Consumet user manual

J.A. Ouassou, J. Straus, B.R. Knudsen, & R. Anantharaman

2021–06–30

Herein, we show how to install and use **Consumet**, an open-source **con**structor of **su**rrogates and **met**amodels. Consumet is written in Python 3, and constructs these models via a combination of penalized regression, adaptive sampling, and information criteria. For details, we refer to our technical paper on the subject [1].

Contents

1 License

Consumet is available as free and open-source software under the MIT license:

```
Copyright (c) 2019 SINTEF Energi AS

Permission is hereby granted, free of charge, to any person obtaining a copy
of this software and associated documentation files (the "Software"), to deal
in the Software without restriction, including without limitation the rights
to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
copies of the Software, and to permit persons to whom the Software is
furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all
copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY,
WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR
IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
```

This is a permissive license that essentially permits you to use the software for any purpose, as long as you just give credit where appropriate. However, outside of any legal obligations, the authors at SINTEF Energy Research kindly request that any useful modifications you make to the code be contributed back to us, so that we can improve the tool over time. If you find the tool useful for research, you may also consider citing our description of the tool in Ref. [1].

2 Installation

Consumet has the following system dependencies:

- `python3` used to implement all of our code;
- `nomad` used for optimization of non-differentiable problems;
- `ipopt` used for optimization of differentiable problems.

In addition, we rely on the following Python libraries:

- `numpy` standard library for numerical programming;
- `scipy` standard library for scientific computing;
- `pyomo` optimization framework used for regression;
- `pycddlib` used for performing vertex calculations;
- `pydoe` used for constructing experimental designs.

The easiest way to obtain such a setup is by downloading and installing the Python3 version of Anaconda for your operating system. Almost almost all the dependencies listed above are then available via the `conda` package manager. Simply open *Anaconda Prompt* from your start menu if you use Windows, or open a normal system terminal if you use Linux or Mac. Then enter the following command in the terminal window in order to install the dependencies via `conda`:

```
conda install -c conda-forge numpy scipy pydoe pyomo pyomo.extras ipopt
```

One exception is however `pycddlib`, which is not currently available via `conda`. This library can however be installed via the `pip` package manager, which should also have been installed as part of Anaconda. To do this, enter the following command into the terminal window:

```
pip install pycddlib
```

The other exception is `nomad`. If you use Windows or Mac, you can download and install prebuilt `nomad` packages from SourceForge. If you use Linux, you will have to download the source code from either SourceForge or the official website, and then manually compile the code in the usual manner (i.e. by running the commands `./configure` and `make` from a terminal). For more documentation on how to install `nomad`, we refer to chapter 2 of the NOMAD User Guide.

On all platforms, you need to manually add the location of the `nomad` executable to your system path. If you use Windows 10, this can be done by going to *Edit the system environment variables* in the control panel, selecting *Path* under *System variables*, and clicking *Edit*. If you installed `nomad` v. 3.9.0 to the default location, you can add `C:\Program Files (x86)\nomad.3.9.0\bin` as a new entry, and save the settings. If you use Linux with the `bash` shell, and copied the `nomad` folder to `/opt/nomad` after compilation, you can add `export PATH=$PATH:/opt/nomad/bin` to the end of your `~/.bashrc`. On all platforms, you can verify that the path has been updated correctly by opening a new terminal window, entering the command `nomad`, and checking that you don't get system errors. For more information, we again refer to the NOMAD User Guide.

Once all the dependencies above have been installed, no special procedures are required to install Consumet. Simply extract all the files of the project to an arbitrary folder on your computer, open a terminal, and continue following the instructions in the next section.

3 Basic usage

In order to use Consumet for your project, you have to provide two files:

`config.ini` — This contains the configuration options used for surrogate construction, such as e.g. the desired model class, model order, and variable bounds. The available configuration options are listed and described in section 5.

`true_model.py` — This should define a function `simulate`, which takes a 1-dimensional list or array as input, and returns a 1-dimensional list or array as output. This defines the $\mathbb{R}^d \rightarrow \mathbb{R}^r$ process that we create a surrogate model for.

These can be placed anywhere you want, and the output files generated by the surrogate modeling tool will then end up in the same folder. Note that you have to change folders to the location of these files *before* executing Consumet in order for the program to find them.

For instance, if we wish to model the Rosenbrock function, we can define `true_model.py`:

```
def simulate(x):
    z = [ (1-x[0])**2 + 100*(x[1]-x[0]**2)**2 ]
    return z
```

Let us now say that we wish to use a 4th-order 2-dimensional Taylor series as our surrogate model, with box constraints $-2 < x_0 < 2$ and $-1 < x_1 < 3$. We can then define `config.ini` as:

```
model_class = taylor
model_order = 4
input_dim   = 2
input_lb    = [-2., -1.]
input_ub    = [ 2.,  3.]
```

After that, we simply need to run Consumet from the folder where these files are. For example, say that you extracted the surrogate modeling tool to a folder named `Consumet` on your desktop, and placed `config.ini` and `true_model.py` in a folder `Simulation` on your desktop. You can then generate the surrogates by opening a terminal and running the following commands:[1]

```
cd Desktop/Simulation
python ../Consumet/bin/consumet.py
```

More examples of how to setup `config.ini` and `true_model.py` are available in the `examples` subfolder of the project documentation. This includes both pure Python examples and examples of how to couple Consumet to MS Excel. Since many commercial software packages provide interfaces to Excel, including e.g. Aspen Plus and Aspen HYSYS, the MS Excel examples may also be of interest for users wishing to generate surrogates for models implemented in those.

When the surrogate model construction is complete, the sampled data will be saved in `Simulation/samples.csv` and the model coefficients in `Simulation/regression.csv`. The formats of these output files are straight-forward. When modeling an $\mathbb{R}^d \longrightarrow \mathbb{R}^r$ process, `samples.csv` will contain 1 column with a sample number, d columns describing the process input $\boldsymbol{x} \in \mathbb{R}^d$, and r columns describing the process output $\boldsymbol{z} \in \mathbb{R}^r$. So if e.g. the 0th sample was at $\boldsymbol{x} = (0.25, 0.75)$ and produced the result $\boldsymbol{z} = (0.3, 60)$, `samples.csv` would contain the line:

```
0,2.500000e-01,7.500000e-01,3.000000e-01,6.000000e+01
```

Before discussing the format of `regression.csv`, it is useful to briefly reiterate from Ref. [1] how we formulate our surrogate models. Firstly, we should mention that the surrogate models are formulated in terms of *standardized variables* $\xi_i := (x_i - x_i^{\min})/(x_i^{\max} - x_i^{\min})$, where $x_i^{\min}$ and $x_i^{\max}$ refer to the input bounds specified in your `config.ini`. For instance, the configuration file for the Rosenbrock example above implies that $\xi_0 = (x_0 + 2)/(2 + 2)$ and $\xi_1 = (x_1 + 1)/(3 + 1)$. This procedure basically maps all input vectors $\boldsymbol{x}$ within bounds to the new variables $\boldsymbol{\xi} \in [0, 1]^d$. In terms of these standardized variables, the final equation that describes the surrogate model output $\boldsymbol{z} = (z_0, \ldots, z_{r-1})$ as function of the standardized process input $\boldsymbol{\xi} = (\xi_0, \ldots, \xi_{d-1})$ is:

$$z_m = \sum_{n_0} \cdots \sum_{n_{d-1}} \theta_{m,n_0,\ldots,n_{d-1}} b_{n_0}(\xi_0) \cdots b_{n_{d-1}}(\xi_{d-1}) \tag{1}$$

[1] On some Linux distributions, `python` refers to `python2`, in which case you have to write `python3` instead.

Here, $b_n(\xi)$ are the one-dimensional basis functions chosen to construct surrogate models, $\theta_{m,n_0,\ldots,n_{d-1}}$ are the corresponding regression coefficients that will be written to `regression.csv`, and the sums should be taken over all the n_i's that are written to file.[2] To make this a bit less abstract, let us focus on the special case of a process that has 2D input and 2D output:

$$z_0 = \sum_{n_0}\sum_{n_1} \theta_{0,n_0,n_1}\, b_{n_0}(\xi_0) b_{n_1}(\xi_1) \qquad z_1 = \sum_{n_0}\sum_{n_1} \theta_{1,n_0,n_1}\, b_{n_0}(\xi_0) b_{n_1}(\xi_1) \tag{2}$$

To make the structure of the results even clearer, we can further limit our scope to Taylor series as basis functions $[b_0(\xi) = 1, b_1(\xi) = \xi, b_2(\xi) = \xi^2]$, and set the model order to 2, which yields:

$$\begin{aligned} z_0 &= \theta_{0,0,0} + \theta_{0,1,0}\xi_0 + \theta_{0,0,1}\xi_1 + \theta_{0,2,0}\xi_0^2 + \theta_{0,0,2}\xi_1^2 + \theta_{0,1,1}\xi_0\xi_1 \\ z_1 &= \theta_{1,0,0} + \theta_{1,1,0}\xi_0 + \theta_{1,0,1}\xi_1 + \theta_{1,2,0}\xi_0^2 + \theta_{1,0,2}\xi_1^2 + \theta_{1,1,1}\xi_0\xi_1 \end{aligned} \tag{3}$$

This illustrates the logic behind the coefficient indices well: $\theta_{1,2,0}$ describes a term in the surrogate model for z_1 that is 2nd-order in ξ_0 and 0th-order in ξ_1, which obviously is the ξ_0^2 term in this case. Since the models are fit using penalized regression, many of these coefficients can be zero, especially if one uses constrained sampling. Now that we have described the structure of our surrogate models, the format of the output file `regression.csv` is trivial: each line simply contains the subscripts of $\theta_{m,n_0,\ldots,n_{d-1}}$ followed by the coefficient value itself. So if the value for the coefficient $\theta_{1,2,0} = 0.25$, the output file `regression.csv` would contain a line like this:

```
1,2,0,2.500000e-01
```

Given such an output-file, it is straight-forward to recreate the surrogate model in Python. For instance, the following code defines a function `rosenbrock` which should behave as expected:

```
# Load regression coefficients from file
import numpy
data = numpy.genfromtxt('regression.csv', delimiter=',')

# Reimplement the polynomial surrogate model
def rosenbrock(x, y):
    return sum([t * x**n * y**m for _, n, m, t in data])
```

If we as basis functions instead of Taylor series used Legendre polynomials P_n, Chebyshev polynomials T_n, or Fourier series, the coefficient $\theta_{1,2,0}$ would similarly describe contributions from terms $P_2(2\xi_0 - 1) = [3(2\xi_0^2 - 1) - 1]/2$, $T_2(2\xi_0 - 1) = 2(2\xi_0 - 1)^2 - 1$, and $\sin(2\pi\xi_0)$, respectively.[3] The code above used to recreate surrogates from `regression.csv` has to be adjusted accordingly. The currently available basis functions are implemented as shown in table 1. For more details we again refer to Ref. [1], as well as the SciPy documentaion for the orthogonal polynomials. New basis functions can also easily be appended to the end of the file `lib/surrogate.py`.

[2] The values for n_i included in the models are constrained by the chosen model order, as described in Ref. [1].

[3] T_n and P_n are evaluated at $2\xi_i - 1$ since they form an orthonormal basis on $[-1, +1]$ but we standardized ξ_i to $[0, 1]$.

Table 1: List of available model classes in Consumet.

Model class	Basis function $b_n(\xi)$
Taylor	`xi**n`
Legendre	`scipy.special.eval_legendre(n, 2*xi-1)`
Chebyshev	`scipy.special.eval_chebyt(n, 2*xi-1)`
Fourier	`numpy.sin(n*pi*xi) if n > 0 else 1`

4 Advanced usage

4.1 Pickled surrogates

To use the constructed surrogates in *other* languages than Python, you have to implement the basis functions discussed in section 3, manually standardize your input variables $\boldsymbol{x}$ to obtain $\boldsymbol{\xi}$, and finally load the surrogate regression coefficients from `regression.csv`. However, if your code is written in Python, there is another simpler alternative: you can import the constructed surrogate models from binary files. In order to do this, you first have to add the location of the Consumet libraries to your Python path, and import the library `surrogate` from that folder. If you installed Consumet to e.g. the folder `/opt/consumet`, this can be done as follows:

```
import sys
sys.path.append('/opt/consumet/lib')

import surrogate
```

You can then use the `pickle` library to open the Consumet output file named `surrogate.pkl`:

```
import pickle

with open('surrogate.pkl', 'rb') as f:
  surrogate = pickle.load(f)
```

The result should be an array of `Surrogate` objects, where each component acts as a normal function. For instance, if you have an input $\boldsymbol{\xi} = (0.25, 0.75)$, `surrogate[0]([0.25,0.75])` would return the value of z_0 at that point, while `surrogate[1]([0.25,0.75])` returns z_1. Note that $\boldsymbol{\xi}$ here refers to the *standardized* input variable, as discussed in section 3.

If you use this feature, we still recommend saving the `regression.csv` and `samples.csv` files. Pickle files are not always stable across Python version updates, and do not always work when copied between computers or operating systems, in which case it is useful to be able to load `regression.csv` instead of redoing the entire surrogate fitting. The file `samples.csv` can be used as an input to the batch sampling routine of the surrogate modeling tool, and can therefore be used to recreate surrogate models without performing any new sampling.

Once you have imported the `Surrogate` objects discussed above, it is also quite easy to transform between raw input variables $\boldsymbol{x}$ and standardized variables $\boldsymbol{\xi}$. Standardization of $\boldsymbol{x}$ can be achieved using the function `Surrogate.standard`. Conversely, the unscaled variables $\boldsymbol{x}$ can be restored from $\boldsymbol{\xi}$ using the function `Surrogate.restore`. Both function takes a list or `numpy`

array as their inputs ($\boldsymbol{x}$ or $\boldsymbol{\xi}$), and return a list or `numpy` array as their outputs. It does not matter which of the `Surrogate` objects in the `surrogate` list above is used for standardization or restoration, as all surrogate models have the same input bounds. Thus, for e.g. the Rosenbrock example discussed in section 3, the fact that $\boldsymbol{\xi} = (0.25, 0.75)$ corresponds to $\boldsymbol{x} = (-1, 2)$ could have been determined by running e.g. `xi = surrogate[0].standard([-1,2])`.

4.2 Constrained sampling

Consumet allows constrained sampling via the option `input_file`, that is, it restricts the sampling domain to a subspace of the overall sampling domain. Constrained sampling is achieved by providing a `csv`-file in which input data to the model is provided. The input data is sampled from the previous unit operations either through a surrogate model or the detailed model. Manipulated variables in the model can however not be constrained as constrained sampling uses input data to the detailed model and is therefore dependent on previous unit operations. Based on these previous results, linear constraints $\boldsymbol{Ax} < \boldsymbol{b}$ are automatically generated by Consumet, by using the provided input file to calculate the constraint parameters $\boldsymbol{A}$ and $\boldsymbol{b}$.

The structure of the `input_file` is as follows. The first row is a header line which describes which columns should be used to generate constraints, and which columns correspond to which input components. If we e.g. wish to discard the 0th and 2nd columns of the file, use the 1st column as x_2, and the 3rd column as x_0, we can write:

```
X,2,X,0
```

Any non-numeric header value such as `X`, `nan`, etc. will discard a column from the file. The rest of the file simply contains the results from previous unit operations. For example, for a data point where the unused variables are 0.25 and 0.75, while $x_0 = 1$ and $x_2 = 3$, the entry would be:

```
0.25,3.0,0.75,1.0
```

All subsequent rows follow this format and will be used for the constrained sampling. Note that the input data (i.e. x_0 and x_2) has to be scaled so that it can be directly used by `true_model.py`.

It is worth noting that the format of this `csv`-file is basically identical to the `samples.csv` file discussed in section 3. Thus, one can easily use the output file `samples.csv` from one surrogate model construction to constrain the sampling domain when constructing surrogate models for later unit operations. The only deviation between the formats is that `samples.csv` does not contain the header line discussed above, which thus has to be added manually.

5 Parameters

Below, we list all the available options you can set in `config.ini` and briefly discuss their uses.

`model_class` — What kind of model to construct. Currently, the choices available are `taylor` (Taylor series), `fourier` (Fourier series), `legendre` (Legendre polynomials), and `chebyshev` (Chebyshev polynomials). New model classes can easily be added to the end of `surrogate.py` if necessary. If

`output_dim` > 1, it is also possible to specify different model classes for each output by setting this option to a list. For instance, one may define `model_class = [taylor, taylor, fourier]` for `output_dim = 3`.

`model_order` — Number of basis functions to use. If e.g. `model_class = taylor` and `model_order = 2`, we would in 2D get the basis set $\{1, x_0, x_0^2, x_1, x_1^2, x_0 x_1\}$. If `output_dim` > 1, it is also possible to specify different model orders for each output by specifying a list; for instance, `model_order = [4,2,3]`.

`input_dim` — Number of input dimensions. If we are trying to construct a surrogate model for a process $\boldsymbol{z} = f(\boldsymbol{x})$, this is the number of components $\boldsymbol{x}$ has.

`input_lb` — Lower bound for each component in $\boldsymbol{x}$ above. This should be a list.

`input_ub` — Upper bound for each component in $\boldsymbol{x}$ above. This should be a list.

`input_file` — Optional `csv`-file used to calculate inequality constraints $\boldsymbol{A}\boldsymbol{x} \leq \boldsymbol{b}$. This can increase accuracy and decrease computation time cf. only specifying box constraints (i.e. upper and lower bounds for the components of $\boldsymbol{x}$). See section 4.2 for more information about the use of this parameter.

`output_dim` — Number of output dimensions. If we are trying to construct a surrogate model for a process $\boldsymbol{z} = f(\boldsymbol{x})$, this is the number of components $\boldsymbol{z}$ has.

`batch_file` — If available, one can load previously sampled data from a file, in which case this option can be set to its filename. This should be a `csv`-file, where first column is the sample index, the next columns are the components of the input variable $\boldsymbol{x}$, and the final columns are the components of the output variable $\boldsymbol{z} = f(\boldsymbol{x})$. Note that the file `samples.csv` generated by the program can be used as a `batch_file` for future simulations.

`batch_doe` — If no batch file is available, or the number of samples is too low for regression, a design-of-experiment method is used to procure the initial samples. This option selects what method to use: `LHS` (Latin Hypercube Sampling), `Sobol` (Sobol Sequence), `MonteCarlo`, or `RegularGrid`.

`batch_num` — Number of batch samples to select via design of experiment. If this number is too low compared to the number of regression parameters in the chosen surrogate model, it is automatically increased to the minimum. The default value is 0, i.e. the minimum number deemed necessary.

`batch_corn` — Whether batch sampling should include input-domain corner points when using a design of experiment. This avoids extrapolation but comes at the cost of $2^{\texttt{input\_dim}}$ additional points. By default, this is set to 0 (off).

`adapt_num` — Maximal number of adaptive sampling iterations before giving up on obtaining the requested precision. If this is zero, adaptive sampling is disabled, and only batch sampling is performed. The default value is 5.

`adapt_tol`	Error tolerance of the adaptive sampling routine. The default value is 10^{-3}, i.e. the maximum error on the input domain should be below 0.1%.
`adapt_type`	Adaptive sampling algorithm to use. The default and recommended option is `seq` (sequential), but `sim` (simultaneous) is also available. Note that these algorithms only produce different results for `output_dim` > 1.
`adapt_pen`	Anti-clustering penalty used when performing adaptive sampling. This option sets the magnitude of the penalty.
`adapt_rad`	Anti-clustering penalty used when performing adaptive sampling. This option sets the radius of the penalty region.
`nomad_exe`	Executable used by `nomad`. Most users won't need to change this.
`nomad_num`	Internal iteration limit used by `nomad`.
`nomad_tol`	Internal tolerance used by `nomad`.
`regpen_crit`	Information criterion used to select regression penalty. The options are `aic` (Akaike), `bic` (Bayesian), `hqic` (Hannan–Quinn), and the versions `aicc`, `bicc`, `hqicc` with low-sample corrections. The default is `aicc`.
`regpen_lim`	Limit on how small a regression parameter can become before it is eliminated from the model. If e.g. `regpen_lim = 1e-4`, then terms in the regression that affect the output z by less than 10^{-4} compared to the dominant term in the model are automatically dropped from the model.
`regpen_num`	Number of logarithmically spaced regression penalties to test.
`regpen_lb`	Lower bound for the regression penalty. This is typically a few orders of magnitude lower than the expected variations in the output variables z.
`regpen_ub`	Upper bound for the regression penalty. This is typically a few orders of magnitude higher than the expected variations in the output variables z.

6 References

[1] J. Straus, B.R. Knudsen, J.A. Ouassou, R. Anantharaman. *Constrained adaptive sampling for domain reduction in surrogate model generation.* AIChE Journal (2021).