NGS-Py Documentation

Release 6.2

Joachim Schoeberl

Jul 10, 2018

Whetting the appetite

Netgen/NGSolve 6 contains a rich Python interface. Program flow as well as geometry description and equation setup can be controlled from Python. You should be familiar with weak formulations of partial differential equations and the finite element method (NGSolve-oriented lecture notes are here: Scientific Computing) and the Python programming language. The interface to Python is inspired by the FEniCS project.

CHAPTER 1

Poisson equation

We solve the Poisson equation on the unit-square, with homogeneous Dirichlet boundary conditions. You can run the example either directly within the Python interpreter (Python version 3 is required!):

> python3 poisson.py

or you can run it with Netgen providing you also a graphical user interface

> netgen poisson.py

Store the following programme as poisson.py

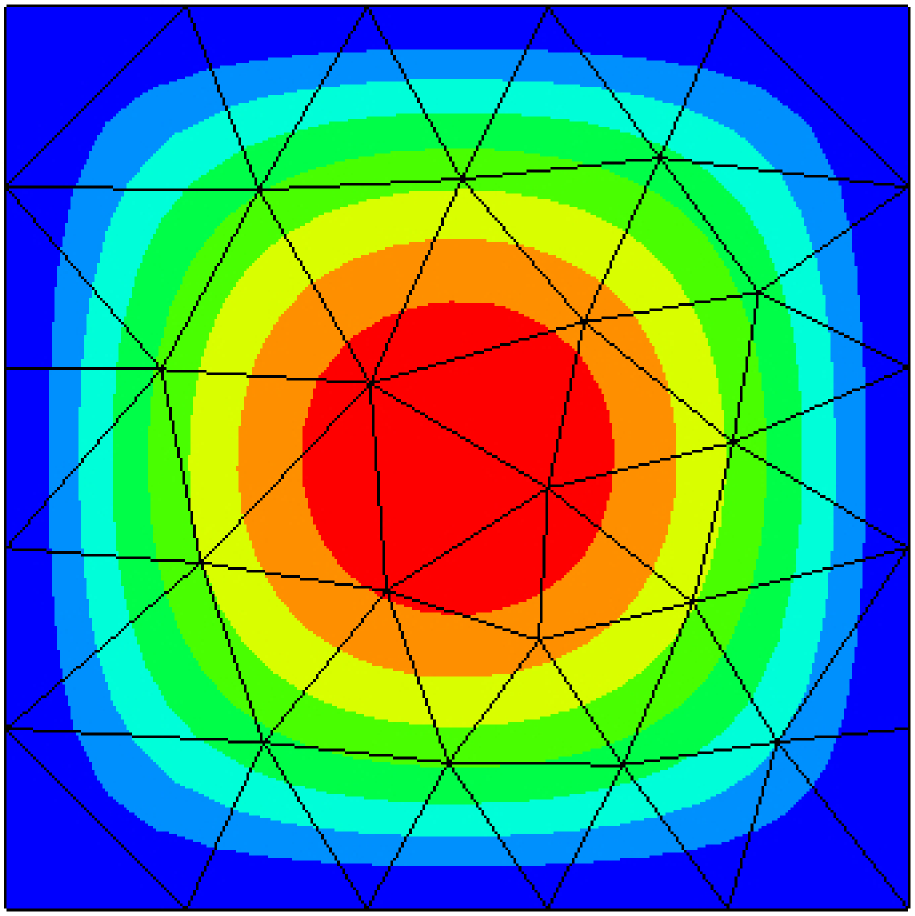
```
# generate a triangular mesh of mesh-size 0.2
mesh = Mesh (unit_square.GenerateMesh(maxh=0.2))

# H1-conforming finite element space
V = H1(mesh, order=3, dirichlet=[1,2,3,4])

# the right hand side
f = LinearForm (V)
f += Source (32 * (y*(1-y)+x*(1-x)))

# the bilinear-form
a = BilinearForm (V, symmetric=True)
a += Laplace (1)

a.Assemble()
f.Assemble()

# the solution field
u = GridFunction (V)
u.vec.data = a.mat.Inverse(V.FreeDofs(), inverse="sparsecholesky") * f.vec
# print (u.vec)

# plot the solution (netgen-gui only)
Draw (u)
Draw (-u.Deriv(), mesh, "Flux")
```

(continues on next page)

(continued from previous page)

```
exact = 16*x*(1-x)*y*(1-y)
print ("L2-error:", sqrt (Integrate ( (u-exact)*(u-exact), mesh)))
```

The solution visualized in Netgen:



CHAPTER 2

Adaptive mesh refinement

We are solving a stationary heat equation with highly varying coefficients. This example shows how to

- model a 2D geometry be means of line segments
- apply a Zienkiewicz-Zhu type error estimator. The flux is interpolated into an H(div)-conforming finite element space.
- loop over several refinement levels

download adaptive.py

```
from ngsolve import *
from netgen.geom2d import SplineGeometry

#   point numbers 0, 1, ... 11
#   sub-domain numbers (1), (2), (3)
#
#
#              7-------------6
#              |             |
#              |     (2)     |
#              |             |
#       3------4-------------5------2
#       |                           |
#       |             11            |
#       |            /  \           |
#       |          10 (3) 9         |
#       |            \  /     (1)   |
#       |             8             |
#       |                           |
#       0---------------------------1
#

def MakeGeometry():
    geometry = SplineGeometry()
```

(continues on next page)

(continued from previous page)

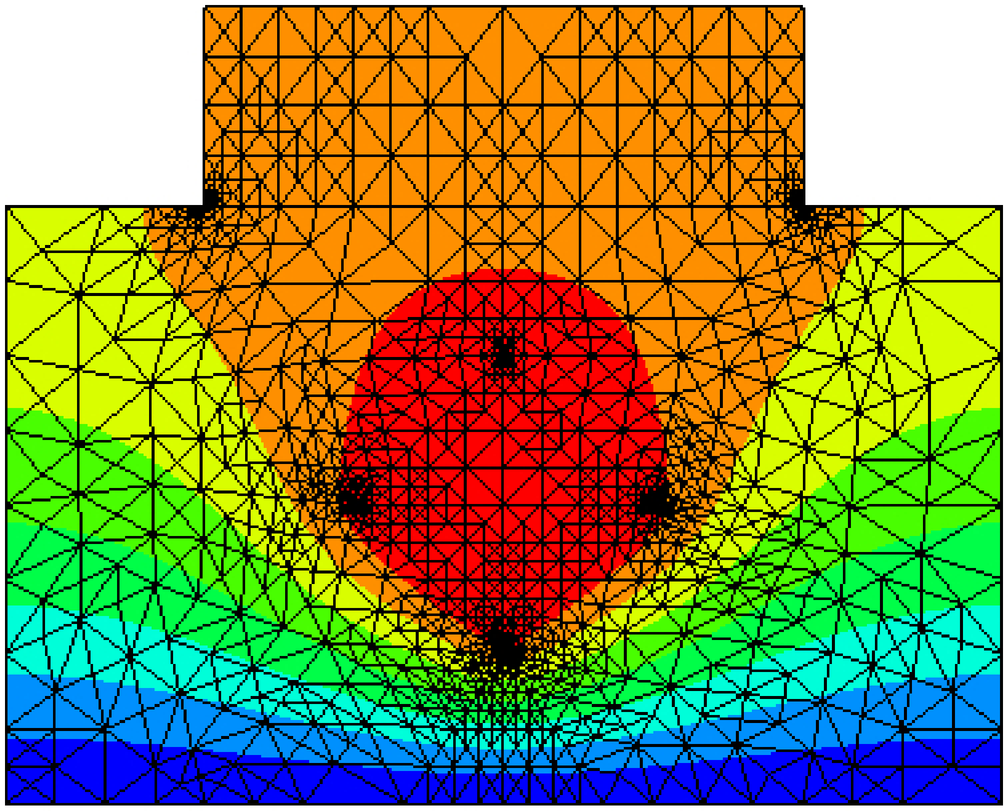
```
    # point coordinates ...
    pnts = [ (0,0), (1,0), (1,0.6), (0,0.6), \
             (0.2,0.6), (0.8,0.6), (0.8,0.8), (0.2,0.8), \
             (0.5,0.15), (0.65,0.3), (0.5,0.45), (0.35,0.3) ]
    pnums = [geometry.AppendPoint(*p) for p in pnts]

    # start-point, end-point, boundary-condition, domain on left side, domain on␣
↪right side:
    lines = [ (0,1,1,1,0), (1,2,2,1,0), (2,5,2,1,0), (5,4,2,1,2), (4,3,2,1,0), (3,0,2,
↪1,0), \
              (5,6,2,2,0), (6,7,2,2,0), (7,4,2,2,0), \
              (8,9,2,3,1), (9,10,2,3,1), (10,11,2,3,1), (11,8,2,3,1) ]

    for p1,p2,bc,left,right in lines:
        geometry.Append( ["line", pnums[p1], pnums[p2]], bc=bc, leftdomain=left,␣
↪rightdomain=right)
    return geometry


mesh = Mesh(MakeGeometry().GenerateMesh (maxh=0.2))

v = H1(mesh, order=3, dirichlet=[1])

# one heat conductivity coefficient per sub-domain
lam = DomainConstantCF([1, 1000, 10])
a = BilinearForm(v, symmetric=True)
a += Laplace(lam)


# heat-source in sub-domain 3
f = LinearForm(v)
f += Source(DomainConstantCF([0, 0, 1]))

c = Preconditioner(a, type="multigrid", flags= { "inverse" : "sparsecholesky" })

u = GridFunction(v)

# the boundary value problem to be solved on each level
bvp = BVP(bf=a, lf=f, gf=u, pre=c)

# finite element space and gridfunction to represent
# the heatflux:
space_flux = HDiv(mesh, order=2)
gf_flux = GridFunction(space_flux, "flux")


def SolveBVP():
    v.Update()
    u.Update()
    a.Assemble()
    f.Assemble()
    bvp.Do()
    Draw (u)



l = []
```

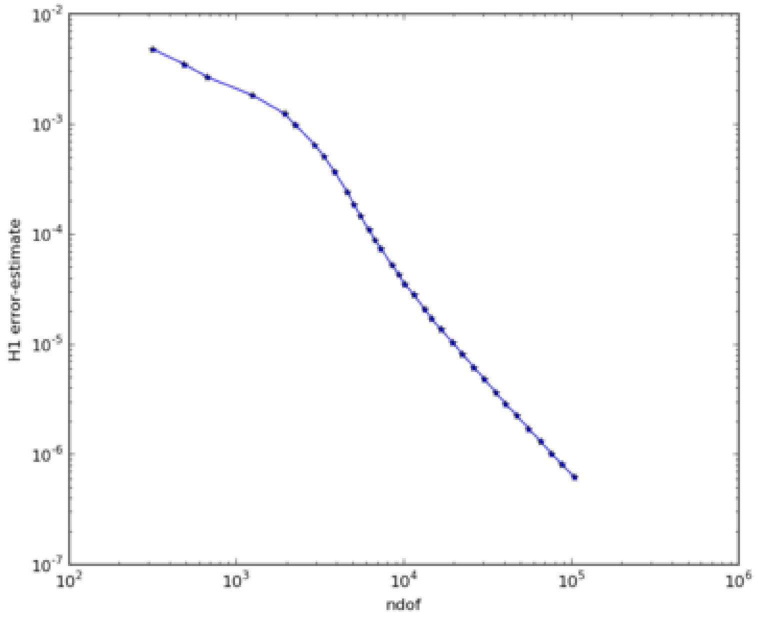
(continues on next page)

(continued from previous page)

```
def CalcError():
    space_flux.Update()
    gf_flux.Update()

    flux = lam * u.Deriv()
    # interpolate finite element flux into H(div) space:
    gf_flux.Set (flux)

    # Gradient-recovery error estimator
    err = 1/lam*(flux-gf_flux)*(flux-gf_flux)
    elerr = Integrate (err, mesh, VOL, element_wise=True)

    maxerr = max(elerr)
    l.append ( (v.ndof, sqrt(sum(elerr)) ))
    print ("maxerr = ", maxerr)

    for el in mesh.Elements():
        mesh.SetRefinementFlag(el, elerr[el.nr] > 0.25*maxerr)



while v.ndof < 100000:
    SolveBVP()
    CalcError()
    mesh.Refine()

SolveBVP()




import matplotlib.pyplot as plt

plt.yscale('log')
plt.xscale('log')
plt.xlabel("ndof")
plt.ylabel("H1 error-estimate")
ndof,err = zip(*l)
plt.plot(ndof,err, "-*")

plt.ion()
plt.show()

input("<press enter to quit>")
```

The solution on the adaptively refined mesh, and the convergence plot from matplotlib:




H1 error-estimate
ndof
10^{-2}
10^{-3}
10^{-4}
10^{-5}
10^{-6}
10^{-7}
10^{2}
10^{3}
10^{4}
10^{5}
10^{6}

CHAPTER 3

Symbolic definition of forms : magnetic field

We compute the magnetic field generated by a coil placed on a C-core. We see how to

- model a 3D constructive solid geometry
- set material properties from domain and boundary labels
- define bilinear- and linear forms symbolically using trial- and testfunctions

download cmagnet.py

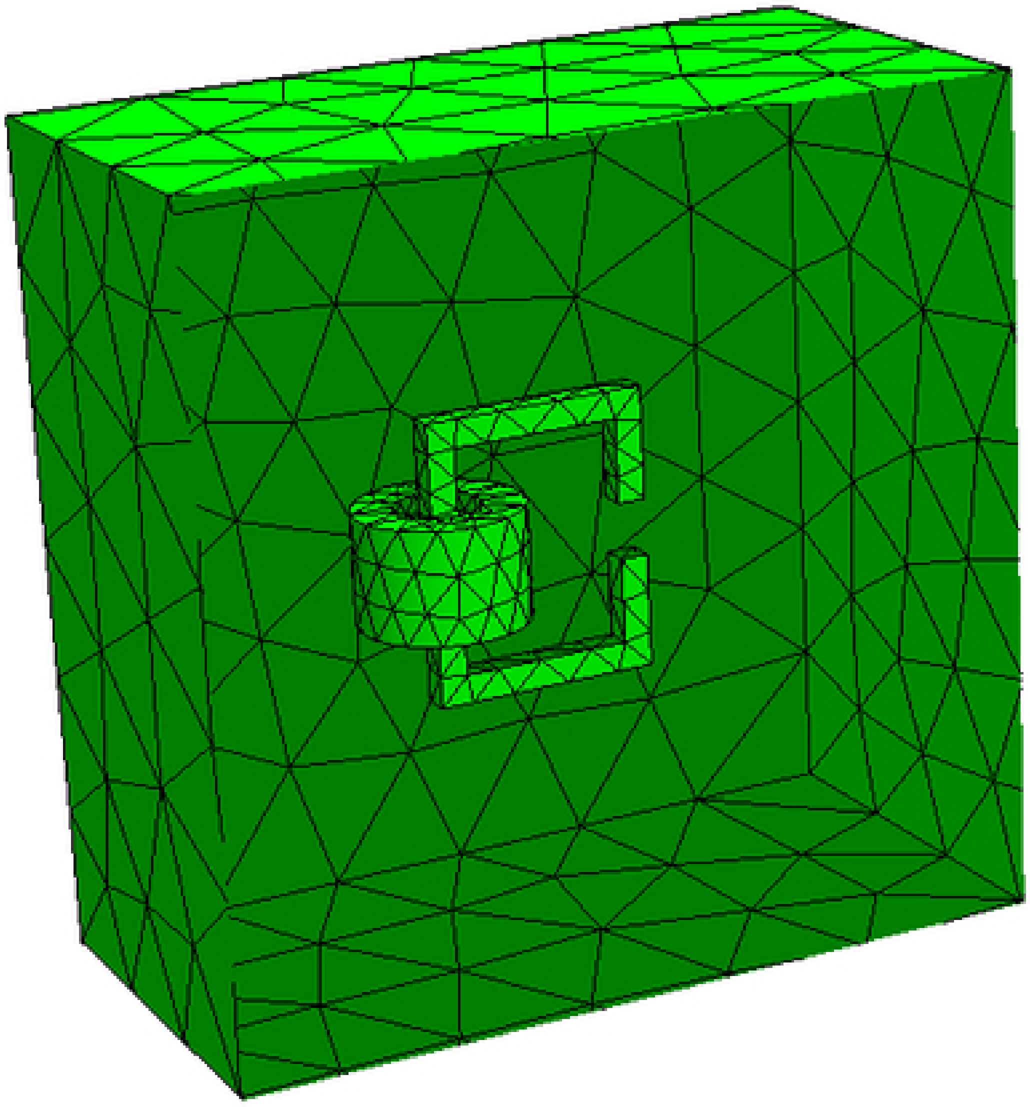
```
from netgen.csg import *
from ngsolve import *


def MakeGeometry():
    geometry = CSGeometry()
    box = OrthoBrick(Pnt(-1,-1,-1),Pnt(2,1,2)).bc("outer")

    core = OrthoBrick(Pnt(0,-0.05,0),Pnt(0.8,0.05,1))- \
           OrthoBrick(Pnt(0.1,-1,0.1),Pnt(0.7,1,0.9))- \
           OrthoBrick(Pnt(0.5,-1,0.4),Pnt(1,1,0.6)).maxh(0.2).mat("core")

    coil = (Cylinder(Pnt(0.05,0,0), Pnt(0.05,0,1), 0.3) - \
            Cylinder(Pnt(0.05,0,0), Pnt(0.05,0,1), 0.15)) * \
            OrthoBrick (Pnt(-1,-1,0.3),Pnt(1,1,0.7)).maxh(0.2).mat("coil")

    geometry.Add ((box-core-coil).mat("air"))
    geometry.Add (core)
    geometry.Add (coil)
    return geometry



ngmesh = MakeGeometry().GenerateMesh(maxh=0.5)
ngmesh.Save("coil.vol")
mesh = Mesh(ngmesh)
```

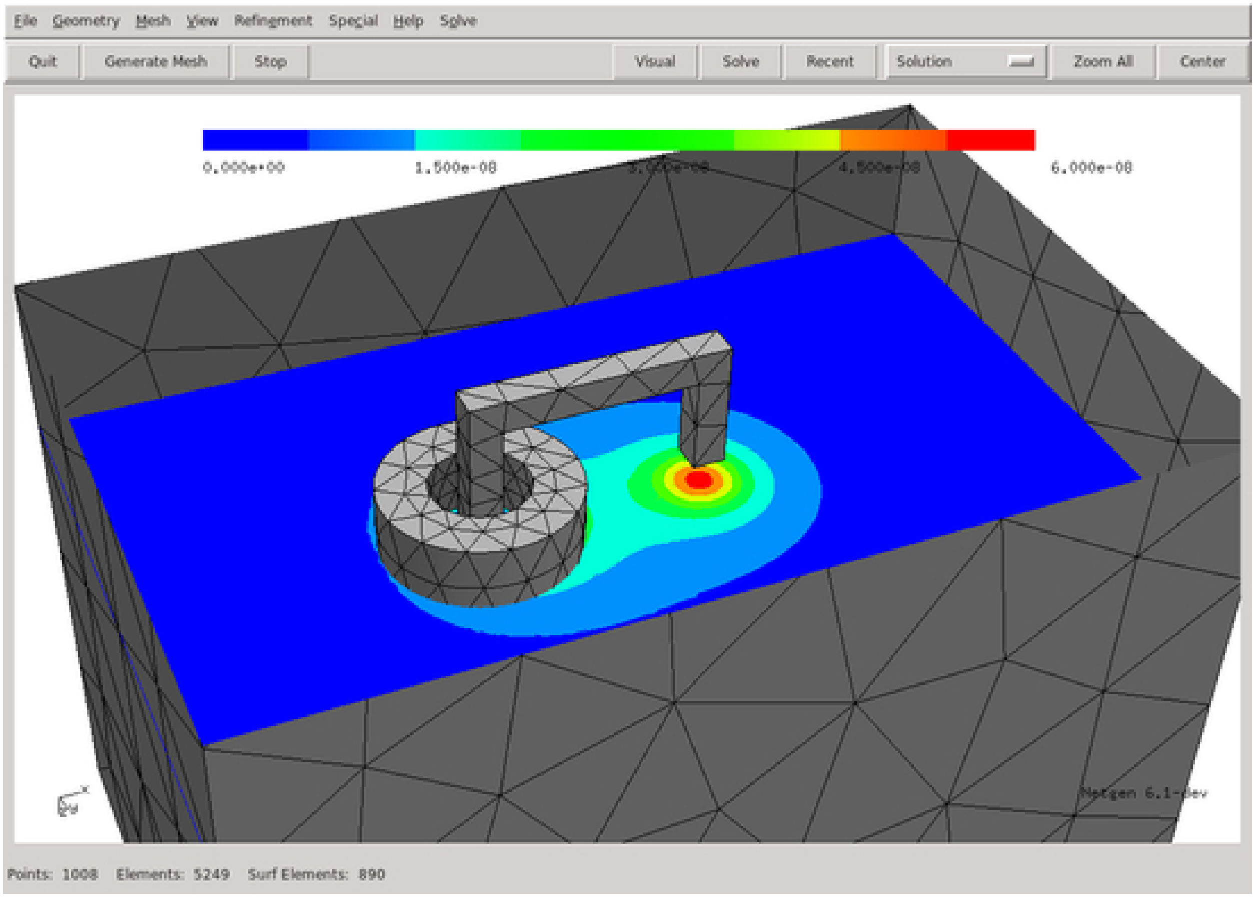
(continues on next page)

(continued from previous page)

```
# curve elements for geometry approximation
mesh.Curve(5)

ngsglobals.msg_level = 5

V = HCurl(mesh, order=4, dirichlet="outer", flags = { "nograds" : True })

# u and v refer to trial and test-functions in the definition of forms below
u = V.TrialFunction()
v = V.TestFunction()

mur = { "core" : 1000, "coil" : 1, "air" : 1 }
mu0 = 1.257e-6
nu_coef = [ 1/(mu0*mur[mat]) for mat in mesh.GetMaterials() ]
print ("nu_coef=", nu_coef)

nu = DomainConstantCF (nu_coef)
a = BilinearForm(V, symmetric=True)
a += SymbolicBFI(nu*curl(u)*curl(v) + 1e-6*nu*u*v)

c = Preconditioner(a, type="bddc")
# c = Preconditioner(a, type="multigrid", flags = { "smoother" : "block" } )
a.Assemble()

coil_domains = [ i for i,mat in enumerate(mesh.GetMaterials()) if mat == "coil" ]
f = LinearForm(V)
f += SymbolicLFI(CoefficientFunction((y,-x,0)) * v, definedon=coil_domains)
f.Assemble()

u = GridFunction(V)

solver = CGSolver(mat=a.mat, pre=c.mat)
u.vec.data = solver * f.vec


Draw (u.Deriv(), mesh, "B-field", draw_surf=False)
```

Mesh and magnitude of magnetic field visualized in Netgen:





Points: 1008 Elements: 5249 Surf Elements: 890

CHAPTER 4

Navier Stokes Equation

We solve the time-dependent incompressible Navier Stokes Equation. For that

- we use the P3/P2 Taylor-Hood mixed finite element pairing
- and perform operator splitting time-integration with the non-linear term explicit, but time-dependent Stokes fully implicit.

The example is from the Schäfer-Turek benchmark: a two-dimensional cylinder, at Reynolds number 100

download navierstokes.py

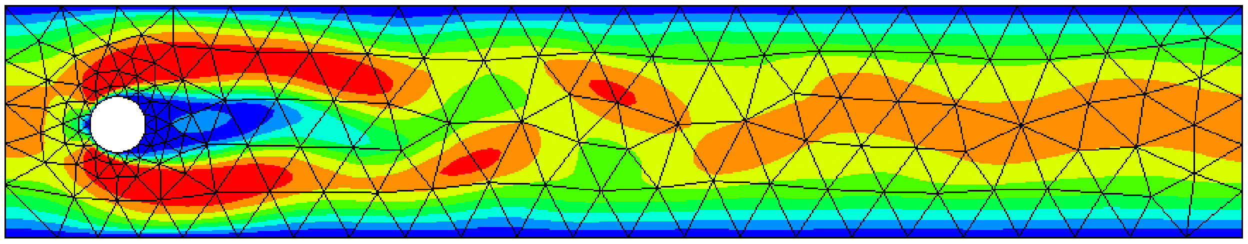
```
from ngsolve import *

# viscosity
nu = 0.001

# timestepping parameters
tau = 0.001
tend = 10


from netgen.geom2d import SplineGeometry
geo = SplineGeometry()
geo.AddRectangle( (0, 0), (2, 0.41), bcs = ("wall", "outlet", "wall", "inlet"))
geo.AddCircle ( (0.2, 0.2), r=0.05, leftdomain=0, rightdomain=1, bc="cyl")
mesh = Mesh( geo.GenerateMesh(maxh=0.08))

mesh.Curve(3)

V = H1(mesh,order=3, dirichlet="wall|cyl|inlet")
Q = H1(mesh,order=2)

X = FESpace([V,V,Q])

ux,uy,p = X.TrialFunction()
vx,vy,q = X.TestFunction()
```

(continues on next page)

(continued from previous page)

```
div_u = grad(ux)[0]+grad(uy)[1]
div_v = grad(vx)[0]+grad(vy)[1]

stokes = nu*grad(ux)*grad(vx)+nu*grad(uy)*grad(vy)+div_u*q+div_v*p # - 1e-6*p*q
a = BilinearForm(X)
a += SymbolicBFI(stokes)
a.Assemble()

# nothing here ...
f = LinearForm(X)
f.Assemble()


u = GridFunction(X)

# parabolic inflow at inlet:
uin = 1.5*4*y*(0.41-y)/(0.41*0.41)
u.components[0].Set(uin, definedon=mesh.Boundaries("inlet"))


# solve Stokes problem for initial conditions:
inv_stokes = a.mat.Inverse(X.FreeDofs())

res = f.vec.CreateVector()
res.data = f.vec - a.mat*u.vec
u.vec.data += inv_stokes * res


# matrix for implicit Euler
mstar = BilinearForm(X)
mstar += SymbolicBFI(ux*vx+uy*vy + tau*stokes)
mstar.Assemble()
inv = mstar.mat.Inverse(X.FreeDofs())

velocity = CoefficientFunction (u.components[0:2])
absvelocity = sqrt(velocity*velocity)

# the non-linear term
conv = LinearForm(X)
conv += SymbolicLFI(velocity*grad(u.components[0])*vx)
conv += SymbolicLFI(velocity*grad(u.components[1])*vy)


# implicit Euler/explicit Euler splitting method:
t = 0
while t < tend:
    print ("t=", t)

    conv.Assemble()           # calculate it
    res.data = a.mat * u.vec + conv.vec
    u.vec.data -= tau * inv * res

    t = t + tau
    Draw (absvelocity, mesh, "velocity")
```

The absolute value of velocity:



CHAPTER 5

Nonlinear elasticity

We solve the geometric nonlinear elasticity equation using a hyper-elastic energy density. We solve the stationary equation using the incremental load method.

The example teaches how to

- define a non-linear variational formulation using SymbolicEnerg
- solve it by Newton's method
- use a parameter for increasing the load

download elasticity.py

```
import netgen.geom2d as geom2d
from ngsolve import *

geo = geom2d.SplineGeometry()
pnums = [ geo.AddPoint (x,y,maxh=0.01) for x,y in [(0,0), (1,0), (1,0.1), (0,0.1)] ]
for p1,p2,bc in [(0,1,"bot"), (1,2,"right"), (2,3,"top"), (3,0,"left")]:
     geo.Append(["line", pnums[p1], pnums[p2]], bc=bc)
mesh = Mesh(geo.GenerateMesh(maxh=0.05))


E, nu = 210, 0.2
mu  = E / 2 / (1+nu)
lam = E * nu / ((1+nu)*(1-2*nu))

fes = H1(mesh, order=2, dirichlet="left", dim=mesh.dim)

u  = fes.TrialFunction()

force = CoefficientFunction( (0,1) )

# some utils ...
def IdentityCF(dim):
    return CoefficientFunction( tuple( [1 if i==j else 0 for i in range(dim) for j in␣
↪range(dim)]), dims=(dim,dim) )
```

(continues on next page)

(continued from previous page)

```
def Trace(mat):
    return sum( [mat[i,i] for i in range(mat.dims[0]) ])

def Det(mat):
    if mat.dims[0] == 2:
        return mat[0,0]*mat[1,1]-mat[0,1]*mat[1,0]

I = IdentityCF(mesh.dim)
F = I + u.Deriv()    # attention: row .. component, col .. derivative
C = F * F.trans
E = 0.5 * (C-I)

def Pow(a, b):
    return exp (log(a)*b)

def NeoHook (C):
    return 0.5 * mu * (Trace(C-I) + 2*mu/lam * Pow(Det(C), -lam/2/mu) - 1)




factor = Parameter(0.1)

a = BilinearForm(fes, symmetric=False)
a += SymbolicEnergy(  NeoHook (C).Compile() )
a += SymbolicEnergy(  (-factor * InnerProduct(force,u) ).Compile() )


u = GridFunction(fes)
u.vec[:] = 0

res = u.vec.CreateVector()
w = u.vec.CreateVector()


for loadstep in range(50):

    print ("loadstep", loadstep)
    factor.Set ((loadstep+1)/10)

    for it in range(5):
        print ("Newton iteration", it)
        print ("energy = ", a.Energy(u.vec))
        a.Apply(u.vec, res)
        a.AssembleLinearization(u.vec)
        inv = a.mat.Inverse(fes.FreeDofs() )
        w.data = inv*res
        print ("err^2 = ", InnerProduct (w,res))
        u.vec.data -= w

    Draw (u, mesh, "displacement")
    SetVisualization (deformation=True)
    input ("<press a key>")
```

Installation instructions for Windows/Mac/Linux

Tutorial on Using NGSpy by Jay Gopalakrishnan

CHAPTER 6

Working with CoefficientFunctions

A CoefficientFunction is a function which can be evaluated on a mesh, and may be used to provide the coefficient or a right-hind-side to the variational formulation. As typical finite element procedures iterate over elements, and map integration points from a reference element to a physical element, the evaluation of a CoefficientFunction requires a mapped integration point. The mapped integration point contains the coordinate, as well as the Jacobian, but also element number and region index. This allows the efficient evaluation of a wide class of functions needed for finite element computations.

Some examples of using coefficient functions are:

```
a += Laplace (CoefficientFunction([1,17,5]))
f += Source (x*(1-x))
```

A *CoefficientFunction* initialized with an array stores one value per region, which may be a domain or a boundary region. Note that domain and boundary indices have 1-based counting. The Cartesian coordinates *x*, *y*, and *z* are pre-defined coordinate coefficient functions. Algebraic operations as - or * combine existing CoefficientFunctions to new ones.

A CoefficientFunction may be real or complex valued, and may be scalar or vector-valued. (Matrix valued CoefficientFunctions will be available in future versions.) Scalar CFs can be combined to vectorial CF by creating a new object from a tuple, and components of a vectorial CF can be accessed by the bracket operator:

```
fx = CoefficientFunction (0)
fy = CoefficientFunction ([1,2,3])
vec_cf = CoefficientFunction( (fx,fy) )
fx_again = vec_cf[0]
```

Since CFs require a mapped integration-point as argument, we first have to generate one. Calling the mesh-object with x, y, and optionally z coordinates generates one for us. Note, this requires a global search for the element containing the point:

```
>>> mip = mesh(0.2,0.4)
>>> (x(mip), y(mip))
(0.2, 0.4)
```

Also a GridFunction is a CoefficientFunction by inheritance, and can be used whenever a CF is required. If the finite element space provides differentiation, then

```
u.Deriv()
```

creates a new CoefficientFunction. Depending on the space, *Deriv* means the gradient, the curl, or the divergence. Direct evaluation of higher order derivatives is not supported, but can be approximated via interpolation:

```
V1 = H1(mesh,order=5)
V2 = H1(mesh,order=4,dim=2)
u = GridFunction(V1)
du = GridFunction(V2)
u.Set(y*sin(3.1415*x))
du.Set(u.Deriv())
uxx,uxy,uyx,uyy = du.Deriv()
Draw (uxx, mesh, "uxx")
Draw (uxy-uyx, mesh, "not_perfect")
```

The symbolic definition of bilinear- and linear-forms uses also CoefficientFunctions, as in this example (*b* is a regular, vectorial CF):

```
u = V.TrialFunction()
v = V.TestFunction()
a += SymbolicBFI (b * u.Deriv() * v)
```

Here, *u* and *v* are so called ProxyFunctions, which do not evaluate to values, but are used as arguments in the definition of the forms. Also *u.Deriv()* is a ProxyFunction. When the form is evaluated in an integration point, the ProxyFunctions are set to unit-vectors, and the CoefficientFunction evaluates to an actual number.

CHAPTER 7

Setting inhomogeneous Dirichlet boundary conditions

A mesh stores boundary elements, which know the *bc* index given in the geometry. The Dirichlet boundaries are given as a list of boundary condition indices to the finite element space:

```
V = FESpace(mesh,order=3,dirichlet=[2,5])
u = GridFunction(V)
```

If bc-labels are used instead of numbers, the list of Dirichlet bc numbers can be generated as follows. Note that bc-nums are 1-based:

```
bcnums = [ i+1 for i,bcname in enumerate(mesh.GetBoundaries()) if bcname in ["dir1",
↪"dir2"] ]
```

The BitArray of free (i.e. unconstrained) dof numbers can be obtained via

```
freedofs = V.FreeDofs()
print (freedofs)
```

Inhomogeneous Dirichlet values are set via

```
u.Set(x*y, BND)
```

This function performs an element-wise L2 projection combined with arithmetic averaging of coupling dofs.

As usual we define biform a and liform f. Here, the full Neumann matrix and non-modified right hand sides are stored.

Boundary constraints are treated by the preconditioner. For example, a Jacobi preconditioner created via

```
c = Preconditioner(a, "local")
```

inverts only the unconstrained diagonal elements, and fills the remaining values with zeros.

The boundary value solver keeps the Dirichlet-values unchanged, and solves only for the free values

```
BVP(bf=a,lf=f,gf=u, pre=c).Do()
```

A do-it-yoursolve version of homogenization is:

```
res = f.vec.CreateVector()
res.data = f.vec - a.mat * u.vec
u.vec.data += a.mat.Inverse(v.FreeDofs()) * res
```

CHAPTER 8

Define and update preconditioners

A preconditioner is defined for a bilinear-form, and aims at providing a cheap, approximative inverse of the matrix. The matrix is restricted to the non-dirichlet (free) degrees of freedom, provided by the underlying FESpace.

The canonical way is to define the preconditioner after the bilinear-form, but before calling Assemble:

```
a = BilinearForm(fes)
a += SymbolicBFI(grad(u)*grad(v))
c = Preconditioner(a, "local")
a.Assemble()
```

The preconditioner registers itself with the bilinear-form. Whenever the form is updated, the preconditioner is updated as well.

You can define the preconditioner after assembling, but then you have to call manually c.Update()

The ratio if this ordering is that some preconditioners (e.g. bddc, amg, …) require access to the element-matrices, which are only available during assembling.

The preconditioners included in NGSolve are the following. Additional user-defined preconditioners can be implemented in plug-ins. An example is given in MyLittleNGSolve

name	preconditioner
local	Jacobi / block-Jacobi
direct	a sparse direct factorization
multigrid	h-version and high-order/low-order multigrid
bddc	p-version domain decomposition

CHAPTER 9

The Trace() operator

Mathematically, the trace operator restricts a domain-function to the boundary of the domain. For function spaces like H(curl) or H(div), the trace operator delivers only tangential, and normal components, respectively.

The trace operator is used in NGS-Py as follows:

```
a.Assemble(u.Trace()*v.Trace(), BND)
```

The evaluation of boundary values involves only degrees of freedom geometrically located on the boundary.

Traces of derivatives can are formed by either one of the following. For example, the derivative of the trace of an H1-function gives the tangential derivative:

```
u.Trace().Deriv()
u.Deriv().Trace()
```

As an popular exception, in NGS-Py, the Trace()-operator for H1-functions is optional.

For element-boundary integrals, the Trace()-operator must not be used. Here, the function is evaluated at a point in the volume-element.

CHAPTER 10

Vectors and matrices

NGSolve contains two different implementations of linear algebra: One deals with *dense* matrices which are typically small, the other one with typically large *sparse* matrices and linear operators as needed in the finite element method.

10.1 Large Linear Algebra

Grid-functions, bilinear-forms and linear-forms create vectors and matrices. You can query them using the *u.vec* or *bfa.mat* attributes. You can print them via *print (u.vec)*, or set values *gf.vec[0:10] = 3.7*.

You can create new vectors of the same size and the same element type via

```
vu = u.vec
help = vu.CreateVector()
```

You can perform vector-space operations

```
help.data = 3 * vu
help.data += mat * vu
print ("(u,h) = ", InnerProduct(help, vu))
```

There are a few things to take care of:

- Use the *.data* attribute to write to an existing vector. The expression *help = 3 * vu* will redefine the object *help* to something like the symbolic expression *product of scalar times vector*.
- You can combine certain operations (e.g. *help.data = 3 * u1 + 4 * u2 + mat * u4*), but not arbitrary operations (as *help.data = mat * (u1+u2)* or *help.data = mat1 * mat2 * u*). The ratio behind is that the operations must be computable without allocating temporary vectors.

You can also work with NGSolve-Data from numpy/scipy, see also here ngspy-numpy ## Small Linear Algebra

With *x = Vector(5)* and *m = Matrix(3,5)* you create a vector and a matrix. You can access elements with brackets, and perform linear algebra operations. *y = m * x* defines a new vector y.

ngsolve provides elementary matrix-vector operations. For other operations, we recommend to use the numpy package. With *m.NumPy()* you get a multi-dimensional numpy array sharing the memory of the matrix m.

CHAPTER 11

Static condensation of internal bubbles

Element-internal unknowns of higher order finite elements can be eliminated from the global linear system. This corresponds to the Schur-complement system for the coupling unknowns on the element boundaries.

To enable this option, set the *eliminate_internal* flag for the bilinear-form. This assembles the global Schur-complement system, and stores harmonic extensions, and internal inverses.

The user is responsible to transform the right-hand side and the solution vector. An example is as follows:

```
from netgen.geom2d import unit_square
from ngsolve import *

mesh = Mesh(unit_square.GenerateMesh(maxh=0.3))

fes = H1(mesh, order=10, dirichlet=[1,2])
u = fes.TestFunction()
v = fes.TrialFunction()

a = BilinearForm(fes, flags = { "eliminate_internal" : True } )
a += SymbolicBFI (grad(u) * grad(v))
a.Assemble()

f = LinearForm(fes)
f += SymbolicLFI (1 * v)
f.Assemble()

u = GridFunction(fes)
```

modify right-hand side

```
f.vec.data += a.harmonic_extension_trans * f.vec
```

solve for external unknowns

```
u.vec.data = a.mat.Inverse(fes.FreeDofs(True)) * f.vec
```

and find element-internal solution

```
u.vec.data += a.harmonic_extension * u.vec
u.vec.data += a.inner_solve * f.vec

Draw (u)
```

CHAPTER 12

Discontinuous Galerkin methods

Discontinuous Galerkin (DG) methods have certain advantages: One can apply upwinding for convection dominated problems, and explicit time-stepping methods are cheap due to block-diagonal or even diagonal mass matrices.

The bilinear-form of a DG method involves integrals over functions defined on neighbouring elements. In NGS-Py we can access the neighbouring element via the *.Other()* method. The following example gives the boundary integral of an upwind scheme for the convection equation. The *CoefficientFunction b* is a given vector-field, the wind. With specialcf.normal(2) we get the outer element normal vector. We have to provide the space dimension of the mesh. Depending on the sign of <*b,n*> we choose the trial function from the current element, or from the neighbour using the *IfPos* function. If the edge is on the domain-boundary, a given boundary value may be specified by the *bnd* argument:

```
b = CoefficientFunction( (y-0.5,0.5-x) )
bn = b*specialcf.normal(2)
a += SymbolicBFI (bn*IfPos(bn, u, u.Other(bnd=ubnd)) * v, element_boundary=True)
```

One can also use the test-function from the neighbour element. The coefficient functions are always evaluated on the current element.

When one works with assembled matrices, there is a drawback of DG methdos: The matrix stencil becomes larger. We have to tell NGSolve to reserver more entries in the matrix using

```
FESpace( ... , flags = { "dgjumps" : True })
```

If we don't assemble the matrix, but work with operator application on the fly, we don't have to specify it.

The above expression leads to a loop over elements, and the boundary integrals are evaluated for the whole element-boundary, which consists of internal and boundary edges (or faces). But sometimes we need different terms for internal and boundary facets. Here we can use the *skeleton* flag. This leads to separate loops for internal and boundary facets. The following definition is mathematically equivalent to the method above. The VOL and BND specifier tell whether we want to loop over internal or boundary edges. Since here every edge is processed only once, the skeleton formulation is slightly more efficient:

```
a += SymbolicBFI ( bn*IfPos(bn, u, u.Other()) * (v-v.Other()), VOL, skeleton=True)
a += SymbolicBFI ( bn*IfPos(bn, u, ubnd) * v, BND, skeleton=True)
```

To solve with the block-diagonal (or even diagonal) mass matrix of an *L2* -finite element space, we can use the *SolveM* method of the FESpace. The *rho* argument allows to specify a density coefficientfunction for the mass-matrix. The operation is performed inplace for the given vector.

```
density = CoefficientFunction(1)
fes.SolveM (rho=density, vec=u)
```

Several examples of DG methods are given in the DG directory of the py_tutorials.

CHAPTER 13

Parallel computing with NGS-Py

There are several options to run NGS-Py in parallel, either in a shared-memory, or distributed memory paradigm.

13.1 Shared memory parallelisation

NGSolve shared memory parallelisation is based on a the task-stealing paradigm. On entering a parallel execution block, worker threads are created. The master thread executes the algorithm, and whenever a parallelized function is executed, it creates tasks. The waiting workers pick up and process these tasks. Since the threads stay alive for a longer time, these paradigm allows to parallelize also very small functions, practically down to the range of 10 micro seconds.

The task parallelization is also available in NGS-Py. By the *with Taskmanager* statement one creates the threads to be used in the following code-block. At the end of the block, the threads are stopped.

```
with Taskmanager():
    a = BilinearForm(fespace)
    a += SymbolicBFI(u*v)
    a.Assemble()
```

Here, the assembling operates in parallel. The finite element space provides a coloring such that elements of the same color can be processed simultaneously. Also helper functions such as sparse matrix graph creation uses parallel loops.

Another typical example for parallel execution are equation solvers. Here is a piece of code of the conjugate gradient solver from NGS-Py:

```
with Taskmanager():

  ...
  for it in range(maxsteps):
      w.data = mat * s
      wd = wdn
      as_s = InnerProduct (s, w)
      alpha = wd / as_s
```

(continues on next page)

(continued from previous page)

```
      u.data += alpha * s
      d.data += (-alpha) * w
```

The master thread executes the algorithm. In matrix - vector product function calls, and also in vector updates and innner products tasks are created and picked up by workers.

13.2 Distributed memory

The distributed memory paradigm requires to build Netgen as well as NGSolve with MPI - support, which must be enabled during the cmake configuration step.

CHAPTER 14

Interfacing to numpy/scipy

In some occasions or for some users it might be interesting to access NGSolve data from python in a fashion which is compatible with numpy and/or scipy. We give a few examples of possible use cases.

14.1 Working with small vectors and dense matrices:

see [Vectors and matrices](ngspy-howto-linalg)

14.2 Working with large vectors

You can get a "view" on an NGSolve-BaseVector using *.FV()* (which will give you a FlatVector) combined with *.NumPy()* which will give a numpy array which operates on the NGSolve-Vector-Data. For example the following works, assuming b to be an NGSolve-Vector:

```
b.FV().NumPy()[:] = abs(b.FV().NumPy()) - 1.0
```

which will give you the component-wise operation (absolute value minus one) applied on the vector b. During this operation data does not need to be copied.

14.3 Working with sparse matrices

You can access the sparse matrix information of a BaseMatrix using

```
rows,cols,vals = a.mat.COO()
```

Note that a bilinear form with flag *symmetric==True* will only give you one half of the matrix. These information can be put into a standard scipy-matrix, e.g. with

```
import scipy.sparse as sp
A = sp.csr_matrix((vals,(rows,cols)))
```

You can use this, for instance, to examine the sparsity pattern of your matrix:

```
import matplotlib.pylab as plt
plt.spy(A)
plt.show()
```

or to compute the condition number (note that we export to a dense matrix here):

```
import numpy as np
np.linalg.cond(A.todense())
```

14.4 Using iterative solvers from scipy

To use iterative solvers from scipy we have to wrap a *LinearOperator* around the NGSolve-matrix. The crucial component is the application of matrix vector product. Here is a very simple example where the scipy-cg-solver is used to solve the linear system (no preconditioner, no Dirichlet-dofs):

```
import scipy
import scipy.sparse.linalg

tmp1 = f.vec.CreateVector()
tmp2 = f.vec.CreateVector()
def matvec(v):
    tmp1.FV().NumPy()[:] = v
    tmp2.data = a.mat * tmp1
    return tmp2.FV().NumPy()

A = scipy.sparse.linalg.LinearOperator( (a.mat.height,a.mat.width), matvec)

u.vec.FV().NumPy()[:], succ = scipy.sparse.linalg.cg(A, f.vec.FV().NumPy())
```

You can also use a sparse matrix format from python to run to previous example, see above. However, for preconditioning actions a sparse matrix is not necessarily set up such that the *LinearOperator* is often more useful.

- [Iterating over elements]

CHAPTER 15

Define and mesh 2D geometries

Netgen-python allows to define 2D geometries by means of boundary curves. Curves can be lines, or rational splines of 2nd order.

```
import netgen.geom2d as
```

We generate a geometry object, and add geometry control points:

```
geo = geom2d.SplineGeometry()
p1 = geo.AppendPoint (0,0)
p2 = geo.AppendPoint (1,0)
p3 = geo.AppendPoint (1,1)
p4 = geo.AppendPoint (0,1)
```

p1 to p4 are indices referring to these points. Next, we add the 4 sides of the square:

```
geo.Append (["line", p1, p2])
geo.Append (["line", p2, p3])
geo.Append (["line", p3, p4])
geo.Append (["line", p4, p1])
```

The lines are oriented such that the domain is on the left side when going from the first point to the second point of the line.

We generate a mesh, where the maxh arguments specifies the desired maximal global mesh-size.

```
mesh = geo.GenerateMesh (maxh=0.1)
```

You can iterate over points and elements of the mesh:

```
for p in mesh.Points():
 x,y,z = p.p
 print ("x = ", x, "y = ", y)
 for el in mesh.Elements2D():
   print (el.vertices)
```

15.1 Curved boundaries

Instead of line segments, we can use second order rational splines, which give an exact representation of ellipses. They are described by three points defining the control polygon. The curve and its tangential direction coincides with the control polygon at the start and the end-point.

The following defines a quarter of a circle:

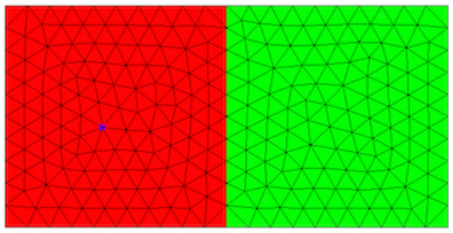
```
geo = geom2d.SplineGeometry()
p1,p2,p3,p4 = [ geo.AppendPoint(x,y) for x,y in [(0,0), (1,0), (1,1), (0,1)] ]
geo.Append (["line", p1, p2])
geo.Append (["spline3", p2, p3, p4])
geo.Append (["line", p4, p1])
```

15.2 Multiple subdomains

As default, the domain is left of the segment. If you have several subdomains, you specify the subdomain numbers left and right of the segment. Outside is given as subdomain number 0. Default values for leftdomain and rightdomain are 1 and 0, respectively. The following example gives two squares:

```
geo = geom2d.SplineGeometry()
p1,p2,p3,p4 = [ geo.AppendPoint(x,y) for x,y in [(0,0), (1,0), (1,1), (0,1)] ]
p5,p6 =  [ geo.AppendPoint(x,y) for x,y in [(2,0), (2,1)] ]
geo.Append (["line", p1, p2], leftdomain=1, rightdomain=0)
geo.Append (["line", p2, p3], leftdomain=1, rightdomain=2)
geo.Append (["line", p3, p4], leftdomain=1, rightdomain=0)
geo.Append (["line", p4, p1], leftdomain=1, rightdomain=0)
geo.Append (["line", p2, p5], leftdomain=2, rightdomain=0)
geo.Append (["line", p5, p6], leftdomain=2, rightdomain=0)
geo.Append (["line", p6, p3], leftdomain=2, rightdomain=0)
```

The obtained mesh looks like below. You get the red color by double-clicking into the subdomain.



15.3 Boundary condition markers

You can specify a boundary condition number for each segment, which can be used to specify boundary conditions in the simulation software. It must be a positive integer.

```
geo.Append (["line", p1, p2], bc=3)
```

15.4 New features since Nov 2, 2015:

You can set segment-wise and domain-wise maximal mesh-size:

```
geo.Append (["line", p1, p2], maxh=0.1)
geo.SetDomainMaxH(2, 0.01)
```

You can give labels for boundary conditions and domains materials:

```
geo.Append (["line", p1, p2], bc="bottom")
geo.SetMaterial(2, "iron")
```

You have templates for defining circles and rectangles:

```
geo.AddCircle(c=(5,0), r=0.5, leftdomain=2, rightdomain=1)
geo.AddRectangle((0,0), (3,2))
```

15.5 New feature since Nov 26, 2015:

Generation of quad-dominated 2D meshes:

```
geo.GenerateMesh(maxh=..., quad_dominated=True)
```

CHAPTER 16

Constructive Solid Geometry CSG

The constructive solid geometry format allows to define geometric primitives such as spheres and cylinders and perform to boolean operations on them. Such objects are of type Solid.

A cube an be defined and meshed as follows:

```python
from netgen.csg import *

left  = Plane (Pnt(0,0,0), Vec(-1,0,0) )
right = Plane (Pnt(1,1,1), Vec( 1,0,0) )
front = Plane (Pnt(0,0,0), Vec(0,-1,0) )
back  = Plane (Pnt(1,1,1), Vec(0, 1,0) )
bot   = Plane (Pnt(0,0,0), Vec(0,0,-1) )
top   = Plane (Pnt(1,1,1), Vec(0,0, 1) )

cube = left * right * front * back * bot * top
geo = CSGeometry()
geo.Add (cube)

mesh = geo.GenerateMesh(maxh=0.1)
mesh.Save("cube.vol")
```

The Plane primitive defines the half-space behind the plane given by an arbitrary point in the plane, and the outgoing normal vector.

Boolean operations are defined as follows:

operator	set operation
*	intersection
+	union
-	intersection with complement

Available primitives are

primitive	csg syntax	meaning
half-space	Plane(Pnt p, Vec n)	point p in plane, normal vector
sphere	Sphere(Pnt c,float r)	sphere with center c and radius r
cylinder	Cylinder(Pnt a, Pnt b, float r)	points a and b define the axes of a infinite cylinder of radius r
brick	OrthoBrick (Pnt a, Pnt b)	axes parallel brick with minimal coordinates a and maximal coordinates b

Using the orthobrick primitive, the cube above can be defined by one statement. Drilling a hole through the cube is defined as follows:

```
from netgen.csg import *

cube = OrthoBrick( Pnt(0,0,0), Pnt(1,1,1) )
hole = Cylinder ( Pnt(0.5, 0.5, 0), Pnt(0.5, 0.5, 1), 0.2)

geo = CSGeometry()
geo.Add (cube-hole)
mesh = geo.GenerateMesh(maxh=0.1)
mesh.Save("cube_hole.vol")
```

If the whole domain consists of several regions, we give several solid to the geometry object. Now, the first domain is the cube with the hole cut out, the second region is the hole. Don't forget that the cylinder is an infinite cylinder, and must be cut to finite length:

```
geo = CSGeometry()
geo.Add (cube-hole)
geo.Add (cube*hole)
```

CHAPTER 17

Working with meshes

In this example we create two geometries (a cube and a sphere fitting inside), and mesh them. Then, we manually merge the surface meshes, and create a unified volume mesh, where the sphere and its complement are two different sub-domains.

```python
from netgen.meshing import *
from netgen.csg import *

# generate brick and mesh it
geo1 = CSGeometry()
geo1.Add (OrthoBrick( Pnt(0,0,0), Pnt(1,1,1) ))
m1 = geo1.GenerateMesh (maxh=0.1)
m1.Refine()

# generate sphere and mesh it
geo2 = CSGeometry()
geo2.Add (Sphere (Pnt(0.5,0.5,0.5), 0.1))
m2 = geo2.GenerateMesh (maxh=0.05)
m2.Refine()
m2.Refine()

print ("***************************")
print ("** merging suface meshes **")
print ("***************************")

# create an empty mesh
mesh = Mesh()

# a face-descriptor stores properties associated with a set of surface elements
# bc .. boundary condition marker,
# domin/domout .. domain-number in front/back of surface elements (0 = void)

fd_outside = mesh.Add (FaceDescriptor(bc=1,domin=1))
fd_inside = mesh.Add (FaceDescriptor(bc=2,domin=2,domout=1))
```

(continues on next page)

(continued from previous page)

```
# copy all boundary points from first mesh to new mesh.
# pmap1 maps point-numbers from old to new mesh

pmap1 = { }
for e in m1.Elements2D():
for v in e.vertices:
if (v not in pmap1):
pmap1[v] = mesh.Add (m1[v])


# copy surface elements from first mesh to new mesh
# we have to map point-numbers:

for e in m1.Elements2D():
mesh.Add (Element2D (fd_outside, [pmap1[v] for v in e.vertices]))



# same for the second mesh:

pmap2 = { }
for e in m2.Elements2D():
for v in e.vertices:
if (v not in pmap2):
pmap2[v] = mesh.Add (m2[v])

for e in m2.Elements2D():
mesh.Add (Element2D (fd_inside, [pmap2[v] for v in e.vertices]))


print ("******************")
print ("** merging done **")
print ("******************")


mesh.GenerateVolumeMesh()
mesh.Save ("newmesh.vol")
```

CHAPTER 18

Manual mesh generation

In this example we create a mesh by hand, i.e. by prescribing all vertices, edges, faces and elements on our own. This example creates a structured grid for the unit square [0,1]x[0,1] using triangles or quadrilateral (choice via switch):

We first include what we need from netgen:

```
from netgen.geom2d import unit_square, MakeCircle, SplineGeometry
from netgen.meshing import Element0D, Element1D, Element2D, MeshPoint, FaceDescriptor,
↪ Mesh
from netgen.csg import Pnt
```

Next, we decide on the parameters for the mesh:

```
quads = True
N=5
```

We create an empty mesh and initialize the geometry and the dimension:

```
mesh = Mesh()
mesh.SetGeometry(unit_square)
mesh.dim = 2
```

Then, we add all mesh points that we will need for the final mesh. Note that these MeshPoints are added to the mesh with ‘mesh.Add(..)’ which return the point index. This index is then stored in the array ‘pnums’. In our case we have the simple structure that we will have (N+1)*(N+1) points in total.

```
pnums = []
for i in range(N + 1):
    for j in range(N + 1):
        pnums.append(mesh.Add(MeshPoint(Pnt(i / N, j / N, 0))))
```

Next, we add the area elements. Between four neighboring points we either span a quadrilateral (if quads==True) or divide the area into two triangle. These are then simply added to the mesh:

```
mesh.SetMaterial(1, "mat")
for j in range(N):
    for i in range(N):
        if quads:
            mesh.Add(Element2D(1, [pnums[i + j * (N + 1)], pnums[i + (j + 1) * (N +␣
↪1)], pnums[i + 1 + (j + 1) * (N + 1)], pnums[i + 1 + j * (N + 1)]]))
        else:
            mesh.Add(Element2D(1, [pnums[i + j * (N + 1)], pnums[i + (j + 1) * (N +␣
↪1)], pnums[i + 1 + j * (N + 1)]]))
            mesh.Add(Element2D(1, [pnums[i + (j + 1) * (N + 1)], pnums[i + 1 + (j +␣
↪1) * (N + 1)], pnums[i + 1 + j * (N + 1)]]))
```

Now, we have to add boundary elements and boundary conditions. Therefore we add a FaceDescriptor:

```
mesh.Add (FaceDescriptor(surfnr=1,domin=1,bc=1))
```

followed by the horizontal boundary elements

```
for i in range(N):
   mesh.Add(Element1D([pnums[N + i * (N + 1)], pnums[N + (i + 1) * (N + 1)]],␣
↪index=1))
   mesh.Add(Element1D([pnums[0 + i * (N + 1)], pnums[0 + (i + 1) * (N + 1)]],␣
↪index=1))
```

and the vertical boundary elements:

```
for i in range(N):
   mesh.Add(Element1D([pnums[i], pnums[i + 1]], index=1))
   mesh.Add(Element1D([pnums[i + N * (N + 1)], pnums[i + 1 + N * (N + 1)]], index=1))
```

This, together results in a valid mesh. Note that we have chosen boundary condition bc=1 on all boundaries.