Documentation of the OTPMML module

Documentation built from package otpmml-1.5

April 10, 2018
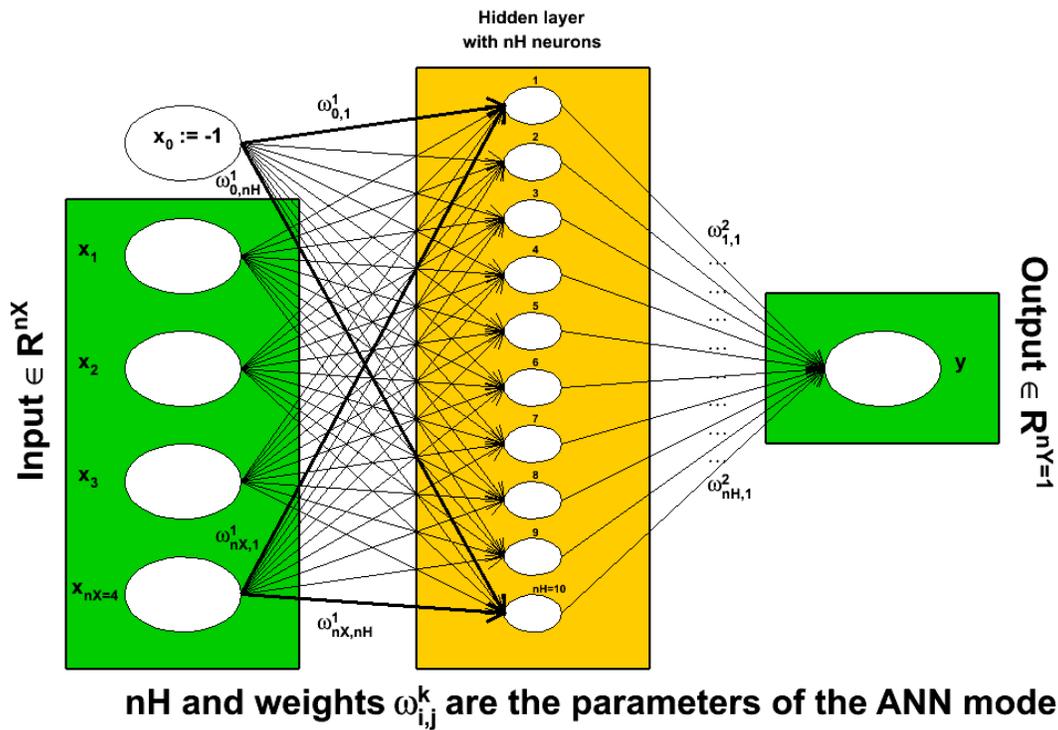
Image of an artificial neural network, copied from URANIE documentation

Abstract

The purpose of this document is to present the OTPMML module.
This document is organized according to the OpenTURNS documentation :

- *Architecture Guide gives UML diagrams of all implemented classes,*

- *Reference Guide gives some theoretical basis,*

- *Use cases Guide details scripts in Python (the Textual Interface language of OpenTURNS) and helps to learn as quickly as possible the manipulation of the otpmml module,*

- *User Manual details the otpmml objects and give the list of their methods,*

- *Validation Guide which provides use cases to validate the otpmml module.*

Contents

1 Architecture guide

This document makes up the general specification design for the architecture of the OTPMML module.

1.1 DAT

```
                         DAT
+Import(in datFile:FileName): Collection<NumericalSample>
+Export(in datFile:FileName,in inputOutput:NumericalSample): void
+Export(in datFile:FileName,in input:NumericalSample,
        in output:NumericalSample): void
```
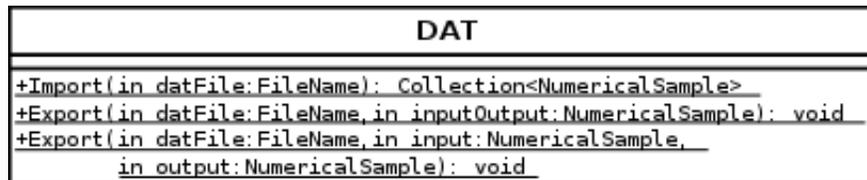
Figure 1: DAT class

The `DAT` class is a utility class to import and export Uranie `.dat` files. It contains only static methods.

1.2 NeuralNetwork

```
                    OT::NumericalMathFunction
                              △
PMMLDoc                       |
                         NeuralNetwork
PMMLNeuralNetwork
              +NeuralNetwork(in pmmlFile:FileName,in modelName:String="")
```
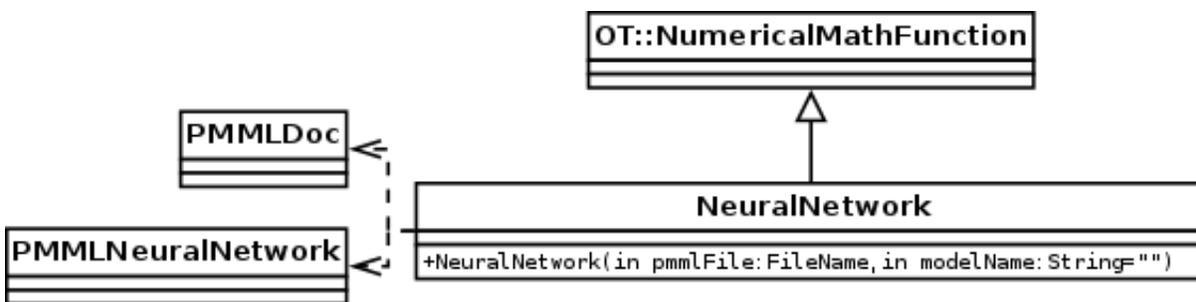
Figure 2: NeuralNetwork class

The `NeuralNetwork` class inherits from `NumericalMathFunction`. It uses `PMMLDoc` and `PMMLNeuralNetwork` internal classes to parse `.pmml` files written by Uranie.

1.3 RegressionModel

```
PMMLDoc                  RegressionModel
          -metamodel_: Pointer<LinearLeastSquares>
          +RegressionModel(in pmmlFile:FileName,in modelName:String="")
          +RegressionModel(in implementation:LinearLeastSquares)
PMMLRegressionModel
          +getLinearLeastSquares(): LinearLeastSquares const
          +exportToPMMLFile(in pmmlFile:FileName): void const
```
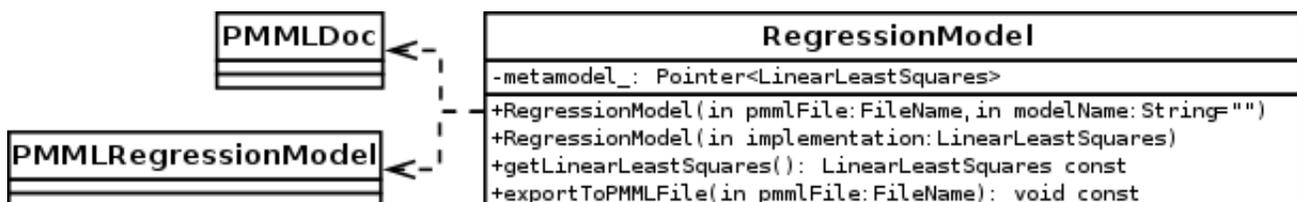
Figure 3: RegressionModel class

The `RegressionModel` class is a wrapper around `RegressionModel` XML elements found in `.pmml` files. It uses `PMMLDoc` and `PMMLRegressionModel` internal classes.

1.4 PMML Internal Classes

1.4.1 PMMLDoc

```
+----------------------------------------------------------------+
|                            PMMLDoc                             |
+----------------------------------------------------------------+
| -document_: xmlDocPtr                                          |
| -rootNode_: xmlNodePtr                                         |
| -xpathContext_: xmlXPathContextPtr                            |
| -xpathNsPrefix_: String                                       |
+----------------------------------------------------------------+
| +PMMLDoc()                                                     |
| +PMMLDoc(in pmmlFile:FileName)                                |
| +~PMMLDoc()                                                    |
| -checkInitialized(): void const                               |
| -getModelNames(in category:String): Collection<String> const  |
| -getXPathQueryScalar(in xpathQuery:String): float const       |
| -getXPathQueryString(in xpathQuery:String): String const      |
| +read(in pmmlFile:FileName): bool                             |
| +write(in pmmlFile:FileName): bool const                      |
| +getNumberOfNeuralNetworks(): int const                       |
| +getNeuralNetworkModelNames(): Collection<String> const       |
| +getNeuralNetwork(in modelName:String=""): PMMLNeuralNetwork const |
| +getNumberOfRegressionModels(): int const                     |
| +getRegressionModelNames(): Collection<String> const          |
| +getRegressionModel(in modelName:String=""): PMMLRegressionModel cor |
| +addRegressionModel(in modelName:String,                      |
|                     in regression:LinearLeastSquares): void   |
| +addHeader(): void                                            |
+----------------------------------------------------------------+
```

Figure 4: PMMLDoc class

The `PMMLDoc` class reads an XML file (in the PMML 3.0 format, which is the format used by Uranie), and provides several methods used by `PMMLNeuralNetwork` and `PMMLRegressionModel` classes. It internally uses LibXML2 library to handle XML data; this library must be initialized before parsing XML data, and memory must be explicitly deallocated when its job is over. For this reason, it had been decided to not expose `PMMLDoc` to the Python interface. Calls to `xmlInitParser` and `xmlCleanupParser` are performed by higher-level classes `NeuralNetwork` and `RegressionModel`.

XPath is used extensively to extract informations from XML data; two methods, `getXPathQueryScalar` and `getXPathQueryString`, are provided for simple usages. `PMMLNeuralNetwork` and `PMMLRegressionModel` classes are declared `friend` so that they can use these methods.

The `xpathContext_` member stores the current XPath context, and is modified by `setXPathContext` methods of `PMMLNeuralNetwork` and `PMMLRegressionModel` classes to point to their respective XML elements.

There are some caveats with XML namespaces when using XPath. In order to parse XML files with or without namespaces, an `xpathNsPrefix_` member has been added.

Note: In `addRegressionModel`, the `LinearLeastSquares` argument cannot be passed as a const reference due to a bug which had been fixed only in OpenTURNS 1.5.

1.4.2   PMMLRegressionModel

```
                    PMMLRegressionModel
  -pmml_: const PMMLDoc*
  -modelName_: const String
  -node_: const xmlNodePtr
  +PMMLRegressionModel(in pmml:PMMLDoc*,in modelName:String,
                             in node:xmlNodePtr)
  -setXPathContext(): void
  -checkValid(): void const
  +getModelName(): String const
  +getIntercept(): float const
  +getTargetVariableName(): String const
  +getCoefficients(): NumericalSample const
```

Figure 5: PMMLRegressionModel class

The `PMMLRegressionModel` class is straightforward. The `checkValid` private method ensures that this regression model can be mapped to a `LinearLeastSquares` instance. The following checks are performed:

- `modelType` attribute, if present, must be equal to `linearRegression`
- `functionName` attribute must be equal to `regression`
- `normalizationMethod` attribute must be equal to `none`
- there must be only one `RegressionTable` child element
- this `RegressionTable` must contain only `NumericPredictor` children and not `CategoricalPredictor`
- `exponent` attributes of `NumericPredictor` must be equal to 1

1.4.3   PMMLNeuralNetwork

```
                     PMMLNeuralNetwork
  -pmml_: const PMMLDoc*
  -modelName_: const String
  -node_: const xmlNodePtr
  +PMMLNeuralNetwork(in pmml:PMMLDoc*,in modelName:String,
                          in node:xmlNodePtr)
  -setXPathContext(): void
  +getModelName(): String const
  +getNumberOfInputs(): int const
  +getNumberOfOutputs(): int const
  +getNumberOfLayers(): int const
  +getLayerSize(in index:int): int const
  +getBiasAtLayer(in index:int): NumericalPoint const
  +getWeightsAtLayer(in index:int): Matrix const
  +getNeuronIdsAtLayer(in index:int): Indices const
  +getNeuralInputName(in id:int): String const
  +getActivationFunctionAtLayer(in index:int): String const
  +getEvaluationFunctionAtLayer(in index:int): NumericalMathFunction const
  +getInputsNormalization(): NumericalSample const
  +getOutputsNormalization(): NumericalSample const
  +getInputsNormalizationFunction(): NumericalMathFunction const
  +getOutputsNormalizationFunction(): NumericalMathFunction const
```

Figure 6: PMMLNeuralNetwork class

The `PMMLNeuralNetwork` class parses XML data and provides accessor methods which are used by `NeuralNetwork`. Most functions could be private, they are public only to help testing and debugging.

Any valid `NeuralNetwork` XML element should be supported. There may be any number of hidden layers, neural layers may have any number of neurons, layers may be sparse, all known activation functions are supported. The only restriction is with normalization of inputs and outputs, piecewise normalization is not supported. This should not be a problem since Uranie stores PMML files with this format.

Note: `NumericalMathFunction` is built from string formulas. In order to preserve accuracy, 20 digits are used. The drawback is that pretty-printing looks uglier than with less digits.

2   Reference Guide

The OTPMML library provides an interface to the `URANIE` platform. In particular, it can:

- read artificial neural networks generated by `URANIE` and transform them into a `NumericalMathFunction` which can be used by OpenTURNS algorithms

- read inputs and outputs generated by `URANIE`

- write inputs and outputs in `URANIE` format

- convert regression model generated by `URANIE` into `LinearLeastSquares` instances, in both directions

2.1   Neural network

Mathematical description

<u>Goal</u>
The aim is to build an artificial neural network (often called neural network) function issued from the `URANIE` platform. This mathematical model is inspired by biological neural networks.

<u>Principles</u>
A single-layer perceptron is given by $n + 1$ data and an activation function:

- $w \in \mathbb{R}^n$ a vector of weights;

- θ a real number, called bias;

- ϕ: activation function, i.e. the rule. Formally, $\phi : \mathbb{R} \to [0, 1]$

The expression of neural network model, evaluated on the point of interest $x \in \mathbb{R}^n$, is given by:

$$\mathcal{M}(x) = \phi(w^T x - \theta) \tag{1}$$

$$= \phi\left(\sum_{i=1}^{n} w_i x_i - \theta\right) \tag{2}$$

Usually, the notation adopted is:

$$\mathcal{M}(x) = \phi\left(\sum_{i=1}^{n+1} w_i x_i\right) \tag{3}$$

with $w_{n+1} = \theta, x_{n+1} = -1$. This model is referred as single layer and described hereafter.
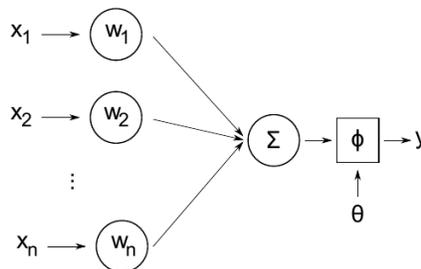
The activation function has to be fixed. The Heaviside function is commonly used and defined by:

$$H(x) = \begin{cases} 0 & x < 0 \\ 1 & x \geq 0 \end{cases}$$

This function is however discontinuous and thus implies more complexity during the learning step. In machine learning, several functions are commonly used:

- The logistic function: $\sigma(x) = \dfrac{1}{1 + e^{-x}}$

- The hyperbolic tangent function: $\tanh(x) = \dfrac{e^x - e^{-x}}{e^x + e^{-x}}$

These functions are implemented in the module.

A multi-layer perceptron consists of multiple layers of perceptrons, each neuron in a layer being connected to neurons of the previous layer. Each layer also contains an activation function (which has the same properties as with single-layer perceptrons) and a bias. The most commonly used multi-layer perceptron is the 2-layer perceptron:
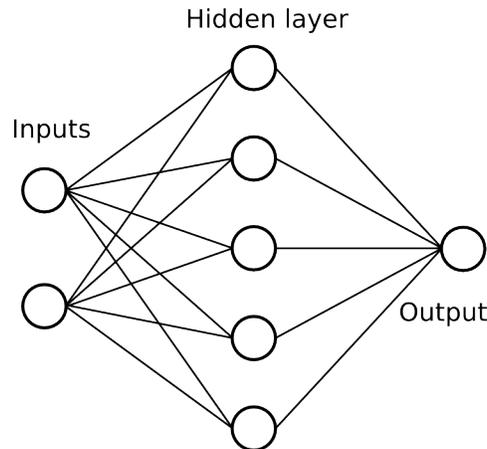
If we note (w, θ, ϕ) (resp. (w^H, θ^H, ϕ^H)) weights, bias and activation function of the output layer (resp. hidden layer), then the expression of this neural network model, evaluated on the point of interest $x \in \mathbb{R}^n$, is given by:

$$\begin{aligned} \mathcal{M}(x) &= \phi \left(\sum_{j=1}^{M} w_j \mathcal{M}_j^H(x) - \theta \right) \\ &= \phi \left[\sum_{j=1}^{M} w_j \phi^H \left(\sum_{k=1}^{M^H} w_k^H x_k - \theta^H \right) - \theta \right] \end{aligned}$$

This can be generalized to any number of layers, but in practice a single hidden layer is sufficient.

Other notations

Link with OpenTURNS methodology

This method is aimed at building a response surface prior to tackling Step C "Uncertainty Propagation". It requires an experimental design together with the corresponding model evaluations.

References and theoretical basics

- Christopher M. Bishop 1995, "Neural Networks for Pattern Recognition", Oxford University Press, Inc, New York.

2.2 Regression model

Mathematical description

<u>Goal</u>

The objectif is to evaluate a linear model regression issued from the `URANIE` platform.

<u>Principles</u>

One considers global approximations of the model response using a polynomial of degree one:

$$y \approx \widehat{h}(\underline{x}) = a_0 + \sum_{i=1}^{n_X} a_i \, x_i$$

where $(a_j\,, j = 0, \ldots, n_X)$ is a set of unknown coefficients.

Several techniques are used to estimate the coefficients of the model.
In the propagation context, an experimental design $\underline{\mathcal{X}} = (x^{(1)}, \ldots, x^{(N)})$, i.e. a set of realizations of input parameters is required, as well as the corresponding model evaluations $\underline{\mathcal{Y}} = (y^{(1)}, \ldots, y^{(N)})$. Thus the coefficients a_j may be computed using a least squares regression approach.

The following minimization problem has to be solved:

$$\text{Find} \quad \widehat{\underline{a}} \quad \text{that minimizes} \quad \mathcal{J}(\underline{a}) = \sum_{i=1}^{N} \left(y^{(i)} - a_0 - \sum_{j=1}^{n_X} a_j \underline{x_j}^{(i)} \right)^2$$

A necessary condition is that the size N of the experimental design is not less than the number $n_X + 1$ of coefficients to estimate.

Other notations

Link with OpenTURNS methodology

Within the global methodology, the method is used to assess the accuracy of a polynomial response surface of a model output prior to tackling the step C: ¡¡Uncertainty propagation¿¿.

References and theoretical basics

- Å. Bjorck, 1996, "Numerical methods for least squares problems", SIAM Press, Philadelphia, PA.

3 Use Cases Guide

This section presents the main functionalities of the module *otpmml* in their context.

3.1 NeuralNetwork import

Class `NeuralNetwork` imports a neural network model from a `PMML` file.
Python script for this use case :

```python
import openturns as ot
from otpmml import NeuralNetwork

# Import the neural network
neural_network = NeuralNetwork("myPMMLFile.pmml")
print (neural_network)
```

3.2 RegressionModel import

Class `RegressionModel` imports a regression model from a `PMML` file.
Python script for this use case :

```python
import openturns as ot
from otpmml import RegressionModel

# Import the model
model = RegressionModel("myPMMLFile.pmml")
print (model)

# LinearLeastSquares accessor
linearLeastSquares = model.getLinearLeastSquares()

# Export model to Pmml file
model.exportToPMMLFile("linearModel.pmml")
```

3.3 Import and export of DAT files

Class `IO` exports both input/output samples into a `DAT` file
Python script for this use case :

```python
import openturns as ot
import otpmml.DAT as DAT
from math import pi, sin

a = 7.0
b = 0.1
# Create the Ishigami function
input_variables = ["xi1","xi2","xi3"]
formula = ["sin(xi1) + (" + str(a) + ") * (sin(xi2)) ^ 2 + (" + str(b) +
    ") * xi3^4 * sin(xi1)"]
```

```
model = ot.NumericalMathFunction(input_variables, formula)
model.setName("Ishigami")
# Generating
dist = ot.ComposedDistribution(3 *[ot.Uniform(-pi,pi)])
X = dist.getSample(100)
Y = model(X)
# Export to DAT file
DAT.Export("data.dat",X, Y)


# Import DAT file
sampleCollection = DAT.Import("data.dat")
input_sample = sampleCollection[0]
output_sample = sampleCollection[1]
```

Export could be done also using only one sample.
Python script for this use case :

```
import openturns as ot
import otpmml.DAT as DAT
from math import pi, sin

a = 7.0
b = 0.1
# Create the Ishigami function
input_variables = ["xi1","xi2","xi3"]
formula = ["sin(xi1) + (" + str(a) + ") * (sin(xi2)) ^ 2 + (" + str(b) +
    ") * xi3^4 * sin(xi1)"]
model = ot.NumericalMathFunction(input_variables, formula)
model.setName("Ishigami")
# Generating
dist = ot.ComposedDistribution(3 *[ot.Uniform(-pi,pi)])
X = dist.getSample(100)
Y = model(X)
# Encapsulating both samples
Z = ot.NumericalSample(X)
Z.stack(Y)
# Export to DAT file
DAT.Export("data.dat",Y)
```

4   User Manual

This section gives an exhaustive presentation of the objects and functions provided by the *otpmml* module, in the alphabetic order.

4.1   DAT

This class is used through its static methods in order to import/export a *NumericalSample* or a collection of *NumericalSample* into a `.dat` file.

Methods:

    Import

        Usage:

            DAT.Import(filename)

        Arguments:

            filename: a string, file that contains data

        Value: a collection of samples of size 2. First sample corresponds to input data, second one to output data.

    Export

        Usage:

            DAT.Export(filename, input, output)

            DAT.Export(filename, inputOutput)

        Arguments:

            filename: a string, file where to export data.

            input: a *NumericalSample*, input sample, usually of dimension ≥ 1.

            output: a *NumericalSample*, output sample, usually of dimension 1.

            inputOutput: a *NumericalSample*, usually of dimension ≥ 1.

        Value: None.

4.2   NeuralNetwork

The class inherits from the *NumericalMathFunction* class.

Usage:

    NeuralNetwork(pmmlFile)

Arguments:

    pmmlFile: a string, PMML file that countains the neural network

Value: a NeuralNetwork, a *NumericalMathFunction* that implements neural network

Details:

    NeuralNetwork constructor

Links

4.3 RegressionModel

Usage:

> *RegressionModel(pmmlFile)*
>
> *RegressionModel(linearLeastSquares)*

Arguments:

> *pmmlFile*: a string, PMML file that countains the regression model
>
> *linearLeastSquares*: a LinearLeastSquare, object encapsulating least squares.

Value: a RegressionModel

Details:

> With the first usage, the class loads a model implemented in `PMML` format
>
> With the second usage, the class encapsulates a LinearLeastSquares attribut

getLinearLeastSquare

> **Usage:** *getLinearLeastSquare()*
>
> **Arguments:** no argument
>
> **Value:** a *LinearLeastSquare*

exportToPMMLFile

> **Usage:** *exportToPMMLFile(filename)*
>
> **Arguments:** *filename*, a string. Name of file for the export of regression model.
>
> **Value:** none.

5 Validation

This section aims at exposing the methodology used to validate numerical results of the module. The validation of `NeuralNetwork` is exposed hereafter.

For that purposes, the ¡¡beam¿¿ example, illustrated in the `ExampleGuide` documentation of `OpenTURNS`, has been choosen:

- A design of experiment and the evaluation of the deviation funcion on that design were provided (`input_output.dat` file);

- Previous data were used to build a neural network model thanks to the `Uranie` module;

- The issued model (`PMML` file) is provided for validation.

In addition, the `modulePMML.py` script had been developed at EDF to parse PMML files. Thus the validation of the neural network parsing uses a script that reads input data from `input_output.dat` file, evaluates the same PMML file with `modulePMML.py` and `OTPMLL`, gets output values, absolute and relative errors.

```python
import openturns as ot
import modulePMML
import otpmml

inputs, outputs = tuple(otpmml.DAT.Import("input_output.dat"))
size = len(inputs)

model = modulePMML.monModelePMML("uranie_ann_poutre.pmml")
pmmlRef = ot.NumericalMathFunction(model)
pmmlOT = otpmml.NeuralNetwork("uranie_ann_poutre.pmml")
eval_module = pmmlRef(inputs)
eval_otpmml = pmmlOT(inputs)

results = ot.NumericalSample(0, 4)
description = ["modulePMML", "OTPMML", "abs. error", "rel. error"]
results.setDescription(description)
for i in xrange(size):
    point = ot.NumericalPoint(inputs[i])
    results.add([ \
            eval_module[i][0], \
            eval_otpmml[i][0], \
            abs(eval_module[i][0] - eval_otpmml[i][0]), \
            abs((eval_module[i][0] - eval_otpmml[i][0])/eval_module[i][0]) \
                ])
print results
```

From that results, we compare in figure 7 outputs issued from `modulePMML.py` and `OTPMML`.
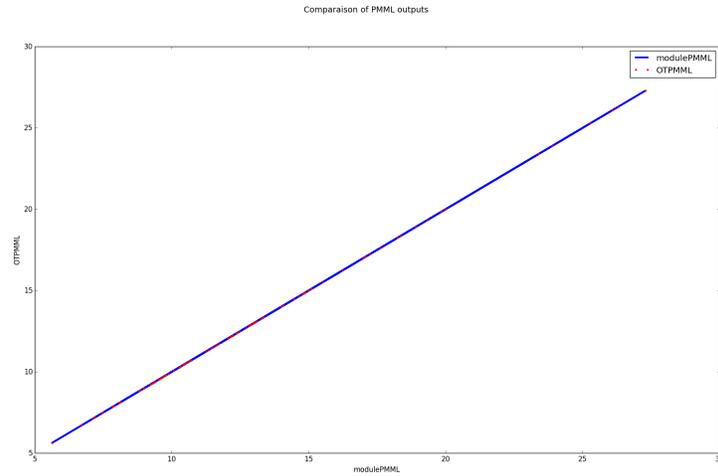
Figure 7: Comparison of outputs

In addition, errors are ploted in figure 8. Comparisons are very good, absolute error is less than 1.5×10^{-14}
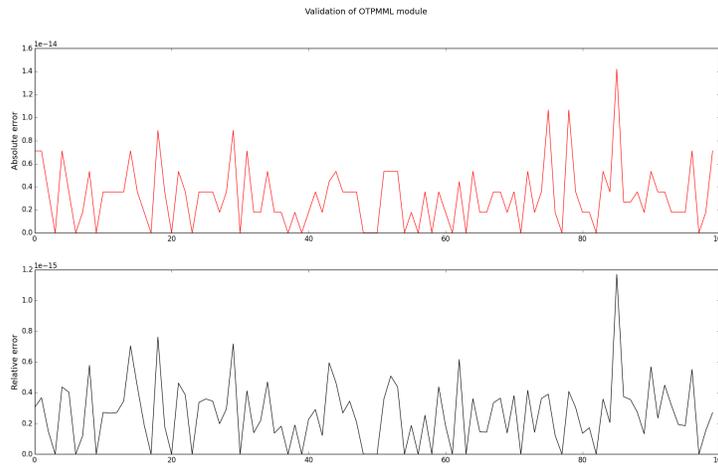
Figure 8: Comparison of outputs

and relative errors less than 1.2×10^{-15} , which validate the parsing.