# Moreesc Documentation

## Release 2.0

**F. Silva, Ch. Vergez, J. Kergomard and Ph. Guillemain**

Jan 07, 2013

# Contents


**1 `Profiles` – Time-varying quantities** **3**
- 1.1 Academic profiles . . . . . 5
- 1.2 Using measured values . . . . . 7

**2 `Valve` – Specifiyng the mechanical resonator** **9**
- 2.1 Simple examples of `Valve` . . . . . 11

**3 `AcousticResonator` – Specifiyng the acoustical resonator** **13**
- 3.1 Physical constants . . . . . 13
- 3.2 General class: everything can be time variable . . . . . 14
- 3.3 Time-invariant version . . . . . 16
- 3.4 Cylindrical bore . . . . . 18
- 3.5 Measured Impedance . . . . . 19
- 3.6 References . . . . . 22

**4 `Simulation` – Time-domain simulation** **23**
- 4.1 Defining the problem . . . . . 23
- 4.2 Configuring the solver and integrate . . . . . 25
- 4.3 Postprocessing . . . . . 25
- 4.4 Data persistence . . . . . 27

**5 Indices and tables** **29**

**Bibliography** **31**

**Python Module Index** **33**

**Python Module Index** **35**

**Index** **37**

**MOREESC** is an application for the calculations of the auto-oscillations in wind instruments, designed originally for single reed instruments but it would be usable for brass too. It takes advantage of a modal decomposition of the acoustic pressure field in the bore, or of its input impedance, to let you compute the sound of any strangely-shaped instrument.

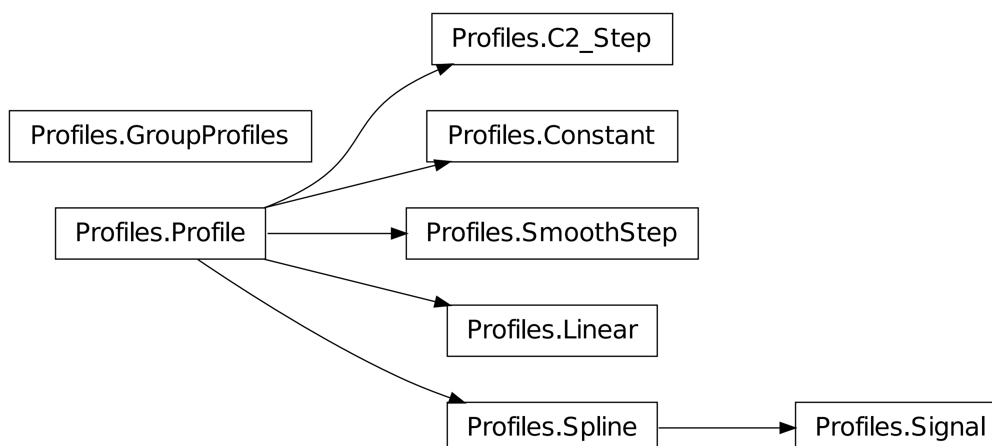
This document is intended to be used as a reference guide, but also be read as a long and detailed tutorial as the useful functions and class are also sequentially described here.

CHAPTER 1

# Profiles – Time-varying quantities

**Moreesc** is intended to deal with time-varying control parameters and time-evolving characteristics of resonator and valve. This module provide the framework for modelling simple or real-world data, and be able to perform efficient computation on them. It builds instances of `Profile` (or one of its subclass) that can easily be evaluated within the calculation of self-sustained oscillations.

In addition to basic constant and linear profiles `Constant` and `Linear`, `Spline` is based on B-splines [BSplines], which are transformed without any approximation into a sequence of Bezier curves [Bezier] for a cheaper evaluation during the calculation of oscillations. Measured signals can thus be parametrized and used with the computation (see `Signal`). `Spline` (and its subclass `Signal`) can easily be manipulated by means of a graphical `Spline.editor()`.




**class** `Profiles.Profile`(*dtype=<type 'float'>*)

General class intended to provide a parametrization of temporal evolution of coefficients of the model. The method used here bases on a decomposition into a sequence of Bezier curves in the (t,value) plane.

The `coefs` attribute consists of a $(2, 4n)$-shaped numpy array, the two lines containing the instants and the values of the nodes, respectively. Groups of 4 rows defines the four Bezier nodes of each Bezier curves (of degree 3). Thus the n-th curves of the Profile is made from

the nodes

$$\forall\ i\ \in\ [0,3], t = \text{coefs}[0, 4*n+i], \quad value = \text{coefs}[1, 4*n+i].$$

### Examples

```
>>> f = Profiles.Profile()  # Dummy empty profile, constant 0.
>>> print f
<moreesc.Profiles.Profile object at 0xb05e78c> empty
>>> print f.coefs           # No coefficients in the empty profile
[]
```

### Attributes

| | | |
|---|---|---|
| coefs | array | The coefficients that parametrizes the `Profile` as a sequence of Bezier curves. |

`__call__(t)`

Evaluate the profile at various instants using the compiled functions `(d|z)profile_eval()`.

**Parameters**

**t: array-like** :

Sequence of instants where to evaluate the profile.

**Returns**

**out: array** :

The values of the profile at those instants. Returned with the same shape as the input.

### Examples

```
>>> print f(0.0) # Initial (at t=0) value of f
0.0
>>> t = np.linspace(0,1, 1024) # Time vector
>>> print f(t) # f sampled at 1024Hz.
array([ 0.,   0.,   0., ...,   0.,   0.,   0.])
```

`save(filename)`

Saves instance to file using pickle [pick]. Note that this format may not be the more appropriate for exchange with other scientific software (see numpy and scipy I/O).

See `Profiles.load_profile()` for loading.

### Examples

```
>>> f.save('/tmp/profile.dat')
>>> g = Profiles.load_profile('/tmp/profile.dat')
>>> print g
<moreesc.Profiles.Profile at 0xb070ccc> ...
```

`Profiles.load_profile(s)`

Load data saved with the `Profile.save()` method:

**Parameters**

**filename: file-like object (file or string)** :

Name of the file to load.

**Returns**
**obj: Profile or subclass** :

Stored Profile

**class** `Profiles.GroupProfiles(`*profiles*, *keys=None*`)`
Defines a group of Profiles that can all be evaluated in a single call. The coefficients of each Profile are concatenated in on big array (`coefs_array`), and information about their respective shapes are grouped in the attribute `shapes_array`.

Individual Profiles stored within an instance of this class can be accessed as if `GroupProfiles` is a simple list, or even as if it is a dictionnary if a dictionnary or keys are provided.

`__call__(`*t*`)`
Evaluate the group of profile at various instants using the compiled functions `(d|z)group_profile_eval()`.

**Parameters**
**t: array-like** :

Sequence of instants where to evaluate the profile.

**Returns**
**out: list of arrays** :

The values of the profiles at those instants. Returned with the same shape as the input. Each element of the list is associated to one of the Profiles

`save(`*filename*`)`
Saves instance to fileusing pickle [pick]. See `Profiles.load_groupprofiles()` for loading.

`Profiles.load_groupprofiles(`*s*`)`
Load data saved with the `GroupProfiles.save()` method:

**Parameters**
**filename: file-like object (file or string)** :

Name of the file to load.

**Returns**
**obj: GroupProfiles** :

Stored GroupProfiles

## 1.1 Academic profiles

The following class are convenient subclass of the general `Profiles.Profile` class. Basic arithmetic (+,-,*,^,/) is available and warning are emitted when approximation is used during the operation.

**class** `Profiles.Constant(`*value*`)`
Bases: `Profiles.Profile`

`Profile` subclass to handle constant real values.

### Attributes

| value | float | The numeric value of the `Profiles.Constant` instance |
|---|---|---|

**class** `Profiles.Linear`(*instants*, *values*)

Bases: `Profiles.Profile`

`Profile` subclass to handle linearly varying values.

**class** `Profiles.Spline`(*tck*)

Bases: `Profiles.Profile`

`Spline` represents a parametrization of a time-varying quantity. It is modelled with BSpline of degree 3, leading (under normal conditions) to a $\mathcal{C}^2$ continuous.

### Attributes

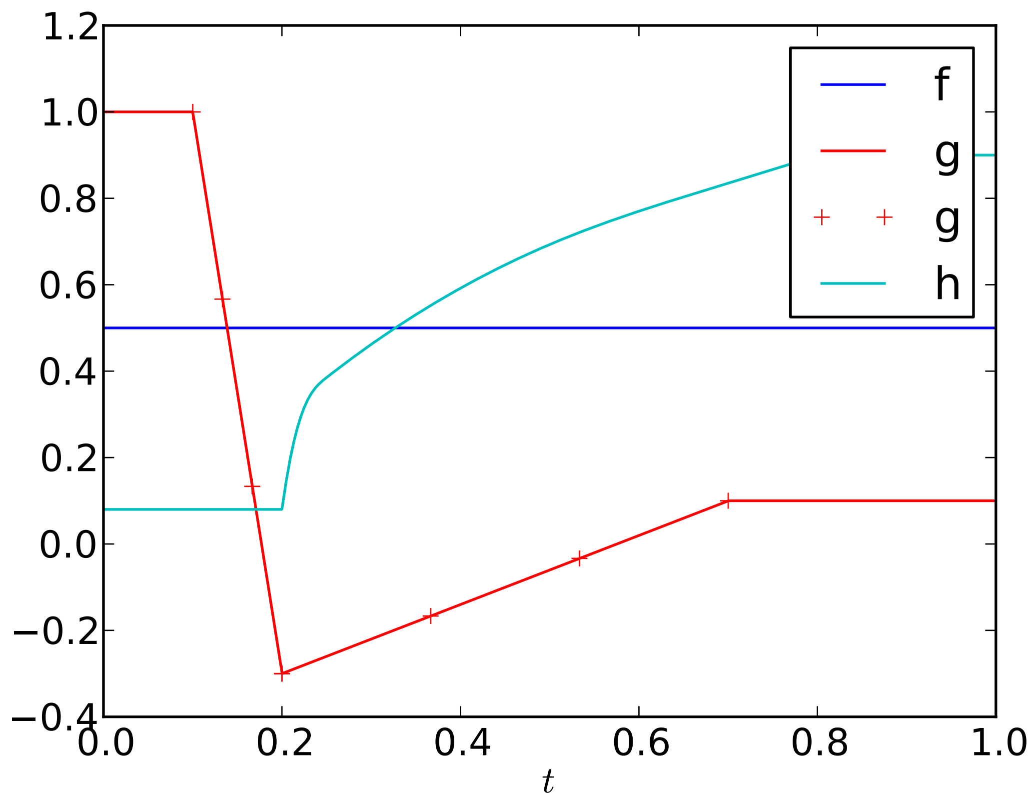
| t,c,k : | Tuple $(t, c, k)$ as used in the scipy spline library [spl] |
|---|---|

`editor()`

Raise a graphical user interface to manipulate the control points of the spline. If called from a `Signal` instance, it allows to change the tolerance of fitting procedure.

## 1.1.1 Examples

```
>>> import matplotlib.pyplot as plt
>>> t = np.linspace(0,1,2014)
>>> f = moreesc.Profiles.Constant(0.5); valf = f(t)
>>> g = moreesc.Profiles.Linear([0.1, .2, .7], [1., -.3, .1]); valg = g(t)
>>> knots = np.array([0.2, 0.2, 0.2, 0.2, .25, .5, .8, .8, .8, .8])
>>> coefs = [np.array([.08, .33, .5, .74, .83, .9]),]
>>> h = moreesc.Profiles.Spline([knots, coefs, 3]); valh = h(t)
>>> plt.plot(t, valf, label='f')
>>> plt.plot(t, valg, g.instants, g.coefs, '+', c='r', label='g')
>>> plt.plot(t, valh, label='h')
>>> plt.legend(); plt.xlabel(r'$t$'); plt.show()
```




## 1.2 Using measured values

**class** `Profiles.Signal`(*time=None*, *signal=None*, *smoothness=1.0*)

Bases: `Profiles.Spline`

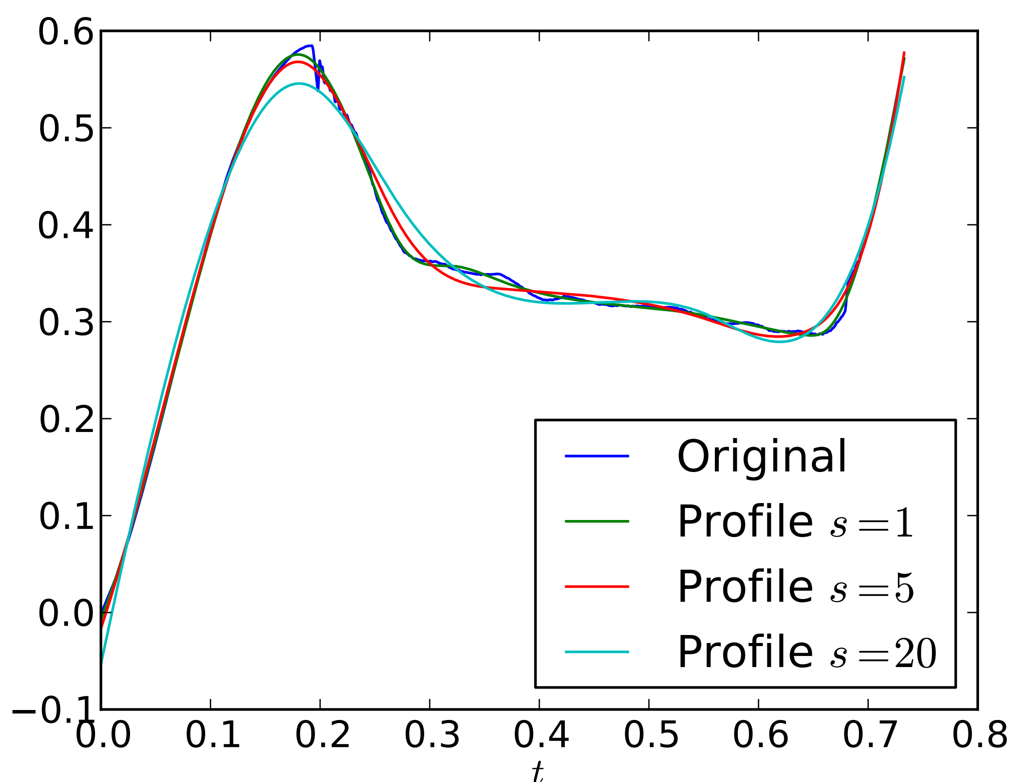
`Spline` represents a parametrization of a signal, for example a measured mouth pressure.

### Attributes

| time, signal | arrays | The samples of the signal to be parametrized. |
|---|---|---|
| s | float | The tolerance of fitting (also called smoothing condition in [spl]). |
| w | float or array | Weighting coefficients (see [spl]). |

### Examples

```
>>> tmp = np.load('Signal.npz'); sig = tmp['sig']; t = tmp['t']
>>> plt.plot(t, sig, label='Original')
>>> f = moreesc.Profiles.Signal(t, sig, smoothness=1.)
>>> plt.plot(t, f(t), label=r'Profile $s=1$')
>>> f.fit_spline(s=5.*f.sref);  plt.plot(t, f(t), label=r'Profile $s=5$')
>>> f.fit_spline(s=20.*f.sref); plt.plot(t, f(t), label=r'Profile $s=20$')
>>> plt.legend(loc='lower right'); plt.xlabel(r'$t$'); plt.show()
```




`__init__`(*time=None*, *signal=None*, *smoothness=1.0*)
Constructor for `Signal` class. :param signal: the samples of the signal to be parametrized, :param time: the sampling instants. The spline curve fitting is applied in this constructor with defaut values of tolerance (`s`) and ponderation (`w`) but it can be refined by calling `fit_spline()` method.

`fit_spline`(*s=None*, *w=None*)
Fit a B-spline representation of the provided signal. For sake of simplicity in the editor, the B-spline is N-D (with N=1) and the parameter values are the samples instants. See scipy.interpolate.splprep for details on the smoothing factor s and the weights w.

CHAPTER 2

# `Valve` – Specifiyng the mechanical resonator

This module provides the class objects useful for the description of the valve.

**class** `Valve.TransferFunction`(*num*, *den*)

A general class to represent systems by means of Laplace transform. It describes a linear dynamical system by numerator and denominator polynomials of the $s$ variable.

$$F(s = \alpha + j\omega) = \frac{\sum_{m=0}^{M} b_m s^{M-m}}{\sum_{=0}^{N} a_n s^{N-n}}$$

with $M < N$, usually evaluated on the frequency axis $s = j\omega$.

### Attributes

| | | |
|---|---|---|
| num,den | `Profiles.GroupProfiles` | Contain the (possibly time variable) coefficients of the numerator and the denominator of the `TransferFunction` instance. |

`__init__`(*num*, *den*)

Constructor of the TransferFunction class.

**Parameters**

**num, den** : list of scalars or list of Profiles

Sequence of the coefficients (potentially time varying) of numerator and denominator polynomials from highest powers of the Laplace variable $s$ to lowest.

`__call__`(*s*, *t=0.0*)

Evaluate the transfer function for the given values of the Laplace variable s. As the coefficients may (slowly) depend on time, the frequency response is itself dynamic (depends on time).

**Parameters**

**s** : array-like

List of values of Laplace variable at which transfer function is evaluated.

**t** : scalar, optional

Instant to consider when evaluating the coefficients of the transfer function. Default is 0.

**Returns**

**H** : list of frequency responses (one per instant required).

`trace`(*f=None, linlog='lin', t=[0.0]*)

Plots representations of the frequency response. The graphics properties can be modified directly on the returned object.

**Parameters**

**f** : array-like, optional

Frequencies at which frequency response is evaluated.

**linlog** : string, optional

The modulus is displayed with a linear axis if `lin`, with a logarithmic axis if `log`. An exception is raised in others cases.

**t** : scalar, optional

Instant to consider when evaluating the coefficients of the transfer function. Default is 0.

**Returns**

**Fig** : `matplotlib.Figure`

The figure object

`get_poles`(*t=0.0*)

Evaluate the location of the poles of the transfer function. As the coefficients may (slowly) depend on time, the poles may vary.

**Parameters**

**t** : scalar, optional

Instant to consider when evaluating the coefficients of the transfer function. Default is 0.

**Returns**

**H** : list of lists of poles (one per instant required).

`get_zeros`(*t=0.0*)

Evaluate the location of the zeros of the transfer function. As the coefficients may (slowly) depend on time, the zeros may vary.

**Parameters**

**t** : scalar, optional

Instant to consider when evaluating the coefficients of the transfer function. Default is 0.

**Returns**

**H** : list of lists of zeros (one per instant required).

**Warning:** The computation of the opening signal resulting from the application of the transfer function on a pressure signal is performed using the canonical observer form with the following state representation

$$\dot{X}(t) = AX(t) + Bp(t) \text{ and } h(t) = CX(t)$$

with

$$A = \begin{pmatrix} -a_1 & 1 & 0 & \cdots & 0 \\ -a_2 & 0 & 1 & & 0 \\ \vdots & \vdots & & \ddots & \vdots \\ -a_{N-1} & 0 & \cdots & 0 & 1 \\ -a_N & 0 & \cdots & \cdots & 0 \end{pmatrix}, B = \begin{pmatrix} 0 & \cdots & 0 & b_M & \cdots & b_0 \end{pmatrix}^T \text{ and } C = \begin{pmatrix} 1 & 0 \cdots & 0 \end{pmatrix}.$$

The trouble is that this formulation solves the following equation:

$$\frac{d^N h}{dt^N} + \sum_{n=0}^{N-1} \frac{d^n}{dt^n}\left(a_{N-n}(t)h(t)\right) = \sum_{m=0}^{M} \frac{d^m}{dt^m}\left(b_{M-m}(t)p(t)\right)$$

which may differ from the original one:

$$\frac{d^N h}{dt^N} + \sum_{n=0}^{N-1} a_{N-n}\frac{d^n h}{dt^n} = \sum_{m=0}^{M} b_{M-m}(t)\frac{d^m p}{dt^m}$$

when the coefficients vary with respect to time. It is assumed that the variation of the coefficients are slow in comparison with the fundamental frequency of the sound to be synthetised.

## 2.1 Simple examples of `Valve`

One common example of `Valve.TransferFunction` is the one degree of freedom oscillator, defined in the subclass `Valve.OneDOFOscillator`:

**class** `Valve.OneDOFOscillator`(*wr*, *qr*, *H0=1.0*, *beating_factor=0.0*)

Bases: `Valve.TransferFunction`

A TransferFunction subclass intended to model a single degree of freedom oscillator.

$$H(s) = \frac{H0}{1 + q_r \frac{s}{\omega_r} + \frac{s^2}{\omega_r^2}}.$$

### Attributes

| wr | Profile | Natural angular frequency of the oscillator |
|---|---|---|
| qr | Profile | Damping of the oscillator |
| HO | Profile | Low frequency gain of the oscillator |

**Warning:** The previous warning result in solving, for the one d.o.f. oscillator, the equation

$$\frac{d^2h}{dt^2} + q_r\omega_r\frac{dh}{dt} + \left(\omega_r^2 + \frac{d}{dt}(q_r\omega_r)\right)h(t) = H_0(t)p(t)$$

This may be interpreted as the numerical dispersion of the scheme, depending on the time variation of the natural frequency and its damping.

For backward-compatibility purpose, the two following convenient classes are provided, inheriting `Valve.OneDOFOscillator` attributes

**class** `Valve.ReedDynamics`(*wr*, *qr*, *kr=1000000.0*, *beating_factor=1000.0*)
Bases: `Valve.OneDOFOscillator`

`ReedDynamics` provides a one degree of freedom mechanical oscillator that models the reed. It defines the characteristics of the transfer function between the pressure difference (between mouth $p_m$ and mouthpiece $p_e$) and the reed channel opening $h$.

$$\frac{d^2h}{dt^2} + q_r\omega_r\frac{dh}{dt} + \omega_r^2\left(h - h_0\right) = \frac{\omega_r^2}{K_r}\left(p_e(t) - p_m(t)\right)$$

with an inward valve behaviour: $K_r > 0$.

### Attributes

| | | |
|---|---|---|
| wr | Profile | Natural angular frequency of a vibrational mode of the reed, |
| qr | Profile | Damping of the same vibrational mode, |
| Kr | Profile | Mechanical stiffness of the reed (quasi static value), |
| W | float | Width of the reed opening. |

If needed, a constant valve «width» can be defined and passed to Fortran code. It may be used when it is assumed that the reed channel opening is rectangular of height $h(t)$ and constant width $W$.

**Warning:** Reed-motion induced flow is by now disabled.

**class** `Valve.LipDynamics`(*wr*, *qr*, *kr=1000000.0*, *beating_factor=0.0*)
Bases: `Valve.OneDOFOscillator`

`LipDynamics` provides a one degree of freedom mechanical oscillator that models the lip. It defines the characteristics of the transfer function between the pressure difference (between mouth $p_m$ and mouthpiece $p_e$) and the lip channel opening $h$.

$$\frac{d^2h}{dt^2} + q_r\omega_r\frac{dh}{dt} + \omega_r^2\left(h - h_0\right) = \frac{\omega_r^2}{K_r}\left(p_e(t) - p_m(t)\right)$$

with an outward valve behaviour: $K_r < 0$.

### Attributes

| | | |
|---|---|---|
| wr | Profile | Natural angular frequency of a vibrational mode of the lip, |
| qr | Profile | Damping of the same vibrational mode, |
| Kr | Profile | Mechanical stiffness of the lip (quasi static value), |
| W | float | Width of the lip opening. |

CHAPTER 3

# `AcousticResonator` – Specifiyng the acoustical resonator

This module provides the class objects useful for the modal description of the acoustic resonator. The base class for the definition is the `AcousticResonator.Impedance` described by its complex modes. Instances are constructed using numerical values or Profile parametrization of poles and residues..

$$Z(s=\alpha+j\omega)=\sum_{n=1}^{N}\frac{C_n}{s-s_n}+\frac{C_n^*}{s-s_n^*}$$

usually evaluated on the frequency axis

$$Z(\omega)=\sum_{n=1}^{N}\frac{C_n}{j\omega-s_n}+\frac{C_n^*}{j\omega-s_n^*}.$$

**Warning:** Only the positive frequency poles (i.e. the poles having positive imaginary part) have to be handled, as Hermitian symmetry is assumed.

## 3.1 Physical constants

Here are the constants used to compute the wavelength or the characteristic impedance for example.

`AcousticResonator.c` = **346.1924588600005**
  Wave speed in free space ($m/s$)

`AcousticResonator.rho` = **1.1851294093959732**
  Density of the air ($kg/m^3$)

Acoustics in quite narrow ducts involves viscous and thermal effects in the vicinity of the rigid walls. The following parameters are useful to compute the dissipation in such resonators.

`AcousticResonator.Cp` = **240.0**
  Specific heat under constant pressure ($Cal/(kg.°C)$)

`AcousticResonator.Cv` = **171.18402282453638**
Specific heat at constant volume ($Cal/(kg.°C)$)

`AcousticResonator.Cpv` = **1.402**
Ratio of the specific heats $C_p/C_v$

`AcousticResonator.nu` = **1.831037488e-05**
Shear coefficient ($kg/m/s$)

`AcousticResonator.lv` = **4.4628661036010312e-08**
Viscous boundary layer thickness ($m$)

`AcousticResonator.kappa` = **0.00624297844**
Thermal conductivity ($Cal/(m.s.°C)$)

`AcousticResonator.lt` = **6.3401161563647621e-08**
Thermal boundary layer thickness ($m$)

If you intent to compare simulations with experiments, you may need to set the temperature of the simulation (default is $298K$), so that wave speed and other acoustical constant used in the computations match the experimental ones.

`AcousticResonator.physical_constants(`*Temp=298.0*`)`
Updates the acoustical constants according to the specified temperature.

**Parameters**
**Temp** : float

The experiment temperature (in Kelvin).

Musical instruments are generally intended to radiate sound

`AcousticResonator.radiation_a`

`AcousticResonator.radiation_b`
`radiation_a` and `radiation_b` are the coefficients of an approximated radiation impedance for an unflanged cylinder [Silva:2009].

## 3.2 General class: everything can be time variable

**class** `AcousticResonator.TimeVariantImpedance(`*sn=None*, *Cn=None*, *reduced=False*, *Zc=None*`)`
A general class to represent the input impedance of a acoustic resonator.

### Attributes

| Zc | Profile | Characteristic impedance of the resonator (used for reduced residues). |
|---|---|---|
| poles | GroupProfiles | Poles $s_n$ of the input impedance. |
| residus | GroupProfiles | Residues $C_n$ associated to the poles. |
| nbmodal | integer | Number of modes in the instance. |

> **Warning:** Moreesc requires a physical (non dimensionless) input impedance to compute the possible self-oscillations. You can either provide full-featured or dimensionless residues, but you must indicate it to the constructer with the `reduced` argument. In the latter case, you must also indicate the characteristic impedance `Z_c`, otherwise it is optional and only used to compute the reflection coefficient in `Impedance.trace()`.

`__init__`(*sn=None*, *Cn=None*, *reduced=False*, *Zc=None*)

**Parameters**

**sn: array or :class:'Profiles.GroupProfiles'** :

Poles of the acoustic resonator. If it is time-invariant, a list or array of N complex values (one per resonance) is sufficient. If not, provide a:class:*Profiles.GroupProfiles* instance (possible complex-valued).

**Cn: array or :class:'Profiles.GroupProfiles'** :

Residue of the input impedance of the acoustic resonator. See `sn` for explanation on format.

**Zc: float or Profile** :

The (possibly time-variant) characteristics impedance of the acoustic resonator. Optional if the residues are not dimensionless, as it is then only used to compute reflection coefficient.

**reduced: bool** :

Flag indicating whether residue value are dimensionless or not. If True, value of the characteristics impedance is mandatory and residus are then dimensioned according to Zc.

`__call__`(*s, t=[0.0]*)

Evaluates the expression $Z(s)$ for the specified values of `s`. The evaluation is done using the compiled function modal_impedance.

**Parameters**

**s** : array-like

List of values of Laplace variable at which transfer function is evaluated. If purely real valued, it is interpreted as a array of frequencies (and thus multiplied by $2\pi$)

**t** : scalar, optional

Instant to consider when evaluating the coefficients of the transfer function. Default is 0.

**Returns**

**Z** : list of frequency responses (one per instant required).

`trace`(*f=None, figs=None, linlog='lin', t=[0.0], reduced=False*)

Plots representations of the frequency response. The graphics properties can be modified directly on the returned object.

**Parameters**

**f** : array-like, optional

Frequencies at which frequency response is evaluated.

**figs** : list, optional

list of two Figure instances in which Z and R are shown.

**linlog** : string, optional

The modulus is displayed with a linear axis if `lin`, with a logarithmic axis if `log`, and an exception is raised otherwise.

**t** : scalar, optional

Instant to consider when evaluating the coefficients of the transfer function. Default is 0.

**Returns**

**Fig** : `matplotlib.Figure`

The figure object

`save(`*filename*`)`

Saves instance to file using pickle [pick]. See `load_impedance()` for loading.

`AcousticResonator.load_impedance(`*s*`)`

Load data saved with the `Time(Inv|V)ariantImpedance.save()` method:

**Parameters**

**filename: file-like object (file or string)** :

Name of the file to load.

**Returns**

**obj: TimeInvariantImpedance or TimeVariantImpedance** :

Stored Time(Inv|V)ariantImpedance

`AcousticResonator.restore(`*filename*`)`

Restore a ResonateurModesComplexes instance from file.

> **Warning:** Obsolete! Allow one to restore impedance created with previous versions of Moreesc

**Parameters**

**filename** : str or file

A handle for the file to load.

**Returns**

**Ze** : `TimeInvariantImpedance`

A `TimeInvariantImpedance` instance constructed with old-style.

## 3.3 Time-invariant version

The time-invariant counterpart is also available, simplifying syntax for user.

**class** `AcousticResonator.TimeInvariantImpedance`(*sn=None*, *Cn=None*, *reduced=False*, *Zc=None*)

Simplified class for time invariant acoustic resonator.

### Attributes

| | | |
|---|---|---|
| Zc | float | Characteristic impedance (required for reduced residues). |
| poles | complex array | Poles $s_n$ |
| residus | complex array | Residues $C_n$ associated to the poles |
| nbmodal | integer | Number of positives modes in the instance. |

`__init__`(*sn=None*, *Cn=None*, *reduced=False*, *Zc=None*)

**Parameters**

**sn: array** :

Poles of the acoustic resonator.

**Cn: array** :

Residue of the input impedance of the acoustic resonator.

**Zc: float** :

The characteristics impedance of the acoustic resonator. Optional if the residues are not dimensionless, as it is then only use to compute reflection coefficient.

**reduced: bool** :

Flag indicating whether residue value are dimensionless or not. If True, value of the characteristics impedance is mandatory and residus are then dimensioned according to Zc.

`save`(*filename*)

Saves instance to file using pickle [pick]. See `load_impedance()` for loading.

Using this class directly may be convenient when you are studying the effect of the frequency or the damping of a particular resonance peak. But manipulating poles and residues may not be right comfortable. Two functions are available to convert poles/residues pairs into natural frequency/damping/ amplitude tuple and reciprocally:

$$\frac{Z_n}{1 + \frac{j}{q_n}(\omega/\omega_n - \omega_n/\omega)} = \frac{jZ_n q_n \omega/\omega_n}{1 + jq_n\omega/\omega_n - \omega^2/\omega_n^2} = \frac{C_n}{j\omega - s_n} + \frac{C_n^*}{j\omega - s_n^*}$$

`AcousticResonator.snCn2wnqnZn`(*sn=None*, *Cn=None*)

Converts the tuple (`sn`, `Cn`) into (`wn`, `qn`, `Zn`).

**Parameters**

**sn** : complex or array-like

Pole (with positive imaginary part)

**Cn** : complex or array-like

Residue

**Returns**

**wn** : array

Natural angular frequency of the resonance

**qn** : array

Damping of the resonance (quality factor: $1/qn$)

**Zn** : complex array

Gain of the resonance (peak magnitude)

`AcousticResonator.wnqnZn2snCn`(*wn=None*, *qn=None*, *Zn=None*)

Converts the tuple (`wn`, `qn`, `Zn`) into (`sn`, `Cn`).

**Parameters**

**wn** : float or array

Natural angular frequency of the resonance

**qn** : float or array

Damping of the resonance (quality factor: $1/qn$)

**Zn** : complex or array-like

Gain of the resonance (peak magnitude)

**Returns**

**sn** : complex array

Pole (with positive imaginary part)

**Cn** : complex array

Residue

## 3.4 Cylindrical bore

A wrapper of `AcousticResonator.TimeInvariantImpedance` is provided. The class `AcousticResonator.Cylinder` defines a cylindrical resonator of length `L`, radius `r` and with the specified number of modes. The boundary conditions used to estimate the modes are

- Radiation impedance at the (fake) bell (see [Silva:2009])
- Neumann condition for pressure at mouthpiece end

**class** `AcousticResonator.Cylinder`(*L=1.0*, *r=0.007*, *radiates=True*, *nbmodes=10*, *losses='visco-thermal'*)

Bases: `AcousticResonator.TimeInvariantImpedance`

Model a cylindrical bore, possibly radiating.

> **Warning:** By now this configuration is static, i.e. some still need to be done to enable linear profiles (or more complex ones, like spline) for the attributes of the cylindre (i.e. length and radius). If such case are wanted, please consider defining an initial cylinder and a final one, and selecting the way the poles and residues are meant to evolve between these two states. Be aware that interpolated configurations may not correspond to the impedance of an intermediate length cylinder...

### Attributes

| | | |
|---|---|---|
| r | float | The inner radius of the cylindrical bore (in *m*) |
| L | float | The geometrical length (in *m*) |
| radiates | bool | Boolean indicating whether radiation from the open end is considered. |

`estimate_poles`(*S0=None*)

Try to locate the poles in the *s* plane. It uses `scipy.optimize.fsolve()` which needs initial guess.

**Parameters**

**S0** : array, optional

Initial guess for poles. If not specified, the default are the poles of the open-closed lossless cylindrical bore.

**Raises**

**ValueError** :

When the root finding is not successfull for one of the poles.

`eval_residues`(*x=0.0*, *xs=0.0*, *poles=None*)

Evaluates the residues of the previously estimated poles. Analytical expression of the residues is given in [Silva:PhD].

**Parameters**

**x, xs** : float, optional

positions of the observer and of the source. Default are 0, so that it computes the input impedance.

**poles** : array, optional

If not specified, the residus associated to each pole are computed.

**Returns**

**residues** : optional

If poles are selected, the methods outputs the associated residus.

## 3.5 Measured Impedance

Although it is still experimental (in the sense of *alpha-released*), it is possible to define an acoustic resonator with experimental data, i.e. a sampled measured input impedance obtained with an apparatus like the ones from CTTM (see [LeRoux:2008]) or the Acoustic Pulse Reflectometry (see [Sharp:PhD]).

**class** `AcousticResonator.MeasuredImpedance`(**args*, ***kwargs*)

Bases: `AcousticResonator.TimeInvariantImpedance`

### Attributes

| | | |
|---|---|---|
| frequences | array | The vector of frequency where the impedance has been evaluated |
| valeurs | array | The vector of values of the impedance |

`estimate_modal_expansion(**kwargs)`
Perform the estimation of a modal expansion of the loaded data.

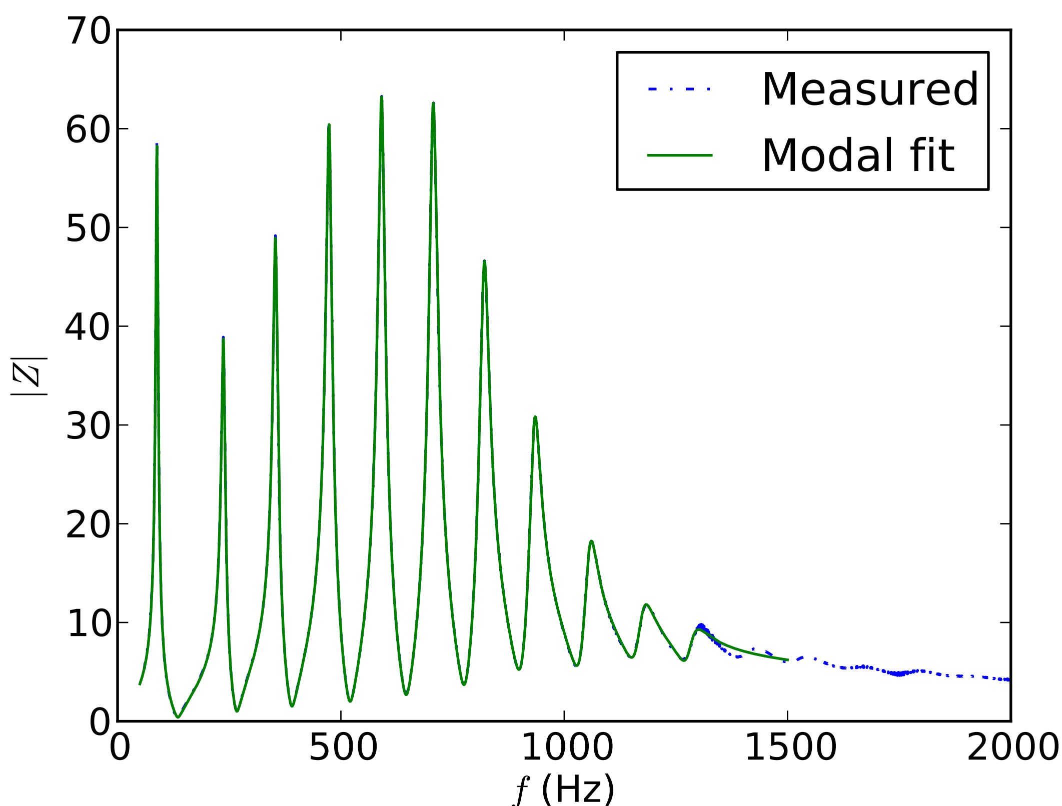
**Parameters**

**algorithm** : str 'Kennelly' or 'bruteforce'

Algorithm used to compute the modal expansion.

**kwargs** : passed to computational routines.

```
>>> import moreesc.AcousticResonator as mac
>>> Zc = mac.rho * mac.c / (np.pi * (8.4e-3)**2 )
>>> Z = mac.MeasuredImpedance(filename='../../data/ImpedanceBaptiste.mat', storage='mat_freqZ', fmin
>>> fapp = [87, 236, 352, 472, 590, 706, 820, 933, 1060, 1184, 1305, 1429]
>>> Z.estimate_modal_expansion(algorithm='bruteforce', lFapp=fapp)
# Data from measurement device is dimensionless here
>>> Z.residues *= Z.Zc
>>> Z.save('Impedance.h5')
>>> f = np.linspace(50., 1500., 2048)
>>> plt.plot(Z.frequencies, np.abs(Z.values), '-.', label='Measured')
>>> plt.plot(f, np.abs(Z(2.j * np.pi * f)/Zc), label='Modal fit')
>>> plt.legend()
>>> plt.xlabel(r'$f$ (Hz)'); plt.ylabel(r'$|Z|$'); plt.show()
```




> **Warning:** The autonomous modal expansion is still experimental and you should be careful and check that the result is correct

### 3.5.1 `ModalExpansionEstimation` – Tools for modelling real world data

Here are presented two ways to perform the modal expansion of a multiple-resonances transfer function.

#### Bruteforce optimization

The first one is to perform a bruteforce optimization on the values of coefficient, natural frequency and quality factor of each of the resonances required.

`ModalExpansionEstimation.bruteforce_optimization`(*freq*, *Z*, *lFapp=None*, *lQapp=None*, *lZapp=None*, *output_snCn=False*, *trace=False*, *optfun='complex'*, *fmin=None*, *fmax=None*)

Estimate the modal expansion of a frequency response with multiple resonances.

**Parameters**

**lFapp** : array-like

List of approximated resonances frequencies (open GUI if none).

**lZapp** : array-like

List of approximated resonances magnitudes (local max search if none).

**lQapp** : array-like

List of approximated resonances quality factors (from phase rotation if none).

**output_snCn** : bool

A flag indicating if the result of the optimization must be returned in poles-residus values (True) or frequency-damping-magnitude values (False, default).

**optfun** : str

You can either optimize upon the complex impedance values (use optfun='complex'):

$$\min J \quad \text{with} \quad J = \sum |\text{model} - \text{ref}|^2,$$

or only on the modulus of impedance (optfun='modulus'):

$$\min J \quad \text{with} \quad J = \sum (|\text{model}| - |\text{ref}|)^2.$$

**Returns**

**Zn, Fn, Qn** : three arrays

The coefficient, natural frequency and quality factor of the resonances

**sn, Cn** : two arrays

The estimated poles and the residues of the tranfer function.

### Kennelly's circle fitting

The other one is based on the fact that a Lorentz resonance is represented by a Kennelly's circle parametrized by the frequency in the complex plane. Even if this is theoretically true for a single ddof oscillator, multiple resonances frequency responses require a procedural fitting procedure. This is implemented here by:

`ModalExpansionEstimation.multiple_circles`(*freq*, *Z*, *lFapp=None*, *output_snCn=False*, *meth_C=None*, *meth_f=None*, *meth_seq=None*, *trace=False*)

Estimate the modal expansion of a frequency response with multiple resonances.

**Parameters**

**lFapp** : array-like

List of approximated resonances frequencies. It may be unsorted in order to mention a specific processing order.

**meth_seq** : str

It defines the sequential algorithm used to minimize the side effect of adjacent resonances when estimating a single peak's parameters:

- if it contains 'eliminate', then the previously estimated resonances are sequentially removed from the analyzed data;
- if it contains 'automagnitude', approximated guess will be sorted by associated magnitude values, and the algorithm will process resonances by decreasing values.

Combinations are possible, e.g. for example 'eliminate_automagnitude'

**output_snCn** : bool

A boolean defining whether the model coefficients (False) or the pole-residue pairs (True) are returned.

**For other parameters, see single_circle docstring.** :

**Returns**

**Zn, Fn, Qn** : three arrays

The coefficient, natural frequency and quality factor of the resonances

**sn, Cn** : two arrays

The estimated poles and the residues of the tranfer function.

These two functions required an initial estimation that can be provided by manually selecting peaks in the graphical visualisation of the curve, or by using the related input arguments. Algorithms for a single mode are described in [Brandon:1983] for circle fitting [LeRoux:PhD] for modal coefficients estimation.

## 3.6 References

# Simulation – Time-domain simulation

Pay attention to the various attributes created by the differents methods. They are exhibited within the functions which generate them.

## 4.1 Defining the problem

**class** `Simulation.TimeDomainSimulation`(*valve=None*, *resonator=None*, *fs=44100.0*, *integrator='vode'*, *kw_integrator=None*, *piecewise_constant=False*, ***kwargs*)

`TimeDomainSimulation` gathers all the informations about the configuration of the numerical experimentation.

### Attributes

| | | |
|---|---|---|
| valve | | A valve object |
| resonator | | An acoustic resonator object. |
| mouth_pressure, opening | `Profiles.Profile` | The time-varying control parameters. |
| fs | float | The apparent sampling frequency. |
| time_vector | array | The time vector. |
| X0 | array | The initial condition state vector $X(0)$. |
| Nx | int | The lenght of the state vector $X$ |
| Nac | int | Number of oscillating acoustical resonances. |

During the initialization of the instance, an initial state vector is computed corresponding to the static regime, may it be stable or not. This may be overridden after the initialization by calling the `conditions_initiales()` method

`__init__`(*valve=None*, *resonator=None*, *fs=44100.0*, *integrator='vode'*, *kw_integrator=None*, *piecewise_constant=False*, ***kwargs*)

Instantiate a new TimeDomainSimulation object given a Valve and a acoustical resonator. Additional arguments are

**Parameters**

**pm** : Profile

Mouth pressure profile

**h0** : Profile

Channel's opening at rest

**fs** : float

the (apparent) sampling frequency of the simulation

**integrator** : 'vode', 'dopri5', 'dop853', 'lsoda', 'Euler', 'EulerRichardson'

the name of the integrator used (see scipy.integrate.ode)

**kw_integrator** : dict

dictionnary of options to pass to the integrator

**piecewise_constant** : bool

flag specifying whether the control parameter are kept constant between two consecutive samples.

**Note that the last three parameters can easily be changed during the** :

**simulation (see the integrator attribute and its set_integrator method)** :

### Examples

```
>>> D  = moreesc.Valve.ReedDynamics(wr=6280., qr=0.4, kr=1e6, W=1.5e-2)
>>> Ze = moreesc.AcousticResonator.Cylinder(L=3., r=7e-3, radiates=False, nbmodes=50)
>>> pm = moreesc.Profiles.Linear(instants=[0., 1.], values=[0., 1e3])
>>> h0 = moreesc.Profiles.Constant(8e-4)
>>> sim = moreesc.Simulation.TimeDomainSimulation(valve=D, resonator=Ze, pm=pm, h0=h0)
```

`set_initial_state`(*X0=None*, **args*, ***kwargs*)

Initialize the state vector. If a vector `X0` is given, it is assigned to the initial conditions within the `X0` attribute. If not, the static state vector associated to the mouth pressure and the valve opening at instant $t = 0$ is evaluated.

**Parameters**

**X0** : array-like, optional

If specified, it must have the same shape as the state vector.

### Examples

```
>>> sim.set_initial_state(X0=np.ones(sim.Nx))
```

or

```
>>> sim.set_initial_state()
```

The differents situations apply

- `X0` is given. It may be a state vector coming from a previous simulation or come from a random process (it is then not in a steady state).
- `X0` is not given. The initial state corresponding to static equilibrium at time $t = 0$ is computed.

## 4.2 Configuring the solver and integrate

You may then run the computation of the trajectory

`TimeDomainSimulation.set_integrator(`*name*, ***kwargs*`)`

`TimeDomainSimulation.integrate(`*t=1.0*, *verbose=True*`)`

Solves the set of ordinary differential equations associated to the configuration using the solver `scipy.integrate.ode()`.

**Parameters**

**time** : float

Setting the time range over which integration is performed.

### Examples

```
>>> sim.integrate(1.)
```

You can interrupt the computation using CTRC-C. It will however update the following attributes according to the last step computed and extract signals (with the extract_signals method).

### Attributes

| time | array | Discrete time vector |
|---|---|---|
| result | 2D array | Raw data results. |
| label | str | A timestamp of the simulation |

## 4.3 Postprocessing

`TimeDomainSimulation.trace(`*tmin=None*, *tmax=None*, *trace_signals=True*, *trace_components=False*, *trace_spectrogram=False*, *trace_spectrums=False*, *trace_all=False*, *trace_instantaneous_frequency=False*, *fmax=5000.0*, *verbose=False*, ***kwargs*`)`

Plots several figures, with possible time range reduction

**Parameters**

**tmin, tmax** : float, optional

Lower and upper bounds of time range to display.

**trace_signals** : bool, optional

whether to plot a figure with pressure, opening and volume flow.

**trace_components** : bool, optional

whether to plot the first components of the pressure field.

**trace_spectrogram** : bool, optional

whether to plot a spectrogram of the pressure signal

**trace_spectrums** : bool, optional

whether to plot spectrums of pressure, opening and volume flow.

**trace_all** : bool, optional

whether to plot all the previously mentioned figures.

**fmax** : float, optional

Maximal frequency to display if `spec` is True.

`TimeDomainSimulation.play(`*where='in'*`)`

Plays the sound produced by the computations.

**Parameters**

**where** : str 'in' or 'out'

The signal to be played ('in' for mouthpiece pressure, 'out' for a pseudo-radiated presure).

`TimeDomainSimulation.save_wav(`*filename=None, fmt='wav', where='out', remove_peaks=False*`)`

Record the pressure into an audio file.

**Parameters**

**filename** : File or str

A description of file (will be over-written if existing).

**format** : str or list of strings

An audio format suitable to audiolab Format class

**where** : str 'in' or 'out'

The signal used to create the wav file ('in' for mouthpiece pressure, 'out' for a pseudo-radiated presure).

**remove_peaks** : bool

Whether to remove peaks

`TimeDomainSimulation.extract_signals()`

Extract the signals of mouthpiece pressure, volume flow and tip opening from the raw solution of the set of ODE. An estimation of the radiated pressure is also computed.

They are then available with through the following attributes

### Attributes

| | | |
|---|---|---|
| pressure | array | The total pressure in the mouthpiece. It is the sum of the components. |
| pm | array | The excitation pressure in the mouth. |
| flow | array | The volume flow entering through the reed channel. |
| opening | array | The varying tip opening. |
| h0 | array | The tip opening at rest. |
| external_pressure | array | An approximation of the radiated sound pressure. |

`TimeDomainSimulation.get_instantaneous_frequency(`*mode='yin', **kwargs*`)`

Get the instantaneous frequency from the pressure signal using Aubio. To recompute using another mode, delete the f_i attribute.

`TimeDomainSimulation.reconstruct_spatial_field(`*vectors, decimate=1*`)`

Reconstructs the time-evolving pressure field within the resonator from a simulation result and knowing the eigenvectors related to the poles of the acoustic resonator.

**Parameters**

**vectors** : array-like

The list of the (possibly complex) eigenvectors Shape: self.Nac x number of nodes

**decimate** : int, optional

A decimation factor for sampling frequency.

**Returns**

**Ptx** : 2d array

The array containing the pressure value for each spatial point and time instant (time signals as columns, instantaneous pictures as rows)

## 4.4 Data persistence

`TimeDomainSimulation.save(`*filename=None*`)`

Saves instance to file using pickle [pick]. See `load_impedance()` for loading.

### Examples

```
>>> sim.save('/tmp/simulation.dat')
>>> sim = moreesc.Simulation.load_simulation('/tmp/simulation.dat')
```

`Simulation.load_simulation(`*s*`)`

Load data saved with the `TimeDomainSimulation.save()` method:

**Parameters**

**filename: file-like object (file or string)** :

Name of the file to load.

**Returns**

**obj: TimeDomainSimulation** :

Stored TimeDomainSimulation

CHAPTER 5

# Indices and tables

- *modindex*
- *genindex*
- *search*

# Bibliography


[BSplines] <http://en.wikipedia.org/wiki/B-spline>_

[Bezier] <http://en.wikipedia.org/wiki/Bezier_curve>_

[NPZ] <http://docs.scipy.org/numpy/docs/numpy.lib.format/>_

[spl] <http://docs.scipy.org/doc/scipy/reference/generated/scipy.interpolate.splprep.html>_

[pick] <http://docs.python.org/library/pickle.html>_

[Silva:PhD] Émergence des auto-oscillations dans les instruments de musique à anche simple, F. Silva, Université Aix-Marseille, 2009.

[Silva:2009] Approximation of the acoustic radiation impedance of a cylindrical pipe, F. Silva, Ph. Guillemain, J. Kergomard, B. Mallaroni & A. Norris, Journal of Sound and Vibrations 322(1-2), pp. 255-263, 2009

[LeRoux:2008] A new impedance tube for large frequency band measurement of absorbing materials, J.C. Le Roux, J.-P. Dalmont and B. Gazengel, Acoustics'08, Paris.

[Sharp:PhD] Acoustic pulse reflectometry for the measurement of musical wind instruments, David B. Sharp, Edinburgh University, 1996.

[LeRoux:PhD] **Le haut-parleur électrodynamique: estimation des paramètres électroacoustiques aux basses fréquences et modélisation de la suspension**, J.-Ch. Le Roux, Université du Maine, 1994.

[Brandon:1983] A weighted least squares method for circle fitting to frequency response data, J. A. Brandon and A. Cowley, Journal of Sound and Vibration 89(3), pp. 419-424, 1983.

# Python Module Index


a

AcousticResonator, 12

m

ModalExpansionEstimation, 20

p

Profiles, 1

s

Simulation, 22

v

Valve, 8

# Python Module Index


a

AcousticResonator, 12

m

ModalExpansionEstimation, 20

p

Profiles, 1

s

Simulation, 22

v

Valve, 8

# Index

## Symbols


__call__() (AcousticResonator.TimeVariantImpedance method), 15
__call__() (Profiles.GroupProfiles method), 5
__call__() (Profiles.Profile method), 4
__call__() (Valve.TransferFunction method), 9
__init__() (AcousticResonator.TimeInvariantImpedance method), 17
__init__() (AcousticResonator.TimeVariantImpedance method), 15
__init__() (Profiles.Signal method), 8
__init__() (Simulation.TimeDomainSimulation method), 23
__init__() (Valve.TransferFunction method), 9


## A


AcousticResonator (module), 12
AcousticResonator.radiation_a (in module AcousticResonator), 14
AcousticResonator.radiation_b (in module AcousticResonator), 14


## B


bruteforce_optimization() (in module ModalExpansionEstimation), 21


## C


c (in module AcousticResonator), 13
Constant (class in Profiles), 5
Cp (in module AcousticResonator), 13
Cpv (in module AcousticResonator), 14
Cv (in module AcousticResonator), 14
Cylinder (class in AcousticResonator), 18


## E


editor() (Profiles.Spline method), 6
estimate_modal_expansion() (AcousticResonator.MeasuredImpedance method), 19
estimate_poles() (AcousticResonator.Cylinder method), 19
eval_residues() (AcousticResonator.Cylinder method), 19
extract_signals() (Simulation.TimeDomainSimulation method), 26


## F


fit_spline() (Profiles.Signal method), 8


## G


get_instantaneous_frequency() (Simulation.TimeDomainSimulation method), 27
get_poles() (Valve.TransferFunction method), 10
get_zeros() (Valve.TransferFunction method), 10
GroupProfiles (class in Profiles), 5


## I


integrate() (Simulation.TimeDomainSimulation method), 25


## K


kappa (in module AcousticResonator), 14

## L

Linear (class in Profiles), 6
LipDynamics (class in Valve), 12
load_groupprofiles() (in module Profiles), 5
load_impedance() (in module AcousticResonator), 16
load_profile() (in module Profiles), 4
load_simulation() (in module Simulation), 27
lt (in module AcousticResonator), 14
lv (in module AcousticResonator), 14

## M

MeasuredImpedance (class in AcousticResonator), 19
ModalExpansionEstimation (module), 20
multiple_circles() (in module ModalExpansionEstimation), 22

## N

nu (in module AcousticResonator), 14

## O

OneDOFOscillator (class in Valve), 11

## P

physical_constants() (in module AcousticResonator), 14
play() (Simulation.TimeDomainSimulation method), 26
Profile (class in Profiles), 3
Profiles (module), 1

## R

reconstruct_spatial_field() (Simulation.TimeDomainSimulation method), 27
ReedDynamics (class in Valve), 12
restore() (in module AcousticResonator), 16
rho (in module AcousticResonator), 13

## S

save() (AcousticResonator.TimeInvariantImpedance method), 17
save() (AcousticResonator.TimeVariantImpedance method), 16
save() (Profiles.GroupProfiles method), 5
save() (Profiles.Profile method), 4
save() (Simulation.TimeDomainSimulation method), 27
save_wav() (Simulation.TimeDomainSimulation method), 26
set_initial_state() (Simulation.TimeDomainSimulation method), 24
set_integrator() (Simulation.TimeDomainSimulation method), 25
Signal (class in Profiles), 7
Simulation (module), 22
snCn2wnqnZn() (in module AcousticResonator), 17
Spline (class in Profiles), 6

## T

TimeDomainSimulation (class in Simulation), 23
TimeInvariantImpedance (class in AcousticResonator), 16
TimeVariantImpedance (class in AcousticResonator), 14
trace() (AcousticResonator.TimeVariantImpedance method), 15
trace() (Simulation.TimeDomainSimulation method), 25
trace() (Valve.TransferFunction method), 10
TransferFunction (class in Valve), 9

## V

Valve (module), 8

## W

wnqnZn2snCn() (in module AcousticResonator), 18