Klasy i obiekty w Pythone cz 2.

Do pobrania

Poniżej znajduje się kod źródłowy programu do całkowania numerycznego funkcji skalarnych jednej zmiennej.

Kod źródłowy

- calculate_integral.py
- gausslegendre.py
- integrands.py
- integrators.py
- mesh.py
- newtoncotes.py
- quadratures.py

Dane

- Plik siatki

Szczegóły implementacji

Program głowny

<sxh python> “”” Calculate integral of scalar function in 1D ”””
import itertools
import argparse
import quadratures

from mesh import Mesh
from integrands import Integrand
from integrators import MeshIntegrator

def parse_command_line():

```
"""Parse command line"""
parser = argparse.ArgumentParser()
parser.add_argument('-q', '--quadrature', dest='quadrature',
                    default='NewtonCotes',
                    choices = quadratures.known_quadratures())
parser.add_argument("-n", '--nodes', type=int, dest='nnodes', default=2,
                    help="number of quadrature nodes")
parser.add_argument("-i", '--integrand', default='x**2', dest='integrand',
                    help="expression to be integrated")
parser.add_argument("meshfile", help="name of the mesh file")
args = parser.parse_args()
return args
```

def main():

```
args = parse_command_line()
mesh = Mesh()
mesh.load(args.meshfile)
integrand = Integrand(args.integrand)
quadrature = quadratures.make_quadrature(args.quadrature, args.nnodes)
integrator = MeshIntegrator()
integral = integrator.integrate(mesh, integrand, quadrature)
print("Integral of %s is %g" %(args.integrand, integral))
```

if name == 'main':

```
main()
```

</sxh>

Siatka

<sxh python> “”“ Define 1D Mesh class ”“”

class Mesh:

```
"""Represents on dimensional mesh
"""
def __init__(self):
  self.nodes = list()
```

```
def load(self, filename):
  """Read from file a list of nodes"""
  with open(filename, 'r') as f:
    for l in f:
      self.nodes.append(tuple(float(x) for x in l.split()))
```

```
def nelem(self):
  return len(self.nodes)-1
```

</sxh>

Funcja podcałkowa

<sxh python> “”“ Define integrand function ”“”

class Integrand:

```
"""Represents integrand as 1D scalar function
```

```
"""
def __init__(self, expression):
  self.expression = expression
```

```
def evaluate(self, x):
  return eval(self.expression)
```

```
def __call__(self, x):
  return self.evaluate(x)
```

</sxh>

Integrator na siatce

<sxh python> “”“ Integrate function over a mesh ”“” import itertools

class MeshIntegrator:

```
"""Calculate integral over a domain discretised by a mesh
"""
```

```
def integrate(self, mesh, integrand, quadrature):
  """Integrate function fun using numerical integration on given mesh
  """
  integral = 0.0
  for i in range(mesh.nelem()):
    integral += self.integrate_element(i, mesh, integrand, quadrature)
  return integral
```

def make_mesh_fun(self, mesh, fun):

```
  xcoord = [ node[0] for node in mesh.nodes ]
  discrete_fun = [ fun(x) for x in xcoord ]
  return discrete_fun
```

def integrate_element(self, i, mesh, integrand, quadrature):

```
  x1 = mesh.nodes[i][0]
  x2 = mesh.nodes[i+1][0]
  qr = quadrature
  integral = 0.0
  for (x, w) in itertools.zip_longest(qr.nodes(x1, x2), qr.weights(x1,x2)):
    integral += w * integrand(x);
  return integral
```

</sxh>

Fabryka kwadratur

<sxh python> “”“ Define factory for different quadratures ”“” import newtoncotes import gausslegendre

def known_quadratures():

```
return ["NewtonCotes", "GaussLegendre"]
```

def make_quadrature(name, nnodes):

```
if name == "NewtonCotes":
    return newtoncotes.NewtonCotes(nnodes)
elif name == "GaussLegendre":
    return gausslegendre.GaussLegendre(nnodes)
raise RuntimeError("Invalid quadrature %s %d" %(name, nnodes))
```

</sxh>

Kwadratury Newtona-Cotesa

<sxh python> “”“ Newton-Cotes numerical integration formulas ”“”

class NewtonCotes:

```
_weights = { 2: (0.5, 0.5),
             3 : tuple(x/6.0 for x in (1, 4, 1))
           }
```

```
def __init__(self, nnodes):
  self.ref_weights = self._weights[nnodes]
  self.degree = len(self.ref_weights)-1
```

```
def nodes(self, a=0.0, b=1.0):
  n = len(self.ref_weights)
  dx = (b-a)/(n-1)
  return [a+i*dx for i in range(n)]
```

```
def weights(self, a=0.0, b=1.0):
  return ((b-a)*x for x in self.ref_weights)
```

class TrapezoidQuadrature(NewtonCotes):

```
  def __init__(self):
      super().__init__(2)
```

class SimpsonQuadrature(NewtonCotes):

```
  def __init__(self):
      super().__init__(3)
```

</sxh>

Kwadratury Gaussa-Legendre'a

<sxh python> “”“ Gauss-Legendre numerical integration formulas ”“” import itertools import math from math import sqrt

def antisymmetrize(seq, central=tuple()):

```
  return list(itertools.chain([-x for x in seq[::-1]], central, seq))
```

def symmetrize(seq, central=tuple()):

```
  return list(itertools.chain(seq[::-1], central, seq))
```

class GaussLegendre:

```
_nodes = { 1: [0.0],
           2: antisymmetrize([sqrt(1.0/3.0)]),
           3: antisymmetrize([sqrt(3.0/5.0)], [0.0]),
           4: antisymmetrize([sqrt((3-2*sqrt(6/5))/7),
                              sqrt((3+2*sqrt(6/5))/7)])
         }
_weights = { 1: [2.0],
             2: symmetrize([1.0]),
             3: symmetrize([5/9], [8/9]),
             4: symmetrize([(18+sqrt(30))/36,
                            (18-sqrt(30))/36])
           }
```

```
def __init__(self, nnodes):
  self.ref_nodes = self._nodes[nnodes]
  self.ref_weights = self._weights[nnodes]
```

```
def nodes(self, a=0.0, b=1.0):
  shift = (a+b)/2
  scale = (b-a)/2
  return [shift + scale*x for x in self.ref_nodes]
```

```
def weights(self, a=0.0, b=1.0):
  jacobian = (b-a)/2
  return [jacobian*w for w in self.ref_weights]
```

</sxh>