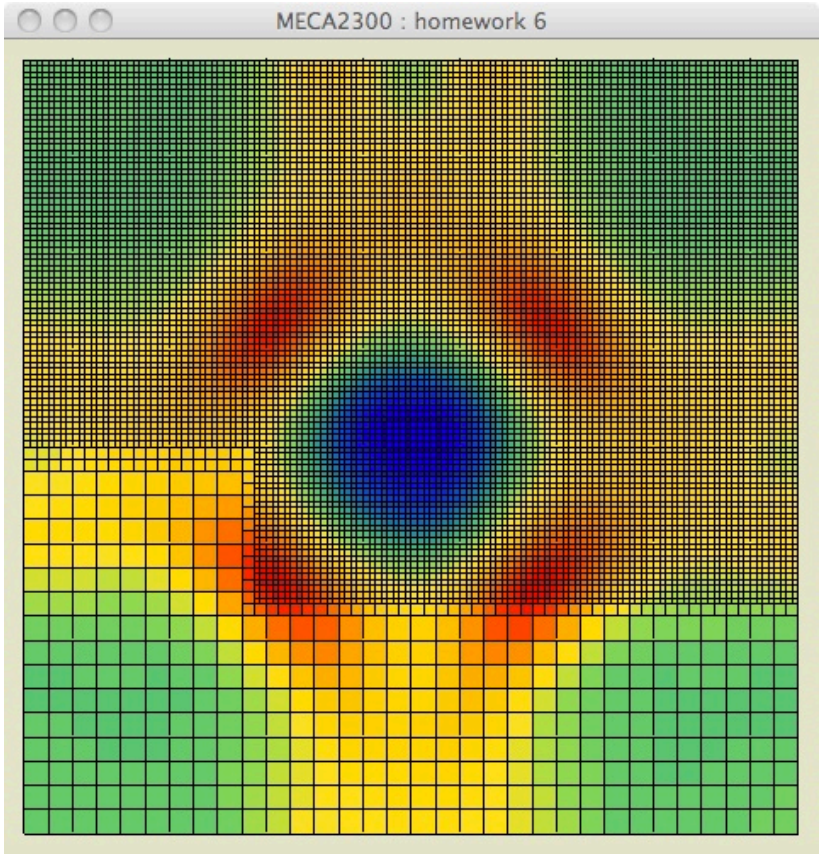
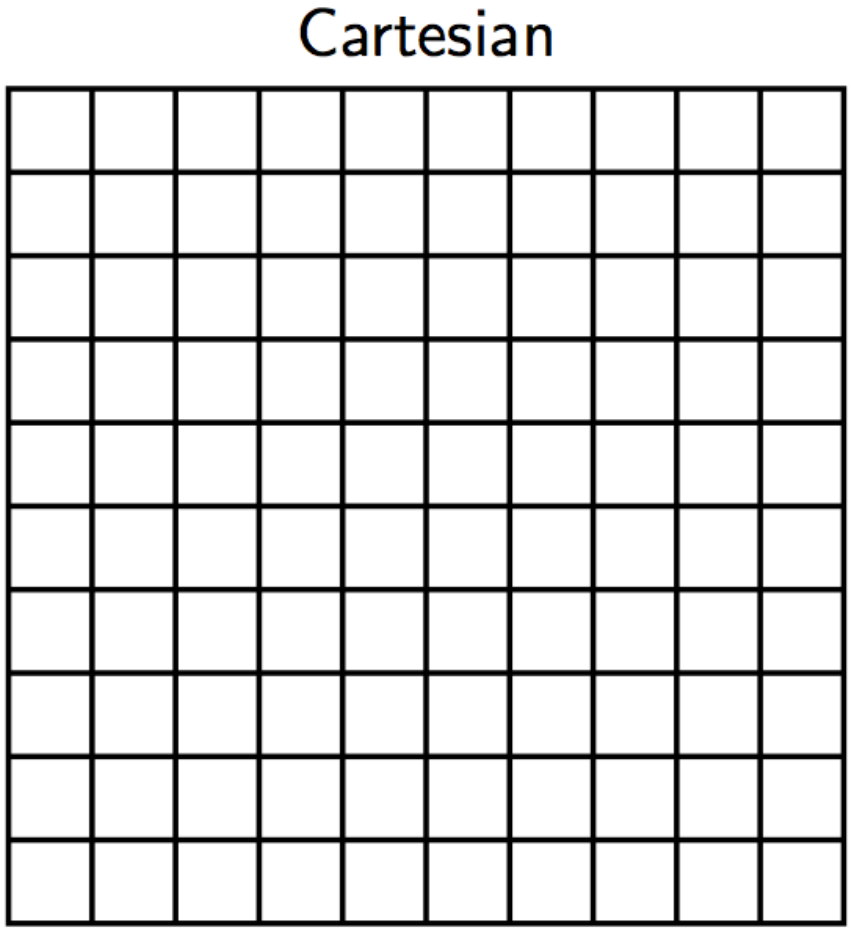
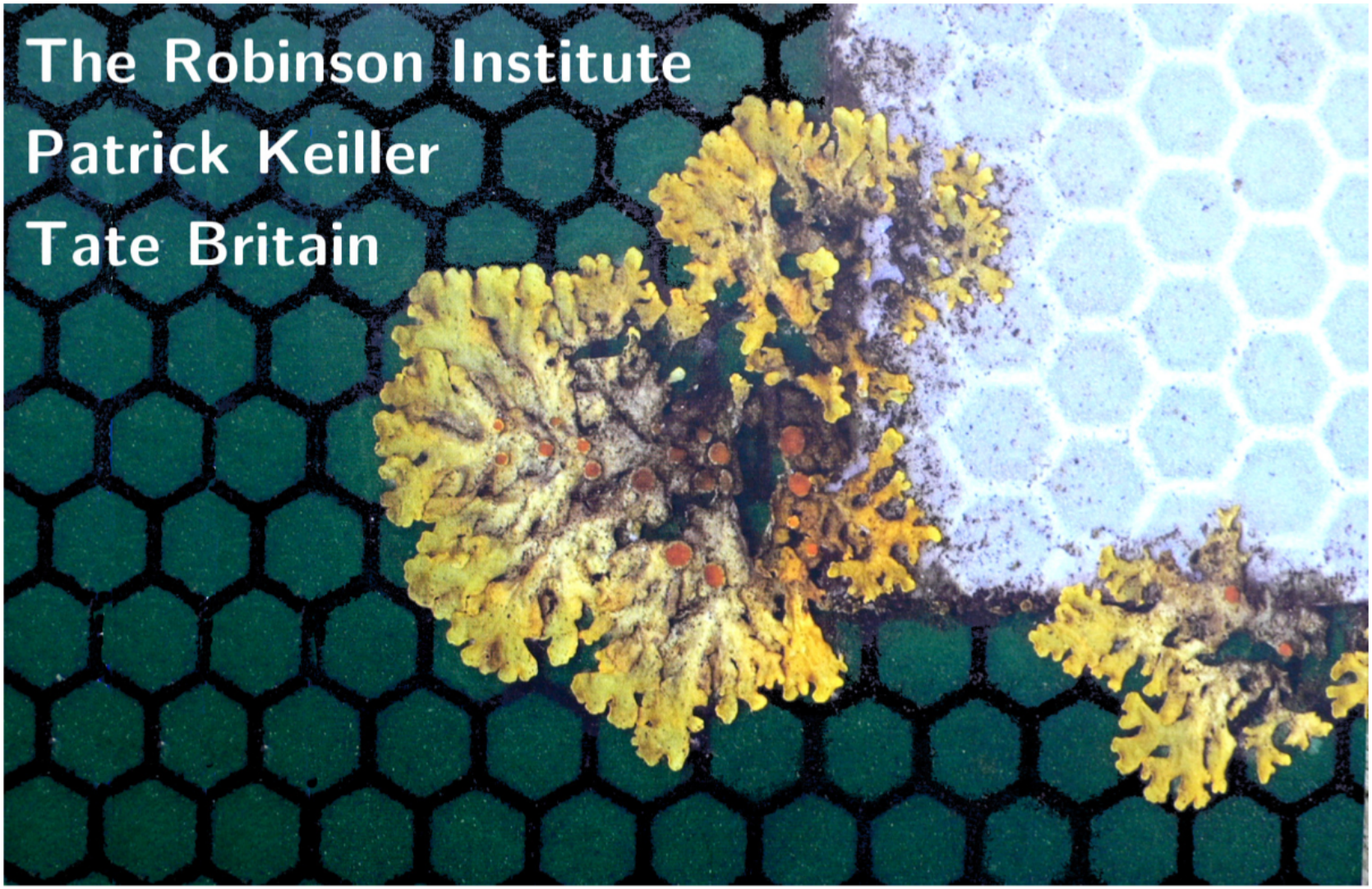
The final goal :-)

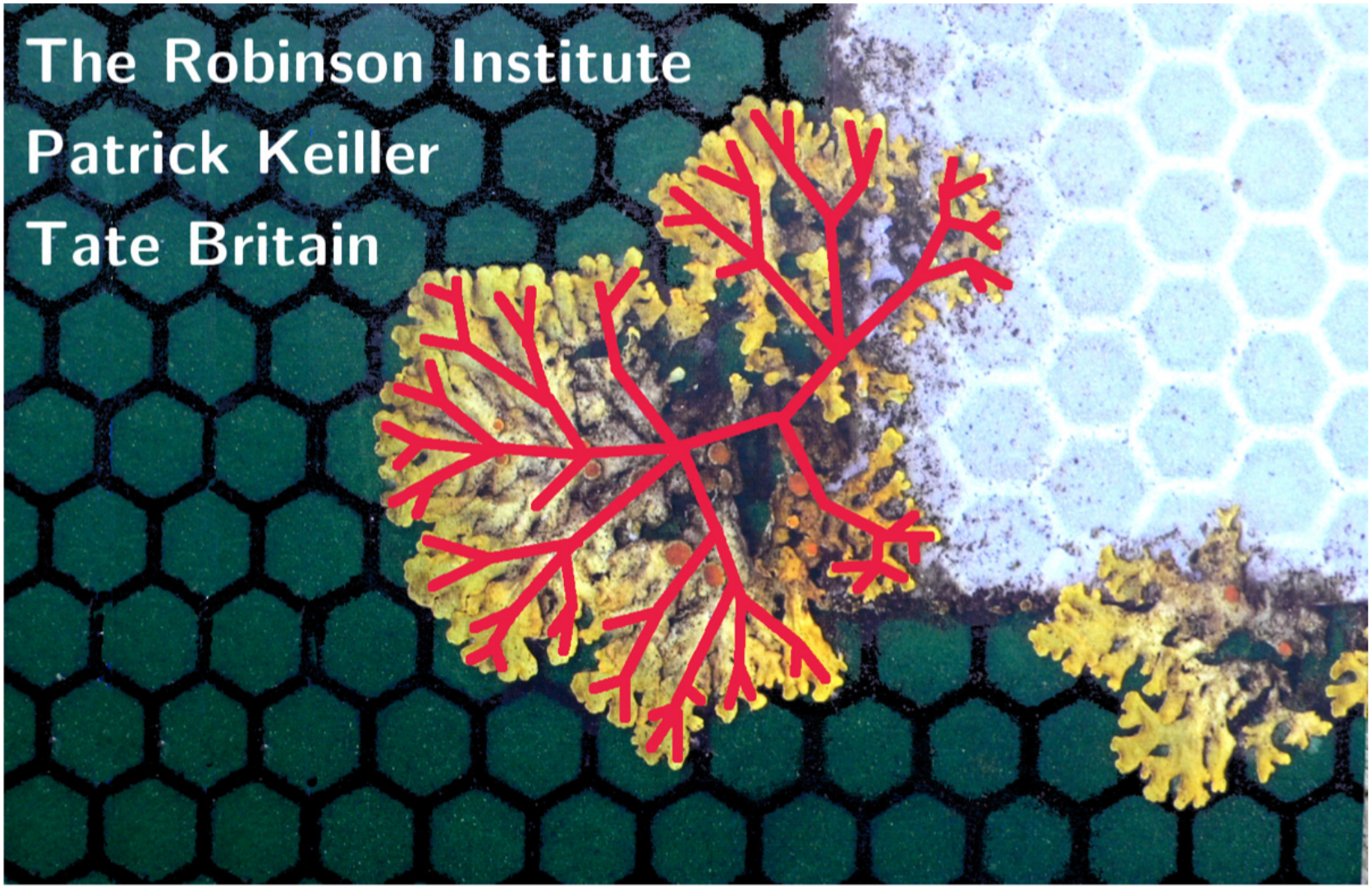


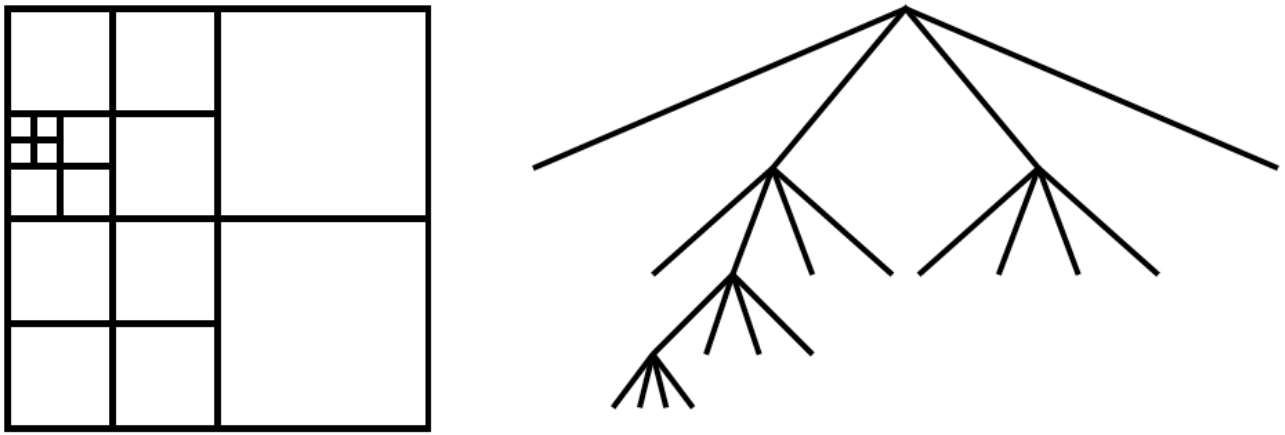
- **Implementing adaptive mesh methods**
- **Understanding Rieman solvers**
- **Using efficient data structures**
- **Trying to optimize the code**
- **Discovering multigrids schemes**

Regular grids




The Robinson Institute
Patrick Keiller
Tate Britain

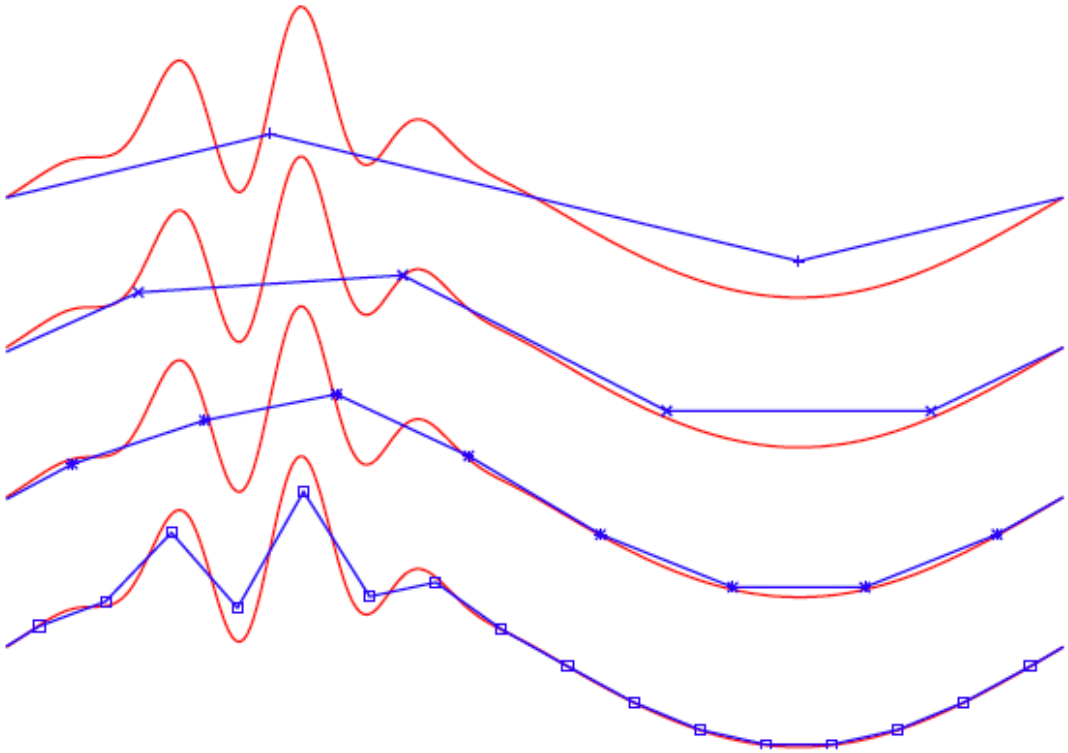

The Robinson Institute
Patrick Keiller
Tate Britain



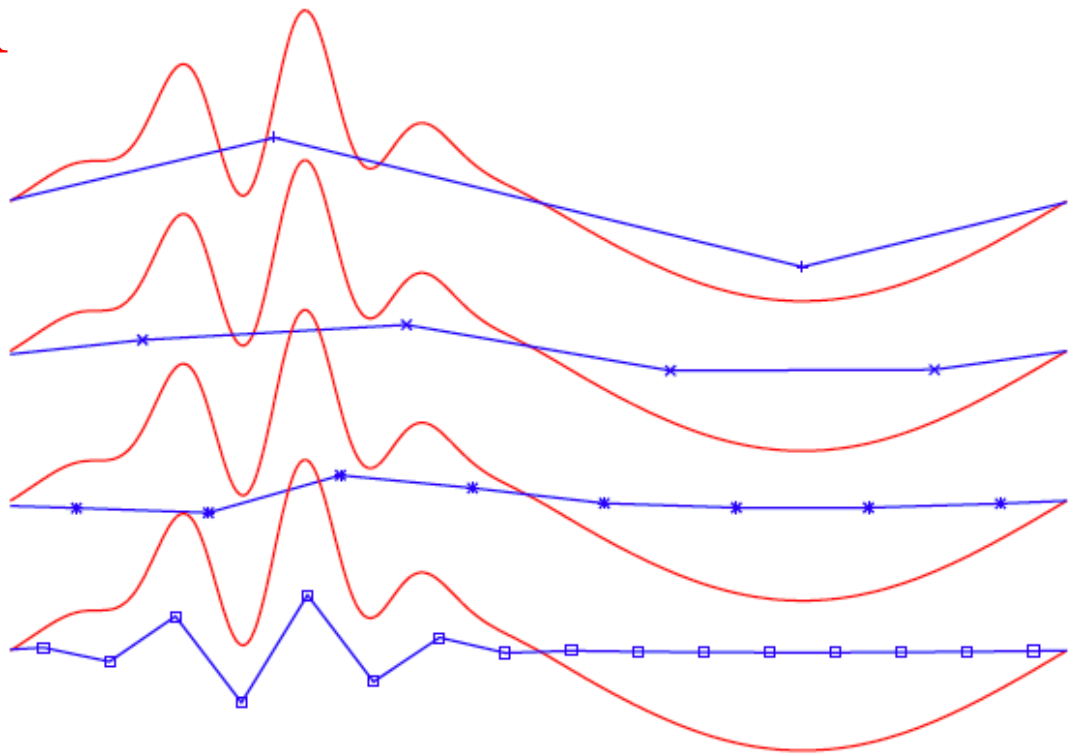
Fully adaptive in space and time
Robust and automatic meshing

Dynamic refinement using quadtrees

A natural multi-scale/frequency representation



A natural multi-scale/frequency representation



- **Efficient multigrid solvers for linear/nonlinear system**
- **A large collection of other efficient « divide-and-conquer » algorithms**
- **Formally linked to wavelets/multifractals**

Explicit Runge-Kutta

$$u'(t) = f(u(t), t)$$

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

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

- **Easy to implement and parallelize**
- **Low cost per time step**
- **Limiters and clippers can be easily introduced**
- **Maximum stable time step limited by CFL condition**
- **Extremely inefficient for stationary problems**

Implementing Runge-Kutta methods

```
typedef struct {
  double **a, *b, *c;
  int nstep;
}Erk;
```

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

```
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

Defining a problem !

```
typedef struct {
  double a;
  double b;
  double c;
  double d;
} Forest;
```

$$\begin{cases} x'(t) &= x(t)\ (a - by(t)) \\ y'(t) &= y(t)\ (dx(t) - c) \\ \\ x(0) &= 10^8 \\ y(0) &= 20 \end{cases}$$

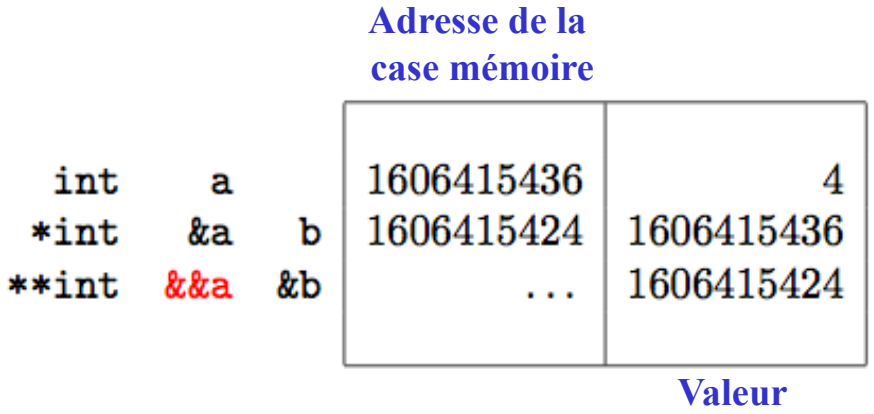
```
void forestUpdate(Forest *forest, double *x, double t, double *f)
{
  f[0] = x[0] * (forest->a - forest->b * x[1]);
  f[1] = x[1] * (forest->d * x[0] - forest->c);
}
```

Integrating the problem

```
int main() {
  Erk *erk = erkNew22();
  Forest *forest = malloc(sizeof(Forest));
  forest->a = 2e-5;
  forest->b = 1e-8;
  forest->c = 6e-5;
  forest->d = 7e-13;
  double x[2] = {10e8, 20};
  double t = 0;
  double dt = 1e3;
  int i;
  for (i = 0; i < 500; ++i) {
    t += dt;
    erkIterate(erk, 2, x, t, dt, (ErkCallback*)forestUpdate, forest);
    if (i %100 == 0 || i< 5) {
      printf("t = %10g    x[0] = %8.2e    x[1] = %8.2e\n", t, x[0], x[1]);
    }
  }
  free(forest);
  erkFree(erk);
}
```

Le C est un langage de bas niveau : les pointeurs ;-(

```
int main(void)
{
   int a = 4;
   printf(" ====   a === %d \n",a);
   printf(" ====  &a === %d \n",&a);
   int *b = &a;  // &&a do not exist : why ?
   printf(" ==== &&a === %d \n",&b);
   exit(0);
}
```



L'utilisation des pointeurs permet d'écrire des codes très efficaces et rapides…
Par contre, programmer est une tâche plus délicate et fastidieuse.
Mais, la rapidité d'un code est critique pour la simulation numérique (et le jeux !) :

Le langage C (ou C++) est bon choix ici !

Le C est un langage de bas niveau les pointeurs ;-(

```
int a = 0;
int *b = &a;
b[0] = 4;
printf(" ====  *b  === %d \n",*b);
printf(" ==== b[0] === %d \n",b[0]);
printf(" ====   a  === %d \n",a);
printf(" ====   b  === %d \n",b);
```

				Adresse de la case mémoire	
int	a	*b	b[0]	1606415436	4
*int	&a	b		1606415424	1606415436

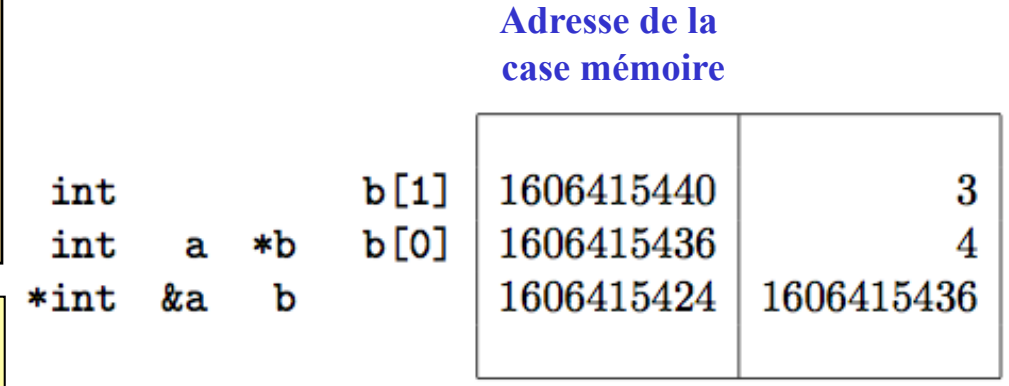
L'utilisation des pointeurs permet d'écrire des codes très efficaces et rapides…
Par contre, programmer est une tâche plus délicate et fastidieuse.
Mais, la rapidité d'un code est critique pour la simulation numérique (et le jeux !) :

Le langage C (ou C++) est bon choix ici !

On peut écrire n'importe où dans la mémoire de l'ordinateur ! Ouuuuuuups : c'est pas joli

```
int a = 0;
int *b = &a;
b[0] = 4;
b[1] = 3;
printf(" ====  *b  === %d \n",*b);
printf(" ==== b[0] === %d \n",b[0]);
printf(" ==== b[1] === %d \n",b[1]);
printf(" ====   a  === %d \n",a);
printf(" ====   b  === %d \n",b);
printf(" ==== &b[0] == %d \n",&b[0]);
printf(" ==== &b[1] == %d \n",&b[1]);
```

Et le pire, c'est que cela marche parfois…
Parfois pas : `Segmentation fault`



Lire le maillage
Allocation dynamique
de mémoire : malloc-free

```
fscanf(file, "Number of nodes %d \n", &theMesh->nNode);
theMesh->X = malloc(sizeof(double)*theMesh->nNode);
theMesh->Y = malloc(sizeof(double)*theMesh->nNode);
for (i = 0; i < theMesh->nNode; ++i) {
  fscanf(file,"%d : %le %le \n",&trash,&theMesh->X[i],&theMesh->Y[i]); }

fscanf(file, "Number of elements %d \n", &theMesh->nElem);
theMesh->elem = malloc(sizeof(int)*3*theMesh->nElem);
   for (i = 0; i < theMesh->nElem; ++i) {
      elem = &(theMesh->elem[i*3]);
      fscanf(file,"%d : %d %d %d \n", &trash,&elem[0],&elem[1],&elem[2]); }
```

Attention

Il ne faut pas allouer toute la mémoire de votre ordi :-(

Ne pas oublier de désallouer !