Notus first steps (0.4.0)

Stéphane Glockner, Antoine Lemoine, Mathieu Coquerelle,
and students, PhDs, Postdocs

Institut de Mécanique et d'Ingénierie de Bordeaux

Université de Bordeaux, Bordeaux-INP, CNRS UMR 52 95

`https://notus-cfd.org`

June 5th 2020

Contents

Notus first steps and its ecosystem

Notus code purposes

Open-source project started from scratch in 2015 (CeCILL Licence)

- Modelisation and simulation of **incompressible fluid flows**, multiphysics
- 2D/3D Finite Volume methods on staggered grids, **Massively parallel**

Intended users

- **Mechanical community**: easy to use and adapt, proven state-of-the-art numerical methods, towards numerical experiments
- **Mathematical community**: develop new numerical schemes, fast and efficient framework for comparative and qualitative tests
- Researchers, students, industrials

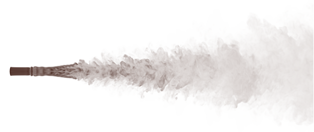
Some key points

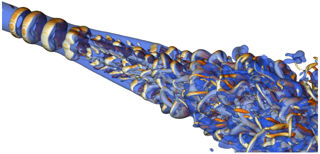
- Take advantage of synergies between Research / Teaching / Industry / HPC
- A clear and complete development environment
- **Mask parallelism** complexities for easy programming
- **Porting** on GENCI, PRACE, mesocentres
- A thoroughly **validated and documented code**, **non-regression** approach

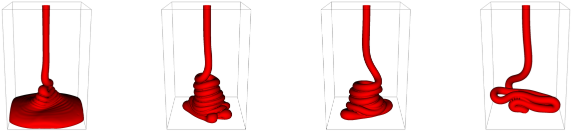
What is not Notus

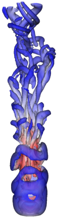
- A concurrent of, a commercial tool, a click button code

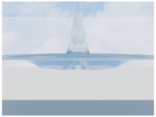
Notus - some examples



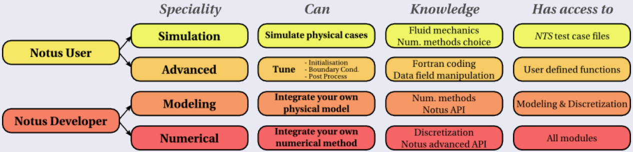








Several user types



Notus first step: focus on “Notus user”, Simulation & Advanced

Contents

Development environment

Development framework

- **Fortran 2008**
  - Allocatable arrays, structured and derived type
  - Module-oriented programming (private or public internal subprograms)
  - Optional arguments & intent attribute
  - Generic subroutine
  - Preprocessor
  - Interoperability with C (binding)
- **Hybrid MPI/OpenMP** parallel coding libraries
- **Git** distributed version control system
- **CMake** cross-platform build system
- **Doxygen** documentation generator from source code
- **Linux only!**
- **Web sites**

  https://notus-cfd.org

  https://doc.notus-cfd.org

  https://git.notus-cfd.org

Compilers and MPI libraries

- GNU compilers ($>$ 7.3) and Open MPI (2.10)
- Intel compilers ($>$ 18) and Intel MPI

Supercomputers

- Irene at TGCC, Occigen at CINES, Jean Zay at IDRIS
- Curta at MCIA
- Condor at I2M

Installation of Notus

Two steps

- Third part libraries
  - BLAS & LAPACK → system
  - Other dependencies: ADIOS (MXML), HYPRE, MUMPS (METIS, Scalapack), LIS, ADIOS2, HDF5, T3PIO
  - Be sure of the version installed → Git repository with tarballs
  - https://git.notus-cfd.org/notus/notus_third_party/
- Notus code
  - https://git.notus-cfd.org/notus/notus

1 - Get and build third part libraries

Clone third part lib repository

```
$ git clone https://git.notus-cfd.org/notus/notus_third_party.git notus_third_part
```

Build libraries

```
Help:
$ ./build_notus_third_party_lib.sh -h

Compilation and installation on Ubuntu 18.04:
$ ./build_notus_third_party_lib.sh -m --with-MPI-include /usr/include/mpi

Compilation and installation on CINES Occigen supercomputer:
$ ./build_notus_third_party_lib.sh -m --use-mkl --cc icc --cxx icpc --fc ifort
--mpicxx mpiicpc
```

→ Readme page: https://git.notus-cfd.org/notus/notus_third_party

Installation of Notus

2 - Get Notus

```
$ git clone https://git.notus-cfd.org/notus/notus.git notus
```

or, if you have a git account:

```
$ git clone git@git.notus-cfd.org:user/notus.git notus
```

```
$ cd notus
$ git remote add official git@git.notus-cfd.org:notus/notus.git
$ git remote update
```

→ to create a gitlab account: https://doc.notus-cfd.org/d3/d64/install_getnotus.html

Installation of Notus

Build Notus with Cmake (Open-source software for managing build process)

- Compiler independant
- Supports directory hierarchies
- Automatically generates file dependencies, supports library dependencies
- Builds a directory tree outside the source tree

CMake and Notus

- `CMakeLists.txt`
  - several development environnement: *GNU, Intel*
  - find third party libraries
  - **Release or debug (default) builds**
  - **→ always debug for development; release for production**
- `build_notus.sh` script whatever the target architecture:

```
To build on a workstation with GCC compilers and OpenMPI:
$ ./build_notus.sh --linux

To build with an Intel compilers suite:
$ ./build_notus.sh --intel

To build on 8 threads:
$ ./build_notus.sh -j 8 --linux
```

Installation of Notus

```
To Build on Curta supercomputer environment:
$ ./build_notus.sh -j 4 --curta

To clean build directory before building Notus:
$ ./build_notus.sh -cj 4 --linux

To use MUMPS solver library:
$ ./build_notus.sh -mj 4 --linux

To build with optimization compiler options (release mode):
$ ./build_notus.sh -rmj 4 --linux

To build with OpenMP library:
$ ./build_notus.sh -ormj 4 --linux


To get help:
$ ./build_notus.sh -h


→ More details: https://doc.notus-cfd.org/d7/de7/install_build.html
```

Contents

Run Notus

Basic way

- Parallel execution → `mpirun` command
  `$ mpirun -np 8 notus test_cases/validation/free_convection/square.nts`
- Test case data base in `test_case` directory
  `verification: laplacian, navier, phase_advection, phase_change, etc.`
  `validation: laminar_flow, free_convection, multiphase, etc.`
- **Use your own directory** to store your `.nts` files
- Complete list of command line options:
  `$ ./notus -h`

Advanced ways

- `notus.py`, script with 2 running modes
  - run mode: run a test case with parameter changes, run using a batch system, specify mpirun command, etc.
  - non-regresion mode: run test cases among the existing verification and validation test cases as well as various tests
  - complete list of command line options:
    ```
    $ ./notus.py -h
    $ ./notus.py non-regression -h
    $ ./notus.py run -h
    ```
- `notus_grid_convergence` to run a grid convergence study

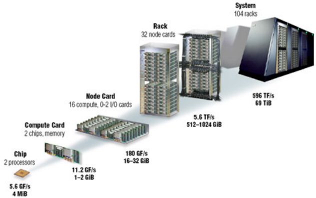
→ More details: https://doc.notus-cfd.org/d9/dfe/run_notus.html

Job submission on a supercomputer

- Share ressources managed thanks to a job scheduler and workload management (Slurm, PBSpro, etc.)
- Command are system dependant → see supercomputing center documentation (CINES, IDRIS, TGCC, MCIA, etc.)
- You have to submit your job (and wait) → `tools/submission_scripts`
- Limit amount of processors and CPU time
- Job dependancy
- For large data sets: remote visualization offered by supercomputng center

Choose the amount of processors you need

3D: 100 000 cells / core
2D: 10 000 cells per core
Fill nodes. Number of nodes as a power of 2.



LLNL BlueGene/L technology

GENCI TGCC Joliot Curie Supercomputer

Notus console output

```
$ mpirun -np 8 notus test_cases/validation/free_convection/square_cavity.nts

Notus - build:  release
commit:  08a8cf8
branch:  ibd-anew
Compiled by ifort
on Tue Feb 13 09:19:09 CET 2018

Initialization
Grid information
Number of ghost cells:  02
Partitioning:  0004 x 0002 x 0001 = 000000008
Global size:  0032 x 0032 x 0001 = 000000001024

Momentum stencil type:  1_STAR
Pressure stencil type:  1_STAR
Energy stencil type:  1_STAR
Write grids and fields into 'test_cases/validation/free_convection/output/square_cavity_000000.bp'

Time iteration n°1 time 0.5000E+00

Momentum solver:  iterations and residual:  34 0.5108E-15
Pressure solver:  iterations and residual:  100 0.2804E-13
Divergence (predicted & corrected):  0.2920E+02 0.8290E-11
Energy solver:  residual:  0.8817E-14
Nusselt number, left boundary:  1.627072605124241E+001
Nusselt number, right boundary:  1.627072605341143E+001
Mean velocity magnitude:  1.748763516276342E-001
Stationarity temperature:  error_linf:  4.3704802876646909E-001
Stationarity_velocity_u:  error_linf:  5.2485424141074921E-001
Stationarity_velocity_v:  error_linf:  1.3221265398959479E+000
Stationarity_velocity error_linf:  1.32212653989595
Divergence (Linf & L2 norms):  2.6182E-09 8.2898E-12

Time iteration n°2 time 1.0000E+00
...
```

Notus console output

```
...

Time iteration n°287 time 0.1435E+03

Momentum solver:  iterations and residual:  21 0.8742E-15
Pressure solver:  iterations and residual:  12 0.4509E-14
Divergence (predicted & corrected):  0.8112E-12 0.1013E-16
Energy solver:  residual:  0.2073E-15
Nusselt number, left boundary:  1.049093321926628E+001
Nusselt number, right boundary:  1.049093321927709E+001
Mean velocity magnitude:  3.792175505471097E-003
Stationarity temperature:  error_linf:  9.4928509497549385E-012
Stationarity_velocity_u:  error_linf:  7.5430200280335313E-013
Stationarity_velocity_v:  error_linf:  3.9763027939732076E-013
Stationarity_velocity error_linf:  7.543020028033531E-013
Divergence (Linf & L2 norms):  4.9960E-16 1.0133E-17

Satisfied convergence

Residual stationarity temperature (L2 norm):  9.492850949754938E-12
Residual stationarity velocity (L2 norm):  7.543020028033531E-13

Write grids and fields into 'test_cases/validation/free_convection/output/square_cavity_000287.bp'
```

Contents

User Interface: `.nts` file

Concept

- ASCII `.nts` files
- Self-explanatory keywords, precise grammar
- Efficient parser that supports:
  - variable declaration
  - formula
  - 'include'
  - if condition and loop
- **Associated documentation** → `test_cases/doc` directory

Organisation

- Physical fluid properties data base: `std/physical_properties.nts` file
- One `.nts` file per test case, block structure:
  - `include and variable declarations`
  - `system{}`
  - `domain{}`
  - `mesh{}`
  - `modeling{}`
  - `numerical_methods{}`
  - `post_processing{}`

User Interface: `.nts` file example

```
include std "physical_properties.nts";
system { measure_cpu_time;}
domain {
  spatial_dimension 2;
  corner_1_coordinates (0.0, 0.0);
  corner_2_coordinates (1.0, 2.0);
}
grid {
  grid_type regular;
  number_of_cells (32, 32);
}
modeling {
  fluids {fluid "water";}
  equations {
    energy {
      boundary_condition{
        left dirichlet 0.0;
        right dirichlet 1.0;
        top neumann 0.0;
        bottom neumann 0.0;
      }
          source_term {constant -2.0;}
          disable_advection_term;
          disable_temporal_term;
    }
  }
}
numerical_parameters {
  time_iterations 1;
  energy {
    solver mumps_metis;
  }
}
post_processing {
  output_library adios;
  output_frequency 1;
  output_fields temperature;
}
```

User Interface: notus language

Variables declaration and operations

- Wherever in the file
- Export to Fortran

```
string s = "Notus";
integer i = 1;
double a = 10.0;
boolean l = true;

a = 3.0d2;
a = 2.0e1;
a = b/c + c + sqrt(a) + cos(b) + pow(b,3);
s = "I" + " love " + "Notus";

integer h2g2 = 42;
export h2g2;
```

Automatic change at execution

- Useful for non-regression mode, parametric study
- Add `no_redefine`

```
integer no_redefine scale = 2;
```

- → `mpirun -np 2 notus -D integer:scale=1 test.nts`

User Interface: system block

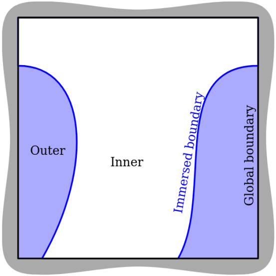
```
system {

   # [OPTIONAL] Overwrite default output directory (default: "output")
   output_directory STRING_EXPRESSION;


   # [OPTIONAL] Checkpoint metric (default: cpu_time)
   checkpoint_metric time_iteration | cpu_time;
   # [OPTIONAL] Frequency of the checkpoint (time iteration or second; default: 86000)
   checkpoint_frequency INTEGER_EXPRESSION;
   # [OPTIONAL] Restart with given file (i.e.: "output/checkpoint/poiseuille_2D_1.bp")
   restart PATH;


   # [OPTIONAL] Measure CPU time in several parts of the code
   measure_cpu_time;
   # [OPTIONAL] Measure CPU time of each time iteration only
   measure_time_iteration_cpu_time;

}
```

Checkpoint / restart

- Restart a simulation at computer precision after:
  $\rightarrow$ the end of CPU time limitated job on a supercomputer
  $\rightarrow$ a system crash
- Alternative writing in file sets 1 & 2

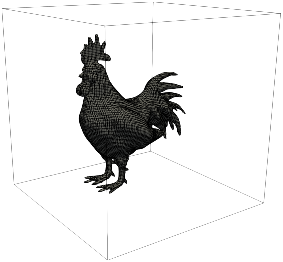
User Interface: domain block

```
domain {
   spatial_dimension 2; # or 3

   # The coordinates of 2 opposite corners of the physical domain
   corner_1_coordinates DOUBLE_ARRAY;
   corner_2_coordinates DOUBLE_ARRAY;

   # [OPTIONAL] Domain periodicity
   periodicity_x;
   periodicity_y;
   periodicity_z;

   # [OPTIONAL] Define a subdomain
   subdomain STRING_EXPRESSION {
      SHAPE # See shapes.nts
      #  - use CSG (Constructive Solid Geometry): union, intersection, and difference
      #  - manage transformations: translation, rotation, scale, and inverse
      #  - Many shapes are supported: sphere, rectangular cuboid, surface meshes, etc.
   }
}
```

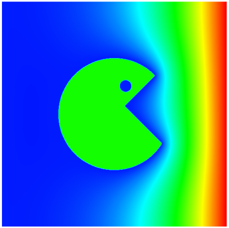


User Interface: `shape` block

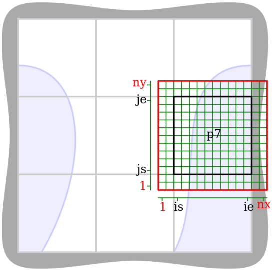
```
circle {
   center DOUBLE_ARRAY;
   radius DOUBLE_EXPRESSION;
   TRANSFORMATION # [OPTIONAL]
}
cuboid {
   corner_1_coordinates DOUBLE_ARRAY;
   corner_2_coordinates DOUBLE_ARRAY;
   TRANSFORMATION # [OPTIONAL]
}
surface_mesh {
   # OBJ Wavefront is the only supported format (yet)
   file PATH;
   TRANSFORMATION # [OPTIONAL]
}
TRANSFORMATION ::= invert
| translate DOUBLE_ARRAY
| scale     DOUBLE_EXPRESSION
| rotate    DOUBLE_EXPRESSION # 2D only
| rotate    DOUBLE_ARRAY, DOUBLE_EXPRESSION # 3D only
```

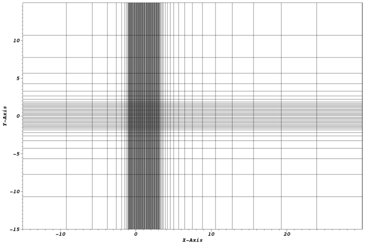


```
# Pacman
{
   difference {
      # Pac-Man's body
      circle {radius 0.25; center (0,0);}
      rectangle { # Mouth
         corner_1_coordinates (-0.1,-0.1);
         corner_2_coordinates (0.1,0.1);
         rotate tau/8.0;
         scale (1.5, 1.0);
         translate (sqrt(0.05), 0);
      }
      # Pac-Man's eye
      circle {radius 0.025; center (0.05, 0.125);}
   }
}
```



```
#one block (non-)uniform grid
grid {
   number_of_cells (32, 32);
   grid_type regular; #regular, chebyshev, exponential
   number_of_ghost_cells INTEGER_EXPRESSION;
}

#composite grid
grid {
   grid_type composite; #Generate a grid by parts.
   grid_x {
      grid_type regular;

      grid_type exponential;
      expansion_ratio DOUBLE_EXPRESSION; # Last step/first step
      first_step DOUBLE_EXPRESSION;      # Impose first step
      last_step  DOUBLE_EXPRESSION;      # Impose last step

      next_bound DOUBLE_EXPRESSION;
      length DOUBLE_EXPRESSION;

      number_of_cells INTEGER_EXPRESSION;
   }
   grid_x {
   ...
   }
   grid_y {
   ...
   }
   ...

   number_of_ghost_cells INTEGER_EXPRESSION;
 }
```





User Interface: `modeling` block

```
modeling {

   fluids {
      # Already defined fluid in std/physical_properties.nts
      fluid "air";

      # Definition of new FLUID_PROPERTIES
      fluid STRING_EXPRESSION {
         density DENSITY_TYPE DOUBLE_EXPRESSION;

         # DENSITY_TYPE can be either 'constant' or 'linear_temperature'
         density constant              DOUBLE_EXPRESSION; # Constant value
         density linear_temperature    DOUBLE_EXPRESSION; # Boussinesq

         viscosity                     DOUBLE_EXPRESSION;
         conductivity                  DOUBLE_EXPRESSION;
         specific_heat                 DOUBLE_EXPRESSION;
         thermal_expansion_coefficient DOUBLE_EXPRESSION;
         reference_temperature         DOUBLE_EXPRESSION;
      }
   }


   species {
      species "species_1" {
         reference_concentration 1.0;
         fluid "air" {
            diffusion_coefficient 2.0;
            solutal_expansion_coefficient 3.0;
         }
         fluid "water" {
            diffusion_coefficient 4.0;
            solutal_expansion_coefficient 5.0;
         }
      }
   }
```

User Interface: modeling block

```
equations {

   navier_stokes {

      boundary_condition {
         # See boundary_conditions.nts
      }
      # [OPTIONAL]
      immersed_boundary_condition {
         wall;
      }

      # [OPTIONAL]
      initial_condition {
         VECTOR_INITIALIZER # See initializer.nts
      }
      pressure_initial_condition {
         SCALAR_INITIALIZER # See initializer.nts
      }

      # [OPTIONAL]
      gravity_term (0, -9.81);
      source_term {
          VECTOR_INITIALIZER # See initializer.nts
      }
      linear_term {
         VECTOR_INITIALIZER # See initializer.nts
      }
      grad_div_term;
      brinkman_term;
      capilarity_term {
         surface_tension DOUBLE_EXPRESSION;
      }
   }
```

User Interface: `initializer` block

```
# Initialize a scalar field with 1.0 everywhere except in a circle
{
   # Initialize at 1.0 everywhere
   constant 1.0;

   # Initialize the scalar field x(1-x) + y(1-y) inside a circle of radius 0.5 centered at (0,0)
   shaped_instructions {
      shape {
         circle {radius 0.5; center (0.0, 0.0);}
      }
      instructions {
         @return @x*(1.0 - @x) + @y*(1.0 - @y);
      }
   }
}
# The above scalar initializer can be written with instructions only
# Instructions are the slowest initializer. For better performances, prefer the
# use of 'constant' or 'shaped_instructions' to minimize the computational cost.
{
   instructions {
      @if (@x*@x + @y*@y < 0.5*0.5) {
         @return @x*(1.0 - @x) + @y*(1.0 - @y);
      } @else {
         @return 1.0;
      }
   }
}

# Initialize a vector field with (0.0, 0.0) everywhere except in a unit square
{
   # Initialize the vector field with (0.0, 0.0) everywhere
   constant (0.0, 0.0);

   # Initialize the vector field with (1.0, 1.0) in a unit square centered at the origin
   shape (1.0, 1.0) {
      rectangle {corner_1_coordinates (-0.5, -0.5); corner_2_coordinates (0.5, 0.5);}
   }
}
```

```
boundary_condition {
   left   BOUNDARY_CONDITITION
   right  BOUNDARY_CONDITITION
   bottom BOUNDARY_CONDITITION
   top    BOUNDARY_CONDITITION
   back   BOUNDARY_CONDITITION
   front  BOUNDARY_CONDITITION
}
BOUNDARY_CONDITITION:
wall                       [ { SHAPE_INITIALIZER } ]
neumann                    [ { SHAPE_INITIALIZER } ]
slip                       [ { SHAPE_INITIALIZER } ]
inlet  DOUBLE_ARRAY        | { VECTOR_INITIALIZER };
moving DOUBLE_EXPRESSION | { VECTOR_INITIALIZER }; # 2D
moving DOUBLE_ARRAY        | { VECTOR_INITIALIZER }; # 3D. Attention: it requires 2D (sic) arrays.
```

Example: parabolic flow on a part of the left boundary (and wall elsewhere except on the right boundary)

```
boundary_condition {
  left wall;
  left inlet{
    shaped_instructions {
       shape {
          line_segment {
             coordinates 1., 2.;
          }
       }
       instructions {
          @return (mean_velocity*6.0*(@y - 1.0)*(1.0 - (@y - 1.0))/(1.0*1.0), 0);
       }
    }
  }
  right neumann;
  top wall;
  bottom wall;
}
```

User Interface: modeling block

```
energy {
   boundary_condition {
      # See boundary_conditions.nts
   }
   # [OPTIONAL]
   immersed_boundary_condition {
      dirichlet DOUBLE_EXPRESSION | SCALAR_INITIALIZER;
      neumann DOUBLE_EXPRESSION | SCALAR_INITIALIZER;
      }

   # [OPTIONAL]
   initial_condition {
      SCALAR_INITIALIZER # See initializer.nts
   }

   # [OPTIONAL]
   disable_advection_term;
   disable_diffusion_term;

   phase_change {
      liquid_phase STRING_EXPRESSION; # Fluid name
      solid_phase  STRING_EXPRESSION; # Fluid name
      latent_heat DOUBLE_EXPRESSION;
      melting_temperature DOUBLE_EXPRESSION;
   }
   source_term {
      SCALAR_INITIALIZER # See initializer.nts
   }
   linear_term {
      SCALAR_INITIALIZER # See initializer.nts
   }
}

species_transport {
   # Select the species
   species "tc_species_1" {
   ...
   }
}
```

User Interface: `modeling` block

```
phase_advection {
   # Select the fluid to advect and associate initial and boundary conditions
   fluid STRING_EXPRESSION {
      boundary_condition {
         # See boundary_conditions.nts
      }

      # [OPTIONAL]
      initial_condition {
         SHAPE # See shapes.nts
      }
   }
}

turbulence {
   # Select an LES model
   les_model mixed_scale;
   ...
   # RANS model
   ...
}
```

User Interface: numerical_parameters block

```
numerical_parameters {

   time_iterations 1000; # Set the number of iteration. Cannot be used with 'final_time'.
   final_time 12.0;      # or set the final time (s). Cannot be used with 'time_iterations'.

   # Fixed time step
   time_step fixed DOUBLE_EXPRESSION;
   # or adaptative time step
   time_step adaptative {
      cfl_factor    DOUBLE_EXPRESSION;
      first_step    DOUBLE_EXPRESSION;
      min_step      DOUBLE_EXPRESSION;
      max_step      DOUBLE_EXPRESSION;
      max_increment DOUBLE_EXPRESSION;
      max_ratio     DOUBLE_EXPRESSION;
   }

   time_order_discretization INTEGER_EXPRESSION; # Can be 1 or 2, 1 by default

   # [OPTIONAL] Stop the simulation before the max time iteration number is all the selected test are satisfied.
   stop_tests {

      # [OPTIONAL] Stop if the elapsed time exceed 10.0 s
      elapsed_time 10.0;

      # [OPTIONAL] Stop the simulation if the incompressibility criterion is small enough
      incompressibility 1e-10;
      stationarity_temperature 1e-10; # [OPTIONAL]
      stationarity_velocity    1e-10; # [OPTIONAL]
      stationarity_species     1e-10; # [OPTIONAL]
   }

   # [OPTIONAL] Numerical parameters relative to materials and Immersed boundary parameters
   materials {
      sampling_level INTEGER_EXPRESSION;
   }
   immersed_boundary STRING_EXPRESSION {
     ...
   }
```

User Interface: numerical_parameters block

```
navier_stokes {
   time_step 1.0; # [OPTIONAL]replace main time step defined above

   # [OPTIONAL], Automatically chosen
   velocity_pressure goda; #  goda or timmermans

   # Select an advection implicit or explicit scheme (pick one)

   advection_scheme implicit o2_centered | o1_upwind | o2_upwind;

   advection_scheme explicit o1_upwind    | o2_upwind       | weno3_upwind |
                             weno5_upwind | weno3_upwind_fd | weno5_upwind_fd {
     temporal_scheme euler | ssp2_o2 | nssp2_o2 | nssp3_o2 | nssp5_o3;
     # [OPTIONS]
     directional_splitting true | false;
     flux_type godunov | lax_wendroff | force | flic;
     flux_limiter low_order | high_order | superbee | minmod | van_leer;
   }

   advection_scheme explicit lw_tvd_sb {
      splitting_method lie_trotter | strang;
   }

   solver_momentum # See basic_solvers.nts
   solver_pressure # See basic_solvers.nts

   immersed_boundary {
      # 1st order method
      method penalization
      # Second order methods
      method direct, linear;
      order 2, 1;
      # Value to assign at outer cells
      outer_value velocity (0.0, 0.0);
   }
}
```

User Interface: numerical_parameters block

```
energy {
   time_step 1.d0; # [OPTIONAL] replace main time step defined above

   # Select an advection implicit or explicit scheme (pick one)

   advection_scheme implicit o2_centered | o1_upwind | o2_upwind

   advection_scheme explicit o1_upwind    | o2_upwind       | weno3_upwind |
                             weno5_upwind | weno3_upwind_fd | weno5_upwind_fd {

     temporal_scheme euler | ssp2_o2 | nssp2_o2 | nssp3_o2 | nssp5_o3;
     # [OPTIONS]
     directional_splitting true | false;
     flux_type godunov | lax_wendroff | force | flic;
     flux_limiter low_order | high_order | superbee | minmod | van_leer;
   }

   advection_scheme explicit lw_tvd_sb {
      splitting_method lie_trotter | strang;
   }

   solver # See basic_solvers.nts

   immersed_boundary {
      method direct;
      order 2;
      outer_value 4.0;
   }
}
```

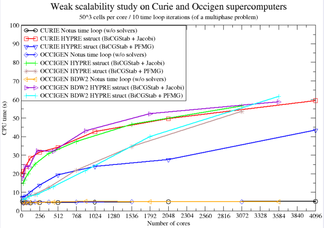
```
  phase_advection {
    time_step 1.0; # [OPTIONAL] replace main time step defined above

    # [OPTIONAL] sampling level to initialize VOF and MOF (default: 10)
    initial_condition_samples 50;

    vof_plic {
      smooth_volume_fraction INTEGER_EXPRESSION;
    }
    mof {
      use_analytic_reconstruction true;  # [OPTIONAL]
      use_filaments BOOLEAN_EXPRESSION;  # [OPTIONAL]
      max_filaments INTEGER_EXPRESSION;  # [OPTIONAL]
      smooth_volume_fraction 2;          # [OPTIONAL]
      ...
    }
    level_set {
      curvature_method normal_divergence; # [OPTIONAL]
      curvature_method closest_points;    # [OPTIONAL], implies compute_closest_point
      compute_closest_point;              # [OPTIONAL]

      time_order_discretization 0; # Euler
      time_order_discretization 1; # RK2 simple
      ...
      flux_type godunov;        # First order Godunov scheme (default)
      ...
      reinitialization;            # Default reinitialization (see below)
      ...
    }
  }
}
```

User Interface: `solver` block

```
# Available basic solver list:
#   - hypre_bicgstab or hypre_gmres
#   - mumps_metis
#   - lis_bi* or lis_*gmres
#   - notus_bicgstab

# Scalar equation
solver hypre_bicgstab {
   max_iteration 50;
   tolerance 1.0d-10;
   initial_preconditioner left_jacobi; # [Optional]

   preconditioner smg {   => more robust
   preconditioner pfmg {  => less robust
      max_iteration 1;
   }
}
# Momentum equation multiphase flow
solver hypre_bicgstab {
   max_iteration 50;
   tolerance 1.0d-10;
   initial_preconditioner left_jacobi; # [Optional]
}
# Momentum equation / scalar with stencil of size 2
solver hypre_parcsr_bicgstab {
   max_iteration 50;
   tolerance 1.0d-10;
   initial_preconditioner left_jacobi; # [Optional]
   preconditioner boomeramg {
      max_iteration 1;
      tolerance 1.0d-14;
      strong_threshold 0.25;
      coarsen_type 6;
      aggressive_coarsening_level 0;
      interpolation_type 0;
      post_interpolation_type 0;
      relaxation_type 6;
   }
}
```

HYPRE

- **Massively parallel solvers and preconditioners**
- **Geometric multigrid** for scalar equations
  Discretization stencil = 1
  Use PFMG (SMG slower but more robust)
- **Algebraic multigrid**
  More general, slower, less robust than SMG
  $\longrightarrow$ Navier-Stokes, scalar equation for stencil 1 or 2
  $\longrightarrow$ `_parcsr` hypre interface



User Interface: `solver` block

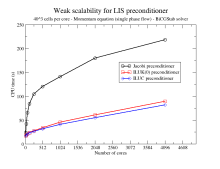
```
# MUMPS Metis
solver mumps_metis {}

# LIS solvers
solver lis_bicgstab{
    max_iteration 400;
    tolerance 1.0d-14;
    initial_preconditioner left_jacobi; # [Optional]

    preconditioner iluk{
        fill_level 1; # default 0
    }
    preconditioner iluc{
        drop_tolerance 0.001; # default 0.05
        rate 5.; # default 5
    }
    preconditioner ilut{
        drop_tolerance 0.001; # default 0.05
        rate 5.; # default 5
    }
}
```

MUMPS (direct solver)

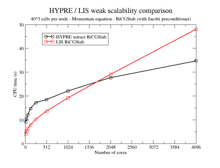
- $\rightarrow$ solution up to CPU precision
- Slower but competitive in 2D whatever the equation to solve
- Only small tests in 3D (high memory requirements)

LIS (iter. solvers and precond.)

- Useful in some cases:
  For momentum equation if Jacobi not enough



  May be quicker then Hypre at low number of processors



User Interface: post_processing block

```
post_processing {
   output_library adios; # none | adios | ensight | pixie | xdmf | adios2
   output_frequency 100;


   # Fluid properties
   output_fields conductivity, density, specific_heat, viscosity;

   # Navier-Stokes related variables
   output_fields velocity, divergence, navier_stokes_source_term, permeability, pressure, etc.

   # Multiphase variables
   output_fields volume_fraction;
   output_fields mof_phases;                                   # Requires mof
   output_fields interface_curvature,level_set_function;       # Requires level_set

   # Species variables
   output_fields species_concentration, species_diffusion_coefficient;

   # Energy variables
   output_fields energy_source_term, temperature;

   # Post-processing variables
   output_fields grid_volume, q_criterion, strain_rate_magnitude, vorticity;

   # Validation/verification variables
   output_fields error, reference_solution, reference_solution_face;

   # Diagnostic quantities computation
   diagnostic_quantities mean_kinetic_energy, mean_pressure, mean_temperature, nusselt_number, wall_shear_stress

   # [OPTIONAL] statistics (compute mean time fields, fluctuation, etc.)
   statistics {
      start_time 1.0;
      compute_time_averaged_fields velocity, pressure, temperature
      compute_fluctuation_fields   velocity, pressure, temperature, species_concentration;
      compute_rms_fields           velocity, pressure, temperature, species_concentration;
   }
```

User Interface: post_processing block

```
# add a set of probe points. Many 'probe_point' blocks can be defined.
probe_point {

   output_frequency INTEGER_EXPRESSION;

   # Define as many point as required (at least one)

   # Add a probe point using coordinates
   point DOUBLE_ARRAY;
   ...

   # Fields to output
   output_fields OUTPUT_FIELD [, OUTPUT_FIELD , [...]];
}

# add a probe line. Many 'probe_line' blocks can be defined.
probe_line {
   output_name STRING_EXPRESSION;       # [OPTIONAL]
   output_frequency INTEGER_EXPRESSION; # [OPTIONAL]

   # Definition of the line segment (only one line segment is accepted)

   # Define the line segment by the coordinates of its end points
   line_segment DOUBLE_ARRAY, DOUBLE_ARRAY;
   samples INTEGER_EXPRESSION; # Define the number of samples

   # Axis-aligned line segments

   # Define the line segment by the coordinates of the cell of its end points (must be axis-aligned)
   line_segment cell INTEGER_ARRAY, INTEGER_ARRAY;
   ...

   # Fields to output
   output_fields OUTPUT_FIELD [, OUTPUT_FIELD , [...]];
}
```

User Interface: post_processing block

Full documentation: test_cases/doc directory

```
advanced_solvers.nts
basic_solvers.nts
boundary_conditions.nts
domain_block.nts
grid_block.nts
initializer.nts
main.nts
modeling_block.nts
notus_language.nts
numerical_parameters_block.nts
post_processing_block.nts
shapes.nts
system_block.nts
```

Contents

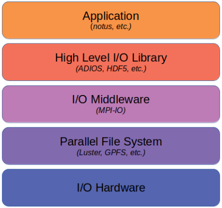
I/O - Visualisation

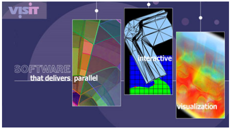
I/O: write on disk output data.

- Hundred of scientific file formats (open, closed, rely on external libraries, etc.)
- Save disk space → binary data files
- How to write efficiently on thousand of processors → parallel I/O.

Visualization: representation and analysis of the data

- 2D/3D field plot
  - VisIt: large-scale scientific visualization
  - ParaView: parallel scientific visualization
- 1D (2D) graph
  - Python's Matplotlib
  - Gnuplot: command-driven interactive 2d and 3d plotting program
  - Xmgrace
- Manipulating images
  - Gimp, ImageJ, ImageMagick
  - mencoder, ffmpeg





I/O - Visualisation: ADIOS & Notus

Domain is partitioned, data are distributed

→ How to write and plot data efficiently on thousands of processors?

Use of ADIOS library (Oak Ridge National Laboratory)

- Open-source
- Adaptable IO System
- Simple and flexible way to describe the data
- Masks IO parallelism
- Different methods: POSIX, MPI-IO, aggregation
- From 1 to 100 000 processors
- `.bp` files

ADIOS & Notus

- A list of data is created, printed at the end of the time loop
- Add a field anywhere in the code:

```
use mod_field_list
call add_field_to_list(print_list, enstrophy, 'enstrophy')
```

- ADIOS used also for checkpoint / restart

Visualisation of the results → VisIt (Lawrence Livermore National Laboratory)

- With ADIOS file format, VisIt is limited to 2 billion cells.

I/O - Visualisation: very large data sets

Pixie

- Based on HDF5 library (`.h5` files)
- Compatible with parallel VisIt (automatic parallel domain decomposition)
- Non-uniform rectilinear grids
- Notus Pixie output less efficient then ADIOS

XDMF

- Data are stored in HDF5 files (`.h5`), XML description file (`.xdmf` file)
- Non-uniform rectilinear grids
- Compatible with Paraview (parallel?) and VisIt (sequential)

ADIOS2

- Version 2 of ADIOS library, toward exascale computations
- Data are stored separatly, XML description file
- Compatible with Paraview (regular rectilinear mesh only)

Ensight

- Based on MPI-IO
- Data are stored separatly, `.case` description file
- Compatible with Visit and Paraview, less efficient then ADIOS or HDF5

Contents

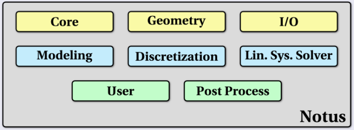
Development environment - Architecture

Project tree

`src`	Fortran source files
`std`	Standard database (fluid characteristics, mesh, object files)
`test_cases`	Test case description files
`tools`	Useful development and validation scripts
`doc`	Doxygen generated documentation

Source tree

- `src/lib` (notus library sources)



- `src/notus`
  - `notus.f90` (main program)
  - `ui/` (user interface routines)
- `src/doc`

Some development keys

Naming

- Hundreds of variables
  - self explanatory variable names (`velocity, pressure, temperature, ...`)
  - as few abbreviations as possible
- Prefix
  - module starts with `mod_`
  - scalar variable module starts with `variables_`
  - field array module starts with `fields_`
  - new derived types module starts with `type_`
  - new types starts with `t_`
    ex: `struct_face_field velocity%u %v`
  - scalar names associated to an equation suffixed (`navier_time_step`, etc.)
- Explicit routine name

  `solve_navier`

  `compute_mean_velocity`

  `add_div_diffusive_flux_to_matrix`
- → nearly "guessable" variables

  → Auto-documentation

  → **Use `git grep` to locate variables, routines, etc.**

Some development keys

Code formating

- tab = 3 characters
- line = 132 characters max

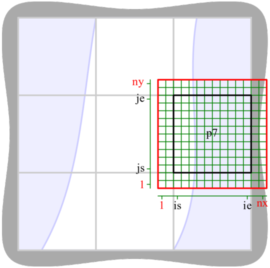
- **Automatic formatting before committing:** `formatcode.sh`

```
Usage:  formatCode.sh [OPTIONS]
-h print usage and exit
-p format only modified files
-f format only given files
-c COMMIT format only the given commit
```

Some development keys - Masking parallelism

Numerical domain and process ghost cells

- The global domain is partitioned into subdomains
- Addition of a few layers of cells surrounding the local domain: $nx \times ny \times nz$ cells



MPI generic routines to exchange data

- 2D/3D, whatever overlapping zone size
- Integer, double
- Cell array, or vector defined on staggered grid

```
call mpi_exchange(pressure)
call mpi_exchange(velocity)
```

- → Mandatory after any spatial derivative computations
- MPI Exchange + Fill boundary ghost nodes

```
call fill_ghost_nodes(scalar,
boundary_condition)
call fill_ghost_nodes(vector, is_vector,
boundary_condition)
```

Global reduction routines

- encapsulate MPI ones
- generic routines for min, max of local arrays, sum of scalars

OpenMP generic algebraic operation for 3-dimensional arrays and face-fields

x = a + b

```
call field_operation_add(a, b,
x)
```

a = a + b*c

```
call field_operation_add_mult(a,
b, c)
```

...

Some development keys - A set of user routines

Concept

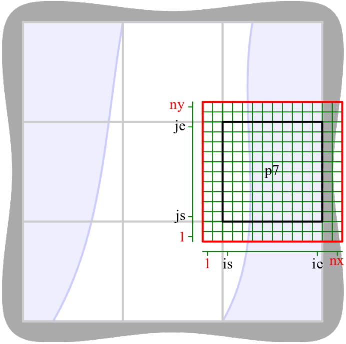
- → Avoid a user to known very well the code
- → User directory `src/lib/user`
- Void routine by default
- Uncomment, modify, compile
- Initial condition
- Boundary conditions
- Source terms
- Computation of physical properties
- Implicit discretization scheme (for scalar equations)

Example

```
do k=1,nz
    do j=1,ny
       energy_boundary_type%left(j,k)=cell_boundary_type_dirichlet
       temperature_boundary_value%left(j,k)=...
    enddo
enddo
```

Some development keys - useful modules

- `use variables_domain`
  → `spatial_dimension, etc.`
- `use variables_grid`
  → `nx, ny, nxu, nyv, is, ie, isu, ieu, etc.`
- `use variables_spatial_step`
  → `dx(nx), dx_u(nxu), etc.`
- `usr variables_time_discretization`
  → `time, global_time_step, time_iteration, etc.`



Contents

Documentation - Doxygen

For writing software reference documentation

- Documentation is written within the code
- Open-source, generates html, pdf, latex files

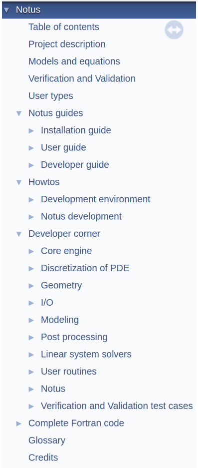
Doxygen and Notus

- https://doc.notus-cfd.org
- Upper level doc: installation, git, architecture, **howtos**, **best practises**, etc. (markdown format)
- One documentation group per src/lib subdirectories (physics, numerical_methods, io, etc.)

```
cat /src/lib/mesh/grid_generation/doc.f90
!> @defgroup grid_generation Grid Generation
!!  @ingroup mesh
!!  @brief Compute grid coordinates and spatial steps
```

- Documentation inside each Fortran files

```
cat /src/lib/mesh/grid_generation/create_regular_mesh.f90
!> Create a regular Cartesian mesh (constant step size per direction).
!!  The mesh is created in two steps:
!!  1.  Provide global face coordinates
!!  2.  Compute local variables (coordinates and space steps)
!!  The second step is automated in complete_mesh_structure
!!  Require the number of points per directions
!!  ingroup grid_generation
subroutine create_regular_mesh()
...
```



Contents

Verification and Validation V&V

Verification

- **proves that the continuous model is solved precisely by the discrete approach**
  - analyses the numerical solution of equations
  - quantifies and reduces of the numerical errors
  - computes spatial and temporal convergence orders
- $\rightarrow$ **mainly a mathematical and computing process, unlinked to physical problem**

Validation

- **analyses the capacity of a model to represent a physical phenomena**
  - compares numerical solution to experimental results
  - identifies and quantifies errors and uncertainties of continuous and discrete models, and experience

$\rightarrow$ Accumulation of evidence that the code works!

Verification

2 main steps

- no bug in the code or unconsistant solution
- quantify numerical errors
  - start from an exact (built) solution
  - compute errors, convergence order
  - compare the given order to the expected one

Error sources

- coding bug
- numerical stability condition not satisfied
- insufficiant spatial or temporal convergence
- iterative methods not converged
- rounding errors

Hypothesis: smoothed solution in the asymptotic convergence zone

N discrete solutions $f_k (1 \leq k \leq N)$

$$f_{h \to 0} = f_k + C h_k^p + \mathrm{O}(h_k^{p+1})$$

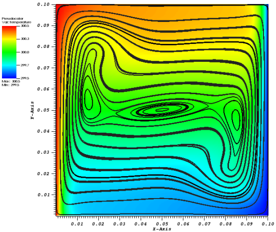
$$p_k = \frac{log(\frac{E_k}{E_{k-1}})}{log(\frac{h_k}{h_{k+1}})}$$

where $E_k = f_{exact} - f_k$

mesh	L_∞ error	Order	L_2 error	Order
10	2.53e-03	n/a	6.87e-04	n/a
20	6.49e-04	1.97	1.69e-04	2.02
40	1.63e-04	1.99	4.22e-05	2.00
80	4.08e-05	2.00	1.05e-05	2.00

Analyses the capacity of a model to represent a physical phenomena

- no exact solution
- post processing of physical parameter (velocity plot, Nusselt numbers, lift, drag, etc.)
- comparison with experience or other code
- quantify error and uncertainty
- 3 meshes → convergence order → Richardson extrapolation



Mesh	Nusselt nb.	Order	Velocity	Order
32	1.0490e+01	na	3.7921e-03	na
64	9.1842e+00	na	3.6811e-03	na
128	8.9013e+00	2.2070	3.6387e-03	1.3913
256	8.8424e+00	2.2635	3.6277e-03	1.9381
512	8.8292e+00	2.1622	3.6249e-03	1.9957
Ext.	8.8254e+00		3.6240e-03	
Réf.	8.8252e+00			

Notus V & V tools

1 - compute convergence order

Run the same case varying a parameter (mesh or time step)

- → *json* file

```
{"number_of_cells": [ 100, 25], "time_step": 0.5},
{"number_of_cells": [ 200, 50], "time_step": 0.25},
{"number_of_cells": [ 400, 100], "time_step": 0.125},
{"number_of_cells": [ 800, 200], "time_step": 0.0625}
```

- Python script: `./notus_grid_convergence -np 8 --doxygen test_case_name`
  - run (interactivly or submission) the test case with different meshes
  - collect the results of the chosen quantities
  - compute convergence order and extrapolated values
  - output to doxygen format

2 - non regression

- **list of V&V test cases files**
- quick or full validation
- run the test cases with bash script
- results in *txt* file: OK, NO, FAIL, etc.
- commit the results (one per architecture) to Git repository
- `notus.py` script

Notus V & V tools: create your own test case

- Work in another directory than validation or verification ones
- As much as possible, use formula inside the .nts file
- Integration into notus test case list:

  → https://doc.notus-cfd.org/db/da5/howto_add_test_case.html

Check Portability and Performances

Portability

- Associated to V & V process
- Numerical solutions should be **independant of**:
  - compiler editors, compiler versions, MPI libraries, etc.
  - computer architectures and processor numbers
- Notus portable on:
  - GNU + OpenMPI; Intel + MPT; Intel + IntelMPI; Intel + BullXMPI
  - Sequential and Parallel versions
  - $\rightarrow$ "Same" results betwwen 10^{-8} and 10^{-15})

Performances

- Compare measured scalability to the expected one
- Identify and measure relevant parts of the code
  - partitiong
  - initialization
  - time loop: equation preparation, solvers (external), I/O
- Lot of functionalities: **identify the relevant test cases**
- Determine optimal use of supercomputers (number of cells per core)

Notus, performance tools

Objectives

- Verify weak and strong scalability
- Verify I/O performance
- Ensure non regression of these performances
- On several supercomputers (from local to GENCI/PRACE)

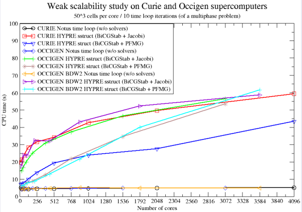
Scalability scripts

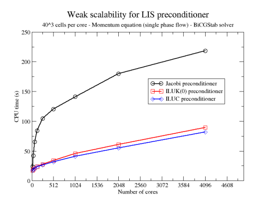
- Template directoy
  - notus template .nts file
  - submission template file (depending of the workload manager)
- Submission bash script
  - `./submit_jobs.sh -t weak -a 9 -c 40 -m 16 -s template_sub_curie -q ccc_msub`
  - `./submit_jobs.sh -t strong -i 3 -a 9 -c 512 -m 16 -s template_sub_curie -q ccc_msub`
  - `./submit_jobs.sh -t strong_node -c 100 -m 16 -s template_sub_curie -q ccc_msub`
  - → copy template directory
  - → adapt template files
  - → submit jobs
- Concatenation bash script
  - `./concatenate_cpu_times.sh -t weak -a 9 -c 40 -m 16`

```
128     0.26000E+01    0.86140E+00    0.94443E+00    0.79417E+00
256     0.29297E+01    0.10660E+01    0.10462E+01    0.81751E+00
512     0.30754E+01    0.11369E+01    0.11025E+01    0.83590E+00
1024    0.38859E+01    0.16025E+01    0.13959E+01    0.88751E+00
2048    0.43207E+01    0.18807E+01    0.15359E+01    0.90404E+00
4096    0.47281E+01    0.22302E+01    0.16268E+01    0.87108E+00
8192    0.65902E+01    0.32613E+01    0.23815E+01    0.94744E+00
```

Weak scalability on Curie and Occigen supercomputers

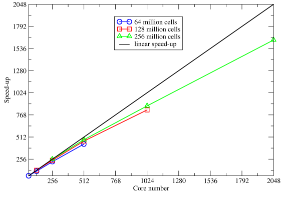
→ 50^3 cells per core, number of core increases, constant CPU time expected



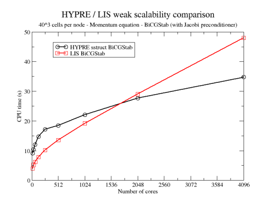


Strong scalability

→ constant number of global cells, number of core increases, linear speed-up expected



HYPRE / LIS comparison: BiCGStab + Jacobi



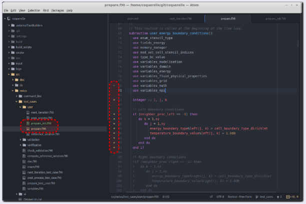
Contents

Developement tools - Editing the source code

Atom Integrated Development Environment

Cross-platform editing, File system browser, Multiple panes, ...

https://doc.notus-cfd.org/dd/dd7/howto_atom.html

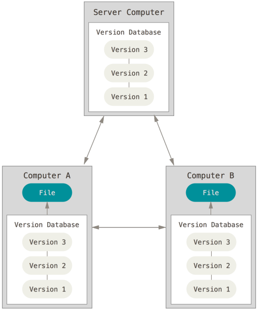


Light and efficient text editors

- From workstation to supercomputer, remote access
- vim → tools/vim_syntax
  https://riptutorial.com/fr/vim
- emacs → tools/emacs/.emacs
  https://www.gnu.org/software/emacs/tour/

About Git VCS

- Records changes to a file(s) over time
- Allows to revert files back to a previous state
- Reverts the entire project back to a previous state
- Compares changes over time
- See who last modified something
- Recovers lost files
- Fully mirrors the repository



→ https://openclassrooms.com/fr/courses/2342361-gerez-votre-code-avec-git-et-github

Development environment - Git

Branch model

- One directory
- One version = one branch
- Official Notus repository `master` and `dev` branches cloned to local repository

Local branches management

create a branch, checkout a branch:

```
$ git branch my-branch
$ git checkout my-branch
```

merge branch:

```
$ git merge branch-to-merge
```

rebase from dev:

```
$ git rebase dev
```

branches available:

```
$ git branch -a
```

get differences between two branches:

```
$ git diff branch_name
```

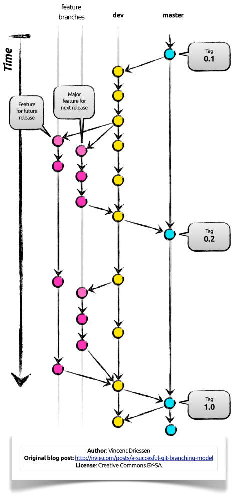
Server dialogue

get the last dev version:

```
$ git pull official dev
```

push a branch to your origin remote repository:

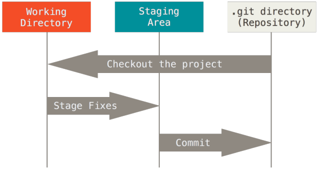
```
$ git push
```



Development environment and porting - Git

The Three States, basic workflow

- File modification in the working directory
- Stage the files
- Commit



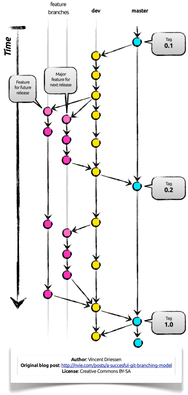
Few commands to start with Git

Change file with text editor

```
$ git status
$ git add file-name
$ git commit -a
```

→ add a coment to your commit

```
$ git commit -a --amend
```



Conclusion

- Use of some standard development tools (Git, CMake, Doxygen)
- Use of specific libraries: IO, solvers
- Single Doxygen documentation: concepts, installation, modeling, subroutines
- Different users (from student to researcher, from modeling to numerical methods)
- Different computers
- A few scripts, easy to use and modify for:
  - installation
  - execution
  - V&V
  - scalability studies

$\rightarrow$ *ongoing project, version 0.4.0 only !*