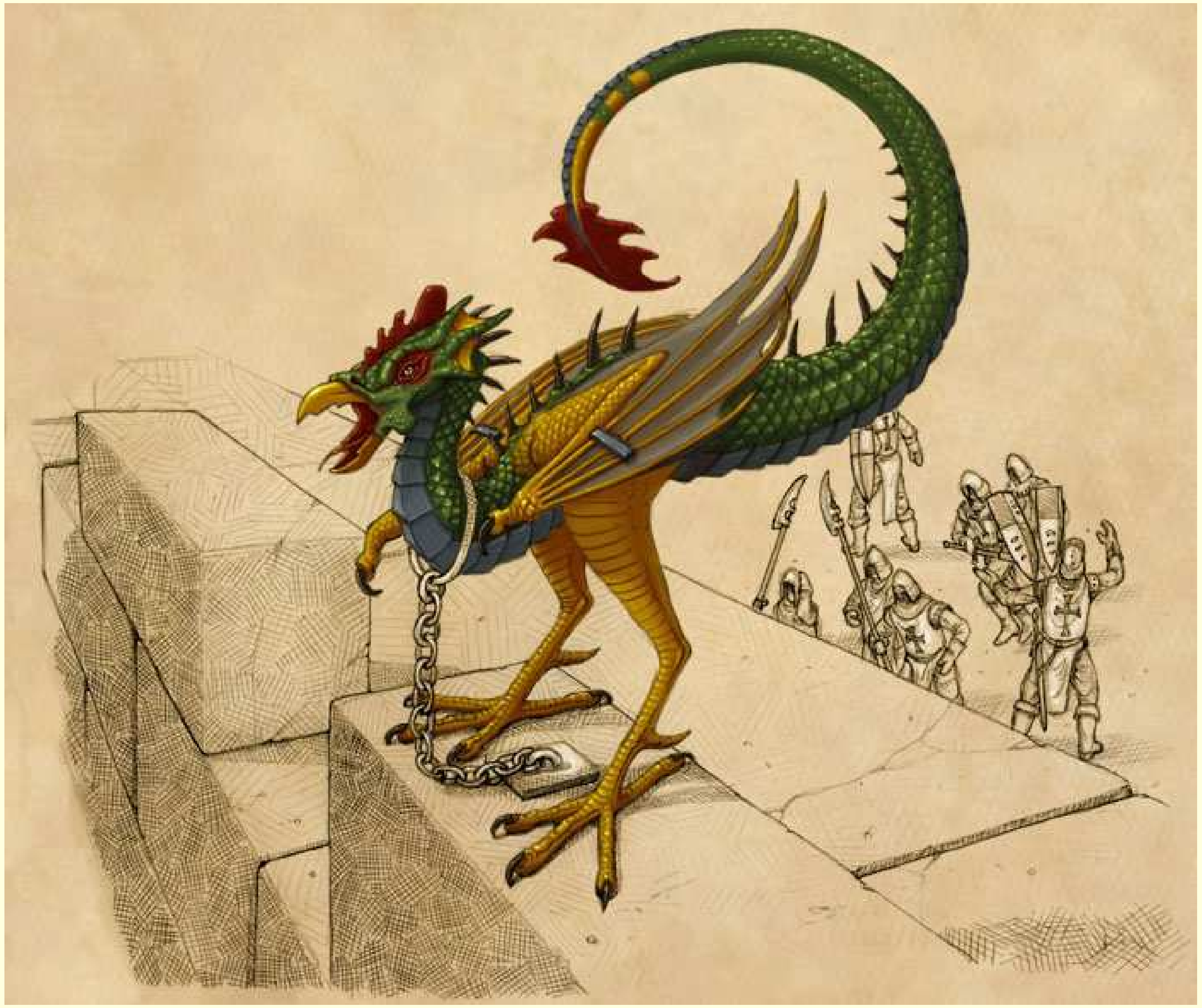
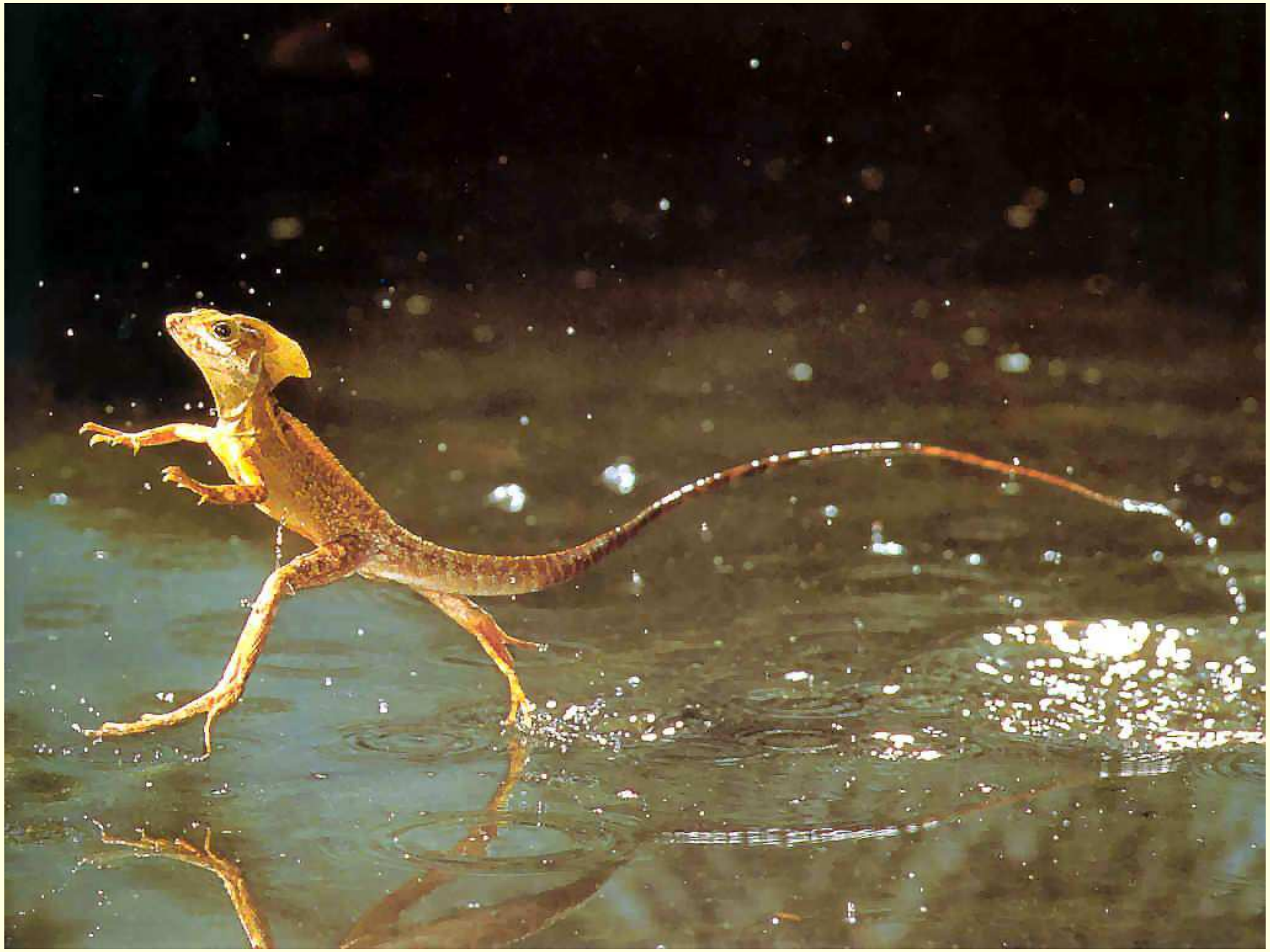
Basilisk



Basilisk according to David Deen

Gerris lacustris

Basiliscus basiliscus



Gerris

Strong points

- Adaptivity, precision, surface tension, flexibility
- Balance simplicity – power of user interface (parameter file)

```
VariableTracerVOF T
# Filter the volume fraction for smoother density transition for
# high density ratios: this helps the Poisson solver
VariableFiltered T1 T 1
PhysicalParams { alpha = 1./RHO(T1) }
# We need Kmax as well as K(mean) for adaptivity
VariableCurvature K T Kmax
SourceTension T 1 K
VariablePosition Y T y
VariablePosition Z T z
SourceViscosity 1e-2*MUR(T1)
# Initial deformed tube (only a quarter of it)
InitFraction {} T ({
     x -= 0.5; y += 0.5; z += 0.5;
     double r = RADIUS*(1. + EPSILON*cos(M_PI*x));
     return r*r - y*y - z*z;
})
```

- Integration with auxilliary tools (UNIX) : scripts, GfsView, post-processing etc...

Weak points

- Performance on a pure Cartesian mesh
- Barrier “expert user” – “beginner programmer” too high: the code is too complex
- Portability on “stupid” systems (and/or stupid system administrators?) i.e. non-POSIX “UNIX” systems, Microsoft Windows, microkernels (Blue Gene), GPUs, etc...
- Code complexity of the ‘auxilliary language’ (compilation of `GfsFunction`, C-object orientation etc...) (70% of code ?!)
- Accumulation of historic workarounds

Basilisk

- Principal objectives: Precision – Simplicity – Performance
  - Precision: identical to Gerris
  - Code simplicity:
    - Implementation: complexity comparable (or less than) that of a (simple) pure Cartesian code, minimal use of complex programming (object orientation etc...)
    - Algorithms: simplified compared to Gerris, elimination of historical workarounds
    - No barrier ‘expert user – beginner programmer”:
      the code is the user interface
    - Portability: only one strict dependency: ISO C99 compiler/library, no auxilliary librairies (glib etc...)
  - Performance: identical to that of a pure Cartesian code (i.e. $>\times 10$ Gerris), much better than Gerris ($>\times 4$) in adaptive mode
- Scientific objective: real estimation of performance gain of quad/octree adaptive methods compared to an optimised pure Cartesian code
- Basic simlplicity allows for more complex numerical schemes

Example : $a = \nabla^2 b$ with a 5-points operator

$$b = \sin(2\,\pi\,x)\,\cos(2\,\pi\,y)$$

Gerris parameter file (minimal)

```
1 0 GfsSimulation GfsBox GfsGEdge {} {
  Refine 7
  Init {} { B = sin(2.*M_PI*x)*cos(2.*M_PI*y) }
  VariableLaplacian A B
}
GfsBox {}
```

Corresponding Gerris code (simplified)

```c
...

typedef struct {
  GfsVariable * a, * b;
} LapData;

void laplacian (FttCell * cell, LapData * p)
{
  GfsGradient g;
  FttCellNeighbors neighbor;
  FttCellFace f;
  GfsGradient ng;

  g.a = g.b = 0.;
  f.cell = cell;
  ftt_cell_neighbors (cell, &neighbor);
  for (f.d = 0; f.d < FTT_NEIGHBORS; f.d++) {
    f.neighbor = neighbor.c[f.d];
    if (f.neighbor) {
      gfs_face_gradient (&f, &ng, p->b->i, -1);
      g.a += ng.a;
      g.b += ng.b;
    }
  }
  gdouble h = ftt_cell_size (cell);
  GFS_VALUE (cell, p->a) = (g.b + g.a*GFS_VALUE (cell, p->b))/(h*h);
}

/* initialisation of variables, mesh creation etc... */
...

LapData p = { a, b };
gfs_domain_traverse_leaves (domain, (FttCellTraverseFunc) laplacian, &p);

...
```

Basilisk code (complete)

```
int main()
{
  scalar a[], b[];

  init_grid(128);

  foreach()
    b[] = sin(2.*pi*x)*cos(2.*pi*y);
  boundary({b});

  foreach()
    a[] = (b[0,1] + b[1,0] + b[0,-1] + b[-1,0] - 4.*b[])/sq(delta);
}
```

Choice of grid implementation at compilation

- default is quadtree

  ```
  % qcc -Wall -O2 lap.c -o lap -lm
  ```

- Cartesian grid

  ```
  % qcc -grid=cartesian -Wall -O2 lap.c -o lap -lm
  ```

- Cartesian grid with OpenMP parallelism

  ```
  % qcc -grid=cartesian -fopenmp -Wall -O2 lap.c -o lap -lm
  ```

How does this work?

A new generic interface for discretisations on "generalised" Cartesian grids, using a minimal extension of C99

a) New types for scalar, vector and tensor fields: `scalar`, `vector`, `tensor`

b) Operations on local *stencils* (5×5 by default) :

```
scalar a[];
```

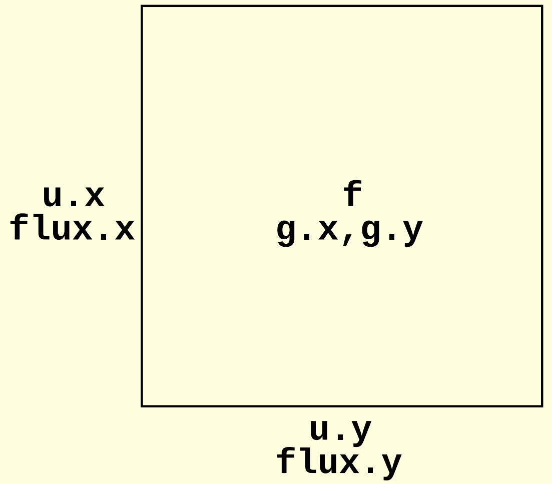
a[-2,2]	a[-1,2]	a[0,2]	a[1,2]	a[2,2]
a[-2,0]	a[-1,0]	a[0,0]	a[1,1]	a[2,1]
a[-2,0]	a[-1,0]	a[0,0]	a[1,0]	a[2,0]
a[-2,-1]	a[-1,-1]	a[0,-1]	a[1,-1]	a[2,-1]
a[-2,-2]	a[-1,-2]	a[0,-2]	a[1,-2]	a[2,-2]

`a[] = a[0,0]` (different from C : `a[0][0]`)

c) Iterators : `foreach()`, `foreach_dimension()`

The corresponding C code is generated automatically at compile time (no runtime overhead + optimisation)

A (slightly) more complex example: Bell–Colella–Glaz advection scheme (cf. *Gerris programming for dummies)*, 100+ lines of code in Gerris



```
void fluxes_upwind_bcg (const scalar f, const face vector u,
                        face vector flux,
                        double dt)
{
  vector g[];
  gradients ({f}, {g});
  foreach()
    foreach_dimension() {
      double un = dt*u.x[]/delta, s = sign(un);
      int i = -(s + 1.)/2.;
      double f2 = f[i,0] + s*min(1., 1. - s*un)*g.x[i,0]*delta/2.;
      double vn = u.y[i,0] + u.y[i,1];
      double fyy = vn < 0. ? f[i,1] - f[i,0] : f[i,0] - f[i,-1];
      f2 -= dt*vn*fyy/(4.*delta);
      flux.x[] = f2*u.x[];
    }
}
```

`foreach_dimension()` does an automatic permutation of indices

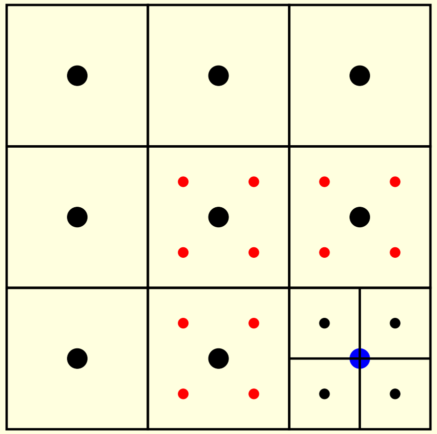
```
foreach_dimension() { double f2 = f[i,0] + s*min(1., 1. - s*un)*g.x[i,0]*delta/2.; }
```

is identical to

```
{ double f2 = f[i,0] + s*min(1., 1. - s*un)*g.x[i,0]*delta/2.; }
{ double f2 = f[0,i] + s*min(1., 1. - s*un)*g.y[0,i]*delta/2.; }
```

How does this work on a quadtree?

- The quadtree implementation guarantees stencil consistency independently of neighbourhood resolution
- The necessary synchronisation is done when applying boundary conditions (`boundary()`)
- No need to modify the purely Cartesian code (some constraints in its formulation)



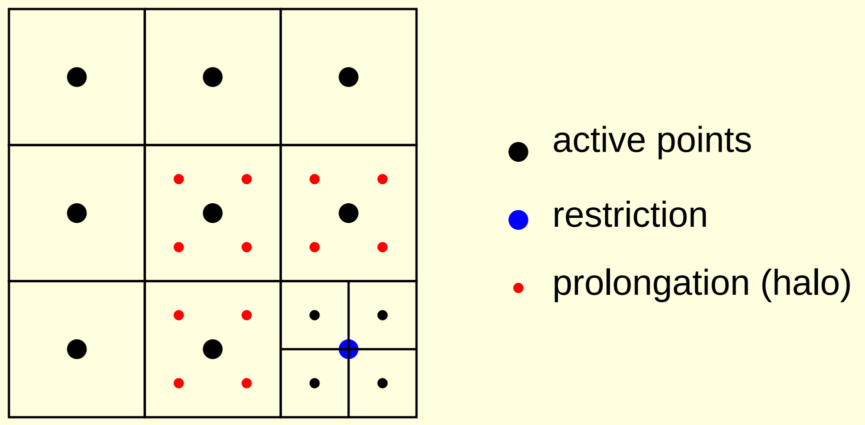
● active points

● restriction

• prolongation (halo)

- Restriction

```
void restriction (scalar v)
{
  v[] = (fine(v,0,0) + fine(v,1,0) + fine(v,0,1) + fine(v,1,1))/4.;
}
```



- Prolongation

```
void prolongation (scalar v)
{
  /* bilinear interpolation from parent */
  v[] = (9.*coarse(v,0,0) +
         3.*(coarse(v,child.x,0) + coarse(v,0,child.y)) +
         coarse(v,child.x,child.y))/16.;
}
```

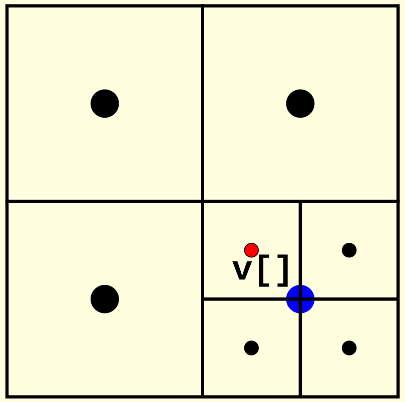
- Note that both restriction and prolongation are simple Cartesian operators

- Boundary conditions

```
void boundary (scalar v, int level)
{
  for (int l = level - 1; l <= 0; l--)
    foreach_level (l)
      restriction (v);
  for (int l = 0; l <= level; l++)
    foreach_halo_level (l)
      prolongation (v);
}
```

Error estimation and wavelets

```
e[] = prolongation(v) - v[] =
  (9.*coarse(v,0,0) +
   3.*(coarse(v,-1,0) + coarse(v,0,1)) +
   coarse(v,-1,1))/16. - v[];
```



● active points

● restriction

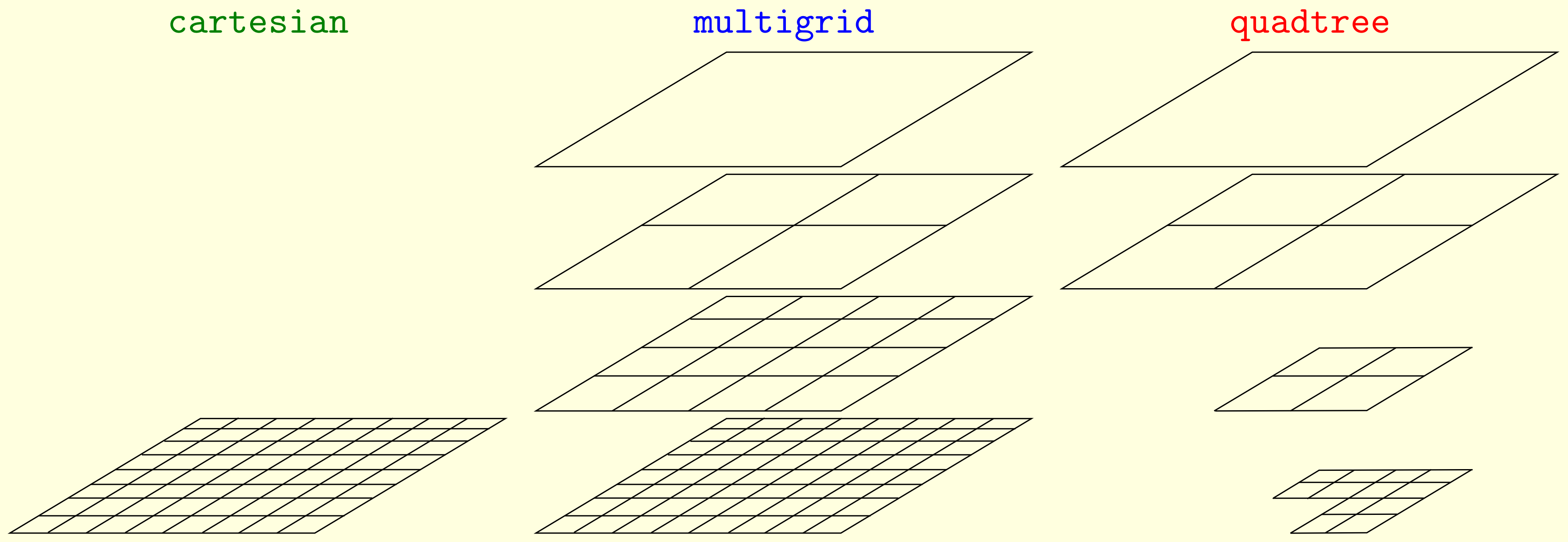
● prolongation

Wavelet decomposition

$$\texttt{v[]} \;=\; \sum_{i=0}^{n} \Delta_i^2\,\texttt{w[]}_i \qquad \text{with } \Delta_i^2 = 4^{-i} \text{ the ``scale factor'' and } \texttt{w[]}_i \text{ the wavelet coefficient}$$

$$\texttt{e[]} \;=\; \Delta_n^2\,\texttt{w[]}_n$$

A hierarchy of discretisations



Opérateurs

foreach(), foreach_face()	coarse(), fine()	foreach_cell()
foreach_dimension()	level, depth(), child.x	foreach_cell_post()
scalar, vector, tensor	foreach_level()	cell, parent
v[i,j]		leaf, active, halo
x, y, delta		
init_grid(), free_grid()		
locate()		

Example: generic multigrid solver

Works on `multigrid` and `quadtree` discretisations

```
void mg_cycle (scalar a, scalar res, scalar dp,
               void (* relax)    (scalar dp, scalar res, int depth),
               int nrelax, int minlevel)
{
  /* restrict residual */
  for (int l = depth() - 1; l <= minlevel; l--)
    foreach_level (l)
      restriction (res);
  /* multigrid traversal */
  for (int l = minlevel; l <= depth(); l++) {
    if (l == minlevel)
      /* initial guess on coarsest level */
      foreach_level (l)
        dp[] = 0.;
    else
      /* prolongation from coarser level */
      foreach_level (l)
        prolongation (dp);
    boundary (dp, l);
    /* relaxation */
    for (int i = 0; i < nrelax; i++) {
      relax (dp, res, l);
      boundary (dp, l);
    }
  }
  /* correction */
  foreach()
    a[] += dp[];
}
```

Application to Poisson equation $\nabla^2 a = b$

The relaxation operator is simply

```
void relax (scalar a, scalar b, int l)
{
  foreach_level (l)
    a[] = (a[1,0] + a[-1,0] + a[0,1] + a[0,-1] - sq(Delta)*b[])/4.;
}
```

The corresponding residual is

```
void residual (scalar a, scalar b, scalar res)
{
  foreach()
    res[] = b[] - (a[1,0] + a[-1,0] + a[0,1] + a[0,-1]
                   - 4.*a[])/sq(Delta);
}
```

Application to granular flows: a yield-stress rheology

- Incompressible variable-density Navier–Stokes
- Two immiscible phases: air and “sand” (VOF interface)
- Adaptive using “VOF wavelet” error estimator
- $\mu(I)$ sand rheology (cf. Pierre-Yves’ talk)
- Use the generic multigrid solver to solve both the (vector) Helmholtz problem

$$\rho\,\boldsymbol{u}_{n+1} - \Delta t\,\nabla\cdot[\eta_n\,(\nabla\boldsymbol{u}_{n+1} + \nabla^T\boldsymbol{u}_{n+1})] = \rho\,\boldsymbol{u}_n$$

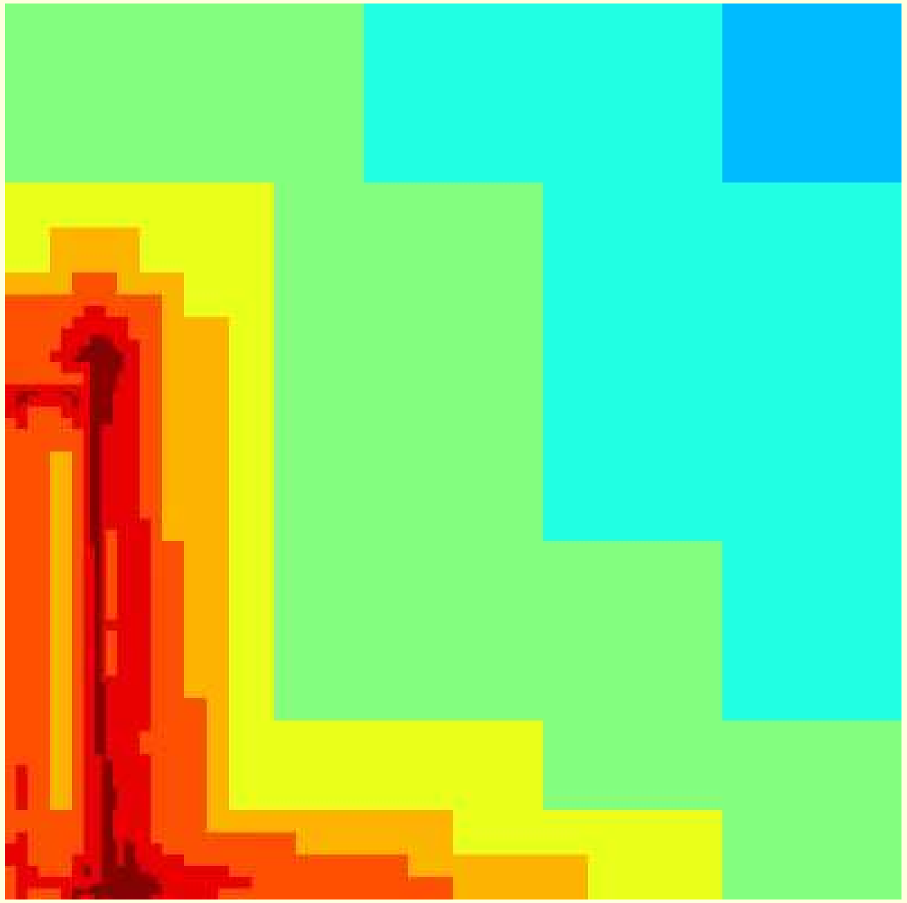
and the (scalar) Poisson problem

$$\nabla\cdot(1/\rho\,\nabla p_{n+1}) = \nabla\cdot\boldsymbol{u}_\star$$

The viscous stress residual is then

```
void residual_viscosity (vector u, face vector eta, scalar alpha,
                         vector res, double dt)
{
  foreach_dimension() {
    /* viscous fluxes */
    face vector Dx[];
    foreach_face(x)
      Dx.x[] = 2.*eta.x[]*(u.x[] -u.x[-1,0])/Delta;
    foreach_face(y)
      Dx.y[] = eta.y[]*(u.x[] - u.x[0,-1] +
                        (u.y[1,-1] + u.y[1,0])/4. -
                        (u.y[-1,-1] + u.y[-1,0])/4.)/Delta;
    /* divergence of the fluxes */
    foreach()
      res[] = r.x[] - u.x[] + dt*alpha[]/Delta*
        (Dx.x[1,0] = Dx.x[] + Dx.y[0,1] - Dx.y[]);
  }
}
```

Collapse of a column of grains: adaptive mesh



Runtime (seconds)	Basilisk	Gerris
Cartesian	1860	20200
Adaptive	26	110
Speed (points.step/sec)		
Cartesian	4.6×10^5	4×10^4
Adaptive	1.0×10^5	2×10^4
# of lines of source code	2300	53000

Literate programming

- Donald Knuth (1980s): the code is a proper document written by a “literate” programmer
- Open science: scientific papers should contain “actual scholarship” not “advertisement for scholarship”
- See http://basilisk.fr

Conclusions

- Minimal extension to C allows an easy implementaion of a wide range of algorithms on Cartesian, multigrid and quadtree grids
- Performances (in Cartesian mode) are identical to that of a "pure" Cartesian grid implementation
- Code simplicity allows to get rid of an 'auxilliary language' as user interface $\Rightarrow$ no barrier user / programmer
- Code simplicity permits the development of original algorithms
- The futur of Gerris?

  What already works

  - (Generalised) adaptive multigrid solver
  - Incompressible variable-density Navier–Stokes solver
  - OpenMP shared-memory parallelism
  - VOF adaptive etc...
  - Height-functions, surface tension
  - Metric: generic, spherical, axisymmetric

Work in progress

- Metric for viscosity/diffusion
- Adaptivity for height-functions/surface tension
- MPI parallelism
- 3D

What Gerris can do and Basilisk cannot (yet)

- Arbitrary domains
- Periodic boundaries
- Solids

What Basilisk can do and Gerris cannot

- Generic systems of conservation laws (e.g. compressible gases)
- Serre–Green–Naghdi equations
- Generic coupled systems of equations (reaction-diffusion etc...)
- 1D grids
- High-order schemes
- etc...

`basilisk.fr`/Tutorial