User Programming & Automation

- What are User Defined Functions
- Introduction to C
- Set-Up C User Routines in Fluent
- Programming in other CFD Commercial Codes
- Automation

Acknowledgement: this handout is partially based on Fluent training material

Introduction to UDF Programming

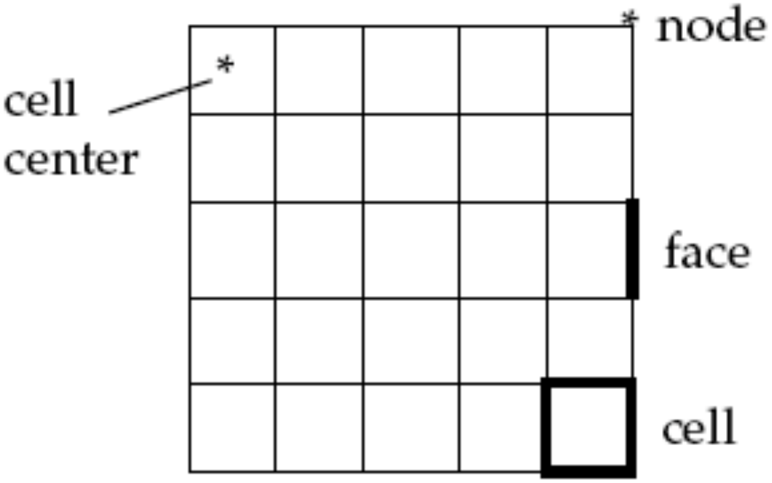
Why programming in commercial codes?

- The codes are general-purpose but cannot anticipate all needs
- New (physical) models can be developed in a user-friendly environment
- Large number of problems (test-cases) can be addressed with the same implementation

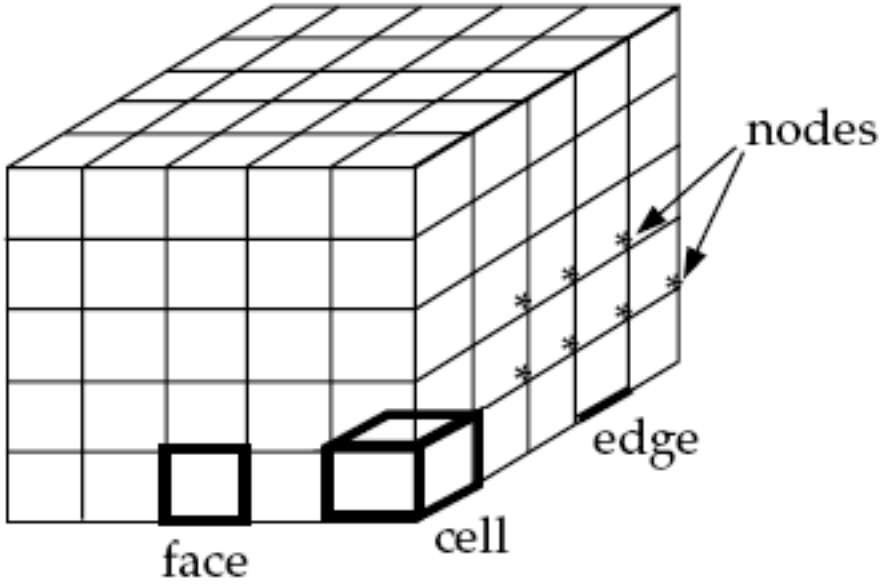
What is a the User Defined Function (UDF)?

- C (Fluent) or FORTRAN (StarCD, CFX) routines programmed by the user linked to the solver to perform certain operations:
    - initialization
    - special boundary condition (i.e. space or time dependent)
    - material properties
    - source terms
    - reaction rates
    - postprocessing and reporting
    - debugging
    - ….

Geometrical Entities - Reminder



2D



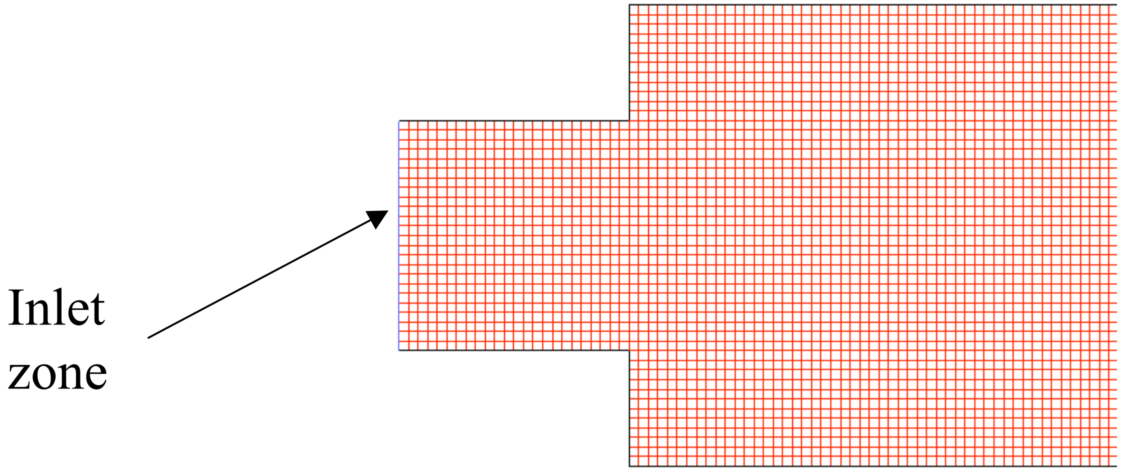
3D

Collection of entities are called “zones”

A Simple UDF Example

Specify a Parabolic Velocity Profile at the Inlet

Goal: The UDF (inlet_parab) set the values of the x-velocity component at the cell faces of the inlet boundary zone



1) Determine the cell-faces belonging to the inlet zone
2) Loop on all those faces
3) Determine the coordinate of the face centroid
4) Specify the x-velocity component

A Simple UDF Example

Parabolic Velocity Profile:

$$u = \frac{1}{2\rho\nu}\left(-\frac{dp}{dx}\right) y\,(h-y)$$

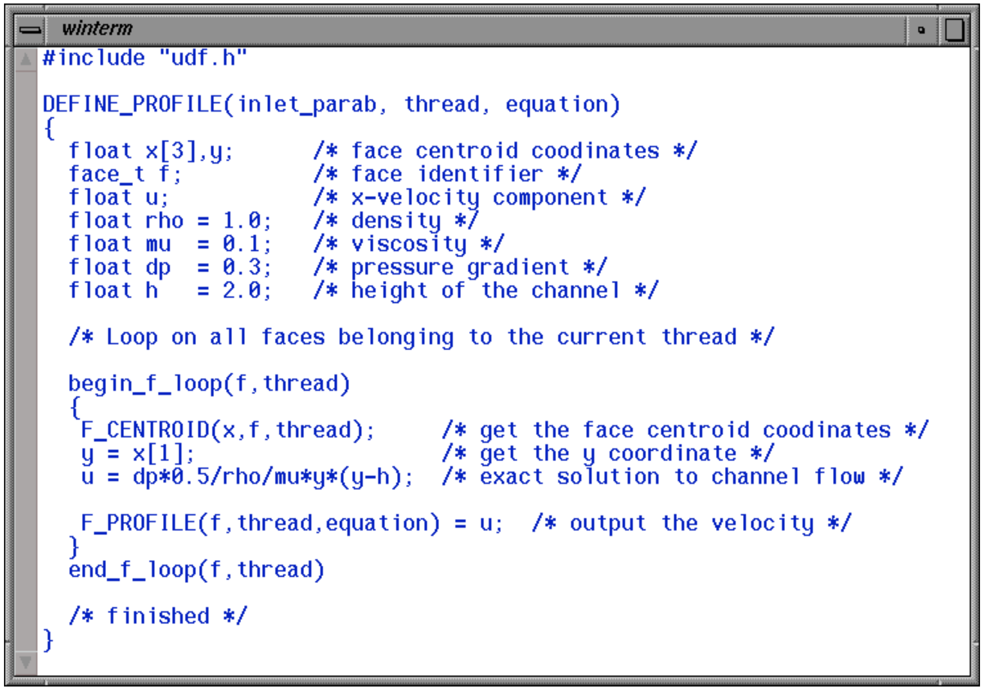
function inlet_parab

Definitions

Loop over all the inlet cell faces

Evaluate the face centroid coordinates

The function return the velocity

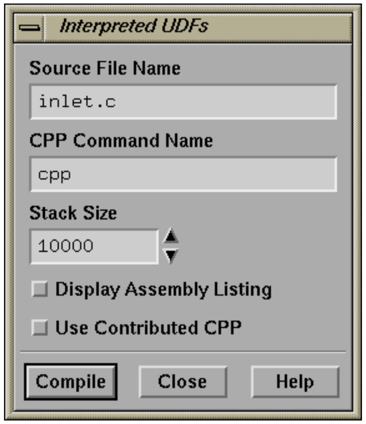


```
#include "udf.h"

DEFINE_PROFILE(inlet_parab, thread, equation)
{
  float x[3],y;      /* face centroid coodinates */
  face_t f;          /* face identifier */
  float u;           /* x-velocity component */
  float rho = 1.0;   /* density */
  float mu  = 0.1;   /* viscosity */
  float dp  = 0.3;   /* pressure gradient */
  float h   = 2.0;   /* height of the channel */

  /* Loop on all faces belonging to the current thread */

  begin_f_loop(f,thread)
  {
   F_CENTROID(x,f,thread);    /* get the face centroid coodinates */
   y = x[1];                  /* get the y coordinate */
   u = dp*0.5/rho/mu*y*(y-h);  /* exact solution to channel flow */

   F_PROFILE(f,thread,equation) = u;  /* output the velocity */
  }
  end_f_loop(f,thread)

  /* finished */
}
```

A Simple UDF Example

Compile/interpret the UDF:

Define → User Defined



Source File Name: inlet.c

CPP Command Name: cpp

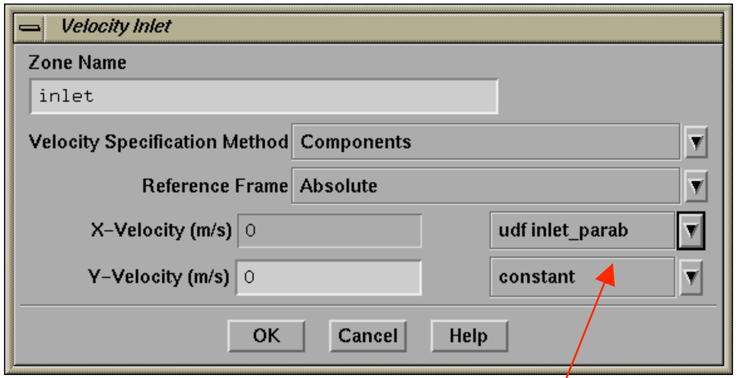
Stack Size: 10000

☐ Display Assembly Listing

☐ Use Contributed CPP

Attach the profile to the inlet zone

Define → Boundary Condition



Zone Name: inlet

Velocity Specification Method: Components

Reference Frame: Absolute

X-Velocity (m/s): 0 udf inlet_parab

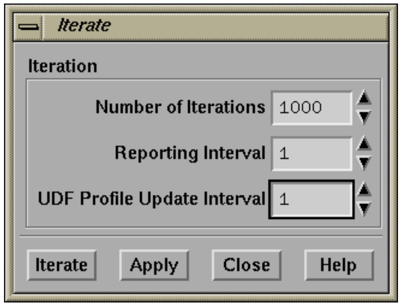
Y-Velocity (m/s): 0 constant

Equivalent to attach the profile from a separate simulation

A Simple UDF Example

Solve the equations

Solve → Iterate



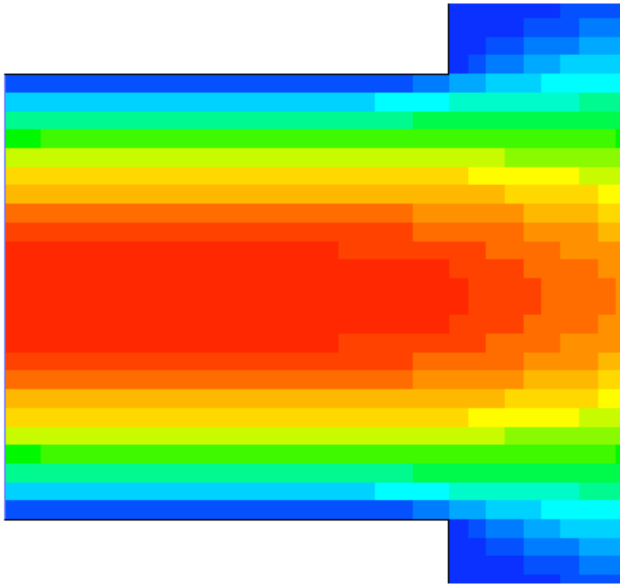
Iteration

Number of Iterations 1000

Reporting Interval 1

UDF Profile Update Interval 1

Final Result



C Programming

Typical C function:

```
/* A simple C function */          A comment line
#include “udf.h”                   A preprocessor directive for including files
#define PI 3.14159                 A preprocessor directive for macro substitution
float a = 1.2345;                  A variable with “global” scope
int myfunction(int x)              Function declaration (integer type)
{                                  Left curly brace opens body of function
   int y,z;                        Variable declarations
   y = 11;                         Set y = 11
   z = a*(x+y)*PI;                 Compute z
   printf(“z = %d”,z);             Print output to screen
   return z;                       Return integer value
}                                  Right curly brace closes body of function
```

C vs. FORTRAN

```c
/* A simple C function */
int myfunction(int x)
{
   int y,z;
   y = 11;
   z = x+y;
   printf(“z = %d”,z);
   return z;

}
```

```fortran
C An equivalent FORTRAN function
       INTEGER FUNCTION MYFUNCTION(X)

       INTEGER X,Y,Z
       Y = 11
       Z = X+Y
       WRITE (*,100) Z
       MYFUNCTION = Z
100    FORMAT(“Z = “,I5)
       END
```

Basic Programming Rules

Statements MUST end with a semicolon → ;

Comments are placed anywhere in the program between → /* …… */

Statements are grouped by curly brackets → { …… }

Variables defined within the body functions are local
Variables defined outside the body functions are global and can be used by all the functions that follows

Variables MUST be ALWAYS defined explicitly

Integer/real/double functions have to return a integer/real/double value

C is case sensitive!

Basic Statements

Arithmetic expressions in C look like FORTRAN

```
a = y + (i-b)/4;
```

Functions which return values can be used in assignment

```
a = mycompute(y,i,b);
```

Functions which do not return values can be called directly

```
mystuff(a,y,i,b);
```

Mathematical and various other default functions are available

```
a = pow(b,2);    /* pow(x,y) returns x raised to y */
```

Data Types and Arrays

Arrays of variables are defined using the notation

```
    int a[10]; /*define an array of ten integers */
```

Note the brackets are square [] not round ()!!

Note that the arrays ALWAYS start from 0

Standard C types are: Integer, Real, Double, Char

C allows to define additional types

```
typedef struct list{int id, float x[3], int id_next};
```

Pointers

A pointer is a variable which contain the address of another variable

Pointers are defined as:

```
int *a_p;     /* pointer to an integer variable */
int a;        /* an integer variable */
```

Set-up a pointer

```
a = 1;
a_p = &a;    /* &a return the address of variable a */
```

`*a_p` returns the content of the address pointed by `a_p`

Operators

Arithmetic Operators

=	assignment
+	addition
-	subtraction
*	multiplication
/	division
%	modulo
++	increment
--	decrement

Logical Operators

<	less than
<=	less than or equal to
>	greater than
>=	greater than or equal to
==	equal to
!=	not equal to

Conditional Statements

`if` and `if-else`

```
if (logical-expression)
  {statements}

if (logical-expression)
  {statements}
else
  {statements}
```

Example

```
if (q != 1) {a = 0; b = 1;}

if (x < 0.)
  y = x/50.;
else
  y = x/25.;

C Equivalent FORTRAN code
       IF (X.LT.0.) THEN
         Y = X/50.
       ELSE
         Y = X/25.
       ENDIF
```

Loop Procedure

`for` loops

```
for (begin ; end ;
increment)
  {statements}
```

where:

`begin` = expression which is executed at beginning of loop

`end` = logical expression which tests for loop termination

`increment` = expression which is executed at the end of each loop iteration (usually incrementing a counter)

Example

```
/* Print integers 1-10 and their squares
*/
int i, j, n = 10;
for (i = 1 ; i <= n ; i++)
{ j = i*i;
  printf("%d %d\n",i,j);
}
C Equivalent FORTRAN code
      INTEGER I,J
      N = 10
      DO I = 1,10
      J = I*I
      WRITE (*,*) I,J
      ENDDO
```

C Preprocessor

It handles Macro substitutions

```
#define A B
#define my_f(x,y) x+y*3-5
```

The preprocessor replaces `A` with `B`

It also handles file inclusion

```
#include “udf.h”
#include “math.h”
```

The files to be included MUST reside in the currect directory

Programming in C

Of course much more than just this….

Additional information are:

http://www.cs.cf.ac.uk/Dave/C/CE.html

Plenty of books:

“The C Programming Language”, Kernighan & Ritchie, 1988

UDF in Commercial CFD Codes

Commercial CFD codes allow the development of User Defined Functions for various applications. Anyway, the core of the software is closed.

UDF must be compiled and linked to the main code

Most codes provide macros and additional tools to facilitate the use of UDFs

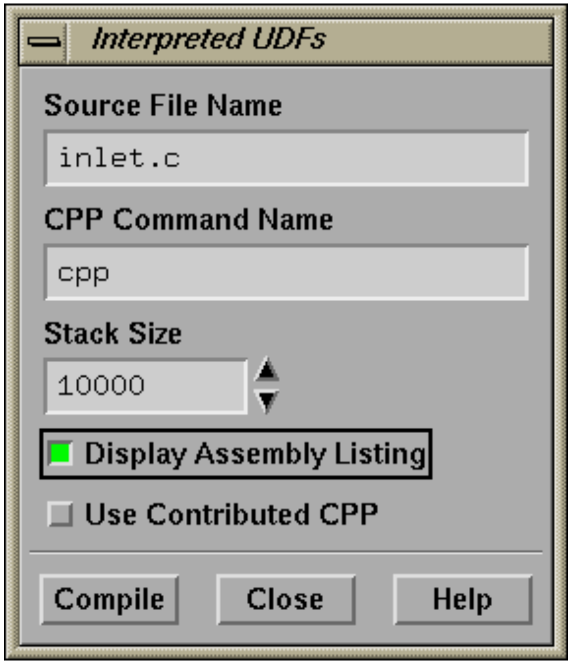
In Fluent there are two options:

- Interpreted
  - The code is executed on a "line-by-line" basis at run time
  - + does not need a separate compiler (completely automatic)
  - - slows down the execution and uses more memory
  - - somewhat limited in scope
- Compiled
  - A library of UDF is compiled and linked to the main code
  - Overcomes all the disadvantages reported above

Interpret the UDFs

Define → User Defined → Interpreted

Display of code translation in assembly (and eventual compiling errors)



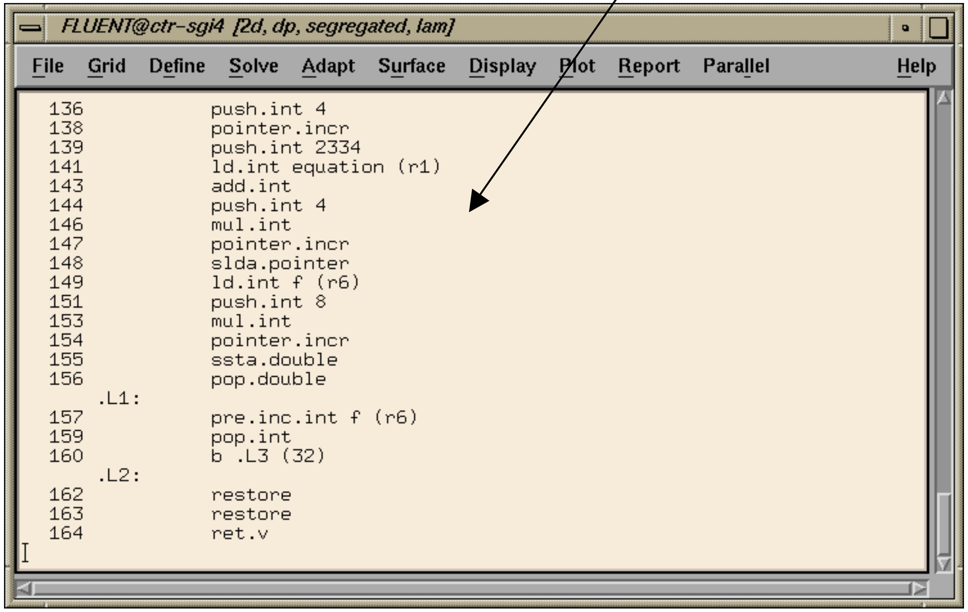
Source File Name: inlet.c

CPP Command Name: cpp

Stack Size: 10000

☑ Display Assembly Listing

☐ Use Contributed CPP

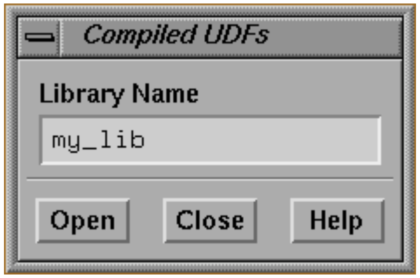


```
  136          push.int 4
  138          pointer.incr
  139          push.int 2334
  141          ld.int equation (r1)
  143          add.int
  144          push.int 4
  146          mul.int
  147          pointer.incr
  148          slda.pointer
  149          ld.int f (r6)
  151          push.int 8
  153          mul.int
  154          pointer.incr
  155          ssta.double
  156          pop.double
       .L1:
  157          pre.inc.int f (r6)
  159          pop.int
  160          b .L3 (32)
       .L2:
  162          restore
  163          restore
  164          ret.v
```

Default stack size might be too small for large arrays!

Compile the UDFs

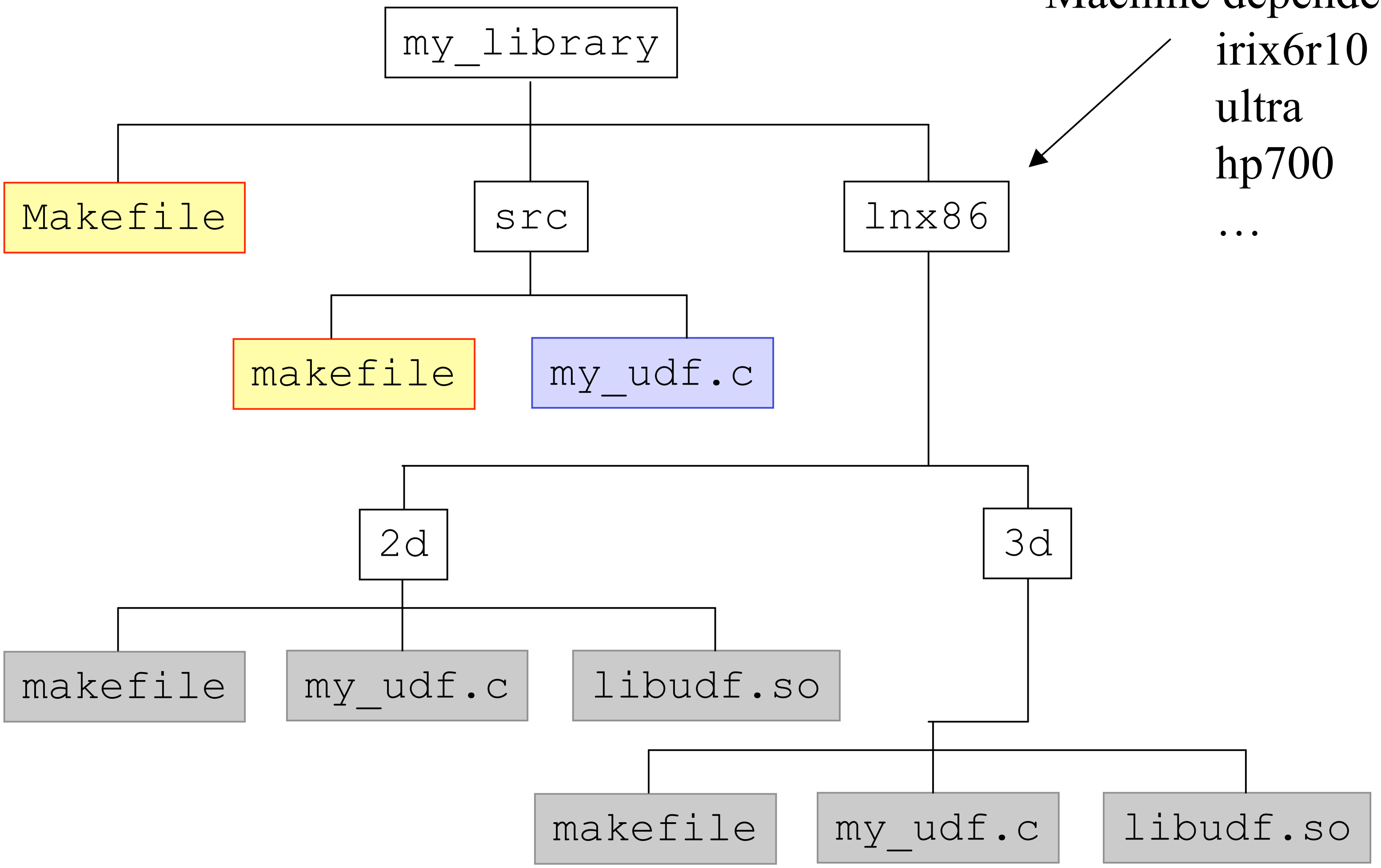
Define → User Defined → Compiled



Library Name: my_lib

The library MUST be precompiled in a directory tree

Directory tree for compiled UDFs



Machine dependent
irix6r10
ultra
hp700
…

Makefiles for UDFs

In the directory

```
/usr/local/Fluent.Inc/fluent6/src
```

There are two files

`akefile2.udf` to be copied in the directory `my_library`
`makefile.udf` to be copied in the directory `my_library/src`

The first one does not require modifications.
In the second one two macros MUST be modified

```
SOURCE = my_udf.c
FLUENT_INC = /usr/local/Fluent.Inc
```

UDFs in FLUENT

Boundary Conditions
Initial Conditions
Adjust Function
Source Terms
Material Properties
Execute on Demand
User Defined Scalars
User Defined Memory

→ Available MACROS

Pointers to threads
Geometry Macros
Cell and Face Variables
Arithmetic and Trigonometric Functions

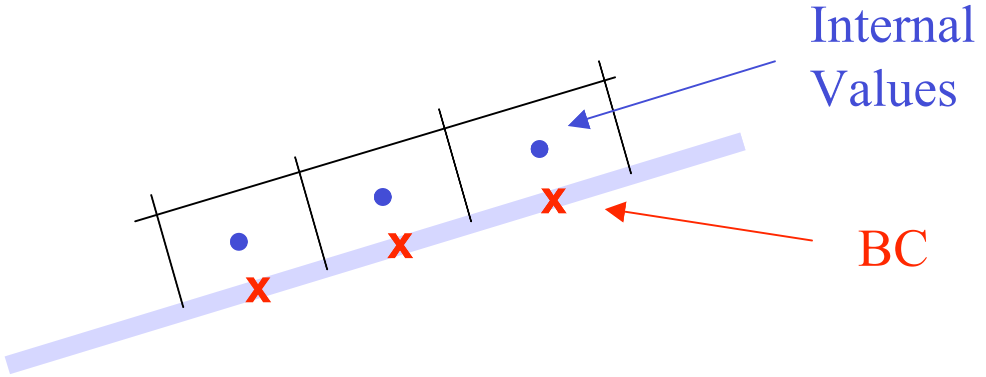
→ Programming Tools

Boundary Conditions

The inlet profile specification is based on the macro DEFINE_PROFILE

DEFINE_PROFILE can be used for wall boundary conditions as well to impose temperature, velocity, shear stress, etc.

It is possible to specify a constant value, a position-dependent or time-dependent values and to link the values on the boundary to the values in the internal cells



Internal Values

BC

Note that the BCs are applied to the faces of the cells (face centroids)

Initial Conditions

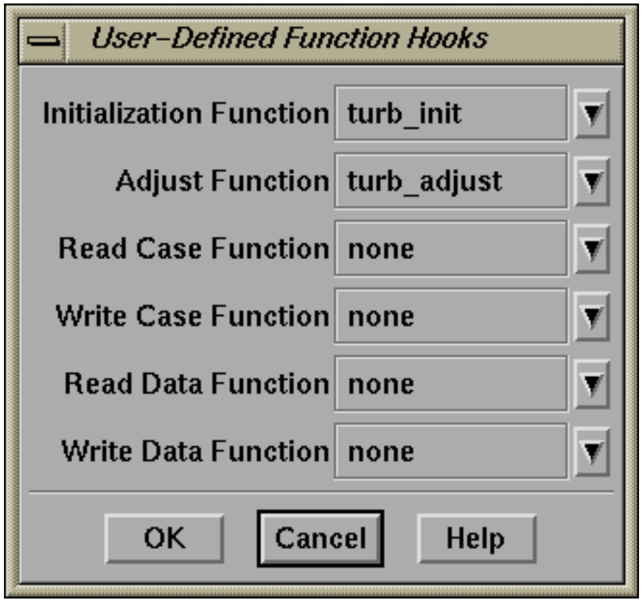
The initial condition specification can be performed using the macro DEFINE_INIT

Space-dependent conditions might be imposed

The routine is executed once at the beginning of the solution process

It is attached in the UDFs hooks

Define → User Defined → Function Hooks



User-Defined Function Hooks

Initialization Function: turb_init

Adjust Function: turb_adjust

Read Case Function: none

Write Case Function: none

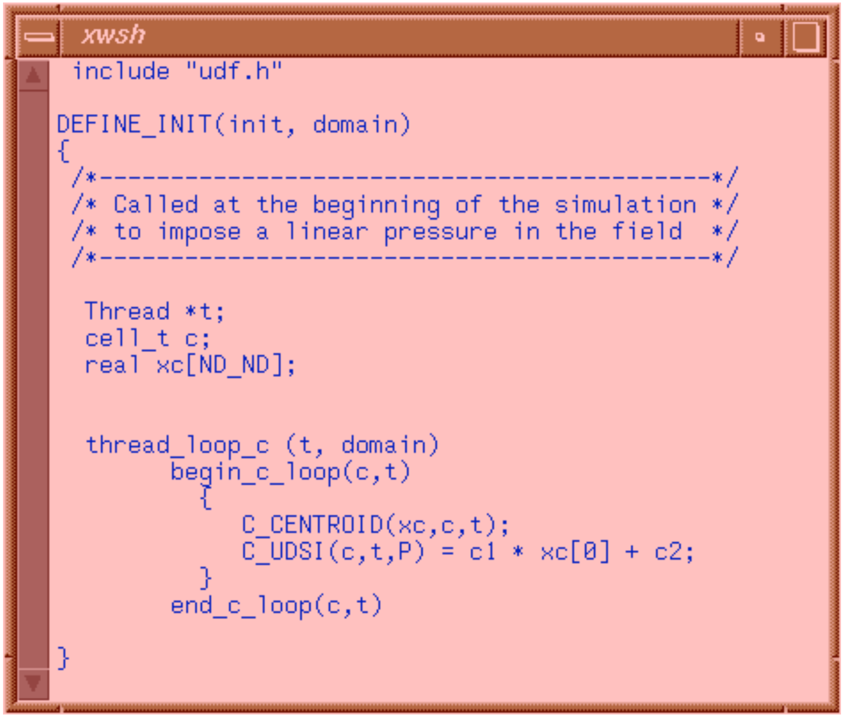
Read Data Function: none

Write Data Function: none

Initial Conditions

Note that the difference between the DEFINE_PROFILE and DEFINE_INIT is that the former performs a loop on the face-centroids belonging to a certain zone whereas the latter loops on the cell-centroids

Example:



```
 include "udf.h"

DEFINE_INIT(init, domain)
{
 /*-------------------------------------------*/
 /* Called at the beginning of the simulation */
 /* to impose a linear pressure in the field  */
 /*-------------------------------------------*/

  Thread *t;
  cell_t c;
  real xc[ND_ND];


  thread_loop_c (t, domain)
        begin_c_loop(c,t)
          {
             C_CENTROID(xc,c,t);
             C_UDSI(c,t,P) = c1 * xc[0] + c2;
          }
        end_c_loop(c,t)

}
```

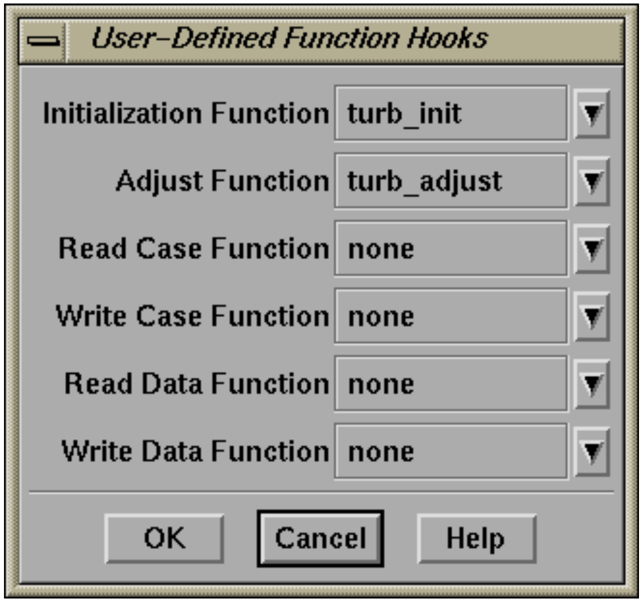
Adjust Function

Similar to the DEFINE_INIT but it is executed every iteration: DEFINE_ADJUST

Can be used for postprocessing, clipping, etc.

It is attached in the UDFs hooks

Define → User Defined → Function Hooks



Initialization Function	turb_init
Adjust Function	turb_adjust
Read Case Function	none
Write Case Function	none
Read Data Function	none
Write Data Function	none

Source Terms

To add a source term in the equations: DEFINE_SOURCE

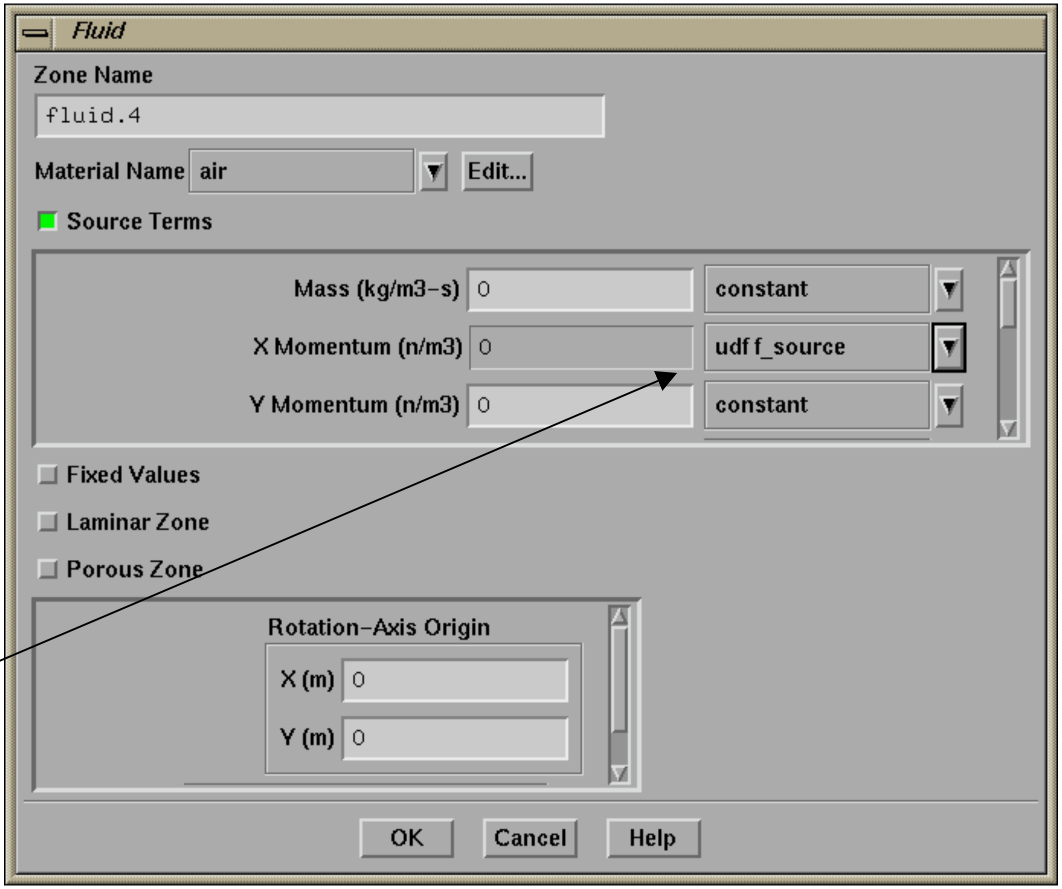
Can be used for:

- Continuity
- Momentum (component by component)
- Turbulent quantities
- Energy

It is different from the previous macros because it works on a cell-by-cell basis (no need to perform loops!)

Source Terms

Define → Boundary Conditions → Fluid



Zone Name: fluid.4

Material Name: air

☑ Source Terms

Mass (kg/m3-s): 0 constant

X Momentum (n/m3): 0 udf f_source

Y Momentum (n/m3): 0 constant

☐ Fixed Values

☐ Laminar Zone

☐ Porous Zone

Rotation-Axis Origin

X (m): 0

Y (m): 0

Activate source terms

Attach the UDF

Source Terms

In FLUENT the source terms are written in the form

$$S(\phi) = A + B\,\phi$$

where ϕ is the dependent variable

A is treated explicitly and B ϕ is treated implicitly

In the DEFINE_SOURCE both terms A and B have to be specified

Material Properties

To define the properties of the materials : DEFINE_PROPERTIES

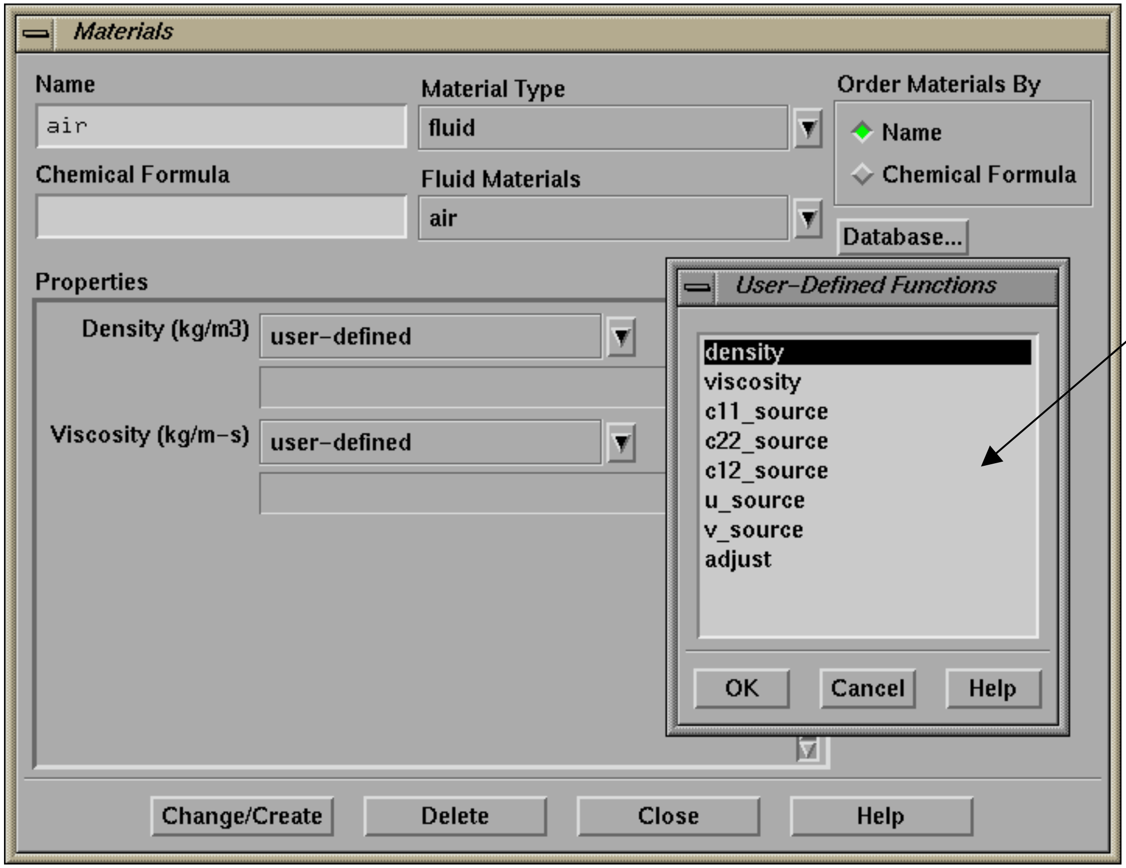
Can be used for:

- Viscosity
- Density
- Thermal Conductivity
- ….

As for the source terms works on a cell-by-cell basis

Material Properties

Define → Material Property



Name: air

Chemical Formula:

Material Type: fluid

Fluid Materials: air

Order Materials By: ● Name ○ Chemical Formula

Properties

Density (kg/m3): user-defined

Viscosity (kg/m-s): user-defined

User-Defined Functions:

- density
- viscosity
- c11_source
- c22_source
- c12_source
- u_source
- v_source
- adjust

All the available UDFs
Are shown in the menu

Execute on Demand

To perform operations instantaneously : EXECUTE_ON_DEMAND

It is executed when activated by the user

Can be used for:
- Debugging
- Postprocessing
- ….

Define → User Defined → Execute on Demand

Function v2f_mu_t

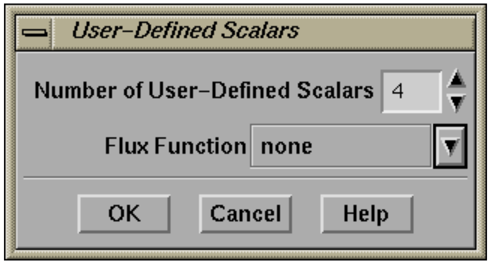
User Defined Scalar

In addition to the Continuity and Momentum Equations (and Energy) generic transport equations can be solved (up to 50!)

To include scalars in the calculations

1) Define the number of scalars

Define → User Defined → User Defined Scalars



User-Defined Scalars

Number of User-Defined Scalars: 4

Flux Function: none

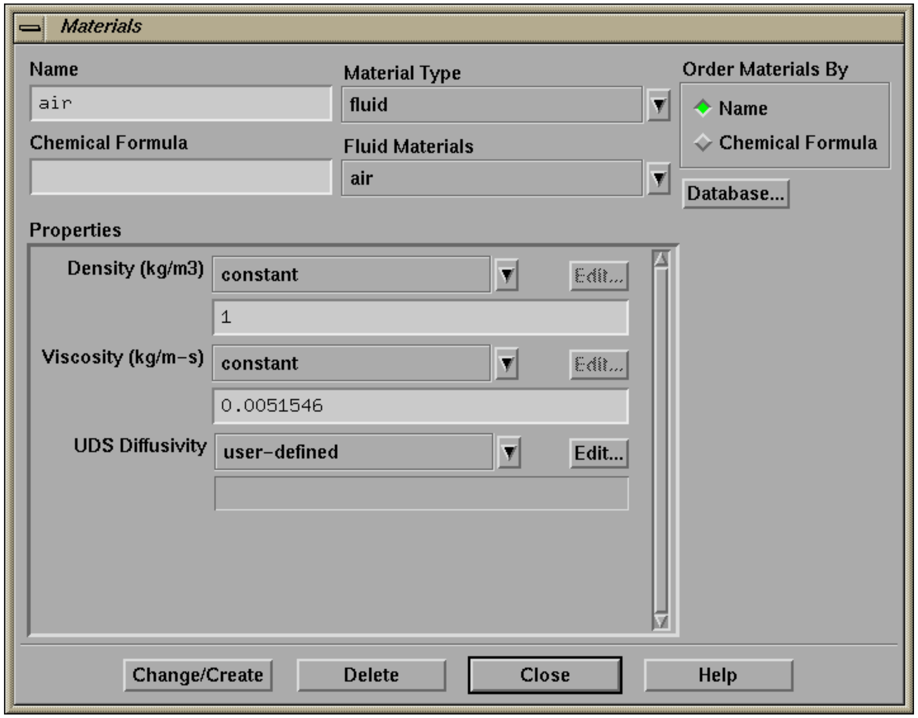
2) Define the diffusion and source terms in the generic equation

$$\frac{\partial \phi}{\partial t} + U_j \frac{\partial \phi}{\partial x_j} = \frac{\partial}{\partial x_j}\left(D_\phi \frac{\partial U_i}{\partial x_j}\right) + S_\phi$$

User Defined Scalar

When UDS are defined in the material property panel the diffusivity of the scalars MUST be defined

Define → Material Property



Name: air

Chemical Formula:

Material Type: fluid

Fluid Materials: air

Order Materials By: ◆ Name ◇ Chemical Formula

Density (kg/m3): constant, 1

Viscosity (kg/m-s): constant, 0.0051546

UDS Diffusivity: user-defined

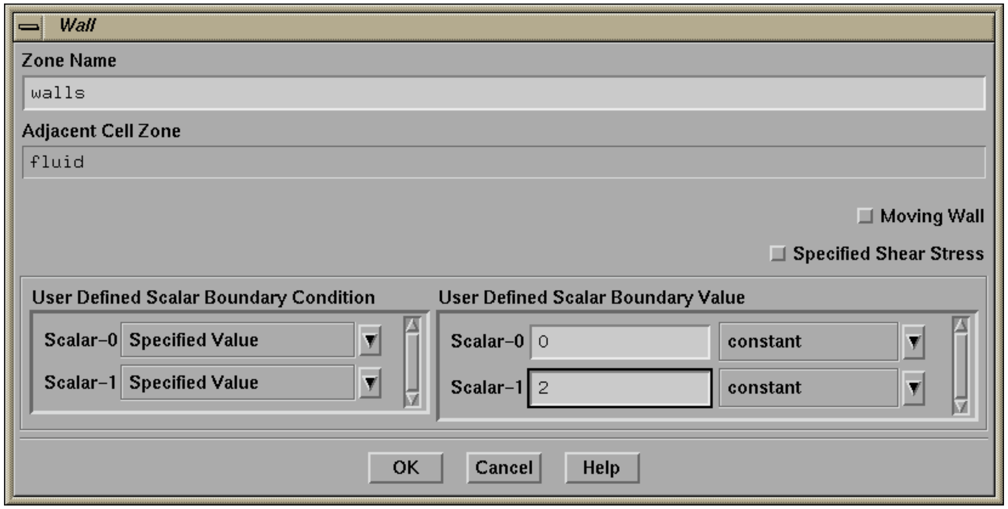
Note that the default diffusivity for the scalars is constant and equal to 1!

User Defined Scalar

Boundary conditions for the scalars can be defined as

Constant Value
Constant Gradient
User Defined

Define → Boundary Conditions → Wall



Zone Name
walls

Adjacent Cell Zone
fluid

☐ Moving Wall

☐ Specified Shear Stress

User Defined Scalar Boundary Condition

Scalar–0	Specified Value
Scalar–1	Specified Value

User Defined Scalar Boundary Value

Scalar–0	0	constant
Scalar–1	2	constant

User Defined Scalar

The Source terms for the scalars are set using the DEFINE_SOURCE macro Introduced before

The scalar equations can be further customized

1) Convective terms can be modified using the macro
   DEFINE_UDS_FLUX

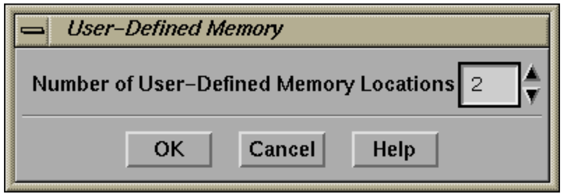
2) Unsteady term can be modified using the macro
   DEFINE_UDS_UNSTEADY

User Defined Memory

The User Defined Scalars are used to solve additional equations eventually coupled to the mean flow equations

Sometimes it is useful to define temporary field variables to store and retrieve values not directly available

UDM are defined: Define → User Defined → User Defined Memory



User-Defined Memory

Number of User-Defined Memory Locations 2

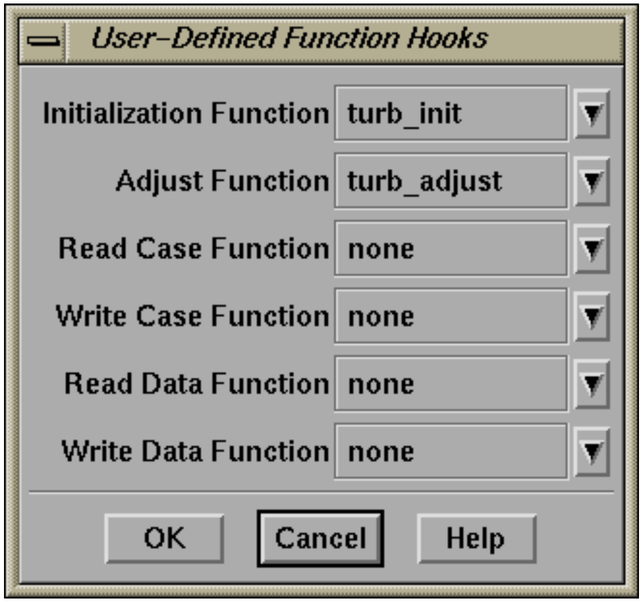
More efficient storage compared to User Defined Scalars

I/O in UDF

User Defined Scalars and User Defined Memory are AUTOMATICALLY Stored in the Fluent Data files

Additional I/O (from and to the Fluent Case and Data files) can be Accomplished using the macro DEFINE_RW_FILE

Define → User Defined → Function Hooks



Initialization Function	turb_init
Adjust Function	turb_adjust
Read Case Function	none
Write Case Function	none
Read Data Function	none
Write Data Function	none

Detailed Programming

The macros introduced before are interpreted/compiled and attached to the various Fluent panels

The detailed programming is based on additional macros that allow to loop on cells to retrieve field variables, etc.

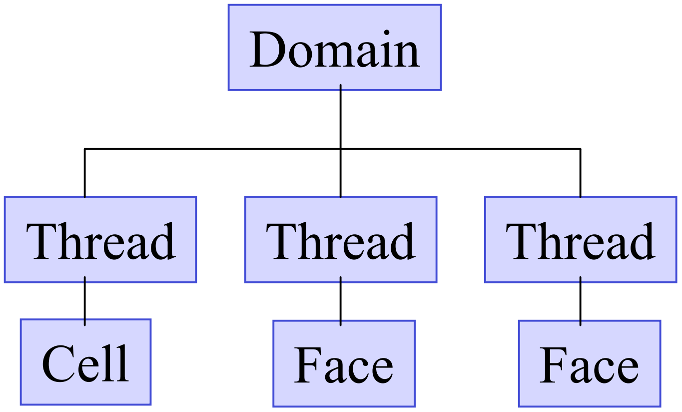
Loop Macros
Geometry Macros
Field Variable Macros
Control Macros
Arithmetic and Trigonometric Macros

Before looking at the Macros, the Fluent Data structure in introduced

Data Structure

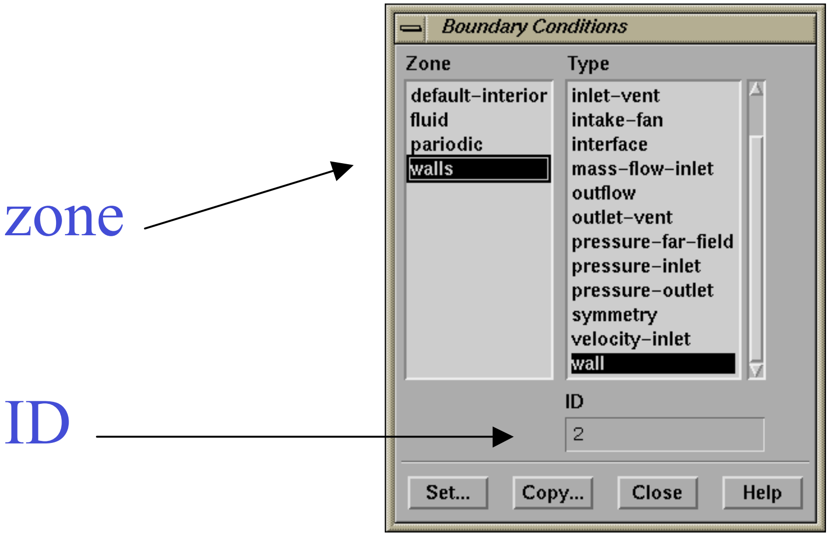
It is based on a hierarchy of structures

Threads (zones) are collection of cells or faces
Domain is a collections of threads



Threads

Every zone is associated to a single ID (available in the boundary conditions menu)



zone

ID

Given the ID of a thread the pointer can be retrieved as:

```
Thread *tf = Lookup_Thread(domain, ID);
```

Given the thread pointer the ID of the zone can be retrieved as:

```
ID = THREAD_ID(thread);
```

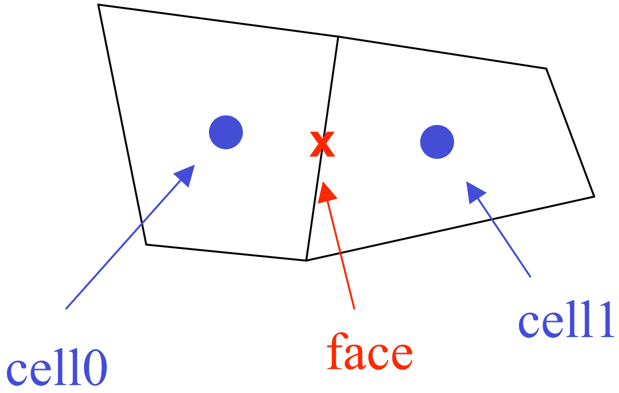
Loop Macros

`thread_loop_c(t, d);`	Loop over cell threads
`thread_loop_f(t, d);`	Loop over face threads
`begin_c_loop(c, t);` `end_c_loop(c, t);`	Loop over cells in a cell thread
`begin_f_loop` `end_f_loop`	Loop over faces in a face thread
`f_edge_loop(f, t, n);`	
`f_node_loop(f, t, n);`	
`c_node_loop(c, t, n);`	
`c_face_loop(c, t, n);`	

Cell-face connectivity

When looping on faces the surrounding cells can be accessed using the macros:

```
cell0_thread = F_C0_THREAD(face,thread)
cell1_thread = F_C1_THREAD(face,thread)
cell0_id = F_C0(face,thread)
cell1_id = F_C1(face,thread)
```



When looping on cells adjacent faces and cells can be accessed using the macros:

```
for(nf=0;nf<C_NFACES(cell,thread);nf++)
 {  face_id       = C_FACE(cell,thread,nf);
    face_thread   = C_FACE_THREAD(cell,thread,nf);
 }
```

Number of faces of each cell is unknown

Geometry Macros

- `C_CENTROID(x, c, t);`
- `F_CENTROID(x, f, t);`
- `F_AREA(A, f, t);`
- `C_VOLUME(c, t);`
- `C_VOLUME_2D(c, t);`
- `C_NNODES(c, t);`
- `C_NFACES(c, t);`
- `F_NNODES(f, t);`

- x, y, z-coordinates of cell centroids
- These definitions are in `metric.h` and `mem.h` files.
- No of nodes in a cell
- No. of faces in a cell
- No. of nodes in a face

Many more available; refer to the FLUENT UDF Manual

Field Variables Macros

`C_R(c,t);`	`(mem.h)` density
`C_P(c,t)`	Pressure
`C_U(c,t)`	Velocity components
`C_V(c,t)`	
`C_W(c,t)`	
`C_T(c,t)`	Temperature
`C_H(c,t)`	Enthalpy
`C_K(c,t)`	Turbulent kinetic energy
`C_D(c,t)`	Turbulent energy dissipation
`C_YI(c,t,i)`	Species mass fraction
`C_UDSI(c,t,i)`	User defined scalar

Field Variables Macros

`C_DUDX(c,t)`	Velocity derivatives
`C_DUDY(c,t)`	
`C_DUDZ(c,t)`	
`C_DVDX(c,t)`	
`C_DVDY(c,t)`	
`C_DVDZ(c,t)`	
`C_DWDX(c,t)`	
`C_DWDY(c,t)`	
`C_DWDZ(c,t)`	
`C_MU_L(c,t)`	Viscosities
`C_MU_T(c,t)`	
`C_MU_EFF(c,t)`	
`C_K_L(c,t)`	Thermal conductivity

Many more available; refer to the FLUENT UDF Manual

Control Macros

- `boolean Data_Valid_P()`
- Equals 1 if data is available, 0 if not.

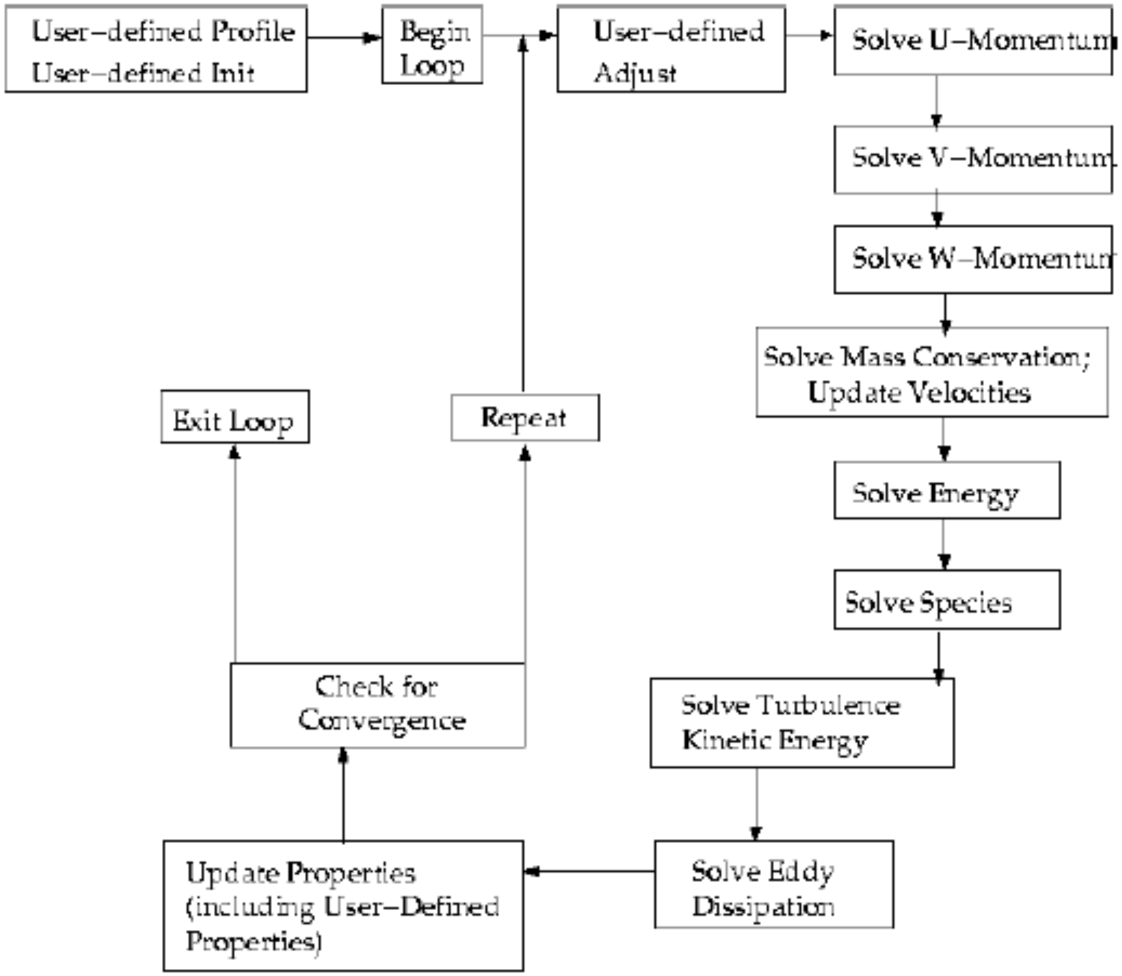
Usage:

```
if(!Data_Valid_P())
       return;
```

- `boolean FLUID_THREAD_P(t0)`
- Checks to see if thread `t0` fluid thread

- `NULLP(T_STORAGE_R_NV(t0, SV_UDSI_G(p1))`
- Checks for storage allocation of user defined scalars

Code FlowChart - Segregated Solver



User–defined Profile
User–defined Init

Begin Loop

User–defined Adjust

Solve U–Momentum

Solve V–Momentum

Solve W–Momentum

Solve Mass Conservation;
Update Velocities

Solve Energy

Solve Species

Solve Turbulence
Kinetic Energy

Solve Eddy
Dissipation

Update Properties
(including User–Defined
Properties)

Check for
Convergence

Exit Loop

Repeat

Additional Information

Many additional macros are available to implement different physical models

- combustion models
- particles-based models
- ….

It is formally possible to develop additional numerical methods

- flux discretizations
- variable reconstruction and clipping
- ….

UDF in other commercial CFD Codes

CFX

CFX (v4) is a structured code; the data structure is much simpler because the field variables are accessed directly. It uses 1D arrays where the quantities are stored in succession

CFX UDFs are written in FORTRAN

For example:

`U(I,J,K)` → `U(IJK)` where `IJK = (K-1)*NI*NJ+(J-1)*NI`

There are no macros, but examples of subroutines to perform the customization

```
USRBC
USRSRC
USRDIF
….
```

UDF in other commercial CFD Codes

Star-CD

StarCD is an unstructured code similar to Fluent in term of data organization
But similar to CFX for the organization of the UDF.

StarCD has threads (as Fluent) and the UDF work normally on a cell-by-cell basis

StarCD UDFs are written in FORTRAN

There are no macros, but examples of subroutines to perform the customization

```
BCDEFW
SORSCA
DIFFUS
```

….

Automation

CFD Solver Automation

- FLUENT can save a “journal” file

To start saving	File → Write → Start Journal
To stop saving	File → Write → Stop Journal

- Journal file are ASCII editable files

- Commands are somewhat “obscure” (keywords, etc.)

- They can be made general by introducing User Defined Parameters

FLUENT Journal Files

Fluent GUI is developed using a LISP dialect called SCHEME

To interact with Fluent Journal files via parameters it is necessary to use SCHEME commands.

For example to fix the under-relaxation in the momentum equation the command is:

```
solve/set/under-relaxation/momentum 0.7
```

Otherwise we can define a variable:

```
(define my_url 0.7)
solve/set/under-relaxation/momentum my_url
```

FLUENT Journal Files

With SCHEME you can perform standard operations:

```
(define my_new_url (+ my_url 0.1))
```

To interact with Fluent Journal files via parameters it is necessary to use SCHEME commands.

Loops and conditional statements

```
(if (> 3 2) 'yes 'no)
```

FLUENT Batch Execution

Batch NO GUI: `fluent 2ddp -g -i <journalfile.jou>`
Batch with GUI: `fluent 2ddp  -i <journalfile.jou>`

Unfortunately journal files automatically saved by FLUENT contain commands that operate directly on the GUI:

```
(cx-do-gui cx-activate-item "MenuBar*WriteSubMenu*Stop Journal")
```

Therefore even for batch executions the GUI has to be activated to use journal files!

The other option is to generate the command files directly using the text-command
And NOT the GUI!!!

Many examples on the Web site of journal files that can be run in batch

FLUENT Example of Scheme file for Adaptation

```
;;;
(custom-field-function/define ,….) …    ! Define Adaptation Function
;;;
;;; set few parameters
;;;
(define maxcells 200000) ….             ! Set Control Parameters
(define minref .1)
;;;
(do ((j 0)                               ! Adaptation Loop
  (i 0 (+ i 1)))
  ((= i 5) j)
  (format "\n Iteration ~a step - STARTED   " i)
;;;
(cx-gui-do cx-activate-item …) …          ! Display Adaptation function
;;;
(cx-gui-do cx-activate-item "MenuBar*AdaptMenu*Iso-Value...") …  ! Adapt
;;;
(cx-gui-do cx-activate-item "MenuBar*SolveMenu*Iterate...") … ! Iterate

;;;
 (format "\n Iteration ~a step - FINISHED \n" i)
)                                          ! End Adaptation Loop
```

Available as adapt.scm from the Website

Example of Automatic Adaptation

Driven Cavity Problem

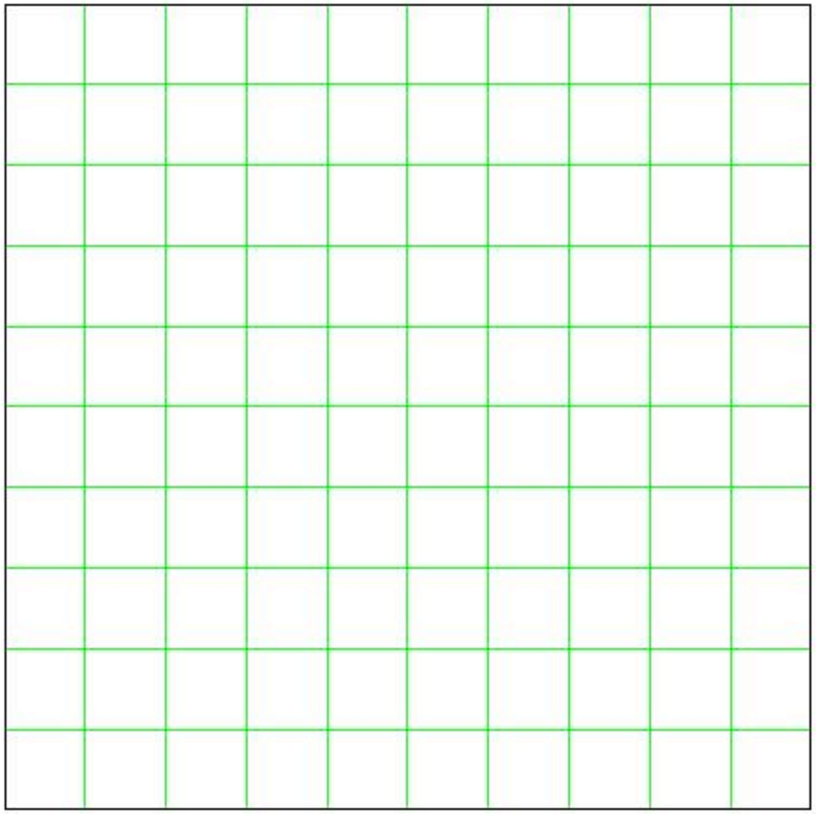
Start Initial Conditions - Uniform Grid

Perform 5 Steps of Adaptation Using Hanging Nodes

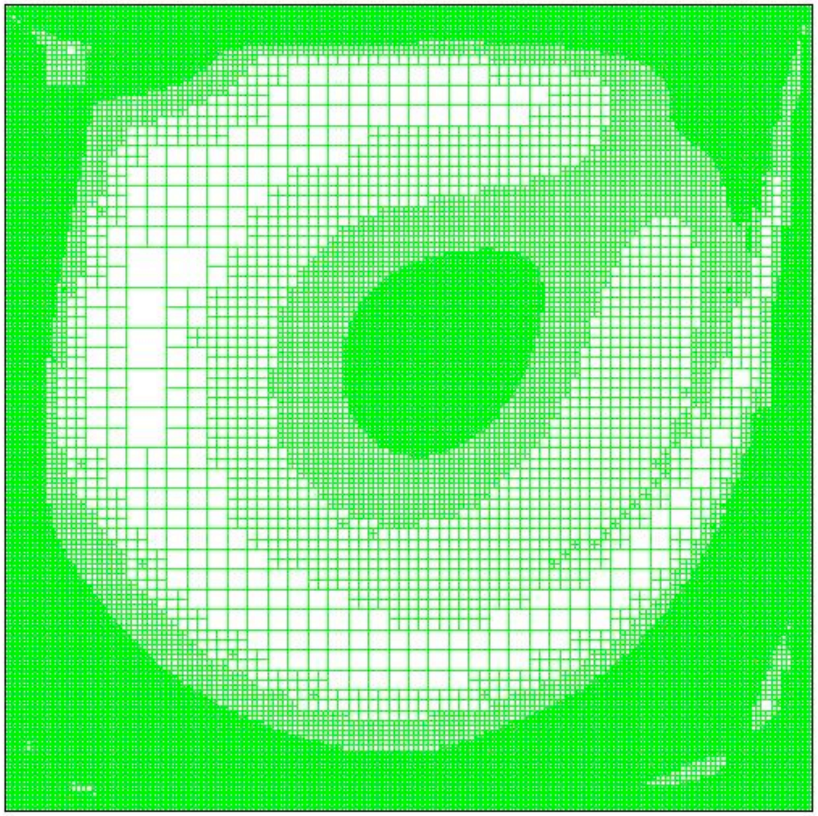
Adaptation Function is:

Scaled Velocity Difference = (Velocity Gradient * Volume$^{1/3}$)/Velocity

Example of Automatic Adaptation

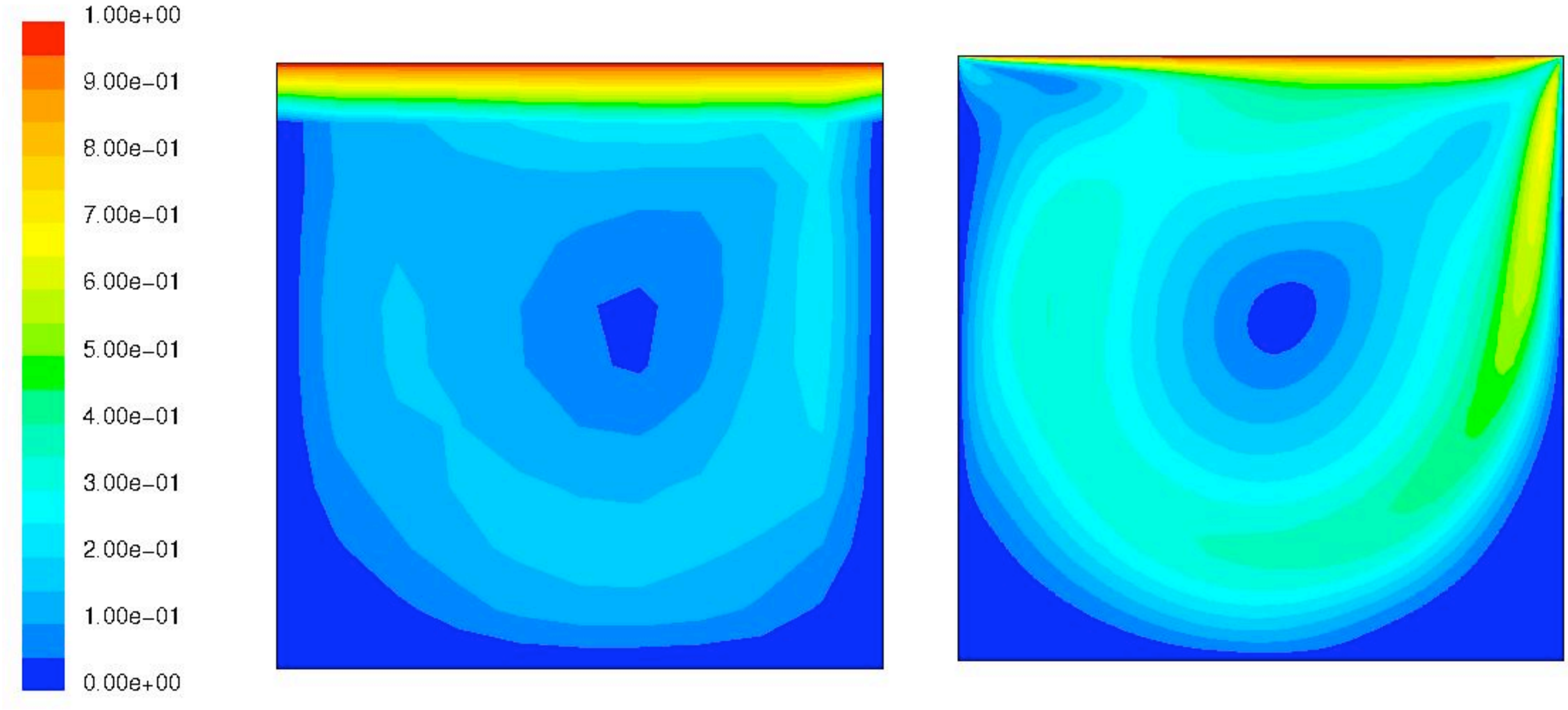


Initial Grid: 100 elements



Final Grid: 40,000 elements

Example of Automatic Adaptation



Initial Grid: 100 elements

Final Grid: 40,000 elements

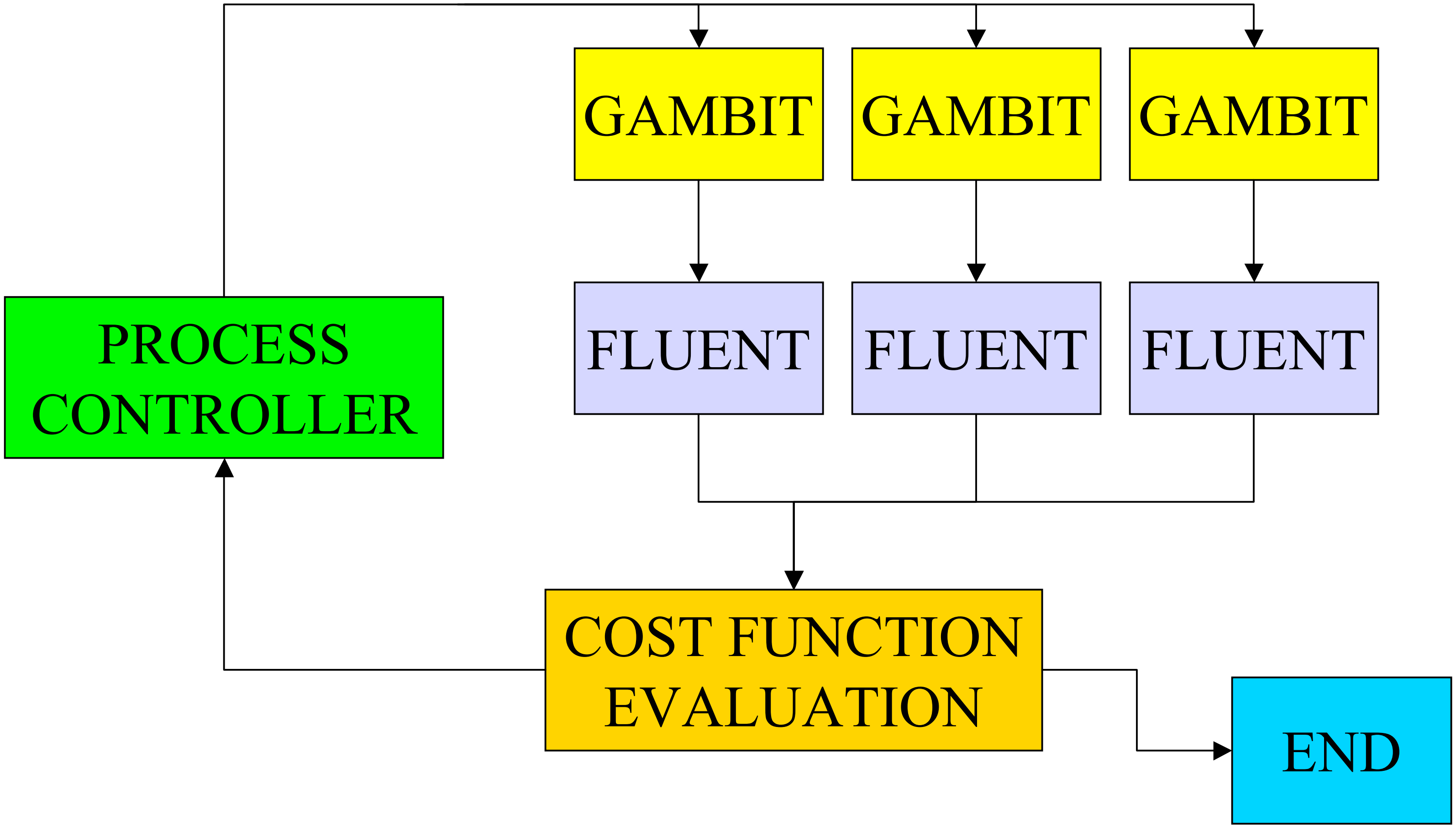
Optimization

Objective: Automatically find the best solution according to a certain goal (cost function)

Approach: Requires several “similar” CFD simulations obtained by varying one or few parameters (geometry, flow conditions, etc.)

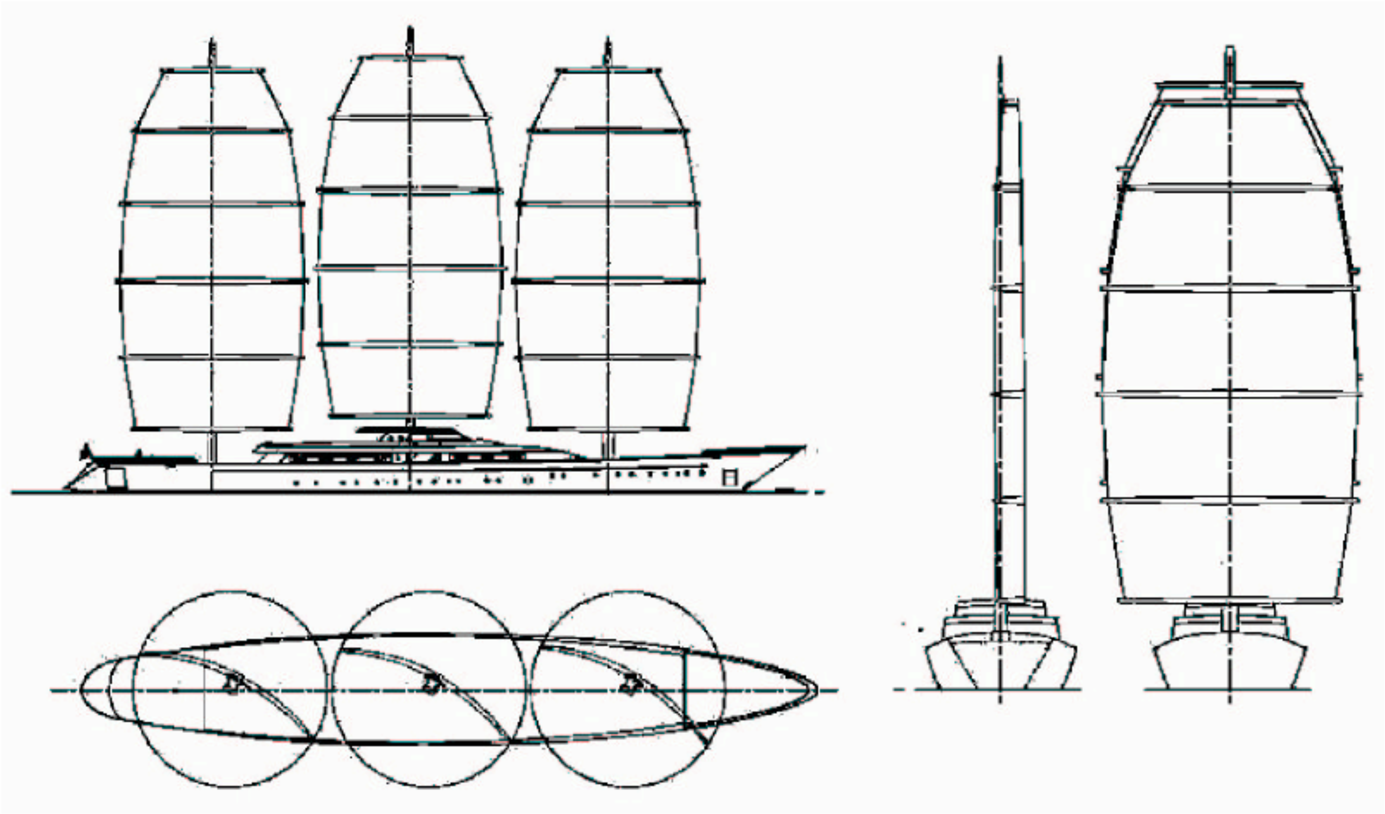
Scripts allow to perform this task very easily!

Optimization Procedure



Optimization Procedure Example

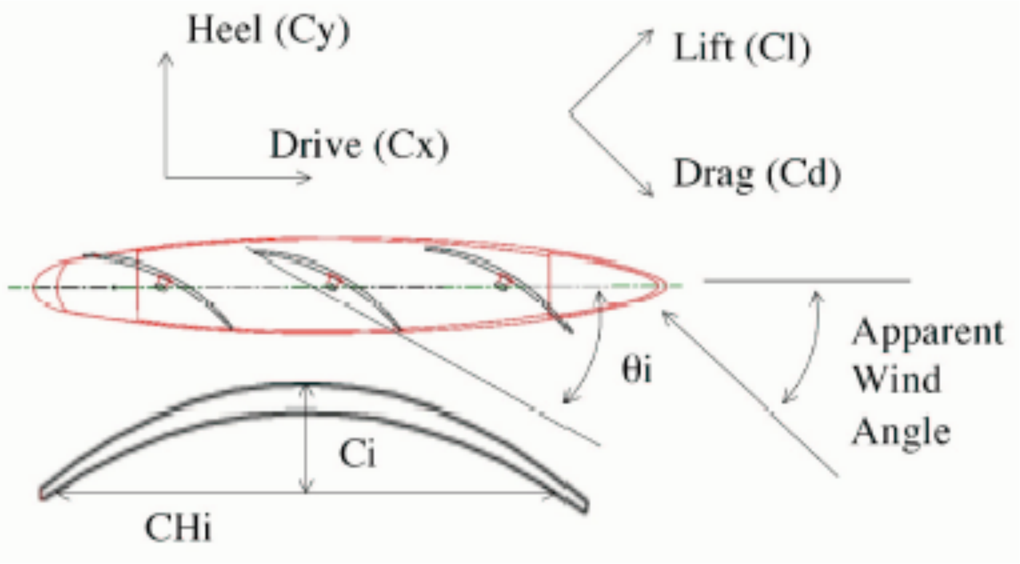
Design and Trim Sails for a Modern Clippper Ship



By T. Doyle

Optimization Procedure Example

Design and Trim Sails for a Modern Clipper Ship

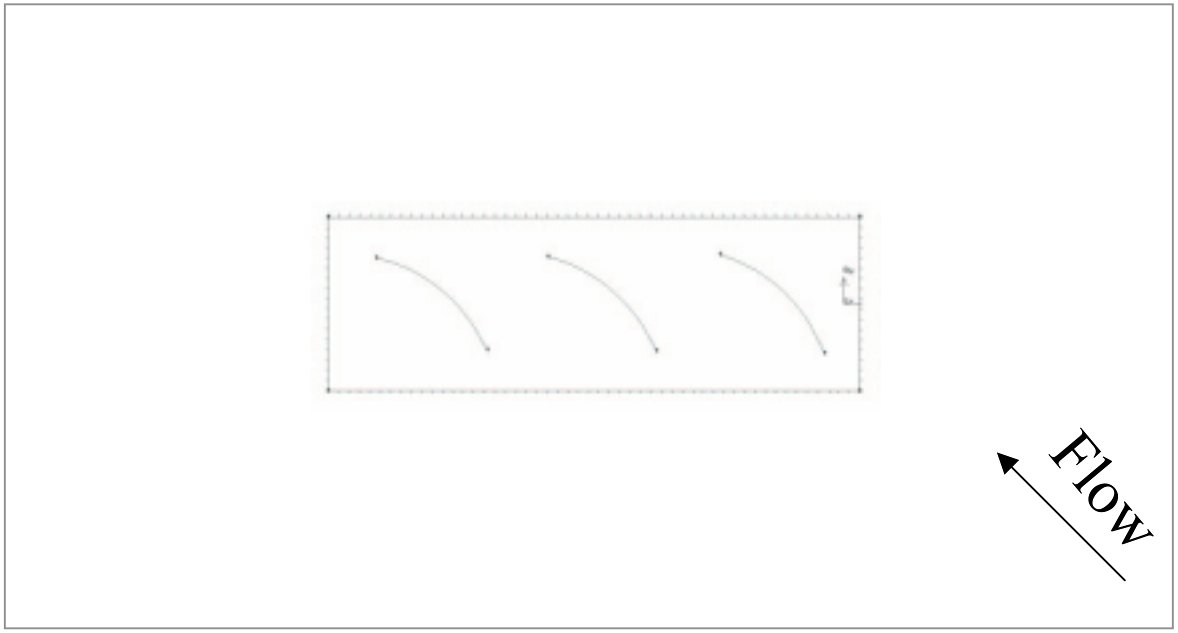


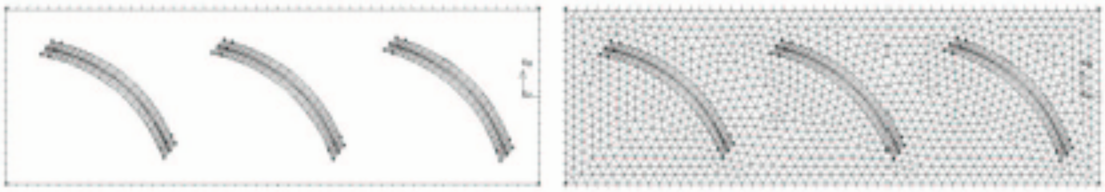
Aerodynamic
Force Definitions

One cost function is defined as Lift/Drag

Optimization Procedure Example

Design and Trim Sails for a Modern Clipper Ship

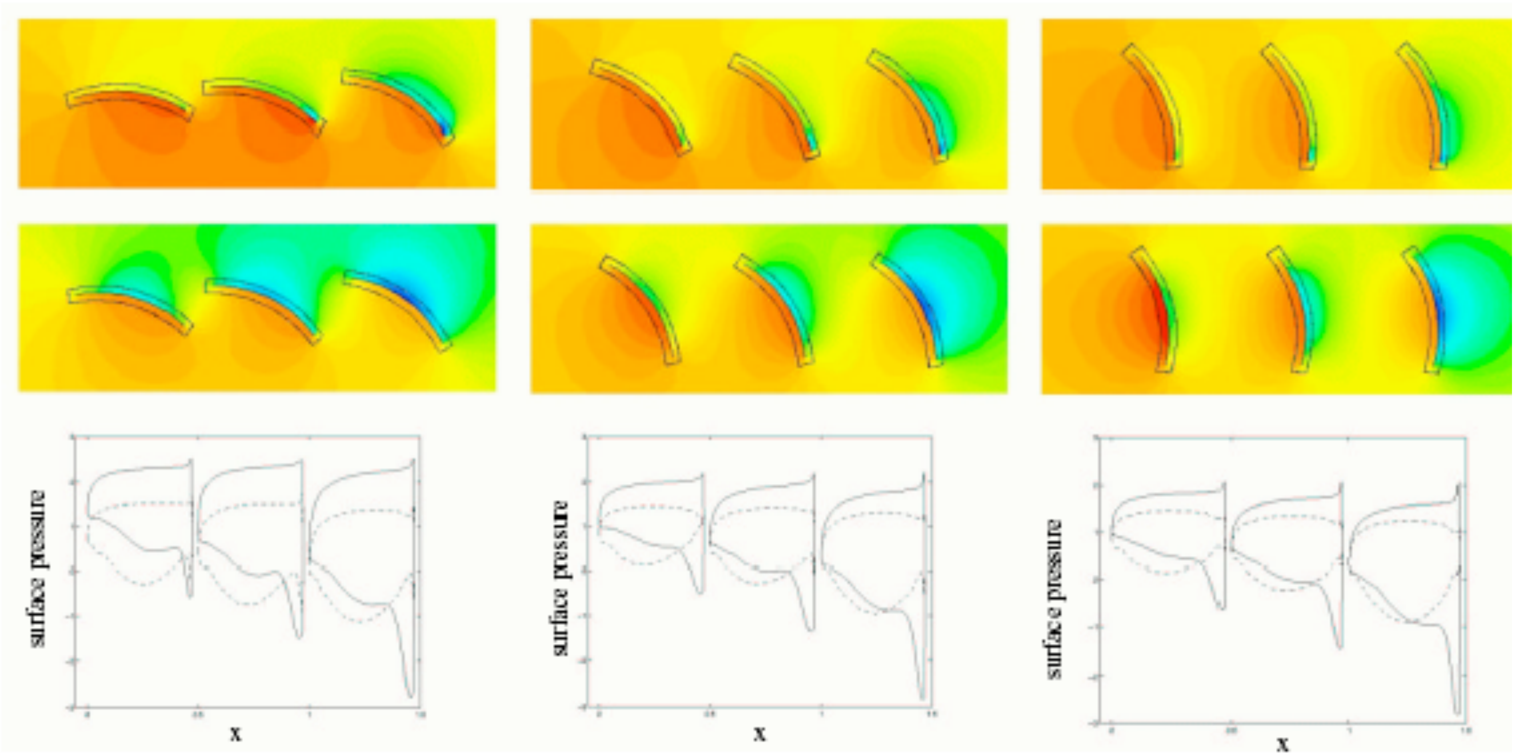




Grid Generation

Optimization Procedure Example

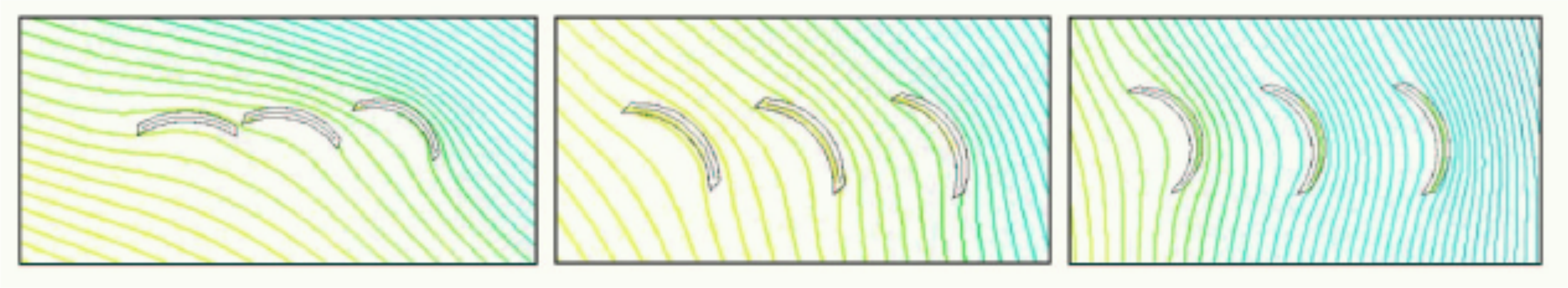
Design and Trim Sails for a Modern Clipper Ship



Flow Solutions

Optimization Procedure Example

Design and Trim Sails for a Modern Clipper Ship



Optimized Solution for Different Wind Directions