parameters **data** – a dict-like instance where keys are FlightPoint attribute names

Returns the created FlightPoint instance

classmethod create_list(*data: pandas.core.frame.DataFrame*) → List[*fastoad.model_base.flight_point.FlightPoint*]
Creates a list of FlightPoint instances from provided DataFrame.

Parameters **data** – a dict-like instance where keys are FlightPoint attribute names

Returns the created FlightPoint instance

classmethod add_field(*name: str*, *annotation_type=<class 'float'>*, *default_value: Optional[Any] = None*, *unit=None*)
Adds the named field to FlightPoint class.

If the field name already exists, the field is redefined.

Parameters

- **name** – field name
- **annotation_type** – field type
- **default_value** – field default value
- **unit** – expected unit for the added field ("-" should be provided for a dimensionless physical quantity)

classmethod remove_field(*name*)
Removes the named field from FlightPoint class.

Parameters **name** – field name

scalarize()
Convenience method for converting to scalars all fields that have a one-item array-like value.

fastoad.model_base.propulsion module

Base classes for propulsion components.

class fastoad.model_base.propulsion.**IPropulsion**
Bases: abc.ABC

Interface that should be implemented by propulsion models.

Using this class allows to delegate to the propulsion model the management of propulsion-related data when computing performances.

The performance model calls *compute_flight_points()* by providing one or several flight points. The method will feed these flight points with results of the model (e.g. thrust, SFC, ..).

The performance model will then be able to call *get_consumed_mass()* to know the mass consumption for each flight point.

Note:

```
If the propulsion model needs fields that are not among defined fields
of the :class`FlightPoint class`, these fields can be made authorized by
:class`FlightPoint class`. Please see part about extensibility in
:class`FlightPoint class` documentation.
```

abstract compute_flight_points(*flight_points: Union[*fastoad.model_base.flight_point.FlightPoint, *pandas.core.frame.DataFrame]*)
Computes Specific Fuel Consumption according to provided conditions.

See *FlightPoint* for available fields that may be used for computation. If a DataFrame instance is provided, it is expected that its columns match field names of FlightPoint (actually, the DataFrame instance should be generated from a list of FlightPoint instances).

Note: About thrust_is_regulated, thrust_rate and thrust

`thrust_is_regulated` tells if a flight point should be computed using `thrust_rate` (when False) or `thrust` (when True) as input. This way, the method can be used in a vectorized mode, where each point can be set to respect a **thrust** order or a **thrust rate** order.

- if `thrust_is_regulated` is not defined, the considered input will be the defined one between `thrust_rate` and `thrust` (if both are provided, `thrust_rate` will be used)
- if `thrust_is_regulated` is `True` or `False` (i.e., not a sequence), the considered input will be taken accordingly, and should of course be defined.
- if there are several flight points, `thrust_is_regulated` is a sequence or array, `thrust_rate` and `thrust` should be provided and have the same shape as `thrust_is_regulated:code:`. The method will consider for each element which input will be used according to `thrust_is_regulated`.

Parameters **flight_points** – FlightPoint or DataFram instance

Returns None (inputs are updated in-place)

abstract get_consumed_mass(*flight_point:* fastoad.model_base.flight_point.FlightPoint, *time_step: float*) → float

Computes consumed mass for provided flight point and time step.

This method should rely on FlightPoint fields that are generated by :meth: *compute_flight_points*.

Parameters

- **flight_point** –
- **time_step** –

Returns the consumed mass in kg

class fastoad.model_base.propulsion.**IOMPropulsionWrapper**

Bases: object

Interface for wrapping a *IPropulsion* subclass in OpenMDAO.

The implementation class defines the needed input variables for instantiating the *IPropulsion* subclass in *setup()* and use them for instantiation in *get_model()*

See OMRubberEngineWrapper for an example of implementation.

abstract setup(*component: openmdao.core.component.Component*)

Defines the needed OpenMDAO inputs for propulsion instantiation as done in *get_model()*

Use *add_inputs* and *declare_partials* methods of the provided *component*

Parameters **component** –

abstract static get_model(*inputs*) → *fastoad.model_base.propulsion.IPropulsion*

This method defines the used *IPropulsion* subclass instance.

Parameters **inputs** – OpenMDAO input vector where the parameters that define the propulsion model are

Returns the propulsion model instance

class fastoad.model_base.propulsion.**BaseOMPropulsionComponent**(***kwargs*)

Bases: openmdao.core.explicitcomponent.ExplicitComponent, abc.ABC

Base class for creating an OpenMDAO component from subclasses of *IOMPropulsionWrapper*.

Classes that implements this interface should add their own inputs in setup() and implement *get_wrapper()*.

Store some bound methods so we can detect runtime overrides.

setup()

Declare inputs and outputs.

Available attributes: name pathname comm options

setup_partials()

Declare partials.

This is meant to be overridden by component classes. All partials should be declared here since this is called after all size/shape information is known for all variables.

compute(*inputs*, *outputs*, *discrete_inputs=None*, *discrete_outputs=None*)

Compute outputs given inputs. The model is assumed to be in an unscaled state.

Parameters

- **inputs** (*Vector*) – Unscaled, dimensional input variables read via inputs[key].
- **outputs** (*Vector*) – Unscaled, dimensional output variables read via outputs[key].

- **discrete_inputs** (*dict or None*) – If not None, dict containing discrete input values.
- **discrete_outputs** (*dict or None*) – If not None, dict containing discrete output values.

abstract static get_wrapper() → *fastoad.model_base.propulsion.IOMPropulsionWrapper*

This method defines the used *IOMPropulsionWrapper* instance.

Returns an instance of OpenMDAO wrapper for propulsion model

class fastoad.model_base.propulsion.**AbstractFuelPropulsion**

Bases: *fastoad.model_base.propulsion.IPropulsion*, abc.ABC

Propulsion model that consume any fuel should inherit from this one.

In inheritors, `compute_flight_points()` is expected to define "sfc" and "thrust" in computed FlightPoint instances.

get_consumed_mass(*flight_point:* fastoad.model_base.flight_point.FlightPoint, *time_step: float*) → float

Computes consumed mass for provided flight point and time step.

This method should rely on FlightPoint fields that are generated by :meth: *compute_flight_points*.

Parameters

- **flight_point** –
- **time_step** –

Returns the consumed mass in kg

class fastoad.model_base.propulsion.**FuelEngineSet**(*engine:* fastoad.model_base.propulsion.IPropulsion, *engine_count*)

Bases: *fastoad.model_base.propulsion.AbstractFuelPropulsion*

Class for modelling an assembly of identical fuel engines.

Thrust is supposed equally distributed among them.

Parameters

- **engine** – the engine model
- **engine_count** –

compute_flight_points(*flight_points: Union[*fastoad.model_base.flight_point.FlightPoint, *pandas.core.frame.DataFrame]*)

Computes Specific Fuel Consumption according to provided conditions.

See *FlightPoint* for available fields that may be used for computation. If a DataFrame instance is provided, it is expected that its columns match field names of FlightPoint (actually, the DataFrame instance should be generated from a list of FlightPoint instances).

Note: About thrust_is_regulated, thrust_rate and thrust

`thrust_is_regulated` tells if a flight point should be computed using `thrust_rate` (when False) or `thrust` (when True) as input. This way, the method can be used in a vectorized mode, where each point can be set to respect a **thrust** order or a **thrust rate** order.

- if `thrust_is_regulated` is not defined, the considered input will be the defined one between `thrust_rate` and `thrust` (if both are provided, `thrust_rate` will be used)

- if `thrust_is_regulated` is `True` or `False` (i.e., not a sequence), the considered input will be taken accordingly, and should of course be defined.
- if there are several flight points, `thrust_is_regulated` is a sequence or array, `thrust_rate` and `thrust` should be provided and have the same shape as `thrust_is_regulated:code:`. The method will consider for each element which input will be used according to `thrust_is_regulated`.

Parameters `flight_points` – FlightPoint or DataFram instance

Returns None (inputs are updated in-place)

Module contents

Base features for FAST-OAD models

fastoad.models package

Subpackages

fastoad.models.performances package

Subpackages

fastoad.models.performances.mission package

Subpackages

fastoad.models.performances.mission.mission_definition package

Subpackages

fastoad.models.performances.mission.mission_definition.mission_builder package

Subpackages

Submodules

fastoad.models.performances.mission.mission_definition.mission_builder.constants module

Constants for mission builder package.

fastoad.models.performances.mission.mission_definition.mission_builder.input_definition module

Management of mission input definitions.

class fastoad.models.performances.mission.mission_definition.mission_builder.input_definition.InputDefi

Class for managing definition of mission inputs.

It stores and processes input definition from mission files:

- provides values to be used for mission computation (management of units and variables)
- provides information for OpenMDAO declaration

parameter_name: *str*
The parameter this input is defined for.

input_value: *Optional[Union[numbers.Number, Iterable, str]]*
Value, matching *input_unit*. At instantiation, it can also be the variable name.

input_unit: *Optional[str] = None*
Unit used for self.input_value.

default_value: *numbers.Number = nan*
Default value. Used if value is a variable name.

is_relative: *bool = False*
True if variable is defined as relative.

part_identifier: *str = ''*
Prefix used when generating variable name because "~" was used in variable name input.

output_unit: *Optional[str] = None*
Unit used for self.value. Automatically determined from self.parameter_name, mainly from unit definition for FlightPoint class.

shape: *Optional[Tuple[int]] = None*
Value of the "shape" openMDAO flag for input declaration.

shape_by_conn: *bool = False*
Value of the "shape_by_conn" openMDAO flag for input declaration.

prefix: *str = ''*
Prefix used when replacement of "~" is needed.

use_opposite: *dataclasses.InitVar[typing.Optional[bool]] = None*
Used only for tests

property value
Value of variable in DEFAULT unit (unit used by mission calculation), or None if input is a variable and set_variable_input() has NOT been called, or the unchanged value if it is not a number.

Type return

classmethod from_dict(*parameter_name*, *definition_dict: dict*, *part_identifier=None*, *prefix=None*)
Instantiates InputDefinition from definition_dict.

definition_dict["value"] is used as *input_value* in instantiation. It can be an actual value or a variable name.

Parameters

- **parameter_name** – used if definition_dict["value"] == "~" (or "-~")
- **definition_dict** – dict with keys ("value", "unit", "default"). "unit" and "default" are optional.
- **part_identifier** – used if "~" is in definition_dict["value"]
- **prefix** – used if "~" is in definition_dict["value"]

set_variable_value(*inputs: Mapping*)
Sets numerical value from OpenMDAO inputs.

OpenMDAO value is assumed to be provided with unit self.input_unit.

Parameters **inputs** –

get_input_definition() → Optional[*fastoad.openmdao.variables.variable.Variable*]
Provides information for input definition in OpenMDAO.

Returns Variable instance with input definition, or None if no variable name was defined.

property variable_name
Used only for tests

fastoad.models.performances.mission.mission_definition.mission_builder.mission_builder module

Mission generator.

class fastoad.models.performances.mission.mission_definition.mission_builder.mission_builder.**MissionBui**

Bases: object

This class builds and computes a mission from a provided definition.

Parameters

- **mission_definition** – a file path or MissionDefinition instance

- **propulsion** – if not provided, the property *propulsion* must be set before calling *build()*
- **reference_area** – if not provided, the property *reference_area* must be set before calling *build()*
- **mission_name** – name of chosen mission, if already decided.
- **variable_prefix** – prefix for auto-generated variable names.

property definition:
fastoad.models.performances.mission.mission_definition.schema.MissionDefinition
The definition of missions as provided in input file.

property propulsion: *fastoad.model_base.propulsion.IPropulsion*
Propulsion model for performance computation.

property reference_area: float
Reference area for aerodynamic polar.

property mission_name
The mission name, in case it has been specified, or if it is unique in the file.

property variable_prefix
The prefix for auto-generated variable names.

build(*inputs: Optional[Mapping] = None*, *mission_name: Optional[str] = None*) → *fastoad.models.performances.mission.mission.Mission*
Builds the flight sequence from definition file.

Parameters

- **inputs** – if provided, any input parameter that is a string which matches a key of *inputs* will be replaced by the corresponding value
- **mission_name** – mission name (can be omitted if only one mission is defined or if `mission` has been defined)

Returns

get_route_names(*mission_name: Optional[str] = None*) → List[str]

Parameters **mission_name** –

Returns a list with names of all routes in the mission, in order.

get_route_ranges(*inputs: Optional[Mapping] = None*, *mission_name: Optional[str] = None*) → List[float]

Parameters

- **inputs** – if provided, any input parameter that is a string which matches a key of *inputs* will be replaced by the corresponding value
- **mission_name** – mission name (can be omitted if only one mission is defined or if `mission` has been defined)

Returns list of flight ranges for each element of the flight sequence that is a route

get_reserve(*flight_points: pandas.core.frame.DataFrame*, *mission_name: Optional[str] = None*) → float
Computes the reserve fuel according to definition in mission input file.

Parameters

- **flight_points** – the dataframe returned by compute_from() method of the instance returned by *build()*
- **mission_name** – mission name (can be omitted if only one mission is defined or if `mission` has been defined)

Returns the reserve fuel mass in kg, or 0.0 if no reserve is defined.

get_input_variables(*mission_name=None*) → *fastoad.openmdao.variables.variable_list.VariableList*

Identify variables for a defined mission.

Parameters **mission_name** – mission name (can be omitted if only one mission is defined or if `mission` has been defined)

Returns a VariableList instance.

get_unique_mission_name() → str

Provides mission name if only one mission is defined in mission file.

Returns the mission name, if only one mission is defined

Raises ***FastMissionFileMissingMissionNameError*** – if several missions are defined in mission file

get_input_weight_variable_name(*mission_name: Optional[str] = None*) → Optional[str]

Search the mission structure for a segment that has a target absolute mass defined and returns the associated variable name.

Parameters **mission_name** – mission name (can be omitted if only one mission is defined or if `mission` has been defined)

Returns The variable name, or None if no target mass found.

fastoad.models.performances.mission.mission_definition.mission_builder.structure_builders module

Classes for managing internal structures of missions.

The mission file provides a "human" definition of the mission. Structures are the translation of this human definition, that is ready to be transformed into a Python implementation.

class fastoad.models.performances.mission.mission_definition.mission_builder.structure_builders.**Abstrac**

Bases: abc.ABC

Base class for building structures of mission parts.

"Structures" are dicts that are derived from the mission definition file so that they can be readily translated into the matching implementation.

Usage:

Subclasses must implement the build method that will create the specific part of the structure dict (name and type fields are automated).

If the structure has to contain the result of another result, `insert_builder()` should be used to ensure a correct processing of the global structure, especially to get a correct resolution of `input_definitions`.

definition: dataclasses.InitVar[dict]

name: str

parent_name: str = None

variable_prefix: str = ''

type = None
Defined by subclass

property structure: dict
A dictionary that is ready to be translated to the matching implementation.

get_input_definitions() → List[*fastoad.models.performances.mission.mission_definition.mission_builder.input_definition.Input*
List of InputDefinition instances in the structure.

process_builder(*builder:* fastoad.models.performances.mission.mission_definition.mission_builder.structure_builders.AbstractStructureB
→ dict
Method to be used when another StructureBuilder object should be inserted in *structure*.

Not using this method will prevent a correct processing of `input_definitions`.

Note: The returned object is always an empty dict. It is actually a memory reference that will allow to fill this "placeholder" later with the final result of the builder, that cannot be completely known when builder is created from read definition.

Parameters **builder** – the builder object

Returns the object that has to be put at location where the builder result should be used

property qualified_name
).

Type Name of the current structure, preceded by the parent names, separated by colons (

class fastoad.models.performances.mission.mission_definition.mission_builder.structure_builders.DefaultS

Bases: *fastoad.models.performances.mission.mission_definition.mission_builder.structure_builders.AbstractStructureBuilder*

Builder for structures that do not need to process the given definition.

Parameters **definition** – the definition for the part only

type = None
Defined by subclass

definition: dataclasses.InitVar[dict]

name: str

class fastoad.models.performances.mission.mission_definition.mission_builder.structure_builders.PolarSt

Bases: *fastoad.models.performances.mission.mission_definition.mission_builder.structure_builders.AbstractStructureBuilder*

Structure builder for polar definition.

Parameters **definition** – the definition for the polar only

type = 'polar'
Defined by subclass

definition: dataclasses.InitVar[dict]

name: *str*

class fastoad.models.performances.mission.mission_definition.mission_builder.structure_builders.**SegmentS**

Bases: *fastoad.models.performances.mission.mission_definition.mission_builder.structure_builders.AbstractStructureBuilder*

Structure builder for segment definition.

Parameters **definition** – the definition for the segment only

type = 'segment'
Defined by subclass

definition: *dataclasses.InitVar[dict]*

name: *str*

class fastoad.models.performances.mission.mission_definition.mission_builder.structure_builders.**PhaseSt**

Bases: *fastoad.models.performances.mission.mission_definition.mission_builder.structure_builders.AbstractStructureBuilder*

Structure builder for phase definition.

Parameters **definition** – the whole content of definition file

type = 'phase'
Defined by subclass

definition: dataclasses.InitVar[dict]

name: str

class fastoad.models.performances.mission.mission_definition.mission_builder.structure_builders.RouteSt

Bases: *fastoad.models.performances.mission.mission_definition.mission_builder.structure_builders.AbstractStructureBuilder*

Structure builder for route definition.

Parameters **definition** – the whole content of definition file

type = 'route'
Defined by subclass

definition: dataclasses.InitVar[dict]

name: str

class fastoad.models.performances.mission.mission_definition.mission_builder.structure_builders.Mission

Bases: *fastoad.models.performances.mission.mission_definition.mission_builder.structure_builders.AbstractStructureBuilder*

Structure builder for mission definition.

Parameters **definition** – the whole content of definition file

property qualified_name
).

Type Name of the current structure, preceded by the parent names, separated by colons (

type = 'mission'
Defined by subclass

definition: dataclasses.InitVar[dict]

name: str

Module contents

Package for providing Python implementation from mission definition file.

Submodules

fastoad.models.performances.mission.mission_definition.exceptions module

Exceptions for mission definition.

exception fastoad.models.performances.mission.mission_definition.exceptions.FastMissionFileMissingMissionNameError
Bases: *fastoad.exceptions.FastError*

Raised when a mission definition is used without specifying the mission name.

fastoad.models.performances.mission.mission_definition.schema module

Schema for mission definition files.

class fastoad.models.performances.mission.mission_definition.schema.**MissionDefinition**(*file_path: Optional[Union[str, os.PathLike]] = None*)

Bases: collections.OrderedDict

Class for reading a mission definition from a YAML file.

Path of YAML file should be provided at instantiation, or in *load()*.

Parameters **file_path** – path of YAML file to read.

load(*file_path: Union[str, os.PathLike]*)
Loads a mission definition from provided file path.

Any existing definition will be overwritten.

Parameters **file_path** – path of YAML file to read.

force_all_block_fuel_usage(*mission_name*)
Sets target fuel consumption to variable "~:block_fuel".

Module contents

fastoad.models.performances.mission.openmdao package

Subpackages

Submodules

fastoad.models.performances.mission.openmdao.base module

Base classes for mission-related OpenMDAO components.

class fastoad.models.performances.mission.openmdao.base.**NeedsOWE**(*num_par_fd=1*, ***kwargs*)
Bases: openmdao.core.system.System

To be inherited when Operating Weight Empty variable is used.

Initialize all attributes.

initialize()
Perform any one-time initialization run at instantiation.

class fastoad.models.performances.mission.openmdao.base.**NeedsMTOW**(*num_par_fd=1*, ***kwargs*)
Bases: openmdao.core.system.System

To be inherited when Max TakeOff Weight variable is used.

Initialize all attributes.

initialize()
Perform any one-time initialization run at instantiation.

class fastoad.models.performances.mission.openmdao.base.**NeedsMFW**(*num_par_fd=1*, ***kwargs*)
Bases: openmdao.core.system.System

To be inherited when Max Fuel Weight variable is used.

Initialize all attributes.

initialize()
Perform any one-time initialization run at instantiation.

class fastoad.models.performances.mission.openmdao.base.**BaseMissionComp**(***kwargs*)
Bases: openmdao.core.system.System

Base class for mission components.

Initialize all attributes.

initialize()
Perform any one-time initialization run at instantiation.

property name_provider: enum.Enum
Enum class that provides mission variable names.

property variable_prefix: str
The prefix of variable names dedicated to the mission .

property mission_name: str
The name of considered mission.

property first_route_name: str
The name of first route (and normally the main one) in the mission.

static get_mission_definition(*mission_file_path: Optional[Union[str, fastoad.models.performances.mission.mission_definition.schema.MissionDefinition]]*) → *fastoad.models.performances.mission.mission_definition.schema.MissionDefinition*

Parameters **mission_file_path** – the file path, or an already built MissionDefinition instance. In the latter case, the returned instance will be the same object.

Returns the MissionDefinition instance built from provided mission_file_path

fastoad.models.performances.mission.openmdao.link_mtow module

OpenMDAO component for computation of sizing mission.

class fastoad.models.performances.mission.openmdao.link_mtow.**ComputeMTOW**(*output_name=None*, *input_names=None*, *vec_size=1*, *length=1*, *val=1.0*, *scaling_factors=None*, ***kwargs*)

Bases: openmdao.components.add_subtract_comp.AddSubtractComp

Computes MTOW from OWE, design payload and consumed fuel in sizing mission.

Allow user to create an addition/subtracton system with one-liner.

setup()
Declare inputs and outputs.

Available attributes: name pathname comm options

fastoad.models.performances.mission.openmdao.mission module

FAST-OAD model for mission computation.

class fastoad.models.performances.mission.openmdao.mission.**OMMission**(***kwargs*)

Bases: openmdao.core.group.Group, *fastoad.models.performances.mission.openmdao.base.BaseMissionComp*, *fastoad.models.performances.mission.openmdao.base.NeedsOWE*

Computes a mission as specified in mission input file.

Set the solvers to nonlinear and linear block Gauss–Seidel by default.

initialize()
Perform any one-time initialization run at instantiation.

setup()
Build this group.

This method should be overidden by your Group's method. The reason for using this method to add subsystem is to save memory and setup time when using your Group while running under MPI. This avoids the creation of systems that will not be used in the current process.

You may call 'add_subsystem' to add systems to this group. You may also issue connections, and set the linear and nonlinear solvers for this group level. You cannot safely change anything on children systems; use the 'configure' method instead.

Available attributes: name pathname comm options

property flight_points: pandas.core.frame.DataFrame
Dataframe that lists all computed flight point data.

class fastoad.models.performances.mission.openmdao.mission.**SpecificBurnedFuelComputation**(***kwargs*)
Bases: openmdao.core.explicitcomponent.ExplicitComponent

Computation of specific burned fuel (mission fuel / payload / mission range).

Store some bound methods so we can detect runtime overrides.

initialize()
Perform any one-time initialization run at instantiation.

property range_variable
Name of range variable.

property burned_fuel_variable
Name of burned fuel variable.

property specific_burned_fuel_variable
Name of specific burned fuel variable (mission fuel / payload / mission range).

property payload_variable
Name of payload variable.

setup()
Declare inputs and outputs.

Available attributes: name pathname comm options

compute(*inputs*, *outputs*, *discrete_inputs=None*, *discrete_outputs=None*)
Compute outputs given inputs. The model is assumed to be in an unscaled state.

Parameters

- **inputs** (*Vector*) – Unscaled, dimensional input variables read via inputs[key].
- **outputs** (*Vector*) – Unscaled, dimensional output variables read via outputs[key].
- **discrete_inputs** (*dict or None*) – If not None, dict containing discrete input values.
- **discrete_outputs** (*dict or None*) – If not None, dict containing discrete output values.

fastoad.models.performances.mission.openmdao.mission_run module

class fastoad.models.performances.mission.openmdao.mission_run.**MissionComp**(***kwargs*)
Bases: openmdao.core.explicitcomponent.ExplicitComponent, *fastoad.models.performances.mission.openmdao.base.BaseMissionComp*

Computes a mission as specified in mission input file.

Store some bound methods so we can detect runtime overrides.

initialize()
Perform any one-time initialization run at instantiation.

setup()
Declare inputs and outputs.

Available attributes: name pathname comm options

setup_partials()
Declare partials.

This is meant to be overridden by component classes. All partials should be declared here since this is called after all size/shape information is known for all variables.

compute(*inputs*, *outputs*, *discrete_inputs=None*, *discrete_outputs=None*)
Compute outputs given inputs. The model is assumed to be in an unscaled state.

Parameters

- **inputs** (*Vector*) – Unscaled, dimensional input variables read via inputs[key].
- **outputs** (*Vector*) – Unscaled, dimensional output variables read via outputs[key].
- **discrete_inputs** (*dict or None*) – If not None, dict containing discrete input values.
- **discrete_outputs** (*dict or None*) – If not None, dict containing discrete output values.

get_engine_wrapper() → Optional[*fastoad.model_base.propulsion.IOMPropulsionWrapper*]
Overloading this method allows to define the engine without relying on the propulsion option.

(useful for tests)

Returns the engine wrapper instance

class fastoad.models.performances.mission.openmdao.mission_run.**AdvancedMissionComp**(***kwargs*)
Bases: *fastoad.models.performances.mission.openmdao.mission_run.MissionComp*

Computes a mission as specified in mission input file.

Compared to *MissionComp*, it allows:

- to use an initializer iteration (simple Breguet) at first call.
- to use the mission as the design mission for the sizing process.

Store some bound methods so we can detect runtime overrides.

initialize()
Perform any one-time initialization run at instantiation.

setup()
Declare inputs and outputs.

Available attributes: name pathname comm options

compute(*inputs*, *outputs*, *discrete_inputs=None*, *discrete_outputs=None*)
Compute outputs given inputs. The model is assumed to be in an unscaled state.

Parameters

- **inputs** (*Vector*) – Unscaled, dimensional input variables read via inputs[key].
- **outputs** (*Vector*) – Unscaled, dimensional output variables read via outputs[key].
- **discrete_inputs** (*dict or None*) – If not None, dict containing discrete input values.

- **discrete_outputs** (*dict or None*) – If not None, dict containing discrete output values.

fastoad.models.performances.mission.openmdao.mission_wrapper module

Mission wrapper.

class fastoad.models.performances.mission.openmdao.mission_wrapper.**MissionWrapper**(*mission_definition: Union[str, fastoad.models.performances.m*, *propulsion: Optional[fastoad.model_base.p* = None, *reference_area: Optional[float]* = None, *mission_name: Optional[str]* = None, *variable_prefix: str* = *'data:mission'*, *force_all_block_fuel_usage: bool* = *False*)

Bases: *fastoad.models.performances.mission.mission_definition.mission_builder.mission_builder.MissionBuilder*

Wrapper around `MissionBuilder` for using with OpenMDAO.

Unlike its parent class, the *mission_name* argument is mandatory at instantiation, unless there is only one mission in the definition file.

Parameters

- **mission_definition** – a file path or MissionDefinition instance
- **propulsion** – if not provided, the property `propulsion` must be set before calling `build()`
- **reference_area** – if not provided, the property `reference_area` must be set before calling `build()`
- **mission_name** – name of chosen mission. Can be omitted if definition file contains only one mission.
- **variable_prefix** – prefix for auto-generated variable names.

- **force_all_block_fuel_usage** – if True and if *mission_name* is provided, the mission definition will be modified to set the target fuel consumption to variable "~:block_fuel"

force_all_block_fuel_usage()
Modifies mission definition to set block fuel as target fuel consumption.

setup(*component: openmdao.core.explicitcomponent.ExplicitComponent*)
To be used during setup() of provided OpenMDAO component.

It adds input and output variables deduced from mission definition file.

Parameters **component** – the OpenMDAO component where the setup is done.

compute(*start_flight_point:* fastoad.model_base.flight_point.FlightPoint, *inputs: openmdao.vectors.vector.Vector*, *outputs: openmdao.vectors.vector.Vector*) → pandas.core.frame.DataFrame
To be used during compute() of an OpenMDAO component.

Builds the mission from input file, and computes it. *outputs* vector is filled with duration, burned fuel and covered ground distance for each part of the flight.

Parameters

- **start_flight_point** – starting point of mission
- **inputs** – the input vector of the OpenMDAO component
- **outputs** – the output vector of the OpenMDAO component

Returns a pandas DataFrame where column names match fields of *FlightPoint*

get_reserve_variable_name() → str

Returns the name of OpenMDAO variable for fuel reserve. This name is among the declared outputs in *setup()*.

fastoad.models.performances.mission.openmdao.payload_range module

Payload-Range diagram computation.

class fastoad.models.performances.mission.openmdao.payload_range.**PayloadRange**(***kwargs*)
Bases: openmdao.core.group.Group, *fastoad.models.performances.mission.openmdao.base.BaseMissionComp*, *fastoad.models.performances.mission.openmdao.base.NeedsOWE*, *fastoad.models.performances.mission.openmdao.base.NeedsMTOW*, *fastoad.models.performances.mission.openmdao.base.NeedsMFW*

OpenMDAO component for computing data for payload-range plots.

Set the solvers to nonlinear and linear block Gauss–Seidel by default.

initialize()
Perform any one-time initialization run at instantiation.

setup()
Build this group.

This method should be overidden by your Group's method. The reason for using this method to add subsystem is to save memory and setup time when using your Group while running under MPI. This avoids the creation of systems that will not be used in the current process.

You may call 'add_subsystem' to add systems to this group. You may also issue connections, and set the linear and nonlinear solvers for this group level. You cannot safely change anything on children systems; use the 'configure' method instead.

Available attributes: name pathname comm options

class fastoad.models.performances.mission.openmdao.payload_range.**PayloadRangeContourInputValues**(***kwargs*)

Bases: openmdao.core.explicitcomponent.ExplicitComponent, *fastoad.models.performances.mission.openmdao.base.BaseMissionComp*, *fastoad.models.performances.mission.openmdao.base.NeedsOWE*, *fastoad.models.performances.mission.openmdao.base.NeedsMTOW*, *fastoad.models.performances.mission.openmdao.base.NeedsMFW*

This class provides input values for missions that will compute the contour of the payload-range diagram.

Store some bound methods so we can detect runtime overrides.

initialize()

Perform any one-time initialization run at instantiation.

setup()

Declare inputs and outputs.

Available attributes: name pathname comm options

compute(*inputs*, *outputs*, *discrete_inputs=None*, *discrete_outputs=None*)

Compute outputs given inputs. The model is assumed to be in an unscaled state.

Parameters

- **inputs** (*Vector*) – Unscaled, dimensional input variables read via inputs[key].
- **outputs** (*Vector*) – Unscaled, dimensional output variables read via outputs[key].
- **discrete_inputs** (*dict or None*) – If not None, dict containing discrete input values.
- **discrete_outputs** (*dict or None*) – If not None, dict containing discrete output values.

class fastoad.models.performances.mission.openmdao.payload_range.**PayloadRangeGridInputValues**(***kwargs*)

Bases: openmdao.core.explicitcomponent.ExplicitComponent, *fastoad.models.performances.mission.openmdao.base.BaseMissionComp*, *fastoad.models.performances.mission.openmdao.base.NeedsOWE*

This class provides input values for missions that will compute points inside the contour of the payload-range diagram.

Store some bound methods so we can detect runtime overrides.

initialize()

Perform any one-time initialization run at instantiation.

setup()

Declare inputs and outputs.

Available attributes: name pathname comm options

compute(*inputs*, *outputs*, *discrete_inputs=None*, *discrete_outputs=None*)

Compute outputs given inputs. The model is assumed to be in an unscaled state.

Parameters

- **inputs** (*Vector*) – Unscaled, dimensional input variables read via inputs[key].
- **outputs** (*Vector*) – Unscaled, dimensional output variables read via outputs[key].

- **discrete_inputs** (*dict or None*) – If not None, dict containing discrete input values.
- **discrete_outputs** (*dict or None*) – If not None, dict containing discrete output values.

Module contents

fastoad.models.performances.mission.segments package

Subpackages

fastoad.models.performances.mission.segments.registered package

Subpackages

fastoad.models.performances.mission.segments.registered.takeoff package

Submodules

fastoad.models.performances.mission.segments.registered.takeoff.end_of_takeoff module

Classes for climb/descent segments.

class fastoad.models.performances.mission.segments.registered.takeoff.end_of_takeoff.EndOfTakeoffSegmen

AbstractTakeOffSegment

Computes a flight path segment where altitude is modified with constant pitch angle. As a result, the slope angle and angle of attack are changing through time. Updates are based on longitudinal dynamics equations simplifies with the assumption of constant pitch angle.

Note: Setting target

Target is an altitude and should be set to the safety altitude.

---

compute_next_flight_point(*flight_points: List[*fastoad.model_base.flight_point.FlightPoint*]*, *time_step: float*) → *fastoad.model_base.flight_point.FlightPoint*

Computes time, altitude, speed, mass and ground distance of next flight point.

Parameters

- **flight_points** – previous flight points
- **time_step** – time step for computing next point

Returns the computed next flight point

complete_flight_point(*flight_point:* fastoad.model_base.flight_point.FlightPoint)

Redefinition, computes data for provided flight point.

Assumes that it is already defined for time, altitude, mass, ground distance and speed (TAS, EAS, or Mach).

Parameters **flight_point** – the flight point that will be completed in-place

get_distance_to_target(*flight_points: List[*fastoad.model_base.flight_point.FlightPoint*]*, *target:* fastoad.model_base.flight_point.FlightPoint) → float

Computes a "distance" from last flight point to target.

Computed does not need to have a real meaning. The important point is that it must be signed so that algorithm knows on which "side" of the target we are. And of course, it should be 0. if flight point is on target.

Parameters

- **flight_points** – list of all currently computed flight_points
- **target** – segment target (will not contain relative values)

Returns

O. if target is attained, a non-null value otherwise

get_next_alpha(*previous_point:* fastoad.model_base.flight_point.FlightPoint, *time_step: float*) → float

Computes angle of attack (alpha) based on gamma_dot, using constant pitch angle assumption.

Parameters

- **previous_point** – the flight point from which next alpha is computed
- **time_step** – the duration between computed flight point and previous_point

static compute_next_gamma(*next_point:* fastoad.model_base.flight_point.FlightPoint, *previous_point:* fastoad.model_base.flight_point.FlightPoint)

Computes slope angle (gamma) based on gamma_dot

Parameters

- **next_point** – the next flight point
- **previous_point** – the flight point from which next gamma is computed

get_gamma_and_acceleration(*flight_point:* fastoad.model_base.flight_point.FlightPoint)
Redefinition : computes slope angle derivative (gamma_dot) and x-acceleration. Replaces CL, CD, lift dan drag values (for ground effect and accelerated flight)

Parameters **flight_point** – parameters after propulsion model has been called (i.e. mass, thrust and drag are available)

property target: *fastoad.model_base.flight_point.FlightPoint*
The base class of the class hierarchy.

When called, it accepts no arguments and returns a new featureless instance that has no instance attributes and cannot be given any.

fastoad.models.performances.mission.segments.registered.takeoff.rotation module

Classes for acceleration/deceleration segments.

class fastoad.models.performances.mission.segments.registered.takeoff.rotation.**RotationSegment**(*name: str = '', target: fastoad.model = <object object>, isa_offset: float = 0.0, propulsion: fastoad.model = <object object>, polar: fastoad.model = <object object>, polar_modifie fastoad.model = UnchangedPolar(), reference_area: float = <object object>, time_step: float = 0.1, maximum_CL:

AbstractGroundSegment

Computes a flight path segment with constant rotation rate while on ground and accelerating.

The target is the lift-off. A protection is included is the aircraft reaches alpha_limit (tail-strike).

rotation_rate: float = 0.05235987755982989
Rotation rate in radians/s, i.e. derivative of angle of attack. Default value is CS-25 specification.

alpha_limit: float = 0.23561944901923448
Angle of attack (in radians) where tail strike is expected. Default value is good for SMR aircraft.

get_distance_to_target(*flight_points: List[*fastoad.model_base.flight_point.FlightPoint*]*, *target:* fastoad.model_base.flight_point.FlightPoint) → float
Computes a "distance" from last flight point to target.

Computed does not need to have a real meaning. The important point is that it must be signed so that algorithm knows on which "side" of the target we are. And of course, it should be 0. if flight point is on target.

Parameters

- **flight_points** – list of all currently computed flight_points
- **target** – segment target (will not contain relative values)

Returns

O. if target is attained, a non-null value otherwise

get_next_alpha(*previous_point:* fastoad.model_base.flight_point.FlightPoint, *time_step: float*) → float
Determine the next AoA based on imposed rotation rate.

Parameters

- **previous_point** – the flight point from which next alpha is computed
- **time_step** – the duration between computed flight point and previous_point

property target: *fastoad.model_base.flight_point.FlightPoint*
The base class of the class hierarchy.

When called, it accepts no arguments and returns a new featureless instance that has no instance attributes and cannot be given any.

fastoad.models.performances.mission.segments.registered.takeoff.takeoff module

Class for takeoff sequence

class fastoad.models.performances.mission.segments.registered.takeoff.takeoff.**TakeOffSequence**(*name: str = '', target: fastoad.model_ = <object object>, isa_offset: float = 0.0, propulsion: fastoad.model_ = <object object>, polar: fastoad.models = <object object>, polar_modifier fastoad.models = UnchangedPolar(), reference_area: float = <object object>, time_step: float = 0.1, maximum_CL:*

types.TakeOffSequence___Base

This class does a time-step simulation of a full takeoff:

- ground speed acceleration up to *rotation_equivalent_airspeed*
- rotation
- climb up to altitude provided in `target` (safety altitude)

rotation_equivalent_airspeed: *float* **= <object object>**
Equivalent airspeed to reach for starting aircraft rotation.

rotation_alpha_limit: *float* **= 0.23561944901923448**
Angle of attack (in radians) where tail strike is expected. Default value is good for SMR aircraft.

end_time_step: *float* **= 0.05**

build_sequence()
Instantiates all segments, using dataclass field values of this macro-segment.

Note: this method is called each time a dataclass field value is modified.

cls_sequence = [<class 'fastoad.models.performances.mission.segments.registered.ground_speed_change.GroundSpeedChangeSegment'>, <class 'fastoad.models.performances.mission.segments.registered.takeoff.rotation.RotationSegment'>, <class 'fastoad.models.performances.mission.segments.registered.takeoff.end_of_takeoff.EndOfTakeoffSegment'>]
List of segment classes that will compose this macro-segment.

Module contents

Classes for simulating takeoff-related flight segments.

Be sure to import this package before interpreting a mission input file.

Submodules

fastoad.models.performances.mission.segments.registered.altitude_change module

Classes for climb/descent segments.

class fastoad.models.performances.mission.segments.registered.altitude_change.**AltitudeChangeSegment**(*name=''*, *target: fastoad.=<object object>*, *isa_offset=0.0*, *propulsion: fastoad.=<object object>*, *polar: fastoad.=<object object>*, *polar_reference=UnchangedPolar()*, *time_step=2.0*, *maximum_

AbstractManualThrustSegment

Computes a flight path segment where altitude is modified with constant speed.

Note: Setting speed

Constant speed may be:

- constant true airspeed (TAS)
- constant equivalent airspeed (EAS)
- constant Mach number

Target should have `"constant"` as definition for one parameter among `true_airspeed`, `equivalent_airspeed` or `mach`. All computed flight points will use the corresponding **start** value. The two other speed values will be computed accordingly.

If not "constant" parameter is set, constant TAS is assumed.

Note: Setting target

Target can be an altitude, or a speed:

- Target altitude can be a float value (in **meters**), or can be set to:
  - *OPTIMAL_ALTITUDE*: in that case, the target altitude will be the altitude where maximum lift/drag ratio is achieved for target speed, depending on current mass.
  - *OPTIMAL_FLIGHT_LEVEL*: same as above, except that altitude will be rounded to the nearest flight level (multiple of 100 feet).
- For a speed target, as explained above, one value TAS, EAS or Mach must be `"constant"`. One of the two other ones can be set as target.

In any case, the achieved value will be capped so it respects *maximum_flight_level*.

---

time_step: float = 2.0
Used time step for computation (actual time step can be lower at some particular times of the flight path).

maximum_flight_level: float = 500.0
The maximum allowed flight level (i.e. multiple of 100 feet).

OPTIMAL_ALTITUDE = 'optimal_altitude'
Using this value will tell to target the altitude with max lift/drag ratio.

OPTIMAL_FLIGHT_LEVEL = 'optimal_flight_level'
Using this value will tell to target the nearest flight level to altitude with max lift/drag ratio.

compute_from_start_to_target(*start:* fastoad.model_base.flight_point.FlightPoint, *target:* fastoad.model_base.flight_point.FlightPoint) → pandas.core.frame.DataFrame
Here should come the implementation for computing flight points between start and target flight points.

Parameters

- **start** –
- **target** – Definition of segment target

Returns a pandas DataFrame where column names match fields of *FlightPoint*

get_distance_to_target(*flight_points: List[*fastoad.model_base.flight_point.FlightPoint*]*, *target:* fastoad.model_base.flight_point.FlightPoint) → float

Computes a "distance" from last flight point to target.

Computed does not need to have a real meaning. The important point is that it must be signed so that algorithm knows on which "side" of the target we are. And of course, it should be 0. if flight point is on target.

Parameters

- **flight_points** – list of all currently computed flight_points
- **target** – segment target (will not contain relative values)

Returns

0. if target is attained, a non-null value otherwise

get_gamma_and_acceleration(*flight_point:* fastoad.model_base.flight_point.FlightPoint) → Tuple[float, float]

Computes slope angle (gamma) and acceleration.

Parameters **flight_point** – parameters after propulsion model has been called (i.e. mass, thrust and drag are available)

Returns slope angle in radians and acceleration in m**2/s

property target: *fastoad.model_base.flight_point.FlightPoint*

The base class of the class hierarchy.

When called, it accepts no arguments and returns a new featureless instance that has no instance attributes and cannot be given any.

fastoad.models.performances.mission.segments.registered.cruise module

Classes for simulating cruise segments.

class fastoad.models.performances.mission.segments.registered.cruise.**CruiseSegment**(*name: str = ''*, *target: fastoad.model_base.flight_po = <object object>*, *isa_offset: float = 0.0*, *propulsion: fastoad.model_base.propulsio = <object object>*, *polar: fastoad.models.performances = <object object>*, *polar_modifier: fastoad.models.performances = UnchangedPolar()*, *reference_area: float = <object object>*, *time_step: float = 60.0*, *maximum_CL: Optional[float] = None*, *altitude_bounds: tuple = (-500.0, 40000.0)*, *mach_bounds: tuple = (-1e-06, 5.0)*, *interrupt_if_getting_further_fro bool = True*, *engine_setting: fas-

AbstractRegulatedThrustSegment

Class for computing cruise flight segment at constant altitude and speed.

Mach is considered constant, equal to Mach at starting point. Altitude is constant. Target is a specified ground_distance. The target definition indicates the ground_distance to be covered during the segment, independently of the initial value.

get_distance_to_target(*flight_points: List[*fastoad.model_base.flight_point.FlightPoint*]*, *target:* fastoad.model_base.flight_point.FlightPoint) → float

Computes a "distance" from last flight point to target.

Computed does not need to have a real meaning. The important point is that it must be signed so that algorithm knows on which "side" of the target we are. And of course, it should be 0. if flight point is on target.

Parameters

- **flight_points** – list of all currently computed flight_points
- **target** – segment target (will not contain relative values)

Returns

O. if target is attained, a non-null value otherwise

property target: *fastoad.model_base.flight_point.FlightPoint*

The base class of the class hierarchy.

When called, it accepts no arguments and returns a new featureless instance that has no instance attributes and cannot be given any.

class fastoad.models.performances.mission.segments.registered.cruise.**OptimalCruiseSegment**(*name: str = '', target: fastoad.model_base. = <object object>, isa_offset: float = 0.0, propulsion: fastoad.model_base. = <object object>, polar: fastoad.models.perfo = <object object>, polar_modifier: fastoad.models.perfo = UnchangedPolar(), reference_area: float = <object object>, time_step: float = 60.0, maximum_CL:*

Class for computing cruise flight segment at maximum lift/drag ratio.

Altitude is set **at every point** to get the optimum CL according to current mass. Target is a specified ground_distance. The target definition indicates the ground_distance to be covered during the segment, independently of the initial value. Target should also specify a speed parameter set to "constant", among *mach*, *true_airspeed* and *equivalent_airspeed*. If not, Mach will be assumed constant.

compute_from_start_to_target(*start:* fastoad.model_base.flight_point.FlightPoint, *target:* fastoad.model_base.flight_point.FlightPoint) → pandas.core.frame.DataFrame

Here should come the implementation for computing flight points between start and target flight points.

Parameters

- **start** –
- **target** – Definition of segment target

Returns a pandas DataFrame where column names match fields of *FlightPoint*

property target: *fastoad.model_base.flight_point.FlightPoint*

The base class of the class hierarchy.

When called, it accepts no arguments and returns a new featureless instance that has no instance attributes and cannot be given any.

class fastoad.models.performances.mission.segments.registered.cruise.**ClimbAndCruiseSegment**(*name: str = '', target: fastoad.model_bas = <object object>, isa_offset: float = 0.0, propulsion: fastoad.model_bas = <object object>, polar: fastoad.models.per = <object object>, polar_modifier: fastoad.models.per = UnchangedPolar(), reference_area: float = <object object>, time_step: float = 60.0, maximum_CL:*

Class for computing cruise flight segment at constant altitude.

Target is a specified ground_distance. The target definition indicates the ground_distance to be covered during the segment, independently of the initial value. Target should also specify a speed parameter set to "constant", among *mach*, *true_airspeed* and *equivalent_airspeed*. If not, Mach will be assumed constant.

Target altitude can also be set to *OPTIMAL_FLIGHT_LEVEL*. In that case, the cruise will be preceded by a climb segment and *climb_segment* must be set at instantiation.

(Target ground distance will be achieved by the sum of ground distances covered during climb and cruise)

In this case, climb will be done up to the IFR Flight Level (as multiple of 100 feet) that ensures minimum mass decrease, while being at most equal to *maximum_flight_level*.

climb_segment: *fastoad.models.performances.mission.segments.registered.altitude_change.AltitudeChangeSegment* **= None**
The AltitudeChangeSegment that can be used if a preliminary climb is needed (its target will be ignored).

maximum_flight_level: *float* **= 500.0**
The maximum allowed flight level (i.e. multiple of 100 feet).

compute_from_start_to_target(*start:* fastoad.model_base.flight_point.FlightPoint, *target:* fastoad.model_base.flight_point.FlightPoint) → pandas.core.frame.DataFrame
Here should come the implementation for computing flight points between start and target flight points.

Parameters

- **start** –
- **target** – Definition of segment target

Returns a pandas DataFrame where column names match fields of *FlightPoint*

property target: *fastoad.model_base.flight_point.FlightPoint*
The base class of the class hierarchy.

When called, it accepts no arguments and returns a new featureless instance that has no instance attributes and cannot be given any.

class fastoad.models.performances.mission.segments.registered.cruise.**BreguetCruiseSegment**(*name: str = '', target: fastoad.model_base. = <object object>, isa_offset: float = 0.0, propulsion: fastoad.model_base. = <object object>, polar: fastoad.models.perfo = <object object>, polar_modifier: fastoad.models.perfo = UnchangedPolar(), reference_area: float = 1.0, time_step: float = 60.0, maximum_CL: Optional[float] =*

Class for computing cruise flight segment at constant altitude using Breguet-Leduc formula.

As formula relies on SFC, the `propulsion` model must be able to fill FlightPoint.sfc when FlightPoint.thrust is provided.

use_max_lift_drag_ratio: *bool* **= False**
if True, max lift/drag ratio will be used instead of the one computed with polar using CL deduced from mass and altitude. In this case, reference_area parameter will be unused

reference_area: *float* **= 1.0**
The reference area, in m**2. Used only if use_max_lift_drag_ratio is False.

compute_from_start_to_target(*start:* fastoad.model_base.flight_point.FlightPoint, *target:* fastoad.model_base.flight_point.FlightPoint) → pandas.core.frame.DataFrame
Here should come the implementation for computing flight points between start and target flight points.

Parameters

- **start** –
- **target** – Definition of segment target

Returns a pandas DataFrame where column names match fields of *FlightPoint*

property target: *fastoad.model_base.flight_point.FlightPoint*
The base class of the class hierarchy.

When called, it accepts no arguments and returns a new featureless instance that has no instance attributes and cannot be given any.

fastoad.models.performances.mission.segments.registered.ground_speed_change module

Classes for acceleration/deceleration segments.

class fastoad.models.performances.mission.segments.registered.ground_speed_change.GroundSpeedChangeSegm

AbstractGroundSegment

Computes a flight path segment where aircraft is accelerated or de-accelerated on the ground

The target must define an airspeed (equivalent, true or Mach) value.

get_distance_to_target(*flight_points: List[*fastoad.model_base.flight_point.FlightPoint*]*, *target:* fastoad.model_base.flight_point.FlightPoint) → float

Computes a "distance" from last flight point to target.

Computed does not need to have a real meaning. The important point is that it must be signed so that algorithm knows on which "side" of the target we are. And of course, it should be 0. if flight point is on target.

Parameters

- **flight_points** – list of all currently computed flight_points
- **target** – segment target (will not contain relative values)

Returns

O. if target is attained, a non-null value otherwise

property target: *fastoad.model_base.flight_point.FlightPoint*

The base class of the class hierarchy.

When called, it accepts no arguments and returns a new featureless instance that has no instance attributes and cannot be given any.

fastoad.models.performances.mission.segments.registered.hold module

Class for simulating hold segment.

class fastoad.models.performances.mission.segments.registered.hold.**HoldSegment**(*name: str = '', target: fastoad.model_base.flight_point.FlightPoint = <object object>, isa_offset: float = 0.0, propulsion: fastoad.model_base.propulsion.IPropulsion = <object object>, polar: fastoad.models.performances.mission.polar.Polar = <object object>, polar_modifier: fastoad.models.performances.mission.polar.PolarModifier = UnchangedPolar(), reference_area: float = <object object>, time_step: float = 60.0, maximum_CL: Optional[float] = None, altitude_bounds: tuple = (-500.0, 40000.0), mach_bounds: tuple = (-1e-06, 5.0), interrupt_if_getting_further_from_target: bool = True, engine_setting: fastoad.constants.EngineSetting = EngineSetting.CLIMB*)

Bases: *fastoad.models.performances.mission.segments.time_step_base.AbstractRegulatedThrustSegment*, *fastoad.models.performances.mission.segments.time_step_base.AbstractFixedDurationSegment*

Class for computing hold flight segment.

Mach is considered constant, equal to Mach at starting point. Altitude is constant. Target is a specified time. The target definition indicates the time duration of the segment, independently of the initial time value.

property target: *fastoad.model_base.flight_point.FlightPoint*

The base class of the class hierarchy.

When called, it accepts no arguments and returns a new featureless instance that has no instance attributes and cannot be given any.

fastoad.models.performances.mission.segments.registered.mass_input module

Class for specifying input mass at "any" point in the mission.

class fastoad.models.performances.mission.segments.registered.mass_input.**MassTargetSegment**(*name: str = '', target: fastoad.model_bas = <object object>, isa_offset: float = 0.0*)

Bases: *fastoad.models.performances.mission.segments.base.AbstractFlightSegment*

Class that simply sets a target mass.

`compute_from()` returns a 1-row dataframe that is the start point with mass set to provided target mass.

class:*~fastoad.models.performances.mission.base.FlightSequence* ensures that mass is consistent for segments prior to this one.

compute_from_start_to_target(*start:* fastoad.model_base.flight_point.FlightPoint, *target:* fastoad.model_base.flight_point.FlightPoint) → pandas.core.frame.DataFrame

Here should come the implementation for computing flight points between start and target flight points.

Parameters

- **start** –
- **target** – Definition of segment target

Returns a pandas DataFrame where column names match fields of *FlightPoint*

property target: *fastoad.model_base.flight_point.FlightPoint*

The base class of the class hierarchy.

When called, it accepts no arguments and returns a new featureless instance that has no instance attributes and cannot be given any.

fastoad.models.performances.mission.segments.registered.speed_change module

Classes for acceleration/deceleration segments.

class fastoad.models.performances.mission.segments.registered.speed_change.**SpeedChangeSegment**(*name: str = '', target: fastoad.model_ = <object object>, isa_offset: float = 0.0, propulsion: fastoad.model_ = <object object>, polar: fastoad.models = <object object>, polar_modifier fastoad.models = UnchangedPolar(), reference_area: float = <object object>, time_step: float = 0.2, maximum_CL:*

AbstractManualThrustSegment

Computes a flight path segment where speed is modified with no change in altitude.

The target must define a speed value among true_airspeed, equivalent_airspeed and mach.

get_distance_to_target(*flight_points: List[*fastoad.model_base.flight_point.FlightPoint*]*, *target:* fastoad.model_base.flight_point.FlightPoint) → float

Computes a "distance" from last flight point to target.

Computed does not need to have a real meaning. The important point is that it must be signed so that algorithm knows on which "side" of the target we are. And of course, it should be 0. if flight point is on target.

Parameters

- **flight_points** – list of all currently computed flight_points
- **target** – segment target (will not contain relative values)

Returns

0. if target is attained, a non-null value otherwise

get_gamma_and_acceleration(*flight_point:* fastoad.model_base.flight_point.FlightPoint) → Tuple[float, float]

Computes slope angle (gamma) and acceleration.

Parameters **flight_point** – parameters after propulsion model has been called (i.e. mass, thrust and drag are available)

Returns slope angle in radians and acceleration in m**2/s

property target: *fastoad.model_base.flight_point.FlightPoint*

The base class of the class hierarchy.

When called, it accepts no arguments and returns a new featureless instance that has no instance attributes and cannot be given any.

fastoad.models.performances.mission.segments.registered.start module

Class for mission start point.

class fastoad.models.performances.mission.segments.registered.start.**Start**(*name: str = ''*, *target: fastoad.model_base.flight_point.FlightPo = <object object>*, *isa_offset: float = 0.0*)

Bases: *fastoad.models.performances.mission.segments.base.AbstractFlightSegment*

Provides a starting point for a mission.

`compute_from()` will return only 1 flight points that matches the target.

compute_from_start_to_target(*start:* fastoad.model_base.flight_point.FlightPoint, *target:* fastoad.model_base.flight_point.FlightPoint) → pandas.core.frame.DataFrame

Here should come the implementation for computing flight points between start and target flight points.

Parameters

- **start** –

- **target** – Definition of segment target

Returns a pandas DataFrame where column names match fields of *FlightPoint*

property target: *fastoad.model_base.flight_point.FlightPoint*

The base class of the class hierarchy.

When called, it accepts no arguments and returns a new featureless instance that has no instance attributes and cannot be given any.

fastoad.models.performances.mission.segments.registered.taxi module

Classes for Taxi sequences.

class fastoad.models.performances.mission.segments.registered.taxi.**TaxiSegment**(*name: str = '', target: fastoad.model_base.flight_point.Fl = <object object>, isa_offset: float = 0.0, propulsion: fastoad.model_base.propulsion.IPr = <object object>, polar: Optional[fastoad.models.performa = None, polar_modifier: fastoad.models.performances.miss = UnchangedPolar(), reference_area: float = 1.0, time_step: float = 60.0, maximum_CL: Optional[float] = None, altitude_bounds: tuple = (-500.0, 40000.0), mach_bounds: tuple = (-1e-06, 5.0), interrupt_if_getting_further_from_ta bool = True, engine_setting: fastoad.constants.EngineSetting = EngineSetting.CLIMB, thrust_rate: float = 1.0, true_airspeed: float = 0.0*)

Bases: *fastoad.models.performances.mission.segments.time_step_base.AbstractManualThrustSegment*, *fastoad.models.performances.mission.segments.time_step_base.AbstractFixedDurationSegment*

Class for computing Taxi phases.

Taxi phase has a target duration (target.time should be provided) and is at constant altitude, speed and thrust rate.

polar: *fastoad.models.performances.mission.polar.Polar* **= None**
The Polar instance that will provide drag data.

reference_area: float **= 1.0**
The reference area, in m**2.

time_step: float **= 60.0**
Used time step for computation (actual time step can be lower at some particular times of the flight path).

true_airspeed: float **= 0.0**

get_gamma_and_acceleration(*flight_point:* fastoad.model_base.flight_point.FlightPoint) → Tuple[float, float]
Computes slope angle (gamma) and acceleration.

Parameters **flight_point** – parameters after propulsion model has been called (i.e. mass, thrust and drag are available)

Returns slope angle in radians and acceleration in m**2/s

compute_from_start_to_target(*start:* fastoad.model_base.flight_point.FlightPoint, *target:* fastoad.model_base.flight_point.FlightPoint) → pandas.core.frame.DataFrame
Here should come the implementation for computing flight points between start and target flight points.

Parameters

- **start** –
- **target** – Definition of segment target

Returns a pandas DataFrame where column names match fields of *FlightPoint*

property target: *fastoad.model_base.flight_point.FlightPoint*
The base class of the class hierarchy.

When called, it accepts no arguments and returns a new featureless instance that has no instance attributes and cannot be given any.

fastoad.models.performances.mission.segments.registered.transition module

Class for very simple transition in some flight phases.

class fastoad.models.performances.mission.segments.registered.transition.**DummyTransitionSegment**(*name: str = '', target: fastoad.mod = <object object>, isa_offset float = 0.0, mass_rati float = 1.0, reserve_ma float = 0.0*)

Bases: *fastoad.models.performances.mission.segments.base.AbstractFlightSegment*

Computes a transient flight part in a very quick and dummy way.

`compute_from()` will return only 2 or 3 flight points.

The second flight point is the end of transition. Its parameters are equal to those provided in `target`.

There is an exception if target does not specify any mass (i.e. self.target.mass == 0). Then the mass of the second flight point is the start mass multiplied by *mass_ratio*.

If *reserve_mass_ratio* is non-zero, a third flight point is added, with parameters equal to flight_point(2), except for mass where:

mass(2) - reserve_mass_ratio * mass(3) = mass(3).

In different words, mass(3) would be the Zero Fuel Weight (ZFW) and reserve can be expressed as a percentage of ZFW.

mass_ratio: `float = 1.0`
The ratio (aircraft mass at END of segment)/(aircraft mass at START of segment)

reserve_mass_ratio: `float = 0.0`
The ratio (fuel mass)/(aircraft mass at END of segment) that will be consumed at end of segment.

compute_from_start_to_target(*start:* fastoad.model_base.flight_point.FlightPoint, *target:* fastoad.model_base.flight_point.FlightPoint) → pandas.core.frame.DataFrame
Here should come the implementation for computing flight points between start and target flight points.

Parameters

- **start** –

- **target** – Definition of segment target

Returns a pandas DataFrame where column names match fields of *FlightPoint*

property target: *fastoad.model_base.flight_point.FlightPoint*

The base class of the class hierarchy.

When called, it accepts no arguments and returns a new featureless instance that has no instance attributes and cannot be given any.

Module contents

Classes for simulating flight segments.

Be sure to import this package before interpreting a mission input file.

Submodules

fastoad.models.performances.mission.segments.base module

Base classes for simulating flight segments.

class fastoad.models.performances.mission.segments.base.**RegisterSegment**(*keyword=''*)

Bases: *fastoad.models.performances.mission.base.RegisterElement*

Decorator for registering IFlightPart classes.

```
>>> @RegisterSegment("segment_foo")
>>> class FooSegment(IFlightPart):
>>>     ...
```

Then the registered class can be obtained by:

```
>>> my_class = RegisterSegment.get_class("segment_foo")
```

class fastoad.models.performances.mission.segments.base.**SegmentDefinitions**(**args*, ***kwargs*)

Bases: object

Class that associates segment names (mission file keywords) and their implementation.

classmethod add_segment(*segment_name: str*, *segment_class: Type[*fastoad.models.performances.mission.base.IFlightPart*]*)

Adds a segment definition.

Parameters

- **segment_name** – segment names (mission file keyword)
- **segment_class** – segment implementation (derived of FlightSegment)

classmethod get_segment_class(*segment_name*) → Optional[Type[*fastoad.models.performances.mission.base.IFlightPart*]]

Provides the segment implementation for provided name.

Parameters **segment_name** –

Returns the segment implementation (derived of FlightSegment)

Raises **FastUnknownMissionSegmentError** – if segment type has not been declared.

class fastoad.models.performances.mission.segments.base.**RegisteredSegment**(**args*, ***kwargs*)

Bases: *fastoad.models.performances.mission.base.IFlightPart*, abc.ABC

Base class for classes that can be associated with a keyword in mission definition file.

When subclassing this class, the attribute "mission_file_keyword" can be set, so that the segment can be used in mission file definition with this keyword:

```
>>> class NewSegment(AbstractFlightSegment, mission_file_keyword="new_segment")
>>>     ...
```

Then in mission definition:

```
phases:
    my_phase:
        parts:
            - segment: new_segment
```

target: *fastoad.model_base.flight_point.FlightPoint*

class fastoad.models.performances.mission.segments.base.**AbstractFlightSegment**(*name: str = ''*, *target: fastoad.model_base.flight_point.Flig = <object object>*, *isa_offset: float = 0.0*)

Bases: *fastoad.models.performances.mission.base.IFlightPart*, abc.ABC

Base class for flight path segment.

As a dataclass, attributes can be set at instantiation.

Important: *compute_from()* is the method to call to achieve the segment computation.

However, when subclassing, the method to overload is *compute_from_start_to_target()*. Generic reprocessing of start and target flight points is done in *compute_from()* before calling *compute_from_start_to_target()*

---

target: *fastoad.model_base.flight_point.FlightPoint* **= <object object>**

A FlightPoint instance that provides parameter values that should all be reached at the end of compute_from(). Possible parameters depend on the current segment. A parameter can also be set to CONSTANT_VALUE to tell that initial value should be kept during all segment.

isa_offset: float **= 0.0**

The temperature offset for ISA atmosphere model.

CONSTANT_VALUE = 'constant'

Using this value will tell to keep the associated parameter constant.

abstract compute_from_start_to_target(*start*, *target*) → pandas.core.frame.DataFrame

Here should come the implementation for computing flight points between start and target flight points.

Parameters

- **start** –
- **target** – Definition of segment target

Returns a pandas DataFrame where column names match fields of *FlightPoint*

compute_from(*start:* fastoad.model_base.flight_point.FlightPoint) → pandas.core.frame.DataFrame
Computes the flight path segment from provided start point.

Computation ends when target is attained, or if the computation stops getting closer to target. For instance, a climb computation with too low thrust will only return one flight point, that is the provided start point.

Important: When subclasssing, if you need to overload *compute_from()*, you should consider overriding *compute_from_start_to_target()* instead. Therefore, you will take benefit of the preprocessing of start and target flight points that is done in *compute_from()*.

Parameters **start** – the initial flight point, defined for *altitude*, *mass* and speed (*true_airspeed*, *equivalent_airspeed* or *mach*). Can also be defined for *time* and/or *ground_distance*.

Returns a pandas DataFrame where column names match fields of *FlightPoint*

complete_flight_point(*flight_point:* fastoad.model_base.flight_point.FlightPoint)
Computes data for provided flight point.

Assumes that it is already defined for time, altitude, mass, ground distance and speed (TAS, EAS, or Mach).

Parameters **flight_point** – the flight point that will be completed in-place

static complete_flight_point_from(*flight_point:* fastoad.model_base.flight_point.FlightPoint, *source:* fastoad.model_base.flight_point.FlightPoint)
Sets undefined values in *flight_point* using the ones from *source*.

The particular case of speeds is taken into account: if at least one speed parameter is defined, all other speed parameters are considered defined, because they will be deduced when needed.

Parameters

- **flight_point** –
- **source** –

static consume_fuel(*flight_point:* fastoad.model_base.flight_point.FlightPoint, *previous:* fastoad.model_base.flight_point.FlightPoint, *fuel_consumption: Optional[float] = None*, *mass_ratio: Optional[float] = None*)
This method should be used whenever fuel consumption has to be stored.

It ensures that "mass" and "consumed_fuel" fields will be kept consistent.

Mass can be modified using the 'fuel_consumption" argument, or the 'mass_ratio' argument. One of them should be provided.

Parameters

- **flight_point** – the FlightPoint instance where "mass" and "consumed_fuel" fields will get new values
- **previous** – FlightPoint instance that will be the base for the computation
- **fuel_consumption** – consumed fuel, in kg, between 'previous' and 'flight_point'. Positive when fuel is consumed.
- **mass_ratio** – the ratio flight_point.mass/previous.mass

fastoad.models.performances.mission.segments.macro_segments module

Base for macro-segments.

class fastoad.models.performances.mission.segments.macro_segments.**MacroSegmentBase**(*name: str = ''*, *target: fastoad.model_base.flight_point.FlightPoint = <object object>*, *_target: Optional[fastoad.model_base.flight_point.FlightPoint] = None*)

Bases: *fastoad.models.performances.mission.base.FlightSequence*

Base class for macro-segments.

A macro-segment is a sequence of flight segments. Parameters of the macro-segment drive the parameters of aggregated segments.

A field value will be applied to all segments that have the concerned field. The exception is the *target* field, that is applied only on last segment.

This class is expected to be used through *MacroSegmentMeta*. It sets the basic mechanism for aggregating flight segments.

Derived classes are expected to have dataclass fields that match dataclass fields of aggregated segment classes.

target: *fastoad.model_base.flight_point.FlightPoint* = <object object>
Target flight point for end of takeoff

cls_sequence = []
List of segment classes that will compose this macro-segment.

build_sequence()
Instantiates all segments, using dataclass field values of this macro-segment.

Since only target of the last segment is set (using target of this macro-segment), derived classes should overload this method to manage at least targets of intermediate segments.

Note: this method is called each time a dataclass field value is modified.

class fastoad.models.performances.mission.segments.macro_segments.**MacroSegmentMeta**(*cls_name*, *bases*, *attrs*, ***, *cls_sequence=None*)

Bases: abc.ABCMeta

Metaclass for macro-segments.

It should be used with

```
>>> class TakeOffSequence( metaclass=MacroSegmentMeta,
>>>                        cls_sequence=[...],
>>>                      ):
```

It will make so that the created class will have dataclass fields that match dataclass fields of all classes in 'cls_sequence'.

fastoad.models.performances.mission.segments.time_step_base module

Base classes for time-step segments

class fastoad.models.performances.mission.segments.time_step_base.**AbstractTimeStepFlightSegment**(*name: str = '', target: fastoad.mod = <object object>, isa_offset float = 0.0, propulsion: fastoad.mod = <object object>, polar: fastoad.mod = <object object>, polar_modif fastoad.mod = UnchangedPolar(), reference_are float = <object object>, time_step float = 0.2, maximum_CL*

Base class for time step computation flight segments.

This class implements the time computation. For this computation to work, subclasses must implement abstract methods *get_distance_to_target()*, *get_gamma_and_acceleration()* and *compute_propulsion()*.

`compute_next_alpha()` also has to be overloaded if angle of attack should be different of 0.

propulsion: *fastoad.model_base.propulsion.IPropulsion* **= <object object>**
A IPropulsion instance that will be called at each time step.

polar: *fastoad.models.performances.mission.polar.Polar* **= <object object>**
The Polar instance that will provide drag data.

polar_modifier:
fastoad.models.performances.mission.polar_modifier.AbstractPolarModifier **=**
UnchangedPolar()

reference_area: float **= <object object>**
The reference area, in m**2.

time_step: float **= 0.2**
Used time step for computation (actual time step can be lower at some particular times of the flight path).

maximum_CL: float **= None**

altitude_bounds: tuple **= (-500.0, 40000.0)**
Minimum and maximum authorized altitude values. If computed altitude gets beyond these limits, computation will be interrupted and a warning message will be issued in logger.

mach_bounds: tuple **= (-1e-06, 5.0)**
Minimum and maximum authorized mach values. If computed Mach gets beyond these limits, computation will be interrupted and a warning message will be issued in logger.

interrupt_if_getting_further_from_target: bool **= True**
If True, computation will be interrupted if a parameter stops getting closer to target between two iterations (which can mean the provided thrust rate is not adapted).

engine_setting: *fastoad.constants.EngineSetting* **= 2**
The EngineSetting value associated to the segment. Can be used in the propulsion model.

abstract get_distance_to_target(*flight_points: List[*fastoad.model_base.flight_point.FlightPoint*]*, *target:* fastoad.model_base.flight_point.FlightPoint) → float
Computes a "distance" from last flight point to target.

Computed does not need to have a real meaning. The important point is that it must be signed so that algorithm knows on which "side" of the target we are. And of course, it should be 0. if flight point is on target.

Parameters

- **flight_points** – list of all currently computed flight_points
- **target** – segment target (will not contain relative values)

Returns

O. if target is attained, a non-null value otherwise

abstract compute_propulsion(*flight_point:* fastoad.model_base.flight_point.FlightPoint)
Computes propulsion data.

Provided flight point is modified in place.

Generally, this method should end with:

```
self.propulsion.compute_flight_points(flight_point)
```

Parameters **flight_point** –

abstract get_gamma_and_acceleration(*flight_point:* fastoad.model_base.flight_point.FlightPoint) → Tuple[float, float]

Computes slope angle (gamma) and acceleration.

Parameters **flight_point** – parameters after propulsion model has been called (i.e. mass, thrust and drag are available)

Returns slope angle in radians and acceleration in m**2/s

get_next_alpha(*previous_point:* fastoad.model_base.flight_point.FlightPoint, *time_step: float*) → float

Determine the next angle of attack.

Parameters

- **previous_point** – the flight point from which next alpha is computed
- **time_step** – the duration between computed flight point and previous_point

complete_flight_point(*flight_point:* fastoad.model_base.flight_point.FlightPoint)

Computes data for provided flight point.

Assumes that it is already defined for time, altitude, mass, ground distance and speed (TAS, EAS, or Mach).

Parameters **flight_point** – the flight point that will be completed in-place

compute_from_start_to_target(*start:* fastoad.model_base.flight_point.FlightPoint, *target:* fastoad.model_base.flight_point.FlightPoint) → pandas.core.frame.DataFrame

Here should come the implementation for computing flight points between start and target flight points.

Parameters

- **start** –
- **target** – Definition of segment target

Returns a pandas DataFrame where column names match fields of *FlightPoint*

compute_next_flight_point(*flight_points: List[*fastoad.model_base.flight_point.FlightPoint*]*, *time_step: float*) → *fastoad.model_base.flight_point.FlightPoint*

Computes time, altitude, speed, mass and ground distance of next flight point.

Parameters

- **flight_points** – previous flight points
- **time_step** – time step for computing next point

Returns the computed next flight point

property target: *fastoad.model_base.flight_point.FlightPoint*

The base class of the class hierarchy.

When called, it accepts no arguments and returns a new featureless instance that has no instance attributes and cannot be given any.

class fastoad.models.performances.mission.segments.time_step_base.**AbstractManualThrustSegment**(*name: str = '', target: fastoad.model_ = <object object>, isa_offset: float = 0.0, propulsion: fastoad.model_ = <object object>, polar: fastoad.models = <object object>, polar_modifier fastoad.models = UnchangedPolar(), reference_area: float = <object object>, time_step: float = 0.2, maximum_CL:*

AbstractTimeStepFlightSegment, abc.ABC

Base class for computing flight segment where thrust rate is imposed.

Variables **thrust_rate** – used thrust rate. Can be set at instantiation using a keyword argument.

thrust_rate: float = 1.0

compute_propulsion(*flight_point:* fastoad.model_base.flight_point.FlightPoint)
Computes propulsion data.

Provided flight point is modified in place.

Generally, this method should end with:

```
self.propulsion.compute_flight_points(flight_point)
```

Parameters **flight_point** –

property target: *fastoad.model_base.flight_point.FlightPoint*
The base class of the class hierarchy.

When called, it accepts no arguments and returns a new featureless instance that has no instance attributes and cannot be given any.

class fastoad.models.performances.mission.segments.time_step_base.**AbstractRegulatedThrustSegment**(*name: str = '', target: fastoad.mo = <object object>, isa_offse float = 0.0, propulsion: fastoad.mo = <object object>, polar: fastoad.mo = <object object>, polar_moa fastoad.mo = UnchangedPolar(), reference_ar float = <object object>, time_ste float = 60.0, maximum_CL*

AbstractTimeStepFlightSegment, abc.ABC

Base class for computing flight segment where thrust rate is adjusted on drag.

time_step: float = 60.0
Used time step for computation (actual time step can be lower at some particular times of the flight path).

compute_propulsion(*flight_point:* fastoad.model_base.flight_point.FlightPoint)
Computes propulsion data.

Provided flight point is modified in place.

Generally, this method should end with:

```
self.propulsion.compute_flight_points(flight_point)
```

Parameters **flight_point** –

get_gamma_and_acceleration(*flight_point:* fastoad.model_base.flight_point.FlightPoint) → Tuple[float, float]
Computes slope angle (gamma) and acceleration.

Parameters **flight_point** – parameters after propulsion model has been called (i.e. mass, thrust and drag are available)

Returns slope angle in radians and acceleration in m**2/s

property target: *fastoad.model_base.flight_point.FlightPoint*
The base class of the class hierarchy.

When called, it accepts no arguments and returns a new featureless instance that has no instance attributes and cannot be given any.

class fastoad.models.performances.mission.segments.time_step_base.**AbstractFixedDurationSegment**(*name: str = '', target: fastoad.model = <object object>, isa_offset: float = 0.0, propulsion: fastoad.model = <object object>, polar: fastoad.model = <object object>, polar_modifier: fastoad.model = UnchangedPolar(), reference_area: float = <object object>, time_step: float = 60.0, maximum_CL:

AbstractTimeStepFlightSegment, abc.ABC

Base class for computing a fixed-duration segment.

time_step: float = 60.0
Used time step for computation (actual time step can be lower at some particular times of the flight path).

get_distance_to_target(*flight_points: List[*fastoad.model_base.flight_point.FlightPoint*]*, *target:* fastoad.model_base.flight_point.FlightPoint) → float
Computes a "distance" from last flight point to target.

Computed does not need to have a real meaning. The important point is that it must be signed so that algorithm knows on which "side" of the target we are. And of course, it should be 0. if flight point is on target.

Parameters

- **flight_points** – list of all currently computed flight_points
- **target** – segment target (will not contain relative values)

Returns

0. if target is attained, a non-null value otherwise

property target: *fastoad.model_base.flight_point.FlightPoint*
The base class of the class hierarchy.

When called, it accepts no arguments and returns a new featureless instance that has no instance attributes and cannot be given any.

class fastoad.models.performances.mission.segments.time_step_base.**AbstractTakeOffSegment**(*name: str = '', target: fastoad.model_base.fl = <object object>, isa_offset: float = 0.0, propulsion: fastoad.model_base.p = <object object>, polar: fastoad.models.perfor = <object object>, polar_modifier: fastoad.models.perfor = UnchangedPolar(), reference_area: float = <object object>, time_step: float = 0.1, maximum_CL:*

AbstractManualThrustSegment, abc.ABC

Class for computing takeoff segment.

time_step: float = 0.1
Used time step for computation (actual time step can be lower at some particular times of the flight path).

compute_from_start_to_target(*start:* fastoad.model_base.flight_point.FlightPoint, *target:* fastoad.model_base.flight_point.FlightPoint) → pandas.core.frame.DataFrame
Here should come the implementation for computing flight points between start and target flight points.

Parameters

- **start** –
- **target** – Definition of segment target

Returns a pandas DataFrame where column names match fields of *FlightPoint*

property target: *fastoad.model_base.flight_point.FlightPoint*
The base class of the class hierarchy.

When called, it accepts no arguments and returns a new featureless instance that has no instance attributes and cannot be given any.

class fastoad.models.performances.mission.segments.time_step_base.**AbstractGroundSegment**(*name: str = '', target: fastoad.model_base.flig = <object object>, isa_offset: float = 0.0, propulsion: fastoad.model_base.pr = <object object>, polar: fastoad.models.perform = <object object>, polar_modifier: fastoad.models.perform = UnchangedPolar(), reference_area: float = <object object>, time_step: float = 0.1, maximum_CL:

AbstractTakeOffSegment, abc.ABC

Class for computing accelerated segments on the ground with wheel friction.

wheels_friction: float = 0.03

get_gamma_and_acceleration(*flight_point:* fastoad.model_base.flight_point.FlightPoint)
For ground segment, gamma is assumed always 0 and wheel friction (with or without brake) is added to drag

complete_flight_point(*flight_point:* fastoad.model_base.flight_point.FlightPoint)
Computes data for provided flight point using AoA and apply polar modification if any

Parameters **flight_point** – the flight point that will be completed in-place

property target: *fastoad.model_base.flight_point.FlightPoint*
The base class of the class hierarchy.

When called, it accepts no arguments and returns a new featureless instance that has no instance attributes and cannot be given any.

class fastoad.models.performances.mission.segments.time_step_base.**FlightSegment**(**args*, ***kwargs*)

Bases: *fastoad.models.performances.mission.segments.time_step_base.AbstractTimeStepFlightSegment*, abc.ABC

Base class for time step computation flight segments.

This class implements the time computation. For this computation to work, subclasses must implement abstract methods `get_get_distance_to_target()`, `get_gamma_and_acceleration()` and `compute_propulsion()`.

property target: *fastoad.model_base.flight_point.FlightPoint*
The base class of the class hierarchy.

When called, it accepts no arguments and returns a new featureless instance that has no instance attributes and cannot be given any.

Module contents

Submodules

fastoad.models.performances.mission.base module

Base classes for mission computation.

class fastoad.models.performances.mission.base.**IFlightPart**(*name: str = ''*)
Bases: abc.ABC, *fastoad.model_base.datacls.BaseDataClass*

Base class for all flight parts.

name: str = ''

target: *fastoad.model_base.flight_point.FlightPoint*

abstract compute_from(*start:* fastoad.model_base.flight_point.FlightPoint) → pandas.core.frame.DataFrame
Computes a flight sequence from provided start point.

Parameters **start** – the initial flight point, defined for *altitude*, *mass* and speed (*true_airspeed*, *equivalent_airspeed* or *mach*). Can also be defined for *time* and/or *ground_distance*.

Returns a pandas DataFrame where column names match fields of *FlightPoint*

class fastoad.models.performances.mission.base.**FlightSequence**(*name: str = ''*, *_target: Optional[fastoad.model_base.flight_point.FlightPoint] = None*)

Bases: *fastoad.models.performances.mission.base.IFlightPart*

Defines and computes a flight sequence.

Use .extend() method to add a list of parts in the sequence.

consumed_mass_before_input_weight: float = 0.0

Consumed mass between sequence start and target mass, if any defined

part_flight_points: List[pandas.core.frame.DataFrame]

compute_from(*start:* fastoad.model_base.flight_point.FlightPoint) → pandas.core.frame.DataFrame

Computes a flight sequence from provided start point.

Parameters **start** – the initial flight point, defined for *altitude*, *mass* and speed (*true_airspeed*, *equivalent_airspeed* or *mach*). Can also be defined for *time* and/or *ground_distance*.

Returns a pandas DataFrame where column names match fields of *FlightPoint*

property target: Optional[*fastoad.model_base.flight_point.FlightPoint*]

Target of the last element of current sequence.

append(*flight_part:* fastoad.models.performances.mission.base.IFlightPart)

Append flight part to the end of the sequence.

clear()

Remove all parts from flight sequence.

extend(*seq*)

Extend flight sequence by appending elements from the iterable.

index(**args*, ***kwargs*)

Return first index of value (see list.index()).

class fastoad.models.performances.mission.base.**RegisterElement**(*keyword=''*)

Bases: object

Base class for decorators that can associate a class with a keyword.

When subclassing, the argument 'base_class' allow to specify a class that should be a parent of all registered classes. A specific check will be done at register time.

```
>>> class RegisterFeature(RegisterElement, base_class=AbstractFeature)
>>>     ...
```

Then the newly created class may be used as decorator like:

```
>>> @RegisterFeature("identifier_foo")
>>> class FooFeature(AbstractFeature):
>>>     ...
```

Then the registered class can be obtained by:

```
>>> my_class = RegisterFeature.get_class("identifier_foo")
```

classmethod get_class(*keyword*) → Optional[type]
Provides the element implementation for provided name.

Parameters **keyword** –

Returns the element implementation

Raises ***FastUnknownMissionElementError*** – if element has not been declared.

classmethod get_classes() → Dict[str, type]

Returns dict that associates keywords to their registered class.

fastoad.models.performances.mission.exceptions module

Exceptions for mission package.

exception fastoad.models.performances.mission.exceptions.**FastFlightSegmentUnexpectedKeywordArgument**(*bad_*
Bases: *fastoad.exceptions.FastUnexpectedKeywordArgument*

Raised when a segment is instantiated with an incorrect keyword argument.

exception fastoad.models.performances.mission.exceptions.**FastFlightPointUnexpectedKeywordArgument**(*bad_ke*
Bases: *fastoad.exceptions.FastUnexpectedKeywordArgument*

Raised when a FlightPoint is instantiated with an incorrect keyword argument.

exception
fastoad.models.performances.mission.exceptions.**FastFlightSegmentIncompleteFlightPoint**
Bases: *fastoad.exceptions.FastError*

Raised when a segment computation encounters a FlightPoint instance without needed parameters.

exception fastoad.models.performances.mission.exceptions.**FastUnknownMissionElementError**(*element_type: str*)

Bases: *fastoad.exceptions.FastError*

Raised when an undeclared element type is requested.

fastoad.models.performances.mission.mission module

Definition of aircraft mission.

class fastoad.models.performances.mission.mission.**Mission**(*name: str = ''*, *_target: Optional[*fastoad.model_base.flight_point.FlightPoint*] = None*, *target_fuel_consumption: Optional[float] = None*, *reserve_ratio: Optional[float] = 0.0*, *reserve_base_route_name: Optional[str] = None*, *fuel_accuracy: float = 10.0*)
Bases: *fastoad.models.performances.mission.base.FlightSequence*

Computes a whole mission.

Allows to define a target fuel consumption for the whole mission.

target_fuel_consumption: Optional[float] = None
If not None, the mission will adjust the first

reserve_ratio: Optional[float] = 0.0

reserve_base_route_name: Optional[str] = None

fuel_accuracy: float = 10.0
Accuracy on actual consumed fuel for the solver. In kg

property consumed_fuel: Optional[float]
Total consumed fuel for the whole mission (after launching *compute_from()*)

property first_route: *fastoad.models.performances.mission.routes.RangedRoute*
First route in the mission.

compute_from(*start:* fastoad.model_base.flight_point.FlightPoint) → pandas.core.frame.DataFrame
Computes a flight sequence from provided start point.

Parameters **start** – the initial flight point, defined for *altitude*, *mass* and speed (*true_airspeed*, *equivalent_airspeed* or *mach*). Can also be defined for *time* and/or *ground_distance*.

Returns a pandas DataFrame where column names match fields of *FlightPoint*

get_reserve_fuel()

Returns the fuel quantity for reserve, obtained after mission computation.

fastoad.models.performances.mission.polar module

Aerodynamic polar data.

class fastoad.models.performances.mission.polar.**Polar**(*cl: numpy.ndarray*, *cd: numpy.ndarray*, *alpha: Optional[numpy.ndarray] = None*)

Bases: object

Class for managing aerodynamic polar data.

Links drag coefficient (CD) to lift coefficient (CL). It is defined by two vectors with CL and CD values. If a vector of angle of attack (alpha) is given, it links alpha and CL

Once defined, for any CL value, CD can be obtained using *cd()*. For any alpha given, CL is obtained using :meth:'cl'.

Parameters

- **cl** – a N-elements array with CL values
- **cd** – a N-elements array with CD values that match CL
- **alpha** – a N-elements array with angle of attack corresponding to CL values

property definition_cl
The vector that has been used for defining lift coefficient.

property definition_cd
The vector that has been used for defining drag coefficient.

property definition_alpha
The vector that has been used for defining AoA.

property optimal_cl
The CL value that provides larger lift/drag ratio.

cd(*cl=None*)
Computes drag coefficient (CD) by interpolation in definition data.

Parameters **cl** – lift coefficient (CL) values. If not provided, the CL definition vector will be used (i.e. CD definition vector will be returned)

Returns CD values for each provide CL values

cl(*alpha*)
The lift coefficient corresponding to alpha (rad)

Parameters **alpha** – the angle of attack at which CL is evaluated

Returns CL value for each alpha.

fastoad.models.performances.mission.polar_modifier module

Aerodynamics polar modifier.

class fastoad.models.performances.mission.polar_modifier.**AbstractPolarModifier**
Bases: abc.ABC

Base class to implement a change to the polar during the mission computation

abstract modify_polar(*polar:* fastoad.models.performances.mission.polar.Polar, *flight_point:* fastoad.model_base.flight_point.FlightPoint) → *fastoad.models.performances.mission.polar.Polar*

Parameters

- **polar** – an instance of Polar
- **flight_point** – an intance of FlightPoint containg only floats

Returns the modified polar for the flight point

class fastoad.models.performances.mission.polar_modifier.**RegisterPolarModifier**(*keyword=''*)
Bases: *fastoad.models.performances.mission.base.RegisterElement*

Decorator for registering AbstractPolarModifier classes.

```
>>> @RegisterPolarModifier("polar_modifier_foo")
>>> class FooPolarModifier(IFlightPart):
>>>     ...
```

Then the registered class can be obtained by:

```
>>> my_class = RegisterPolarModifier.get_class("polar_modifier_foo")
```

class fastoad.models.performances.mission.polar_modifier.**UnchangedPolar**
Bases: *fastoad.models.performances.mission.polar_modifier.AbstractPolarModifier*

Default polar modifier returning the polar without changes

modify_polar(*polar:* fastoad.models.performances.mission.polar.Polar, *flight_point:* fastoad.model_base.flight_point.FlightPoint) → *fastoad.models.performances.mission.polar.Polar*

Parameters

- **polar** – a polar instance
- **flight_point** – a FlightPoint instance

Returns the polar instance

class fastoad.models.performances.mission.polar_modifier.**GroundEffectRaymer**(*span: float*, *landing_gear_height: float*, *induced_drag_coefficient: float*, *k_winglet: float*, *k_cd: float*, *ground_altitude: float = 0.0*)

Bases: *fastoad.models.performances.mission.polar_modifier.AbstractPolarModifier*

Evaluates the drag in ground effect, using Raymer's model: 'Aircraft Design A conceptual approach', D. Raymer p304

span: float
Wingspan

landing_gear_height: float
Main landing gear height

induced_drag_coefficient: float
Induced drag coefficient, multiplies CL**2 to obtain the induced drag

k_winglet: float
Winglet effect tuning coefficient

k_cd: float
Total drag tuning coefficient

ground_altitude: float = 0.0
Altitude of ground w.r.t. sea level

modify_polar(*polar:* fastoad.models.performances.mission.polar.Polar, *flight_point:* fastoad.model_base.flight_point.FlightPoint) → *fastoad.models.performances.mission.polar.Polar*

Compute the ground effect based on altitude from ground and return an updated polar

Parameters

- **polar** – a Polar instance used as basis to apply ground effect
- **flight_point** – a flight point containing the flight conditions

for calculation of ground effect :return: a copy of polar with ground effect

fastoad.models.performances.mission.routes module

Classes for computation of routes (i.e. assemblies of climb, cruise and descent phases).

class fastoad.models.performances.mission.routes.**RangedRoute**(*name: str = ''*, *_target: Optional[fastoad.model_base.flight_point.FlightPoint] = None*, *climb_phases: List[fastoad.models.performances.mission.base.FlightS = <object object>*, *cruise_segment: fastoad.models.performances.mission.segments.registered. = <object object>*, *descent_phases: List[fastoad.models.performances.mission.base.FlightS = <object object>*, *flight_distance: float = <object object>*, *distance_accuracy: float = 500.0*, *solve_distance: bool = True*)

Bases: *fastoad.models.performances.mission.base.FlightSequence*

Computes a route so that it covers the specified ground distance.

The computed route will be made of:

- any number of climb phases
- one cruise segment
- any number of descent phases.

climb_phases: List[*fastoad.models.performances.mission.base.FlightSequence*] = <object object>
Any number of flight phases that will occur before cruise.

cruise_segment: *fastoad.models.performances.mission.segments.registered.cruise.CruiseSegment* = <object object>
The cruise phase.

descent_phases: List[*fastoad.models.performances.mission.base.FlightSequence*] = <object object>
Any number of flight phases that will occur after cruise.

flight_distance: float = <object object>
Target ground distance for whole route

distance_accuracy: float = 500.0
Accuracy on actual total ground distance for the solver. In meters

solve_distance: bool = True
If True, cruise distance will be adjusted to match *flight_distance*

property cruise_distance
Ground distance to be covered during cruise, as set in target of *cruise_segment*.

property cruise_speed: Optional[Tuple[str, float]]
Type (among *true_airspeed*, *equivalent_airspeed* and *mach*) and value of cruise speed.

compute_from(*start:* fastoad.model_base.flight_point.FlightPoint) → pandas.core.frame.DataFrame
Computes a flight sequence from provided start point.

Parameters **start** – the initial flight point, defined for *altitude*, *mass* and speed (*true_airspeed*, *equivalent_airspeed* or *mach*). Can also be defined for *time* and/or *ground_distance*.

Returns a pandas DataFrame where column names match fields of *FlightPoint*

fastoad.models.performances.mission.util module

Utilities for mission computation.

fastoad.models.performances.mission.util.**get_closest_flight_level**(*altitude*, *base_level=0*, *level_step=10*, *up_direction=True*)

Computes the altitude (in meters) of a flight level close to provided altitude.

Flight levels are multiples of 100 feet.

see examples below:

```
>>> # Getting the IFR flight level immediately above
>>> get_closest_flight_level(4400. * foot)
5000.0
>>> # Getting the IFR flight level immediately below
>>> get_closest_flight_level(4400. * foot, up_direction=False)
4000.0
>>> # Getting the next even IFR flight level
>>> get_closest_flight_level(4400. * foot, level_step = 20)
6000.0
>>> # Getting the next odd IFR flight level
>>> get_closest_flight_level(3100. * foot, base_level=10, level_step = 20)
5000.0
```

Parameters

- **altitude** – in meters
- **base_level** – base flight level for computed steps
- **level_step** – number of flight level per step
- **up_direction** – True if next flight level is upper. False if lower

Returns the altitude in meters of the asked flight level.

Module contents

Performance module for mission simulation.

Module contents

Package for performance modules.

Module contents

This package contains the OAD models of FAST-OAD.

It has to be declared as FAST-OAD plugin.

These models are based on following references:

fastoad.module_management package

Subpackages

Submodules

fastoad.module_management.constants module

The place for module-level constants.

class fastoad.module_management.constants.**ModelDomain**(*value=<no_arg>*, *names=None*, *module=None*, *type=None*, *start=1*, *boundary=None*)

Bases: aenum.Enum

Enumeration of model domains.

GEOMETRY = 'Geometry'

AERODYNAMICS = 'Aerodynamics'

HANDLING_QUALITIES = 'Handling Qualities'

WEIGHT = 'Weight'

PERFORMANCE = 'Performance'

PROPULSION = 'Propulsion'

OTHER = 'Other'

UNSPECIFIED = 'Unspecified'

fastoad.module_management.exceptions module

Exceptions for module_management package.

exception fastoad.module_management.exceptions.**FastBundleLoaderDuplicateFactoryError**(*factory_name: str*)

Bases: *fastoad.exceptions.FastError*

Raised when trying to register a factory with an already used name.

Parameters **factory_name** –

exception fastoad.module_management.exceptions.**FastBundleLoaderUnknownFactoryNameError**(*factory_name: str*)

Bases: *fastoad.exceptions.FastError*

Raised when trying to instantiate a component from an unknown factory.

Parameters **factory_name** –

exception fastoad.module_management.exceptions.**FastBadSystemOptionError**(*identifier*, *option_names*)

Bases: *fastoad.exceptions.FastError*

Raised when some option name is not conform to OpenMDAO system definition.

Parameters

- **identifier** – system identifier
- **option_names** – incorrect option names

exception fastoad.module_management.exceptions.**FastIncompatibleServiceClassError**(*registered_class: type*, *service_id: str*, *base_class: type*)

Bases: *fastoad.exceptions.FastError*

Raised when trying to register as service a class that does not implement the specified interface.

Parameters

- **registered_class** –
- **service_id** –
- **base_class** – the unmatched interface

exception fastoad.module_management.exceptions.**FastNoSubmodelFoundError**(*service_id: str*)

Bases: *fastoad.exceptions.FastError*

Raised when a submodel is required, but none has been declared.

Parameters **service_id** –

exception fastoad.module_management.exceptions.**FastTooManySubmodelsError**(*service_id: str*, *candidates: Sequence[str]*)

Bases: *fastoad.exceptions.FastError*

Raised when several candidates are declared for a required submodel, but none has been selected.

Parameters

- **service_id** –
- **candidates** –

exception fastoad.module_management.exceptions.**FastUnknownSubmodelError**(*service_id: str*, *submodel_id: str*, *submodel_ids: List[str]*)

Bases: *fastoad.exceptions.FastError*

Raised when a submodel identifier is unknown for given required service.

Parameters

- **service_id** –
- **submodel_id** –
- **submodel_ids** –

exception fastoad.module_management.exceptions.**FastNoDistPluginError**

Bases: *fastoad.exceptions.FastError*

Raised when no installed package with FAST-OAD plugin is available.

exception fastoad.module_management.exceptions.**FastUnknownDistPluginError**(*dist_name*)

Bases: *fastoad.exceptions.FastError*

Raised when a distribution name is not found among distribution with FAST-OAD plugins.

exception fastoad.module_management.exceptions.**FastSeveralDistPluginsError**

Bases: *fastoad.exceptions.FastError*

Raised when no distribution name has been specified but several distribution with FAST-OAD plugins are available.

exception fastoad.module_management.exceptions.**FastNoAvailableConfigurationFileError**

Bases: *fastoad.exceptions.FastError*

Raised when a configuration file is asked, but none is available in plugins.

exception fastoad.module_management.exceptions.**FastUnknownConfigurationFileError**(*configuration_file*, *dist_name*)

Bases: *fastoad.exceptions.FastError*

Raised when a configuration file is not found for named distribution.

exception fastoad.module_management.exceptions.**FastSeveralConfigurationFilesError**(*dist_name*)

Bases: *fastoad.exceptions.FastError*

Raised when no configuration file has been specified but several configuration files are provided with the distribution.

exception fastoad.module_management.exceptions.**FastNoAvailableSourceDataFileError**

Bases: *fastoad.exceptions.FastError*

Raised when a source data file is requested, but none is available in plugins.

exception fastoad.module_management.exceptions.**FastUnknownSourceDataFileError**(*source_data_file*, *dist_name*)

Bases: *fastoad.exceptions.FastError*

Raised when a source data file is not found for named distribution.

exception fastoad.module_management.exceptions.**FastSeveralSourceDataFilesError**(*dist_name*)
Bases: *fastoad.exceptions.FastError*

Raised when no source data file has been specified but several source data files are provided with the distribution.

fastoad.module_management.service_registry module

Module for registering services.

class fastoad.module_management.service_registry.**RegisterService**(*service_id: str*, *provider_id: str*, *desc=None*)
Bases: object

Decorator class that allows to register services and associated providers.

This class also provides class methods for getting service providers and information about them.

The basic registering of a class is done with:

```
@RegisterService("my.service.id", "id.of.the.provider")
class MyService:
    ...
```

A child of this class may define a particular base class or interface that should be parent to all registered service providers.

The definition of the base class is done when subclassing, e.g.:

```
class RegisterSomeService( RegisterService, base_class=ISomeService):
    "Allows to register classes that implement interface ISomeService."
```

Parameters

- **service_id** – the identifier of the provided service
- **provider_id** – the identifier of the service provider to register
- **desc** – description of the service provider. If not provided, the docstring of decorated class will be used.

get_properties(*service_class: Type[fastoad.module_management.service_registry.T]*) → dict
Override this method to modify the properties that will be associated to the registered service provider.

This basic version ensures the associated description property is the one provided when instantiating this decorator class, if it is provided. Otherwise, it will be the docstring of the decorated class.

Parameters **service_class** – the class that will be registered as service provider

Returns the dictionary of properties that will be associated to the registered service provider

classmethod explore_folder(*folder_path: str*)
Explores provided folder and looks for service providers to register.

Parameters **folder_path** –

classmethod get_provider_ids(*service_id: str*) → List[str]

Parameters **service_id** –

Returns the list of identifiers of providers of the service.

classmethod get_provider(*service_provider_id: str*, *options: Optional[dict] = None*) → Any
Instantiates the desired service provider.

Parameters

- **service_provider_id** – identifier of a registered service provider
- **options** – options that should be associated to the created instance

Returns the created instance

classmethod get_provider_description(*instance_or_id: Union[str, fastoad.module_management.service_registry.T]*) → str

Parameters **instance_or_id** – an identifier or an instance of a registered service provider

Returns the description associated to given instance or identifier

classmethod get_provider_domain(*instance_or_id: Union[str, openmdao.core.system.System]*) → *fastoad.module_management.constants.ModelDomain*

Parameters **instance_or_id** – an identifier or an instance of a registered service provider

Returns the model domain associated to given instance or identifier

class fastoad.module_management.service_registry.**RegisterSpecializedService**(*provider_id: str*, *desc=None*, *domain: Optional[*fastoad.module_management *= None*, *options: Optional[dict] = None*)

Bases: *fastoad.module_management.service_registry.RegisterService*

Base class for decorator classes that allow to register a particular service.

The service may be associated to a base class (or interface). The registered class must inherit from this base class.

Unlike *RegisterService*, this class has to be subclassed, because the service identifier is defined when subclassing.

The definition of the base class is done by subclassing, e.g.:

```
class RegisterSomeService( RegisterSpecializedService,
                           base_class=ISomeService,
                           service_id="my.particularservice"):
    "Allows to register classes that implement interface ISomeService."
```

Then basic registering of a class is done with:

```
@RegisterSomeService("my.particularservice.provider")
class ParticularService(ISomeService):
    ...
```

Parameters

- **provider_id** – the identifier of the service provider to register
- **desc** – description of the service. If not provided, the docstring will be used.

- **domain** – a category for the registered service provider
- **options** – a dictionary of options that can be associated to the service provider

service_id: *str*

get_properties(*service_class: Type[fastoad.module_management.service_registry.T]*) → dict

Override this method to modify the properties that will be associated to the registered service provider.

This basic version ensures the associated description property is the one provided when instantiating this decorator class, if it is provided. Otherwise, it will be the docstring of the decorated class.

Parameters **service_class** – the class that will be registered as service provider

Returns the dictionary of properties that will be associated to the registered service provider

classmethod get_provider_ids() → List[str]

Returns the list of identifiers of providers of the service.

class fastoad.module_management.service_registry.**RegisterPropulsion**(*provider_id: str*, *desc=None*, *domain: Optional[*fastoad.module_management.constants. *= None*, *options: Optional[dict] = None*)

Bases: fastoad.module_management.service_registry._RegisterSpecializedOpenMDAOService

Decorator class for registering an OpenMDAO wrapper of a propulsion-dedicated model.

Parameters

- **provider_id** – the identifier of the service provider to register
- **desc** – description of the service. If not provided, the docstring will be used.
- **domain** – a category for the registered service provider
- **options** – a dictionary of options that can be associated to the service provider

service_id: *str* = 'fastoad.wrapper.propulsion'

class fastoad.module_management.service_registry.**RegisterOpenMDAOSystem**(*provider_id: str*, *desc=None*, *domain: Optional[*fastoad.module_management.cons *= None*, *options: Optional[dict] = None*)

Bases: fastoad.module_management.service_registry._RegisterSpecializedOpenMDAOService

Decorator class for registering an OpenMDAO system for use in FAST-OAD configuration.

If a variable_descriptions.txt file is in the same folder as the class module, its content is loaded (once, even if several classes are registered at the same level).

Parameters

- **provider_id** – the identifier of the service provider to register
- **desc** – description of the service. If not provided, the docstring will be used.
- **domain** – a category for the registered service provider
- **options** – a dictionary of options that can be associated to the service provider

service_id: *str* = 'fast.openmdao.system'

class fastoad.module_management.service_registry.**RegisterSubmodel**(*service_id: str*, *provider_id: str*, *desc=None*, *options: Optional[dict] = None*)

Bases: fastoad.module_management.service_registry._RegisterOpenMDAOService

Decorator class that allows to submodels.

Submodels are OpenMDAO systems that fulfill a requirement (service id) in a FAST-OAD module.

active_models defines the submodel to be used for any service identifier it has as key. See *get_submodel()* for more details.

The registering of a class is done with:

```
@RegisterSubmodel("my.service", "id.of.the.provider")
class MyService:
    ...
```

Then the submodel can be instantiated and used with:

```
submodel_instance = RegisterSubmodel.get_submodel("my.service")
some_model.add_subsystem("my_submodel", submodel_instance, promotes=["*"])
...
```

Parameters

- **service_id** – the identifier of the provided service
- **provider_id** – the identifier of the service provider to register
- **desc** – description of the service. If not provided, the docstring will be used.
- **options** – a dictionary of options that will be defaults when instantiating the system

active_models: Dict[str, Optional[str]] = {}

Dictionary (key = service id, value=provider id) that defines submodels to be used for associated services.

classmethod get_submodel(*service_id: str*, *options: Optional[dict] = None*)

Provides a submodel for the given service identifier.

If *active_models* has *service_id* as key:

- if the associated value is a non-empty string, a submodel will be instantiated with this string as submodel identifier. If the submodel identifier matches nothing, an error will be raised.
- if the associated value is None, an empty submodel (om.Group()) will be instantiated. You may see it as a way to deactivate a particular submodel.

If *active_models* has *service_id* has NOT as key:

- if no submodel is declared for this *service_id*, an error will be raised.
- if one and only one submodel is declared for this *service_id*, it will be instantiated.
- if several submodels are declared for this *service_id*, an error will be raised.

If an actual (not empty) submodel is defined, provided options will be used.

Parameters

- **service_id** –

- **options** –

Returns the instantiated submodel

classmethod cancel_submodel_deactivations()
Reactivates all submodels that have been deactivated.

Module contents

Management of modules using Pelix/iPOPO

fastoad.openmdao package

Subpackages

fastoad.openmdao.variables package

Submodules

fastoad.openmdao.variables.variable module

Class for managing an OpenMDAO variable.

class fastoad.openmdao.variables.variable.**Variable**(*name*, ***kwargs*)
Bases: Hashable

A class for storing data of OpenMDAO variables.

Instantiation is expected to be done through keyword arguments only.

Beside the mandatory parameter 'name, kwargs is expected to have keys 'value', 'units' and 'desc', that are accessible respectively through properties *name()*, *value()*, *units()* and *description()*.

Other keys are possible. They match the definition of OpenMDAO's method Component.add_output() described here.

These keys can be listed with class method *get_openmdao_keys()*. **Any other key in kwargs will be silently ignored.**

Special behaviour: *description()* will return the content of kwargs['desc'] unless these 2 conditions are met:

- kwargs['desc'] is None or 'desc' key is missing
- a description exists in FAST-OAD internal data for the variable name

Then, the internal description will be returned by *description()*

Parameters **kwargs** – the attributes of the variable, as keyword arguments

name
Name of the variable

metadata: Dict
Dictionary for metadata of the variable

classmethod read_variable_descriptions(*file_parent: str*, *update_existing: bool = True*)
Reads variable descriptions in indicated folder or package, if it contains some.

The file variable_descriptions.txt is looked for. Nothing is done if it is not found (no error raised also).

Each line of the file should be formatted like:

```
my:variable||The description of my:variable, as long as needed, but on one␣
↪line.
```

Parameters

- **file_parent** – the folder path or the package name that should contain the file
- **update_existing** – if True, previous descriptions will be updated. if False, previous descriptions will be erased.

classmethod update_variable_descriptions(*variable_descriptions: Union[Mapping[str, str], Iterable[Tuple[str, str]]]*)

Updates description of variables.

Parameters **variable_descriptions** – dict-like object with variable names as keys and descriptions as values

classmethod get_openmdao_keys()

Returns the keys that are used in OpenMDAO variables

property value

value of the variable

property val

value of the variable (alias of property "value")

property units

units associated to value (or None if not found)

property description

description of the variable (or None if not found)

property desc

description of the variable (or None if not found) (alias of property "description")

property is_input

I/O status of the variable.

- True if variable is a problem input
- False if it is an output
- None if information not found

get_openmdao_kwargs(*keys: Optional[Iterable] = None*) → dict

Provides a dict usable as keyword args by OpenMDAO add_input()/add_output().

The dict keys will be the ones provided, or a default set if no keys are provided.

Parameters **keys** –

Returns the kwargs dict

fastoad.openmdao.variables.variable_list module

Class for managing a list of OpenMDAO variables.

class fastoad.openmdao.variables.variable_list.**VariableList**(*iterable=()*, */*)

Bases: list

Class for storing OpenMDAO variables.

A list of *Variable* instances, but items can also be accessed through variable names. It also has utilities to be converted from/to some other data structures (python dict, OpenMDAO IndepVarComp, pandas DataFrame)

See documentation of *Variable* to see how to manipulate each element.

There are several ways for adding variables:

```
# Assuming these Python variables are ready...
var_1 = Variable('var/1', value=0.)
metadata_2 = {'value': 1., 'units': 'm'}

# ... a VariableList instance can be populated like this:
vars_A = VariableList()
vars_A.append(var_1)              # Adds directly a Variable instance
vars_A['var/2'] = metadata_2      # Adds the variable with given name and given␣
↪metadata
```

Note: Adding a Variable instance with a name that is already in the VariableList instance will replace the previous Variable instance instead of adding a new one.

```
# It is also possible to instantiate a VariableList instance from another␣
↪VariableList
# instance or a simple list of Variable instances
vars_B = VariableList(vars_A)
vars_C = VariableList([var_1])

# An existing VariableList instance can also receive the content of another␣
↪VariableList
# instance.
vars_C.update(vars_A)             # variables in vars_A will overwrite variables␣
↪with same
                                  # name in vars_C
```

After that, following equalities are True:

```
print( var_1 in vars_A )
print( 'var/1' in vars_A.names() )
print( 'var/2' in vars_A.names() )
```

names() → List[str]

Returns names of variables

metadata_keys() → List[str]

Returns the metadata keys that are common to all variables in the list

append(*var:* fastoad.openmdao.variables.variable.Variable) → None
Append var to the end of the list, unless its name is already used. In that case, var will replace the previous Variable instance with the same name.

update(*other_var_list: list*, *add_variables: bool = True*)
Uses variables in other_var_list to update the current VariableList instance.

For each Variable instance in other_var_list:

- if a Variable instance with same name exists, it is replaced by the one in other_var_list (special case: if one in other_var_list has an empty description, the original description is kept)
- if not, Variable instance from other_var_list will be added only if add_variables==True

Parameters

- **other_var_list** – source for new Variable data
- **add_variables** – if True, unknown variables are also added

to_ivc() → openmdao.core.indepvarcomp.IndepVarComp

Returns an OpenMDAO IndepVarComp instance with all variables from current list

to_dataframe() → pandas.core.frame.DataFrame
Creates a DataFrame instance from a VariableList instance.

Column names are "name" + the keys returned by `Variable.get_openmdao_keys()`. Values in Series "value" are floats or lists (numpy arrays are converted).

Returns a pandas DataFrame instance with all variables from current list

classmethod from_dict(*var_dict: Union[Mapping[str, dict], Iterable[Tuple[str, dict]]]*) → *fastoad.openmdao.variables.variable_list.VariableList*
Creates a VariableList instance from a dict-like object.

Parameters **var_dict** –

Returns a VariableList instance

classmethod from_ivc(*ivc: openmdao.core.indepvarcomp.IndepVarComp*) → *fastoad.openmdao.variables.variable_list.VariableList*
Creates a VariableList instance from an OpenMDAO IndepVarComp instance

Parameters **ivc** – an IndepVarComp instance

Returns a VariableList instance

classmethod from_dataframe(*df: pandas.core.frame.DataFrame*) → *fastoad.openmdao.variables.variable_list.VariableList*
Creates a VariableList instance from a pandas DataFrame instance.

The DataFrame instance is expected to have column names "name" + some keys among the ones given by `Variable.get_openmdao_keys()`.

Parameters **df** – a DataFrame instance

Returns a VariableList instance

classmethod from_problem(*problem: openmdao.core.problem.Problem*, *use_initial_values: bool = False*, *get_promoted_names: bool = True*, *promoted_only: bool = True*, *io_status: str = 'all'*) → *fastoad.openmdao.variables.variable_list.VariableList*

Creates a VariableList instance containing inputs and outputs of an OpenMDAO Problem.

The inputs (is_input=True) correspond to the variables of IndepVarComp components and all the unconnected input variables.

Note: Variables from _auto_ivc are ignored.

Parameters

- **problem** – OpenMDAO Problem instance to inspect
- **use_initial_values** – if True, or if problem has not been run, returned instance will contain values before computation
- **get_promoted_names** – if True, promoted names will be returned instead of absolute ones (if no promotion, absolute name will be returned)
- **promoted_only** – if True, only promoted variable names will be returned
- **io_status** – to choose with type of variable we return ("all", "inputs, "outputs")

Returns VariableList instance

classmethod from_unconnected_inputs(*problem: openmdao.core.problem.Problem*, *with_optional_inputs: bool = False*) → *fastoad.openmdao.variables.variable_list.VariableList*

Creates a VariableList instance containing unconnected inputs of an OpenMDAO Problem.

Warning: problem.setup() must have been run.

If *optional_inputs* is False, only inputs that have numpy.nan as default value (hence considered as mandatory) will be in returned instance. Otherwise, all unconnected inputs will be in returned instance.

Parameters

- **problem** – OpenMDAO Problem instance to inspect
- **with_optional_inputs** – If True, returned instance will contain all unconnected inputs. Otherwise, it will contain only mandatory ones.

Returns VariableList instance

Module contents

Package for managing OpenMDAO variables

Submodules

fastoad.openmdao.exceptions module

Module for custom Exception classes linked to OpenMDAO

exception fastoad.openmdao.exceptions.**FASTOpenMDAONanInInputFile**(*input_file_path: str*, *nan_variable_names: List[str]*)

Bases: *fastoad.exceptions.FastError*

Raised if NaN values are read in input data file.

fastoad.openmdao.problem module

class fastoad.openmdao.problem.**FASTOADProblem**(**args*, ***kwargs*)

Bases: openmdao.core.problem.Problem

Vanilla OpenMDAO Problem except that it can write its outputs to a file.

It also runs *ValidityDomainChecker* after each *run_model()* or *run_driver()* (but it does nothing if no check has been registered).

Initialize attributes.

input_file_path
File path where *read_inputs()* will read inputs

output_file_path
File path where *write_outputs()* will write outputs

additional_variables
Variables that are not part of the problem but that should be written in output file.

run_model(*case_prefix=None*, *reset_iter_counts=True*)
Run the model by calling the root system's solve_nonlinear.

Parameters

- **case_prefix** (*str or None*) – Prefix to prepend to coordinates when recording. None means keep the preexisting prefix.
- **reset_iter_counts** (*bool*) – If True and model has been run previously, reset all iteration counters.

run_driver(*case_prefix=None*, *reset_iter_counts=True*)
Run the driver on the model.

Parameters

- **case_prefix** (*str or None*) – Prefix to prepend to coordinates when recording. None means keep the preexisting prefix.
- **reset_iter_counts** (*bool*) – If True and model has been run previously, reset all iteration counters.

Returns Failure flag; True if failed to converge, False is successful.

Return type bool

setup(**args*, ***kwargs*)
Set up the problem before run.

write_needed_inputs(*source_file_path: Optional[str] = None*, *source_formatter: Optional[fastoad.io.formatter.IVariableIOFormatter] = None*)

Writes the input file of the problem using its unconnected inputs.

Written value of each variable will be taken:

1. from input_data if it contains the variable
2. from defined default values in component definitions

Parameters

- **source_file_path** – if provided, variable values will be read from it
- **source_formatter** – the class that defines format of input file. if not provided, expected format will be the default one.

write_outputs()

Writes all outputs in the configured output file.

read_inputs()

Reads inputs of the problem.

property analysis: *fastoad.openmdao.problem.ProblemAnalysis*

Information about inner structure of this problem.

The collected data (internally stored) are used in several steps of the computation.

This analysis is performed once. Each subsequent usage reuses the obtained data.

To ensure the analysis is run again, use *reset_analysis()*.

reset_analysis()

Ensure a new problem analysis is done at new usage of *analysis*.

class fastoad.openmdao.problem.**AutoUnitsDefaultGroup**(***kwargs*)

Bases: openmdao.core.group.Group

OpenMDAO group that automatically use self.set_input_defaults() to resolve declaration conflicts in variable units.

Set the solvers to nonlinear and linear block Gauss–Seidel by default.

configure()

Configure this group to assign children settings.

This method may optionally be overidden by your Group's method.

You may only use this method to change settings on your children subsystems. This includes setting solvers in cases where you want to override the defaults.

You can assume that the full hierarchy below your level has been instantiated and has already called its own configure methods.

Available attributes: name pathname comm options system hiearchy with attribute access

class fastoad.openmdao.problem.**FASTOADModel**(***kwargs*)

Bases: *fastoad.openmdao.problem.AutoUnitsDefaultGroup*

OpenMDAO group that defines active submodels after the initialization of all its subsystems, and inherits from *AutoUnitsDefaultGroup* for resolving declaration conflicts in variable units.

It allows to have a submodel choice in the initialize() method of a FAST-OAD module, but to possibly override it with the definition of *active_submodels* (i.e. from the configuration file).

Set the solvers to nonlinear and linear block Gauss–Seidel by default.

active_submodels
Definition of active submodels that will be applied during setup()

setup()
Build this group.

This method should be overidden by your Group's method. The reason for using this method to add subsystem is to save memory and setup time when using your Group while running under MPI. This avoids the creation of systems that will not be used in the current process.

You may call 'add_subsystem' to add systems to this group. You may also issue connections, and set the linear and nonlinear solvers for this group level. You cannot safely change anything on children systems; use the 'configure' method instead.

Available attributes: name pathname comm options

fastoad.openmdao.problem.**get_variable_list_from_system**(*system: openmdao.core.system.System*, *get_promoted_names: bool = True*, *promoted_only: bool = True*, *io_status: str = 'all'*) → *fastoad.openmdao.variables.variable_list.VariableList*

Creates a VariableList instance containing inputs and outputs of any OpenMDAO System.

Convenience method that creates a FASTOADProblem problem with only provided *system* and uses `from_problem()`.

class fastoad.openmdao.problem.**ProblemAnalysis**(*problem: openmdao.core.problem.Problem*)
Bases: object

Class for retrieving information about the input OpenMDAO problem.

At least one setup operation is done on a copy of the problem. Two setup operations will be done if the problem has unfed dynamically shaped inputs.

problem: openmdao.core.problem.Problem
The analyzed problem

problem_variables: *fastoad.openmdao.variables.variable_list.VariableList*
All variables of the problem

dynamic_input_vars: *fastoad.openmdao.variables.variable_list.VariableList*
List variables that are inputs OF THE PROBLEM and dynamically shaped.

subsystem_order: list
Order of subsystems

ivc_var_names: list
Names of variables that are output of an IndepVarComp

analyze()
Gets information about inner structure of the associated problem.

fills_dynamically_shaped_inputs(*problem: openmdao.core.problem.Problem*)
Adds to the problem an IndepVarComp, that provides dummy variables to fit the dynamically shaped inputs of the analyzed problem.

Adding this IVC to the problem will allow to complete the setup operation.

The input problem should be the analyzed problem or a copy of it.

fastoad.openmdao.validity_checker module

For checking validity domain of OpenMDAO variables.

class fastoad.openmdao.validity_checker.**CheckRecord**(*variable_name*, *status*, *limit_value*, *limit_units*, *value*, *value_units*, *source_file*, *logger_name*)

Bases: tuple

A namedtuple that contains result of one variable check

limit_units
Alias for field number 3

limit_value
Alias for field number 2

logger_name
Alias for field number 7

source_file
Alias for field number 6

status
Alias for field number 1

value
Alias for field number 4

value_units
Alias for field number 5

variable_name
Alias for field number 0

class fastoad.openmdao.validity_checker.**ValidityStatus**(*value*)

Bases: enum.IntEnum

Simple enumeration for validity status.

OK = 0

TOO_LOW = -1

TOO_HIGH = 1

class fastoad.openmdao.validity_checker.**ValidityDomainChecker**(*limits: Optional[Dict[str, tuple]] = None*, *logger_name: Optional[str] = None*)

Bases: object

Decorator class that checks variable values against limit bounds

This class aims at producing a status of out of limits variables at the end of an OpenMDAO computation.

The point is to allow to define limit bounds when defining an OpenMDAO system, but to make the check on the OpenMDAO problem after the run.

When defining an OpenMDAO system, use this class as Python decorator to define validity domains:

```
@ValidityDomainChecker
class MyComponent(om.ExplicitComponent):
    ...
```

The above code will check values against lower and upper bounds that have been defined when adding OpenMDAO outputs.

Next code shows how to define lower and upper bounds, for inputs and/or outputs.

```
@ValidityDomainChecker(
    {
        "a:variable:with:two:bounds": (-10.0, 1.0),
        "a:variable:with:lower:bound:only": (0.0, None),
        "a:variable:with:upper:bound:only": (None, 4.2),
    },
)
class MyComponent(om.ExplicitComponent):
    ...
```

The defined domain limits supersedes lower and upper bounds from OpenMDAO output definitions, but only in the frame of ValidityDomainChecker. In any case, OpenMDAO process is not affected by usage of ValidityDomainChecker.

Validity status can be obtained through log messages from Python logging module after problem has been run with:

```
...
problem.run_model()
ValidityDomainChecker.check_problem_variables(problem)
```

Warnings: - Units of limit values defined in ValidityDomainChecker are assumed to be the

same as in add_input() and add_output() statements of decorated class

- Validity check currently only applies to scalar values

Parameters

- **limits** – a dictionary where keys are variable names and values are two-values tuples that give lower and upper bound. One bound can be set to None.
- **logger_name** – The named of the logger that will be used. If not provided, name of current module (i.e. "__name__"") will be used.

classmethod check_problem_variables(*problem: openmdao.core.problem.Problem*) → List[*fastoad.openmdao.validity_checker.CheckRecord*]

Checks variable values in provided problem.

Logs warnings for each variable that is out of registered limits.

problem.setup() must have been run.

Parameters **problem** –

Returns the list of checks

classmethod check_variables(*variables:* fastoad.openmdao.variables.variable_list.VariableList, *activated_only: bool = True*) → List[*fastoad.openmdao.validity_checker.CheckRecord*]

Check values of provided variables against registered limits.

Parameters

- **variables** –

- **activated_only** – if True, only activated checkers are considered.

Returns the list of checks

static log_records(*records: List[fastoad.openmdao.validity_checker.CheckRecord]*)

Logs warnings through Python logging module for each CheckRecord in provided list if it is not OK.

Parameters **records** –

Returns

fastoad.openmdao.whatsopt module

WhatsOpt-related operations.

fastoad.openmdao.whatsopt.**write_xdsm**(*problem: openmdao.core.problem.Problem*, *xdsm_file_path: Optional[str] = None*, *depth: int = 2*, *wop_server_url: Optional[str] = None*, *dry_run: bool = False*)

Makes WhatsOpt generate a XDSM in HTML file.

Parameters

- **problem** – a Problem instance. final_setup() must have been run.
- **xdsm_file_path** – the path for HTML file to be written (will overwrite if needed)
- **depth** – the depth analysis for WhatsOpt
- **wop_server_url** – URL of WhatsOpt server (if None, ether.onera.fr/whatsopt will be used)
- **dry_run** – if True, will run wop without sending any request to the server. Generated XDSM will be empty. (for test purpose only)

Module contents

fastoad.source_data_files package

Module contents

Submodules

fastoad.api module

This module gathers key FAST-OAD classes and functions for convenient import.

fastoad.constants module

Definition of globally used constants.

class fastoad.constants.**FlightPhase**(*value=<no_arg>*, *names=None*, *module=None*, *type=None*, *start=1*, *boundary=None*)

Bases: aenum.Enum

Enumeration of flight phases.

TAXI_OUT = 'taxi_out'

TAKEOFF = 'takeoff'

INITIAL_CLIMB = 'initial_climb'

CLIMB = 'climb'

CRUISE = 'cruise'

DESCENT = 'descent'

LANDING = 'landing'

TAXI_IN = 'taxi_in'

class fastoad.constants.**EngineSetting**(*value=<no_arg>*, *names=None*, *module=None*, *type=None*, *start=1*, *boundary=None*)

Bases: aenum.IntEnum

Enumeration of engine settings.

classmethod convert(*name: str*) → *fastoad.constants.EngineSetting*

Parameters **name** –

Returns the EngineSetting instance that matches the provided name (case-insensitive)

TAKEOFF = 1

CLIMB = 2

CRUISE = 3

IDLE = 4

class fastoad.constants.**RangeCategory**(*value=<no_arg>*, *names=None*, *module=None*, *type=None*, *start=1*, *boundary=None*)

Bases: aenum.Enum

Definition of lower and upper limits of aircraft range categories, in Nautical Miles.

can be used like:

```
>>> range_value = 800.
>>> range_value in RangeCategory.SHORT
True
```

which is equivalent to:

```
>>>  RangeCategory.SHORT.min() <= range_value <= RangeCategory.SHORT.max()
```

SHORT = (0.0, 1500.0)

SHORT_MEDIUM = (1500.0, 3000.0)

MEDIUM = (3000.0, 4500.0)

LONG = (4500.0, 6000.0)

VERY_LONG = (6000.0, 1000000.0)

min()

Returns minimum range in category

max()

Returns maximum range in category

fastoad.exceptions module

Module for custom Exception classes

exception fastoad.exceptions.**FastError**
Bases: Exception

Base Class for exceptions related to the FAST framework.

exception fastoad.exceptions.**NoSetupError**
Bases: *fastoad.exceptions.FastError*

No Setup Error.

This exception indicates that a setup of the OpenMDAO instance has not been done, but was expected to be.

exception fastoad.exceptions.**XMLReadError**
Bases: *fastoad.exceptions.FastError*

XML file read Error.

This exception indicates that an error occurred when reading an xml file.

exception fastoad.exceptions.**FastUnknownEngineSettingError**
Bases: *fastoad.exceptions.FastError*

Raised when an unknown engine setting code has been encountered

exception fastoad.exceptions.**FastUnexpectedKeywordArgument**(*bad_keyword*)
Bases: *fastoad.exceptions.FastError*

Raised when an instantiation is done with an incorrect keyword argument.

Module contents

CHAPTER

TWO

INDICES AND TABLES

- genindex
- modindex
- search

BIBLIOGRAPHY

[GHM+19] Justin S. Gray, John T. Hwang, Joaquim R. R. A. Martins, Kenneth T. Moore, and Bret A. Naylor. OpenMDAO: an open-source framework for multidisciplinary design, analysis, and optimization. *Structural and Multidisciplinary Optimization*, 59(4):1075–1104, April 2019. doi:10.1007/s00158-019-02211-z.

PYTHON MODULE INDEX

f

INDEX

A

B

C

D

E

F

G

M

N

O

P

Q

R

S

T

U

V

W

X