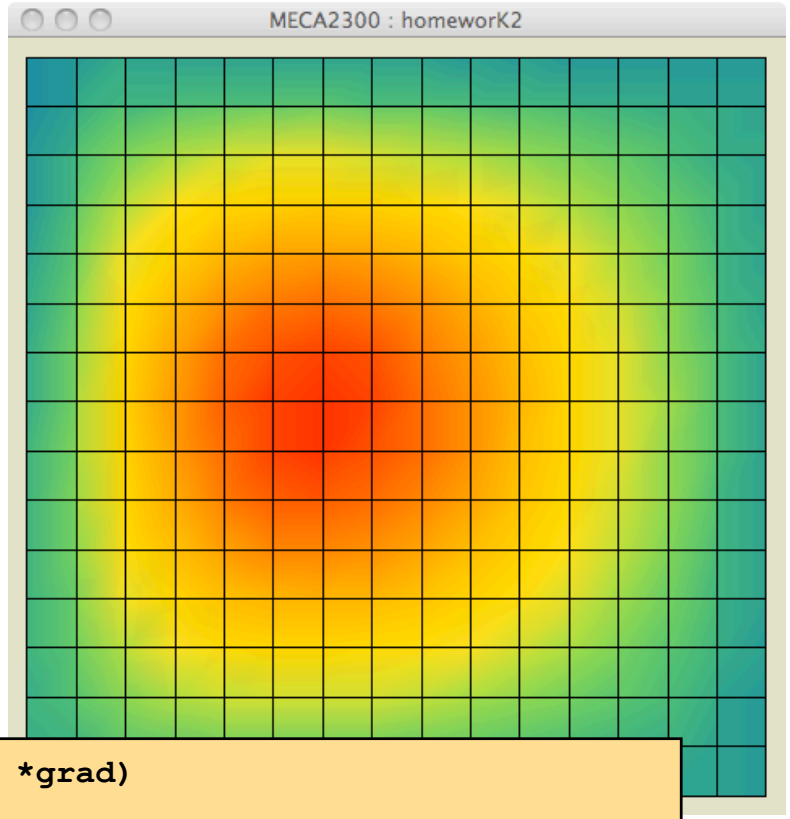
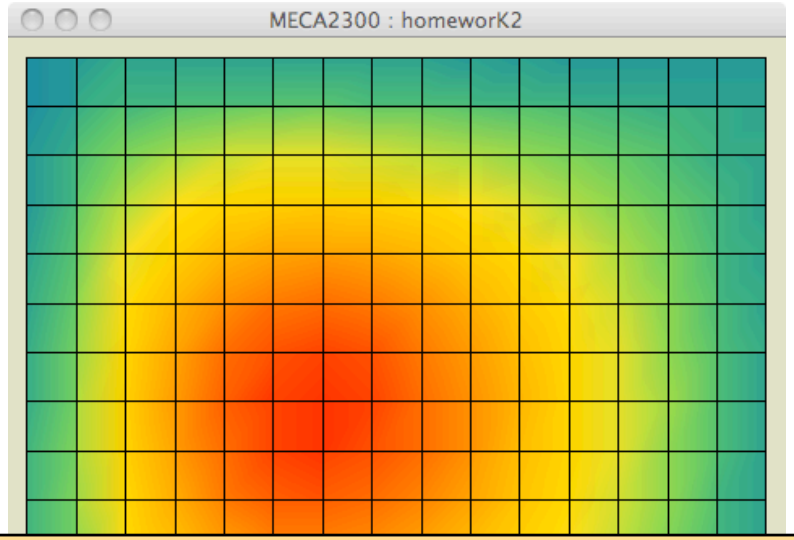
Computing the gradient !




```
void gridGradient(Grid *grid, double *field, double *grad)
{
  int i;
  for (i = 0; i < 2 * grid->fieldSize; ++i) {
    grad[i] = 0.;
  }
  for (i = 0; i < vectorSize(grid->face); ++i) {
    Face *f = &grid->face[i];
    double *gradn = f->orientation == VERTICAL ? grad : grad + grid->fieldSize;
    gradn[f->cell[0]->id] += field[f->cell[1]->id] / (2 * grid->dx);
    gradn[f->cell[1]->id] -= field[f->cell[0]->id] / (2 * grid->dx);
  }
}
```

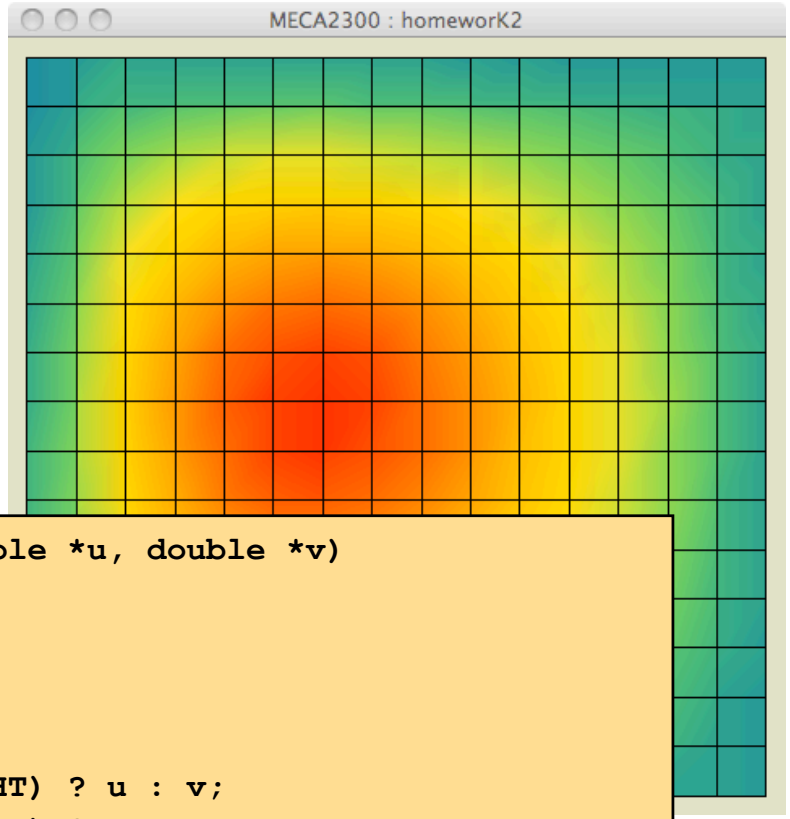
Reconstruction of the solution




```
void gridFaceReconstruct(Grid *grid, Face *f, double *field, double *grad,
                         double *u0, double *u1)
{
  double *gradn = f->orientation == VERTICAL ? grad : grad + grid->fieldSize;
  *u0 = field[f->cell[0]->id] + gradn[f->cell[0]->id] * grid->dx / 2;
  *u1 = field[f->cell[1]->id] - gradn[f->cell[1]->id] * grid->dx / 2;
}
```

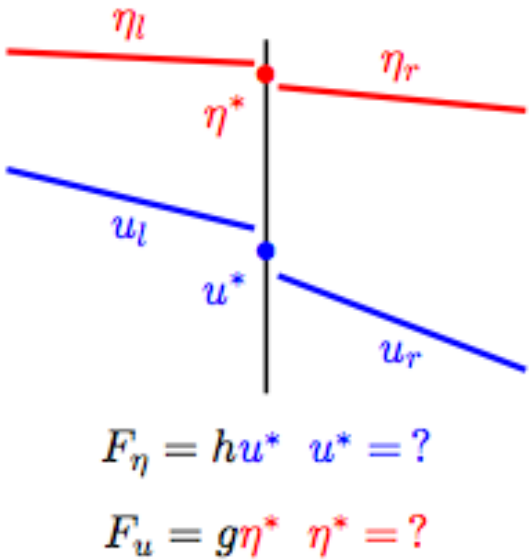
```
void gridCornerReconstruct(Grid *grid, Cell *c, double *u, double *grad, double *uc)
{
  double *gx = grad, *gy = grad + grid->fieldSize;
  double D = grid->dx / 2;
  uc[0] = u[c->id] - gx[c->id] * D - gy[c->id] * D;
  uc[1] = u[c->id] + gx[c->id] * D - gy[c->id] * D;
  uc[2] = u[c->id] + gx[c->id] * D + gy[c->id] * D;
  uc[3] = u[c->id] - gx[c->id] * D + gy[c->id] * D;
}
```

Imposing the boundary conditions




```
void sweBoundaryConditions(Swe *swe, double *h, double *u, double *v)
{
  int i;
  Grid *g = swe->grid;
  for (i = 0; i < vectorSize(g->boundary); ++i) {
    Boundary *b = &g->boundary[i];
    double *un = (b->side == LEFT || b->side == RIGHT) ? u : v;
    double *ut = (b->side == LEFT || b->side == RIGHT) ? v : u;
    h[b->cell[1]->id] = h[b->cell[0]->id];
    un[b->cell[1]->id] = -un[b->cell[0]->id];
    ut[b->cell[1]->id] = ut[b->cell[0]->id];
  }
}
```

The so-called Riemann solver



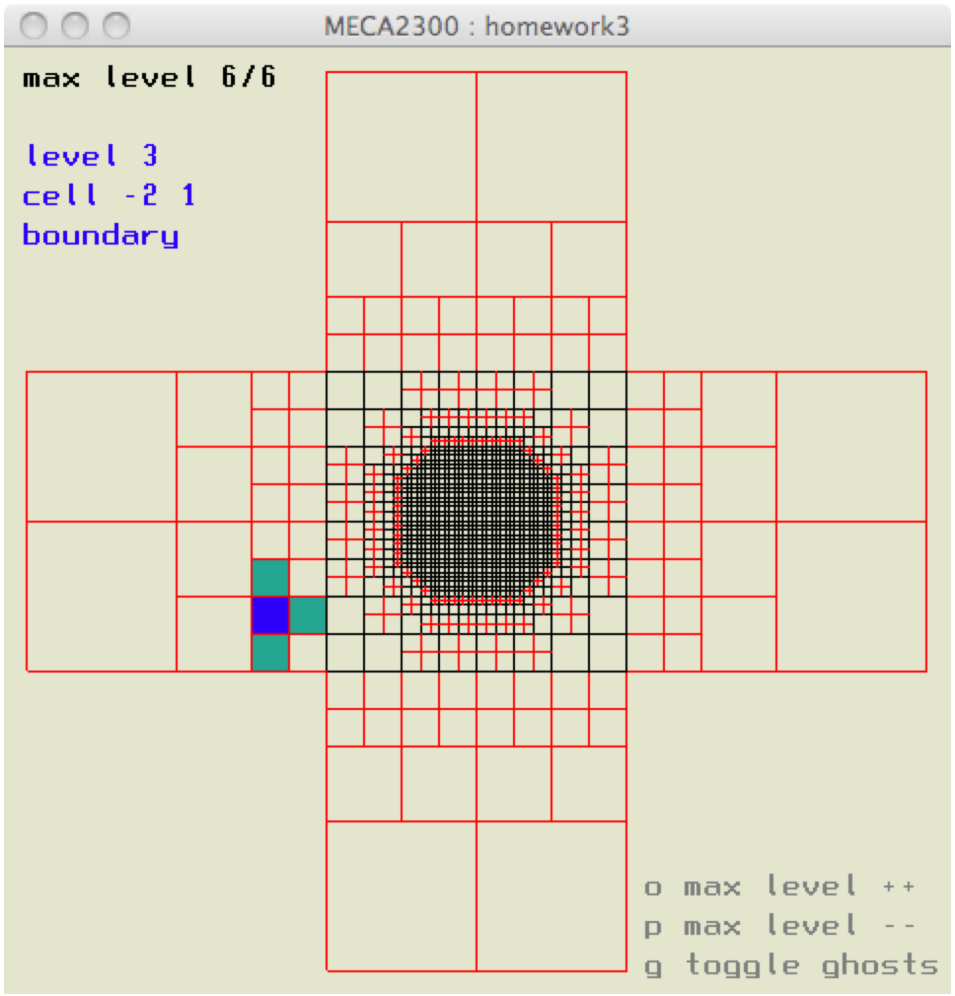

```
void sweRiemann(Swe *swe, double hl, double hr,
       double ul, double ur, double vl, double vr,
       double *fh, double *fu, double *fv)
{

  double hm = (hl + hr) / 2;
  double l = sqrt(swe->g * hm);
  *fh = (ur + ul + l * (hl - hr)) / 2;
  *fu = (swe->g * hr * hr / 2 + swe->g * hl * hl / 2 + l * (ul - ur)) / 2;
  *fv = 0;

  double um = (ur  + ul ) / 2;
  *fu +=  (um < 0 ? um * ur / hr : um * ul / hl);
  *fv +=  (um < 0 ? um * vr / hr: um * vl / hl);
}
```

And the last piece !

```
for (i = 0; i < vectorSize(grid->cell); ++i) {
  Cell *c = &grid->cell[i];
  double x, y;
  gridCellCenter(grid, c, &x, &y);
  double cor = swe->f0 + swe->beta * y;
  double windu = swe->w * sin(M_PI * y / (grid->n * grid->dx));
  double windv = 0;
  FU[c->id] =-swe->gamma * u[c->id] + cor * v[c->id] + windu;
  FV[c->id] =-swe->gamma * v[c->id] - cor * u[c->id] + windv;
  FH[c->id] = 0.;
}
```

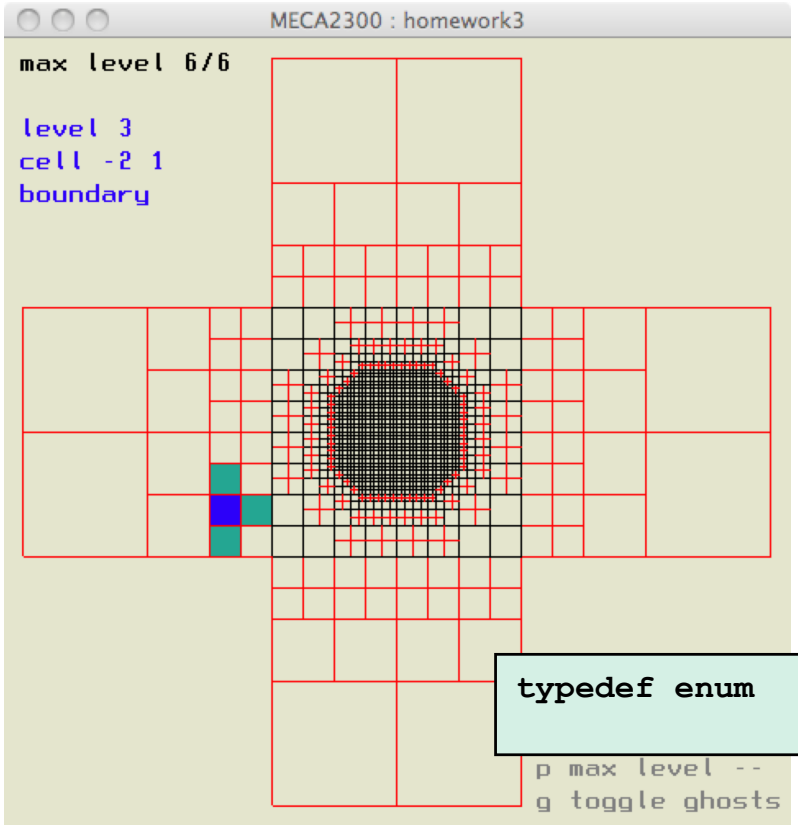
```
for (i = 0; i < vectorSize(grid->face); ++i) {
  Face *f = &grid->face[i];
  double hl, ul, vl, hr, ur, vr, fh, fu, fv;
  gridFaceReconstruct(grid, f, h, gradh, &hl, &hr);
  gridFaceReconstruct(grid, f, u, gradu, &ul, &ur);
  gridFaceReconstruct(grid, f, v, gradv, &vl, &vr);
  if (f->orientation == VERTICAL)
    sweRiemann(swe, hl, hr, ul, ur, vl, vr, &fh, &fu, &fv);
  else
    sweRiemann(swe, hl, hr, vl, vr, ul, ur, &fh, &fv, &fu);
  FH[f->cell[0]->id] -= fh / grid->dx;
  FH[f->cell[1]->id] += fh / grid->dx;
  FU[f->cell[0]->id] -= fu / grid->dx;
  FU[f->cell[1]->id] += fu / grid->dx;
  FV[f->cell[0]->id] -= fv / grid->dx;
  FV[f->cell[1]->id] += fv / grid->dx;
}
```

Homework 3



- **Implementing quadtrees !**
- **Adding ghost cells**
- **Rafining the mesh in a recursive way**

Defining cells, faces and boudaries !



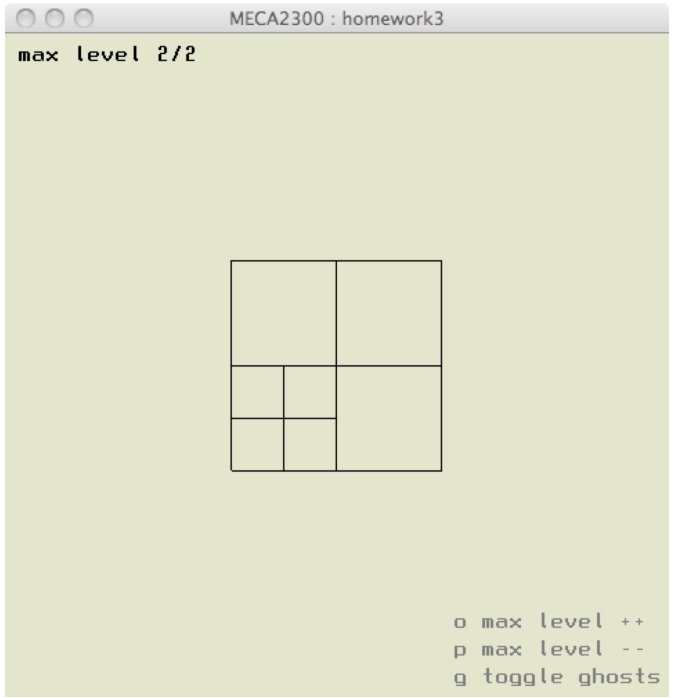

```
typedef struct _Cell{
  int i, j, level;
  struct _Cell *child[4];
  struct _Cell *neighbour[4];
  struct _Cell *parent;
  CellFlag flag;
} Cell;
```

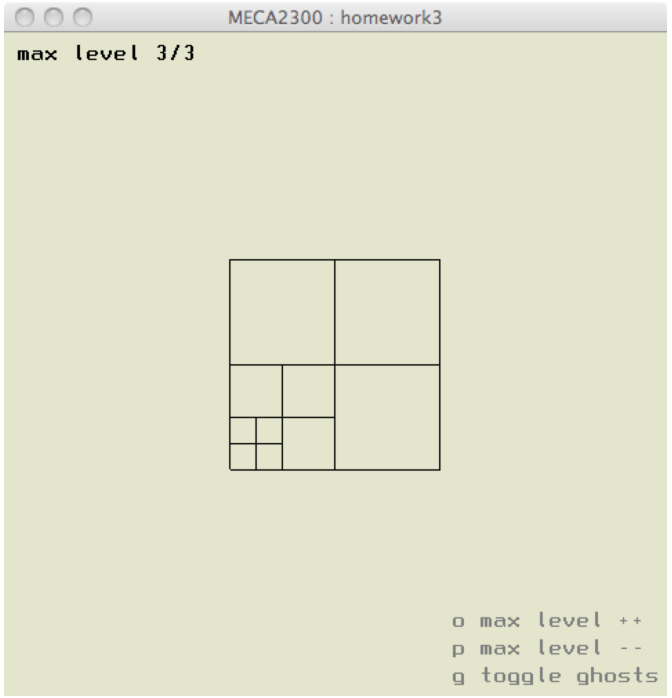
```
typedef enum  {INACTIVE_INTERIOR=0, INACTIVE_BOUNDARY,
               ACTIVE_LEAF, ACTIVE_PARENT} CellFlag;
```

Refinement function

```
int myLevel(double x, double y)
{
    return (hypot(x, y) < 0.3) ? 2 : 1;
}
```

```
int myLevel(double x, double y)
{
    return (hypot(x, y) < 0.3) ? 3 : 1;
}
```





Refinement function

```
int myLevel(double x, double y)
{
    return (hypot(x, y) < 0.3) ? 4 : 1;
}
```

```
int myLevel(double x, double y)
{
    return (hypot(x, y) < 0.3) ? 5 : 1;
}
```