pinvit

November 16, 2022

1 2.2 Programming an eigenvalue solver

We solve the generalized eigenvalue problem

$$Au = \lambda Mu,$$

where A comes from $\int \nabla u \nabla v$, and M from $\int uv$, on the space H_0^1.

This tutorial shows how to implement linear algebra algorithms.

[ ]:
```
from ngsolve import *
from ngsolve.webgui import Draw
from netgen.geom2d import unit_square
import math
import scipy.linalg
from scipy import random

mesh = Mesh(unit_square.GenerateMesh(maxh=0.1))
```

We setup a stiffness matrix A and a mass matrix M, and declare a preconditioner for A:

[ ]:
```
fes = H1(mesh, order=4, dirichlet=".*")
u = fes.TrialFunction()
v = fes.TestFunction()

a = BilinearForm(fes)
a += grad(u)*grad(v)*dx
pre = Preconditioner(a, "multigrid")

m = BilinearForm(fes)
m += u*v*dx

a.Assemble()
m.Assemble()

u = GridFunction(fes)
```

The inverse iteration is

$$u_{n+1} = A^{-1}\ Mu_n,$$

where the Rayleigh quotient

$$\rho_n = \frac{\langle Au_n, u_n\rangle}{\langle Mu_n, u_n\rangle}$$

converges to the smallest eigenvalue λ_1, with rate of convergence λ_1/λ_2, where λ_2 is the next smallest eigenvalue.

The preconditioned inverse iteration (PINVIT), see [Knyazef+Neymeyr], replaces A^{-1} by an approximate inverse C^{-1}:

$$\rho_n = \frac{\langle Au_n, u_n\rangle}{\langle Mu_n, u_n\rangle} w_n = C^{-1}(Au_n - \rho_n Mu_n) u_{n+1} \ = u_n + \alpha w_n$$

The optimal step-size α is found by minimizing the Rayleigh-quotient on a two-dimensional space:

$$u_{n+1} = \arg \min_{v \in \{u_n, w_n\}} \frac{\langle Av, v\rangle}{\langle Mv, v\rangle}$$

This minimization problem can be solved by a small eigenvalue problem

$$ay = \lambda my$$

with matrices

$$a = \begin{pmatrix} \langle Au_n, u_n\rangle & \langle Au_n, w_n\rangle \\ \langle Aw_n, u_n\rangle & \langle Aw_n, w_n\rangle \end{pmatrix}, \quad m = \begin{pmatrix} \langle Mu_n, u_n\rangle & \langle Mu_n, w_n\rangle \\ \langle Mw_n, u_n\rangle & \langle Mw_n, w_n\rangle \end{pmatrix}.$$

Then, the new iterate is

$$u_{n+1} = y_1 u_n + y_2 w_n$$

where y is the eigenvector corresponding to the smaller eigenvalue.

1.0.1 Implementation in NGSolve

First, we create some help vectors. `CreateVector` creates new vectors of the same format as the existing vector, i.e., same dimension, same real/ complex type, same entry-size, and same MPI-parallel distribution if any.

[ ]:
```
r = u.vec.CreateVector()
w = u.vec.CreateVector()
Mu = u.vec.CreateVector()
Au = u.vec.CreateVector()
Mw = u.vec.CreateVector()
Aw = u.vec.CreateVector()
```

Next, we pick a random initial vector, which is zeroed on the Dirichlet boundary.

Below, the `FV` method (short for `FlatVector`) lets us access the abstract vector's linear memory chunk, which in turn provides a "numpy" view of the memory. The projector clears the entries at the Dirichlet boundary:

[ ]:
```
r.FV().NumPy()[:] = random.rand(fes.ndof)
u.vec.data = Projector(fes.FreeDofs(), True) * r
```

Finally, we run the PINVIT algorithm. Note that the small matrices a and m defined above are called `asmall` and `msmall` below. They are of type `Matrix`, a class provided by NGSolve for dense matrices.

[ ]:
```
for i in range(20):
    Au.data = a.mat * u.vec
    Mu.data = m.mat * u.vec
    auu = InnerProduct(Au, u.vec)
    muu = InnerProduct(Mu, u.vec)
    # Rayleigh quotient
    lam = auu/muu
    print (lam / (math.pi**2))
    # residual
    r.data = Au - lam * Mu
    w.data = pre.mat * r.data
    w.data = 1/Norm(w) * w
    Aw.data = a.mat * w
    Mw.data = m.mat * w

    # setup and solve 2x2 small eigenvalue problem
    asmall = Matrix(2,2)
    asmall[0,0] = auu
    asmall[0,1] = asmall[1,0] = InnerProduct(Au, w)
    asmall[1,1] = InnerProduct(Aw, w)
    msmall = Matrix(2,2)
    msmall[0,0] = muu
    msmall[0,1] = msmall[1,0] = InnerProduct(Mu, w)
    msmall[1,1] = InnerProduct(Mw, w)
    # print ("asmall =", asmall, ", msmall = ", msmall)


    eval,evec = scipy.linalg.eigh(a=asmall, b=msmall)
    # print (eval, evec)
    u.vec.data = float(evec[0,0]) * u.vec + float(evec[1,0]) * w

Draw (u)
len(u.vec.data)
```

1.1 Simultaneous iteration for several eigenvalues

Here are the steps for extending the above to `num` vectors.

Declare a `GridFunction` with multiple components to store several eigenvectors:

[ ]:
```
num = 5
u = GridFunction(fes, multidim=num)
```

Create list of help vectors, and a set of random initial vectors in `u`, with zero boundary conditions:

[ ]:
```
r = u.vec.CreateVector()
Av = u.vec.CreateVector()
Mv = u.vec.CreateVector()

vecs = []
for i in range(2*num):
    vecs.append (u.vec.CreateVector())

for v in u.vecs:
    r.FV().NumPy()[:] = random.rand(fes.ndof)
    v.data = Projector(fes.FreeDofs(), True) * r
```

Compute `num` residuals, and solve a small eigenvalue problem on a $2 \times$ `num` dimensional space:

[ ]:
```
asmall = Matrix(2*num, 2*num)
msmall = Matrix(2*num, 2*num)
lams = num * [1]

for i in range(20):

    for j in range(num):
        vecs[j].data = u.vecs[j]
        r.data = a.mat * vecs[j] - lams[j] * m.mat * vecs[j]
        vecs[num+j].data = pre.mat * r

    for j in range(2*num):
        Av.data = a.mat * vecs[j]
        Mv.data = m.mat * vecs[j]
        for k in range(2*num):
            asmall[j,k] = InnerProduct(Av, vecs[k])
            msmall[j,k] = InnerProduct(Mv, vecs[k])

    ev,evec = scipy.linalg.eigh(a=asmall, b=msmall)
    lams[:] = ev[0:num]
    print (i, ":", [lam/math.pi**2 for lam in lams])

    for j in range(num):
        u.vecs[j][:] = 0.0
```

```
        for k in range(2*num):
            u.vecs[j].data += float(evec[k,j]) * vecs[k]

Draw (u)
```

The *multidim-component* select in the *Visualization* dialog box allows to switch between the components of the multidim-GridFunction.

1.2 Implementation using MultiVector

The simultaneous iteration can be optimized by using `MultiVectors` introduced in NGSolve 6.2.2007. These are arrays of vectors of the same format. You can think of a MultiVector with m components of vector size n as an $n \times m$ matrix.

- a `MultiVector` consisting of num vectors of the same format as an existing vector vec is created via MultiVector(vec, num).
- we can iterate over the components of a MultiVector, and the bracket operator allows to access a subset of vectors
- linear operator application is optimized for MultiVector
- vector operations are optimized and called as mv * densematrix: $x = y * mat$ results in x[i] = sum_j y[j] * mat[j,i] (where x and y are 'MultiVector's, and mat is a dense matrix)
- pair-wise inner products of two MultiVectors is available, the result is a dense matrix: InnerProduct(x,y)[i,j] =
- mv.Orthogonalize() uses modified Gram-Schmidt to orthogonalize the vectors. Optionally, a matrix defining the inner product can be provided.
- with mv.Append(vec) we can add another vector to the array of vectos. A new vector is created, and the values are copied
- mv.AppendOrthogonalize(vec) appends a new vector, and orthogonalizes it against the existing vectors, which are assumed to be orthogonal.

[ ]:
```
uvecs = MultiVector(u.vec, num)
vecs = MultiVector(u.vec, 2*num)

for v in vecs[0:num]:
    v.SetRandom()
uvecs[:] = pre * vecs[0:num]
lams = Vector(num * [1])
```

[ ]:
```
for i in range(20):
    vecs[0:num] = a.mat * uvecs - (m.mat * uvecs).Scale (lams)
    vecs[num:2*num] = pre * vecs[0:num]
    vecs[0:num] = uvecs

    vecs.Orthogonalize() # m.mat)

    asmall = InnerProduct (vecs, a.mat * vecs)
    msmall = InnerProduct (vecs, m.mat * vecs)
```

```python
    ev,evec = scipy.linalg.eigh(a=asmall, b=msmall)
    lams = Vector(ev[0:num])
    print (i, ":", [l/math.pi**2 for l in lams])

    uvecs[:] = vecs * Matrix(evec[:,0:num])
```

The operations are implemented using late evaluation. The operations return expressions, and computation is done within the assignment operator. The advantage is to avoid dynamic allocation. An exception is InnerProduct, which allows an expression in the second argument (and then needs vector allocation in every call).

[ ]:

```python

```

[ ]:

```python

```