FAST-OAD

Release unknown

unknown

May 06, 2021

CONTENTS

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

For a detailed description of the input files and the command line interface, check out the *usage section*.

If you prefer to work 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 1.0, FAST-OAD aims at providing a stable core software to propose a safe base for development of custom models.

Models in FAST-OAD 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 1.0.2

- 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)

1.4.2 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)

1.4.3 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)

1.4.4 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.

1.4.5 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)

1.4.6 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)

1.4.7 Version 0.5.1-beta

- Now avoids apparition of numerous deprecation warnings from OpenMDAO.

1.4.8 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.

1.4.9 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).

1.4.10 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.

1.4.11 Version 0.3.1-beta

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

1.4.12 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.

1.4.13 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)

1.4.14 Version 0.2.1-beta

- Fixed compatibility with wop 1.9 for XDSM generation

1.4.15 Version 0.2.0b

- First beta release

1.4.16 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

– *Using FAST-OAD through Python*

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.

    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
```

(continues on next page)

(continued from previous page)

```
    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

```
# 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)
  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
```

(continues on next page)

(continued from previous page)

```
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 can be set with its own solvers to resolve cycles it may contains.

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, but additional keys can be used, depending on the options of the module. The list of available options of a module is available through the `list_modules` sub-command (see *How to get list of registered modules*).

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

```yaml
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

```yaml
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 generate a configuration file

FAST-OAD can provide a ready-to use configuration file with:

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

This generates the file *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 :ref`Generate conf file`), you may download our CeRAS01_baseline.xml and use it as source for generating your input 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, see WhatsOpt documentation for how to gain access to the online WhatsOpt server, or see WhatsOpt developer documentation to run your own server.

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

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

Note: it may take a couple of minutes

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

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

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

For example, for the three variables above, the matching part in XML file will be:

```xml
<data>
    <geometry>
        <wing>
            <area units="m**2">150.0</area>
        </wing>
    </geometry>
    <weight>
        <fuselage>
            <mass units="kg">10000.0</mass>
            <CG>
                <x units="m">20.0</x>
            </CG>
        </fuselage>
    </weight>
</data>
```

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

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

1.5.5 Mission module

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

Mission module

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

Mission file

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

mission description

Table 1: Mission elements

Type	Parts	Description
segment	N/A	The basic bricks that are provided by FAST-OAD.
phase	segment(s)	A free assembly of one or more segments.
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	A mission is what is computed by `fastoad.performances.mission`. Generally, it begins when engine starts and ends when engine stops.

Phase 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.

Example:

```
phases:
  initial_climb:                                  # Phase name
    engine_setting: takeoff                           # ---------------
    polar: data:aerodynamics:aircraft:takeoff         #   Common segment
    thrust_rate: 1.0                                  #   parameters
    time_step: 0.2                                    # ---------------
    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
    climb:                                          # Phase name
      ...                                             # Definition of the phase...
```

Route 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:

```yaml
routes:
  main_route:
    range:
      value: 3000.
      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 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 only the `parts` attribute that lists the *phase* and/or *route* names that compose the mission, with optionally a last item that is the `reserve` (see below).

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.**

About mission start:

- Each mission begins by default by taxi-out and takeoff phases, but these phases are not defined in the mission file. One reason for that is that the mass input for the mission is the TakeOff Weight, which is the aircraft weight at the end of takeoff phase.
- A taxi-out phase is automatically computed at begin of the mission. To ignore this phase, simply put its duration to 0. in the input data file.
- The takeoff data are simple inputs of the mission model. They have to be computed in a dedicated takeoff model (available soon), or provided in the input data file.

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:

```yaml
missions:
  sizing:
    parts:
      - route: main_route
      - route: diversion
      - phase: holding
      - phase: landing
      - phase: taxi_in
      - reserve:
          ref: main_route
          multiplier: 0.03
  operational:
    parts:
      - route: main_route
      - phase: landing
      - phase: taxi_in
```

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*
- *Segment target*
- *Special segment parameters*

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* Special parameters are detailed in *last section*.

Available segments are:

- `speed_change`
- `altitude_change`
- `cruise`
- `optimal_cruise`
- `holding`
- `taxi`

speed_change

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.

Python documentation: *SpeedChangeSegment*

Example:

```
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
```

altitude_change

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).

Python documentation: *AltitudeChangeSegment*

Examples:

```
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
```

```
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
```

```
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
  mach: constant                            #  level that provides maximum lift/drag
```

(continues on next page)

(continued from previous page)

```
  altitude:                                  #  at current mass.
    value: optimal_flight_level
```

cruise

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 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).

Python documentation: `ClimbAndCruiseSegment`

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
```

optimal_cruise

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.

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

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

Python documentation: `OptimalCruiseSegment`

```
segment: optimal_cruise
polar: data:aerodynamics:aircraft:cruise    # High speed aerodynamic polar
engine_setting: cruise
```

(continues on next page)

(continued from previous page)

```
target:
  ground_distance:                          # Cruise for 2000 nautical miles
    value: 2000
    unit: NM
```

holding

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.

Python documentation: *HoldSegment*

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
```

taxi

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.

Python documentation: *TaxiSegment*

Example:

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

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.

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*
- *polar*

engine_setting

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

This setting is used by the "rubber engine" propulsion model (see *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

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:

```yaml
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 set directly. As for *hard-coded value and unit*, if the concerned parameter is not dimensionless, SI units will be assumed.

Example:

```yaml
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. Then the value and unit will be set by some FAST-OAD module, or by the input file.

Example:

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

Contextual OpenMDAO variable

It is also possible to provide only a suffix for the variable name. Then the complete variable name will be decided by the hierarchy the defined parameter belongs to. The associated variable name will be `data:mission:<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.

A basic contextual variable is identified by a single tilde (~). In such case, `<suffix>` is the parameter name.

A generic contextual variable is **preceded** by a tilde. In such case, `<suffix>` is the name provided as value (without the tilde).

Example 1 : generic contextual variable in a route

```
routes:
  route_A:
    range: ~distance   # "distance" will be the used variable name
    parts:
      - ...

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

`route_A` contains the parameter `range` where a contextual variable name is affected. `route_A` is used as a step by both `mission_1` and `mission_2`.

Then the mission computation has among its inputs:

- `data:mission:mission_1:route_A:distance`
- `data:mission:mission_2:route_A:distance`

Example 2 : basic contextual variable in a flight phase

```yaml
phases:
  phase_a:
    thrust_rate: ~     # "thrust_rate" will be the used variable name

routes:
  route_A:
    range: ...
    parts:
      - phase_a
      - ...

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

`phase_a` contains the parameter `thrust_rate` where a contextual variable name is affected. `phase_a` is a 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:

- `data:mission:mission_1:route_A:phase_a:thrust_rate`
- `data:mission:mission_2:phase_a:thrust_rate`

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:

- *propulsion_id*
- *mission_file_path*
- *out_file*
- *mission_name*
- *use_initializer_iteration*
- *adjust_fuel*
- *compute_TOW*

- *add_solver*
- *is_sizing*

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.

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

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.

```yaml
# 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 FlightPoint class

The *FlightPoint* class is designed to store flight parameters for one flight point.

It is meant to be the class that performance modules will work with, and that will be exchanged with propulsion models.

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:

```python
>>> 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")
```

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*.

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/

[ ... ]

# 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
│   ├── __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
```

(continues on next page)

(continued from previous page)

```
│   └── 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 (see *How to add custom OpenMDAO modules to 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.

To declare your custom modules as a FAST-OAD plugin, you have to package them the usual way and declare them as a plugin with `fastoad_model` as plugin group name.

This can be done classically with setuptools. It can also be done with Poetry, which is the way described below:

Plugin declaration

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

```
...

[tool.poetry.plugins."fastoad_model"]
"internal_models" = "start_trek.drives"

...
```

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 `STST_drive_models`, as below:

```
...

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

(continues on next page)

(continued from previous page)

```
...
```

It 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.

Poetry can also publish to another destination.

Please see here for detailed information.

1.6 fastoad

1.6.1 fastoad package

Subpackages

fastoad.cmd package

Subpackages

Submodules

fastoad.cmd.api module

API

`fastoad.cmd.api.`**`generate_configuration_file`**(*configuration_file_path: str*, *overwrite: bool = False*)

Generates a sample configuration file.

Parameters

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

Raises ***`FastFileExistsError`*** – if overwrite==False and configuration_file_path already exists

`fastoad.cmd.api.`**`generate_inputs`**(*configuration_file_path: str*, *source_path: Optional[str] = None*, *source_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_path** – path of file data will be taken from
- **source_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 *FastFileExistsError* – 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.

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

fastoad.cmd.api.**list_modules**(*configuration_file_path: Optional[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 configuration_file_path is given and if it defines paths where there are registered systems, they will be listed too.

param configuration_file_path

param out the output stream or a path for the output file (None means sys.stdout)

param overwrite if True and out is a file path, the file will be written even if one already exists

param verbose if True, shows detailed information for each system if False, shows only identifier and path of each system

param 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

Raises *FastFileExistsError* – 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: str = 'n2.html', overwrite: bool = False*)
Write the N2 diagram of the problem in file n2.html

Parameters

- **configuration_file_path** –
- **n2_file_path** –
- **overwrite** –

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

Parameters

- **configuration_file_path** –
- **xdsm_file_path** –
- **overwrite** –
- **depth** –
- **wop_server_url** –
- **api_key** –

Returns

`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

`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

`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.exceptions module

Exception for cmd package

exception `fastoad.cmd.exceptions.`**`FastFileExistsError`**(**args*)

Bases: *`fastoad.exceptions.FastError`*

Raised when asked for writing a file that already exists

fastoad.cmd.fast module

Command Line Interface.

class `fastoad.cmd.fast.`**`Main`**

Bases: `object`

Class for managing command line and doing associated actions

`run`()

Main function.

`fastoad.cmd.fast.`**`main`**()

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*) → 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.

Returns bar plot figure

fastoad.gui.analysis_and_plots.**mass_breakdown_sun_plot**(*aircraft_file_path: str, file_formatter=None*)

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.

Returns sunburst 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(*change=None*) → IPython.core.display.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.FASTOADProblemConfigura*

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.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.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.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.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

class fastoad.io.configuration.configuration.**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.

Parameters ****kwargs** (*dict*) – dict of arguments available here and in all descendants of this Group.

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 hieararchy with attribute access

fastoad.io.configuration.exceptions module

Exceptions for package configuration

exception fastoad.io.configuration.exceptions.**FASTConfigurationBaseKeyBuildingError**(*original_e Exception, key: str, value=No*

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**

Bases: *fastoad.io.configuration.exceptions.FASTConfigurationBaseKeyBuildingError*

Class for managing errors that result from trying to set an attribute by eval.

Constructor

exception fastoad.io.configuration.exceptions.**FASTConfigurationNanInInputFile**(*input_file_path: str, nan_variable_nam List[str]*)

Bases: *fastoad.exceptions.FastError*

Raised if NaN values are read in input data file.

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
List of variable names as set in *set()*

property xpaths
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

unit_translation
Used for converting read units in units recognized by OpenMDAO Dict keys can use regular expressions.

set_translator(*translator: fastoad.io.xml.translator.VarXpathTranslator*)
Sets the VarXpathTranslator() instance that rules how OpenMDAO variable are matched to XML Path.

Parameters **translator** –

read_variables(*data_source: Union[str, IO]*) → *fastoad.openmdao.variables.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.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.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.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.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.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.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.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**(*file_path: str, formatter: Optional[fastoad.io.formatter.IVariableIOFormatter] = None*)

Bases: *fastoad.openmdao.variables.VariableList*

Class for managing FAST-OAD data files.

Behaves like *VariableList* class but has *load()* and *save()* methods.

Parameters

- **file_path** – if file exists, it will be loaded.
- **formatter** – a class that determines the file format to be used. Defaults to FAST-OAD native format. See *VariableIO* for more information.

property file_path
Path of data file.

property formatter
Class that defines the file format.

load()
Loads file content.

save()
Saves current state of variables in file.

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**(*altitude: Union[float, Sequence[float]], delta_t: float = 0.0, altitude_in_feet: bool = True*)

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
Temperature increment applied to whole temperature profile.

property temperature
Temperature in K.

property pressure
Pressure in Pa.

property density
Density in kg/m3.

property speed_of_sound
Speed of sound in m/s.

property kinematic_viscosity
Kinematic viscosity in m2/s.

property mach
Mach number.

property true_airspeed
True airspeed (TAS) in m/s.

property equivalent_airspeed
Equivalent airspeed (EAS) in m/s.

property unitary_reynolds
Unitary Reynolds number in 1/m.

class fastoad.model_base.atmosphere.**AtmosphereSI** (*altitude: Union[float, Sequence[float]], delta_t: float = 0.0*)

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.flight_point module

Structure for managing flight point data.

class fastoad.model_base.flight_point.**FlightPoint**(*time: float = 0.0, altitude: Optional[float] = None, ground_distance: float = 0.0, mass: Optional[float] = None, 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, 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, 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.

ground_distance: float = 0.0
Covered ground distance in meters.

mass: float = None
Mass 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.

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.

name: str = None
Name of current phase.

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.

Parameters ****kwargs** (*dict of keyword arguments*) – Keyword arguments that will be mapped into the Component options.

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.

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.aerodynamics package

Subpackages

fastoad.models.aerodynamics.components package

Subpackages

Submodules

fastoad.models.aerodynamics.components.cd0 module

class fastoad.models.aerodynamics.components.cd0.**CD0**(***kwargs*)

Bases: openmdao.core.group.Group

Set the solvers to nonlinear and linear block Gauss–Seidel by default.

Parameters ****kwargs** (*dict*) – dict of arguments available here and in all descendants of this Group.

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

fastoad.models.aerodynamics.components.cd0_fuselage module

FAST - Copyright (c) 2016 ONERA ISAE

class fastoad.models.aerodynamics.components.cd0_fuselage.**Cd0Fuselage**(***kwargs*)
Bases: openmdao.core.explicitcomponent.ExplicitComponent

Store some bound methods so we can detect runtime overrides.

Parameters ****kwargs** (*dict of keyword arguments*) – Keyword arguments that will be mapped into the Component options.

initialize()
Perform any one-time initialization run at instantiation.

setup()
Declare inputs and outputs.

Available attributes: name pathname comm options

compute(*inputs, outputs*)
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.aerodynamics.components.cd0_ht module

FAST - Copyright (c) 2016 ONERA ISAE

class fastoad.models.aerodynamics.components.cd0_ht.**Cd0HorizontalTail**(***kwargs*)
Bases: openmdao.core.explicitcomponent.ExplicitComponent

Store some bound methods so we can detect runtime overrides.

Parameters ****kwargs** (*dict of keyword arguments*) – Keyword arguments that will be mapped into the Component options.

initialize()
Perform any one-time initialization run at instantiation.

setup()
Declare inputs and outputs.

Available attributes: name pathname comm options

compute(*inputs, outputs*)
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.aerodynamics.components.cd0_nacelle_pylons module

FAST - Copyright (c) 2016 ONERA ISAE

class fastoad.models.aerodynamics.components.cd0_nacelle_pylons.**Cd0NacelleAndPylons**(***kwargs*)

Bases: openmdao.core.explicitcomponent.ExplicitComponent

Store some bound methods so we can detect runtime overrides.

Parameters ****kwargs** (*dict of keyword arguments*) – Keyword arguments that will be mapped into the Component options.

initialize()

Perform any one-time initialization run at instantiation.

setup()

Declare inputs and outputs.

Available attributes: name pathname comm options

compute(*inputs*, *outputs*)

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.aerodynamics.components.cd0_total module

FAST - Copyright (c) 2016 ONERA ISAE

class fastoad.models.aerodynamics.components.cd0_total.**Cd0Total**(***kwargs*)

Bases: openmdao.core.explicitcomponent.ExplicitComponent

Store some bound methods so we can detect runtime overrides.

Parameters ****kwargs** (*dict of keyword arguments*) – Keyword arguments that will be mapped into the Component options.

initialize()

Perform any one-time initialization run at instantiation.

setup()

Declare inputs and outputs.

Available attributes: name pathname comm options

compute (*inputs*, *outputs*)
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.aerodynamics.components.cd0_vt module

FAST - Copyrigvt (c) 2016 ONERA ISAE

class fastoad.models.aerodynamics.components.cd0_vt.**Cd0VerticalTail** (***kwargs*)
Bases: openmdao.core.explicitcomponent.ExplicitComponent

Store some bound methods so we can detect runtime overrides.

Parameters ****kwargs** (*dict of keyword arguments*) – Keyword arguments that will be mapped into the Component options.

initialize ()
Perform any one-time initialization run at instantiation.

setup ()
Declare inputs and outputs.

Available attributes: name pathname comm options

compute (*inputs*, *outputs*)
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.aerodynamics.components.cd0_wing module

FAST - Copyright (c) 2016 ONERA ISAE

class fastoad.models.aerodynamics.components.cd0_wing.**Cd0Wing** (***kwargs*)
Bases: openmdao.core.explicitcomponent.ExplicitComponent

Store some bound methods so we can detect runtime overrides.

Parameters ****kwargs** (*dict of keyword arguments*) – Keyword arguments that will be mapped into the Component options.

initialize()
Perform any one-time initialization run at instantiation.

setup()
Declare inputs and outputs.

Available attributes: name pathname comm options

compute(*inputs*, *outputs*)
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.aerodynamics.components.cd_compressibility module

Compressibility drag computation.

class fastoad.models.aerodynamics.components.cd_compressibility.**CdCompressibility**(***kwargs*)
Bases: `openmdao.core.explicitcomponent.ExplicitComponent`

Computation of drag increment due to compressibility effects.

Formula from §4.2.4 of [[DCAC14]]. This formula can be used for aircraft before year 2000. Earlier aircraft have more optimized wing profiles that are expected to limit the compressibility drag below 2 drag counts. Until a better model can be provided, the variable *tuning:aerodynamics:aircraft:cruise:CD:compressibility:characteristic_mach_increment* allows to move the characteristic Mach number, thus moving the CD divergence to higher Mach numbers.

Store some bound methods so we can detect runtime overrides.

Parameters ****kwargs** (`dict of keyword arguments`) – Keyword arguments that will be mapped into the Component options.

setup()
Declare inputs and outputs.

Available attributes: name pathname comm options

compute(*inputs*, *outputs*)
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.aerodynamics.components.cd_trim module

FAST - Copyright (c) 2016 ONERA ISAE

class fastoad.models.aerodynamics.components.cd_trim.**CdTrim**(***kwargs*)
Bases: openmdao.core.explicitcomponent.ExplicitComponent

Store some bound methods so we can detect runtime overrides.

Parameters ****kwargs** (*dict of keyword arguments*) – Keyword arguments that will be mapped into the Component options.

initialize()
Perform any one-time initialization run at instantiation.

setup()
Declare inputs and outputs.

Available attributes: name pathname comm options

compute(*inputs*, *outputs*)
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.aerodynamics.components.compute_low_speed_aero module

FAST - Copyright (c) 2016 ONERA ISAE

class fastoad.models.aerodynamics.components.compute_low_speed_aero.**ComputeAerodynamicsLowS**
Bases: openmdao.core.explicitcomponent.ExplicitComponent

Czalpha from Raymer Eq 12.6 TODO: complete source

Store some bound methods so we can detect runtime overrides.

Parameters ****kwargs** (*dict of keyword arguments*) – Keyword arguments that will be mapped into the Component options.

setup()
Declare inputs and outputs.

Available attributes: name pathname comm options

compute(*inputs*, *outputs*)
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.aerodynamics.components.compute_max_cl_landing module

FAST - Copyright (c) 2016 ONERA ISAE

class fastoad.models.aerodynamics.components.compute_max_cl_landing.**ComputeMaxClLanding**(***k*

Bases: openmdao.core.explicitcomponent.ExplicitComponent

Store some bound methods so we can detect runtime overrides.

Parameters ****kwargs** (*dict of keyword arguments*) – Keyword arguments that will be mapped into the Component options.

setup()

Declare inputs and outputs.

Available attributes: name pathname comm options

compute(*inputs*, *outputs*)

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.aerodynamics.components.compute_polar module

FAST - Copyright (c) 2016 ONERA ISAE

class fastoad.models.aerodynamics.components.compute_polar.**PolarType**(*value*)

Bases: enum.Enum

An enumeration.

HIGH_SPEED = 'high_speed'

LOW_SPEED = 'low_speed'

TAKEOFF = 'takeoff'

LANDING = 'landing'

class fastoad.models.aerodynamics.components.compute_polar.**ComputePolar**(***kwargs*)

Bases: openmdao.core.explicitcomponent.ExplicitComponent

Store some bound methods so we can detect runtime overrides.

Parameters ****kwargs** (*dict of keyword arguments*) – Keyword arguments that will be mapped into the Component options.

initialize()
Perform any one-time initialization run at instantiation.

setup()
Declare inputs and outputs.

Available attributes: name pathname comm options

compute(*inputs*, *outputs*)
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.aerodynamics.components.compute_polar.**get_optimum_ClCd**(*ClCd*)

fastoad.models.aerodynamics.components.compute_reynolds module

FAST - Copyright (c) 2016 ONERA ISAE

class fastoad.models.aerodynamics.components.compute_reynolds.**ComputeReynolds**(***kwargs*)
Bases: openmdao.core.explicitcomponent.ExplicitComponent

Store some bound methods so we can detect runtime overrides.

Parameters ****kwargs** (*dict of keyword arguments*) – Keyword arguments that will be mapped into the Component options.

initialize()
Perform any one-time initialization run at instantiation.

setup()
Declare inputs and outputs.

Available attributes: name pathname comm options

compute(*inputs*, *outputs*)
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.aerodynamics.components.high_lift_aero module

Computation of lift and drag increment due to high-lift devices

class fastoad.models.aerodynamics.components.high_lift_aero.**ComputeDeltaHighLift**(***kwargs*)

Bases: openmdao.core.explicitcomponent.ExplicitComponent

Provides lift and drag increments due to high-lift devices

Store some bound methods so we can detect runtime overrides.

Parameters ****kwargs** (*dict of keyword arguments*) – Keyword arguments that will be mapped into the Component options.

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.aerodynamics.components.initialize_cl module

FAST - Copyright (c) 2016 ONERA ISAE

class fastoad.models.aerodynamics.components.initialize_cl.**InitializeClPolar**(***kwargs*)

Bases: openmdao.core.explicitcomponent.ExplicitComponent

Store some bound methods so we can detect runtime overrides.

Parameters ****kwargs** (*dict of keyword arguments*) – Keyword arguments that will be mapped into the Component options.

initialize()

Perform any one-time initialization run at instantiation.

setup()

Declare inputs and outputs.

Available attributes: name pathname comm options

compute(*inputs*, *outputs*)

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.aerodynamics.components.oswald module

Computation of Oswald coefficient

class fastoad.models.aerodynamics.components.oswald.**OswaldCoefficient**(***kwargs*)

Bases: openmdao.core.explicitcomponent.ExplicitComponent

Computes Oswald efficiency number

Store some bound methods so we can detect runtime overrides.

Parameters ****kwargs** (*dict of keyword arguments*) – Keyword arguments that will be mapped into the Component options.

initialize()

Perform any one-time initialization run at instantiation.

setup()

Declare inputs and outputs.

Available attributes: name pathname comm options

compute(*inputs*, *outputs*)

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.aerodynamics.external package

Subpackages

fastoad.models.aerodynamics.external.xfoil package

Subpackages

fastoad.models.aerodynamics.external.xfoil.xfoil699 package

Module contents

Submodules

fastoad.models.aerodynamics.external.xfoil.xfoil_polar module

This module launches XFOIL computations

class fastoad.models.aerodynamics.external.xfoil.xfoil_polar.**XfoilPolar**(***kwargs*)
Bases: openmdao.components.external_code_comp.ExternalCodeComp

Runs a polar computation with XFOIL and returns the 2D max lift coefficient

Intialize the ExternalCodeComp component.

Parameters ****kwargs** (*dict of keyword arguments*) – Keyword arguments that will be mapped into the Component options.

initialize()
Perform any one-time initialization run at instantiation.

setup()
Declare inputs and outputs.

Available attributes: name pathname comm options

compute(*inputs*, *outputs*)
Run this component.

User should call this method from their overriden compute method.

Parameters

- **inputs** (*Vector*) – Unscaled, dimensional input variables read via inputs[key].
- **outputs** (*Vector*) – Unscaled, dimensional output variables read via outputs[key].

Module contents

Module for OpenMDAO-embedded XFOIL

Module contents

Submodules

fastoad.models.aerodynamics.aerodynamics module

FAST - Copyright (c) 2016 ONERA ISAE

class fastoad.models.aerodynamics.aerodynamics.**Aerodynamics**(***kwargs*)
Bases: openmdao.core.group.Group

Set the solvers to nonlinear and linear block Gauss–Seidel by default.

Parameters ****kwargs** (*dict*) – dict of arguments available here and in all descendants of this Group.

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.models.aerodynamics.aerodynamics_high_speed module

FAST - Copyright (c) 2016 ONERA ISAE

class fastoad.models.aerodynamics.aerodynamics_high_speed.**AerodynamicsHighSpeed**(***kwargs*)

Bases: openmdao.core.group.Group

Computes aerodynamic polar of the aircraft in cruise conditions.

Drag contributions of each part of the aircraft are computed though analytical models.

Set the solvers to nonlinear and linear block Gauss–Seidel by default.

Parameters ****kwargs** (*dict*) – dict of arguments available here and in all descendants of this Group.

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.models.aerodynamics.aerodynamics_landing module

Aero computation for landing phase

class fastoad.models.aerodynamics.aerodynamics_landing.**AerodynamicsLanding**(***kwargs*)

Bases: openmdao.core.group.Group

Computes aerodynamic characteristics at landing.

- Computes CL and CD increments due to high-lift devices at landing.
- Computes maximum CL of the aircraft in landing conditions.

Maximum 2D CL without high-lift is computed using XFoil (or provided as input if option use_xfoil is set to False). 3D CL is deduced using sweep angle.

Contribution of high-lift devices is modelled according to their geometry (span and chord ratio) and their deflection angles.

Set the solvers to nonlinear and linear block Gauss–Seidel by default.

Parameters `**kwargs` (*dict*) – dict of arguments available here and in all descendants of this Group.

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.aerodynamics.aerodynamics_landing.**ComputeMachReynolds**(***kwargs*)

Bases: openmdao.core.explicitcomponent.ExplicitComponent

Mach and Reynolds computation

Store some bound methods so we can detect runtime overrides.

Parameters `**kwargs` (*dict of keyword arguments*) – Keyword arguments that will be mapped into the Component options.

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.aerodynamics.aerodynamics_landing.**Compute3DMaxCL**(***kwargs*)

Bases: openmdao.core.explicitcomponent.ExplicitComponent

Computes 3D max CL from 2D CL (XFOIL-computed) and sweep angle

Store some bound methods so we can detect runtime overrides.

Parameters `**kwargs` (*dict of keyword arguments*) – Keyword arguments that will be mapped into the Component options.

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.aerodynamics.aerodynamics_low_speed module

FAST - Copyright (c) 2016 ONERA ISAE

class fastoad.models.aerodynamics.aerodynamics_low_speed.**AerodynamicsLowSpeed** (***kwargs*)
Bases: openmdao.core.group.Group

Models for low speed aerodynamics

Set the solvers to nonlinear and linear block Gauss–Seidel by default.

Parameters ****kwargs** (*dict*) – dict of arguments available here and in all descendants of this Group.

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.models.aerodynamics.aerodynamics_takeoff module

class fastoad.models.aerodynamics.aerodynamics_takeoff.**AerodynamicsTakeoff** (***kwargs*)
Bases: openmdao.core.group.Group

Computes aerodynamic characteristics at takeoff.

- Computes CL and CD increments due to high-lift devices at takeoff.

Set the solvers to nonlinear and linear block Gauss–Seidel by default.

Parameters ****kwargs** (*dict*) – dict of arguments available here and in all descendants of this Group.

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.models.aerodynamics.constants module

Module contents

fastoad.models.geometry package

Subpackages

fastoad.models.geometry.geom_components package

Subpackages

fastoad.models.geometry.geom_components.fuselage package

Submodules

fastoad.models.geometry.geom_components.fuselage.compute_cnbeta_fuselage module

Estimation of yawing moment due to sideslip

class fastoad.models.geometry.geom_components.fuselage.compute_cnbeta_fuselage.**ComputeCnBet**

Bases: openmdao.core.explicitcomponent.ExplicitComponent

Yawing moment due to sideslip estimation

Store some bound methods so we can detect runtime overrides.

Parameters ****kwargs** (*dict of keyword arguments*) – Keyword arguments that will be mapped into the Component options.

setup()

Declare inputs and outputs.

Available attributes: name pathname comm options

compute(*inputs*, *outputs*)

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.geometry.geom_components.fuselage.compute_fuselage module

Estimation of geometry of fuselase part A - Cabin (Commercial)

class fastoad.models.geometry.geom_components.fuselage.compute_fuselage.**ComputeFuselageGeor**

Bases: openmdao.core.explicitcomponent.ExplicitComponent

Geometry of fuselage part A - Cabin (Commercial) estimation

Store some bound methods so we can detect runtime overrides.

Parameters ****kwargs** (*dict of keyword arguments*) – Keyword arguments that will be mapped into the Component options.

setup()

Declare inputs and outputs.

Available attributes: name pathname comm options

compute(*inputs*, *outputs*)

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.geometry.geom_components.fuselage.compute_fuselage.**ComputeFuselageGeor**

Bases: openmdao.core.explicitcomponent.ExplicitComponent

Geometry of fuselage part A - Cabin (Commercial) estimation

Store some bound methods so we can detect runtime overrides.

Parameters ****kwargs** (*dict of keyword arguments*) – Keyword arguments that will be mapped into the Component options.

setup()

Declare inputs and outputs.

Available attributes: name pathname comm options

compute(*inputs*, *outputs*)

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

Estimation of fuselage geometry

fastoad.models.geometry.geom_components.ht package

Subpackages

fastoad.models.geometry.geom_components.ht.components package

Submodules

fastoad.models.geometry.geom_components.ht.components.compute_ht_chords module

Estimation of horizontal tail chords and span

class fastoad.models.geometry.geom_components.ht.components.compute_ht_chords.**ComputeHTChor**

Bases: openmdao.core.explicitcomponent.ExplicitComponent

Horizontal tail chords and span estimation

Store some bound methods so we can detect runtime overrides.

Parameters ****kwargs** (*dict of keyword arguments*) – Keyword arguments that will be mapped into the Component options.

setup()

Declare inputs and outputs.

Available attributes: name pathname comm options

compute(*inputs*, *outputs*)

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.geometry.geom_components.ht.components.compute_ht_cl_alpha module

Estimation of horizontal tail lift coefficient

class fastoad.models.geometry.geom_components.ht.components.compute_ht_cl_alpha.**ComputeHTC**

Bases: openmdao.core.explicitcomponent.ExplicitComponent

Horizontal tail lift coefficient estimation

Store some bound methods so we can detect runtime overrides.

Parameters ****kwargs** (*dict of keyword arguments*) – Keyword arguments that will be mapped into the Component options.

setup()

Declare inputs and outputs.

Available attributes: name pathname comm options

compute(*inputs*, *outputs*)

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.geometry.geom_components.ht.components.compute_ht_mac module

Estimation of horizontal tail mean aerodynamic chords

class fastoad.models.geometry.geom_components.ht.components.compute_ht_mac.**ComputeHTMAC**(***k*

Bases: openmdao.core.explicitcomponent.ExplicitComponent

Horizontal tail mean aerodynamic chord estimation

Store some bound methods so we can detect runtime overrides.

Parameters ****kwargs** (*dict of keyword arguments*) – Keyword arguments that will be mapped into the Component options.

setup()

Declare inputs and outputs.

Available attributes: name pathname comm options

compute(*inputs*, *outputs*)

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.geometry.geom_components.ht.components.compute_ht_sweep module

Estimation of horizontal tail sweeps

class fastoad.models.geometry.geom_components.ht.components.compute_ht_sweep.**ComputeHTSweep**

Bases: openmdao.core.explicitcomponent.ExplicitComponent

Horizontal tail sweeps estimation

Store some bound methods so we can detect runtime overrides.

Parameters ****kwargs** (*dict of keyword arguments*) – Keyword arguments that will be mapped into the Component options.

setup()

Declare inputs and outputs.

Available attributes: name pathname comm options

compute(*inputs*, *outputs*)

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

Estimation of horizontal tail geometry (components)

Submodules

fastoad.models.geometry.geom_components.ht.compute_horizontal_tail module

Estimation of geometry of horizontal tail

class fastoad.models.geometry.geom_components.ht.compute_horizontal_tail.**ComputeHorizontalT**

Bases: openmdao.core.group.Group

Horizontal tail geometry estimation

Set the solvers to nonlinear and linear block Gauss–Seidel by default.

Parameters ****kwargs** (*dict*) – dict of arguments available here and in all descendants of this Group.

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

Module contents

Estimation of horizontal tail geometry (global)

fastoad.models.geometry.geom_components.nacelle_pylons package

Submodules

fastoad.models.geometry.geom_components.nacelle_pylons.compute_nacelle_pylons module

Estimation of nacelle and pylon geometry

class fastoad.models.geometry.geom_components.nacelle_pylons.compute_nacelle_pylons.**Compute**
Bases: openmdao.core.explicitcomponent.ExplicitComponent

Nacelle and pylon geometry estimation

Store some bound methods so we can detect runtime overrides.

Parameters ****kwargs** (*dict of keyword arguments*) – Keyword arguments that will be mapped into the Component options.

setup()
Declare inputs and outputs.

Available attributes: name pathname comm options

compute(*inputs*, *outputs*)
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

Estimation of nacelle and pylons

fastoad.models.geometry.geom_components.vt package

Subpackages

fastoad.models.geometry.geom_components.vt.components package

Submodules

fastoad.models.geometry.geom_components.vt.components.compute_vt_chords module

Estimation of vertical tail chords and span

class fastoad.models.geometry.geom_components.vt.components.compute_vt_chords.**ComputeVTChor**

Bases: openmdao.core.explicitcomponent.ExplicitComponent

Vertical tail chords and span estimation

Store some bound methods so we can detect runtime overrides.

Parameters ****kwargs** (*dict of keyword arguments*) – Keyword arguments that will be mapped into the Component options.

setup()

Declare inputs and outputs.

Available attributes: name pathname comm options

compute(*inputs*, *outputs*)

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.geometry.geom_components.vt.components.compute_vt_clalpha module

Estimation of vertical tail lift coefficient

class fastoad.models.geometry.geom_components.vt.components.compute_vt_clalpha.**ComputeVTCla**

Bases: openmdao.core.explicitcomponent.ExplicitComponent

Vertical tail lift coefficient estimation

Store some bound methods so we can detect runtime overrides.

Parameters `**kwargs` (*dict of keyword arguments*) – Keyword arguments that will be mapped into the Component options.

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.geometry.geom_components.vt.components.compute_vt_distance module

Estimation of vertical tail distance

class fastoad.models.geometry.geom_components.vt.components.compute_vt_distance.**ComputeVTD**

Bases: openmdao.core.explicitcomponent.ExplicitComponent

Vertical tail distance estimation

Store some bound methods so we can detect runtime overrides.

Parameters `**kwargs` (*dict of keyword arguments*) – Keyword arguments that will be mapped into the Component options.

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.geometry.geom_components.vt.components.compute_vt_mac module

Estimation of vertical tail mean aerodynamic chords

class fastoad.models.geometry.geom_components.vt.components.compute_vt_mac.**ComputeVTMAC**(***k*

Bases: openmdao.core.explicitcomponent.ExplicitComponent

Vertical tail mean aerodynamic chord estimation

Store some bound methods so we can detect runtime overrides.

Parameters ****kwargs** (*dict of keyword arguments*) – Keyword arguments that will be mapped into the Component options.

setup()

Declare inputs and outputs.

Available attributes: name pathname comm options

compute(*inputs, outputs*)

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.geometry.geom_components.vt.components.compute_vt_sweep module

Estimation of vertical tail sweeps

class fastoad.models.geometry.geom_components.vt.components.compute_vt_sweep.**ComputeVTSweep**

Bases: openmdao.core.explicitcomponent.ExplicitComponent

Vertical tail sweeps estimation

Store some bound methods so we can detect runtime overrides.

Parameters ****kwargs** (*dict of keyword arguments*) – Keyword arguments that will be mapped into the Component options.

setup()

Declare inputs and outputs.

Available attributes: name pathname comm options

compute(*inputs, outputs*)

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

Estimation of vertical tail geometry (components)

Submodules

fastoad.models.geometry.geom_components.vt.compute_vertical_tail module

Estimation of geometry of vertical tail

class fastoad.models.geometry.geom_components.vt.compute_vertical_tail.**ComputeVerticalTailC**

Bases: openmdao.core.group.Group

Vertical tail geometry estimation

Set the solvers to nonlinear and linear block Gauss–Seidel by default.

Parameters ****kwargs** (*dict*) – dict of arguments available here and in all descendants of this Group.

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

Module contents

Estimation of vertical tail geometry (global)

fastoad.models.geometry.geom_components.wing package

Subpackages

fastoad.models.geometry.geom_components.wing.components package

Submodules

fastoad.models.geometry.geom_components.wing.components.compute_b_50 module

Estimation of wing B50

class fastoad.models.geometry.geom_components.wing.components.compute_b_50.**ComputeB50**(***kwa*
Bases: openmdao.core.explicitcomponent.ExplicitComponent

Wing B50 estimation

Store some bound methods so we can detect runtime overrides.

Parameters ****kwargs** (*dict of keyword arguments*) – Keyword arguments that will be mapped into the Component options.

setup()
Declare inputs and outputs.

Available attributes: name pathname comm options

compute(*inputs*, *outputs*)
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.geometry.geom_components.wing.components.compute_cl_alpha module

Estimation of wing lift coefficient

class fastoad.models.geometry.geom_components.wing.components.compute_cl_alpha.**ComputeCLal**
Bases: openmdao.core.explicitcomponent.ExplicitComponent

Wing lift coefficient estimation

Store some bound methods so we can detect runtime overrides.

Parameters ****kwargs** (*dict of keyword arguments*) – Keyword arguments that will be mapped into the Component options.

setup()
Declare inputs and outputs.

Available attributes: name pathname comm options

compute(*inputs*, *outputs*)
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.geometry.geom_components.wing.components.compute_l1_l4 module

Estimation of wing chords (l1 and l4)

class fastoad.models.geometry.geom_components.wing.components.compute_l1_l4.**ComputeL1AndL4W**

Bases: openmdao.core.explicitcomponent.ExplicitComponent

Wing chords (l1 and l4) estimation

Store some bound methods so we can detect runtime overrides.

Parameters ****kwargs** (*dict of keyword arguments*) – Keyword arguments that will be mapped into the Component options.

setup()

Declare inputs and outputs.

Available attributes: name pathname comm options

compute(*inputs*, *outputs*)

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.geometry.geom_components.wing.components.compute_l2_l3 module

Estimation of wing chords (l2 and l3)

class fastoad.models.geometry.geom_components.wing.components.compute_l2_l3.**ComputeL2AndL3W**

Bases: openmdao.core.explicitcomponent.ExplicitComponent

Wing chords (l2 and l3) estimation

Store some bound methods so we can detect runtime overrides.

Parameters ****kwargs** (*dict of keyword arguments*) – Keyword arguments that will be mapped into the Component options.

setup()

Declare inputs and outputs.

Available attributes: name pathname comm options

compute(*inputs*, *outputs*)

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.geometry.geom_components.wing.components.compute_mac_wing module

Estimation of wing mean aerodynamic chord

class fastoad.models.geometry.geom_components.wing.components.compute_mac_wing.**ComputeMACW**

Bases: openmdao.core.explicitcomponent.ExplicitComponent

Wing mean aerodynamic chord estimation

Store some bound methods so we can detect runtime overrides.

Parameters ****kwargs** (*dict of keyword arguments*) – Keyword arguments that will be mapped into the Component options.

setup()

Declare inputs and outputs.

Available attributes: name pathname comm options

compute(*inputs*, *outputs*)

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.geometry.geom_components.wing.components.compute_mfw module

Estimation of max fuel weight

class fastoad.models.geometry.geom_components.wing.components.compute_mfw.**ComputeMFW**(***kwarg*

Bases: openmdao.core.explicitcomponent.ExplicitComponent

Max fuel weight estimation

Store some bound methods so we can detect runtime overrides.

Parameters ****kwargs** (*dict of keyword arguments*) – Keyword arguments that will be mapped into the Component options.

setup()

Declare inputs and outputs.

Available attributes: name pathname comm options

compute(*inputs*, *outputs*)

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.geometry.geom_components.wing.components.compute_sweep_wing module

Estimation of wing sweeps

class fastoad.models.geometry.geom_components.wing.components.compute_sweep_wing.**ComputeSwe**

Bases: openmdao.core.explicitcomponent.ExplicitComponent

Wing sweeps estimation

Store some bound methods so we can detect runtime overrides.

Parameters ****kwargs** (*dict of keyword arguments*) – Keyword arguments that will be mapped into the Component options.

setup()

Declare inputs and outputs.

Available attributes: name pathname comm options

compute(*inputs*, *outputs*)

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.geometry.geom_components.wing.components.compute_toc_wing module

Estimation of wing ToC

class fastoad.models.geometry.geom_components.wing.components.compute_toc_wing.**ComputeToCWi**

Bases: openmdao.core.explicitcomponent.ExplicitComponent

Wing ToC estimation

Store some bound methods so we can detect runtime overrides.

Parameters ****kwargs** (*dict of keyword arguments*) – Keyword arguments that will be mapped into the Component options.

setup()

Declare inputs and outputs.

Available attributes: name pathname comm options

compute (*inputs*, *outputs*)
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.geometry.geom_components.wing.components.compute_wet_area_wing module

Estimation of wing wet area

class fastoad.models.geometry.geom_components.wing.components.compute_wet_area_wing.**Compute**
Bases: openmdao.core.explicitcomponent.ExplicitComponent

Wing wet area estimation

Store some bound methods so we can detect runtime overrides.

Parameters ****kwargs** (*dict of keyword arguments*) – Keyword arguments that will be mapped into the Component options.

setup ()
Declare inputs and outputs.

Available attributes: name pathname comm options

compute (*inputs*, *outputs*)
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.geometry.geom_components.wing.components.compute_x_wing module

Estimation of wing Xs

class fastoad.models.geometry.geom_components.wing.components.compute_x_wing.**ComputeXWing** (
Bases: openmdao.core.explicitcomponent.ExplicitComponent

Wing Xs estimation

Store some bound methods so we can detect runtime overrides.

Parameters ****kwargs** (*dict of keyword arguments*) – Keyword arguments that will be mapped into the Component options.

setup()
Declare inputs and outputs.

Available attributes: name pathname comm options

compute(*inputs*, *outputs*)
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.geometry.geom_components.wing.components.compute_y_wing module

Estimation of wing Ys (sections span)

class fastoad.models.geometry.geom_components.wing.components.compute_y_wing.**ComputeYWing**(*
Bases: openmdao.core.explicitcomponent.ExplicitComponent

Wing Ys estimation

Store some bound methods so we can detect runtime overrides.

Parameters **kwargs (*dict of keyword arguments*) – Keyword arguments that will be mapped into the Component options.

setup()
Declare inputs and outputs.

Available attributes: name pathname comm options

compute(*inputs*, *outputs*)
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

Estimation of wing geometry (components)

Submodules

fastoad.models.geometry.geom_components.wing.compute_wing module

Estimation of wing geometry

class fastoad.models.geometry.geom_components.wing.compute_wing.**ComputeWingGeometry**(***kwargs*)

Bases: openmdao.core.group.Group

Wing geometry estimation

Set the solvers to nonlinear and linear block Gauss–Seidel by default.

Parameters ****kwargs** (*dict*) – dict of arguments available here and in all descendants of this Group.

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

Module contents

Estimation of wing (global)

Submodules

fastoad.models.geometry.geom_components.compute_total_area module

Estimation of total aircraft wet area

class fastoad.models.geometry.geom_components.compute_total_area.**ComputeTotalArea**(***kwargs*)

Bases: openmdao.core.explicitcomponent.ExplicitComponent

Total aircraft wet area estimation

Store some bound methods so we can detect runtime overrides.

Parameters ****kwargs** (*dict of keyword arguments*) – Keyword arguments that will be mapped into the Component options.

setup()

Declare inputs and outputs.

Available attributes: name pathname comm options

compute (*inputs*, *outputs*)
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

Estimation of geometry components

fastoad.models.geometry.profiles package

Subpackages

Submodules

fastoad.models.geometry.profiles.get_profile module

Airfoil reshape function

fastoad.models.geometry.profiles.get_profile.**get_profile** (*file_name: str = 'BACJ.txt', chord_length=1.0, thickness_ratio=None*) → *fastoad.models.geometry.profiles.profile.Profile*

Reads profile from indicated resource file and returns it after resize

Parameters

- **file_name** – name of resource
- **chord_length** – set to None to get original chord length
- **thickness_ratio** –

Returns the Profile instance

fastoad.models.geometry.profiles.profile module

Management of 2D wing profiles

class fastoad.models.geometry.profiles.profile.**Coordinates2D**(*x*, *y*)

Bases: tuple

Create new instance of Coordinates2D(x, y)

property x

Alias for field number 0

property y

Alias for field number 1

class fastoad.models.geometry.profiles.profile.**Profile**(*chord_length: float = 0.0*)

Bases: object

Class for managing 2D wing profiles :param chord_length: :param x: :param y:

chord_length: float

in meters

property thickness_ratio

thickness-to-chord ratio

set_points(*x: Sequence*, *z: Sequence*, *keep_chord_length: bool = True*, *keep_relative_thickness: bool = True*)

Sets points of the 2D profile.

Provided points are expected to be in order around the profile (clockwise or anti-clockwise).

Parameters

- **x** – in meters
- **z** – in meters
- **keep_relative_thickness** –
- **keep_chord_length** –

get_mean_line() → pandas.core.frame.DataFrame

Point set of mean line of the profile.

DataFrame keys are 'x' and 'z', given in meters.

get_relative_thickness() → pandas.core.frame.DataFrame

Point set of relative thickness of the profile.

DataFrame keys are 'x' and 'thickness' and are relative to chord_length. 'x' is from 0. to 1.

get_upper_side() → pandas.core.frame.DataFrame

Point set of upper side of the profile.

DataFrame keys are 'x' and 'z', given in meters.

get_lower_side() → pandas.core.frame.DataFrame

Point set of lower side of the profile.

DataFrame keys are 'x' and 'z', given in meters.

get_sides() → pandas.core.frame.DataFrame

Point set of the whole profile

Points are given from trailing edge to trailing edge, starting by upper side.

Module contents

Management of wing profiles

Submodules

fastoad.models.geometry.compute_aero_center module

Estimation of aerodynamic center

class fastoad.models.geometry.compute_aero_center.**ComputeAeroCenter**(***kwargs*)
Bases: openmdao.core.explicitcomponent.ExplicitComponent

Aerodynamic center estimation

Store some bound methods so we can detect runtime overrides.

Parameters ****kwargs** (*dict of keyword arguments*) – Keyword arguments that will be mapped into the Component options.

setup()
Declare inputs and outputs.

Available attributes: name pathname comm options

compute(*inputs*, *outputs*)
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.geometry.geometry module

FAST - Copyright (c) 2016 ONERA ISAE

class fastoad.models.geometry.geometry.**Geometry**(***kwargs*)
Bases: openmdao.core.group.Group

Computes geometric characteristics of the (tube-wing) aircraft:

- fuselage size can be computed from payload requirements
- wing dimensions are computed from global parameters (area, taper ratio…)
- tail planes are dimensioned from HQ requirements

Set the solvers to nonlinear and linear block Gauss–Seidel by default.

Parameters ****kwargs** (*dict*) – dict of arguments available here and in all descendants of this Group.

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

Module contents

Estimation of global geometry components

fastoad.models.handling_qualities package

Subpackages

fastoad.models.handling_qualities.tail_sizing package

Submodules

fastoad.models.handling_qualities.tail_sizing.compute_ht_area module

Estimation of horizontal tail area

class fastoad.models.handling_qualities.tail_sizing.compute_ht_area.**ComputeHTArea**(***kwargs*)
Bases: `openmdao.core.explicitcomponent.ExplicitComponent`

Computes area of horizontal tail plane

Area is computed to fulfill aircraft balance requirement at rotation speed

Store some bound methods so we can detect runtime overrides.

Parameters ****kwargs** (*dict of keyword arguments*) – Keyword arguments that will be mapped into the Component options.

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.handling_qualities.tail_sizing.compute_tail_areas module

Computation of tail areas w.r.t. HQ criteria

class fastoad.models.handling_qualities.tail_sizing.compute_tail_areas.**ComputeTailAreas**(***k*

Bases: openmdao.core.group.Group

Computes areas of vertical and horizontal tail.

- Horizontal tail area is computed so it can balance pitching moment of aircraft at rotation speed.
- Vertical tail area is computed so aircraft can have the CNbeta in cruise conditions

Set the solvers to nonlinear and linear block Gauss–Seidel by default.

Parameters ****kwargs** (*dict*) – dict of arguments available here and in all descendants of this Group.

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.models.handling_qualities.tail_sizing.compute_vt_area module

Estimation of vertical tail area

class fastoad.models.handling_qualities.tail_sizing.compute_vt_area.**ComputeVTArea**(***kwargs*)

Bases: openmdao.core.explicitcomponent.ExplicitComponent

Computes area of vertical tail plane

Area is computed to fulfill lateral stability requirement (with the most aft CG) as stated in :cite:raymer:1992.

Store some bound methods so we can detect runtime overrides.

Parameters ****kwargs** (*dict of keyword arguments*) – Keyword arguments that will be mapped into the Component options.

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

Submodules

fastoad.models.handling_qualities.compute_static_margin module

Estimation of static margin

class fastoad.models.handling_qualities.compute_static_margin.**ComputeStaticMargin**(***kwargs*)
Bases: openmdao.core.explicitcomponent.ExplicitComponent

Computation of static margin i.e. difference between CG ratio and neutral point.

Store some bound methods so we can detect runtime overrides.

Parameters ****kwargs** (*dict of keyword arguments*) – Keyword arguments that will be mapped into the Component options.

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.loops package

Subpackages

Submodules

fastoad.models.loops.compute_wing_area module

Computation of wing area

class fastoad.models.loops.compute_wing_area.**ComputeWingArea**(***kwargs*)

Bases: openmdao.core.group.Group

Computes needed wing area for:

- having enough lift at required approach speed
- being able to load enough fuel to achieve the sizing mission

Set the solvers to nonlinear and linear block Gauss–Seidel by default.

Parameters ****kwargs** (*dict*) – dict of arguments available here and in all descendants of this Group.

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.models.loops.compute_wing_position module

Computation of wing position

class fastoad.models.loops.compute_wing_position.**ComputeWingPosition**(***kwargs*)

Bases: openmdao.core.explicitcomponent.ExplicitComponent

Computes the wing position for a static margin target

Store some bound methods so we can detect runtime overrides.

Parameters ****kwargs** (*dict of keyword arguments*) – Keyword arguments that will be mapped into the Component options.

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 package

Subpackages

fastoad.models.performances.mission package

Subpackages

fastoad.models.performances.mission.mission_definition package

Subpackages

Submodules

fastoad.models.performances.mission.mission_definition.exceptions module

Exceptions for mission definition.

exception fastoad.models.performances.mission.mission_definition.exceptions.**FastMissionFile**
Bases: *fastoad.exceptions.FastError*

Raised when a mission definition is used without specifying the mission name.

fastoad.models.performances.mission.mission_definition.mission_builder module

Mission generator.

class fastoad.models.performances.mission.mission_definition.mission_builder.**MissionBuilder**

Bases: object

This class builds and computes a mission from a provided definition.

Parameters

- **mission_definition** – as 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()*

property definition
The mission definition instance.

If it is set as a file path, then the matching file will be read and interpreted.

property propulsion
Propulsion model for performance computation.

property reference_area
Reference area for aerodynamic polar.

build(*inputs: Optional[Mapping] = None*, *mission_name: Optional[str] = None*) → *fastoad.models.performances.mission.base.FlightSequence*
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)

Returns

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)

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)

Returns the reserve fuel mass in kg, or 0.0 if no reserve is defined.

get_input_variables(*mission_name=None*) → Dict[str, str]

Identify variables for a defined mission.

Parameters **mission_name** – mission name (can be omitted if only one mission is defined)

Returns a dict where key, values are names, units.

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

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: `dict`

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.

class fastoad.models.performances.mission.mission_definition.schema.**SegmentNames**(*value*)

Bases: `enum.Enum`

Class that lists available flight segments.

Enum values are linked to matching implementation with `get_segment_class()`.

ALTITUDE_CHANGE = 'altitude_change'

TRANSITION = 'transition'

CRUISE = 'cruise'

OPTIMAL_CRUISE = 'optimal_cruise'

BREGUET = 'breguet'

SPEED_CHANGE = 'speed_change'

HOLDING = 'holding'

TAXI = 'taxi'

classmethod string_values() → Set[str]

Returns the list of available segments as strings

classmethod get_segment_class(*value: Union[*fastoad.models.performances.mission.mission_definition.schema.Segr *str]*) → type

Parameters value – a SegmentNames instance or a string among possible values of SegmentNames

Returns the matching implementation class

Module contents

fastoad.models.performances.mission.openmdao package

Subpackages

Submodules

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.

Parameters

- **output_name** (*str*) – (required) name of the result variable in this component's namespace.
- **input_names** (*iterable of str*) – (required) names of the input variables for this system

- **vec_size** (*int*) – Length of the first dimension of the input and output vectors (i.e number of rows, or vector length for a 1D vector) Default is 1
- **length** (*int*) – Length of the second dimension of the input and ouptut vectors (i.e. number of columns) Default is 1 which results in input/output vectors of size (vec_size,)
- **scaling_factors** (*iterable of numeric*) – Scaling factors to apply to each input. Use [1,1,. . .] for addition, [1,-1,. . .] for subtraction Must be same length as input_names Default is None which results in a scaling factor of 1 on each input (element-wise addition)
- **val** (*float or list or tuple or ndarray*) – The initial value of the variable being added in user-defined units. Default is 1.0.
- ****kwargs** (*str*) – Any other arguments to pass to the addition system (same as add_output method for ExplicitComponent) Examples include units (str or None), desc (str)

setup()

Declare inputs and outputs.

Available attributes: name pathname comm options

fastoad.models.performances.mission.openmdao.mission module

OpenMDAO component for time-step computation of missions.

class fastoad.models.performances.mission.openmdao.mission.**Mission**(***kwargs*)

Bases: openmdao.core.group.Group

Computes a mission as specified in mission input file.

Set the solvers to nonlinear and linear block Gauss–Seidel by default.

Parameters ****kwargs** (*dict*) – dict of arguments available here and in all descendants of this Group.

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

Dataframe that lists all computed flight point data.

class fastoad.models.performances.mission.openmdao.mission.**MissionComponent**(***kwargs*)

Bases: openmdao.core.explicitcomponent.ExplicitComponent

Computes a mission as specified in mission input file

Options:

- propulsion_id: (mandatory) the identifier of the propulsion wrapper.
- **out_file: if provided, a csv file will be written at provided path with all computed** flight points.
- mission_wrapper: the MissionWrapper instance that defines the mission.
- **use_initializer_iteration: 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. Set this option to False if you do expect this model to be computed only once.
- **is_sizing: if True, TOW will be considered equal to MTOW and mission payload will be** considered equal to design payload.

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**(**args*, ***kwargs*)

Bases: *fastoad.models.performances.mission.mission_definition.mission_builder.MissionBuilder*

Wrapper around *MissionBuilder* for using with OpenMDAO.

This class builds and computes a mission from a provided definition.

Parameters

- **mission_definition** – as 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()`

setup(*component: openmdao.core.explicitcomponent.ExplicitComponent*, *mission_name: Optional[str] = None*)
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.
- **mission_name** – mission name (can be omitted if only one mission is defined)

compute(*inputs: openmdao.vectors.vector.Vector*, *outputs: openmdao.vectors.vector.Vector*, *start_flight_point:* fastoad.model_base.flight_point.FlightPoint) → 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

- **inputs** – the input vector of the OpenMDAO component
- **outputs** – the output vector of the OpenMDAO component
- **start_flight_point** – the starting flight point just after takeoff

Returns a pandas DataFrame where columns 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()`.

Module contents

fastoad.models.performances.mission.segments package

Subpackages

Submodules

fastoad.models.performances.mission.segments.altitude_change module

Classes for climb/descent segments.

class fastoad.models.performances.mission.segments.altitude_change.**AltitudeChangeSegment**(*ta*
fa
to
pr
si
fa
to
po
la
fa
to
re
en
en
flo
tin
flo
=
2.
en
gi
fa
to
=
<
gi
ne
Se
tin
2>
al
ti-
tu
tu
pl
=
(-
50
40
m
tu
pl
=
(0
5.
na
st
=
'',
in
te
ru
bo
=
Tr
th
flo
=
1.
m

Bases: *fastoad.models.performances.mission.segments.base.ManualThrustSegment*

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: 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.

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 columns names match fields of *FlightPoint()*

fastoad.models.performances.mission.segments.base module

Base classes for simulating flight segments.

class fastoad.models.performances.mission.segments.base.**FlightSegment**(*target: fastoad.model_base.flight_point.F propulsion: fastoad.model_base.propulsion.IP polar: fastoad.models.performances.mis reference_area: float*, *time_step: float = 0.2*, *engine_setting: fastoad.constants.EngineSetting = <EngineSetting.CLIMB: 2>*, *altitude_bounds: tuple = (-500.0, 40000.0)*, *mach_bounds: tuple = (0.0, 5.0)*, *name: str = ''*, *interrupt_if_getting_further_from_ bool = True*)

Bases: `fastoad.models.performances.mission.base.IFlightPart`

Base class for flight path segment.

As a dataclass, attributes can be set at instantiation.

target: *fastoad.model_base.flight_point.FlightPoint*
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.

propulsion: *fastoad.model_base.propulsion.IPropulsion*
A IPropulsion instance that will be called at each time step.

polar: *fastoad.models.performances.mission.polar.Polar*
The Polar instance that will provide drag data.

reference_area: float
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).

engine_setting: *fastoad.constants.EngineSetting* = 2
The EngineSetting value associated to the segment. Can be used in the propulsion model.

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 = (0.0, 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.

name: str = ''
The name of the current flight sequence.

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).

CONSTANT_VALUE = 'constant'
Using this value will tell to keep the associated parameter constant.

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.

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 columns 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

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

class fastoad.models.performances.mission.segments.base.**ManualThrustSegment**(*target: fastoad.model_base.fligh propulsion: fastoad.model_base.prop polar: fastoad.models.performa reference_area: float*, *time_step: float = 0.2*, *engine_setting: fastoad.constants.Engine = <EngineSetting.CLIMB: 2>*, *altitude_bounds: tuple = (-500.0, 40000.0)*, *mach_bounds: tuple = (0.0, 5.0)*, *name: str = ''*, *interrupt_if_getting_furthe bool = True*, *thrust_rate: float = 1.0*)

Bases: *fastoad.models.performances.mission.segments.base.FlightSegment*, 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

class fastoad.models.performances.mission.segments.base.**RegulatedThrustSegment**(**args*, ***kwargs*)

Bases: *fastoad.models.performances.mission.segments.base.FlightSegment*, 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).

class fastoad.models.performances.mission.segments.base.**FixedDurationSegment**(*target: fastoad.model_base.flig propulsion: fastoad.model_base.pro polar: fastoad.models.perform reference_area: float, time_step: float = 60.0, engine_setting: fastoad.constants.Engi = <EngineSetting.CLIMB: 2>, altitude_bounds: tuple = (-500.0, 40000.0), mach_bounds: tuple = (0.0, 5.0), name: str = '', interrupt_if_getting_furt bool = True*)

Bases: *fastoad.models.performances.mission.segments.base.FlightSegment*, abc.ABC

Class for computing phases where duration is fixed.

Target duration is provide as target.time. When using *compute_from()*, if start.time is not 0, end time will be start.time + target.time.

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_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.

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 columns names match fields of *FlightPoint()*

fastoad.models.performances.mission.segments.cruise module

Classes for simulating cruise segments.

class fastoad.models.performances.mission.segments.cruise.**CruiseSegment**(*target: fastoad.model_base.flight_poi propulsion: fastoad.model_base.propulsion polar: fastoad.models.performances.r reference_area: float, time_step: float = 60.0, engine_setting: fastoad.constants.EngineSettin = <EngineSetting.CLIMB: 2>, altitude_bounds: tuple = (-500.0, 40000.0), mach_bounds: tuple = (0.0, 5.0), name: str = '', interrupt_if_getting_further_fro bool = True*)

Bases: *fastoad.models.performances.mission.segments.base.RegulatedThrustSegment*

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.

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.

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 columns names match fields of *FlightPoint()*

target: *fastoad.model_base.flight_point.FlightPoint*

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.

propulsion: *fastoad.model_base.propulsion.IPropulsion*

A IPropulsion instance that will be called at each time step.

polar: *fastoad.models.performances.mission.polar.Polar*

The Polar instance that will provide drag data.

reference_area: float

The reference area, in m**2.

class fastoad.models.performances.mission.segments.cruise.**OptimalCruiseSegment**(*target: fastoad.model_base propulsion: fastoad.model_base polar: fastoad.models.perf reference_area: float*, *time_step: float = 60.0*, *engine_setting: fastoad.constants.En = <EngineSetting.CLIMB: 2>*, *altitude_bounds: tuple = (-500.0, 40000.0)*, *mach_bounds: tuple = (0.0, 5.0)*, *name: str = ''*, *interrupt_if_getting_f bool = True*)

Bases: *fastoad.models.performances.mission.segments.cruise.CruiseSegment*

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:* 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.

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 columns names match fields of *FlightPoint()*

target: ***fastoad.model_base.flight_point.FlightPoint***
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.

propulsion: ***fastoad.model_base.propulsion.IPropulsion***
A IPropulsion instance that will be called at each time step.

polar: ***fastoad.models.performances.mission.polar.Polar***
The Polar instance that will provide drag data.

reference_area: **float**
The reference area, in m**2.

class fastoad.models.performances.mission.segments.cruise.**ClimbAndCruiseSegment**(*target: fastoad.model_bas propulsion: fastoad.model_bas polar: fastoad.models.per reference_area: float*, *time_step: float = 60.0*, *engine_setting: fastoad.constants.E = <EngineSetting.CLIMB: 2>*, *altitude_bounds: tuple = (-500.0, 40000.0)*, *mach_bounds: tuple = (0.0, 5.0)*, *name: str = ''*, *interrupt_if_getting_ bool = True*, *climb_segment: Optional[fastoad.m = None*,

Bases: *fastoad.models.performances.mission.segments.cruise.CruiseSegment*

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.altitude_change.AltitudeC***

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:* 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.

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 columns names match fields of *FlightPoint()*

class fastoad.models.performances.mission.segments.cruise.**BreguetCruiseSegment**(*target: fastoad.model_base… propulsion: fastoad.model_base… polar: fastoad.models.perf… reference_area: float = 1.0*, *time_step: float = 60.0*, *engine_setting: fastoad.constants.E… = <EngineSetting.CLIMB: 2>*, *altitude_bounds: tuple = (-500.0*, *40000.0)*, *mach_bounds: tuple = (0.0*, *5.0)*, *name: str = ''*, *interrupt_if_getting_f… bool = True*, *use_max_lift_dr… bool =*

Bases: *fastoad.models.performances.mission.segments.cruise.CruiseSegment*

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.

climb_and_descent_distance: float = 0.0

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.

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 columns names match fields of *FlightPoint()*

fastoad.models.performances.mission.segments.hold module

Class for simulating hold segment.

class fastoad.models.performances.mission.segments.hold.**HoldSegment**(**args*, ***kwargs*)

Bases: *fastoad.models.performances.mission.segments.base.RegulatedThrustSegment*, *fastoad.models.performances.mission.segments.base.FixedDurationSegment*

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.

target: *fastoad.model_base.flight_point.FlightPoint*
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.

propulsion: *fastoad.model_base.propulsion.IPropulsion*
A IPropulsion instance that will be called at each time step.

polar: *fastoad.models.performances.mission.polar.Polar*
The Polar instance that will provide drag data.

reference_area: float
The reference area, in m**2.

fastoad.models.performances.mission.segments.speed_change module

Classes for acceleration/deceleration segments.

class fastoad.models.performances.mission.segments.speed_change.**SpeedChangeSegment**(*target: fastoad.model propulsion: fastoad.model polar: fastoad.model reference_area: float, time_step: float = 0.2, engine_settin fastoad.consta = <EngineSetting.CLIMB 2>, altitude_bound tuple = (-500.0, 40000.0), mach_boun tuple = (0.0, 5.0), name: str = '', interrupt_if_get bool = True, thrust_rate float = 1.0*)

Bases: *fastoad.models.performances.mission.segments.base.ManualThrustSegment*

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.

target: ***fastoad.model_base.flight_point.FlightPoint***
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.

propulsion: ***fastoad.model_base.propulsion.IPropulsion***
A IPropulsion instance that will be called at each time step.

polar: ***fastoad.models.performances.mission.polar.Polar***
The Polar instance that will provide drag data.

reference_area: **float**
The reference area, in m**2.

fastoad.models.performances.mission.segments.taxi module

Classes for Taxi sequences.

class fastoad.models.performances.mission.segments.taxi.**TaxiSegment**(*target: fastoad.model_base.flight_point.Flig propulsion: fastoad.model_base.propulsion.IPro polar: Optional[fastoad.models.performanc = None, reference_area: float = 1.0, time_step: float = 60.0, engine_setting: fastoad.constants.EngineSetting = <EngineSetting.CLIMB: 2>, altitude_bounds: tuple = (-500.0, 40000.0), mach_bounds: tuple = (0.0, 5.0), name: str = '', interrupt_if_getting_further_from_targ bool = True, thrust_rate: float = 1.0*)

Bases: fastoad.models.performances.mission.segments.base. ManualThrustSegment, fastoad.models.performances.mission.segments.base. FixedDurationSegment

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).

fastoad.models.performances.mission.segments.transition module

Class for very simple transition in some flight phases.

class fastoad.models.performances.mission.segments.transition.**DummyTransitionSegment**(*target: fastoad.mo propulsion: Optional[fa = None, polar: Optional[fa = None, reference_ar float = 1.0, time_ste float = 0.2, engine_set fastoad.con = <EngineSetting.CLI 2>, altitude_bo tuple = (-500.0, 40000.0 mach_bo tuple = (0.0, 5.0), name: str = '', interrupt_if_ bool =*

Bases: `fastoad.models.performances.mission.segments.base.FlightSegment`

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 and its mass is the start mass multiplied by `mass_ratio`. Other parameters are equal to those provided in `target`.

If `reserve_mass_ratio` is non-zero, a third flight point, 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.

propulsion: *fastoad.model_base.propulsion.IPropulsion* = None
Unused

reference_area: float = 1.0
Unused

polar: *fastoad.models.performances.mission.polar.Polar* = None
Unused

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.

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 columns names match fields of `FlightPoint()`

Module contents

Classes for simulating flight segments.

Submodules

fastoad.models.performances.mission.base module

Base classes for mission computation.

class fastoad.models.performances.mission.base.**IFlightPart**
Bases: `abc.ABC`

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 columns names match fields of *FlightPoint*

class fastoad.models.performances.mission.base.**FlightSequence**
Bases: *fastoad.models.performances.mission.base.IFlightPart*

Defines and computes a flight sequence.

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 columns names match fields of *FlightPoint*

property flight_sequence
List of IFlightPart instances that should be run sequentially.

fastoad.models.performances.mission.exceptions module

Exceptions for mission package.

exception fastoad.models.performances.mission.exceptions.**FastFlightSegmentUnexpectedKeyword**
Bases: *fastoad.exceptions.FastUnexpectedKeywordArgument*

Raised when a segment is instantiated with an incorrect keyword argument.

exception fastoad.models.performances.mission.exceptions.**FastFlightPointUnexpectedKeywordA**
Bases: *fastoad.exceptions.FastUnexpectedKeywordArgument*

Raised when a FlightPoint is instantiated with an incorrect keyword argument.

exception fastoad.models.performances.mission.exceptions.**FastFlightSegmentIncompleteFlight**
Bases: *fastoad.exceptions.FastError*

Raised when a segment computation encounters a FlightPoint instance without needed parameters.

fastoad.models.performances.mission.polar module

Aerodynamic polar data.

class fastoad.models.performances.mission.polar.**Polar**(*cl: numpy.ndarray*, *cd: numpy.ndarray*)
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.

Once defined, for any CL value, CD can be obtained using *cd()*.

Parameters

- **cl** – a N-elements array with CL values
- **cd** – a N-elements array with CD values that match CL

property definition_cl
The vector that has been used for defining lift coefficient.

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

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.**SimpleRoute**(*climb_phases: List[*fastoad.models.performances.mission.b *cruise_segment:* fas-toad.models.performances.mission.segments *descent_phases: List[*fastoad.models.performances.mission.b

Bases: *fastoad.models.performances.mission.base.FlightSequence*

Computes a simple route.

The computed route will be 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*]
Any number of flight phases that will occur before cruise.

cruise_segment: *fastoad.models.performances.mission.segments.cruise.CruiseSegment*
The cruise phase.

descent_phases: List[*fastoad.models.performances.mission.base.FlightSequence*]
Any number of flight phases that will occur after cruise.

property cruise_distance
Ground distance to be covered during cruise, as set in target of *cruise_segment*.

property cruise_speed
Type (among *true_airspeed*, *equivalent_airspeed* and *mach*) and value of cruise speed.

class fastoad.models.performances.mission.routes.**RangedRoute**(*climb_phases: List[fastoad.models.performances.mission.b cruise_segment: fastoad.models.performances.mission.segments descent_phases: List[fastoad.models.performances.mission.b flight_distance: float, distance_accuracy: float = 500.0*)

Bases: *fastoad.models.performances.mission.routes.SimpleRoute*

Computes a route so that it covers the specified ground distance.

flight_distance: float
Target ground distance for whole route

distance_accuracy: float = 500.0
Accuracy on actual total ground distance for the solver. In meters

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 columns 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.

fastoad.models.propulsion package

Subpackages

fastoad.models.propulsion.fuel_propulsion package

Subpackages

fastoad.models.propulsion.fuel_propulsion.rubber_engine package

Subpackages

Submodules

fastoad.models.propulsion.fuel_propulsion.rubber_engine.constants module

Constants for rubber engine analytical models

fastoad.models.propulsion.fuel_propulsion.rubber_engine.exceptions module

Exceptions for rubber_engine package.

exception `fastoad.models.propulsion.fuel_propulsion.rubber_engine.exceptions.`**FastRubberEng**

Bases: `Exception`

Raised when provided parameter combination is incorrect.

fastoad.models.propulsion.fuel_propulsion.rubber_engine.openmdao module

OpenMDAO wrapping of RubberEngine.

class fastoad.models.propulsion.fuel_propulsion.rubber_engine.openmdao.**OMRubberEngineWrappe**

Bases: *fastoad.model_base.propulsion.IOMPropulsionWrapper*

Wrapper class of for rubber engine model.

It is made to allow a direct call to *RubberEngine* in an OpenMDAO component.

Example of usage of this class:

```
import openmdao.api as om

class MyComponent(om.ExplicitComponent):
    def initialize():
        self._engine_wrapper = OMRubberEngineWrapper()

    def setup():
        # Adds OpenMDAO variables that define the engine
        self._engine_wrapper.setup(self)

        # Do the normal setup
        self.add_input("my_input")
        [finish the setup...]

    def compute(self, inputs, outputs, discrete_inputs=None, discrete_
↪outputs=None):
        [do something]

        # Get the engine instance, with parameters defined from OpenMDAO inputs
        engine = self._engine_wrapper.get_model(inputs)

        # Run the engine model. This is a pure Python call. You have to define
        # its inputs before, and to use its outputs according to your needs
        sfc, thrust_rate, thrust = engine.compute_flight_points(
            mach,
            altitude,
            engine_setting,
            use_thrust_rate,
            thrust_rate,
            thrust
            )

        [do something else]

)
```

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** –

static get_model(*inputs*) → *fastoad.model_base.propulsion.IPropulsion*

Parameters **inputs** – input parameters that define the engine

Returns a *RubberEngine* instance

class fastoad.models.propulsion.fuel_propulsion.rubber_engine.openmdao.**OMRubberEngineCompon**

Bases: *fastoad.model_base.propulsion.BaseOMPropulsionComponent*

Parametric engine model as OpenMDAO component

See *RubberEngine* for more information.

Store some bound methods so we can detect runtime overrides.

Parameters `**kwargs` (*dict of keyword arguments*) – Keyword arguments that will be mapped into the Component options.

setup()

Declare inputs and outputs.

Available attributes: name pathname comm options

static get_wrapper() → *fastoad.models.propulsion.fuel_propulsion.rubber_engine.openmdao.OMRubberEngineWrappe*

This method defines the used *IOMPropulsionWrapper* instance.

Returns an instance of OpenMDAO wrapper for propulsion model

fastoad.models.propulsion.fuel_propulsion.rubber_engine.rubber_engine module

Parametric turbofan engine.

class fastoad.models.propulsion.fuel_propulsion.rubber_engine.rubber_engine.**RubberEngine**(*by*
flo
ov
al
flo
tu
bi
flo
m
flo
m
i-
m
flo
de
sig
flo
de
flo
=
-

50
de
flo
=
-

10

Bases: *fastoad.model_base.propulsion.AbstractFuelPropulsion*

Parametric turbofan engine.

It computes engine characteristics using analytical model from following sources:

Parameters

- **bypass_ratio** –
- **overall_pressure_ratio** –
- **turbine_inlet_temperature** – (unit=K) also noted T4
- **mto_thrust** – (unit=N) Maximum TakeOff thrust, i.e. maximum thrust on ground at speed 0, also noted F0
- **maximum_mach** –
- **design_altitude** – (unit=m)
- **delta_t4_climb** – (unit=K) difference between T4 during climb and design T4
- **delta_t4_cruise** – (unit=K) difference between T4 during cruise and design T4

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)

compute_flight_points_from_dt4 (*mach: Union[*float*, Sequence], altitude: Union[*float*, Sequence], delta_t4: Union[*float*, Sequence], thrust_is_regulated: Optional[Union[*bool*, Sequence]] = None, thrust_rate: Optional[Union[*float*, Sequence]] = None, thrust: Optional[Union[*float*, Sequence]] = None*) → Tuple[Union[float, Sequence], Union[float, Sequence], Union[float, Sequence]]

Same as `compute_flight_points()` except that delta_t4 is used directly instead of specifying flight engine_setting.

Parameters

- **mach** – Mach number
- **altitude** – (unit=m) altitude w.r.t. to sea level
- **delta_t4** – (unit=K) difference between operational and design values of turbine inlet temperature in K
- **thrust_is_regulated** – tells if thrust_rate or thrust should be used (works element-wise)
- **thrust_rate** – thrust rate (unit=none)
- **thrust** – required thrust (unit=N)

Returns SFC (in kg/s/N), thrust rate, thrust (in N)

sfc_at_max_thrust (*atmosphere: fastoad.model_base.atmosphere.Atmosphere, mach: Union[float, Sequence[float]]*) → numpy.ndarray

Computation of Specific Fuel Consumption at maximum thrust.

Uses model described in [[Rou05]], p.41.

Parameters

- **atmosphere** – Atmosphere instance at intended altitude
- **mach** – Mach number(s)

Returns SFC (in kg/s/N)

sfc_ratio (*altitude: Union[float, Sequence[float]], thrust_rate: Union[float, Sequence[float]], mach: Union[float, Sequence[float]] = 0.8*) → numpy.ndarray

Computation of ratio $\frac{SFC(F)}{SFC(Fmax)}$, given altitude and thrust_rate $\frac{F}{Fmax}$.

Uses a patched version of model described in [[Rou02]], p.85.

Warning: this model is very limited

Parameters

- **altitude** –
- **thrust_rate** –
- **mach** – only used for logger checks as model is made for Mach~0.8

Returns SFC ratio

max_thrust (*atmosphere: fastoad.model_base.atmosphere.Atmosphere, mach: Union[float, Sequence[float]], delta_t4: Union[float, Sequence[float]]*) → numpy.ndarray

Computation of maximum thrust.

Uses model described in [[Rou05]], p.57-58

Parameters

- **atmosphere** – Atmosphere instance at intended altitude (should be <=20km)
- **mach** – Mach number(s) (should be between 0.05 and 1.0)
- **delta_t4** – (unit=K) difference between operational and design values of turbine inlet temperature in K

Returns maximum thrust (in N)

installed_weight() → float
Computes weight of installed engine, depending on MTO thrust (F0).

Uses model described in [[Rou05]], p.74

Returns installed weight (in kg)

length() → float
Computes engine length from MTO thrust and maximum Mach.

Model from [[Ray99]], p.74

Returns engine length (in m)

nacelle_diameter() → float
Computes nacelle diameter from MTO thrust and bypass ratio.

Model of engine diameter from [[Ray99]], p.235. Nacelle diameter is considered 10% greater ([[kro01]])

Returns nacelle diameter (in m)

Module contents

Provides a parametric model for turbofan:

- as a pure Python
- as OpenMDAO modules

Module contents

Module contents

Package for propulsion modules

fastoad.models.weight package

Subpackages

fastoad.models.weight.cg package

Subpackages

fastoad.models.weight.cg.cg_components package

Submodules

fastoad.models.weight.cg.cg_components.compute_cg_control_surfaces module

Estimation of control surfaces center of gravity

class fastoad.models.weight.cg.cg_components.compute_cg_control_surfaces.**ComputeControlSur**
Bases: openmdao.core.explicitcomponent.ExplicitComponent

Control surfaces center of gravity estimation

Store some bound methods so we can detect runtime overrides.

Parameters ****kwargs** (*dict of keyword arguments*) – Keyword arguments that will be mapped into the Component options.

setup()

Declare inputs and outputs.

Available attributes: name pathname comm options

compute(*inputs*, *outputs*)

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.weight.cg.cg_components.compute_cg_loadcase1 module

Estimation of center of gravity for load case 1

class fastoad.models.weight.cg.cg_components.compute_cg_loadcase1.**ComputeCGLoadCase1**(***kwarg*

Bases: openmdao.core.explicitcomponent.ExplicitComponent

Center of gravity estimation for load case 1

Store some bound methods so we can detect runtime overrides.

Parameters ****kwargs** (*dict of keyword arguments*) – Keyword arguments that will be mapped into the Component options.

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.weight.cg.cg_components.compute_cg_loadcase2 module

Estimation of center of gravity for load case 2

class fastoad.models.weight.cg.cg_components.compute_cg_loadcase2.**ComputeCGLoadCase2**(***kwarg*

Bases: openmdao.core.explicitcomponent.ExplicitComponent

Center of gravity estimation for load case 2

Store some bound methods so we can detect runtime overrides.

Parameters ****kwargs** (*dict of keyword arguments*) – Keyword arguments that will be mapped into the Component options.

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.weight.cg.cg_components.compute_cg_loadcase3 module

Estimation of center of gravity for load case 3

class fastoad.models.weight.cg.cg_components.compute_cg_loadcase3.**ComputeCGLoadCase3**(***kwarg*

Bases: openmdao.core.explicitcomponent.ExplicitComponent

Center of gravity estimation for load case 3

Store some bound methods so we can detect runtime overrides.

Parameters ****kwargs** (*dict of keyword arguments*) – Keyword arguments that will be mapped into the Component options.

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.weight.cg.cg_components.compute_cg_loadcase4 module

Estimation of center of gravity for load case 4

class fastoad.models.weight.cg.cg_components.compute_cg_loadcase4.**ComputeCGLoadCase4**(***kwarg*

Bases: openmdao.core.explicitcomponent.ExplicitComponent

Center of gravity estimation for load case 4

Store some bound methods so we can detect runtime overrides.

Parameters ****kwargs** (*dict of keyword arguments*) – Keyword arguments that will be mapped into the Component options.

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.weight.cg.cg_components.compute_cg_others module

Estimation of other components center of gravities

class fastoad.models.weight.cg.cg_components.compute_cg_others.**ComputeOthersCG**(***kwargs*)

Bases: openmdao.core.explicitcomponent.ExplicitComponent

Other components center of gravities estimation

Store some bound methods so we can detect runtime overrides.

Parameters ****kwargs** (*dict of keyword arguments*) – Keyword arguments that will be mapped into the Component options.

setup()

Declare inputs and outputs.

Available attributes: name pathname comm options

compute(*inputs*, *outputs*)

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.weight.cg.cg_components.compute_cg_ratio_aft module

Estimation of center of gravity ratio with aft

class fastoad.models.weight.cg.cg_components.compute_cg_ratio_aft.**ComputeCGRatioAft**(***kwargs*)

Bases: openmdao.core.group.Group

Set the solvers to nonlinear and linear block Gauss–Seidel by default.

Parameters ****kwargs** (*dict*) – dict of arguments available here and in all descendants of this Group.

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.weight.cg.cg_components.compute_cg_ratio_aft.**ComputeCG**(***kwargs*)

Bases: openmdao.core.explicitcomponent.ExplicitComponent

Store some bound methods so we can detect runtime overrides.

Parameters ****kwargs** (*dict of keyword arguments*) – Keyword arguments that will be mapped into the Component options.

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.weight.cg.cg_components.compute_cg_ratio_aft.**CGRatio**(***kwargs*)
Bases: openmdao.core.explicitcomponent.ExplicitComponent

Store some bound methods so we can detect runtime overrides.

Parameters ****kwargs** (*dict of keyword arguments*) – Keyword arguments that will be mapped into the Component options.

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.weight.cg.cg_components.compute_cg_tanks module

Estimation of tanks center of gravity

class fastoad.models.weight.cg.cg_components.compute_cg_tanks.**ComputeTanksCG**(***kwargs*)
Bases: openmdao.core.explicitcomponent.ExplicitComponent

Tanks center of gravity estimation

Store some bound methods so we can detect runtime overrides.

Parameters ****kwargs** (*dict of keyword arguments*) – Keyword arguments that will be mapped into the Component options.

initialize()
Perform any one-time initialization run at instantiation.

setup()
Declare inputs and outputs.

Available attributes: name pathname comm options

compute(*inputs*, *outputs*)
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.weight.cg.cg_components.compute_cg_wing module

Estimation of wing center of gravity

class fastoad.models.weight.cg.cg_components.compute_cg_wing.**ComputeWingCG**(***kwargs*)

Bases: openmdao.core.explicitcomponent.ExplicitComponent

Wing center of gravity estimation

Store some bound methods so we can detect runtime overrides.

Parameters ****kwargs** (*dict of keyword arguments*) – Keyword arguments that will be mapped into the Component options.

setup()

Declare inputs and outputs.

Available attributes: name pathname comm options

compute(*inputs*, *outputs*)

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.weight.cg.cg_components.compute_global_cg module

Estimation of global center of gravity

class fastoad.models.weight.cg.cg_components.compute_global_cg.**ComputeGlobalCG**(***kwargs*)

Bases: openmdao.core.group.Group

Global center of gravity estimation

Set the solvers to nonlinear and linear block Gauss–Seidel by default.

Parameters ****kwargs** (*dict*) – dict of arguments available here and in all descendants of this Group.

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.models.weight.cg.cg_components.compute_ht_cg module

Estimation of horizontal tail center of gravity

class fastoad.models.weight.cg.cg_components.compute_ht_cg.**ComputeHTcg**(***kwargs*)

Bases: openmdao.core.explicitcomponent.ExplicitComponent

Horizontal tail center of gravity estimation

Store some bound methods so we can detect runtime overrides.

Parameters ****kwargs** (*dict of keyword arguments*) – Keyword arguments that will be mapped into the Component options.

setup()

Declare inputs and outputs.

Available attributes: name pathname comm options

compute(*inputs*, *outputs*)

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.weight.cg.cg_components.compute_max_cg_ratio module

Estimation of maximum center of gravity ratio

class fastoad.models.weight.cg.cg_components.compute_max_cg_ratio.**ComputeMaxCGratio**(***kwargs*

Bases: openmdao.core.explicitcomponent.ExplicitComponent

Maximum center of gravity ratio estimation

Store some bound methods so we can detect runtime overrides.

Parameters ****kwargs** (*dict of keyword arguments*) – Keyword arguments that will be mapped into the Component options.

setup()

Declare inputs and outputs.

Available attributes: name pathname comm options

compute(*inputs*, *outputs*)

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.weight.cg.cg_components.compute_vt_cg module

Estimation of vertical tail center of gravity

class fastoad.models.weight.cg.cg_components.compute_vt_cg.**ComputeVTcg**(***kwargs*)
Bases: openmdao.core.explicitcomponent.ExplicitComponent

Vertical tail center of gravity estimation

Store some bound methods so we can detect runtime overrides.

Parameters ****kwargs** (*dict of keyword arguments*) – Keyword arguments that will be mapped into the Component options.

setup()
Declare inputs and outputs.

Available attributes: name pathname comm options

compute(*inputs*, *outputs*)
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.weight.cg.cg_components.update_mlg module

Estimation of main landing gear center of gravity

class fastoad.models.weight.cg.cg_components.update_mlg.**UpdateMLG**(***kwargs*)
Bases: openmdao.core.explicitcomponent.ExplicitComponent

Main landing gear center of gravity estimation

Store some bound methods so we can detect runtime overrides.

Parameters ****kwargs** (*dict of keyword arguments*) – Keyword arguments that will be mapped into the Component options.

setup()
Declare inputs and outputs.

Available attributes: name pathname comm options

compute(*inputs*, *outputs*)
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

Estimation of centers of gravity

Submodules

fastoad.models.weight.cg.cg module

FAST - Copyright (c) 2016 ONERA ISAE

class fastoad.models.weight.cg.cg.**CG**(***kwargs*)

Bases: openmdao.core.group.Group

Model that computes the global center of gravity

Set the solvers to nonlinear and linear block Gauss–Seidel by default.

Parameters ****kwargs** (*dict*) – dict of arguments available here and in all descendants of this Group.

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.weight.cg.cg.**ComputeAircraftCG**(***kwargs*)

Bases: openmdao.core.explicitcomponent.ExplicitComponent

Compute position of aircraft CG from CG ratio

Store some bound methods so we can detect runtime overrides.

Parameters ****kwargs** (*dict of keyword arguments*) – Keyword arguments that will be mapped into the Component options.

setup()

Declare inputs and outputs.

Available attributes: name pathname comm options

compute(*inputs*, *outputs*)

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.weight.mass_breakdown package

Subpackages

fastoad.models.weight.mass_breakdown.a_airframe package

Submodules

fastoad.models.weight.mass_breakdown.a_airframe.a1_wing_weight module

Estimation of wing weight

class fastoad.models.weight.mass_breakdown.a_airframe.a1_wing_weight.**WingWeight**(***kwargs*)

Bases: openmdao.core.explicitcomponent.ExplicitComponent

Wing weight estimation

This is done by summing following estimations:

- mass from sizing to flexion forces
- mass from sizing to shear forces
- mass of ribs
- mass of reinforcements
- mass of secondary parts

Based on [[DCAC14]], mass contribution A1

Store some bound methods so we can detect runtime overrides.

Parameters ****kwargs** (*dict of keyword arguments*) – Keyword arguments that will be mapped into the Component options.

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.weight.mass_breakdown.a_airframe.a2_fuselage_weight module

Estimation of fuselage weight

class fastoad.models.weight.mass_breakdown.a_airframe.a2_fuselage_weight.**FuselageWeight**(***k*

Bases: openmdao.core.explicitcomponent.ExplicitComponent

Fuselage weight estimation

Based on a statistical analysis. See [[DCAC14]], mass contribution A2

Store some bound methods so we can detect runtime overrides.

Parameters ****kwargs** (*dict of keyword arguments*) – Keyword arguments that will be mapped into the Component options.

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.weight.mass_breakdown.a_airframe.a3_empennage_weight module

Estimation of empennage weight

class fastoad.models.weight.mass_breakdown.a_airframe.a3_empennage_weight.**EmpennageWeight**(

Bases: openmdao.core.explicitcomponent.ExplicitComponent

Weight estimation for tail planes

Based on formulas in [[DCAC14]], mass contribution A3

Store some bound methods so we can detect runtime overrides.

Parameters ****kwargs** (*dict of keyword arguments*) – Keyword arguments that will be mapped into the Component options.

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.weight.mass_breakdown.a_airframe.a4_flight_control_weight module

Estimation of flight controls weight

class fastoad.models.weight.mass_breakdown.a_airframe.a4_flight_control_weight.**FlightContro**
Bases: openmdao.core.explicitcomponent.ExplicitComponent

Flight controls weight estimation

Based on formulas in [[DCAC14]], mass contribution A4

Store some bound methods so we can detect runtime overrides.

Parameters ****kwargs** (*dict of keyword arguments*) – Keyword arguments that will be mapped into the Component options.

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.weight.mass_breakdown.a_airframe.a5_landing_gear_weight module

Estimation of landing gear weight

class fastoad.models.weight.mass_breakdown.a_airframe.a5_landing_gear_weight.**LandingGearWe**
Bases: openmdao.core.explicitcomponent.ExplicitComponent

Weight estimation for landing gears

Based on formulas in [[DCAC14]], mass contribution A5

Store some bound methods so we can detect runtime overrides.

Parameters ****kwargs** (*dict of keyword arguments*) – Keyword arguments that will be mapped into the Component options.

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.weight.mass_breakdown.a_airframe.a6_pylons_weight module

Estimation of pylons weight

class fastoad.models.weight.mass_breakdown.a_airframe.a6_pylons_weight.**PylonsWeight**(***kwargs*)
Bases: openmdao.core.explicitcomponent.ExplicitComponent

Weight estimation for pylons

Based on formula in [[DCAC14]], mass contribution A6

Store some bound methods so we can detect runtime overrides.

Parameters ****kwargs** (*dict of keyword arguments*) – Keyword arguments that will be mapped into the Component options.

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.weight.mass_breakdown.a_airframe.a7_paint_weight module

Estimation of paint weight

class fastoad.models.weight.mass_breakdown.a_airframe.a7_paint_weight.**PaintWeight**(***kwargs*)

Bases: openmdao.core.explicitcomponent.ExplicitComponent

Weight estimation for paint

Based on formula in [[DCAC14]], mass contribution A7

Store some bound methods so we can detect runtime overrides.

Parameters ****kwargs** (*dict of keyword arguments*) – Keyword arguments that will be mapped into the Component options.

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

Estimation of airframe weight

fastoad.models.weight.mass_breakdown.b_propulsion package

Submodules

fastoad.models.weight.mass_breakdown.b_propulsion.b1_engine_weight module

Estimation of engine weight

class fastoad.models.weight.mass_breakdown.b_propulsion.b1_engine_weight.**EngineWeight**(***kwa

Bases: openmdao.core.explicitcomponent.ExplicitComponent

Engine weight estimation

Uses model described in [[Rou05]], p.74

Store some bound methods so we can detect runtime overrides.

Parameters ****kwargs** (*dict of keyword arguments*) – Keyword arguments that will be mapped into the Component options.

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.weight.mass_breakdown.b_propulsion.b2_fuel_lines_weight module

Estimation of fuel lines weight

class fastoad.models.weight.mass_breakdown.b_propulsion.b2_fuel_lines_weight.**FuelLinesWeig**
Bases: openmdao.core.explicitcomponent.ExplicitComponent

Weight estimation for fuel lines

Based on formula in [[DCAC14]], mass contribution B2

Store some bound methods so we can detect runtime overrides.

Parameters ****kwargs** (*dict of keyword arguments*) – Keyword arguments that will be mapped into the Component options.

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.weight.mass_breakdown.b_propulsion.b3_unconsumables_weight module

Estimation of fuel lines weight

class fastoad.models.weight.mass_breakdown.b_propulsion.b3_unconsumables_weight.**Unconsumab**

Bases: openmdao.core.explicitcomponent.ExplicitComponent

Weight estimation for oil and unusable fuel

Based on formula in [[DCAC14]], mass contribution B3

Store some bound methods so we can detect runtime overrides.

Parameters ****kwargs** (*dict of keyword arguments*) – Keyword arguments that will be mapped into the Component options.

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

Estimation of propulsion weight

fastoad.models.weight.mass_breakdown.c_systems package

Submodules

fastoad.models.weight.mass_breakdown.c_systems.c1_power_systems_weight module

Estimation of power systems weight

class fastoad.models.weight.mass_breakdown.c_systems.c1_power_systems_weight.**PowerSystemsWe**

Bases: openmdao.core.explicitcomponent.ExplicitComponent

Weight estimation for power systems (generation and distribution)

This includes:

- Auxiliary Power Unit (APU)
- electric systems
- hydraulic systems

Based on formulas in [[DCAC14]], mass contribution C1

Store some bound methods so we can detect runtime overrides.

Parameters `**kwargs` (*dict of keyword arguments*) – Keyword arguments that will be mapped into the Component options.

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.weight.mass_breakdown.c_systems.c2_life_support_systems_weight module

Estimation of life support systems weight

class fastoad.models.weight.mass_breakdown.c_systems.c2_life_support_systems_weight.**LifeSup**

Bases: `openmdao.core.explicitcomponent.ExplicitComponent`

Weight estimation for life support systems

This includes:

- insulation
- air conditioning / pressurization
- de-icing
- internal lighting system
- seats and installation of crew
- fixed oxygen
- permanent security kits

Based on formulas in [[DCAC14]], mass contribution C2

Store some bound methods so we can detect runtime overrides.

Parameters `**kwargs` (*dict of keyword arguments*) – Keyword arguments that will be mapped into the Component options.

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.weight.mass_breakdown.c_systems.c3_navigation_systems_weight module

Estimation of navigation systems weight

class fastoad.models.weight.mass_breakdown.c_systems.c3_navigation_systems_weight.**Navigatio**

Bases: openmdao.core.explicitcomponent.ExplicitComponent

Weight estimation for navigation systems

Based on figures in [[DCAC14]], mass contribution C3

Store some bound methods so we can detect runtime overrides.

Parameters ****kwargs** (*dict of keyword arguments*) – Keyword arguments that will be mapped into the Component options.

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.weight.mass_breakdown.c_systems.c4_transmissions_systems_weight module

Estimation of transmissions systems weight

class fastoad.models.weight.mass_breakdown.c_systems.c4_transmissions_systems_weight.**Transm**

Bases: openmdao.core.explicitcomponent.ExplicitComponent

Weight estimation for transmission systems

Based on figures in [[DCAC14]], mass contribution C4

Store some bound methods so we can detect runtime overrides.

Parameters ****kwargs** (*dict of keyword arguments*) – Keyword arguments that will be mapped into the Component options.

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.weight.mass_breakdown.c_systems.c5_fixed_operational_systems_weight module

Estimation of fixed operational systems weight

class fastoad.models.weight.mass_breakdown.c_systems.c5_fixed_operational_systems_weight.**F**
Bases: openmdao.core.explicitcomponent.ExplicitComponent

Weight estimation for fixed operational systems (weather radar, flight recorder, . . .)

Based on formulas in [[DCAC14]], mass contribution C5

Store some bound methods so we can detect runtime overrides.

Parameters ****kwargs** (*dict of keyword arguments*) – Keyword arguments that will be mapped into the Component options.

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.weight.mass_breakdown.c_systems.c6_flight_kit_weight module

Estimation of flight kit weight

class fastoad.models.weight.mass_breakdown.c_systems.c6_flight_kit_weight.**FlightKitWeight**(

Bases: openmdao.core.explicitcomponent.ExplicitComponent

Weight estimation for flight kit (tools that are always on board)

Based on figures in [[DCAC14]], mass contribution C6

Store some bound methods so we can detect runtime overrides.

Parameters **`**kwargs`** (*dict of keyword arguments*) – Keyword arguments that will be mapped into the Component options.

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

Estimation of weight of all-mission systems

fastoad.models.weight.mass_breakdown.d_furniture package

Submodules

fastoad.models.weight.mass_breakdown.d_furniture.d1_cargo_configuration_weight module

Estimation of cargo configuration weight

class fastoad.models.weight.mass_breakdown.d_furniture.d1_cargo_configuration_weight.**Cargo(**

Bases: openmdao.core.explicitcomponent.ExplicitComponent

Weight estimation for equipments for freight transport (applies only for freighters)

Based on [[DCAC14]], mass contribution D1

Store some bound methods so we can detect runtime overrides.

Parameters **`**kwargs`** (*dict of keyword arguments*) – Keyword arguments that will be mapped into the Component options.

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.weight.mass_breakdown.d_furniture.d2_passenger_seats_weight module

Estimation of passenger seats weight

class fastoad.models.weight.mass_breakdown.d_furniture.d2_passenger_seats_weight.**PassengerS**
Bases: openmdao.core.explicitcomponent.ExplicitComponent

Weight estimation for passenger seats

Based on [[DCAC14]], mass contribution D2

Store some bound methods so we can detect runtime overrides.

Parameters ****kwargs** (*dict of keyword arguments*) – Keyword arguments that will be mapped into the Component options.

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.weight.mass_breakdown.d_furniture.d3_food_water_weight module

Estimation of food water weight

class fastoad.models.weight.mass_breakdown.d_furniture.d3_food_water_weight.**FoodWaterWeight**

Bases: openmdao.core.explicitcomponent.ExplicitComponent

Weight estimation for food and water

Includes trolleys, trays, cutlery. . .

Based on [[DCAC14]], mass contribution D3

Store some bound methods so we can detect runtime overrides.

Parameters ****kwargs** (*dict of keyword arguments*) – Keyword arguments that will be mapped into the Component options.

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.weight.mass_breakdown.d_furniture.d4_security_kit_weight module

Estimation of security kit weight

class fastoad.models.weight.mass_breakdown.d_furniture.d4_security_kit_weight.**SecurityKitWe**

Bases: openmdao.core.explicitcomponent.ExplicitComponent

Weight estimation for security kit

Based on [[DCAC14]], mass contribution D4

Store some bound methods so we can detect runtime overrides.

Parameters ****kwargs** (*dict of keyword arguments*) – Keyword arguments that will be mapped into the Component options.

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.weight.mass_breakdown.d_furniture.d5_toilets_weight module

Estimation of toilets weight

class fastoad.models.weight.mass_breakdown.d_furniture.d5_toilets_weight.**ToiletsWeight**(***kw*
Bases: openmdao.core.explicitcomponent.ExplicitComponent

Weight estimation for toilets

Based on [[DCAC14]], mass contribution D5

Store some bound methods so we can detect runtime overrides.

Parameters ****kwargs** (*dict of keyword arguments*) – Keyword arguments that will be mapped into the Component options.

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

Estimation of furniture weight

fastoad.models.weight.mass_breakdown.e_crew package

Submodules

fastoad.models.weight.mass_breakdown.e_crew.crew_weight module

Estimation of crew weight

class `fastoad.models.weight.mass_breakdown.e_crew.crew_weight.`**`CrewWeight`**(***kwargs*)

Bases: `openmdao.core.explicitcomponent.ExplicitComponent`

Weight estimation for aircraft crew

Based on [[DCAC14]], mass contribution E

Store some bound methods so we can detect runtime overrides.

Parameters ****kwargs** (*dict of keyword arguments*) – Keyword arguments that will be mapped into the Component options.

`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

Estimation of crew weight

Submodules

fastoad.models.weight.mass_breakdown.cs25 module

Computation of load cases

class `fastoad.models.weight.mass_breakdown.cs25.`**`Loads`**(***kwargs*)

Bases: `openmdao.core.explicitcomponent.ExplicitComponent`

Computes gust load cases

Load case 1: with wings with almost no fuel Load case 2: at maximum take-off weight

Based on formulas in [[DCAC14]], §6.3

Store some bound methods so we can detect runtime overrides.

Parameters ****kwargs** (*dict of keyword arguments*) – Keyword arguments that will be mapped into the Component options.

`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.weight.mass_breakdown.mass_breakdown module

Main components for mass breakdown.

class fastoad.models.weight.mass_breakdown.mass_breakdown.**MassBreakdown** (***kwargs*)
Bases: openmdao.core.group.Group

Computes analytically the mass of each part of the aircraft, and the resulting sum, the Overall Weight Empty (OWE).

Some models depend on MZFW (Max Zero Fuel Weight), MLW (Max Landing Weight) and MTOW (Max TakeOff Weight), which depend on OWE.

This model cycles for having consistent OWE, MZFW and MLW.

Options: - payload_from_npax: If True (default), payload masses will be computed from NPAX, if False

design payload mass and maximum payload mass must be provided.

Set the solvers to nonlinear and linear block Gauss–Seidel by default.

Parameters ****kwargs** (*dict*) – dict of arguments available here and in all descendants of this Group.

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.weight.mass_breakdown.mass_breakdown.**AirframeWeight** (***kwargs*)
Bases: openmdao.core.group.Group

Computes mass of airframe.

Set the solvers to nonlinear and linear block Gauss–Seidel by default.

Parameters ****kwargs** (*dict*) – dict of arguments available here and in all descendants of this Group.

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.weight.mass_breakdown.mass_breakdown.**PropulsionWeight**(***kwargs*)
Bases: openmdao.core.group.Group

Computes mass of propulsion.

Set the solvers to nonlinear and linear block Gauss–Seidel by default.

Parameters ****kwargs** (*dict*) – dict of arguments available here and in all descendants of this Group.

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.weight.mass_breakdown.mass_breakdown.**SystemsWeight**(***kwargs*)
Bases: openmdao.core.group.Group

Computes mass of systems.

Set the solvers to nonlinear and linear block Gauss–Seidel by default.

Parameters ****kwargs** (*dict*) – dict of arguments available here and in all descendants of this Group.

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.weight.mass_breakdown.mass_breakdown.**FurnitureWeight**(***kwargs*)
Bases: openmdao.core.group.Group

Computes mass of furniture.

Set the solvers to nonlinear and linear block Gauss–Seidel by default.

Parameters `**kwargs` (*dict*) – dict of arguments available here and in all descendants of this Group.

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.weight.mass_breakdown.mass_breakdown.**OperatingWeightEmpty**(***kwargs*)

Bases: `openmdao.core.group.Group`

Operating Empty Weight (OEW) estimation.

This group aggregates weight from all components of the aircraft.

Set the solvers to nonlinear and linear block Gauss–Seidel by default.

Parameters `**kwargs` (*dict*) – dict of arguments available here and in all descendants of this Group.

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.models.weight.mass_breakdown.payload module

Payload mass computation

class fastoad.models.weight.mass_breakdown.payload.**ComputePayload**(***kwargs*)

Bases: `openmdao.core.explicitcomponent.ExplicitComponent`

Computes payload from NPAX

Store some bound methods so we can detect runtime overrides.

Parameters `**kwargs` (*dict of keyword arguments*) – Keyword arguments that will be mapped into the Component options.

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.weight.mass_breakdown.update_mlw_and_mzfw module

Main component for mass breakdown

class fastoad.models.weight.mass_breakdown.update_mlw_and_mzfw.**UpdateMLWandMZFW**(***kwargs*)

Bases: openmdao.core.explicitcomponent.ExplicitComponent

Computes Maximum Landing Weight and Maximum Zero Fuel Weight from Overall Empty Weight and Maximum Payload.

Store some bound methods so we can detect runtime overrides.

Parameters ****kwargs** (*dict of keyword arguments*) – Keyword arguments that will be mapped into the Component options.

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

Estimation of Aircraft Weight

Submodules

fastoad.models.weight.weight module

Weight computation (mass and CG)

class fastoad.models.weight.weight.**Weight**(***kwargs*)

Bases: `openmdao.core.group.Group`

Computes masses and Centers of Gravity for each part of the empty operating aircraft, among these 5 categories: airframe, propulsion, systems, furniture, crew

This model uses MTOW as an input, as it allows to size some elements, but resulting OWE do not aim at being consistent with MTOW.

Consistency between OWE and MTOW can be achieved by cycling with a model that computes MTOW from OWE, which should come from a mission computation that will assess needed block fuel.

Set the solvers to nonlinear and linear block Gauss–Seidel by default.

Parameters ****kwargs** (*dict*) – dict of arguments available here and in all descendants of this Group.

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

Module contents

Submodules

fastoad.models.constants module

Module for management of options and factorizing their definition.

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*)

Bases: enum.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_*
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**(*facto*
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_cla 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

fastoad.module_management.service_registry module

Module for registering services.

class fastoad.module_management.service_registry.**RegisterService**(*provider_id: str*, *desc=None*, *domain: Optional[fastoad.module_management.c = None*, *options: Optional[dict] = None*)

Bases: object

Decorator class that allows to register a service, associated to a base class (or interface).

The registered class must inherit from this base class.

The definition of the base class is done by subclassing, e.g.:

```
class RegisterSomeService( RegisterService, base_class=ISomeService):
    "Allows to register classes that implement interface ISomeService."
```

Then basic registering of a class is done with:

```
@RegisterSomeService("my.particularservice")
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

classmethod explore_folder (*folder_path: str*)
Explores provided folder and looks for service providers to register.

Parameters folder_path –

classmethod get_provider_ids () → List[str]

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, 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.**RegisterPropulsion** (*provider_id: str, desc=None, domain: Optional[fastoad.module_manageme = None, options: Optional[dict] = None*)

Bases: fastoad.module_management.service_registry._RegisterOpenMDAOService

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_man = None, options: Optional[dict] = None*)

Bases: fastoad.module_management.service_registry._RegisterOpenMDAOService

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

classmethod explore_folder(*folder_path: str*)

Explores provided folder and looks for OpenMDAO systems to register.

Also, if there is a file for variable description at root of provided folder, it is read.

Parameters folder_path –

classmethod get_system(*identifier: str, options: Optional[dict] = None*) → openmdao.core.system.System

Specialized version of *RegisterService.get_provider()* that allows to define OpenMDAO options on-the-fly.

Parameters

- **identifier** – identifier of the registered class
- **options** – option values at system instantiation

Returns an OpenMDAO system instantiated from the registered class

service_id: str = 'fast.openmdao.system'

Module contents

Management of modules using Pelix/iPOPO

fastoad.openmdao package

Subpackages

Submodules

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.

Parameters

- **model** (*<System> or None*) – The top-level <System>. If not specified, an empty <Group> will be created.
- **driver** (*<Driver> or None*) – The driver for the problem. If not specified, a simple "Run Once" driver will be used.
- **comm** (*MPI.Comm or <FakeComm> or None*) – The global communicator.
- **name** (*str*) – Problem name. Can be used to specify a Problem instance when multiple Problems exist.
- ****options** (*named args*) – All remaining named args are converted to options.

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.
- **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.

- **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 boolean

write_outputs()
Writes all outputs in the configured output file.

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

property limit_units
Alias for field number 3

property limit_value
Alias for field number 2

property logger_name
Alias for field number 7

property source_file
Alias for field number 6

property status
Alias for field number 1

property value
Alias for field number 4

property value_units
Alias for field number 5

property 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.VariableList) → List[*fastoad.openmdao.validity_checker.CheckRecord*]

Check values of provided variables against registered limits.

Parameters `variables` –

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.variables module

Module for managing OpenMDAO variables

class `fastoad.openmdao.variables.`**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 units

units associated to value (or None if not found)

property description

description of the variable (or None if not found)

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

class fastoad.openmdao.variables.**VariableList**(*iterable=()*, */*)

Bases: list

Class for storing OpenMDAO variables

A list of Variable instances, but items can also be accessed through variable names.

There are 2 ways for adding a variable:

```
# 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 = VariableList()
vars.append(var_1)              # Adds directly a Variable instance
vars['var/2'] = metadata_2      # Adds the variable with given name and given
↪metadata
```

After that, following equalities are True:

```
print( var_1 in vars )
print( 'var/1' in vars.names() )
print( 'var/2' in vars.names() )
```

Note: Adding a Variable instance that has a name that is already in the VariableList instance will replace the previous Variable instance instead of adding a new one.

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) → 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:* fastoad.openmdao.variables.VariableList, *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
- 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.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.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.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*) → *fastoad.openmdao.variables.VariableList*

Creates a VariableList instance containing inputs and outputs of an OpenMDAO Problem.

Warning: problem.setup() must have been run.

The inputs (is_input=True) correspond to the variables of IndepVarComp components and all the unconnected variables.

Note: Variables from _auto_ivc are ignored.

Parameters

- **problem** – OpenMDAO Problem instance to inspect
- **use_initial_values** – if True, 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

Returns VariableList instance

classmethod from_unconnected_inputs(*problem: openmdao.core.problem.Problem, with_optional_inputs: bool = False*) → *fastoad.openmdao.variables.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

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*)
Bases: enum.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*)
Bases: enum.IntEnum

Enumeration of engine settings.

TAKEOFF = 1

CLIMB = 2

CRUISE = 3

IDLE = 4

classmethod convert(*name: str*) → *fastoad.constants.EngineSetting*

Parameters **name** –

Returns the EngineSetting instance that matches the provided name (case-insensitive)

class fastoad.constants.**RangeCategory**(*value*)
Bases: enum.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

[kro01] 2001. URL: https://web.archive.org/web/20010307121417/http://adg.stanford.edu/aa241/propulsion/nacelledesign.html.

[Ray99] Daniel P. Raymer. *Aircraft Design: A Conceptual Approach, Third edition*. AIAA (American Institute of Aeronautics & Astronautics, 1999. ISBN 1563473437.

[Rou02] Elodie Roux. *Modèles Moteurs... Réacteurs double flux civils et réacteurs militaires à faible taux de dilution avec Post-Combustion*. INSA-SupAéro-ONÉRA, 2002. URL: http://elodieroux.com/ReportFiles/ModelesMoteurVersionPublique.pdf.

[Rou05] Elodie Roux. *Pour une approche analytique de la Dynamique du Vol*. PhD thesis, SupAéro, 2005. URL: http://depozit.isae.fr/theses/2005/2005_Roux_Elodie.pdf.

[kro01] 2001. URL: https://web.archive.org/web/20010307121417/http://adg.stanford.edu/aa241/propulsion/nacelledesign.html.

[DCAC14] Willy Pierre Dupont, Christian Colongo, Olivier Atinault, and Christophe Cros. *Preliminary Design of a Commercial Transport Aircraft*. ISAE-SUPAERO, 2014.

[Ray99] Daniel P. Raymer. *Aircraft Design: A Conceptual Approach, Third edition*. AIAA (American Institute of Aeronautics & Astronautics, 1999. ISBN 1563473437.

[Rou02] Elodie Roux. *Modèles Moteurs... Réacteurs double flux civils et réacteurs militaires à faible taux de dilution avec Post-Combustion*. INSA-SupAéro-ONÉRA, 2002. URL: http://elodieroux.com/ReportFiles/ModelesMoteurVersionPublique.pdf.

[Rou05] Elodie Roux. *Pour une approche analytique de la Dynamique du Vol*. PhD thesis, SupAéro, 2005. URL: http://depozit.isae.fr/theses/2005/2005_Roux_Elodie.pdf.

PYTHON MODULE INDEX

f

INDEX

D

E

G

H

I

K

L

N

O

P

R

S

U

V