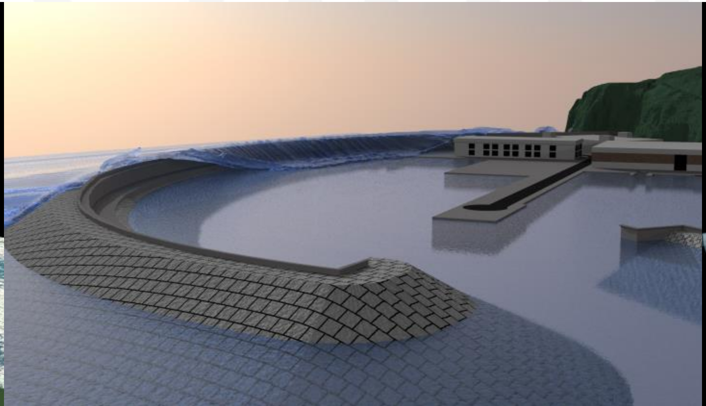

IHCantabria
UNIVERSIDAD DE CANTABRIA
R+D+i for a Sustainiable Development

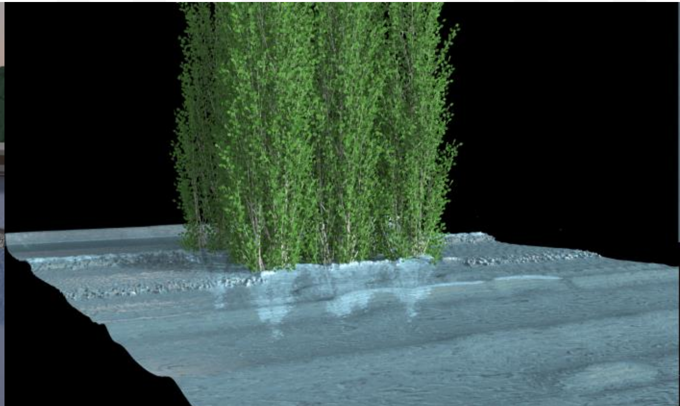
(Beginners Course)

IHFOAM applied to Coastal Engineering

Breaking solitary waves on a mild slope (2D)

Gabriel Barajas, Javier L. Lara, María Maza





- Now we are going to create a 3D case from a 2D one:
  - Copy a 2D case and change the folder name:

    $ cp –r ~/OpenFOAM-v1812/tutorials/multiphase/interFoam/laminar/ waveExampleSolitary ~/IHFoamCourse/.

- Rename the case:
  - Rename the case:

    $ mv ~/IHFoamCourse/waveExampleStokesV ~/IHFoamCourse/synolakis
  - Set OpenFOAM environment:

    $ source ~/OpenFOAM/OpenFOAM-v1812/etc/bashrc
  - Ensure everything you don't need is deleted

    $ cd ~/IHFoamCourse/synolakis

    $ ./Allclean

OpenFOAM workflow

Geometric Domain

↓

Numeric Domain

Mesh generation

↓

Numerical Setup

Boundary conditions, Initial conditions, Numerical parameters, Pre-processing

↓

Run the Simulation

Execute serial or parallel

↓

Post-processing

Wave gauges, run-up, etc.

OpenFOAM case

- 0
  - alpha.water
  - p_rgh
  - U
  - k
  - epsilon
  - nut
- constant
  - g
  - transportProperties
  - turbulenceProperties
  - waveProperties
- system
  - blockMeshDict
  - setFieldsDict
  - snappyHexMeshDict
  - extrudeMeshDict
  - fvSchemes
  - fvSolution
  - decomposeParDict
  - controlDict

- Update ***system/blockMeshDict*** to fit the laboratory set-up (synolakis 1986):

```
convertToMeters 1;

vertices
(
    ( 0 0.0 0.0)
    ( 10.0 0.0 0.0)
    ( 10.0 0.01 0.0)
    ( 0 0.01 0.0)
    ( 0.0 0.0 1.0)
    ( 10.0 0.0 1.0)
    ( 10.0 0.01 1.0)
    ( 0.0 0.01 1.0)
);

blocks
(
    hex (0 1 2 3 4 5 6 7) (2000 1 200) simpleGrading (1 1 1)
);
```

- Create the base mesh:
  $ blockMesh
- Check the base mesh quality:
  $ checkMesh

- In the ***0.org*** folder, display the VoF (***alpha.water***), the velocity (***U***) and the pressure (***p_rgh***) files :

alpha.water

```
dimensions      [0 0 0 0 0 0 0];
internalField   uniform 0;

boundaryField
{
    inlet
    {
        type            waveAlpha;
        value           uniform 0;
    }
    outlet
    {
        type            zeroGradient;
    }
    ground
    {
        type            zeroGradient;
    }
    sides
    {
        type            empty;
    }
    top
    {
        type            inletOutlet;
        inletValue      uniform 0;
        value           uniform 0;
    }
}
```

U

```
dimensions      [0 1 -1 0 0 0 0];
internalField   uniform (0 0 0);

boundaryField
{
    inlet
    {
        type            waveVelocity;
        value           uniform (0 0 0);
    }
    outlet
    {
        type            waveVelocity;
        value           uniform (0 0 0);
    }
    sides
    {
        type            empty;
    }
    ground
    {
        type            fixedValue;
        value           uniform (0 0 0);
    }
    top
    {
        type            pressureInletOutletVelocity;
        value           uniform (0 0 0);
    }
}
```

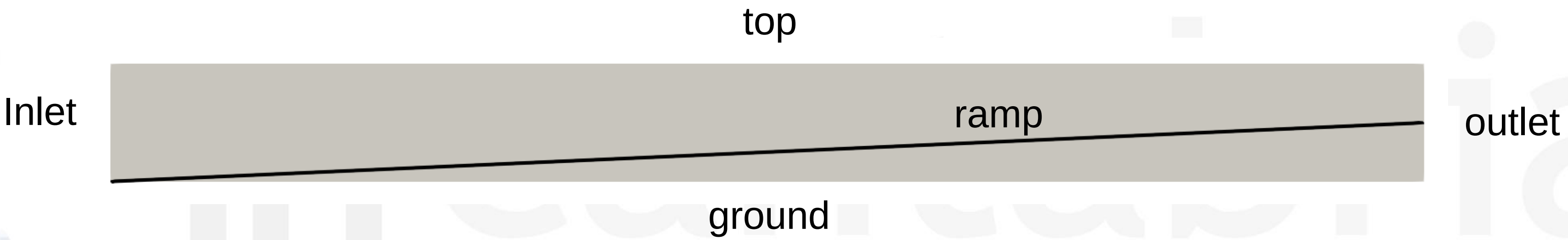
p_rgh

```
dimensions      [1 -1 -2 0 0 0 0];
internalField   uniform 0;

boundaryField
{
    inlet
    {
        type            fixedFluxPressure;
        value           uniform 0;
    }
    outlet
    {
        type            fixedFluxPressure;
        value           uniform 0;
    }
    ground
    {
        type            fixedFluxPressure;
        value           uniform 0;
    }
    sides
    {
        type            empty;
    }
    top
    {
        type            totalPressure;
        p0              uniform 0;
    }
}
```

- Define and create a mild slope (using Autocad, Rhino, etc.).
- Check the .stl file; open Paraview, load the .stl file and check that the geometry fits the base mesh:

$ touch ih.foam && paraview



- Update the case to take into account the new geometry:

$ mkdir constant/triSurface
$ cp ramp.stl constant/triSurface/.

- Using ***snappyHexMesh***, as mesh generator to take the existing base mesh and remesh it to fit the real geometry of the experiments.

- Copy snappyHexMeshDict from a tutorial:
  $ cp –r ~/OpenFOAM-v1806/tutorials/multiphase/interFoam/RAS/mixerVesselAMI/system/snappyHexMeshDict ~/IHFoamCourse/overSetWaves/system/.

- This intermediate mesh, is created from the dictionary ***system/snappyHexMeshDict***:

  - ***CastellatedMesh***:
    - Mesh Refinement in prescribed regions.
    - Detection of the domain (surface and volume).
    - Removal of cells outside the domain.

  - ***Snap***:
    - Mesh morphing to follow the provided geometry.
    - Layer addition could also be done.

```
// Which of the steps to run
castellatedMesh true;
snap            true;
addLayers       false;
```

```
geometry
{
    ramp.stl
    {
        type triSurfaceMesh;
        name ramp;
    }
}
```

→ Definition of the a new boundary (mesh refinement and removal of cells around it).

```
nCellsBetweenLevels 3;
```

→ Minimun number of cells before going to the next level of resolution (Number of buffer layers between different levels.)

```
features
(
);
```

→ List of feature edges, that describe Sharp cornes, for refinement.

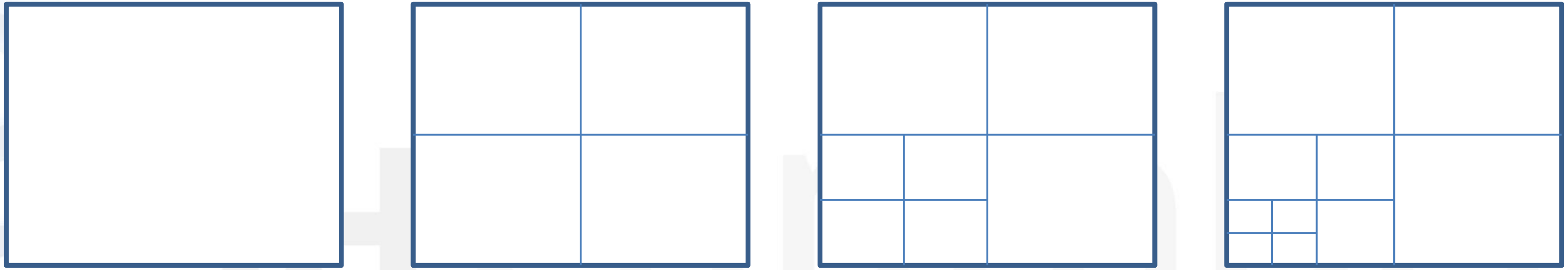
```
refinementSurfaces
{
    ramp
    {
        level ( 1 1 );
    }
}
```

→ Surface based refinements, based on two levels for every surface (the first is the minimum level, the second level is the maximum level).

```
refinementRegions
{
}
```

→ Volume based refinements, based on two levels for every volume (the first is the minimum level, the second level is the maximum level).

- Refinement levels in OpenFOAM: increase in the refinement level reduces the cell size by half.



Level 0 Level 1 Level 2 Level 3

```
locationInMesh (0.25 0.001 0.30);
```

→ Cartesian points (x, y, z) to identify the volumen to retain the final mesh.

- Create the intermediate mesh:
  $ snappyHexMesh -overwrite.

- Ajust the mesh: as it has been created by ***snappyHexMesh*** with several cells in the spanwise direction, it must be modified and extrude it to be purely two dimensional (2D).

```
constructFrom patch;
sourceCase ".";
sourcePatches (side1);

// If construct from patch: patch to use for back (can be same as sourcePatch)
exposedPatchName side2;

// Flip surface normals before usage.
flipNormals true;

//- Linear extrusion in point-normal direction
extrudeModel        linearNormal;

nLayers             1;

expansionRatio      1.0;

linearNormalCoeffs
{
    thickness       0.02;
}

// Do front and back need to be merged? Usually only makes sense for 360
// degree wedges.
mergeFaces false;
```

- ***sourcePatches***: name of the patch to extrude.
- ***exposedPatchName***: name of the patch opposed to the extruded one (sourcePatches)
- ***nLayers***: number of divisions from sourcePatches to exposedPatchName
- ***thickness***: length of the extrusion.

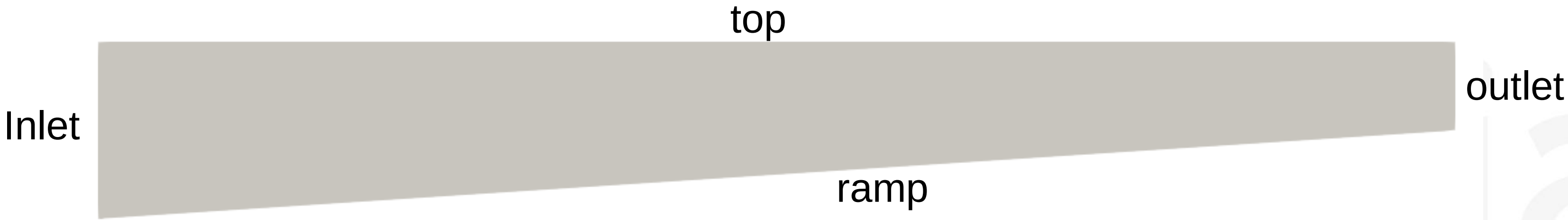
- Extrude the intermediate mesh:
  $ extrudeMesh

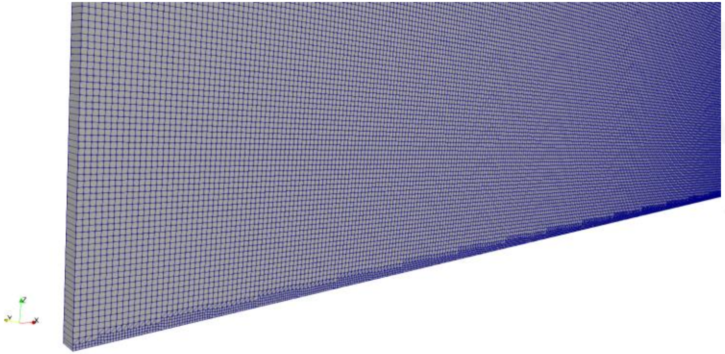
- Check the final mesh quality:
  $ checkMesh

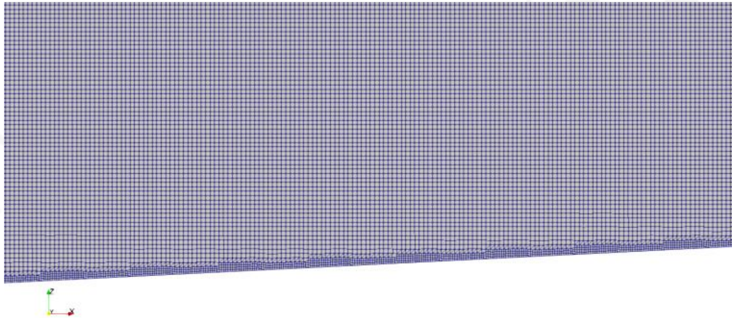
- Check your final mesh with Paraview:

  $ paraview

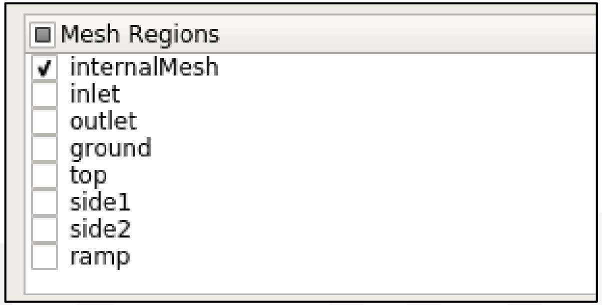
- Load the ih.foam file and press "*Apply*". (Remember to tick "*Skip Zero Time*", as the boundary conditions in the 0 folder have not been updated yet.)







- The final boundaries can be checked with Paraview (in the *Mesh Regions* dialog box) or they can be checked in the ***constant/polyMesh/boundary*** file.




```
7
(
    inlet
    {
        type            patch;
        nFaces          200;
        startFace       708588;
    }
    outlet
    {
        type            patch;
        nFaces          102;
        startFace       708788;
    }
    ground
    {
        type            wall;
        inGroups        1(wall);
        nFaces          1000;
        startFace       708890;
    }
    top
    {
        type            patch;
        nFaces          2000;
        startFace       709890;
    }
    side1
    {
        type            empty;
        inGroups        1(empty);
        nFaces          710788;
        startFace       711890;
    }
    side2
    {
        type            empty;
        inGroups        1(empty);
        nFaces          0;
        startFace       1422678;
    }
    ramp
    {
        type            wall;
        inGroups        1(wall);
        nFaces          2000;
        startFace       1422678;
    }
)
```

- Once the final boundaries are known, update ***0.org*** folder:
  - alpha.water:

```
ground
{
    type            zeroGradient;
}
```

→

```
ramp
{
    type            zeroGradient;
}
```

  - p_rgh:

```
ground
{
    type            fixedFluxPressure;
    value           uniform 0;
}
```

→

```
ramp
{
    type            fixedFluxPressure;
    value           uniform 0;
}
```

  - U:

```
ground
{
    type            fixedValue;
    value           uniform (0 0 0);
}
```

→

```
ramp
{
    type            fixedValue;
    value           uniform (0 0 0);
}
```

```
outlet
{
    type            waveVelocity;
    value           uniform (0 0 0);
}
```

→

```
outlet
{
    type            noSlip;
}
```

- The case is defined as turbulent in:
  $ more constant/turbulenceProperties

```
simulationType  RAS;

RAS
{
    RASModel        kEpsilon;

    turbulence      on;

    printCoeffs     on;
}
```

- Therefore, the turbulent kinematic energy (***k***), the turbulent dissipation (***epsilon***) and the turbulent viscosity (***nut***) variables must be defined and added to the ***0.org*** folder:

Breaking solitary wave on a mild slope (2D)

k

```
dimensions      [0 2 -2 0 0 0 0];
internalField   uniform 0.0084;

boundaryField
{
    inlet
    {
        type            zeroGradient;
    }
    outlet
    {
        type            zeroGradient;
    }
    side1
    {
        type            empty;
    }
    side2
    {
        type            empty;
    }
    ground
    {
        type            kqRWallFunction;
        value           uniform 0.0084;
    }
    ramp
    {
        type            kqRWallFunction;
        value           uniform 0.0084;
    }
    top
    {
        type            inletOutlet;
        inletValue      uniform 0.0084;
        value           uniform 0.0084;
    }
}
```

epsilon

```
dimensions      [0 2 -3 0 0 0 0];
internalField   uniform  0.0005094;

boundaryField
{
    inlet
    {
        type            zeroGradient;
    }
    outlet
    {
        type            zeroGradient;
    }
    side1
    {
        type            empty;
    }
    side2
    {
        type            empty;
    }
    ground
    {
        type            epsilonWallFunction;
        value           uniform 0.0005094;
    }
    ramp
    {
        type            epsilonWallFunction;
        value           uniform 0.0005094;
    }
    top
    {
        type            inletOutlet;
        inletValue      uniform 0.0005094;
        value           uniform 0.0005094;
    }
}
```

nut

```
dimensions      [0 2 -1 0 0 0 0];
internalField   uniform 0;

boundaryField
{
    inlet
    {
        type            calculated;
        value           uniform 0;
    }
    outlet
    {
        type            calculated;
        value           uniform 0;
    }
    side1
    {
        type            empty;
    }
    side2
    {
        type            empty;
    }
    ground
    {
        type            nutkWallFunction;
        value           uniform 0;
    }
    ramp
    {
        type            nutkWallFunction;
        value           uniform 0;
    }
    top
    {
        type            calculated;
        value           uniform 0;
    }
}
```

- Update the wave conditions in ***constant/waveProperties***:
  - ***WaveModel***: Boussinesq solitary wave
  - ***nPaddle***: 1 single wave paddle (2d)
  - ***waveHeight***: H = 0.07 m.
  - ***activeAbsorption***: no (absorption yes/no at generation)

```
inlet
{
    alpha               alpha.water;

    waveModel           Boussinesq;

    nPaddle             1;

    waveHeight          0.07;

    waveAngle           0.0;

    rampTime            0.0;

    activeAbsorption no;

    wavePeriod          0.0;
}
```

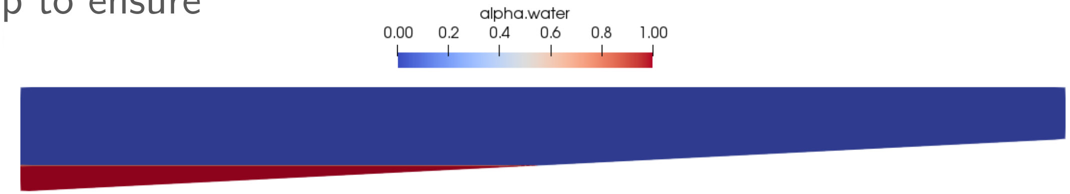
- Update the initial set-up in ***system/setFieldsDict***:

```
defaultFieldValues
(
    volScalarFieldValue alpha.water 0
);

regions
(
    boxToCell
    {
        box (-1.0 -1.0 -1.0) (10.0 1.0 0.25);
        fieldValues
        (
            volScalarFieldValue alpha.water 1
        );
    }
);
```

- $ cp -r 0.org 0
  $ setFields

- Open Paraview and plot the initial set-up to ensure eveything is correct:
  $ paraview



Breaking solitary wave on a mild slope (2D)

- Water and air properties are defined in:
  $ more constant/transportProperties

```
phases (water air);

water
{
    transportModel  Newtonian;
    nu              [0 2 -1 0 0 0 0] 1e-06;
    rho             [1 -3 0 0 0 0 0] 1000;
}

air
{
    transportModel  Newtonian;
    nu              [0 2 -1 0 0 0 0] 1.48e-05;
    rho             [1 -3 0 0 0 0 0] 1;
}

sigma           [1 0 -2 0 0 0 0] 0.07;
```

- Gravity is defined in:
  $ more constant/g

```
dimensions      [0 1 -2 0 0 0 0];
value           ( 0 0 -9.81 );
```

Numerical Schemes:

- ***system/fvSchemes:*** is the file that sets the numerical scheme for the different terms.

```
ddtSchemes
{
    default         Euler;
}

gradSchemes
{
    default         Gauss linear;
}

divSchemes
{
    div(rhoPhi,U)   Gauss linearUpwind grad(U);
    div(phi,alpha)  Gauss vanLeer;
    div(phirb,alpha) Gauss linear;
    div(((rho*nuEff)*dev2(T(grad(U))))) Gauss linear;

    div(phi,k)      Gauss upwind;
    div(phi,epsilon) Gauss upwind;
}
```

Temporal discretization (***Euler*** is a first order implicit discretisation scheme).

Gradient derivative terms, that is surface normal gradient terms (***Gauss linear*** is second order discretisation scheme)

Divergence terms, such as advection terms and other terms that are often diffusive in nature (***Gauss linearUpwind grad(U)*** is second order, ***Gauss vanLeer*** is second order and ***Gauss upwind*** is a first order numerical scheme)

Numerical Schemes:

```
laplacianSchemes
{
    default          Gauss linear limited 0.5;
}

interpolationSchemes
{
    default          linear;
}

snGradSchemes
{
    default          limited 0.5;;
}
```

Laplacian terms, such as the diffusion term in the momentum equation (**Gauss linear limited** is second order accuracy, values between 0 and 1 to handle a non-orthogonal mesh)

Cell to face interpolations of values (**linear** is second order discretisation scheme)

Component of gradient normal to a cell face (**limited** is second order accuracy, values between 0 and 1 to handle a non-orthogonal mesh)

Algorithm control:

- In ***system/fvSolutions*** are defined the equations solvers, tolerances and algorithms.
- Controls for ***MULES***, solver of the VoF equation:
  - ***nAlphaCorr***: loops over VoF equation
  - ***nAlphaSubcycles***: number of sub-cycles within the VoF equation
  - ***cAlpha***: artificial compression velocity.

```
solvers
{
    "alpha.water.*"
    {
        nAlphaCorr      1;
        nAlphaSubCycles 3;
        cAlpha          1;
    }
```

Algorithm control:

- For each variable solved in the particular equation, the type of solver and parameters that are used by the solver must be defined.
- Normally, the last iteration (variables are solved multiple times within a solution step) is solved with different parameters.

```
pcorr
{
    solver          PCG;
    preconditioner  DIC;
    tolerance       1e-6;
    relTol          0.1;
}
pcorrFinal
{
    solver          PCG;
    preconditioner  DIC;
    tolerance       1e-7;
    relTol          0;
}
p_rgh
{
    solver          PCG;
    preconditioner  DIC;
    tolerance       1e-6;
    relTol          0.1;
}
p_rghFinal
{
    solver          PCG;
    preconditioner  DIC;
    tolerance       1e-7;
    relTol          0;
}
"(U|k|epsilon)"
{
    solver          PBiCG;
    preconditioner  DILU;
    tolerance       1e-6;
    relTol          0.1;
}
"(U|k|epsilon)UFinal"
{
    solver          PBiCG;
    preconditioner  DILU;
    tolerance       1e-7;
    relTol          0;
}
```

Algorithm control:

- ***PIMPLE*** algorithm solves the pressure-velocity coupling in the Navier-Stokes equations.

- ***PIMPLE*** algorithm combines PISO and SIMPLE.
  - ***momentumPredictor***: switch the control for solving the momentum predictor.
  - ***nCorrectors***: number of times the algorithm solves the pressure equation and momentum corrector in each step
  - ***nOrthogonalCorrectors***: specifies repeated solutions of the pressure equation, used to update the explicit non-ortogonal correction.

```
PIMPLE
{
    momentumPredictor no;
    nCorrectors      2;
    nNonOrthogonalCorrectors 0;
}
```

- Define simulation parameters in ***system/controlDict***
  - Solver: ***interFoam*** (incompressible two phase flow)
  - ***startTime*** (start time for the simulation), ***endTime*** (end time for the simulation), ***deltaT*** (time step of the simulation).
  - ***writeInterval*** (controls the timing of write output), ***purgeWrite*** (integer representing a limit on the number of time directories that are stored).
  - ***maxCo*** (maximun Courant Number), ***maxAlphaCo*** (maximun Courant number for the pase fields), ***maxDeltaT*** (upper limit of the time step).

```
application      interFoam;

startFrom        latestTime;

startTime        0;

stopAt           endTime;

endTime          20.0;

deltaT           0.01;

writeControl     adjustableRunTime;

writeInterval    0.1;

purgeWrite       0;

writeFormat      ascii;

writePrecision   6;

writeCompression off;

timeFormat       general;

timePrecision    6;

runTimeModifiable yes;

adjustTimeStep   yes;

maxCo            0.65;
maxAlphaCo       0.65;
maxDeltaT        0.05;
```

- Update the runtime postprocessing sensors (***system/controlDict***)
  - to get the iso-Surface of the free surface elevation:

```
functions
{
    freeSurface
    {
        type surfaces;
        functionObjectLibs
        (
            "libsampling.so"
        );
        writeControl outputTime;
        outputInterval 1;
        surfaceFormat stl;
        interpolationScheme cellPoint;
        surfaces
        (
            topFreeSurface
            {
                type isoSurface;
                isoField alpha.water;
                isoValue 0.5;
                interpolate true;
            }
        );
        fields
        (
            alpha.water
        );
    }
```

Breaking solitary wave on a mild slope (2D)

- Update the runtime postprocessing sensors (***system/controlDict***)
  - to get the free surface at some specific positions along the domain.

```
line
{
    type            sets;
    libs            ("libsampling.so");
    enabled         true;
    writeControl    writeTime;
    writeInterval   1;

    interpolationScheme cellPoint;
    setFormat       raw;
    sets
    (
        line1
        {
            type    uniform;
            axis    distance;
            start   ( 1.0 1.0 0.0 );
            end     ( 1.0 1.0 0.7 );
            nPoints 1001;
        }
        line2
        {
            type    uniform;
            axis    distance;
            start   ( 2.0 1.0 0.0 );
            end     ( 2.0 1.0 0.7 );
            nPoints 1001;
        }
        line3
        {
            type    uniform;
            axis    distance;
            start   ( 3.0 1.0 0.0 );
            end     ( 3.0 1.0 0.7 );
            nPoints 1001;
        }
```

```
            line4
            {
                type    uniform;
                axis    distance;
                start   ( 5.0 1.0 0.0 );
                end     ( 5.0 1.0 0.7 );
                nPoints 1001;
            }
            line5
            {
                type    uniform;
                axis    distance;
                start   ( 7.0 1.0 0.0 );
                end     ( 7.0 1.0 0.7 );
                nPoints 1001;
            }
            line6
            {
                type    uniform;
                axis    distance;
                start   ( 9.0 1.0 0.0 );
                end     ( 9.0 1.0 0.7 );
                nPoints 1001;
            }
        );

        fixedLocations  false;
        fields
        (
            U alpha.water
        );
    }
}
```

- Decompose case:
  - ***system/decomposeParDict***: if we want to run our simulation in parallel we can decompose it using this file:
    - ***numberOfSubdomains***: set the number of parts in which we are going to split our domain.
    - ***n***: it should be equal to the number of subdomains

Run the command:

$ decomposePar

```
numberOfSubdomains 2;

method          hierarchical;

hierarchicalCoeffs
{
    n           (2 1 1);
    delta       0.001;
    order       xyz;
}
```

- Run the case!

  $ mpirun –np 2 interFoam –parallel > log.synolakis &

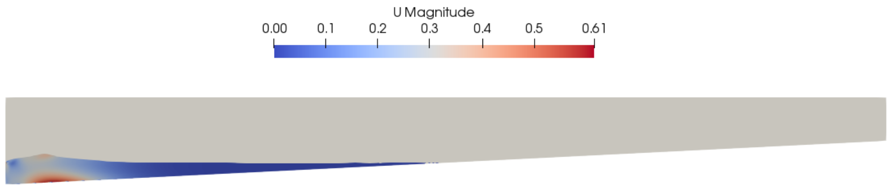
  $ tail –f log.synolakis

  $ kill PID number

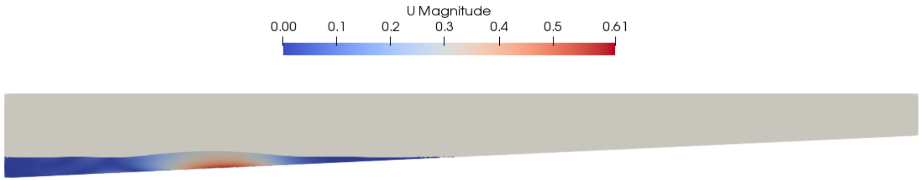
- Postprocessing with Paraview:

  $ paraFoam –touch

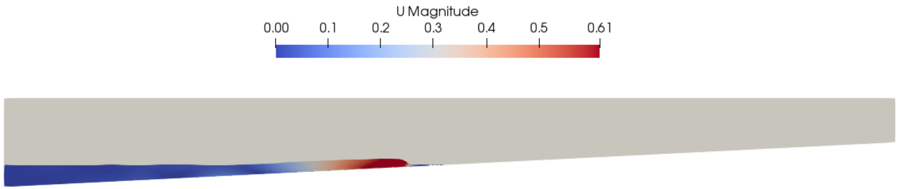
Breaking solitary wave on a mild slope (2D)



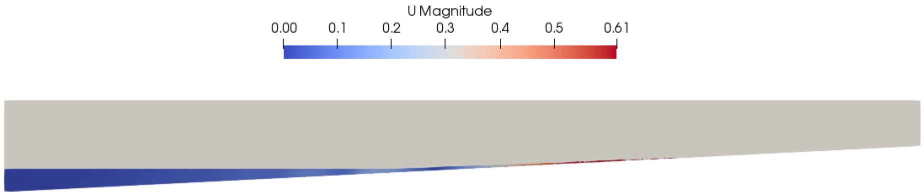
Time: 1.45 s.



Time: 3.00 s.



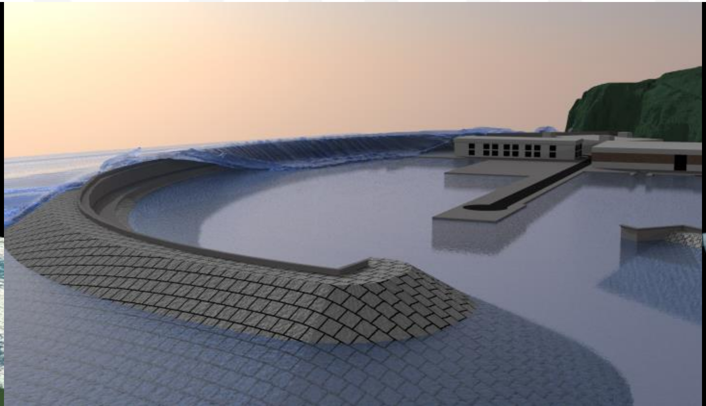
Time: 4.49 s.

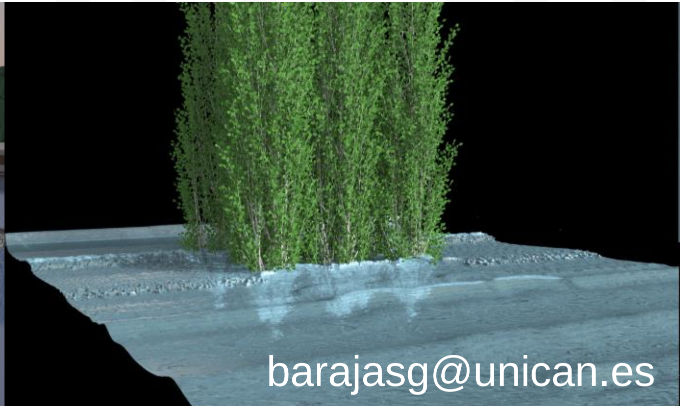


Time: 6.60 s.

IHFOAM

Gabriel Barajas, Javier L. Lara, María Maza





barajasg@unican.es