pyProximation Documentation

Release 1.2.0

Mehdi Ghasemi

Oct 17, 2016

Contents:

CHAPTER

ONE

INTRODUCTION

This is a brief documentation for using *pyProximation*. pyProximation was originally written to solve systems of integro-differential equations via collocation method in arbitrary number of variables. The current implementation of the method is based on a finite dimensional orthogonal system of functions.

Requirements and dependencies

Since this is a python package, so clearly one needs have python available. pyProximation relies on the following packages:

- **for numerical calculation:**
  - numpy,
  - scipy ,
- **for symbolic computation, either:**
  - sympy or,
  - sage or,
  - symengine,
- **for visualization:**
  - mathplotlib,
  - mayavi.

For the symbolic computation, pyProximation need the presence of either *sympy* or *sage*.

Download

pyProximation can be obtained from https://github.com/mghasemi/pyProximation.

Installation

To install *pyProximation*, run the following in terminal:

```
sudo python setup.py install
```

Documentation

The documentation of *pyProximation* is prepaqred via sphinx.

License

pyProximation is distributed under MIT license:

MIT License

CHAPTER

TWO

SYMBOLIC TOOLKITS

The 'pyProximation' library performs symbolic computations as well as numerical calculations. To perform numerical calculations, it relies on the excelent libraries 'NumPy' and 'SciPy'. There are various toolkits that are able to handel symbolic computations and each one is suitable for certain purposes. 'pyProximation' takes advantage of three different toolkits:

1. *sympy*
2. *sage*
3. *symengine*

SymPy

SymPy is a Python library for symbolic computation. It provides computer algebra capabilities either as a standalone application, as a library to other applications, or live on the web as SymPy Live or SymPy Gamma. SymPy is trivial to install and to inspect because is written entirely in Python with few dependencies. This ease of access combined with a simple and extensible code base in a well known language make SymPy a computer algebra system with a relatively low barrier to entry.

SymPy includes features ranging from basic symbolic arithmetic to calculus, algebra, discrete mathematics and quantum physics. It is capable of formatting the result of the computations as LaTeX code.

SymPy is free software and is licensed under New BSD License. The lead developers are Ondřej Čertík and Aaron Meurer.

Usage

Sympy can be imported by:

```
from sympy import *
```

To introduce a symbolic variable called x, use:

```
x = Symbol('x')
```

and to define a symbolic function y depending in symbolic variables x, t, use:

```
y = Function('y')(x, t)
```

To introduce an equation like $y(x,t) = xt$, use *Eq*:

```
equation = Eq(y, x*t)
```

For differentiation of *y* with respect to *x* of order *n*, i.e. $\frac{\partial^n y}{\partial x^n}$, enter:

```
diff(y, x, n)
```

For integration of *y* with respect to *x*, i.e., $\int y dx$, write:

```
integrate(y, x)
```

and for definite integral $\int_a^b y dx$ write:

```
integrate(y, (x, a, b))
```

Sage

SageMath (previously Sage or SAGE, System for Algebra and Geometry Experimentation) is a free open-source mathematics software system licensed under the GPL with features covering many aspects of mathematics, including algebra, combinatorics, numerical mathematics, number theory, and calculus. It builds on top of many existing open-source packages: NumPy, SciPy, matplotlib, Sympy, Maxima, GAP, FLINT, R and many more. Access their combined power through a common, Python-based language or directly via interfaces or wrappers.

Usage

When present, Sage can be imported by:

```
from sage.all import *
```

To introduce a symbolic variable called *x*, use:

```
x = var('x')
```

and to define a symbolic function *y* depending in symbolic variables *x*, *t*, use:

```
y = function('y')(x, t)
```

To introduce an equation like $y(x,t) = xt$, use ==:

```
equation = (y == x*t)
```

For differentiation of *y* with respect to *x* of order *n*, i.e. $\frac{\partial^n y}{\partial x^n}$, enter:

```
diff(y, x, n)
```

For integration of *y* with respect to *x*, i.e., $\int y dx$, write:

```
integral(y, x)
```

and for definite integral $\int_a^b y dx$ write:

```
integral(y, (x, a, b))
```

SymEngine

SymEngine is a standalone fast C++ symbolic manipulation library. Optional thin wrappers allow usage of the library from other languages, e.g.:

- C wrappers allow usage from C, or as a basis for other wrappers;
- Python wrappers allow easy usage from Python and integration with SymPy and Sage;
- Ruby wrappers;
- Julia wrappers;
- Haskell wrappers.

It is licensed under MIT license.

Usage

SymEngine can be imported by:

```
from symengine import *
```

To introduce a symbolic variable called *x*, use:

```
x = Symbol('x')
```

and to define a symbolic function *y* depending in symbolic variables *x*, *t*, use:

```
y = function_symbol('y', x, t)
```

SymEngine does not have specific command for defining an equation, so work with equations, one should always equalities with 0. To introduce an equation like $y(x,t) = xt$, use:

```
equation = y - x*t
```

For differentiation of *y* with respect to *x*, i.e. $\frac{\partial y}{\partial x}$, enter:

```
diff(y, x)
```

To achieve a certain order, repeat the above code as many times as necessary.

The current version of *SymEngine* does not support integration.

CHAPTER

THREE

MEASURES

Density function

The *pyProximation* implements two different scenarios for measure spaces:

1. Continuous case, where support of the measure is given as a compact subspace (box) of $\mathbb{R}^n$, and
2. Discrete case, where a finite set of points and their weights are given.

Continuous measure spaces

For the continuous case, generally assume that the support of the measure is a product of closed interval, i.e., $\prod_{i=1}^{n}[a_i, b_i]$, where for each i, $a_i < b_i$. Such a set can be defined as a list of ordered pairs of real numbers as `[(a1, b1), (a2, b2), ..., (an, bn)]`. Moreover, when we speak about a subset of the support, we always refer to a box, defined as a list of 2-tuples.

In this case, the measure is given implicitly as a density function $w(x)$. So the measure of a set *S* is given by

$$\mu(S) = \int_S w(x)dx.$$

For example, the following code defines the Lebesgue measure on $[-1, 1] \times [-1, 1]$ and finds the measure of the set $[0, 1] \times [-1, 0]$:

```
# import the Measure class
from pyProximation import Measure
# define the support of the measure
D = [(-1, 1), (-1, 1)]
# define the measure with the constant density 1
M = Measure(D, 1)
# define a set called S
S = [(0, 1), (-1, 0)]
# find the measure of the set S
print M.measure(S)
```

Discrete measure spaces

In this case, the measure is basically a convex combination of Dirac measure. Given a set $X = \{x_1, \ldots, x_n\}$ and corresponding non-negative weights $w_1, \ldots, w_n$, one defines a measure as $\mu = \sum_{i=1}^{n} w_i \delta_{x_i}$. Then the measure of a subset $S = \{x_{i_1}, \ldots, x_{i_k}\}$ of *X* is given by

$$\mu(S) = \int_S d\mu = \sum_{j=1}^{k} w_{i_j}$$

The following is a sample code for discrete case:

```
# import the Measure class
from pyProximation import Measure
# define the support and density
D = {'x1':1, 'x2':.5, 'x3':1.1, 'x4':.6}
# define the measure
M = Measure(D)
# define a set called S
S = ['x2', 'x3']
# find the measure of the set S
print M.measure(S)
```

Integrals

Suppose that a measure space (X, μ) and a measurable function f on X are given. The method `integral` computes $\int_X f d\mu$. If μ is discrete, then f can be a dictionary with keys as points of domain and values as evaluation at each point. Otherwise, f is simply a numerical function:

```
from pyProximation import Measure
from numpy import sqrt
# define the density function
w = lambda x:1./sqrt(1.-x**2)
# define the support
D = [(-1, 1)]
# initiate the measure space
M = Measure(D, w)
# set f(x) = x^2
f = lambda x: x**2
# integrate f(x) w.r.t. w(x)
print M.integral(f)
```

Or in two dimensions:

```
from pyProximation import Measure
from numpy import sqrt
# define the density function
w = lambda x, y:y**2/sqrt(1.-x**2)
# define the support
D = [(-1, 1), (-1, 1)]
# initiate the measure space
M = Measure(D, w)
# set f(x, y) = x^2 + y
f = lambda x, y: x**2 + y
# integrate f(x, y) w.r.t. w(x, y)
print M.integral(f)
```

p-norms

Given a measure space (X, μ) and a measurable function f, the p-norm of f, for a positive p is defined as:

$$\|f\|_p = \left(\int_X |f|^p d\mu \right)^{1/p}.$$

The method `norm(p, f)` calculates the above quantity.

Drawing samples

Suppose that (X, μ) is a measure space and one wishes to draw a sample of size n from X according to the distribution μ. This can be done by the method `sample(n)` which returns a list of n random points from the support, according to μ.

CHAPTER

FOUR

HILBERT SPACES

Orthonormal system of functions

Let *X* be a topological space and μ be a finite Borel measure on *X*. The bilinear function $\langle \cdot, \cdot \rangle$ defined on $L^2(X, \mu)$ as $\langle f, g \rangle = \int_X fgd\mu$ is an inner product which turns $L^2(X, \mu)$ into a Hilbert space.

Let us denote the family of all continuous real valued functions on a non-empty compact space *X* by $\mathrm{C}(X)$. Suppose that among elements of $\mathrm{C}(X)$, a subfamily *A* of functions are of particular interest. Suppose that *A* is a subalgebra of $\mathrm{C}(X)$ containing constants. We say that an element $f \in \mathrm{C}(X)$ can be approximated by elements of *A*, if for every $\epsilon > 0$, there exists $p \in A$ such that $|f(x) - p(x)| < \epsilon$ for every $x \in X$. The following classical results guarantees when every $f \in \mathrm{C}(X)$ can be approximated by elements of *A*.

Note: Stone-Weierstrass:

Every element of $\mathrm{C}(X)$ can be approximated by elements of *A* if and only if for every $x \neq y \in X$, there exists $p \in A$ such that $p(x) \neq p(y)$.

Despite the strong and important implications of the Stone-Weierstrass theorem, it leaves every computational details out and does not give an specific algorithm to produce an estimator for f with elements of *A*, given an error tolerance ϵ, and the search for a such begins.

Define $\|f\|_\infty$ (the $\sup$ norm of f) of a given function $f \in \mathrm{C}(X)$ by

$$\|f\|_\infty = \sup_{x \in X} |f(x)|,$$

Then the above argument can be read as: For every $f \in \mathrm{C}(X)$ and every $\epsilon > 0$, there exists $p \in A$ such that $\|f - p\|_\infty < \epsilon$.

Let $(V, \langle \cdot, \cdot \rangle)$ be an inner product space with $\|v\|_2 = \langle v, v \rangle^{\frac{1}{2}}$. A basis $\{v_\alpha\}_{\alpha \in I}$ is called an orthonormal basis for *V* if $\langle v_\alpha, v_\beta \rangle = \delta_{\alpha\beta}$, where $\delta_{\alpha\beta} = 1$ if and only if $\alpha = \beta$ and is equal to *0* otherwise. Every given set of linearly independent vectors can be turned into a set of orthonormal vectors that spans the same sub vector space as the original. The following well-known result gives an algorithm for producing such orthonormal from a set of linearly independent vectors:

Note: Gram–Schmidt

Let $(V, \langle \cdot, \cdot \rangle)$ be an inner product space. Suppose $\{v_i\}_{i=1}^n$ is a set of linearly independent vectors in *V*. Let

$$u_1 := \frac{v_1}{\|v_1\|_2}$$

and (inductively) let

$$w_k := v_k - \sum_{i=1}^{k-1} \langle v_k, u_i \rangle u_i \text{ and } u_k := \frac{w_k}{\|w_k\|_2}.$$

Then $\{u_i\}_{i=1}^n$ is an orthonormal collection, and for each *k*,

$$span\{u_1, u_2, \cdots, u_k\} = span\{v_1, v_2, \cdots, v_k\}.$$

Note that in the above note, we can even assume that $n = \infty$.

Let $B = \{v_1, v_2, \dots\}$ be an ordered basis for $(V, \langle \cdot, \cdot \rangle)$. For any given vector $w \in V$ and any initial segment of *B*, say $B_n = \{v_1, \dots, v_n\}$, there exists a unique $v \in \text{span}(B_n)$ such that $\|w - v\|_2$ is the minimum:

Note: Let $w \in V$ and *B* a finite orthonormal set of vectors (not necessarily a basis). Then for $v = \sum_{u \in B} \langle u, w \rangle u$

$$\|w - v\|_2 = \min_{z \in \text{span}(B)} \|w - z\|_2.$$

Now, let μ be a finite measure on *X* and for $f, g \in \mathrm{C}(X)$ define $\langle f, g \rangle = \int_X f g d\mu$. This defines an inner product on the space of functions. The norm induced by the inner product is denoted by $\| \cdot \|_2$. It is evident that

$$\|f\|_2 \leq \|f\|_\infty \mu(X), \ \forall f \in \mathrm{C}(X),$$

which implies that any good approximation in $\| \cdot \|_\infty$ gives a good $\| \cdot \|_2$-approximation. But generally, our interest is the other way around. Employing Gram-Schmidt procedure, we can find $\| \cdot \|_2$ within any desired accuracy, but this does not guarantee a good $\| \cdot \|_\infty$-approximation. The situation is favorable in finite dimensional case. Take $B = \{p_1, \dots, p_n\} \subset \mathrm{C}(X)$ and $f \in \mathrm{C}(X)$, then there exists $K_f > 0$ such that for every $g \in \text{span}(B \cup \{f\})$,

$$K_f \|g\|_\infty \leq \|g\|_2 \leq \|g\|_\infty \mu(X).$$

Since *X* is assumed to be compact, $\mathrm{C}(X)$ is separable, i.e., $\mathrm{C}(X)$ admits a countable dimensional dense subvector space (e.g. polynomials for when *X* is a closed, bounded interval). Thus for every $f \in \mathrm{C}(X)$ and every $\epsilon > 0$ one can find a big enough finite *B*, such that the above inequality holds. In other words, good enough $\| \cdot \|_2$-approximations of f give good $\| \cdot \|_\infty$-approximations, as desired.

OrthSystem

Given a measure space, the `OrthSystem` class implements the described procedure, symbolically. Therefore, it relies on a symbolic environment. Currently, three such environments are acceptable:

1. *sympy*
2. *sage*
3. *symengine*

Legendre polynomials

Let $d\mu(x) = dx$, the regular Lebesgue measure on $[-1, 1]$ and $B = \{1, x, x^2, \dots, x^n\}$. The orthonormal polynomials constructed from *B* are called *Legendre* polynomials. The n^{th} Legendre polynomial is denoted by $P_n(x)$.

The following code generates Legendre polynomials up to a given order:

```
# the symbolic package
from sympy import *
from pyProximation import Measure, OrthSystem
# the symbolic variable
x = Symbol('x')
# set a limit to the order
n = 6
# define the measure
D = [(-1, 1)]
M = Measure(D, 1)
S = OrthSystem([x], D, 'sympy')
# link the measure to S
S.SetMeasure(M)
# set B = {1, x, x^2, ..., x^n}
B = S.PolyBasis(n)
# link B to S
S.Basis(B)
# generate the orthonormal basis
S.FormBasis()
# print the result
print B.OrthBase
```

Chebyshev polynomials

Let $d\mu(x) = \frac{dx}{\sqrt{1-x^2}}$ on $[-1, 1]$ and B as in Legendre polynomials. The orthonormal polynomials associated to this setting are called *Chebyshev* polynomials and the n^{th} one is denoted by $T_n(x)$.

The following code generates Chebyshev polynomials up to a given order:

```
# the symbolic package
from sympy import *
from numpy import sqrt
from pyProximation import Measure, OrthSystem
# the symbolic variable
x = Symbol('x')
# set a limit to the order
n = 6
# define the measure
D = [(-1, 1)]
w = lambda x: 1./sqrt(1. - x**2)
M = Measure(D, w)
S = OrthSystem([x], D, 'sympy')
# link the measure to S
S.SetMeasure(M)
# set B = {1, x, x^2, ..., x^n}
B = S.PolyBasis(n)
# link B to S
S.Basis(B)
# generate the orthonormal basis
S.FormBasis()
# print the result
print S.OrthBase
```

Approximation

Let (X, μ) be a compact Borel measure space and $\mathcal{O} = \{u_1, u_2, \dots\}$ an orthonormal basis of function whose span is dense in $L^2(X, \mu)$. Given a function $f \in L^2(X, \mu)$, then f can be approximated as

$$f = \lim_{n\to\infty} \sum_{i=1}^{n} \langle f, u_i \rangle u_i$$

`OrthSystem.Series` calculates the coefficients $\langle f, u_i \rangle$:

Truncated Fourier series

Let $d\mu(x) = dx$, the regular Lebesgue measure on $[c, c + 2l]$ and $B = \{1, \sin(\pi x), \cos(\pi x), \sin(2\pi x), \cos(2\pi x), \dots, \sin(n\pi x), \cos(n\pi x), \}$. The following code calculates the Fourier series approximation of $f(x) = \sin(x)e^x$:

```
from sympy import *
from numpy import sqrt
from pyProximation import Measure, OrthSystem
# the symbolic variable
x = Symbol('x')
# set a limit to the order
n = 4
# define the measure
D = [(-1, 1)]
w = lambda x: 1./sqrt(1. - x**2)
M = Measure(D, w)
S = OrthSystem([x], D, 'sympy')
# link the measure to S
S.SetMeasure(M)
# set B = {1, x, x^2, ..., x^n}
B = S.FourierBasis(n)
# link B to S
S.Basis(B)
# generate the orthonormal basis
S.FormBasis()
# number of elements in the basis
m = len(S.OrthBase)
# set f(x) = sin(x)e^x
f = sin(x)*exp(x)
# extract the coefficients
Coeffs = S.Series(f)
# form the approximation
f_app = sum([S.OrthBase[i]*Coeffs[i] for i in range(m)])
print f_app
```

CHAPTER

FIVE

INTERPOLATION

Lagrange interpolation

Suppose that a list of $m+1$ points $\{(x_0, y_0), \ldots, (x_m, y_m)\}$ in $\mathbb{R}^2$ is given, such that $x_i \neq x_j$, if $i \neq j$. Then

$$p(\mathbf{x}) = \sum_{i=0}^{m} y_i \ell_i(\mathbf{x}),$$

is a polynomial of degree *m* which passes through all the given points. Here

$$\ell_i(\mathbf{x}) = \prod_{j \neq i} \frac{\mathbf{x} - x_j}{x_i - x_j},$$

are *Lagrange Basis Polynomials*.

This procedure can be extended to multivariate case as well.

Suppose that a list of points $\{x_1, \ldots, x_\rho\}$ in $\mathbb{R}^n$ and a list of corresponding values $\{y_1, \ldots, y_\rho\}$ are given. Let's denote by $\mathbf{X}$ the tuple $(\mathbf{X}_1, \ldots, \mathbf{X}_n)$ of variables and $\mathbf{X}_1^{e_1} \cdots \mathbf{X}_n^{e_n}$ by $\mathbf{X}^{\mathbf{e}}$, where $\mathbf{e} = (e_1, \ldots, e_n)$.

If for some $m > 0$, we have $\rho = \binom{m+n}{m}$, then the number of given points matches the number of monomials of degree at most *m* in the polynomial basis. Denote the exponents of these monomials by $\mathbf{e}_i$, $i = 1, \ldots, \rho$ and let

$$D = \begin{pmatrix} x_1^{\mathbf{e}_1} & \ldots & x_1^{\mathbf{e}_\rho} \\ \vdots & & \vdots \\ x_\rho^{\mathbf{e}_1} & \ldots & x_\rho^{\mathbf{e}_\rho} \end{pmatrix}$$

and for $1 \leq j \leq \rho$:

$$D_j = \begin{pmatrix} x_1^{\mathbf{e}_1} & \ldots & x_1^{\mathbf{e}_\rho} \\ \vdots & \vdots & \vdots \\ x_{j-1}^{\mathbf{e}_1} & \ldots & x_{j-1}^{\mathbf{e}_\rho} \\ \mathbf{X}^{\mathbf{e}_1} & \ldots & \mathbf{X}^{\mathbf{e}_\rho} \\ x_{j+1}^{\mathbf{e}_1} & \ldots & x_{j+1}^{\mathbf{e}_\rho} \\ \vdots & \vdots & \vdots \\ x_\rho^{\mathbf{e}_1} & \ldots & x_\rho^{\mathbf{e}_\rho} \end{pmatrix}.$$

Then the polynomial

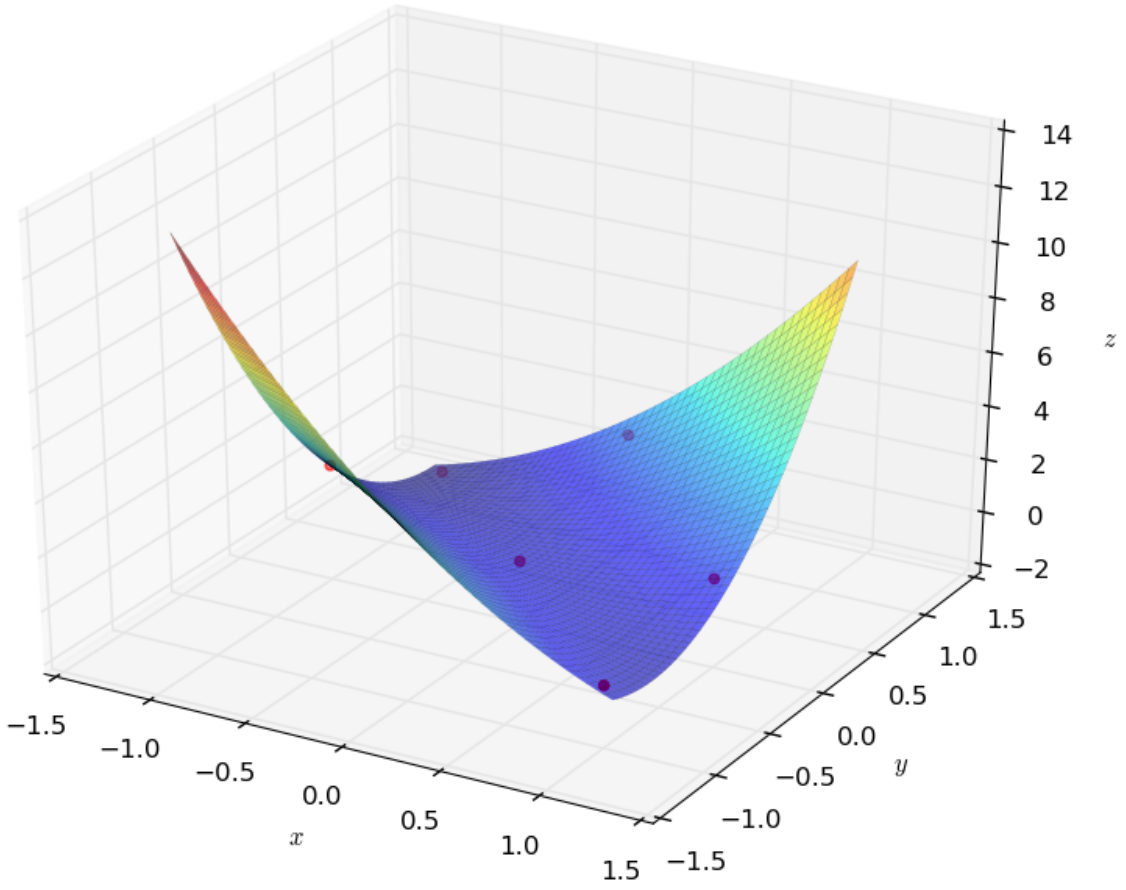
$$p(\mathbf{X}) = \sum_{i=1}^{\rho} y_i \frac{|D_i|}{|D|},$$

interpolates the given list of points and their corresponding values. Here $|M|$ denotes the determinant of *M*.

The above procedure is implemented in the `Interpolation` module. The following code provides an example in 2 dimensional case:

```
from sympy import *
from pyProximation import Interpolation
# define symbolic variables
x = Symbol('x')
y = Symbol('y')
# initiate the interpolation instance
Inter = Interpolation([x, y], 'sympy')
# list of points
points = [(-1, 1), (-1, 0), (0, 0), (0, 1), (1, 0), (1, -1)]
# corresponding values
values = [-1, 2, 0, 2, 1, 0]
# interpolate
p = Inter.Interpolate(points, values)
# print the result
print p
G = Graphics('sympy')
points3d = [(-1, 1, -1), (-1, 0, 2), (0, 0, 0), (0, 1, 2), (1, 0, 1), (1, -1, 0)]
G.Point(points3d, color='red')
G.Plot3D(p, (x, -1.1, 1.1), (y, -1.1, 1.1))
```

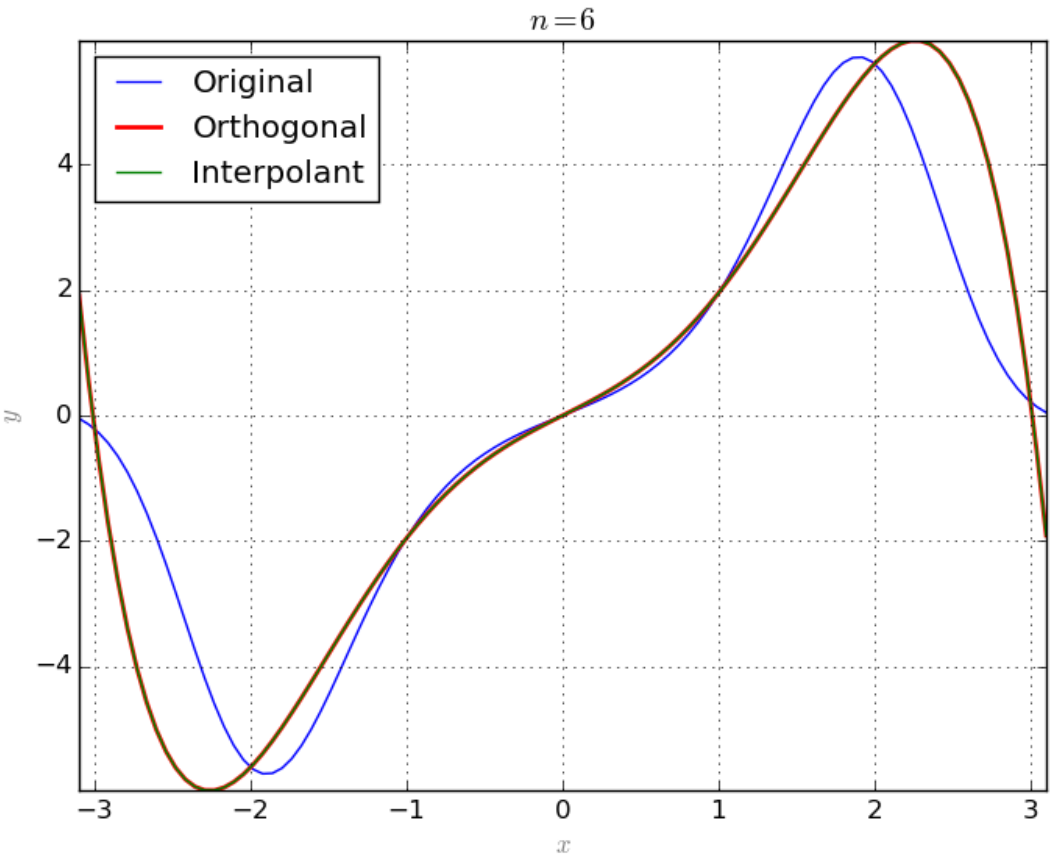
This will be the result:



L^2-approximation with discrete measures

Suppose that $\mu = \sum_1^n \delta_{x_i}$ is a measure with *n*-points in its support. Then the orthogonal system of polynomials consists of at most *n+1* polynomials. Approximation with these *n+1* polynomials is essentially same as interpolation:

```
# symbolic variable
x = Symbol('x')
# function to be approximated
g = sin(x)*exp(sin(x)*x)#x*sin(x)
# its numerical equivalent
g_ = lambdify(x, g, 'numpy')
# number of approximation terms
n = 6
# half interval length
l = 3.1
# interpolation points and values
Xs = [[-3], [-2], [-1], [0], [1], [2], [3]]
Ys = [g_(Xs[i][0]) for i in range(7)]
# a discrete measure
supp = {-3:1, -2:1, -1:1, 0:1, 1:1, 2:1, 3:1}
M = Measure(supp)
# orthogonal system
S = OrthSystem([x], [(-l, l)])
# link the measure
S.SetMeasure(M)
# polynomial basis
B = S.PolyBasis(n)
# link the basis to the orthogonal system
S.Basis(B)
# form the orthonormal basis
S.FormBasis()
# calculate coefficients
cfs = S.Series(g)
# orthogonal approximation
aprx = sum([S.OrthBase[i]*cfs[i] for i in range(len(B))])
# interpolate
Intrp = Interpolation([x])
intr = Intrp.Interpolate(Xs, Ys)
# plot the results
G = Graphics('sympy', numpoints=100)
G.SetTitle("$n = %d$"%(n))
G.Plot2D(g, (x, -l, l), color='blue', legend='Original')
G.Plot2D(aprx, (x, -l, l), color='red', legend='Orthogonal', thickness=2)
G.Plot2D(intr, (x, -l, l), color='green', legend='Interpolant')
G.save('OrthIntrp.png')
```


$n=6$
Original
Orthogonal
Interpolant
4
2
0
−2
−4
y
−3
−2
−1
0
1
2
3
x

CHAPTER

SIX

COLLOCATION

Integro-differential equations

A system of integro-differential equations is a system of equations that involve both integrals and derivatives of a function. The general first-order, linear (only with respect to the term involving derivative) integro-differential equation is of the form

$$\frac{d}{dx}u(x) + \int_{x_0}^{x} f(t, u(t))dt = g(x, u(x)), \quad u(x_0) = u_0.$$

Collocation method

In collocation method, one assumes that the solution of the equation is of a certain form and tries to exploit information about the assumed solution. More specifically, assume that $u_1, u_2, \ldots, u_n$ are orthonormal functions and the solution is of the form $f = \sum_i a_i u_i$ where a_i are unknowns. Plug f in to the equation and choose n different admissible points. plug those points in to the resulting equation to eliminate x. Then we obtain n equations in terms of $a_1, \ldots, a_n$. Solving the resulted system of algebraic equations gives an approximation for f.

Collocation class

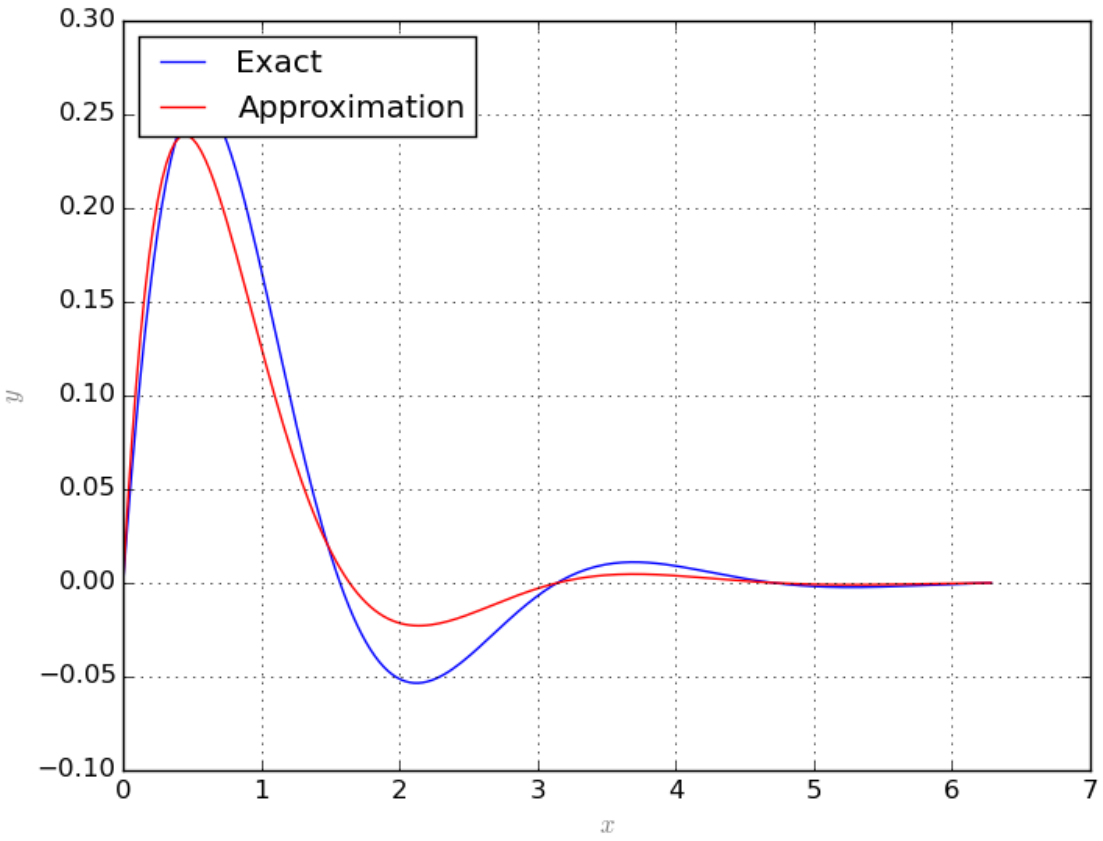
Single equation

The `Collocation` class implements the above described method. The following example solves the equation $\frac{dy}{dx} + 2y + 5\int ydx = 1, y(0) = y(2\pi) = 0$ where the exact solution is $\frac{1}{2}\sin(2x)e^{-x}$:

```
from sympy import *
from pyProximation import *
# symbolic variable
x = Symbol('x')
# symbolic function
y = Function('y')(x)
# number of approximation terms
n = 10
# orthogonal system
S = OrthSystem([x], [(0, 2*pi)])
# polynomial basis
B = S.PolyBasis(n)
# link the basis to the orthogonal system
S.Basis(B)
```

```
# form the orthonormal basis
S.FormBasis()
# form the equation
EQ = Eq(diff(y, x) + 2*y +5*integrate(y, x), 1)
# initiate collocation object with x as variable and y as function
C = Collocation([x], [y])
# link the orthogonal system to the collocation object and the unknown function `y`
C.SetOrthSys(S, y)
# link the equation to the collocation object
C.Equation([EQ])
# initial and boundary conditions
C.Condition(Eq(y, 0), [0])
C.Condition(Eq(y, 0), [2*pi])
# set the solver to 'scipy'
C.setSolver('scipy')
# solve to collocation system and print the solution
Apprx = C.Solve()
print Apprx[0]
```

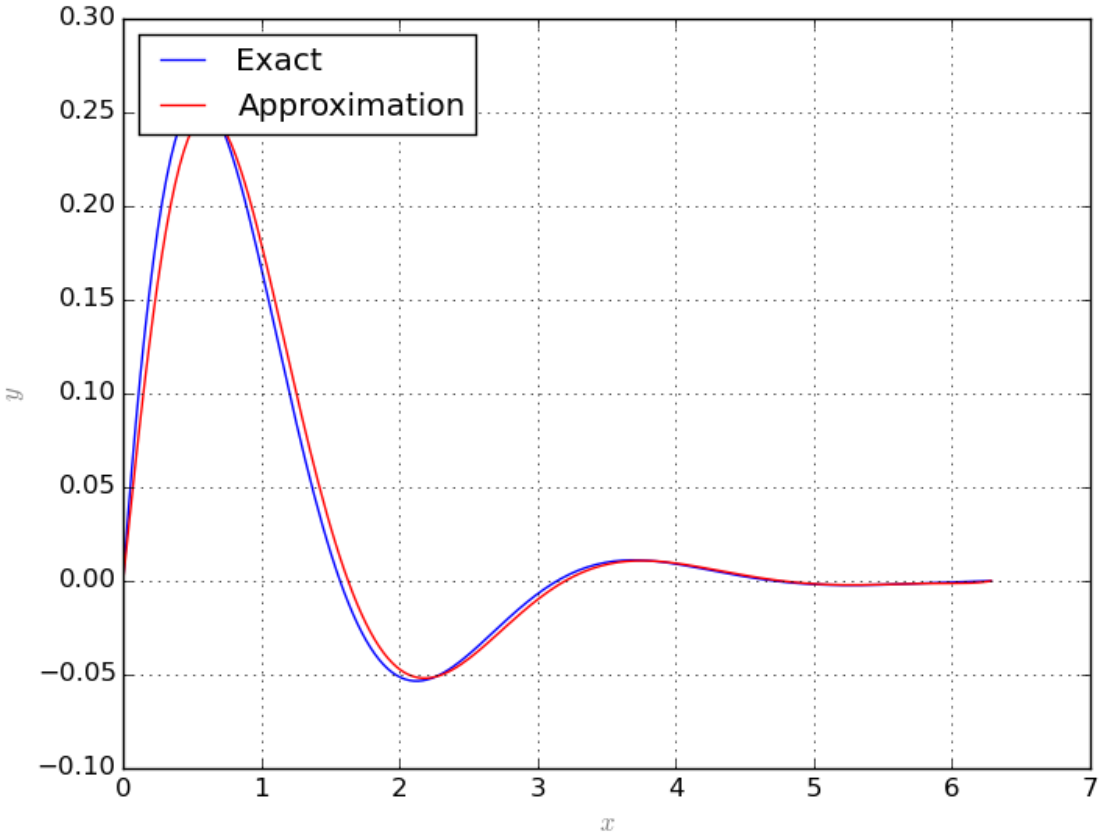
In the above example the `Collocation` class selects collocation points itself according to the measure chosen for orthogonal system, in this case, the usual Lebesgue measure. Thus, the class samples enough number of points uniformly from the domain. The solution and the exact answer are depicted below:



One can provide prefered collocation points to the solver. The following repeats the previous example where collocation points are selected in a way from the domain within a fixed distance from each other:

```
from sympy import *
from pyProximation import *
# symbolic variable
x = Symbol('x')
# symbolic function
y = Function('y')(x)
# number of approximation terms
n = 10
# orthogonal system
S = OrthSystem([x], [(0, 2*pi)])
# polynomial basis
B = S.PolyBasis(n)
# link the basis to the orthogonal system
S.Basis(B)
# form the orthonormal basis
S.FormBasis()
# form the equation
EQ = Eq(diff(y, x) + 2*y +5*integrate(y, x), 1)
# initiate collocation object with x as variable and y as function
C = Collocation([x], [y])
# link the orthogonal system to the collocation object and the unknown function `y`
C.SetOrthSys(S, y)
# link the equation to the collocation object
C.Equation([EQ])
# initial and boundary conditions
C.Condition(Eq(y, 0), [0])
C.Condition(Eq(y, 0), [2*pi])
# a list of collocation points
pnts = [[i*2*pi/n] for i in range(1,n)]
# link the collocation point to the object
C.CollPoints(pnts)
# set the solver to 'scipy'
C.setSolver('scipy')
# solve to collocation system and print the solution
Apprx = C.Solve()
print Apprx[0]
```

The result shows slight improvement in the solution:



Note: Each point of collocation must be given as a *list* or *tuple*.

System of equations

The `Collocation` class is also able to handle systems of equations. Consider the following system of partial differential equations:

$$\begin{cases} \frac{\partial x}{\partial t} + x + 4y &= 10 \\ x - \frac{\partial y}{\partial t} - y &= 0, \end{cases}$$

with initial conditions $x(0) = 4$ and $y(0) = 3$. The exact solution to the above system is:

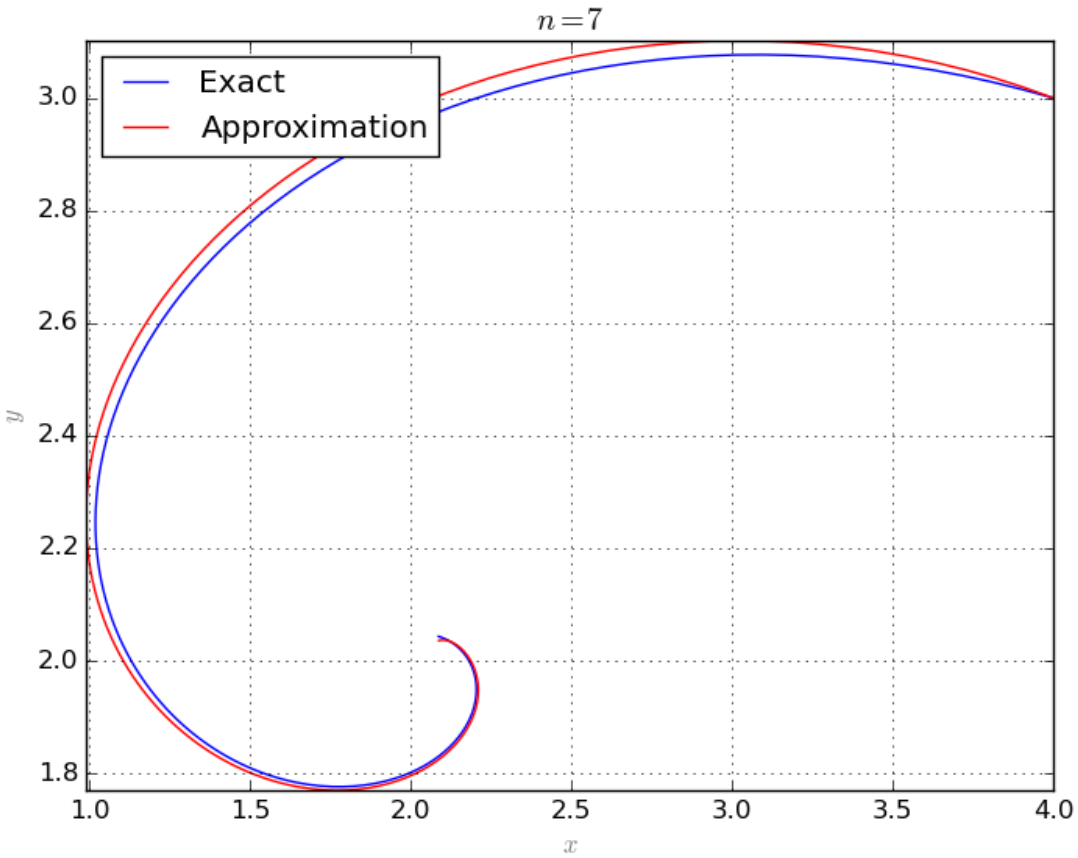
$$\begin{aligned} x(t) &= 2(1 + e^{-t}\cos(2t) - e^{-t}\sin(2t)) \\ y(t) &= 2 + e^{-t}\cos(2t) + e^{-t}\sin(2t) \end{aligned}$$

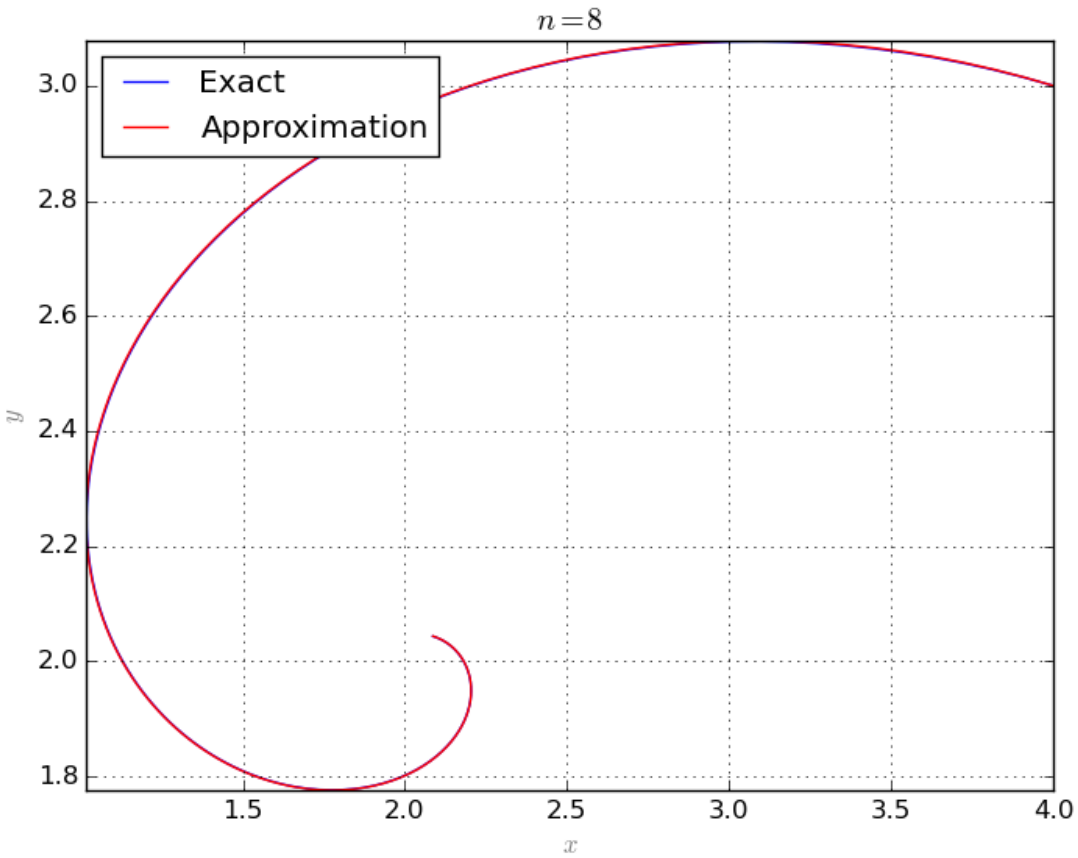
The following code solves the system and plots the exact and approximate solutions:

```
from sympy import *
from pyProximation import *
# symbolic variable
t = Symbol('t')
# symbolic function
x = Function('x')(t)
```

```
y = Function('y')(t)
# number of approximation terms
n = 8
# orthogonal system
S = OrthSystem([t], [(0, pi)])
# polynomial basis
B = S.PolyBasis(n)
# link the basis to the orthogonal system
S.Basis(B)
# form the orthonormal basis
S.FormBasis()
# form the equation
EQ1 = Eq(diff(x, t) + x + 4*y, 10)
EQ2 = Eq(x - diff(y, t) - y, 0)
# initiate collocation object with x as variable and y as function
C = Collocation([t], [x, y])
# link the orthogonal system to the collocation object and unknown functions
C.SetOrthSys(S, x)
C.SetOrthSys(S, y)
# link the equation to the collocation object
C.Equation([EQ1, EQ2])
# initial and boundary conditions
C.Condition(Eq(x, 4), [0])
C.Condition(Eq(y, 3), [0])
# set the solver to 'scipy'
C.setSolver('scipy')
# solve to collocation system and print the solution
Apprx = C.Solve()
# print the answers
print Apprx[0]
print Apprx[1]
# the exact solution
f = [2*(1 + exp(-t)*cos(2*t) - exp(-t)*sin(2*t)), 2 + exp(-t)*cos(2*t) + exp(-
↪t)*sin(2*t)]
# plot the exact and approximate solutions
G = Graphics('sympy', numpoints=200)
G.SetTitle("$n = %d$"%(n))
G.ParamPlot2D(f, (t, 0, pi), color='blue', legend='Exact')
G.ParamPlot2D(Apprx, (t, 0, pi), color='red', legend='Approximation')
G.save('Plot-%d.png'%(n))
```

The followings are results for $n = 7$ and $n = 8$:


$n=7$
Exact
Approximation
3.0
2.8
2.6
2.4
2.2
2.0
1.8
y
1.0
1.5
2.0
2.5
3.0
3.5
4.0
x


$n=8$
Exact
Approximation
3.0
2.8
2.6
2.4
2.2
2.0
1.8
y
1.5
2.0
2.5
3.0
3.5
4.0
x

CHAPTER

SEVEN

DISTRIBUTING OVER SUBREGIONS

Sometimes the system of equations is too complicated and it might be very costly to produce a reasonably good estimation by raising the size of the function basis. A possible solution is to divide the domain into various smaller subregions and try to solve the system for those smaller subregions. Use the partial solutions to force appropriate boundary condition to the remaining subregions in order to get consistent solutions on all subregions. This task is done by use of the *SubRegion* class.

Note: Currently, this only works with symbolic package *sympy*.

CHAPTER

EIGHT

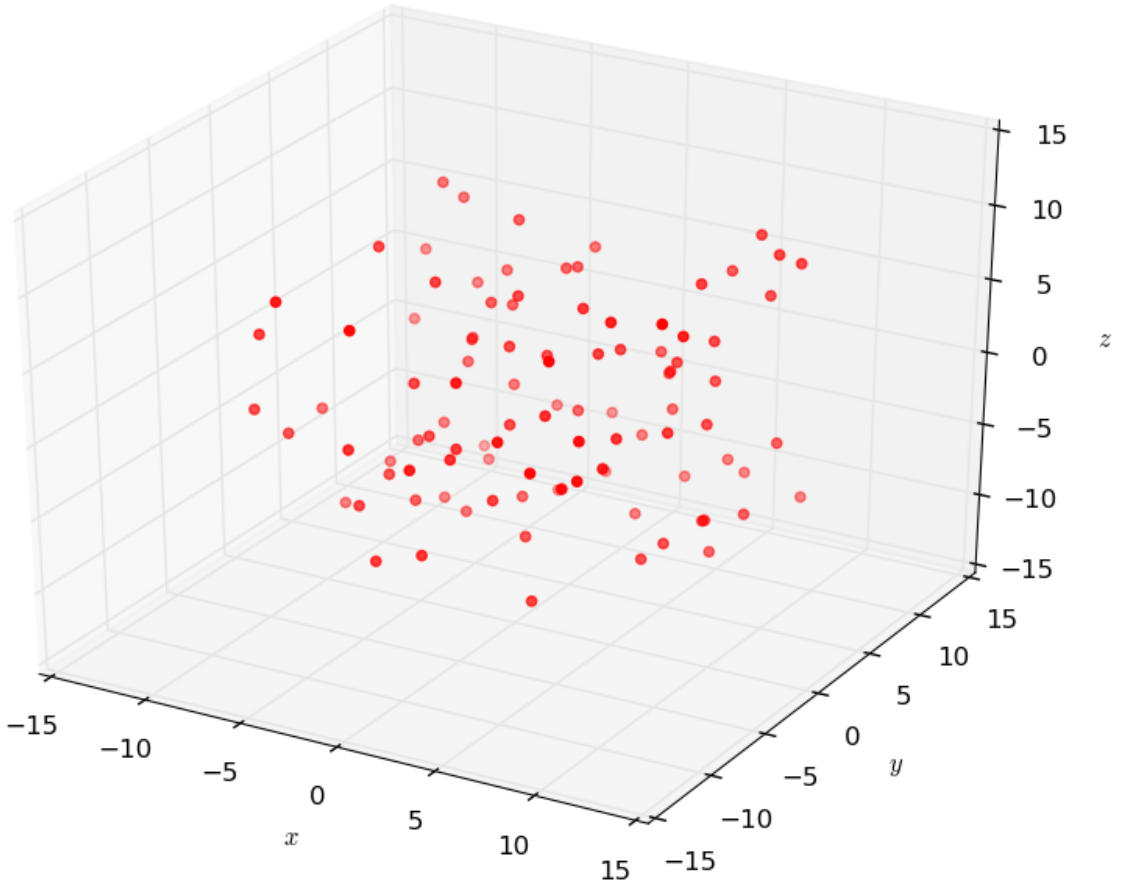
GRAPHICS

This small module provides simple functionality to draw 2D and 3D plots. Since the functions that appear in this package are of three different types, it must be declared to the `Graphic` class, what type of function it is dealing with. The three different types are:

1. `numeric`, the default type,
2. `sympy`, the symbolic **sympy** functions,
3. `sage`, the symbolic **sage** functions

The following plots a list of random points in 3D:

```
from random import uniform
# import the module
from pyProximation import Graphics
# list of random points
points = [(uniform(-10, 10), uniform(-10, 10), uniform(-10, 10)) for _ in range(100)]
# the graphic object
G = Graphics()
# link the points to the object
G.Point(points, color='red')
# save to file
G.save('points.png')
```

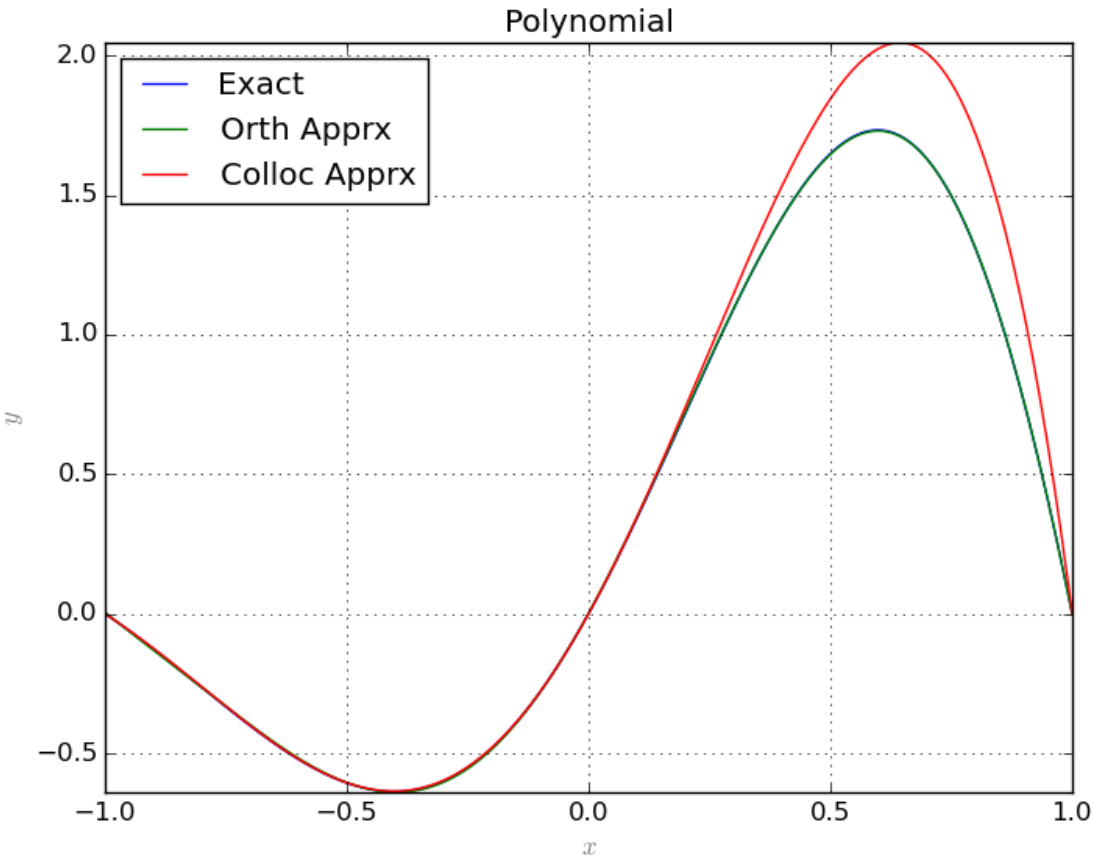


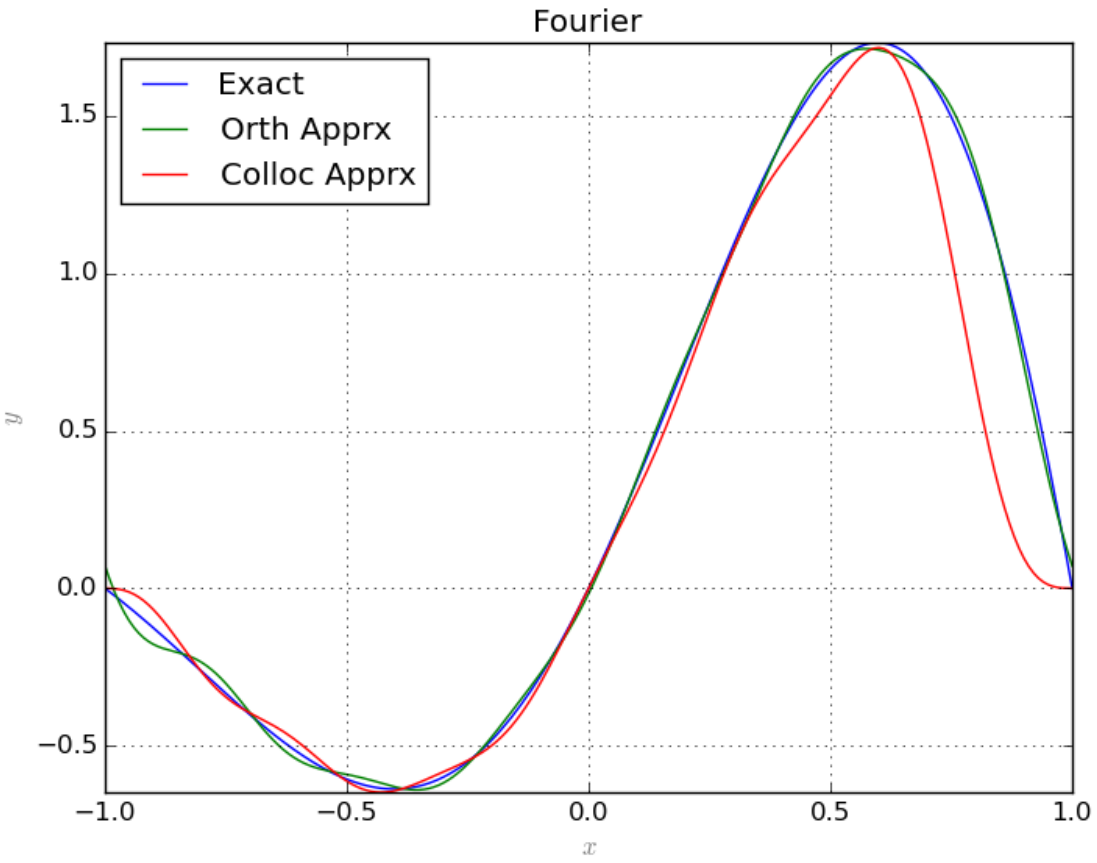
Two Dimensional Plots

The method `Plot2D` is to plot curves in 2 dimensions. The following code shows an example:

```
# chOose the symbolic environment
Symbolic = 'sympy'
# define the symbolic variables accordingly
if Symbolic == 'sympy':
        from sympy import *
        x = Symbol('x')
        y = Function('y')(x)
elif Symbolic == 'sage':
        from sage.all import *
        x = var('x')
        y = function('y')(x)
# import the modules
from pyProximation import *
# size of basis
n = 6
# set the measure
M = Measure([(-1, 1)], lambda x:1./sqrt(1.-x**2))
# Orthogonal system of functions
S = OrthSystem([x], [(-1, 1)], Symbolic)
```

```
# link the measure to the orthogonal system
S.SetMeasure(M)
# monomial basis
B = S.PolyBasis(n)
# Fourier basis
#B = S.FourierBasis(n)
# link the basis
S.Basis(B)
# form the orthogonal system
S.FormBasis()
# the exact solution
Z = sin(pi*x)*exp(x)
R = diff(Z, x, x) - diff(Z, x)
# the pde
if Symbolic == 'sympy':
        EQ1 = (Eq(diff(y, x, x) - diff(y, x), R))
elif Symbolic == 'sage':
        EQ1 = (diff(y, x, x) - diff(y, x) == R)
# corresponding coefficients
series = S.Series(Z)
# orthogonal approximation
ChAprx = sum([S.OrthBase[i]*series[i] for i in range(m)])
# set up the collocation class
C = Collocation([x], [y], Symbolic)
# link to the orthogonal system
C.SetOrthSys(S)
# link the equation
C.Equation([EQ1])
# initial conditions
if Symbolic == 'sympy':
        C.Condition(Eq(y, 0), [0])
        C.Condition(Eq(y, sin(-pi)*exp(-1)), [-1])
        C.Condition(Eq(y, sin(pi)*exp(1)), [1])
elif Symbolic == 'sage':
        C.Condition(y == 0, [0])
        C.Condition(y == sin(-pi)*exp(-1), [-1])
        C.Condition(y == sin(pi)*exp(1), [1])
# collocation points
m = len(S.OrthBase)
pnts = [[-1 + i*2./m] for i in range(m)]
# link the collocation points
C.CollPoints(pnts)
# set solver
C.setSolver('scipy')
# solve
Apprx = C.Solve()
print Apprx[0]
# plot the results
G = Graphics(Symbolic)
G.Plot2D(Z, (x, -1, 1), color='blue', legend='Exact')
G.Plot2D(ChAprx, (x, -1, 1), color='green', legend='Orth Apprx')
G.Plot2D(Apprx[0], (x, -1, 1), color='red', legend='Colloc Apprx')
G.save('PlotsPoly.png'%(n))
#G.save('PlotsFourier.png'%(n))
```


Polynomial
Exact
Orth Apprx
Colloc Apprx
2.0
1.5
1.0
0.5
0.0
−0.5
−1.0
−0.5
0.0
0.5
1.0
y
x



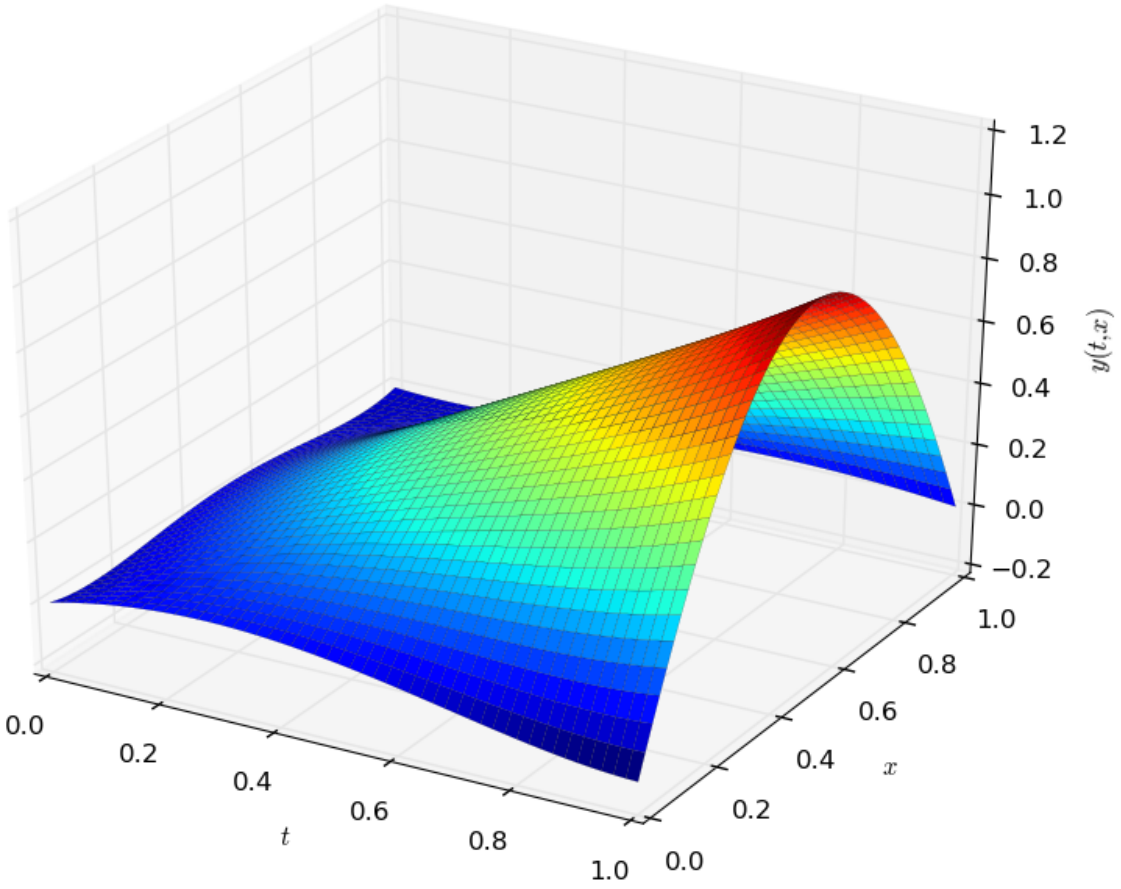
For an example of parametric plots see *this*.

Three Dimensional Plots

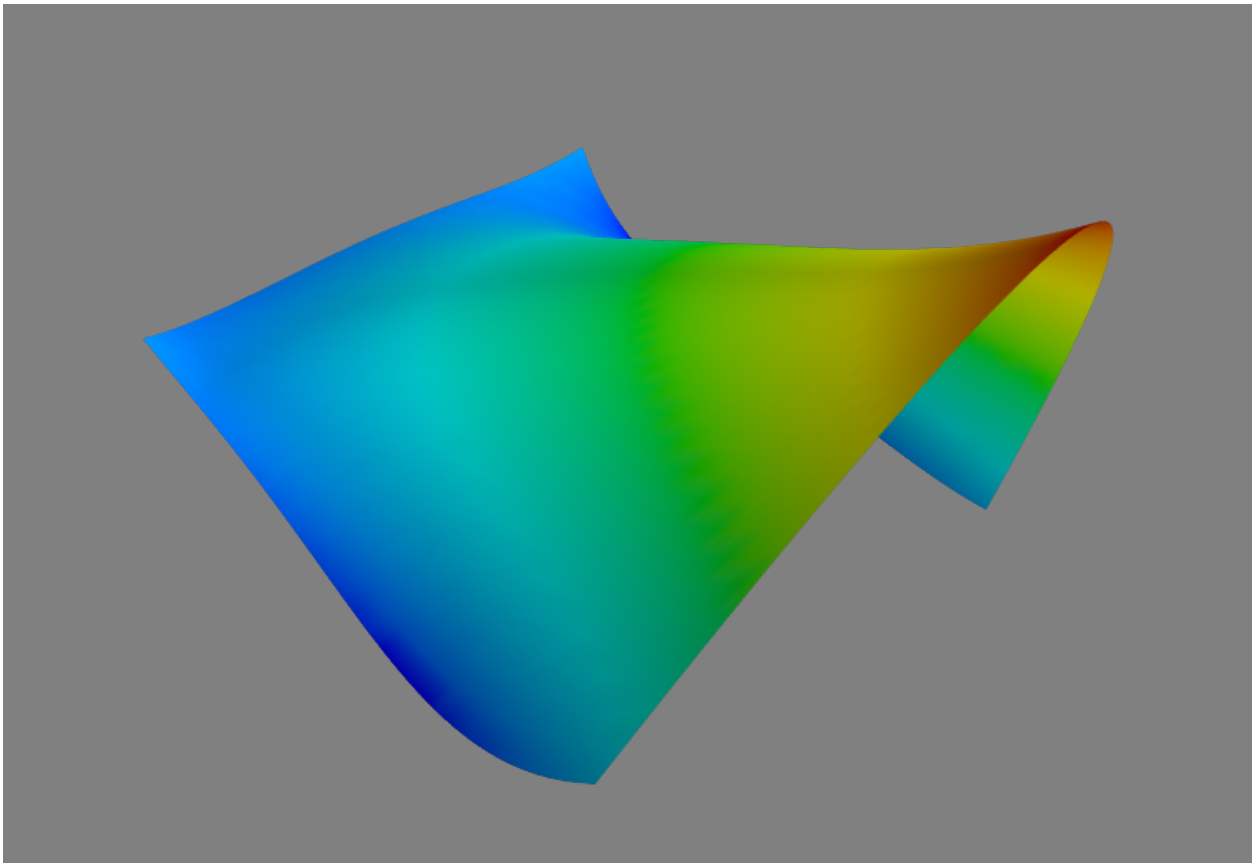
To generate static 3 dimensional plots, `Graphics` implement `Plot3D`. The following example illustrates the usage of this method:

```
# select the symbolic tool
Symbolic = 'sympy'
if Symbolic == 'sympy':
        from sympy import *
        x = Symbol('x')
        t = Symbol('t')
        y = Function('y')(t, x)
elif Symbolic == 'sage':
        from sage.all import *
        t = var('t')
        x = var('x')
        y = function('y')(t, x)
# import the modules
from pyProximation import *
# degree of basis
n = 4
# orthogonal system
```

```
S = OrthSystem([t, x], [(0, 1), (0, 1)])
# monomial basis
B = S.PolyBasis(n)
# link the basis
S.Basis(B)
# form the orthonormal basis
S.FormBasis()
# construct a pde
Z = t*sin(pi*x)
R = diff(Z, t) - diff(Z, x)
if Symbolic == 'sympy':
        EQ1 = Eq(diff(y, t) - diff(y, x), R)
elif Symbolic == 'sage':
        EQ1 = diff(y, t) - diff(y, x) == R
# collocation object
C = Collocation([t, x], [y])
# link the orthogonal system
C.SetOrthSys(S)
# link the equation
C.Equation([EQ1])
# some initial & boundary conditions
if Symbolic == 'sympy':
        C.Condition(Eq(y, 0), [0, 0])
        C.Condition(Eq(y, 0), [0, .3])
        C.Condition(Eq(y, 0), [1, 1])
        C.Condition(Eq(y, 1), [1, .5])
        C.Condition(Eq(y, 0), [0, .7])
elif Symbolic == 'sage':
        C.Condition(y == 0, [0, 0])
        C.Condition(y == 0, [0, .3])
        C.Condition(y == 0, [1, 1])
        C.Condition(y == 1, [1, .5])
        C.Condition(y == 0, [0, .7])
# set the solver
C.setSolver('scipy')
# solve the collocation system
Apprx = C.Solve()
print Apprx[0]
# plot the result
G = Graphics(Symbolic)
G.SetLabelX("$t$")
G.SetLabelY("$x$")
G.SetLabelZ("$y(t, x)$")
G.Plot3D(Apprx[0], (t, 0, 1), (x, 0, 1))
# save the image
G.save('PDEplot.png'%(n))
# open the interactive window
G.interact()
```



A second method called `interact` is invoked at the end which called the `mayavi` library to show an interactive view of the surface.



CHAPTER

NINE

CODE DOCUMENTATION

class base.**Foundation**

This class contains common features of all modules.

DetSymEnv()

Returns a list. The list consists of all symbolic tools present among 'sympy','sage' and 'symengine'.

class measure.**Measure**(*dom*, *w=None*)

An instance of this class is a measure on a given set *supp*. The support is either

- a python variable of type *set*, or
- a list of tuples which represents a box in euclidean space.

Initializes a measure object according to the inputs:

- ***dom* must be either**
  - a list of 2-tuples
  - a non-empty dictionary
- ***w* must be a**
  - a function if *dom* defines a region
  - left blank (None) if *dom* is a dictionary

boxCheck(*B*)

Checks the structure of the box *B*. Returns *True* id *B* is a list of 2-tuples, otherwise it returns *False*.

check(*dom*, *w*)

Checks the input types and their consistency, according to the *__init__* arguments.

integral(*f*)

Returns the integral of f with respect to the currwnt measure over the support.

measure(*S*)

Returns the measure of the set *S*. *S* must be a list of 2-tuples.

norm(p, f)

Computes the norm-p of the f with respect to the current measure.

sample(*num*)

Samples from the support according to the measure.

class orthsys.**OrthSystem**(*variables*, *var_range*, *env='sympy'*)

OrthogonalSystem class produces an orthogonal system of functions according to a suggested basis of functions and a given measure supported on a given region.

This basically performs a 'Gram-Schmidt' method to extract the orthogonal basis. The inner product is obtained by integration of the product of functions with respect to the given measure (more accurately, the distribution).

To initiate an instance of this class one should provide a list of symbolic variables *variables* and the range of each variable as a list of lists *var_range*.

To initiate an orthogonal system of functions, one should provide a list of symbolic variables *variables* and the range of each these variables as a list of lists *var_range*.

Basis(*base_set*)
To specify a particular family of function as a basis, one should call this method with a list *base_set* of linearly independent functions.

FormBasis()
Call this method to generate the orthogonal basis corresponding to the given basis via *Basis* method. The result will be stored in a property called *OrthBase* which is a list of function that are orthogonal to each other with respect to the measure *measure* over the given range *Domain*.

FourierBasis(*n*)
Generates a Fourier basis from variables consisting of all *sin* & *cos* functions with coefficients at most *n*.

PolyBasis(*n*)
Generates a polynomial basis from variables consisting of all monomials of degree at most *n*.

Series(*f*)
Given a function *f*, this method finds and returns the coefficients of the series that approximates *f* as a linear combination of the elements of the orthogonal basis.

SetMeasure(*M*)
To set the measure which the orthogonal system will be computed, simply call this method with the corresponding distribution as its parameter *dm*; i.e, the parameter is *d(m)* where *m* is the original measure.

SetOrthBase(*base*)
Sets the orthonormal basis to be the given *base*.

TensorPrd(*Bs*)
Takses a list of symbolic bases, each one a list of symbolic expressions and returns the tensor product of them as a list.

inner(*f*, *g*)
Computes the inner product of the two parameters with respect to the measure *measure*.

project(*f*, *g*)
Finds the projection of *f* on *g* with respect to the inner product induced by the measure *measure*.

class interpolation.**Interpolation**(*var*, *env='sympy'*)
The `Interpolation` class provides polynomial interpolation routines in multi variate case.

var is the list of symbolic variables and *env* is the the symbolic tool.

Delta(*idx=-1*)
Construct the matrix corresponding to *idx*'th point, if *idx>0* Otherwise returns the discriminant.

Interpolate(*points*, *vals*)
Takes a list of points *points* and corresponding list of values *vals* and return the interpolant.

Since in multivariate case, there is a constraint on the number of points, it checks for the valididty of the input. In case of failure, describes the type of error occured according to the inputs.

MinNumPoints()
Returns the minimum number of points still required.

Monomials()
Generates the minimal set of monomials for interpolation.

class collocation.**Collocation**(*variables*, *ufunc*, *env='sympy'*)
The *Collocation* class tries to approximate the solutions of a system of partial differential equations with respect to an orthogonal system of functions.

To initiate an instance of this class one needs to provide two set of parameters:

1. List of independent symbolic variables *variables*;
2. List of unknown functions to be found that depend on the independent variables *ufunc*.

CollPoints(*pnts*)
Accepts alist of collocation point *pnts*, to form the algebraic system of equations and find the coefficients of the orthogonal functions from *OrthogonalSystem.OrthBase*. Each point must be either a list or a tuple.

Condition(*eq*, *val*)
List of initial and boundary conditions.

Equation(*eq*)
To enter the system of equations, use this meyhod with a list of equations as input.

FindDomain()
Finds the region that all variables are defined.

PlugPoints()
Internal use: plug in collocation points to elliminate independent variables and keep the coefficients.

SetOrthSys(*obj*, *func*)
To approximate the solutions of the system of pdes, the class requires an orthogonal system of functions *OrthogonalSystem*. This method accepts such a system.

Solve()
Solves the collocation equations and keep a dictionary of coefficients in *self.Coeffs* and returns a list of functions in the span of orthoginal system.

collocate()
Internal use: generates the system of equations for coefficients to be used via collocation points.

setSampleMeasure(*meas*)
Sets the measure over the domain for sampling collocation points in case of necessity.

setSolver(*solver*, *optn='lm'*)
Currently only two solvers are supported:

1.the *sage*'s defult solver for rather simple system of algebraic equations.

2. the *scipy*'s *fsolves* to handel more complex and larger systems. It also supports the following solvers from scipy:

- *hybr*
- *lm* (defauls)
- *broyden1*
- *broyden2*
- *anderson*
- *krylov*
- *df-sane*

class subregion.**SubRegion**(*collsys*, *num_parts=[]*)

The *SubRegion* class partitions the region into sub-regions, solve the system of Integro-differential equations on each and glue them together.

It takes:

1. a collocation instance *collsys*;
2. an optional list of positive integers *num_parts* which shows the number of equal length partitions for each variable.

AnalyseConditions()

Associates boundary conditions to each sub-collocation system. Starting from one corner, associates boundary conditions to all adjacent subregions based on the solution of the current subregion.

ClosedForm(*func*)

Compiles a piece-wise defined symbolic function from the solution for *func* which is an unknown from the original collocation system.

InitMiniSys()

Initializes a set of collocation systems for all subregions.

Solve()

Solves the collocation system for each region and generates boundary conditions for adjacent regions.

corners(*tpl*)

Generates end corner points of the region represented *tpl* as collocation points (used internally).

class graphics.**Graphics**(*env='numeric'*, *numpoints=50*)

This class tends to provide basic graphic tools based on *matplotlib* and *mayavi*.

Accepts one optional argument *env* which determines the types of the function to be visualized:

- *numeric*: is a numerical function (regular python functions)
- *sympy*: Sympy symbolic function
- *sage*: Sage symbolic function

ParamPlot2D(*funcs*, *rng*, *color='blue'*, *legend=''*, *thickness=1*)

Appends a parametric curve to the Graphics object. The parameters are as follows:

- `funcs`: the tupleof functions to be plotted,
- `rng`: a triple of the form *(t, a, b)*, where *t* is the *funcs*'s independents variable, over the range *[a, b]*,
- `color`: the color of the current curve,
- `legend`: the text for the legend of the current crve.

Plot2D(*func*, *xrng*, *color='blue'*, *legend=''*, *thickness=1*)

Appends a curve to the Graphics object. The parameters are as follows:

- *func*: the function to be plotted,
- *xrng*: a triple of the form *(x, a, b)*, where *x* is the *func*'s independents variable, over the range *[a, b]*,
- *color*: the color of the current curve,
- *legend*: the text for the legend of the current crve.

Plot3D(*func*, *xrng*, *yrng*)

Sets a surface to the Graphics object. The parameters are as follows:

- `func`: the function to be plotted,

- `xrng`: a triple of the form *(x, a, b)*, where *x* is the first *func's independents variable, over the range '[a, b]*,
- `yrng`: a triple of the form *(y, c, d)*, where *x* is the second *func*'s independents variable, over the range *[c, d]*.

Point (*pnts*, *color='blue'*, *marker='o'*, *legend=''*)
Adds a list of points to the plot.

SetLabelX (*lbl*)
Sets the label for *X* axis

SetLabelY (*lbl*)
Sets the label for *Y* axis

SetLabelZ (*lbl*)
Sets the label for *Z* axis

SetTitle (*ttl*)
Sets the title of the graph.

interact ()
Shows an interavtive demo of the 3D surface, using *mayavi*, so it requires *mayavi* for python to be installed.

save (*fname='fig.png'*)
Saves the outpu of the *Graphics* object to the file *fname*.

CHAPTER

TEN

INDICES AND TABLES

- genindex
- modindex
- search

T