Implementing Runge-Kutta methods

```
Erk *erkNew22()
{
  double a[] = {0, 0, 1, 0};
  double b[] = {0.5, 0.5};
  double c[] = {0, 1};
  return erkNew(2, a, b, c);
}
```

0	0	0
1	1	0
	1/2	1/2

$$
\begin{aligned}
U_{i+1} &= U_i + \Delta t(\tfrac{1}{2}K_1 + \tfrac{1}{2}K_2) \\
K_1 &= f(T_i, U_i) \\
K_2 &= f(T_i + \Delta t, U_i + \Delta t K_1)
\end{aligned}
$$

Solution of homework 1 !

$$
\begin{aligned}
U_{i+1} &= U_i + \Delta t(\tfrac{1}{2}K_1 + \tfrac{1}{2}K_2) \\
K_1 &= f(T_i, U_i) \\
K_2 &= f(T_i + \Delta t, U_i + \Delta t K_1)
\end{aligned}
$$

```
void erkIterate(Erk *erk, int size, double *sol, double t,
     double dt, void(*f)(void *, double*, double, double*), void *data)
{
  int istep, i, ik;
  double *x = malloc(sizeof(double) * size);
  double **k = malloc(sizeof(double*) * erk->nstep);
  for (istep = 0; istep < erk->nstep; ++istep) {
    for (i = 0; i < size; ++i)
      x[i] = sol[i];
    for (ik = 0; ik < istep; ++ik) {
      if (erk->a[istep][ik] != 0.) {
        for (i = 0; i < size; ++i)
          x[i] += k[ik][i] * erk->a[istep][ik] * dt;}}
    k[istep] = malloc(sizeof(double) * size);
    f(data, x, t + dt * erk->c[istep], k[istep]); }
  for (istep = 0; istep < erk->nstep; ++istep) {
    for (i = 0; i < size; ++i)
      sol[i] += k[istep][i] * erk->b[istep] * dt;
    free(k[istep]);}
  free(k); free(x);
}
```

Fun vectors in C !

Ou comment attacher la taille d'un vecteur à un pointeur, sans devoir utiliser la structure auxiliaire dans tout le code !

C'est une utilisation particulière de l'algèbre des pointeurs, pour rendre plus compact le code :-) **To use with care !**

```
typedef struct {
  size_t elementSize;
  size_t nElement;
  size_t nAllocated;
} VectorHeader;
```

```
void *vectorNew(size_t elementSize, size_t nElement)
{
  size_t nAllocated = nElement == 0 ? 1 : nElement;
  VectorHeader *h = malloc(elementSize * nAllocated + sizeof(VectorHeader));
  h->elementSize = elementSize;
  h->nElement = nElement;
  h->nAllocated = nAllocated;
  return h + 1;
}
```

Fun vectors in C !

```
void * vectorResize(void *v, size_t nElement)
{
  VectorHeader *h = ((VectorHeader*)(v)-1);
  if (h->nAllocated <= nElement){
    h->nAllocated *= 2;
    h = realloc(v, h->nAllocated * h->elementSize + sizeof(VectorHeader));
  }
  h->nElement = nElement;
  return h + 1;
}
```

```
size_t vectorSize(void *v)
{
  return ((VectorHeader*)(v)-1)->nElement;
}
```

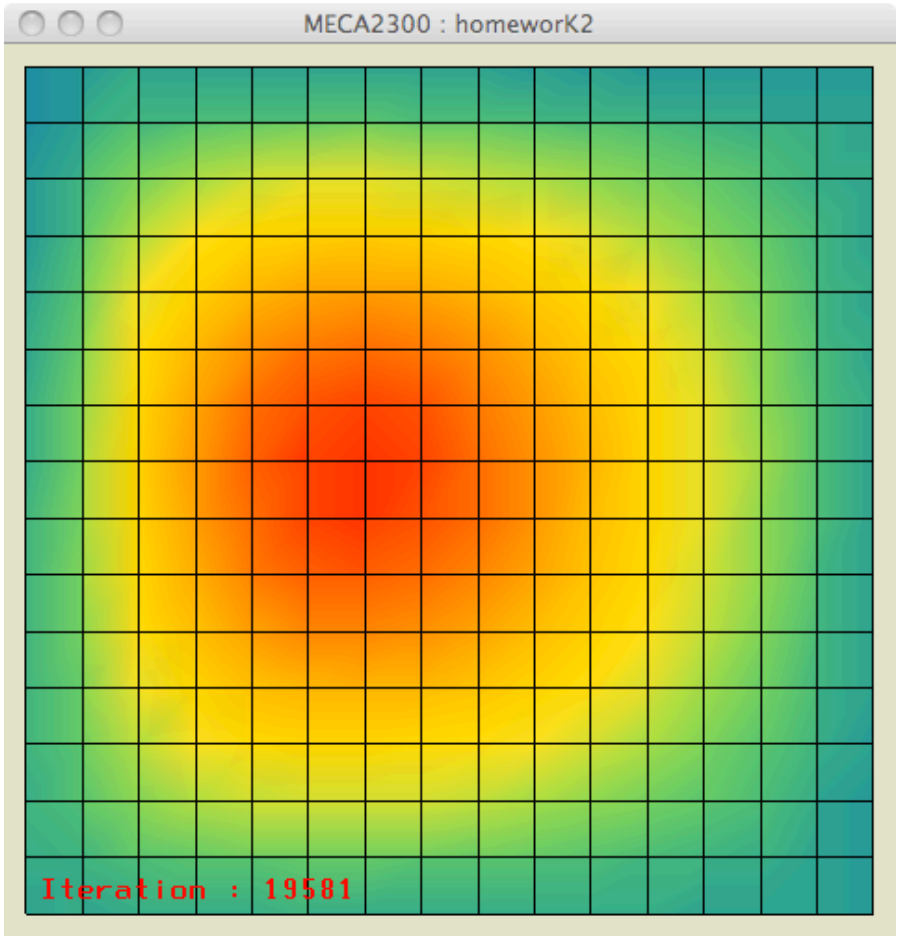
Fun vectors in C !

```
void * vectorResize(void *v, size_t nElement)
{
  VectorHeader *h = VECTOR_HEADER(v);
  if (h->nAllocated <= nElement){
    h->nAllocated *= 2;
    h = realloc(v, h->nAllocated * h->elementSize + sizeof(VectorHeader));
  }
  h->nElement = nElement;
  return h + 1;
}
```

```
size_t vectorSize(void *v)
{
  return VECTOR_HEADER(v)->nElement;
}
```

```
#define VECTOR_HEADER(v)  ((VectorHeader*)(v)-1)
```

Homework 2



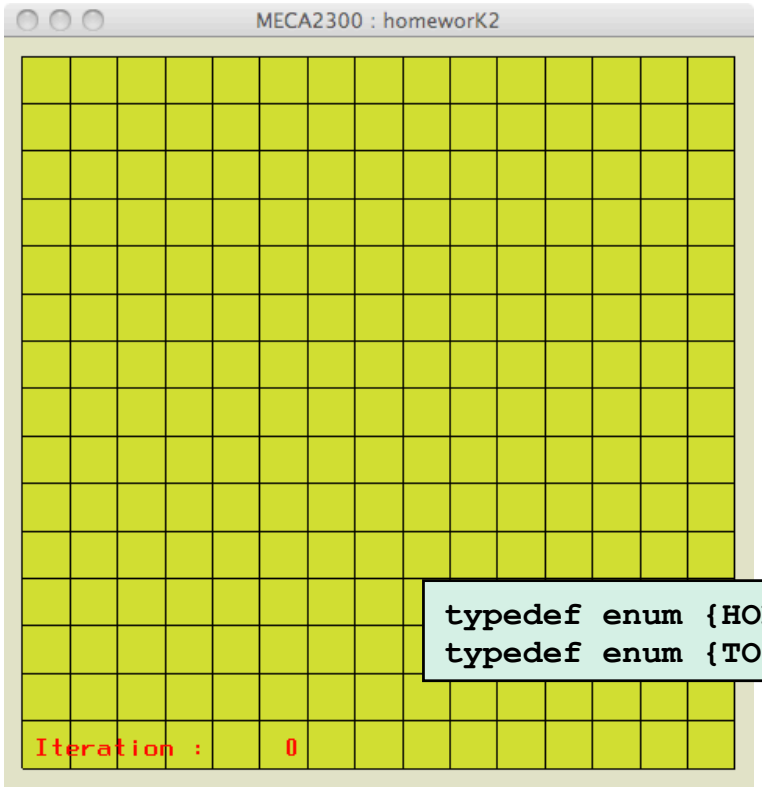
- **Implementing shallow waters equations**
- **Regular grid**
- **Finite volumes**
- **Rieman solvers**

Defining cells, faces and boudaries !

```
typedef struct {
  int i, j;
  int id;
} Cell;
```

```
typedef struct {
  Cell *cell[2];
  faceOrientation orientation;
} Face;
```

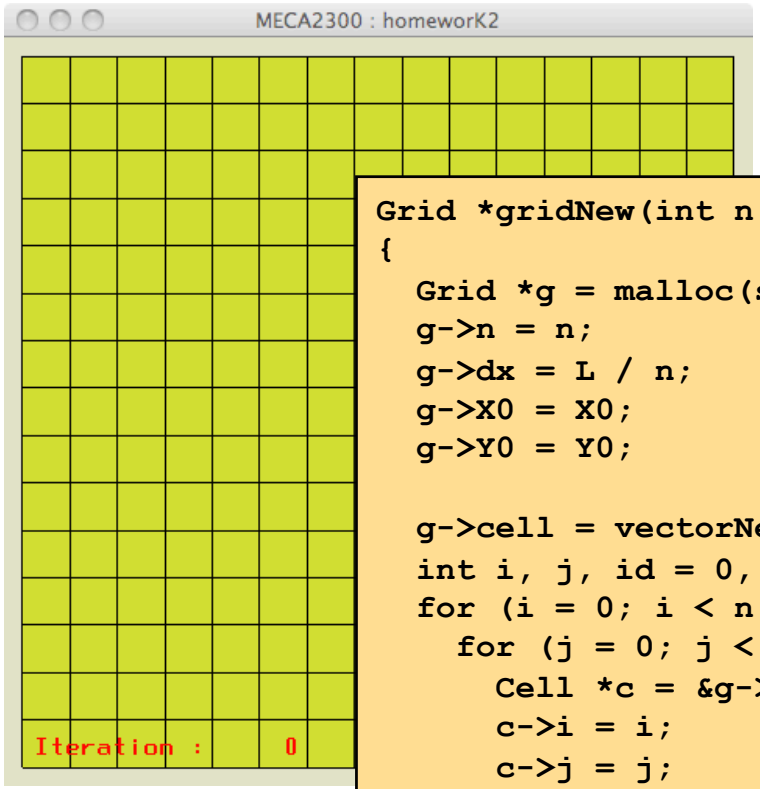
```
typedef struct {
  Cell *cell[2];
  boundarySide side;
} Boundary;
```




```
typedef enum {HORIZONTAL=0,VERTICAL=1} faceOrientation;
typedef enum {TOP=0,LEFT=1,BOTTOM=2,RIGHT=3} boundarySide;
```

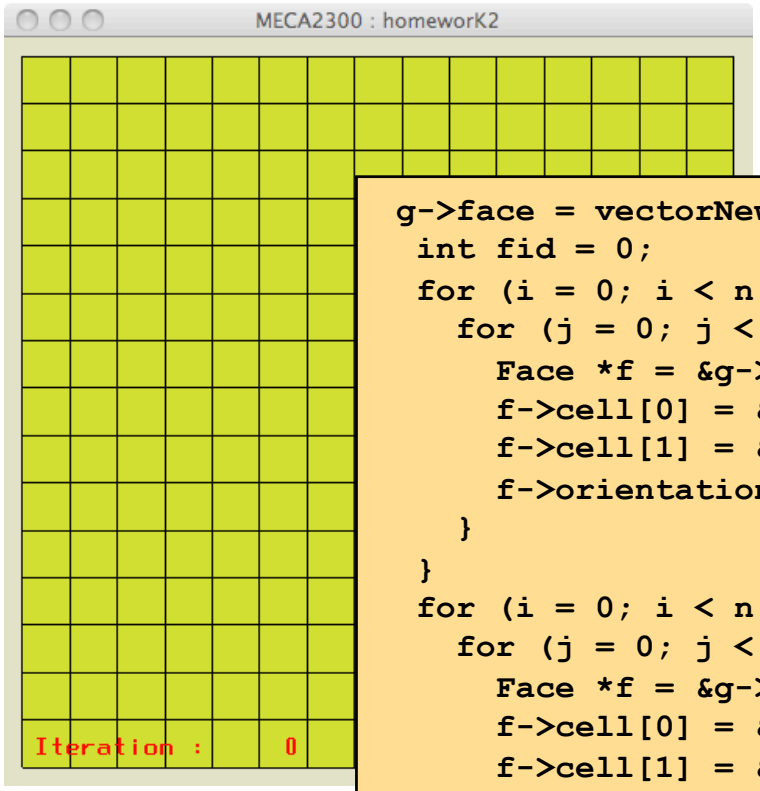
The grid consists of cells …

```
typedef struct {
  int fieldSize;
  Cell *cell;
  Cell *ghostCell;
  Face *face;
  Boundary *boundary;
  int n;
  double dx;
  double X0, Y0;
} Grid;
```



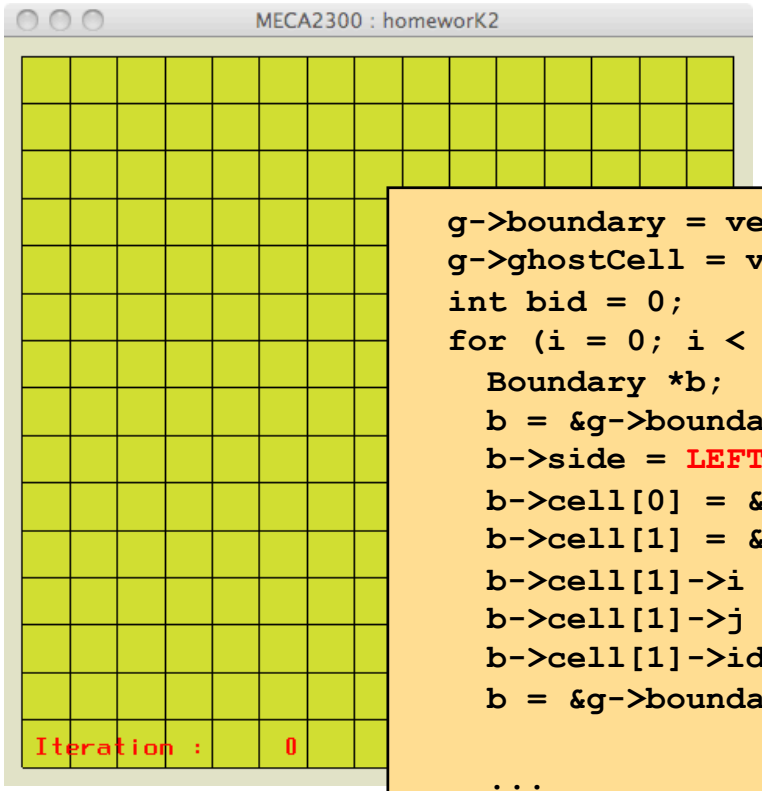

```
Grid *gridNew(int n, double X0, double Y0, double L)
{
  Grid *g = malloc(sizeof(Grid));
  g->n = n;
  g->dx = L / n;
  g->X0 = X0;
  g->Y0 = Y0;

  g->cell = vectorNew(sizeof(Cell), n * n);
  int i, j, id = 0, ghostid = 0;
  for (i = 0; i < n; ++i) {
    for (j = 0; j < n; ++j) {
      Cell *c = &g->cell[id];
      c->i = i;
      c->j = j;
      c->id = id++; }}

...
```

The grid consists of faces …



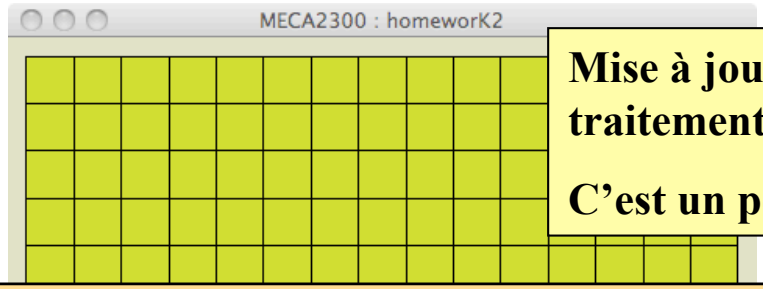

```
g->face = vectorNew(sizeof(Face), 2 * (n + 1) * n);
 int fid = 0;
 for (i = 0; i < n - 1; ++i) {
   for (j = 0; j < n; ++j) {
     Face *f = &g->face[fid++];
     f->cell[0] = &g->cell[i * n + j];
     f->cell[1] = &g->cell[(i + 1) * n + j];
     f->orientation = VERTICAL;
   }
 }
 for (i = 0; i < n; ++i) {
   for (j = 0; j < n - 1; ++j) {
     Face *f = &g->face[fid++];
     f->cell[0] = &g->cell[i * n + j];
     f->cell[1] = &g->cell[i * n + j + 1];
     f->orientation = HORIZONTAL;
   }
 }
```

The grid consists of boudaries …




```
  g->boundary = vectorNew(sizeof(Boundary), 4 * n);
  g->ghostCell = vectorNew(sizeof(Cell), 4 * n);
  int bid = 0;
  for (i = 0; i < n; ++i) {
    Boundary *b;
    b = &g->boundary[bid];
    b->side = LEFT;
    b->cell[0] = &g->cell[0 * n + i];
    b->cell[1] = &g->ghostCell[bid++];
    b->cell[1]->i = -1;
    b->cell[1]->j = i;
    b->cell[1]->id = id++;
    b = &g->boundary[bid];

    ...
}
```

The grid consists of boudaries …



Mise à jour de la structure topologique pour le traitement des frontières !

C'est un peu technique et donc on vous le donne !

```
   for (i = 0; i < vectorSize(g->boundary); ++i) {
    if (g->boundary[i].side == LEFT || g->boundary[i].side == BOTTOM) {
      g->face[fid].cell[0] = g->boundary[i].cell[1];
      g->face[fid].cell[1] = g->boundary[i].cell[0];
    } else {
      g->face[fid].cell[0] = g->boundary[i].cell[0];
      g->face[fid].cell[1] = g->boundary[i].cell[1];
    }
    g->face[fid].orientation = (g->boundary[i].side == RIGHT
                          || g->boundary[i].side == LEFT) ? VERTICAL : HORIZONTAL;
    fid ++;
  }
  g->fieldSize = id;
  return g;
}
```

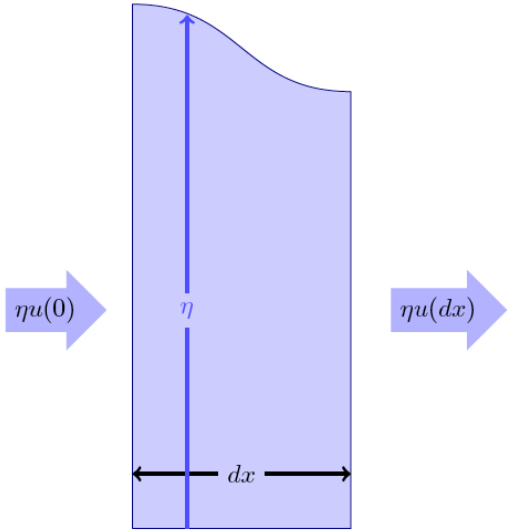
Reconstruction of the data…

```
void gridGradients(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;

     // To be done by you !
   }
}
```

```
void gridFaceReconstruct(Grid *grid, Face *f,
                                     double *field, double *grad,
                                     double *u0, double *u1)
{
     // To be done by you !
}
```

Mass balance

$$\frac{\partial \eta}{\partial t} + \eta_0 \frac{\partial u}{\partial x} = 0$$



$$dx\left(\rho \frac{\partial \eta}{\partial t}\right) = \rho \eta u(0) - \rho \eta u(dx)$$

$$\frac{\partial \eta}{\partial t} + \eta_0 \left(\frac{u(dx) - u(0)}{dx}\right) = 0$$

$$\downarrow$$

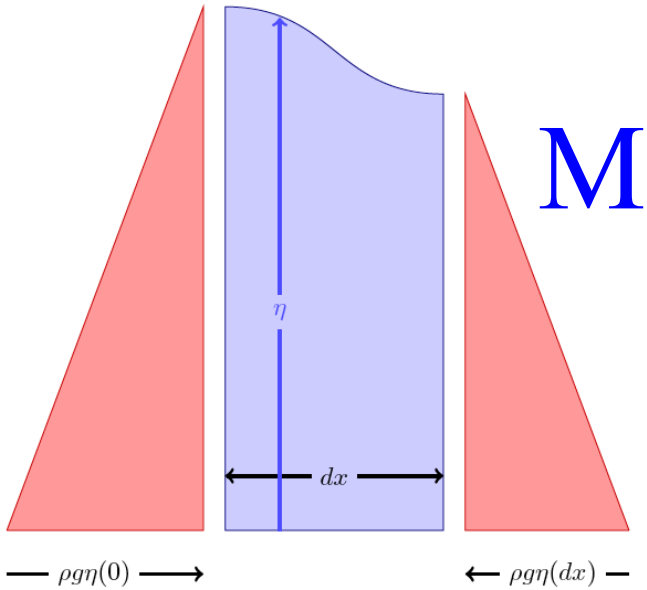
$$\frac{\partial \eta}{\partial t} + \eta_0 \frac{\partial u}{\partial x} = 0$$

$$dx\left(\rho\eta_0\frac{\partial u}{\partial t}\right) = \rho g\frac{\eta^2(0)}{2} - \rho g\frac{\eta^2(dx)}{2}$$

$$\eta_0\frac{\partial u}{\partial t} + g\eta_0\frac{\partial \eta}{\partial x} = 0$$

$$\downarrow$$

$$\frac{\partial u}{\partial t} + g\frac{\partial \eta}{\partial x} = 0$$



Momentum balance

$$\frac{\partial u}{\partial t} + g\frac{\partial \eta}{\partial x} = 0$$

$$\begin{cases} \dfrac{1}{\eta_0}\dfrac{\partial \eta}{\partial t} + \dfrac{\partial u}{\partial x} = 0 \\ \dfrac{\partial u}{\partial t} + g\dfrac{\partial \eta}{\partial x} = 0 \end{cases}$$

Linear Shallow Water Equations

$$\frac{\partial^2 \eta}{\partial t^2} = g\eta_0 \frac{\partial^2 \eta}{\partial x^2}$$

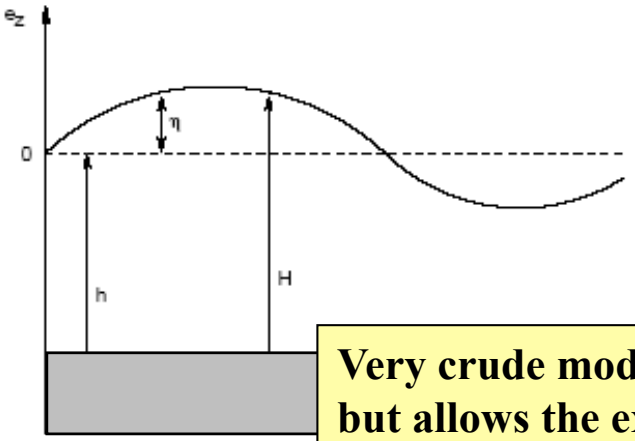
Wave Equation

$$c = \sqrt{g\eta_0} \approx 200 \ [m/s]$$

Gravity 9,81 m/s
Average depth of Pacific 4000 m

Waves are (very) fast !

$$
\begin{cases}
\dfrac{\partial \eta}{\partial t} + \dfrac{\partial}{\partial x}\left(hu\right) + \dfrac{\partial}{\partial y}\left(hv\right) &= 0 \\
\dfrac{\partial u}{\partial t} + \dfrac{\partial}{\partial x}\left(g\eta\right) &= -\gamma u + fv + \dfrac{\tau}{\rho h} \\
\dfrac{\partial v}{\partial t} + \dfrac{\partial}{\partial y}\left(g\eta\right) &= -\gamma v - fu
\end{cases}
$$



Very crude model for geophysical flows, but allows the existence of inertia-gravity waves

The so-called Shallow Water Equations

An analytical problem as a numerical validation : Stommel :-)

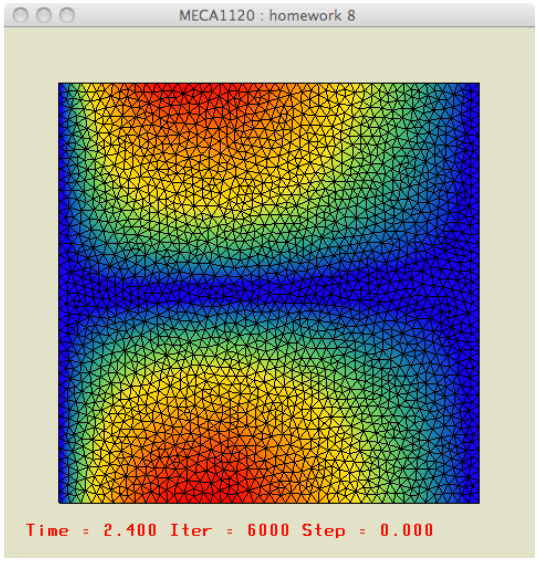
$$\begin{cases} \dfrac{\partial \eta}{\partial t} + \dfrac{\partial}{\partial x}\left(hu\right) + \dfrac{\partial}{\partial y}\left(hv\right) = 0 \\ \dfrac{\partial u}{\partial t} + \dfrac{\partial}{\partial x}\left(g\eta\right) = -\gamma u + fv + \dfrac{\tau}{\rho h} \\ \dfrac{\partial v}{\partial t} + \dfrac{\partial}{\partial y}\left(g\eta\right) = -\gamma v - fu \end{cases}$$

Forcing wind term [N m^{-2}]

Gravity [m s^{-2}]

Coriolis factor [s^{-1}]

Dissipation coefficient [s^{-1}]



$$Fr = \frac{U}{c} = \frac{U}{\sqrt{gH}}$$

$$Ro = \frac{U}{Lf}$$

Defining the Stommel problem:-)

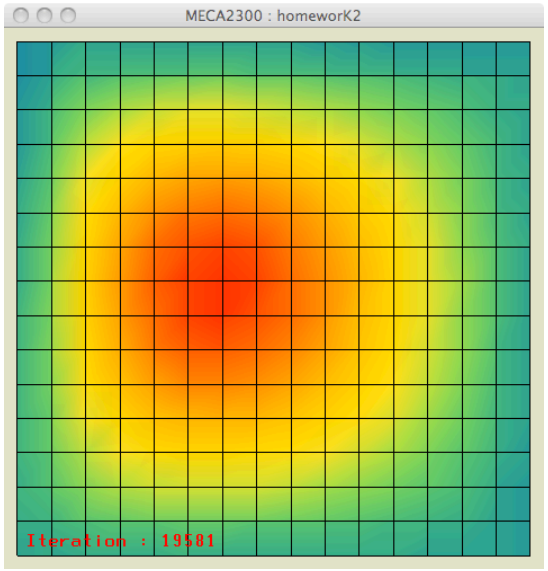
$$\begin{cases} \dfrac{\partial \eta}{\partial t} + \dfrac{\partial}{\partial x}\left(hu\right) + \dfrac{\partial}{\partial y}\left(hv\right) = 0 \\ \dfrac{\partial u}{\partial t} + \dfrac{\partial}{\partial x}\left(g\eta\right) = -\gamma u + fv + \dfrac{\tau}{\rho h} \\ \dfrac{\partial v}{\partial t} + \dfrac{\partial}{\partial y}\left(g\eta\right) = -\gamma v - fu \end{cases}$$

Forcing wind term [N m^{-2}]

Gravity [m s^{-2}]

Coriolis factor [s^{-1}]

Dissipation coefficient [s^{-1}]




```
typedef struct {
  Grid *grid;
  double g;
  double z;
  double gamma;
  double beta;
  double f0;
} Swe;
```

Defining the material parameters

$$\begin{cases} \dfrac{\partial \eta}{\partial t} + \dfrac{\partial}{\partial x}\left(hu\right) + \dfrac{\partial}{\partial y}\left(hv\right) = 0 \\ \dfrac{\partial u}{\partial t} + \dfrac{\partial}{\partial x}\left(g\eta\right) = -\gamma u + fv + \dfrac{\tau}{\rho h} \\ \dfrac{\partial v}{\partial t} + \dfrac{\partial}{\partial y}\left(g\eta\right) = -\gamma v - fu \end{cases}$$

Forcing wind term [N m^{-2}]

Gravity [m s^{-2}]

Coriolis factor [s^{-1}]

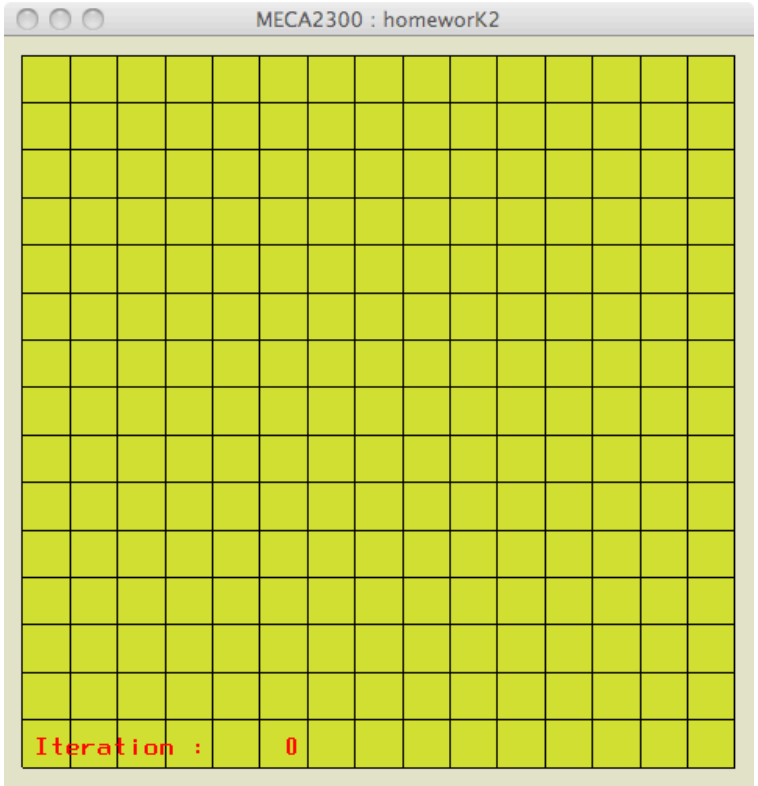
Dissipation coefficient [s^{-1}]

```
typedef struct {
  Grid *grid;
  double g;
  double z;
  double gamma;
  double beta;
  double f0;
} Swe;
```

```
Swe *sweNew(Grid *grid)
{
  Swe *swe = (Swe*) malloc(sizeof(Swe));
  swe->grid = grid;
  swe->g = 9.81;
  swe->z = -1000;
  swe->gamma = 0.5e-6;
  swe->f0 = 1e-4;
  swe->beta = 1e-11;
  return swe;
}
```

Initialising the model !

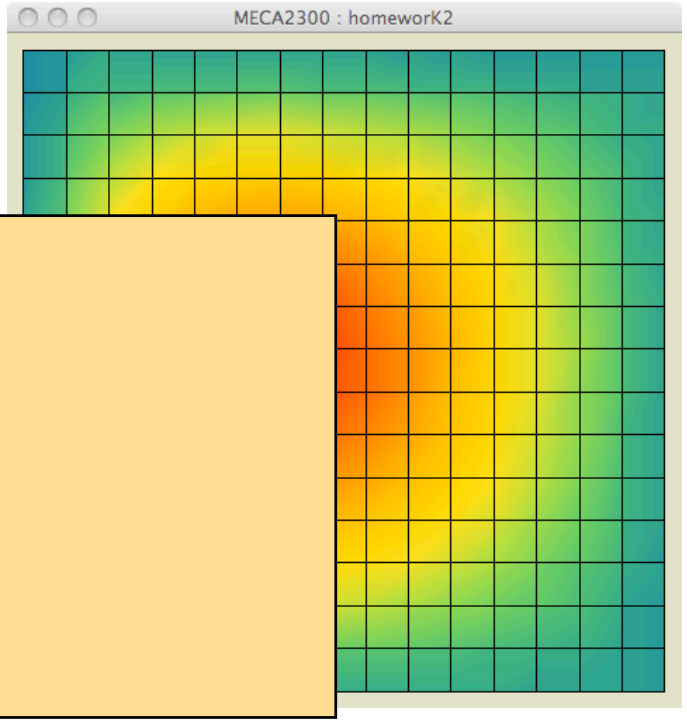
```
typedef struct {
  Grid *grid;
  double g;
  double z;
  double gamma;
  double beta;
  double f0;
} Swe;
```




```
double sweInit(Swe *swe, double *sol)
{
  int i, j;
  Grid *g = swe->grid;
  double *H = sol;
  double *U = sol + g->fieldSize;
  double *V = sol + 2 * g->fieldSize;
  for (i = 0; i < vectorSize(g->cell); ++i) {
    Cell *c = &g->cell[i];
    U[c->id] = 0.;
    V[c->id] = 0.;
    H[c->id] = 1000.;
  }
}
```

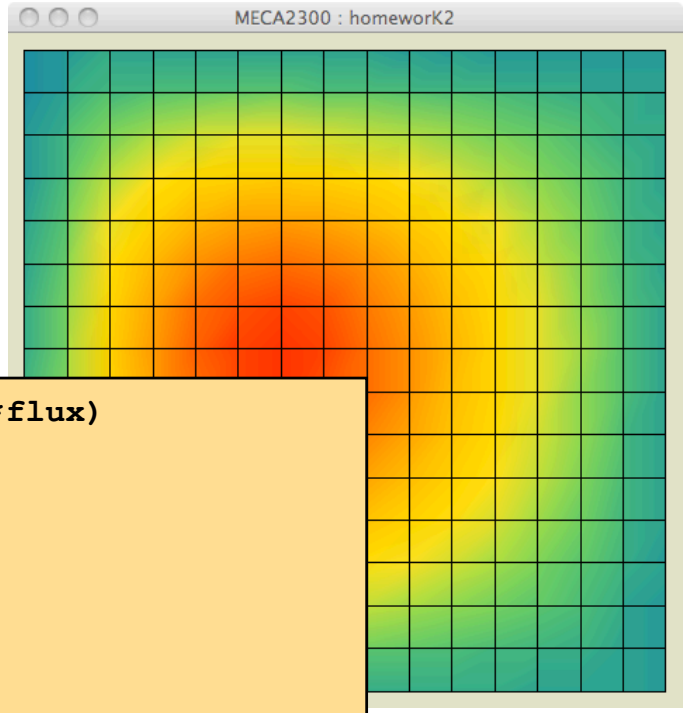
Integrating the model !

```
typedef struct {
  Grid *grid;
  double g;
  double z;
  double gamma;
  double beta;
  double f0;
} Swe;
```




```
Erk *erk = erkNew22();
double L = 1e6;
Grid *grid = gridNew(15,-L/2,-L/2,L);
Swe *swe = sweNew(grid);
double *v = malloc(sizeof(double)*grid->fieldSize*3);
sweInit(swe, v);
double dt = 100;
int i;
for (i = 0; i < 10000; ++i)
    erkIterate(erk, 3*grid->fieldSize,v,0.0,dt,
                    (ErkCallback*)sweCompute,swe);
```

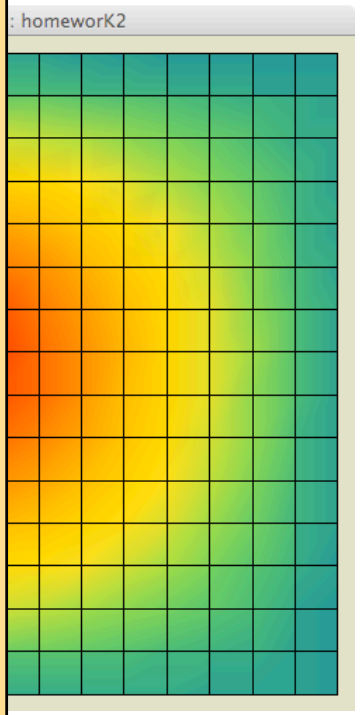
Integrating the model !




```
void sweCompute(Swe *swe, double *s, double t, double *flux)
{
  Grid *grid = swe->grid;
  double *h = s;
  double *u = s + grid->fieldSize;
  double *v = s + 2 * grid->fieldSize;
  double *FH = flux
  double *FU = flux + grid->fieldSize;
  double *FV = flux + 2 * grid->fieldSize;

  int i;
  for (i = 0; i < vectorSize(grid->cell); ++i) {
    Cell *c = &grid->cell[i];
    double x, y;
    gridCellCenter(grid, c, &x, &y);

     // To be computed by you !

    FU[c->id] = 0.0;
    FV[c->id] = 0.0;
    FH[c->id] = 0.0; }
  sweBoundaryConditions(swe, h, u, v);
...
```

Integrating the model !

```
  double *gradh = malloc(sizeof(double) * 2 * grid->fieldSize);
  double *gradu = malloc(sizeof(double) * 2 * grid->fieldSize);
  double *gradv = malloc(sizeof(double) * 2 * grid->fieldSize);
  gridGradients(grid, u, gradu);
  gridGradients(grid, v, gradv);
  gridGradients(grid, h, gradh);
  sweBoundaryConditionsGradient(swe, gradh, gradu, gradv);

  for (i = 0; i < vectorSize(grid->face); ++i) {
    Face *f = &grid->face[i];

    // To do : compute the fluxes with the Rieman solver !

    sweRiemann( )

    // End of todo work !

  }

  free(gradh);
  free(gradu);
  free(gradv);
}
```



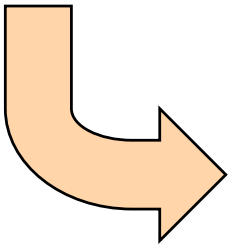
Un modèle unidimensionnel le long de l'interface…

$$\begin{cases} \dfrac{\partial \eta}{\partial t} + h\dfrac{\partial u}{\partial x} = 0 \\ \dfrac{\partial u}{\partial t} + g\dfrac{\partial \eta}{\partial x} = 0 \end{cases}$$

```
void sweRiemann(Swe *swe, double hl, double hr, double ul, double ur, double vl,
                          double vr, double *fh, double *fu, double *fv)
{
   // To do : compute the fluxes !
}
```

Un solveur de Riemann…

$$\begin{cases} \dfrac{\partial \eta}{\partial t} + h \dfrac{\partial u}{\partial x} = 0 \\ \dfrac{\partial u}{\partial t} + g \dfrac{\partial \eta}{\partial x} = 0 \end{cases}$$



$$\begin{bmatrix} \dfrac{\partial \eta}{\partial t} \\ \dfrac{\partial u}{\partial t} \end{bmatrix} = \underbrace{\begin{bmatrix} 0 & h \\ g & 0 \end{bmatrix}}_{\mathbf{A}} \begin{bmatrix} \dfrac{\partial \eta}{\partial x} \\ \dfrac{\partial u}{\partial x} \end{bmatrix}$$

Calculons les valeurs propres de A pour découpler les deux équations…

$$\begin{bmatrix} \frac{\partial \eta}{\partial t} \\ \frac{\partial u}{\partial t} \end{bmatrix} = \underbrace{\begin{bmatrix} 0 & h \\ g & 0 \end{bmatrix}}_{\mathbf{A}} \begin{bmatrix} \frac{\partial \eta}{\partial x} \\ \frac{\partial u}{\partial x} \end{bmatrix}$$

$$\lambda = \pm\sqrt{gh}$$

Deux valeurs propres

Deux vecteurs propres

$$\mathbf{v} = \begin{bmatrix} 1 \\ \pm\sqrt{\frac{g}{h}} \end{bmatrix}$$

Effectuons un changement de variables…

$$\begin{bmatrix} r \\ s \end{bmatrix} = \underbrace{\begin{bmatrix} \frac{1}{2} & \frac{1}{2}\sqrt{\frac{h}{g}} \\ \frac{1}{2} & -\frac{1}{2}\sqrt{\frac{h}{g}} \end{bmatrix}}_{\mathbf{R}^{-1}} \begin{bmatrix} \eta \\ u \end{bmatrix}$$

r et s sont appelées les invariants de Riemann :-)

Matrice des deux vecteurs propres

$$\mathbf{R} = \begin{bmatrix} 1 & 1 \\ \sqrt{\frac{g}{h}} & -\sqrt{\frac{g}{h}} \end{bmatrix}$$

Et on obtient…

$$\left(\frac{\partial \eta}{\partial t} + h\frac{\partial u}{\partial x}\right) + \sqrt{\frac{h}{g}}\left(\frac{\partial u}{\partial t} + g\frac{\partial \eta}{\partial x}\right) = 0$$

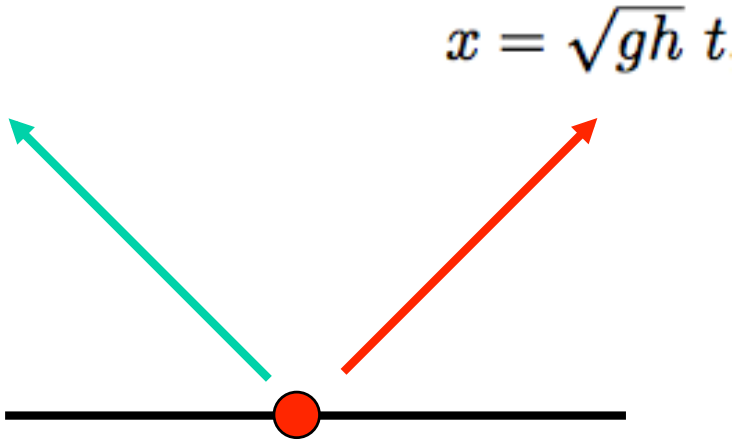
$$\frac{\partial}{\partial t}\Big(\underbrace{\eta + \sqrt{\frac{h}{g}}\,u}_{r}\Big) + \sqrt{gh}\,\frac{\partial}{\partial x}\Big(\underbrace{\eta + \sqrt{\frac{h}{g}}\,u}_{r}\Big) = 0$$

$$\left(\frac{\partial \eta}{\partial t} + h\frac{\partial u}{\partial x}\right) - \sqrt{\frac{h}{g}}\left(\frac{\partial u}{\partial t} + g\frac{\partial \eta}{\partial x}\right) = 0$$

$$\frac{\partial}{\partial t}\Big(\underbrace{\eta - \sqrt{\frac{h}{g}}\,u}_{s}\Big) - \sqrt{gh}\,\frac{\partial}{\partial x}\Big(\underbrace{\eta - \sqrt{\frac{h}{g}}\,u}_{s}\Big) = 0$$

… deux équations de transport découplées !

Les invariants de Riemann sont constants le long des courbes caractéristiques !



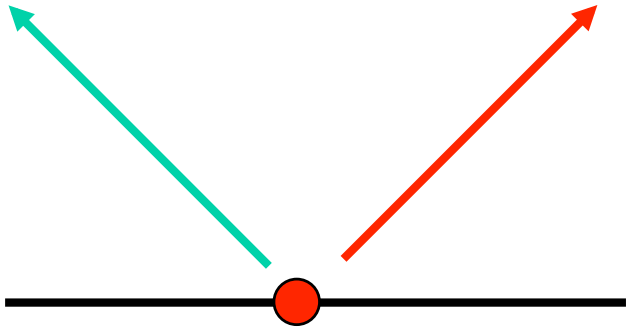
$$\frac{\partial r}{\partial t} + \sqrt{gh}\,\frac{\partial r}{\partial x} = 0$$

$\downarrow$ Car $\sqrt{gh} = \frac{dx}{dt}$ sur la courbe caractéristique

$$\underbrace{\frac{\partial r}{\partial t} + \frac{dx}{dt}\,\frac{\partial r}{\partial x}}_{\frac{dr}{dt}} = 0$$

Et on sait
ce qu'il faut
faire pour
une équation
de transport pur !

$$x = \sqrt{gh}\, t$$



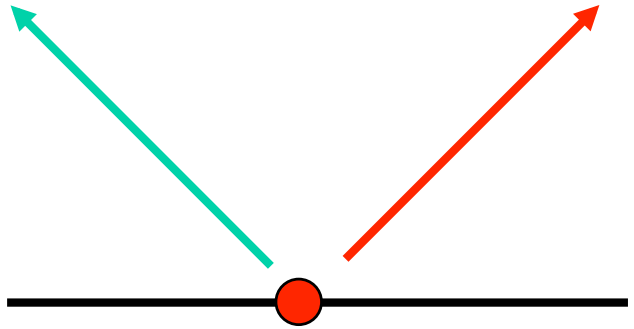
$$\left(\eta + \sqrt{\tfrac{h}{g}}\, u\right)^* = r^* = r_L = \left(\eta_L + \sqrt{\tfrac{h}{g}}\, u_L\right)$$

$$\left(\eta - \sqrt{\tfrac{h}{g}}\, u\right)^* = s^* = s_R = \left(\eta_R - \sqrt{\tfrac{h}{g}}\, u_R\right)$$

Le solveur dit de Riemann :-)

Et en termes de vitesses et d'élévation

$$x = \sqrt{gh}\, t$$



$$\left(\eta + \sqrt{\tfrac{h}{g}}\, u\right)^* = r^* = r_L = \left(\eta_L + \sqrt{\tfrac{h}{g}}\, u_L\right)$$

$$\left(\eta - \sqrt{\tfrac{h}{g}}\, u\right)^* = s^* = s_R = \left(\eta_R - \sqrt{\tfrac{h}{g}}\, u_R\right)$$

$$\eta^* = \left(\frac{r_L + s_R}{2}\right) = \left(\frac{\eta_L + \eta_R}{2}\right) + \sqrt{\frac{h}{g}}\left(\frac{u_L - u_R}{2}\right)$$

$$u^* = \sqrt{\frac{g}{h}}\left(\frac{r_L - s_R}{2}\right) = \left(\frac{u_L + u_R}{2}\right) + \sqrt{\frac{g}{h}}\left(\frac{\eta_L - \eta_R}{2}\right)$$

Le solveur dit de Riemann :-)

Last part of the work !

```
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;

 // To do : provide the zeros !

    h[b->cell[1]->id] = 0.;
    un[b->cell[1]->id] = 0.;
    ut[b->cell[1]->id] = 0.;
  }
}

void sweBoundaryConditionsGradient(Swe *swe, double *gh, double *gu, double *gv)
{
 // To do : provide the values of the gratient!

}
```