Array operations: questions

```cpp
real[int] a(5), b(5), c(5), d(5);
a = 1;
b = 2;
c = 3;
a[2] = 0;
d = (a ? b : c);
cout << "d=" << d << endl;
cout << "== 2 2 3 2 2" << endl;
d = (a ? 1 : 10);
cout << "(a ? 1 : 10) " << d << endl;
d = (a ? b : -1);
cout << "(a ? b : -1) " << d << endl;
d = (a ? -2 : c);
cout << "(a ? -2 : c) " << d << endl;
```

Output:
```
d=5
2 2 3 2 2
== 2 2 3 2 2
(a ? 1 : 10) 5
1 1 10 1 1
(a ? b : -1) 5
2 2 -1 2 2
(a ? -2 : c) 5
-2 -2 3 -2 -2
```

Matlab-like colon notation

- The book says colon notation works for subscripts “like Matlab”
- The following code:

```cpp
int N=5;
real[int] a(N), b(N), c(N);
a = [1, -1, 2, 2, 5];
cout << "a = " << a << endl;
```

- Produces
  ```
a = 5
0:5:2 = 2;
cout << "a = " << a << endl;
```
  ```
  2 1 2 1 2
  ```

- `a(0:4:2) = 2;` produces `2 1 2 1 2`
- `a(0:4:1) = 2;` produces `2 2 2 2 2`

**Stay away from this syntax!**

1D array operations

```cpp
int N=5;
real[int] a(N), b(N), c(N);
a = [1, -1, 2, 2, 5];
cout << "a = " << a << endl;
```

- The following code:

```cpp
b = a + a;
cout << "b = a + a : " << b << endl;
b += a;
cout << "b += a : " << b << endl;
b += 2*a;
cout << "b += 2*a : " << b << endl;
b /= 2;
cout << "b /= 2 : " << b << endl;
b += a;
cout << "b += a; b = " << b << endl; // same b = b .* a
b ./= a;
cout << "b ./= a; b = " << b << endl; // same b = b ./ a
c = a + b;
cout << "c = a + b : c= " << c << endl;
c = 2*a + 4*b;
cout << "c = 2*a + 4*b : c= " << c << endl;
c = a .* b;
cout << "c = a .* b : c= " << c << endl;
c = a ./ b;
cout << "c = a ./ b : c= " << c << endl;
c = 2 * b;
cout << "c * 2 : c= " << c << endl;
```
Array methods

```cpp
int N=5;
real[int] a(N);
a = [1, -1, 2, 2, 5];
cout << " a = " << a << endl;
```

```cpp
    cout << " |a|_1 = " << a.l1 << endl;
cout << " |a|_2 = " << a.l2 << endl;
cout << " |a|_infty = " << a.linfty << endl;
cout << " sum a_i = " << a.sum << endl;
cout << " max a_i = " << a.max << " a[ " << a.imax << " ] = " << a[a.imax] << endl;
cout << " min a_i = " << a.min << " a[ " << a.imin << " ] = " << a[a.imin] << endl;
cout << " a'*a = " << (a' * a) << endl;
cout << " a.quantile(0.2) = " << a.quantile(0.2) << endl;
cout << " a.sort = " << a.sort << endl; // changes a !
```

Results in

```cpp
a = 5
1 -1 2 2 5
```

```cpp
||a||_1 = 11
||a||_2 = 5.91608
||a||_infty = 5
sum a_i = 9
max a_i = 5 a[ 4 ] = 5
min a_i = -1 a[ 1 ] = -1
a’a = 35
a.quantile(0.2) = 1
```

2D Arrays

```cpp
int N=3,M=4;
real[int,int] A(N,M);
real[int] b(N), c(M);
b = [1,2,3];
c = [4,5,6,7];
complex[int,int] C(N,M);
complex[int] cb=[1,2,3], cc=[10i,20i,30i,40i];
```

```cpp
int [int] I = [2,0,1];
int [int] J = [2,0,1,3];
A = 1; // set the full matrix
cout << " I. A = " << A << endl;
A(2,:) = 4; // the full row 2
cout << " II. A = " << A << endl;
A(:,1) = 5; // the full column 1
cout << " III. A = " << A << endl;
A(0:N-1, 2) = 2; // set the full column 2
cout << " IV. A = " << A << endl;
A(1, 0:2) = 3; // set row 1 from 0 to 2
cout << " A = " << A << endl;
```

Array mapping

```cpp
int N=5;
real[int] a(N), b(N), c(N);
int[int] I=[2,3,4,-1,3];
a = [1, -1, 2, 2, 5];
cout << " a = " << a << endl;
```

```cpp
    cout << " I = " << I << endl;
```

```cpp
b = a = -3;
b = a(I ); // for( i=0; i<b.n; i++) if( I[i] >=0 ) b[ i]=a[ I[i] ];
c( I ) = a; // for( i=0; i<b.n; i++) if( I[i] >=0 ) c[ I[i] ]=a[ i ];
cout << " b = a(I ) : " << b << " c( I ) = a " << c << endl;
c( I ) += a; // for( i=0; i<b.n; i++) if( I[i] >=0 ) C( I[i] )+=a[ i ];
cout << " b = a(I ) : " << b << " c( I ) = a " << c << endl;
```

```cpp
.................. RESULTS .........
```

```cpp
a = 5
1 -1 2 2 5
```

```cpp
I = 5
2 3 4 -1 3
```

```cpp
b = a(I ): 5
2
```

```cpp
c(I ) = a 5
2
```

```cpp
b = a(I ): 5
2
```

```cpp
c(I ) = a 5
2
```

```cpp
b = a(I ): 5
2
```

```cpp
c(I ) = a 5
16 16 17 20 18
```

2D Arrays

```cpp
int N=3,M=4;
real[int,int] A(N,M);
real[int] b(N), c(M);
b = [1,2,3];
c = [4,5,6,7];
complex[int,int] C(N,M);
complex[int] cb=[1,2,3], cc=[10i,20i,30i,40i];
```

```cpp
int [int] I = [2,0,1];
int [int] J = [2,0,1,3];
A = 1; // set the full matrix
cout << " I. A = " << A << endl;
A(2,:) = 4; // the full row 2
cout << " II. A = " << A << endl;
A(:,1) = 5; // the full column 1
cout << " III. A = " << A << endl;
A(0:N-1, 2) = 2; // set the full column 2
cout << " IV. A = " << A << endl;
A(1, 0:2) = 3; // set row 1 from 0 to 2
cout << " A = " << A << endl;
```
### Output from previous code

I. \( A = \begin{bmatrix} 3 & 4 \\ 1 & 1 \\ 1 & 1 \\ 1 & 1 \end{bmatrix} \)

II. \( A = \begin{bmatrix} 3 & 4 \\ 1 & 1 \\ 1 & 1 \\ 4 & 4 \end{bmatrix} \)

III. \( A = \begin{bmatrix} 3 & 4 \\ 1 & 5 \\ 1 & 1 \\ 4 & 5 \end{bmatrix} \)

IV. \( A = \begin{bmatrix} 3 & 4 \\ 1 & 5 \\ 2 & 1 \\ 4 & 5 \end{bmatrix} \)

\[
A = \begin{bmatrix} 3 & 4 \\ 1 & 5 \\ 2 & 1 \\ 3 & 3 \\ 4 & 5 \end{bmatrix}
\]

---

### Outer products

```cpp
int N=3,M=4;

complex[int,int] C(N,M);
complex[int] cb=[1,2,3], cc=[10i,20i,30i,40i];

C = cb*cc';
C += 3*cb*cc';
C -= 5i *cb*cc';
cout << " C = " << C << endl;
```

**Results**

\[
C = \begin{bmatrix} -50 & -40 & -100 & -80 \\ -150 & -120 & -200 & -160 \\ -300 & -240 & -400 & -320 \\ -450 & -360 & -600 & -480 \end{bmatrix}
\]

---

### Sparse matrices

```cpp
int N=3,M=4;

real[int,int] A(N,M);
real[int] b(N), c(M);

b = [1,2,3];
c = [4,5,6,7];

A = 2.*b*c'; // outer product
```

```cpp
// the way to transform a array to a sparse matrix
matrix B;
B = A;
```

```cpp
B(I,J) = B(i,j) = A(i,j) = b[I(i)] * c[J(j)];
```

**Results**

\[
A = \begin{bmatrix} 8 & 10 & 12 & 14 \\ 16 & 20 & 24 & 28 \\ 24 & 30 & 36 & 42 \end{bmatrix}
\]

```cpp
B = A; // Sparse Matrix (Morse)
```

```cpp
# first line: n m (is symmetric) nbcoef
# after for each nonzero coefficient: i j a_ij where (i,j) \in 1,...,nx1,...,m
3 4 0 12
1 8
1 10
1 1 3 1 2 14
2 1 16
2 2 20
2 3 24
2 4 28
3 1 24
3 2 30
3 3 36
3 4 42
```

---

```cpp
B = b*c';
```

```cpp
// outer product B[i,j] = b(i)*c(j)
B = B(I,J) = B(i,j) = b[I(i)] * c[J(j)];
```

```cpp
B = (2*b*c')(I,J) ; // outer product B[i,j] = b(I(i))*c(J(j))
```

```cpp
B = (3.*b*c')( I^-1 J^-1 ); // outer product B(I,J) = b(i)*c(j)
```
### Elementary functions

```cpp
int N=3, M=4;
real[int] b(N), c(M);
b = [1, 2, 3];
c = [4, 5, 6, 7];

complex[int] cb=[1, 2, 3], cc=[10i, 20i, 30i, 40i];
cout << " b = " << b << endl;
b = exp(b);
cout << " exp(b) = " << b << endl;
cb += complex(10.)*cc(0:2);
cout << " cb = " << cb << endl;
```

**Results**

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
<td>3</td>
<td>1 2 3</td>
</tr>
<tr>
<td>exp(b)</td>
<td>3</td>
<td>2.718281828 7.389056099 20.08553692</td>
</tr>
<tr>
<td>cb</td>
<td>3</td>
<td>(1, 100) (2, 200) (3, 300)</td>
</tr>
<tr>
<td>exp(cb)</td>
<td>3</td>
<td>(2.3440257, -1.3764445) (3.5998571, -6.452843) (-0.44382246, -20.080633)</td>
</tr>
<tr>
<td>exp(cb).im</td>
<td>3</td>
<td>-1.376444521 -6.45284272 -20.08063284</td>
</tr>
</tbody>
</table>

### Example 23, sound propagation

- **Sound in air satisfies**
  \[
  \frac{\partial^2 u}{\partial t^2} - c^2 \Delta u = 0
  \]

- **If monochromatic boundary conditions** \(g(x)e^{ikt}\)

- **Then solution** \(u(x, t) = v(x)e^{ikt}\)

- \(v\) is a solution of Helmholz’s equation
  \[
  k^2 v + c^2 \Delta v = 0 \quad \text{in} \quad \Omega
  \]

- **Eigenvalues of Laplacian are negative!**
- **Real eigenvalues of Helmholtz \(\Rightarrow\) possible singularity!**

```cpp
example23.edp
```

**Verbosity=10; // increase informational printing**

```cpp
real konc2; // (k/c)^2
if (true)
  konc2 = 1.0; // good solution
else
  konc2 = 14.6993; // eigenvalue! => bad solution
```

**func g = y*(1-y); // sound source**

```cpp
fespace Vh(Th,P1);
Vh u,v;
solve sound(u,v)=int2d(Th)(u*v * konc2 - dx(u)*dx(v) - dy(u)*dy(v))
  - int1d(Th,a4)(g*v);
plot(u, wait=1);
```

**Sound pressure results**

Maximum pressure = 0.18
example23.edp Eigenpairs code

Vh u1,u2;
real sigma = 1; // value of the shift
// OP = A - sigma B ; // the shifted matrix
varf op(u1,u2)= int2d(Th)( dx(u1)*dx(u2) + dy(u1)*dy(u2) - sigma* u1*u2 );
varf b([u1],[u2]) = int2d(Th)( u1*u2 );
matrix OP = op(Vh, Vh, solver=UMFPACK);
matrix B = b(Vh, Vh, solver=UMFPACK);

int nev=20; // number of computed eigen value close to sigma
real[int] ev(nev); // to store the nev eigenvalue
Vh[int] eV(nev); // to store the nev eigenVector
int k=EigenValue(OP, B, sym=true, sigma=sigma, value=ev, vector=eV, tol=1e-10, maxit=0, ncv=0);
assert(k == nev); // k is number of eigenvalues found
for (int n=0; n<nev; n++){
  cout << "Eigenvalue " << n << " = " << ev[n] << endl;
}
for (int n=0; n<nev; n++){
  plot(eV[n],wait=1);
}

times: compile 0.02s, execution 1.14s, mpirank:0

example23.edp eigenvalues
Real symmetric eigenvalue problem: A*x - B*x*lambda
Eigenvalue 0 = -2.22045e-15
Eigenvalue 1 = 0.395913
Eigenvalue 2 = 1.39024
Eigenvalue 3 = 2.2087
Eigenvalue 4 = 2.97306
Eigenvalue 5 = 4.37777
Eigenvalue 6 = 5.13117
Eigenvalue 7 = 7.05942
Eigenvalue 8 = 7.94178
Eigenvalue 9 = 8.61535
Eigenvalue 10 = 10.2941
Eigenvalue 11 = 10.7342
Eigenvalue 12 = 11.421
Eigenvalue 13 = 12.4144
Eigenvalue 14 = 14.6993
Eigenvalue 15 = 14.9511
Eigenvalue 16 = 16.7588
Eigenvalue 17 = 18.5701
Eigenvalue 18 = 19.4472
Eigenvalue 19 = 20.0441
times: compile 0.02s, execution 1.14s, mpirank:0

example23.edp fourteenth eigenvector
Fourteenth eigenvalue = 14.6993

example23.edp Solution with konc2 = 14.6993
Maximum pressure \approx 10^5
WARNING about Dirichlet b.c.
▶ Dirichlet eigenvalues imposed with large diagonal entry!
▶ Other matrix entries remain nonzero
▶ Put on \(a4, u1=0\) on both OP and B, spurious nonzero eigenvalues
▶ Put on \(a4, u1=0\) on only OP spurious nonzero eigenvalues become 0.

Exercise 25 (10 points)
The eigenvalues and eigenvectors for Laplace’s equation on the unit square are well known. Supposing Dirichlet boundary conditions on the left and bottom sides of the square, and Neumann conditions on the right and top of the square, the eigenpairs are, for positive integers \(m\) and \(n\),

\[
\lambda_{m,n} = -\frac{\pi^2}{4}(m^2 + n^2), \\
u_{m,n} = \sin\left(\frac{m\pi x}{2}\right)\sin\left(\frac{n\pi y}{2}\right).
\]

1. There is additional discussion of the eigenproblem in Section 9.4 of the FreeFem++ book.
2. Approximately confirm the above eigenvalue expression for \(m, n = 1, 2\) by computing the eigenvalues using FreeFem++.
3. How many mesh points on each side of the square did you use to get the accuracy you achieved. What accuracy do you observe with twice as many mesh points on each side of the square?
4. Please send me plots of the four eigenvectors you found.
5. Be sure to send me your .edp files.

Example 24, transient convective cooling
▶ Cooling of a bimetal plate in 2D

\[
\frac{\partial u}{\partial t} - \nabla \cdot \kappa \nabla u = 0 \text{ in } \Omega \times (0,T) \\
u(x,y,0) = u_0 + \frac{x}{L}u_1 \\
\kappa \frac{\partial u}{\partial n} + \alpha (u - u_e) = 0 \text{ on } \Gamma \times (0,T)
\]
▶ Euler implicit time discretization

\[
\int_{\Omega} \left( \frac{u^n - u^{n-1}}{\Delta t} w + \kappa \nabla u^n \cdot \nabla w \right) dx + \int_{\Gamma} \alpha (u^n - u_e)w ds = 0
\]
▶ \(\kappa = 0.2\) on the lower half, \(2.0\) on the upper half
Example 24, nonlinear boundary condition

▶ Change boundary conditions for radiation

\[
\kappa \frac{\partial u}{\partial n} + \alpha(u - u_e) + c((u + 273)^4 - (u_e + 273)^4) = 0
\]

▶ Solution by Picard iteration (successive substitution) after employing the identity \(a^4 - b^4 = (a - b)(a + b)(a^2 + b^2)\)

\[
\kappa \frac{\partial u^m}{\partial n} + \alpha(u^m - u_e) + c(u^m - u_e)(u^{m-1} + u_e + 546)
\]

\[
((u^{m-1} + 273)^2 + (u_e + 273)^2) = 0
\]
Flow around an airfoil

- "Infinity" is a circle, C, of radius 5
- Airfoil, S, is NACA0012, given by

\[ y = 0.17735 \sqrt{x} - 0.075597x - 0.212836x^2 + 0.17363x^3 - 0.06254x^4 \]

- Equations

\[ \Delta \psi = 0 \text{ in } \Omega, \quad \psi|_S = 0, \quad \psi|_C = u_\infty y, \]

and \( \partial \Omega = C \cup S. \)

Example 26, Heat convection around an airfoil

- Potential flow (incompressible, irrotational flow)
- \( \nabla \cdot u = 0 \text{ and } \nabla \times u = 0 \)
- So there is \( \psi \) satisfying \( \Delta \psi = 0 \)
- And \( u = (u_x, u_y) \) where \( u_x = \partial \psi / \partial y \) and \( u_y = -\partial \psi / \partial x. \)
- \( \psi \) = constant at walls since \( \partial u / \partial n = 0 \)
- Prescribed velocity at a boundary translates into nonconstant Dirichlet data for \( \psi \)

example26.edp code

```plaintext
real S=99;
border C(t=0,2*pi) { x=3*cos(t); y=3*sin(t);}

border Splus(t=0,1){ x = t; y = 0.17735 *sqrt(t) - 0.075597*t - 0.212836*(t^2) + 0.17363*(t^3) - 0.06254*(t^4); label=S;}

border Sminus(t=1,0){ x =t; y= -(0.17735 *sqrt(t) - 0.075597*t + 0.212836*(t^2) - 0.17363*(t^3) + 0.06254*(t^4)); label=S;}

mesh Th= buildmesh(C(50)+Splus(70)+Sminus(70));

fespace Vh(Th,P2);
Vh psi,w;

solve potential(psi, w) = int2d(Th)(dx(psi)*dx(w) + dy(psi)*dy(w)) + on(C, psi=y) + on(S, psi=0);

plot(psi, wait=true);
plot(Th, wait=true);
```

Example 26 flow results

```
border D(t=0.2){x=1+t; y=0;}
mesh Sh = buildmesh(C(25) + Splus(-90) + Sminus(-90) + D(200));

fespace Wh(Sh, P1);
Wh v,vv;

int steel = Sh(0.5,0).region, air = Sh(-1,0).region;

fespace W0(Sh,P0);
W0 k = 0.01*(region==air) + 0.1*(region==steel);
W0 u1 = dy(psi)*(region==air), u2 = -dx(psi)*(region==air);
Wh vold = 120*(region==steel);

real dt=0.05, nbT=50;
bool factoredMatrix = false;
problem thermic(v, vv, init=factoredMatrix, solver=LU)=
  int2d(Sh)( v *vv/dt + k*( dx(v)*dx(vv) + dy(v)*dy(vv) )
         + 10*(u1*dx(v) + u2*dy(v))*vv )
  - int2d(Sh)(vold*vv/dt);
Velocity=10X, no upwinding
for(int i=0;i<nbT;i++){
  v = vold;
  thermic;
  factoredMatrix = true;
}
plot(v,wait=1);
plot(Th,wait=1);
plot(Sh,wait=1);
```
Upwind differencing?

- Upwind differencing is “required” when convective effects become much larger than diffusive effects.
- Velocity was multiplied by 10 to make convective effects “visible.”

Effect of increasing velocity

- Velocity $\times 1$
- Velocity $\times 10$
- Velocity $\times 100$

Mesh control

- Mesh near “trailing edge” is a problem
  - Singularity in Potential flow solution
- Example 16 (shedding using FEniCS): mesh refinement
- Above, inserted line with many points
- Could insert regions with given number points on boundaries

Rotating hill

- $\Omega$ is unit disk, $\|x\| \leq 1$
- Fixed velocity field $(u_1, u_2) = (y, -x)$
- Pure convection
  \[ \frac{\partial c}{\partial t} + u \cdot \nabla c = 0 \]
- $c(x, 0) = c^0(x)$
- Exact solution $c(x, t) = R(t)c^0(x, 0)$, where $R(t)$ represents a rotation by $\theta = -t$
- example27.edp uses convect.
- $c^0(x)$ is a Gaussian
Example 27 uses discontinuous Galerkin

This is the so-called dual-$P^{DC}_1$ formulation (Ref. Ern[11])

$$
\int_\Omega \frac{c^n - c^{n-1}}{\Delta t} \, dx + \int_E (\alpha |\mathbf{n} \cdot \mathbf{u}| - 0.5 \mathbf{n} \cdot \mathbf{c}) w \, ds = \int_{E^-} |\mathbf{n} \cdot \mathbf{u}| c w \, ds
$$

- $E$ is the set of inner edges
- $E^-$ is the set of boundary edges with $\mathbf{n} \cdot \mathbf{u} < 0$ (none here)
- $[c]$ is jump $(c^+ - c^-)$ where $+$ refers to the right of the oriented edge

// example28.edp, from file chapt3/convects.edp
// With Discontinuous Galerkin
border C(t=0, 2*pi) { x=cos(t); y=sin(t); }
mesh Th = buildmesh(C(100));

fespace Vh(Th,P1dc);
Vh w, ccold, v1 = y, v2 = -x, cc = exp(-10*( (x-0.3)^2 + (y-0.3)^2) );
real u, alpha=0.5, dt = 0.05;
macro n() (N.x*v1 + N.y*v2) //

problem ADual(cc, w) = int2d(Th)( (cc/dt + (v1*dx(cc) + v2*dy(cc)) )*w )
  + intalledges(Th)( (1-nTonEdge)*w*( alpha*abs(n) - n/2 )*jump(cc) ) // - int1d(Th,C)( (n(u)<0)*abs(n(u) )*cc*w ) // unused: cc=0 on boundary
  - int2d(Th)( ccold*w/dt ) ;

real [int] viso=[-0.1, 0, 0.5, 0.1, 0.5, 0.2, 0.25, 0.3, 0.35, 0.4, 0.45, 0.5, 0.55, 0.6, 0.65, 0.7, 0.75, 0.8, 0.9, 1];

for (real t=0; t< 2*pi ; t+=dt){
  ccold=cc;
  ADual;
  plot(cc, fill=true, dim=3, viso=viso,
    cmm="example28, min=" + cc[].min + ", max=" + cc[].max);
}

plot(cc,wait=true,fill=false,value=true,viso=viso,
    cmm="example28, min=" + cc[].min + ", max=" + cc[].max);

---

// example28.edp code
// example28.edp output

// example27.edp code
// example27.edp output
Some details of new functions

The integral over *interior* edges is given by

$$\int_{E} (\alpha|n \cdot u| - 0.5n \cdot u[c]w \, ds$$

And is implemented as

```cpp
intalledges(Th) ( (1-nTonEdge)*w*( al*abs(n) - n/2 )*jump(cc))
```

- `intalledges` is an integral over all edges, including boundary edges.
- `nTonEdge` is the number of triangles meeting at this edge. In a 2d geometry, this can only be 1 or 2.
- `(1-nTonEdge)` is (-1) for interior edges, 0 for boundary edges.
- `jump(cc)` computes $cc^- - cc^+$
- The explanation on p. 35 is slightly misleading.

Comparison of 27 and 28

27 declined to 0.64, 28 declined to .70
**Example 29: faster**

```plaintext
varf aadual(cc, w) = int2d(Th)( (cc/dt + (v1*dx(cc) + v2*dy(cc)) )*w )
    + intalledges(Th)( (1-nTonEdge)*w*( alpha*abs(n) - n/2 )*jump(cc) );
varf bbdual(ccold, w) = - int2d(Th)( ccold*w/dt);

bool reuseMatrix = false;
matrix AA = aadual(Vh,Vh);
matrix BB = bbdual(Vh,Vh);
set (AA, init=reuseMatrix, solver=UMFPACK);
Vh rhs=0;
for (real t=0; t< 2*pi ; t+=dt) {
    ccold = cc;
    rhs[] = BB * ccold[];
    cc[] = AA^-1 * rhs[];
    reuseMatrix = true;
    plot(cc, fill=true, dim=3, viso=viso,
         cmm="t="+t + " , min=" + cc[].min + " , max=" + cc[].max);
}
```

**Exercise 26 (10 points)**

Consider the three forms of the rotating hill problem, Examples 27, 28, and 29.

1. Run each to a limit of $16\pi$ instead of $2\pi$. Send me a copy of the final plot from each case. What would you conclude from your experiment, if anything?

2. Return to Example 29 and revise it so that it uses ordinary P1 elements and the usual finite element discretization of convection. Run it twice, once until a limit of $2\pi$, and once to a limit of $16\pi$. What would you conclude from this experiment?

---

**Macros**

- Macros can be defined either with parameters:
  ```plaintext
  macro <identifier>(<parameter list>) <replacement token list> // EOM
  ```
- Or without parameters:
  ```plaintext
  macro <identifier>() <replacement token list> // EOM
  ```
- Macros **must** end with the comment characters `//`.

**Examples of macros**

```plaintext
macro n() (N.x*v1 + N.y*v2) // EOM
macro div(u,v) (dx(u) + dy(v)) // EOM
macro epsilon(u1,u2) [ dx(u1), dy(u2), (dy(u1)+dx(u2)) ] // EOM
macro Grad(u1,u2) [ dx(u1), dy(u1), dx(u2), dy(u2) ] // EOM
macro UgradV(u1,u2,v1,v2) [ [u1,u2]'*[dx(v1),dy(v1)] , [u1,u2]'*[dx(v2),dy(v2)] ] // EOM
```
The equations governing elastic (small) deformation can be written

\[-\nabla \cdot \sigma = t \text{ in } \Omega\]

\[\sigma = \lambda (\nabla \cdot u) I + 2\mu \epsilon\]

\[\epsilon = \frac{1}{2} (\nabla u + \nabla u^T)\]

\[\epsilon_{ij} = \frac{1}{2} \left( \frac{\partial u_i}{\partial x_j} + \frac{\partial u_j}{\partial x_i} \right)\]

where \(\sigma\) is a stress tensor and \(\epsilon\) is a deformation tensor. The deformation \(u\) is a vector. In 2D, both \(u\) and \(x\) have 2 components. The variational form becomes

\[\int_{\Omega} \lambda \nabla \cdot u \nabla \cdot v + 2\mu \epsilon(u) : \epsilon(v) - \int_{\Omega} vf = 0\]

Example 31, Driven cavity for Stokes flow

- Extremely slow incompressible flow: Stokes
  
  \[-\Delta u + \nabla p = 0\]
  
  \[\nabla \cdot u = 0\]

- \(u = (u_1, u_2)\) is velocity, \(p\) is a scalar pressure.
- Square with lid moving to the right
- Dirichlet conditions on velocity
- Pressure needs to eliminate the constant null eigenvector
example31.edp code

```c
verbose = 0;
int n = 3;
mesh Th=square(10*n, 10*n);
fespace Uh(Th, P1b);
Uh u,v,uu,vv;
fespace Ph(Th, P1);
Ph p,pp;
solve stokes([u, v, p],[uu, vv, pp]) =
  int2d(Th)(dx(u)*dx(uu) + dy(u)*dy(uu) + dx(v)*dx(vv) + dy(v)*dy(vv)
  + dx(p)*uu + dy(p)*vv + pp*(dx(u)+dy(v))
  -1e-10*p*pp)
  + on(1,2,4, u=0, v=0) + on(3, u=1, v=0);
plot([u,v],p,wait=true);
```

example31.edp results

Projection algorithm for NSE

- NSE (with Dirichlet boundary conditions)
  \[ \frac{\partial u}{\partial t} + u \cdot \nabla u + \nabla p - \nu \Delta u = 0 \]

- Chorin's algorithm
  \begin{align*}
  \frac{\tilde{u} - u^{m-1} \circ X^{m-1}}{\Delta t} + \nabla p^{m-1} - \nu \Delta u^{m-1} & = 0 \\
  -\Delta p^{m} & = -\nabla \cdot u^{m-1} \circ X^{m-1}, \text{ Neumann b.c.}
  \end{align*}

- Rennacher's improvement
  \[ -\Delta q = \nabla \cdot u - \nabla \cdot \tilde{u} \]
  \[ u^{m} = \tilde{u} + \Delta t \nabla q, \quad p^{m} = p^{m-1} - q - p^{m} - q \]

Steady flow over a backward-facing step

- Step in time
- Refine mesh periodically
- Stop when step-to-step change is small


```plaintext
example32.edp code

border a0(t=1,0) { x=0; y=t; label=1; }
border a1(t=0,1) { x=2*t; y=0; label=2; }
border a2(t=0,1) { x=2; y=-t/2; label=2; }
border a3(t=0,1) { x=2+18*t^1.2; y=-0.5; label=3; }
border a4(t=0,1) { x=20; y=-0.5 + 1.5*t; label=3; }
border a5(t=1,0) { x=20*t; y=1; label=4; }

int n=1;
mesh Th = buildmesh(a0(3*n) + a1(20*n) + a2(10*n) + a3(150*n) +
                    a4(5*n) + a5(100*n));
plot(Th);

fespace Vh(Th,P1);
real nu = 0.0025, dt = 0.2; // Reynolds=200
func uBCin = 4*y*(1-y) * (y>0) * (x<2);
func uBCout = 4./1.5 *(y+0.5) * (1-y) * (x>19);
Vh w, u = uBCin, v = 0, p = 0, q = 0;
Vh ubc = uBCin + uBCout;
influx0 = int1d(Th,1) (ubc*N.x),
outflux0 = int1d(Th,3) (ubc*N.x);
real area= int2d(Th)(1.);
bool reuseMatrix = false;

for(int n=0; n<300; n++){
    Vh uold = u, vold = v, pold = p;
    Vh f = convect([uold,vold], -dt, uold);
    real outflux = int1d(Th, 3) (f*N.x);
    f = f - (influx0 + outflux) / outflux0 * uBCout;
    outflux = int1d(Th, 3) (f*N.x);
    assert( abs( influx0 + outflux ) < 1e-10);
    // WARNING the the output flux must be 0 ...

    solve pb4u(u, w, init=reuseMatrix, solver=UMFPACK)
        = int2d(Th)(u*w/dt + nu*(dx(u)*dx(w) + dy(u)*dy(w)))
            - int2d(Th)((convect([uold,vold], -dt, uold)/dt - dx(p))*w)
            + on(1, u = 4*y*(1-y)) + on(2, 4, u = 0) + on(3, u=f);
    plot(u);

    solve pb4v(v, w, init=reuseMatrix, solver=UMFPACK)
        = int2d(Th)(v*w/dt + nu*(dx(v)*dx(w) + dy(v)*dy(w)))
            - int2d(Th)((convect([uold,vold], -dt, vold)/dt - dy(p))*w)
            + on(1, 2, 3, 4, v = 0);
    plot(v);

    real meanpq = int2d(Th)(pold - q)/area;
    if(n%50==49){
        Th = adaptmesh(Th, [u,v], q, err=0.04, nbvx=100000);
        plot(Th, wait=true);
        ubc = uBCin + uBCout; // reinterpolate B.C.
        influx0 = int1d(Th,1) (ubc*N.x),
        outflux0 = int1d(Th,3) (ubc*N.x);
    }
p = pold - q - meanpq;
u = u + dx(q)*dt;
v = v + dy(q)*dt;
    real err = sqrt(int2d(Th)( square(u - uold) + square(v - vold))/Th.area);
    cout << " iter " << n << " Err L2 = " << err << endl;
    if( err < 1e-3 ) break;
}
plot(Th, wait=true);
plot(p, wait=true);
plot(u, wait=true);
```

dummy image