SHIMMER - Update

DISMA - PoliTO

February 19, 2024

SHIMMER project requirements (recall)

- General description (Page 122)
  - To expand the capabilities of the existent network models available within the consortium and to prepare them for open-source release.
  - Validation and benchmarking of the models against available data and commercial models
- Requirements for the open-source "model" **task 4.2.4** (Page 157)
  - Multi-component description of gas
  - High-pressure transmission networks
  - Highly meshed distribution networks
  - Non-pipe elements
- Schedule (Page 135, Fig 6)
  - WP4.2 runs from 1st year to the middle of the last one.
- Deliverable (Page 147)
  - Open-source fluid-dynamic "model" with gas quality tracking with handbook and tutorials (Table 7)
  - Due date MS8 Open-source "model" validated month 18 (Table 8)

Publications needed

Introduction

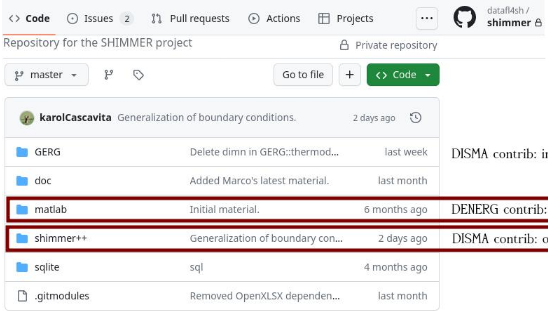
- Develop an efficient open-source library in C++ for the numerical modelling of natural gas transport through a long-distance network.
- Additional requirements:
  - Steady and unsteady state simulations
  - Complex mixture of gases: hydrogen blending
- Previous open-source tools in the literature:
  - *GasNetSim* [1]: Steady/Unsteady-regime. Complex mixture composition of gases. Python.
  - *Pandapipes* [2]: Steady and quasi-steady regimes in pipes. Gas mixture evaluated globally (no-varying nodal gas prop). Python.
  - *MORGEN*: Research software for Model Order Reduction order. Matlab.

[1] Y. Lu, T. Pesch and A. Benigni, "GasNetSim: An Open-Source Package for Gas Network Simulation with Complex Gas Mixture Compositions," 2022 Open Source Modelling and Simulation of Energy Systems (OSMSES), Aachen, Germany, 2022.

[2] Lohmeier D, Cronbach D, Drauz SR, Braun M, Kneiske TM. Pandapipes: An Open-Source Piping Grid Calculation Package for Multi-Energy Grid Simulations. Sustainability. 2020; 12(23):9899

Repository

- Repository in `https://github.com/datafl4sh/shimmer`
  - Matlab code by DENERG
  - C++ library developed by DISMA
  - Tracking of the development campaign in devoted Issues
  - Documentation of thesis, slides and shimmer project docs



DISMA contrib: interface between codes

DENERG contrib: Marco's code

DISMA contrib: open-source lib

Work advancement

Where we are

System boundaries

- Efforts in the in-memory representation and in the numerical methods stages.

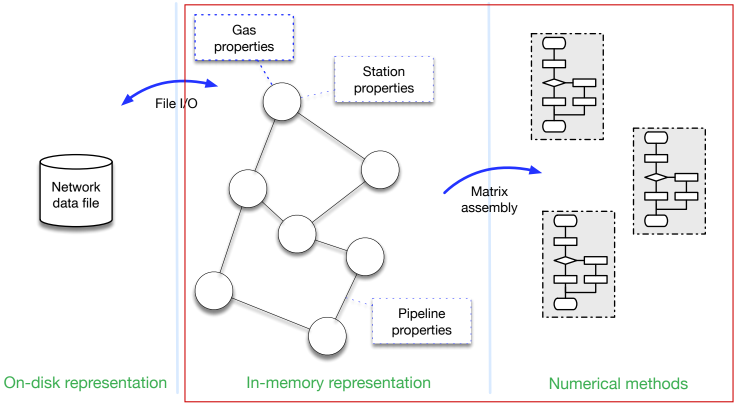


Figure 1: Taken from architecture proposal back in September.

System boundaries

- Efforts in the in-memory representation and in the numerical methods stages.

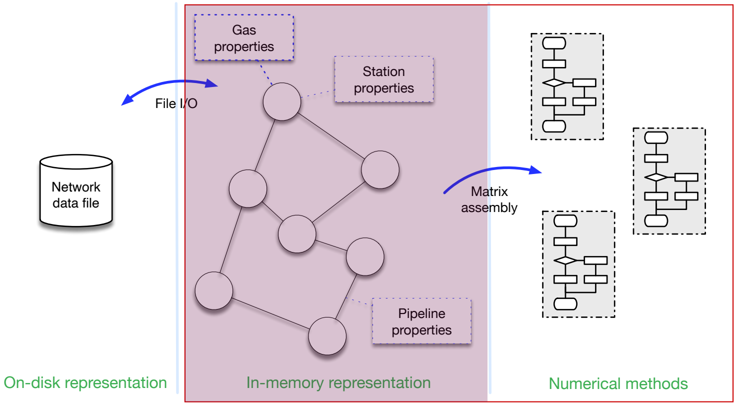


Figure 1: Taken from architecture proposal back in September.

In-memory representation stage

- We use a Boost Graph Library: generic programming, open-source and header-only library

```
adjacency_list<OutEdgeList, VertexList, Directed,
               VertexProperties, EdgeProperties, GraphProperties, EdgeList>
```

- We use an undirected graph with values in pipes/nodes assigned via the graph properties

```
using namespace boost;
using graph_type = adjacency_list< listS, vecS,
                                   undirectedS,
            vertex_properties, edge_properties>;

graph_type graph;
```

```
enum class edge_type {
    pipe,
    resistor,
    compressor,
    regulator,
    valve,
};
```

```
struct vertex_properties {
    std::string     name;
    int             number;
    double          height;
    vector_t        gas_mixture;
};

struct edge_properties {
    edge_type   type;
    int         number;
    double      length;
    double      diameter;
    double      friction_factor;
};
```

- Matrices directly built from graph: Incidence matrix, Resistance matrix, Inertia term

Numerical methods stage

- Efforts in the in-memory representation and in the numerical methods stages.

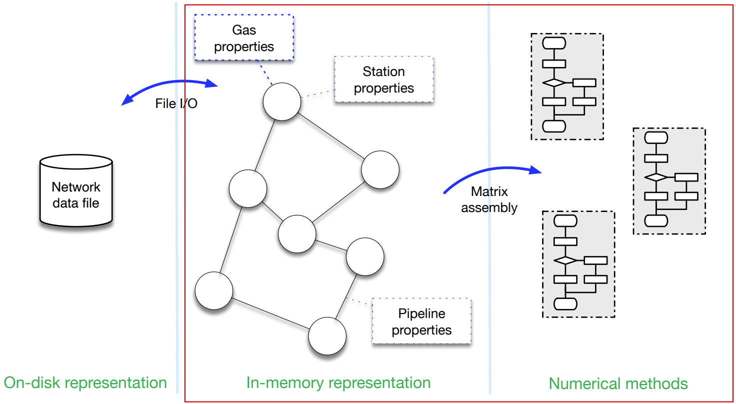


Figure 2: Taken from architecture proposal (Matteo's presentation, back in September).

Numerical methods stage

- Efforts in the in-memory representation and in the numerical methods stages.

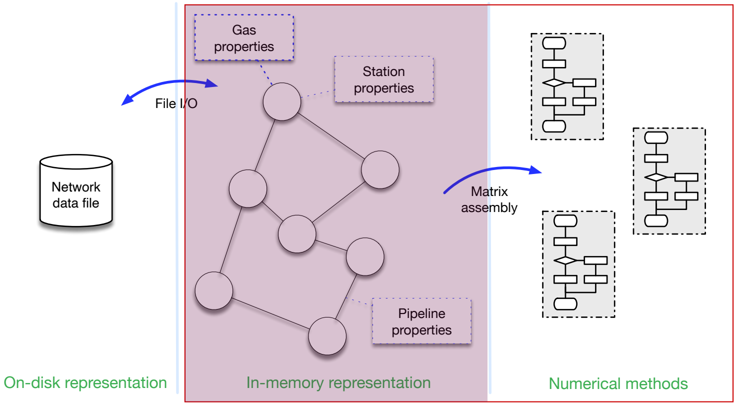


Figure 2: Taken from architecture proposal (Matteo's presentation, back in September).

Numerical methods stage

- Efforts in the in-memory representation and in the numerical methods stages.

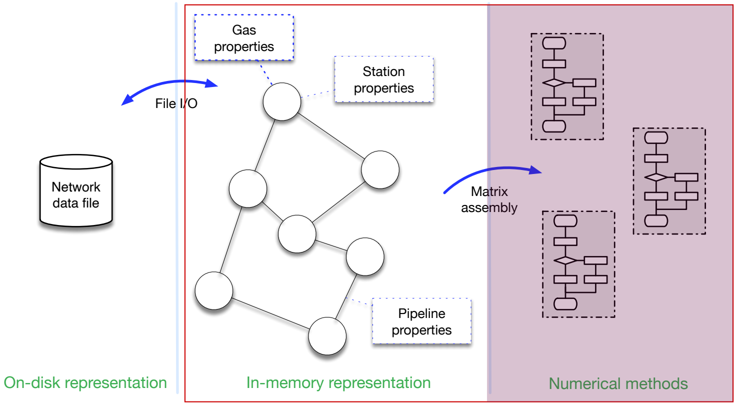
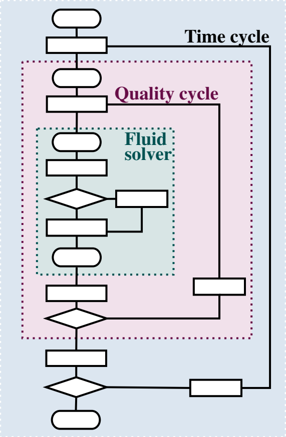


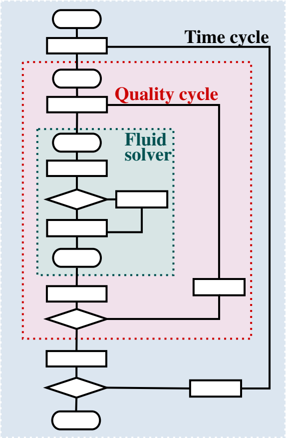
Figure 2: Taken from architecture proposal (Matteo's presentation, back in September).

Numerical methods stage



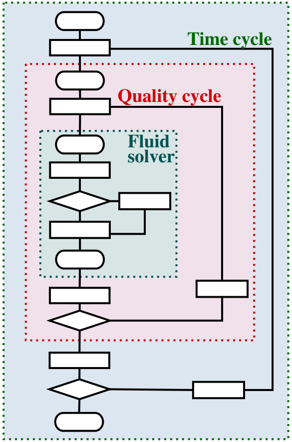
- Linearized Fluid Solver
  - Computation of the equation of state (GERG)
  - Friction factor average computation
    - Viscosity calculator
    - Infrastructure for complex gas composition
  - Matrix system of the iterative solver
    - Incidence matrix and its modified version referred to pressures
    - Resistance matrix
    - Φ matrix
    - Boundary condition treatment
- Quality Tracking Cycle
- Time Cycle
- Initialization

Numerical methods stage



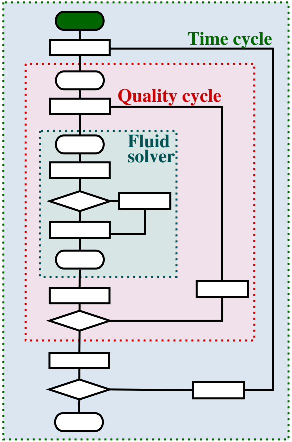
- Linearized Fluid Solver
  - Computation of the equation of state (GERG)
  - Friction factor average computation
    - Viscosity calculator
    - Infrastructure for complex gas composition
  - Matrix system of the iterative solver
    - Incidence matrix and its modified version referred to pressures
    - Resistance matrix
    - Φ matrix
    - Boundary condition treatment
- Quality Tracking Cycle
- Time Cycle
- Initialization

Numerical methods stage



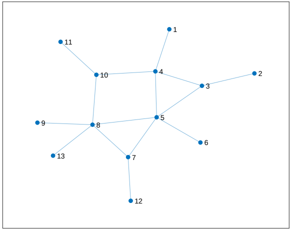
- Linearized Fluid Solver
  - Computation of the equation of state (GERG)
  - Friction factor average computation
    - Viscosity calculator
    - Infrastructure for complex gas composition
  - Matrix system of the iterative solver
    - Incidence matrix and its modified version referred to pressures
    - Resistance matrix
    - Φ matrix
    - Boundary condition treatment
- Quality Tracking Cycle
- Time Cycle
- Initialization

Numerical methods stage



- Linearized Fluid Solver
  - Computation of the equation of state (GERG)
  - Friction factor average computation
    - Viscosity calculator
    - Infrastructure for complex gas composition
  - Matrix system of the iterative solver
    - Incidence matrix and its modified version referred to pressures
    - Resistance matrix
    - Φ matrix
    - Boundary condition treatment
- Quality Tracking Cycle
- Time Cycle
- Initialization

Validation



Simple test case:

- Simple equation of state
- Mono-component gas
- Small amount (but different) of nodes and pipes
- Pressure inlet and flux output vary in time

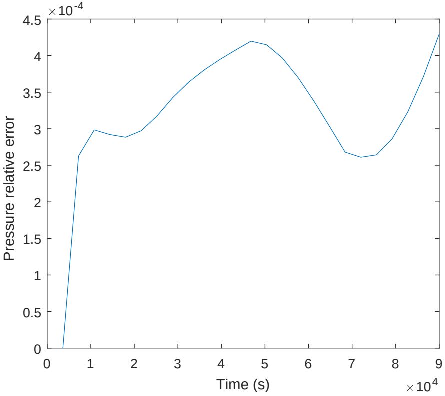
Complex test case: to be done

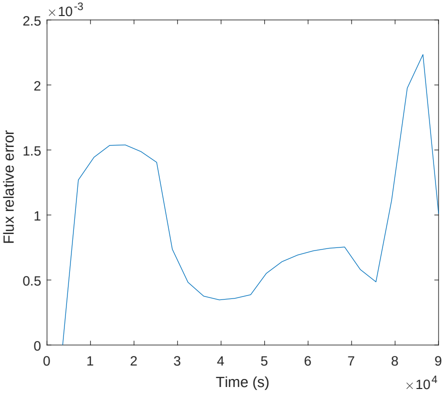
- Complex equation of state with complex gas mixture (GERG)
- Large gas network

```
time_solver<papay> ts(graph, temperature, Pset, flux_ext, inlet_nodes);
ts.initialization(Pguess, Gguess, Lguess);
ts.advance(dt, num_steps, tolerance, y_nodes, y_pipes);
```

Results for the simple test case

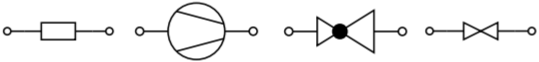
- Time advance:



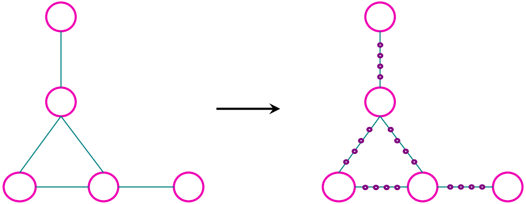


What shimmer++ does not

- Does not transform units. So input data must be given in SI units
- Non-pipe elements (yet)



- No refinement per pipe (grid creator)

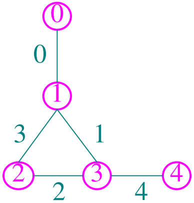


Process

How we did it

Graph initialization

- Graph representation and nodes specification



		Node properties			Mixture composition		
		G[kg/s]	p[Pa]	H[m]	CH4	N2	...
Nodes	0	5000	-60	10.0	-	-	-
	1	0	20	20.0	-	-	-
	2	0	25	30.0	-	-	-
	3	0	35	40.0	-	-	-
	4	0	50	50.0	-	-	-

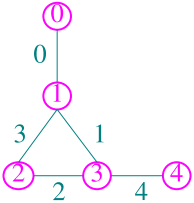
- Insertion of vertices using the *boost* :: *add_vertex* function

```
// Insert station config (name, no., pressure, flux, height)
auto v0 = add_vertex( { "Station 0", 0, 5e3,-60, 10}, graph) );
auto v1 = add_vertex( { "Station 1", 1,  0,  20, 20}, graph) );
auto v2 = add_vertex( { "Station 2", 2,  0,  25, 30}, graph) );
auto v3 = add_vertex( { "Station 3", 3,  0,  35, 40}, graph) );
auto v4 = add_vertex( { "Station 4", 4,  0,  50, 50}, graph) );

std::vector<vertex_descriptor> vds = {v0,v1,v2,v3,v4};
```

Graph initialization

- Graph representation and pipes specification



		Nodes		Pipe properties		
		In	Out	L[m]	D[m]	epsi[m]
Pipes	0	0	1	80.0	0.6	1.2e-5
	1	1	3	90.0	0.6	1.2e-5
	2	3	2	100.0	0.6	1.2e-5
	3	1	2	110.0	0.6	1.2e-5
	4	3	4	120.0	0.6	1.2e-5

- Insertion of pipes using the *boost* :: *add_edges* function

```
using pipe_t = edge_type::pipe;

add_edge( vds[0], vds[1], edge_properties({pipe_t, 0,  80, 0.6, 1.2e-5), graph});
add_edge( vds[1], vds[3], edge_properties({pipe_t, 1,  90, 0.6, 1.2e-5), graph});
add_edge( vds[3], vds[2], edge_properties({pipe_t, 2, 100, 0.6, 1.2e-5), graph});
add_edge( vds[1], vds[2], edge_properties({pipe_t, 3, 110, 0.6, 1.2e-5), graph});
add_edge( vds[3], vds[4], edge_properties({pipe_t, 4, 120, 0.6, 1.2e-5), graph});
```

Incidence matrix

- The incidence matrix establish the connection between nodes and pipes

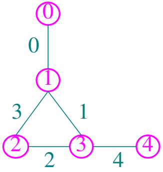
```cpp
class incidence
{
    sparse_matrix_t mat_;
    sparse_matrix_t mat_in_;
    sparse_matrix_t mat_out_;

    void
    compute(const graph_type& g);

public:

    incidence(){};
    incidence(const graph_type& g)
    {
       compute(g);
    };

    const sparse_matrix_t& mat();
    const sparse_matrix_t& mat_in();
    const sparse_matrix_t& mat_out();
};
```



		PIPES				
		0	1	2	3	4
NODES	0	1				
	1	-1	1		1	
	2			-1	-1	
	3		-1	1		1
	4					-1

```cpp
incidence::compute(const graph_type& g)
{
   auto range = edges(g);
   for(auto it = range.first;it! = range.second; it++){
       auto pipe = g[*it];
       auto node_in  = g[source(*it, g)];
       auto node_out = g[target(*it, g)];

       mat_in_( node_in.number,  pipe.number) = 1;
       mat_out_(node_out.number, pipe.number) = 1;
   }
  mat_ = mat_in_ - mat_out_;
}
```

Incidence matrix

- The incidence matrix establish the connection between nodes and pipes

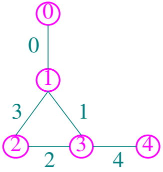
```
class incidence
{
    sparse_matrix_t mat_;
    sparse_matrix_t mat_in_;
    sparse_matrix_t mat_out_;

    void
    compute(const graph_type& g);

public:

    incidence(){};
    incidence(const graph_type& g)
    {
        compute(g);
    };

    const sparse_matrix_t& mat();
    const sparse_matrix_t& mat_in();
    const sparse_matrix_t& mat_out();
};
```



NODES \ PIPES	0	1	2	3	4
0	1				
1	-1	1		1	
2			-1	-1	
3		-1	1		1
4					-1

```
incidence::compute(const graph_type& g)
{
    auto range = edges(g);
    for(auto it = range.first;it! = range.second; it++){
        auto pipe = g[*it];
        auto node_in  = g[source(*it, g)];
        auto node_out = g[target(*it, g)];

        mat_in_( node_in.number,  pipe.number) = 1;
        mat_out_(node_out.number, pipe.number) = 1;
    }
   mat_ = mat_in_ - mat_out_;
}
```

Incidence matrix

- The incidence matrix establish the connection between nodes and pipes

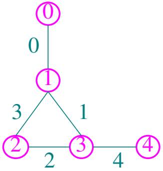
```cpp
class incidence
{
    sparse_matrix_t mat_;
    sparse_matrix_t mat_in_;
    sparse_matrix_t mat_out_;

    void
    compute(const graph_type& g);

public:

    incidence(){};
    incidence(const graph_type& g)
    {
        compute(g);
    };

    const sparse_matrix_t& mat();
    const sparse_matrix_t& mat_in();
    const sparse_matrix_t& mat_out();
};
```



		PIPES				
		0	1	2	3	4
NODES	0	1				
	1	-1	1		1	
	2			-1	-1	
	3		-1	1		1
	4					-1

```cpp
incidence::compute(const graph_type& g)
{
    auto range = edges(g);
    for(auto it = range.first;it! = range.second; it++){
        auto pipe = g[*it];
        auto node_in  = g[source(*it, g)];
        auto node_out = g[target(*it, g)];

        mat_in_( node_in.number,  pipe.number) = 1;
        mat_out_(node_out.number, pipe.number) = 1;
    }
   mat_ = mat_in_ - mat_out_;
}
```

Incidence matrix

- The incidence matrix establish the connection between nodes and pipes

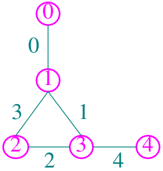
```
class incidence
{
    sparse_matrix_t mat_;
    sparse_matrix_t mat_in_;
    sparse_matrix_t mat_out_;

    void
    compute(const graph_type& g);

public:

    incidence(){};
    incidence(const graph_type& g)
    {
        compute(g);
    };

    const sparse_matrix_t& mat();
    const sparse_matrix_t& mat_in();
    const sparse_matrix_t& mat_out();
};
```



	PIPES 0	1	2	3	4
NODES 0	1				
1	-1	1		1	
2			-1	-1	
3		-1	1		1
4					-1

```
incidence::compute(const graph_type& g)
{
    auto range = edges(g);
    for(auto it = range.first;it! = range.second; it++){
        auto pipe = g[*it];
        auto node_in  = g[source(*it, g)];
        auto node_out = g[target(*it, g)];

        mat_in_( node_in.number,  pipe.number) = 1;
        mat_out_(node_out.number, pipe.number) = 1;
    }
    mat_ = mat_in_ - mat_out_;
}
```

Gas network model (recall)

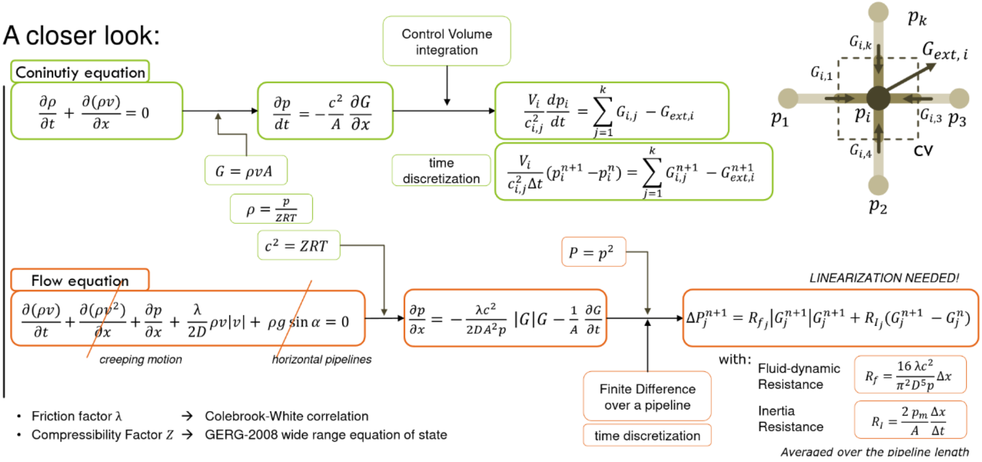


Figure 3: Taken from Marco's presentation.

Gas network model (recall)

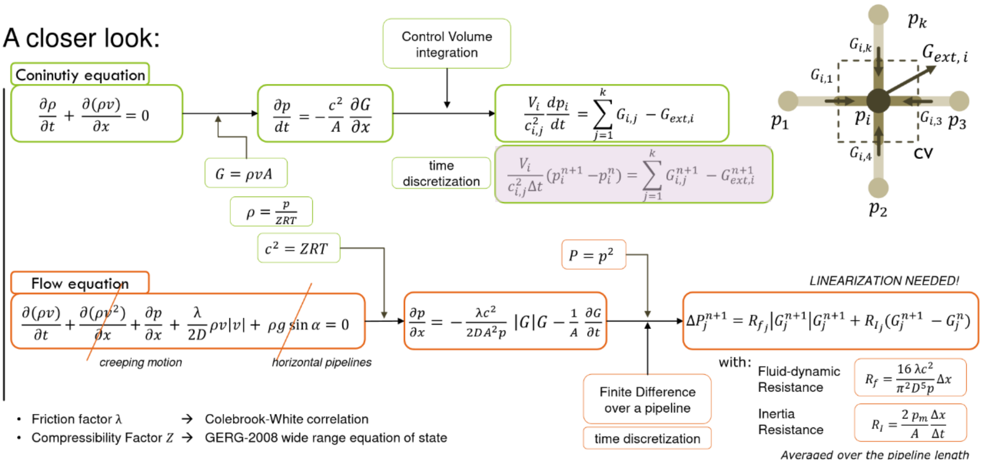


Figure 3: Taken from Marco's presentation.

Gas network model (recall)

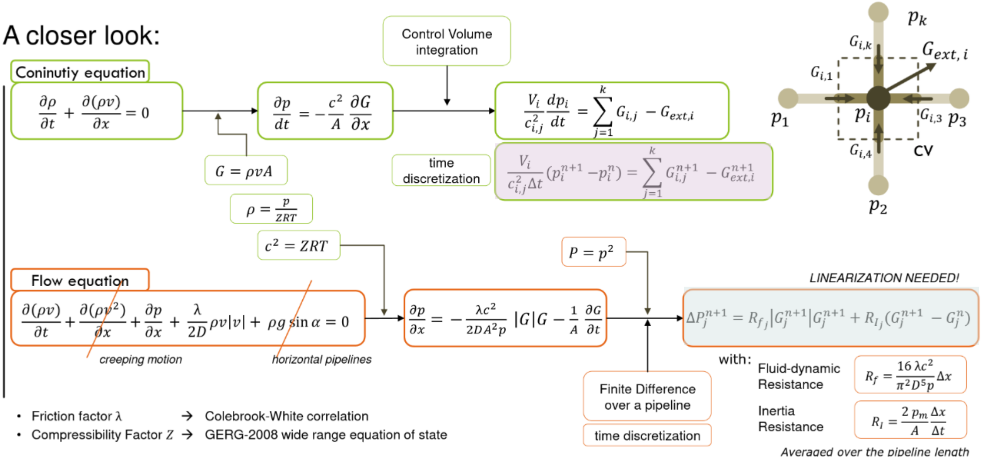


Figure 3: Taken from Marco's presentation.

Linearized fluid-dynamic solver

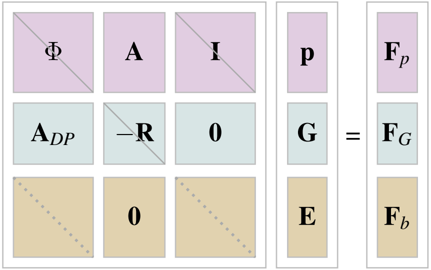
- Code implementation mimics the system

```
void linearized_fluid_solver::run(){
 for(size_t iter=0;iter<=MAX_ITERS;iter++){
    auto mass_system = continuity( ...);
    auto mom_system  = momentum( ...);
    auto bnd_system  = boundary(...);

    auto [LHS, rhs]= assemble(mass_system,
                              mom_system,
                      bnd_system, graph);

    solver.compute(LHS);
    vector_t sol = solver.solve(rhs);

    if(residual < tolerance)
       return;
 }}
```



Diagonal matrix

- Functions parameters:

```
     continuity( dt, Tm, pressure, pressure_old, incidence, graph, x_nodes, glaw_nodes);
     momentum( dt, Tm, flux, flux_old, pressure, incidence, graph, x_pipes, glaw_pipes);
     boundary(p_in, vel, flux_ext, incidence, graph, inlet_nodes);
```

Code good practices

- Modularity
- Flexibility
- Unitary testing

Code good practices

- Modularity: Not repeat chunks of code

```
template<typename EQ_OF_STATE>
    class time_solver
    {
    public:
        time_solver(const graph_type& g, ...);

        void  init(...){
            EQ_OF_STATE eos;
            fluid_solver lfs(false, tolerance, dt, Tm_, incidence_, graph);
            lfs.run(inlet_nodes_, Pset_(0), flux_ext_.row(0),  var_, &eos);
        }

        void  advance(...){
            for(size_t it = 1; it < num_steps; it++, t+=dt){
                EQ_OF_STATE eos;
                fluid_solver lfs(true, tolerance, dt, Tm_,incidence_, graph);
                lfs.run(inlet_nodes_, Pset_(it), flux_ext_.row(it),  var_, &eos);
            }
        }
    };
```

- Flexibility

Code good practices

- Modularity
- Flexibility: Use of different equations of state

```
time_solver<papay> ts(graph, temperature, Pset, flux_ext, inlet_nodes);
ts.initialization(Pguess, Gguess, Lguess);
ts.advance(dt, num_steps, tolerance, graph, y_nodes, y_pipes);
```

```
time_solver<gerg> ts(graph, temperature, Pset, flux_ext, inlet_nodes);
ts.initialization(Pguess, Gguess, Lguess);
ts.advance(dt, num_steps, tolerance, graph, y_nodes, y_pipes);
```

- Unitary testing

Code good practices

- Modularity
- Flexibility
- Unitary testing
  - Test incidence matrix
  - Test basic geometry computations
  - Test on ADP, R, Φ matrices
  - Test assembled system
  - Test Linearized fluid-dynamic solver

```cpp
int main(int argc, char **argv)
{
    using triple_t = std::array<double, 3>;
    using sparse_matrix_t = Eigen::SparseMatrix<double>;

    infrastructure_graph igraph;
    make_init_graph(igraph);

    incidence inc(igraph);
    const sparse_matrix_t& mat  = inc.matrix();

    std::cout << __FILE__ << std::endl;
    bool pass = verify_test("incidence matrix", mat, ref);

    return !pass;
}
```

- Purposes:
  - Check regression of the code to previous state when developing code
  - Some of them acts as validation tests cases against DENERG code
  - Intended also for future handbook/tutorials asked for the deliverable

Future and ongoing work

What is next

List of tasks

1. On-disk representation layer
   - DENERG: Iterate on the data model and finalize its design
   - DENERG: Provide formatted & cleaned-up data in Excel/CSV files
   - DISMA: Communication between On-disk representation and in-memory representation stages
2. Numerical methods layer
   - DENERG: Quality tracking needs review
   - DENERG: Validate the implementation
     - Complex test case with GERG, complex gas mixture, large network.
   - DISMA: GERG translation from Matlab to C++
   - DISMA: Boundary conditions

Thank you