SimpleCFD

Simon Lee

May 31, 2016

Contents

1 Todos

Todo list

2 Introduction

I have long had the idea to write my own, but very simple, CFD solver.

My initial goal is the solve for the case of a flow through a rectangular box/duct. The only reason for this is that the grid/mesh should be very easy.

"main.c" 3≡

```
#include "mesh.h"
#include <stdio.h>
int main(void){
    struct_mesh mesh;
    generate_mesh(&mesh);
    mesh_print_vtk(&mesh,"test.vtk");
    printf("Hello World %d %d\n",mesh.num_points, mesh.num_cells);
    return 0;
}
◇
```

3 mesh

3.1 Introduction

The code only works with rectagular cells in an unstructured grid. The mesh is defined by a point cloud and each cell has 8 nodes.

Each node is defined by a 3 dimensional vector.

3.1.1 Data Structure

⟨ *mesh-data-structures* 4 ⟩ ≡

```
typedef struct {
    int vtk_type;
    int *nodes;
} struct_cell;

typedef struct {
    int num_points;
    int num_cells;
    float scale;
    int **points;
    struct_cell *cells;
    int mem_allocated;
} struct_mesh;

typedef struct {
    int num_points;
    int num_cells;
    float scale;
    int **points;
    struct_cell *cells;
    int mem_allocated;
} struct_domain;
◇
```

Fragment referenced in 12a.

3.1.2 Mesh Definition

Mesh definition to be input at runtime.

⟨ *mesh-def* 5 ⟩ ≡

```
int domain_min[3];
int domain_max[3];

int domain_seedsize[3];

domain_min[0] = 0;
domain_min[1] = 0;
domain_min[2] = 0;

domain_max[0] = 20;
domain_max[1] = 10;
domain_max[2] = 10;

/* this is actually half delta */
domain_seedsize[0] = 5;
domain_seedsize[1] = 5;
domain_seedsize[2] = 5;

mesh->num_points = 0;
mesh->num_cells = 0;
mesh->scale = 1e-3;
```

◇

Fragment referenced in 8.

3.2 Generating Mesh

3.2.1 Generating Vertex List

⟨ *generate-mesh-vertex* 6 ⟩ ≡

```
/* Generate Vertices */
    p = 0;

    pos[0] = 0;
    pos[1] = 0;
    pos[2] = 0;

    while(pos[2] <= domain_max[2]){
        while(pos[1] <= domain_max[1]){
            while(pos[0] <= domain_max[0]){
                points[p][0] = pos[0];
                points[p][1] = pos[1];
                points[p][2] = pos[2];
                p++;

                /* Increment position */
                pos[0] += domain_seedsize[0];
                ++vertex_count[0];
            }
            pos[1] += domain_seedsize[1];
            ++vertex_count[1];
        }
        pos[2] += domain_seedsize[2];
        ++vertex_count[2];
    }
```
◇

Fragment referenced in 8.

⟨ *generate-mesh-cells* 7 ⟩ ≡

```
/* Create Cells */
    for(c=0; c < mesh->num_cells ; c++){
        cells[c].nodes = malloc(9*sizeof(int));
        cells[c].vtk_type = 11;

        n = 0;
        while(pos[2] <= domain_max[2]){
            while(pos[1] <= domain_max[1]){
                while(pos[0] <= domain_max[0]){
                    xindex = (int) (x-xmin)/(2*dx);
                    yindex = (int) (y-ymin)/(2*dy);
                    zindex = (int) (z-zmin)/(2*dz);
                    cells[c].nodes[n]  = zindex * (xcount+1) * (ycount*1);
                    cells[c].nodes[n] += yindex * (xcount+1);
                    cells[c].nodes[n] += xindex;
                    n++;
                }
            }
        }
    }
◇
```

Fragment referenced in 8.

⟨ *func-generate-mesh* 8 ⟩ ≡

```
int generate_mesh(struct_domain * domain, struct_mesh * mesh){
    int n,c,p;

    int pos[3];

    int xindex;
    int yindex;
    int zindex;

    int **points;
    struct_cell *cells;

    int vertex_count[0] = 0;
    int vertex_count[1] = 0;
    int vertex_count[2] = 0;

    ⟨ mesh-def 5 ⟩

    printf("mesh points %d\n", mesh->num_points);
    printf("mesh cells %d\n", mesh->num_cells);

/* Mememory Allocation */

    // Lets guess a mesh size...
    mesh->num_points = (vertex_count[0]+1)*(vertex_count[1]+1)*(vertex_count[2]+1)
    mesh->num_cells = xcount * ycount * zcount;

    //... and allocate it
    mesh->mem_allocated = 0;
    mesh_mem_allocate(mesh);


    ⟨ generate-mesh-vertex 6 ⟩

    ⟨ generate-mesh-cells 7 ⟩

    mesh->points = points;
    mesh->cells = cells;
    return 0;
}
```

◇

Fragment referenced in 12b.

3.2.2 Dynamic Memory Allocation

Memory Allocation Memory for the mesh get allocations based on the number of vertices and cells stored within the mesh data structure.

Reallocate memory for mesh

⟨ *mesh-mem-allocation* 9a ⟩ ≡

```
/* Allocate memory for mesh */
int mesh_mem_allocate(struct_mesh * mesh){

    if(mesh->mem_allocated ==0){
        /* Allocate Memeory for Points */
        mesh->points = malloc(mesh->num_points*sizeof(*points) + mesh->num_points*3*sizeof(**po

        int *data;
        data = &points[mesh->num_points];
        for(n=0; n < mesh->num_points; n++){
            points[n] = data + n * 3;
        }

        /* Allocate Memeory for Cells */
        cells = malloc(mesh->num_cells*sizeof(*cells));

        mesh->mem_allocated = 1;
    }
    else {
        /* Reallocate Memory */
    }

   return 0;
}
```

◇

Fragment referenced in 12b.

Free Memory

⟨ *mesh-mem-free* 9b ⟩ ≡

```
/* Free memory allocated to mesh */
int mesh_mem_free(struct_mesh * mesh){

   free(mesh->points)
   free(mesh->cells)
   return 0;
}
```

◇

Fragment referenced in 12b.

3.3 Exporting Mesh

The mesh can be exported to a vtk file format to be read in Paraview.

⟨ *func-mesh-print-vtk* 10a ⟩ ≡

```
#Export mesh to VTK
int mesh_print_vtk(struct_mesh *mesh, char *filename) {
    int n;
    FILE * fp;
    fp = fopen(filename,"w");
```

⟨ *snippet-mesh-print-vtk-header* 10b ⟩

⟨ *snippt-mesh-print-vtk-vertices* 11a ⟩

⟨ *snippt-mesh-print-vtk-cells* 11b ⟩

```
    fclose(fp);

    return 0;
    }
```

◇

Fragment referenced in 12b.

3.3.1 VTK Header

It currently defaults to an unstructed grid

⟨ *snippet-mesh-print-vtk-header* 10b ⟩ ≡

```
/* Print file header */
    fprintf(fp, "# vtk DataFile Version 2.0\n"
                "SimpleCFD VTK Output\n"
                "ASCII\n"
                "DATASET UNSTRUCTURED_GRID\n");
```

◇

Fragment referenced in 10a.

3.3.2 Print Vertices

⟨ *snippt-mesh-print-vtk-vertices* 11a ⟩ ≡

```
/* Print Vertices */
    fprintf(fp, "\nPOINTS %d FLOAT\n", mesh->num_points);

    for(n=0; n < mesh->num_points; n++){
        fprintf(fp, "%+e %+e %+e\n", (float) mesh->points[n][0] * mesh->scale,
                                     (float) mesh->points[n][1] * mesh->scale,
                                     (float) mesh->points[n][2] * mesh->scale);
    }
◇
```

Fragment referenced in 10a.

3.3.3 Printing Cells Information

Each cell is defined by 8 vertices. Type 12 refers to a VTK relationship between nodes.

⟨ *snippt-mesh-print-vtk-cells* 11b ⟩ ≡

```
/* Print Cell Information */
    fprintf(fp, "\nCELLS %d %d \n", mesh->num_cells, mesh->num_cells * 9);

    for(n=0; n < mesh->num_cells; n++){
        fprintf(fp, "8 %d %d %d %d %d %d %d %d\n", mesh->cells[n].nodes[0],
                                                   mesh->cells[n].nodes[1],
                                                   mesh->cells[n].nodes[2],
                                                   mesh->cells[n].nodes[3],
                                                   mesh->cells[n].nodes[4],
                                                   mesh->cells[n].nodes[5],
                                                   mesh->cells[n].nodes[6],
                                                   mesh->cells[n].nodes[7]);
    }

/* Print Cell Types */
    fprintf(fp, "\nCELL_TYPES %d \n", mesh->num_cells);

    for(n=0; n < mesh->num_cells; n++){
        fprintf(fp, "12 \n");
    }
◇
```

Fragment referenced in 10a.

3.4 Source file

All the functions get written to a common source file.

"mesh.h" 12a≡

```
#ifndef MESH_H
#define MESH_H

⟨ mesh-data-structures 4 ⟩

int generate_mesh(struct_mesh * mesh);

int mesh_print_vtk(struct_mesh *mesh, char *filename);

#endif
```
◇

"mesh.c" 12b≡

```
#include "mesh.h"
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

⟨ mesh-mem-free 9b ⟩

⟨ mesh-mem-allocation 9a ⟩

⟨ func-generate-mesh 8 ⟩

⟨ func-mesh-print-vtk 10a ⟩
```
◇

4 preprocessor

⟨ *preprocessor.c* 12c ⟩ ≡

◇

Fragment never referenced.

5 postprocessor

⟨ *postprocessor.c* 13a ⟩ ≡

◇

Fragment never referenced.

6 Solver

⟨ *solver.c* 13b ⟩ ≡

◇

Fragment never referenced.

A Utilities

To automate working with SimpleCFD , a few utilities have been created.

SimpleCFD uses GNU Make to generate all the required files. It is used to produce to produce the document and source files and compile as required.

A.1 Root Makefile

The makefile that sits in the root directory of the project has targets to allow it to compile the source, documentation and clean temporary files. By default it outputs the source code.

"../Makefile" 13c≡

```
#Define phony targets
.PHONY: all doc

# Output the Source Code
all: SimpleCFD.w
        ⟨ make-src 14a ⟩

# Make the documents
doc: SimpleCFD.w
        ⟨ make-doc 14b ⟩
```

◇

A.1.1 Source Output

The all target 'TANGLES' the source text into the src folder with nuweb and executes make within the source folder. It also explicitly scaffolds folders as necessary.

⟨ *make-src* 14a ⟩ ≡

```
mkdir -p src
nuweb -t -p src $^
$(MAKE) -C src
```

◇

Fragment referenced in 13c.

A.1.2 Documentation Output

The documentation is produced by 'weaving' the source text. All the documentation output is generated into a separation folder.

⟨ *make-doc* 14b ⟩ ≡

```
mkdir -p doc
nuweb -o -p doc $^
pdflatex --output-directory doc $(basename $^).tex
```

◇

Fragment referenced in 13c.

Note: For all the cross references to be picked up correctly, this needs to be run at least twice.

B Building and Compilation of Source

The source code is compiled using GNU Make.

B.1 Compile Configurations

The configuration options of the complilation step sits in common folder to be sharded across make files.

"config.mk" 15a≡
```
# Common Configuration File

## Default Compilers
CXX=g++
CC=gcc


## Flags for the C compiler
CCFLAGS  = -g
CCFLAGS += -Wpointer-arith
CCFLAGS += -Wshadow -Winit-self
CCFLAGS += -Wextra
CCFLAGS += -Wfloat-equal
CCFLAGS += -Wall
CCFLAGS += -std=c99
CCFLAGS += -pedantic
CCFLAGS += -O3
```
◇

B.2 Comiling Source

"Makefile" 15b≡
```
#Makefile for Codebase

## Include the common configuration file
include config.mk

##Define phony targets
.PHONY: all clean

SOURCES=$(wildcard *.c)
OBJECTS=$(SOURCES:.c=.o)
TARGET=../simplecfd

all: $(TARGET)

$(TARGET): $(OBJECTS)
	$(CC) -o $@ $^

%.o: %.c
	$(CC) $(CCFLAGS) -c $<
```
◇