Modelling Anaerobic Digesters in Three Dimensions: Integration of Biochemistry with Computational Fluid Dynamics

by

David L. F. Gaden

A Thesis submitted to the Faculty of Graduate Studies of

The University of Manitoba

in partial fulfilment of the requirements of the degree of

DOCTOR OF PHILOSOPHY

Department of Mechanical and Manufacturing Engineering

University of Manitoba

Winnipeg

Abstract

Anaerobic digestion is a process that simultaneously treats waste and produces renewable energy in the form of biogas. Applications include swine and cattle waste management, which is still dominated by aerobic digestion, a less environmental alternative. The low adoption rates of anaerobic digestion is partly caused by the lack of modelling basis for the technology. This is due to the complexity of the process, as it involves dozens of interrelated biochemical reactions driven by hundreds of species of micro-organisms, immersed in a three-phase, non-Newtonian fluid. As a consequence, no practical computer models exist, and therefore, unlike most other engineering fields, the design process for anaerobic digesters still relies heavily on traditional methods such as trial and error. The current state-of-the-art model is Anaerobic Digestion Model No. 1 (ADM1), published by the International Water Association in 2001. ADM1 is a bulk model, therefore it does not account for the effects of concentration gradients, stagnation regions, and particle settling. To address this, this thesis works toward the creation of the first three-dimensional spatially resolved anaerobic digestion model, called Anaerobic Digestion Model with Multi-Dimensional Architecture (ADM-MDA), by developing a framework. The framework, called Coupled Reaction-Advection Flow Transient Solver (CRAFTS), is a general reaction solver for single-phase, incompressible fluid flows. It is a novel partial differential and algebraic equation (PDAE) solver that also employs a novel programmable logic controller (PLC) emulator, allowing users to define their own control logic. All aspects of the framework are verified for proper function, but still need validation against experimental results. The biochemistry from ADM1 is input into CRAFTS, resulting in a manifestation of ADM-MDA; however the numerical stiffness of ADM1 is found to conflict with the second order accuracy of CRAFTS, and the resulting model can only operate under restricted conditions. Preliminary results show spatial effects predicted by the CRAFTS model, and non-observable in the bulk model, impact the digester in a non-trivial manner and lead to measurable differences in their respective outputs. A detailed discussion of suggested work to arrive at a practical spatially resolved anaerobic digestion model is also provided.

Acknowledgments

I would like to acknowledge the generous funding provided by Manitoba Hydro and NSERC. I would like to thank the members of the grad committee, Dr. D. Bagley, Dr. N. Cicek and Dr. H. Soliman for their patience and their guidance. Dr. D. Batstone from the University of Queensland provided me with the code and assistance for ADM1, and our discussions were insightful into the model development. I shared an office and several courses with Amir Birjandi, and he made coming into work fun and enjoyable. Amir and Jonathon Serhal helped me perform my early experimental work with the stirplate, and I am greatly appreciative of those long hours they laboured with me. Dr. S. Ormiston was neither a member of the grad committee, nor an advisor, but his contributions were significant: his technical help in designing the algorithm was invaluable in achieving a working solver, as I implemented everything he suggested; and he even let me deliver a couple of lectures in his Heat Transfer class for my professional development. I would like to thank my advisor, Dr. E. Bibeau, for his trust, patience, and frequent injections of perspective. When he suggested I study manure for my PhD, I thought he was kidding.

I would like to acknowledge the assistance I received from the OpenFOAM community. Whenever an OpenFOAM problem refused to give way, Dr. H. Jasak always took time out of his busy schedule to send me simple three-line emails that lead me straight to the solution. Late in the model development, I encountered a show-stopping bug, but I was not alone in trying to fix it, with thanks to Ivor Clifford for his collaboration.

Finally, I would like to thank my wife, Ashley. Without her, this PhD thesis would never have been written. She worked a job so I could work on the thesis. She supported me in every way imaginable. It is as much her accomplishment as it is mine. Thank-you.

Table of Contents

List of Figures

List of Tables

Abbreviations

ADM1	Anaerobic Digestion Model No. 1
ADM-MDA	Anaerobic Digestion Model with Multi-Dimensional Architecture
ADV	Acoustic Doppler Velocimetry
ASM	Activated Sludge Model
BOD	Biochemical Oxygen Demand
CEL	Common Expression Language
CFD	Computational Fluid Dynamics
COD	Chemical Oxygen Demand
CRAFTS	Coupled Reaction-Advection-Flow Transient Solver
CSTR	Continuous Stirred Tank Reactor
DAE	Differential Algebraic Equation
DE	Differential Equation
HRT	Hydraulic Retention Time
IWA	International Water Association
ODE	Ordinary Differential Equation
OLR	Organic Loading Rate
PDAE	Partial Differential Algebraic Equation
PDE	Partial Differential Equation

PIV	Particle Image Velocimetry
PLC	Programmable Logic Controller
RMSE	Root Mean Square Error
SLOC	Source Lines of Code
STR	Stirred Tank Reactor
SRT	Solids Retention Time
TS	Total Solids
UDF	User-Defined Function
VFA	Volatile Fatty Acids
VOC	Volatile Organic Compounds
VS	Volatile Solids
VSS	Volatile Suspended Solids

Nomenclature

α	thermal diffusivity
β	coefficient of thermal expansion
Γ	the coefficient of diffusion
γ	the ratio of the coarse step and two fine steps performance measures
Γ_{var}	the coefficient of diffusion for species var
Δ	error size
λ	relaxation factor
λ_p	the performance feedback blend factor
μ	dynamic viscosity
ν_{ij}	Petersen matrix coefficient, column i, row j
ρ	density
ρ_j	the reaction rate for reaction j
ρ_j^*	the uninhibited reaction rate for reaction j
ρ_{ref}	reference density
$\rho_{T,a}$	the gas transfer rate for species a
$\boldsymbol{\tau}$	the deviator stress tensor
ϕ_{var}	an arbitrary scalar quantity
a	a discretization coefficient

a_e	the discretization coefficient for the control volume to the east
$a_{e,n}$	the a_e coefficient for the n th control volume
a_i	the discretization coefficient for the neighbouring control volume
$a_i^{y_n}$	the a_i value for the n th variable
a_p	the discretization coefficient for the current control volume
$a_{p,n}$	the a_p coefficient for the n th control volume
$a_{p_{i,j}}$	the a_p coefficient for the i th by j th control volume
$a_p^{y_m y_n}$	the a_p coefficient relating variable y_m and y_n
a_w	the discretization coefficient for the control volume to the west
$a_{w,n}$	the a_w coefficient for the n th control volume
B	a user specified coefficient for calculating the performance feedback blend factor
b_p	the source term coefficient
$b_{p,n}$	the source term coefficient for the n th control volume
$b_p^{y_n}$	the value of b_p for the n th variable
C_a	the carbon content of species a
C_o	the Courant number
$\mathbf{F}$	a set of arbitrary functions
f	an arbitrary function
f_i	the i th function in the set of functions $\mathbf{F}$

$f_{a,b}$	the empirical yield of species a from species b
$\mathbf{G}$	a set of arbitrary functions
$\mathbf{g}$	gravitational vector
$\hat{\mathbf{g}}$	a unit vector pointing in the direction of gravity
g	an arbitrary function
g_i	the i th function in the set of functions $\mathbf{G}$
gpm_a	the grams of COD yielded per mole of species a
H	a logistic function for controlling the gas cut-off
h	numerical step size, or
	height
I	an inhibition factor
i	an indexing variable
I_i	the i th inhibition factor
j	an indexing variable
k	an indexing variable
$K_{a,b}$	the acid-base equilibrium coefficient of species b
$k_{AB,a}$	the base to acid coefficient of species a
k_b	the coefficient of bubble formation
k_i	the i th Runge-Kutta coefficient
K_c	the gas cut-off coefficient

$k_{dec,a}$	the kinetic rate parameter for decay of species a
k_{dis}	the kinetic rate parameter for disintegration
$K_{H,a}$	the gas transfer coefficient (Henry’s coefficient) of species a
$k_{hyd,a}$	the kinetic rate parameter for hydrolysis into species a
$K_{I,h_2,a}$	the parameter for inhibition from hydrogen on species a
$K_{I,IN}$	the parameter for inhibition from inorganic nitrogen
K_{I,nh_3}	the parameter for inhibition from ammonium
$k_{L,a}$	the overall gas-liquid transfer coefficient
$k_{m,a}$	the maximum specific uptake of species a
$K_{S,a}$	the half saturation value of species a
K_w	the acid-base equilibrium coefficient of water
l	characteristic length
M	the maximum number in an indexed set
m	an indexing variable, or
	mass
$\dot{m}_d$	the rate of mass transfer into the gas volume by diffusion
$\dot{m}_b$	the rate of mass transfer into the gas volume by bubbles
m_{var}	the mass of species var
N	the maximum number in an indexed set
N_a	the nitrogen content of species a

n_i	the conversion factor from moles to kilograms for the i th gas specie
$\hat{\mathbf{n}}$	directional vector
n	an indexing variable
O	the error level order of magnitude
p	pressure
p_a	gas partial pressure of species a
p_{atm}	atmospheric pressure
p_c	gas line check valve crack-pressure
p_{hyd}	hydrostatic pressure
p_{tot}	gas total pressure
pH	an acid / base measure of a liquid based on hydrogen ion concentration
$pH_{UL,a}$	the upper limit of pH inhibition for species a
$pH_{LL,a}$	the lower limit of pH inhibition for species a
Q_g	the volume flow rate of the gas
Q_l	the volume flow rate of the liquid
R	universal gas constant
$\mathbf{r}$	a position vector
$\mathbf{r}_{ref}$	a reference position vector
r_{var}	the volumetric generation of species var, i.e. the reaction source term
$\mathbf{S}$	the stress tensor

S	concentration
$\mathbf{s}$	distance vector
$\hat{\mathbf{s}}$	directional vector
S_ϕ	the transport source term for arbitrary species ϕ
$S_{c,i}$	the saturation point of the i th gas specie
$S_{g,a}$	concentration of species a in the gas volume
$\mathbf{S}_M$	the momentum source vector field
S_{var}	the concentration of species var
T	temperature
T_{ref}	reference temperature
t	time
$\mathbf{U}$	the velocity vector field
U_x	the velocity in the x direction
U_y	the velocity in the y direction
U_z	the velocity in the z direction
V_g	the volume in the gas portion of the digester (in the head space)
V_l	the volume in the liquid portion of the digester
X_a	the concentration of particulate matter of species a
$\mathbf{x}$	a set of independent variables
x	an independent variable

x_i	the i th independent variable in the set $\mathbf{x}$
x_n	the value of x at iteration n
$\mathbf{y}$	a set of dependent variables
y	a dependent variable
Y_a	the yield of biomass on species a
y_i	the i th dependent variable in the set $\mathbf{y}$, or
	the value of y in the neighbouring control volumes
y_p	the value of y in the current control volume
y_{n_i}	the value of y_i for the n th variable
y_{n_p}	the value of y_p for the n th variable
$\mathbf{z}$	a set of dependent variables
z_i	the i th dependent variable in the set $\mathbf{z}$
z	a dependent variable

Chapter 1 Introduction

1.1 Purpose and objective

The purpose of this research is to advance the state-of-the-art of computer modelling for anaerobic digesters. Improved models can lead to improved design tools and control systems, which results in better digester designs and reductions in operating costs. The current state-of-the-art digester model is Anaerobic Digestion Model no. 1 (ADM1), which is a bulk model with a strong focus on the biochemical reactions; however it does not include spatial variation, nor does it have a fluid flow model. Therefore, the objective of this research is to develop an anaerobic digestion model that implements the ADM1 biochemistry model, but also includes spatial resolution and fluid flow. This model is Anaerobic Digestion Model with Multi-Dimensional Architecture (ADM-MDA). Many solvers exist for ADM1, but with the added complexity of spatial resolution, none are suitable for ADM-MDA. Therefore, a secondary objective of this research is to develop a solver that can handle ADM-MDA. This solver is the Coupled Reaction-Advection-Flow Transient Solver (CRAFTS). CRAFTS is a general solver and is not limited to ADM-MDA problems alone; however, when it is used for these problems, it is referred to as CRAFTS-implemented ADM-MDA.

To summarize, the key terms in consideration with this research are:

- **ADM1**: the current state-of-the-art bulk biochemistry model;
- **ADM-MDA**: a three dimensional reformulation of the ADM1 equations, but not including the numerical solver that can handle these equations;
- **CRAFTS**: a custom written solver that can solve complex reacting flows, including but not limited to flows with ADM-MDA biochemistry; and
- **CRAFTS-implemented ADM-MDA**: CRAFTS being used with ADM-MDA.

1.2 Overview

Anaerobic digestion, a process that breaks down biomass in a low-oxygen environment, has been used reliably by animals since long before humans walked the Earth. Today, the process is well suited to waste-management and energy production, as it reduces complex biological wastes to simpler constitutive components while simultaneously producing an energy-rich biogas. Sunlight is the main energy source for biogas, therefore it is sustainable. Despite the prevalence of anaerobic digestion in nature, attempting to harness this process industrially has proven to be challenging as it can be unreliable, difficult to control, and relatively costly. The high ownership costs are critical in that they reduce both market adoption rates and the penetration of renewables in industrialized nations.

Unlike other engineering fields where computer models have become principal components of the design process, anaerobic digestion models remain largely unused by industry; rather, anaerobic digester designers rely more heavily on experience-based empirical methods, including trial and error. However, the lack of modeling success is more likely to be caused by the complexity of the digestion process rather than inadequate progress in model development. Anaerobic digestion is a three-phase process involving dozens of interrelated biochemical reactions driven by hundreds of species of microorganisms whose populations are constantly in flux, with their survival dependent on many parameters. Even if these biochemical reactions can be properly represented by a model, numerical difficulties make using the model challenging.

Biochemical computer models for anaerobic digesters have seen some success when validated against lab-scale experimental data; however, validation against data from full-scale digesters has been more difficult to achieve. This shortcoming could partly be attributed to a scaling issue whereby *lab-scale* digesters do not behave the same as *full-scale* digesters. Batstone et al. (2005) validated an anaerobic digester model against two different size reactors and found entirely different physical dynamics occurring in each. Anaerobic digestion models are almost exclusively bulk models, with properties assumed uniform throughout the reactor, irrespective of the presence of gradients. There are few remaining industrial process flow systems where such an approach is confined to zero dimensions. Furthermore, anaerobic digesters are modelled almost exclusively from a biochemical perspective, with the dynamics of fluid flow largely being ignored. These are valid assumptions for a continuously stirred beaker in a lab environment, but not for full-scale thousand to million litre digesters where significant stagnation regions, and various concentration, shear stress and temperature gradients can develop. A conventional anaerobic digestion model cannot capture these phenomena; however, a spatially resolved model with an integrated fluid flow package would. Such a model is presented here: Anaerobic Digestion Model with Multi-Dimensional Architecture (ADM-MDA). ADM-MDA uses ADM1 (Batstone et al., 2002) for the biochemistry and Computational Fluid Dynamics (CFD) for the flow. The purpose of this PhD is to develop such a model and answer:

1. Can ADM-MDA outperform conventional bulk models, and if so, by how much?
2. Can the differences in their predictions justify the additional complexity and computational effort required by ADM-MDA?
3. Can ADM-MDA lead to better design, manufacture and operation of anaerobic digesters?

In developing the model, further questions arise, including:

4. What numerical approach is most suitable to seamlessly combine the two disparate sets of physics: anaerobic digestion and fluid dynamics?
5. What numerical platform is the most suitable for long-term development of the model?
6. What are reliable assumptions for selecting spatial and temporal resolutions for the model?

The timing of this research coincides with the renewed demand for biogas worldwide to address energy stakeholders' sustainability issues.

1.3 Proposed research limitations

Integrating a bulk biochemistry model with a spatially resolved fluid dynamics model introduces many new variable interrelationships that previously could not be expressed, and as such, are not well understood. For example, it is unknown how localized shear stress of the fluid can impact the hydrolysis stage. This aspect is neither a requirement of the proposed research, nor an indicator of its success, but rather, an opportunity for future research endeavours. In the scope of the current research, conventional bulk methods are used where necessary, with some preliminary investigation into the effects of these interrelationships. Additionally, anaerobic digesters are used for a variety of primary purposes, including waste management, energy production, odour management, and pollution mitigation at all scales; therefore, design and operation of the digester depends on the application. The proposed numerical framework will be applicable to most varieties of anaerobic digesters; however, to narrow the scope of the proposed research, and simplify the discussion of the results, a specific application is selected: the use of anaerobic digesters in waste management for the swine and cattle industries, representative of the requirements in Manitoba. Furthermore, attention is required to improve the control of process temperatures and costs in Manitoba as applications are additionally constrained by cold climate and a low energy price structure.

1.4 Livestock industry waste management

In Canada, there are 11.8 million pigs (Statistics Canada, 2002-present – b) and 14.0 million cattle (Statistics Canada, 2002 – present – a). Extrapolating from 2006 manure production levels (Hofmann, 2008) per animal, collectively, these animals produce an estimated 486,000 tonnes of manure every day, or 177.4 gigatonnes per year. Additionally, there are human wastes, agricultural residues, and other manures available in Canada to generate biogas. Furthermore, Canada represents only 2% of the world population. Managing this volume is a significant environmental challenge but could be an important opportunity to assist the integration of intermittent renewables, such as wind and solar, through the production and storage of biogas.

1.4.1 Characterizing manure

Wastes, such as manure, can be harmful to the environment, particularly in large concentrations. Metrics that typically characterize the environmental impact potential of wastes are:

- biochemical oxygen demand (BOD);
- chemical oxygen demand (COD); and
- the percent of volatile solids (VS).

BOD and COD are a measure of the amount of organic components in waste. More specifically, they are the amount of oxygen needed by biological organisms to break down the waste to simpler constituents. COD is similar to BOD, except COD also includes the oxidation of non-biodegradable components. Volatile solids are components that vaporize and ignite at 550°C.

Manure also carries nutrients, including nitrogen, phosphorus and potassium. Although naturally occurring, improper disposal of manure leads to unnaturally high concentrations of these nutrients, which can disrupt an ecosystem. Of particular concern is phosphorus, which can cause algal blooms in lakes and waterways (Schindler, 1977). Furthermore, phosphorus is critically important in Manitoba, given the algal growth in Lake Winnipeg and Lake Manitoba, and the recent legislation regulating its release into the environment (Cicek et al., 2006).

Table 1 shows typical values of waste quality indicators for raw manure (American Society of Agricultural Engineers, 2003).

Table 1: Cattle and swine manure characteristics

	Dairy Cattle	**Beef Cattle**	**Swine**
Total solids [%]	13.95	14.66	13.10
Volatile solids [%]	11.63	12.41	10.12
BOD [g O_2/kg manure]	18.60	27.59	36.90
COD [g O_2/kg manure]	127.9	134.5	100.0
Potassium [g K/kg manure]	3.372	3.621	3.452

Additional waste management problems include pathogen destruction (Wright, 1998) and odour control (Young, 1984). Pathogens include water-borne diseases such as Salmonella or Listeria. Any waste management strategy should be capable of destroying these biological agents. Swine and cattle operations are often close to residential areas and residents are becoming less tolerant of their odours (Ma et al., 2005). Therefore, odour control is an important aspect to waste management.

1.4.2 Waste management strategies

Waste management strategies can be broken down into two categories: solid or semi-solid system, and liquid systems. Liquid systems collect, handle, and store manure with a moisture content of 80% or greater. In 2003, 85.7% of swine farm operations used a liquid storage system (Bourque et al., 2003). On the other hand, 45.7% of dairy cattle operations used a liquid storage system, and less than 6% of beef cattle operations used a liquid storage system. In terms of the objectives of this research, spatial discretization would benefit models for all types of manure systems; however, the incorporation of a flow model would only benefit liquid manure systems. Therefore, the focus will be on liquid manure systems.

1.4.2.1 *Aerobic digestion versus anaerobic digestion*

There are two common liquid waste management strategies. These are:

- aerobic digestion; and
- anaerobic digestion.

Aerobic digestion is a process that breaks down biomass in an oxygenated environment. This method is the conventional strategy, and remains the most common method in use. Typical installations involve a large open lagoon that contains a liquid mixture of manure. After retention of this mixture for several months, it is used as a fertilizer and sprayed onto nearby fields. No methanogens can survive in a well-oxygenated environment; therefore, aerobic digestion cannot produce the methane-rich biogas characteristic of anaerobic digestion. Aerobic digestion produces an order of magnitude more sludge – the solids that remain after digestion, which requires further processing before disposal. The open configuration of the lagoon leads to a risk of environmental contamination in the event of overtopping caused by flooding, resulting in a nutrient-rich runoff, which contributes to algal problems in the water systems. In order to maintain aerobic conditions, the lagoons must be properly aerated, a costly process (Wright et al., 1998). Often neglected, this gives rise to anaerobic conditions, which leads to odour problems, given the open configuration. These odours can be a nuisance for the local population.

Anaerobic digestion is similar to aerobic digestion, except it occurs in a low-oxygen environment. Typical installations include large tanks, or covered lagoons. There are numerous configurations, discussed in Section 1.5.2. The key difference between anaerobic digestion and aerobic digestion is the fact that anaerobic digestion occurs in an enclosed vessel in the absence of oxygen. Since the system is closed, there is an opportunity to dephosphorize the biomass and mitigate algal problems in the water system, a prospect that is actively being investigated (Britton et al., 2005; Mavinic et al., 2007; Türker et al., 2007; and Jordaan et al., 2010). Furthermore, the closed configuration reduces the risk of environmental contamination and odour pollution. Anaerobic digesters are sensitive to temperature; installations in cold climates require additional insulation and possibly heating (Wright et al., 1998). Furthermore, as previously mentioned, anaerobic digesters have reliability problems. Linder (2009) and Mattocks et al. (2005) each make the case that the oil embargo of the 1970's saw a flurry of activity in anaerobic digesters, and that most of these subsequently failed. Linder shows the trend continued well into the 1990's. Lusk's (1998) review of the farm scale anaerobic digester industry in the United States reveals that only 38% of installed farm-digesters were still in operation at the time. The remaining were either not operating (39%) or the farm had closed (23%). Scruton et al. (2004) find that many of those still in operation are performing below their designed outputs, and speculate that anaerobic digester technology is still in the research phase. This has led to a popular belief that anaerobic digesters are not reliable, although more recent successes, such as the USEPA AgStar program, appear to be turning the tide of opinion (Mattocks, 2005).

Despite the benefits of anaerobic digestion over aerobic digestion, their reliability and costs remain significant deterrents to their use. The waste management system is a large capital investment. For example, a 2,200 m^3 anaerobic digester system for a 6,500 head pig operation in Lamar, Colorado, United States, constructed in 1999 cost $368,000 USD (Moser et al., 2002). With this size of investment, the possibility of failure makes reliability a primary consideration.

1.5 Introduction to anaerobic digestion

Anaerobic digestion is a complex multi-stage process involving physical, biochemical and fluid dynamic processes. Before addressing the challenges this poses to computer modelling, an overview of anaerobic digestion is presented.

1.5.1 History of anaerobic digesters

The earliest use of anaerobic digesters may have occurred early in the 10th century BCE in Assyria, where anecdotal evidence suggests biogas was used to heat bath water (Khanal, 2008; He, 2010). The adventurer and trader, Marco Polo (1254-1324), reported seeing covered sewage tanks in China, and Chinese literature provide evidence that these existed 2,000 to 3,000 years ago (He, 2010). This suggests anaerobic digestion was occurring, but whether the practitioners recognized the value of the biogas remains unclear. The first confirmed anaerobic digester, built in 1759, provided power for a leper colony in Bombay, India (Voegeli et al., 2008). Combustible gas was discovered by Henry Cavendish in 1766 (hydrogen), followed ten years later by Alessandro Volta's discovery of methane from anaerobic digestion taking place in swamps. The first full-scale anaerobic digester for wastewater occurred in France in 1881 (Khanal, 2008). The design was adapted and imported to Exeter, England where the biogas fuelled streetlamps in 1895 (Khanal, 2008). Similarly, in Shantou, China, a biogas streetlamp company, founded in 1928, operated digesters with designs based on a prototype household digester (He, 2010). Around 1960, a surge of small-scale anaerobic digester installations for household purposes occurred in China (He, 2010). The oil embargo of the 1970's caused a flurry of activity with anaerobic digesters worldwide, but most of the digesters built at this time failed, leading to a drop in interest and loss of trust in anaerobic digesters (He, 2010; Voegeli et al., 2008; Linder, 2009; Mattocks et al., 2005). More recently, there has been a resurgence of interest in anaerobic digesters (Mattocks et al., 2005; Batstone et al., 2006a).

1.5.2 Types of anaerobic digesters

The term "anaerobic digester" includes engineered digesters such as tanks or covered lagoons, but it also includes natural digesters such as the stomach of an animal, or the bottom of a pond. The research is concerned with engineered digesters, although the natural ones are excellent frames of reference. Engineered digesters can be categorized according to three different characteristics:

- configuration;
- microorganism growth strategy; and
- temperature operating range.

1.5.2.1 Configuration

Figure 1 shows two main configurations of anaerobic digesters for farm-based waste management:

- plug-flow digesters; and
- stirred-tank reactors (STR) digesters.

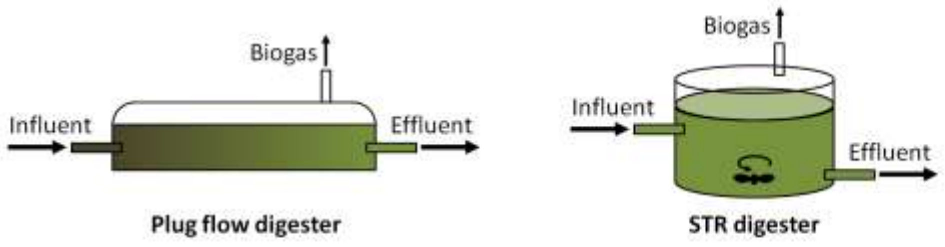


Figure 1: Anaerobic digester configurations

Plug-flow digesters have waste fed from one end, and effluent simultaneously removed from the other end. There is no mixing involved. The position of a particle of biomass relative to the inlet and outlet is an indication of how long it has been in the digester. Therefore, the distance from the inlet correlates with the digestion stage. The biomass fed into a plug-flow digester must have a sufficiently high total solids concentration to ensure minimal mixing occurs. Due to the lack of fluid flow, this type of digester is not considered in this research, although it is expected the resulting model may still be useful for plug-flow digesters.

Stirred-tank reactor (STR) digesters are vessels containing a liquid mixture of water and biomass, characterised by their ability to stir or otherwise agitate the fluid within. STRs that do not stop stirring are known as continuous stirred-tank reactor (CSTR) digesters, and are frequently employed in a laboratory environment. Regular operation of an STR digester involves periodic injection of the liquid mixture at the inlet, with simultaneous removal from the outlet.

Other more complex and specialized configurations exist for anaerobic digesters, such as upflow anaerobic sludge blanket digesters, anaerobic clarigesters, and temperature-phased anaerobic digesters. It is best to use a simpler anaerobic digestion system as a template for the purposes of numerical model design and validation; therefore, these are outside project scope.

1.5.2.2 Microorganism growth strategy

The microorganism growth strategy refers to where the microorganisms grow in the digester. This can be either suspended-growth, or fixed-film. Suspended-growth digesters have microorganism populations embedded within the biomass. Microorganisms naturally contained within this digester take time to grow to the required population levels after adding fluid. Furthermore, drawing fluid from the digester also removes microorganisms. Since the necessary microorganisms are naturally contained within the biomass, this is the simplest growth strategy. In other words, suspended-growth digesters do not make any special accommodations for microorganism growth. The vast majority of anaerobic digesters are of this type. In fixed-film digesters, there are surfaces onto which the microorganisms can attach themselves and grow to form long biofilms. Embedding a porous material into the digester and passing the biomass through it forms the biofilm. The advantage of this strategy is the fact that the microorganism population levels can be maintained at an optimal level, and no time needs to be given for microorganism growth. This can lead to a greater biomass throughput; however, it has been established that hydrolysis is the bottleneck stage of the anaerobic digestion process (Ferrer, 2010), therefore fixed-film digesters are unnecessary if the biomass contains any significant quantities of particulate matter that requires hydrolysis, as is the case with most waste flows.

1.5.2.3 Temperature operating range

Microorganisms that drive anaerobic digestion can only function within a narrow temperature range. Anaerobic digesters operate within the three temperature ranges to which microorganism species have adapted. These are:

- psychrophilic – temperature range from 5°C to 20°C;
- mesophilic – temperature range from 30°C to 35°C; and
- thermophilic – temperature range from 50°C to 60°C.

Psychrophilic is not very commonly used. It requires a specialized set of microorganisms that are not normally found in the waste biomass matrix; therefore, the digester has to be seeded with microorganisms in order to operate. There is also little biochemical activity in a psychrophilic reactor, therefore the hydraulic retention time (HRT) for the biomass is large, typically greater than 100 days (Gervardi, 2003). However, this is attractive in cold climates as the manure does not have to be heated significantly and heat losses are reduced.

Mesophilic is commonly used; most natural anaerobic digesters are mesophilic. One of the advantages of this type of digester is it requires less energy input to maintain its temperature range than a thermophilic digester.

The thermophilic digesters are most commonly used where there is an abundance of waste heat, such as for industrial waste management. Digesters operating in this range have the greatest output rate of biogas. The higher temperature also leads to greater destruction of pathogens.

The main impact temperature range has on modelling of biomass degradation is the temperature inhibition function, and some of the modelling coefficients, such as growth rates, therefore all three temperature classes of digesters can readily be included in the model. Process heating requirements increase with the temperature range, which can be addresses with an energy model.

1.5.3 Inputs and outputs

The influent can be any form of biomass; however, in engineering applications, it is almost exclusively waste products in the form of a liquid water mixture. This mixture is rich in COD and BOD, often including volatile organic compounds (VOCs) and pathogens. Solids, in the form of macro-scale particulate matter, are often embedded within the mixture. The digestion process produces biogas, an energy-rich mixture whose primary components are methane and carbon dioxide, but also includes small amounts of hydrogen sulphide, a corrosive gas that complicates biogas end-use technologies. The effluent leaving the digester is a mixture of wastewater and solids known as digestate. The digestate is composed of dead cells and any organic or inorganic compounds that could not be digested. The exact nature of the digestate depends on the nature of the influent and the digestion process itself, however, it is typically nutrient rich and useful as fertilizer or compost. The digestion process itself is not 100% efficient, as the effluent is never completely digested; it still has measurable COD and BOD, and some particulate matter that has not disintegrated; however, these are significantly reduced. Furthermore, the digestion process removes up to 99% of pathogens.

1.5.4 Biochemistry

The five stages in the anaerobic digestion process are:

1. disintegration;
2. hydrolysis;
3. acidogenesis;

4. acetogenesis; and
5. methanogenesis.

1.5.4.1 Disintegration

Disintegration is the fragmenting of large pieces of particulate matter into smaller solid pieces. This is the anaerobic digestion-specific process that is purely physical. Mixing accelerates this process. Disintegration occurs concurrently with hydrolysis, and together they are the slowest stage of the total anaerobic digestion process (Ferrer, 2010).

1.5.4.2 Hydrolysis

Hydrolysis is the solubilisation of particulate organic matter. This is a biochemical process whereby microorganisms break down the long chain polymers of organic matter into smaller components that other microorganisms can use. For instance, cellulose breaks down into sugars, and proteins break down into amino acids. Numerous species of microorganisms drive hydrolysis, each with their own substrate preference and enzymes. From a modelling perspective, hydrolysis is far too complex to resolve fully. Rather, three first-order empirical relations are normally used to model hydrolysis as a process.

1.5.4.3 Acidogenesis

Acidogenesis is a process in which soluble compounds break down into simpler components known as volatile fatty acids (VFA). Products include propionate, butyrate, and palmitate. This process differs from other similar processes, such as acetogenesis, by the fact that it does not require an additional electron acceptor or donor. This process only occurs after hydrolysis. As with hydrolysis, acidogenesis encompasses dozens of reactions driven by numerous species of microorganisms, depending on the substrate. A few characteristic substrates and their common reactions are selected to model this stage as a whole.

1.5.4.4 Acetogenesis

Acetogenesis is the production of acetate from volatile fatty acids, driven by a group of microorganisms known as acetogens. Unlike the previous steps, acetogenesis encompasses only a handful of reactions and microorganism species. The reactions require an external electron acceptor, such as hydrogen ions, which results in an increased level of free hydrogen. This by-product is inhibitory to acetogenesis, therefore, on its own, acetogenesis cannot be sustained. However, methanogens scavenge free hydrogen during methanogenesis, and free hydrogen concentration is kept low. Acetogens and methanogens have a symbiotic relationship.

1.5.4.5 Methanogenesis

Methanogenesis is the production of methane, and is the final stage of anaerobic digestion. This process encompasses only two main reactions, and a handful of microorganisms, collectively referred to as methanogens. There are two pathways for methanogenesis:

- hydrogenotrophic methanogenesis – the production of methane by consuming hydrogen; and
- aceticlastic methanogenesis – the production of methane by breaking down acetate.

Methanogens prefer the former pathway and therefore keep free hydrogen levels low.

1.6 Literature review

Most studies related to anaerobic digesters are experimental, as opposed to numerical. The complexity of the anaerobic process and the limited success encountered in modelling digesters may explain this.

1.6.1 Experimental

Anaerobic digesters have a plethora of experimental variables available for study, and consequently the experimental studies show a wide diversity. Experimental studies can be grouped into categories according to their focus:

- the overall process;

- the effluent;
- fluid flow;
- design;
- control systems; and
- microbiology.

Given the breadth of material available, only a small cross-section is presented. A more detailed review of the studies mentioned below is provided in Appendix A.

1.6.1.1 Studies focusing on the overall process

Some experiments look at the anaerobic digestion process as a whole, without going into fine detail (e.g. Lusk, 1991; Hansen et al., 1992; Sanchez et al., 1995; Barros et al., 2008; Ojolo et al., 2008; Fantozzi et al., 2009; Comino et al., 2009; Illmer et al., 2009; Alrawi et al., 2010; Cline et al., 2010; and Ferrer et al. 2010). These experiments often measure overall parameters, such as the effluent quality, biogas output, and economics. For example, Lusk (1991) compared the economic performance of two different farm-based anaerobic digesters: an earthen psychrophilic digester, and a mesophilic STR digester.

1.6.1.2 Studies focusing on the effluent

The treatment of wastewater is often the primary purpose of an anaerobic digestion system, in which case, the quality of the effluent is most important. Several studies focus on the effluent, such as quantifying it, or evaluating strategies to improve its quality (e.g. Harikishan et al., 2003; Chen et al., 2006; Vázquez-Padín et al., 2009; and Waeger et al., 2010). Some studies focus specifically on the removal of phosphorus using struvite precipitation (Britton et al., 2005; Mavinic et al., 2007; Türker et al., 2007; and Jordaan et al., 2010), an area of research aimed at reducing algae blooms in local waterways.

1.6.1.3 Studies focusing on fluid flow

Several studies focus on the motion of fluid within the digester (e.g. Comerford et al., 1985; Leighton et al., 1996; and Karim et al., 2004, 2005a, 2005b, 2005c). These experiments include dye tracer studies, particle tracking, and characterization of mixing strategies. For example, Comerford et al. (1985) evaluated the practicality of using a heater to mix the digester through free convection. They found it performed poorly: the area below the heater became a dead space, and solids tended to settle out.

1.6.1.4 Studies focusing on design

Some papers pertain mostly to design, providing advice, given the authors' experiences over the course of anaerobic digester experimentation (e.g. Fischer, 1979; Kalia, 1988; Axaopoulos, 2001; Scruton et al., 2004; and Wu et al., 2009). For example, Fischer (1979) provides guidelines for sizing an internal heat exchanger, while Scruton et al. (2004) recommend against internal heat exchangers, owing to digester plugging and maintenance difficulties.

1.6.1.5 Studies focusing on control systems

Studies related to anaerobic digester control systems occupy a significant area in the literature (e.g. Esteban et al., 1997; Bernard et al., 2001; and Espinosa-Solares et al., 2009). An important aspect of digester control is the ability to detect a stressed or upset digester. Marchiam et al. (1993) showed that an increase in propionic acid compared to acetic acid was a good overload indicator. Propionic acid is an intermediate product of the acidogenesis stage, and is consumed during the acetogenesis stage. A build-up of propionic acid indicates a problem with the later stages of anaerobic digestion. Ferrer et al. (2010) also provide concentration ratios indicative of digester upset.

1.6.1.6 Studies focusing on microbiology

Several studies focus on the microbiology of anaerobic digesters under various conditions (e.g. Hori et al., 2006; Ariesyady et al., 2007; Steinberg et al., 2008; and Ike et al., 2010). For example, Hori et al.

(2006) quantified the microbial populations as they transitioned before, during, and after an induced stressful event.

1.6.2 Numerical

From a numerical modelling perspective, anaerobic digesters have seen successes in two main areas: fluid flow, and biochemistry. There is limited success in combining the two.

1.6.2.1 Modelling fluid flow

Computational fluid dynamics (CFD) is the use of computer models to solve fluid flow problems, and is a well established field. There have been numerous CFD studies into the flow within an anaerobic digester. These studies generally ignore the biochemical reactions, and focus only on resolving the fluid motion. Rudniak et al. (2004) performed a CFD and experimental investigation into a lab-scale impeller-stirred reactor, with a focus on the thermodynamics of exothermal chemical reactions. Vesvikar et al. (2005) studied a gas-mixed mock digester with Newtonian fluids. The authors used both CFD and radioactive particle tracking, and found good agreement between the two, with the exception of velocity data. Wu et al. (2008) performed a CFD study into flow in lab-scale and pilot-scale digesters, focusing on the effect of using a non-Newtonian fluid model. Meroney et al. (2009) studied the effect of draft tube mixing on pilot-scale and full-scale anaerobic digesters using CFD and tracer injection experimental data. Terashima et al. (2009) performed a CFD investigation into mixing in full-scale anaerobic digesters. The authors incorporated rheological properties of the biomass into the simulation using a pseudo-plastic viscosity model. They compared the results against a dye tracer experiment, and found the results to be in good agreement. The authors also created a non-dimensional parameter, "uniformity index" that proved useful in interpreting the data. Yu et al. (2011) performed an experimental and CFD study into the use of a helical impeller for mixing digesters with high percent total solids. The authors address the trade-off between improving mixing and increasing shear rate, which can be detrimental for biological components.

1.6.2.2 Modelling biochemistry

Early biochemistry models for anaerobic digesters were simple and steady-state, most of which applied only to CSTR digesters (Tramšek et al., 2007); however as anaerobic digester technology became more complex, the models also needed to be more complex (Parker, 2005). Up until 2002 anaerobic digester modelling focused on model fundamentals (Batstone et al., 2006a), and the field was populated with a diversity of multispecies models with differing assumptions and configurations (Parker, 2005), leading to confusion as to which models could or could not be compared (Kleerebezem, 2006). In an attempt to create a universal framework for anaerobic digestion models, and building on earlier success with activated sludge models (Parker, 2005), the International Water Association (IWA) published Anaerobic Digestion Model No. 1 (ADM1) (Batstone et al., 2002). Based on the subsequent scarcity of papers featuring non-ADM1 models of anaerobic digesters, some of the original ADM1 authors make the assertion that it has, therefore, been successful (Batstone, et al. 2006a). There have been non-ADM1 models employed since this time (Siegrist et al., 2002; Keshtkar et al., 2003; Paquin et al., 2006; Biswas et al., 2007; Rodríguez et al., 2008; Jones et al., 2008; von der Schulenburg et al., 2009; and Valle-Guadarrama et al. 2011); however, this may be partly attributed to the complexity of ADM1, and the need for a simplified model for specific applications. There have been considerably more ADM1-related publications.

Depending on the specific implementation, ADM1 has approximately one hundred coefficients, each of which affects the model behaviour. The full list is presented in the Appendix, Section D.2. One of the chief criticisms of ADM1 is that some of these coefficients are difficult, or impossible to determine experimentally (Kleerebezem, 2006). A large cross-section of ADM1 studies focuses on determining these coefficients (Jeong et al., 2004; Picioreanu et al., 2005; de Gracia et al., 2006; Batstone et al., 2006b; Tramšek et al., 2007; Page et al., 2008; Mu et al., 2008; Tartakovsky et al., 2008; Lee at al., 2009; and Batstone et al., 2009). Some studies also perform a sensitivity analysis (Jeong et al., 2004; Tartakovsky et al., 2008; and Lee at al., 2009) in which coefficients are individually varied and their relative impact evaluated.

Since its publication, there has been a plethora of studies in which ADM1 modifications are introduced and validated. This suggests that the anaerobic digestion community has accepted ADM1 as the standard model foundation. Presented here are some of these modifications. Blumensaat et al. (2005) modified ADM1 for two-stage anaerobic digestion, and validated it against a pilot-scale reactor. Parker et al. (2006) modified ADM1 to include odorous emissions modelling; however, the parameters for the model had large uncertainty. Rodríguez et al. (2006) modified the carbohydrate fermentation model, but found no significant effect compared to the original model. Huete et al. (2006) created a modified version of ADM1 that characterizes all components in terms of elemental mass fractions, and is therefore mass-conserving. The authors also provided an example of its implementation. de Gracia et al. (2006), de Gracia et al. (2007) and Galí et al. (2009) further extend this model. Kauder et al. (2007) combined ADM1 with the IWA’s Activated Sludge Model (ASM) for a sequencing batch reactor. Lübken et al. (2007) modified the disintegration model and other parameters to better capture inlet conditions, and performed an energy balance of the digestion process. Jeppsson et al. (2007) reports on a larger project undertaken to evaluate control schemes for wastewater treatment plants. The project successfully incorporates ADM1 into a wider simulation system. Brdjanovic et al. (2007) combined ADM1 with ASM to improve process control at a wastewater treatment plant in India. Nielsen et al. (2008) modified the ion model in ADM1 and validated the changes against a series of lab-scale experiments.

Even as recently as 2005, little experience existed in the use of ADM1 (Blumensaat et al., 2005). Since then, several validations have been published for ADM1. These papers compare model results with experimental results. Most of the studies focusing on modifications or parameters also included validations (Blumensaat et al., 2005; Lübken et al., 2007; Page et al., 2008; and Galí et al., 2009). Additionally, Shang et al. (2005) validated ADM1 against two full-scale wastewater treatment plants under steady-state conditions, and found most values within 10%, although there were few data points. Lee et al. (2009) performed a sensitivity analysis, parameter calibration, and validation for a lab-scale temperature-phased anaerobic digester, and found good agreement between model and experiment. Ozkan-Yucel et al. (2010) calibrated and validated ADM1 against a full-scale wastewater treatment plant under dynamic conditions. The model results are in good agreement with the experimental data.

Although these papers report on good validations with ADM1, the objective is to acquire a practical model. ADM1 is not being widely used in the design process, which is testimony to the fact that it is not yet a practical model. These successful validations were achieved by calibrating the model coefficients to the first portion of the experimental data. With more than a hundred coefficients available to modify, it is possible to influence the model behaviour significantly, and obtain a good fit with nearly any dataset, whether it be from a plug-flow digester, or a stirred tank reactor digester. The spatial effects of each specific digester are inadvertently accounted for when these coefficients are calibrated. For instance, a digester with an unusually large stagnation zone will see lower growth rate coefficients and higher decay coefficients. However, should the flow change, and the stagnation zone shrink, the calibrated model will no longer offer accurate predictions. Furthermore, a model that requires calibration, also requires a sample dataset ahead of time, which precludes numerical experimentation. The objective of this research is to develop a model that is largely independent of calibration.

1.6.2.3 Modelling fluid flow and biochemistry together

Fleming (2002) presented an anaerobic digestion model that incorporates both a biochemistry model as well as a CFD flow model for anaerobic digesters; however, no studies have ever been published that use this model. Several factors hinder this model and have prevented its acceptance:

1. It uses a dated, relatively unsophisticated biochemistry model. The biochemistry model is based on Hill's monod reaction model, published in 1983 (Hill 1983a, Hill 1983b). This model has only four reacting species; whereas ADM1 has more than thirty.
2. It is not a general anaerobic digestion model. It is specific only to mesophilic covered lagoon digesters. This type of digester is relatively rare compared to stirred tank reactors and plug flow digesters.
3. It was overshadowed by the publication of ADM1. Fleming published his model in the same year as ADM1. The latter went on to become the industry standard model for anaerobic digesters.

Recall that the purpose of this research is to advance the state-of-the-art of computer modelling for anaerobic digesters. In order for any researcher in the anaerobic digestion community to find this model useful, it must be based on ADM1, and it must be flexible enough to work with a wide variety of anaerobic digester designs. Fleming's model does not meet this criteria, but ADM-MDA does. In other words, ADM-MDA specifically addresses the problems cited above for Fleming's model.

A general anaerobic digester model requires not only the inclusion of CFD, it also requires a spatially resolved ADM1 model. Some studies have accomplished some aspects of this undertaking. Keshtkar et al. (2003) simulated non-ideal mixing conditions in an anaerobic digester by breaking the fluid into two bulk regions: a flow-through region, and a retention region. By varying the interface between the two regions, non-ideal flow conditions were approximated. The authors used a non-ADM1 biochemical model. Batstone et al. (2005) employed two strategies to simulate plug-flow conditions: the first was Aquasim 2.1d, a proprietary implementation of ADM1 for plug-flow reactors that involves one spatial dimension of discretization; the second was to link multiple ADM1 simulations in series to produce an approximation of one-dimensional discretization. The authors compared a lab-scale reactor with a pilot-scale reactor and found that the lab-scale reactor behaved like a plug-flow digester; whereas the pilot-scale reactor behaved like a CSTR digester. Mu et al. (2008) and Tartakovsky et al. (2008) developed a model for anaerobic digesters that has one dimension of spatial resolution; however, the spatial variation is achieved using an empirical relation, not by discretization.

Anaerobic digesters with fixed-film microbial growth, known as a biofilm, have obvious spatial variations, exhibiting strong concentration gradients at the interface between the biofilm and the fluid. Studies have focused on spatial resolution inside these reactors (Picioreanu et al., 2004; Picioreanu et al., 2005; Laspidou et al., 2005; Batstone et al., 2006b; and von der Schulenburg et al., 2009). Most of these focus on the spatial distribution of the biofilm itself, employing ADM1 only at the interface between the biofilm and the fluid, and applying a bulk model to the fluid region (Picioreanu et al., 2004; and Picioreanu et al., 2005), or ignoring the bulk fluid entirely (Batstone et al., 2006b). von der Schulenburg et al. (2009) resolves fluid flow within porous media, but again, the biochemical model is only applied close to the biomass where advection is assumed to be zero. Laspidou et al. (2005) employs the mathematical concepts of cellular automata to modelling biofilm growth, an entirely different construct than the other models. Cellular automata are strictly Cartesian in space, and simulations with these produce grid-related artefacts. The model employs a simple biochemistry model, and does not have a CFD flow model.

Chapter 2 Theory and method

Implementing ADM-MDA requires an understanding of:

- numerical analysis theory;
- ADM1 theory;
- CFD theory; and
- CRAFTS and ADM-MDA theory.

2.1 Numerical analysis theory

ADM-MDA combines methods from a diversity of areas in the field of numerical analysis, including:

- root finding;
- ordinary differential equations;
- adaptive timestep control;
- timescales and stiffness;
- differential algebraic equations;
- partial differential equations;
- partial differential algebraic equations; and
- equation coupling.

2.1.1 Root finding

One of the most fundamental tasks in numerical analysis is finding the zeros of a function. ADM1 uses this for its ion model and its implicit solver for dissolved hydrogen gas concentration. The Newton-Raphson method is a commonly-used root finder. Given a function $f(x)$, an approximation for its root is given by iterating:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}, \tag{1}$$

where *f'*(*x*) is its derivative with respect to *x*. Newton-Raphson is powerful owing to its rapid convergence properties, but it also suffers from poor global behaviour: a bad first guess for *x* can send it far from the solution, with little hope of recovery (Press et al., 1992).

2.1.2 Ordinary differential equations

Fundamentally, ADM1 differs from CFD by the nature of its governing equations. ADM1 solves a set of ordinary differential equations (ODEs); whereas CFD solves a set of partial differential equations (PDEs). This is a key difference as each type of equation is handled differently, by convention.

All ODEs can be expressed as a set of first order ODEs, given by:

$$\frac{dy_i(t)}{dt} = f_i(t, y_0, y_1, \ldots, y_N), \quad i = 0 \ldots N, \tag{2}$$

where the functions f_i are known, and *t* is the only independent variable. In the case of ADM1, *t* is time. There are several well-established ODE solver algorithms, such as:

- Euler's method, useful only for instruction;
- the fourth-order Runge-Kutta method, useful for general problems; and
- the semi-implicit Bulirsch-Stoer method, useful for stiff sets of equations.

The theory behind these algorithms is detailed in Appendix B. In practical terms, these algorithms can be treated as a black box, whereby their inner-workings are ignored, and only the inputs and outputs are addressed, as shown in Figure 2. To use these ODE solver algorithms, a numerical solver must supply:

- a *time derivative routine*, in which the ODE set is defined; and
- a *Jacobian routine*, in which the derivative of each ODE with respect to all the dependent variables is defined.

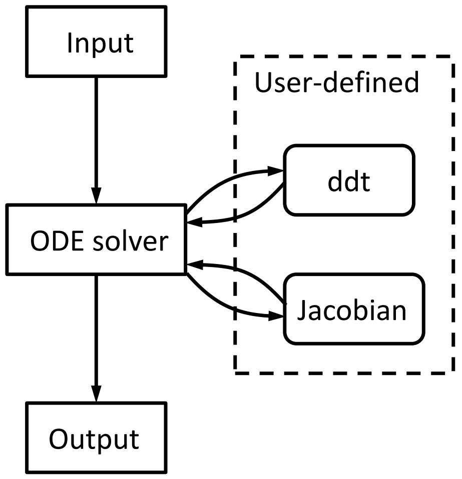


Figure 2: Flowchart of an ODE solver as a "black box"

When the ODE solver algorithm operates, it will call these routines at its discretion and arrive at a solution. However, the performance of the algorithm strongly depends on the *timestep* or stepsize used. Therefore, the ODE solver algorithm also employs *adaptive timestep control*.

2.1.3 Adaptive timestep control

Algorithms that modify their timestep use adaptive timestep control. The simplest form of this is herein called backward-looking timestep control. This uses the results from timestep t_n to determine the new timestep for t_{n+1}. There is no attempt to retry a timestep should it fail to meet the criteria. This is useful for algorithms that cannot easily repeat a timestep.

More sophisticated timestep control applies a measure to the results to determine whether the step was acceptable, and repeats the step should it fail. This behaviour is herein called forward-looking timestep control. Estimated error size is often used as the measure.

In order for a numerical algorithm to have any application, a measure of its error is required. Some algorithms have inherent error predictors that make this easy; however, if one is not available, *timestep-*

doubling can always be employed. This method compares the solution resulting from a single, large timestep against the solution from two half-timesteps, as shown in Figure 3, where:

- point 0 is the initial position;
- point 1 is the result of one large step of size 2*h*;
- point 3 is the result of two smaller steps of size *h*, with the interim value at point 2; and
- Δ is the error estimate.

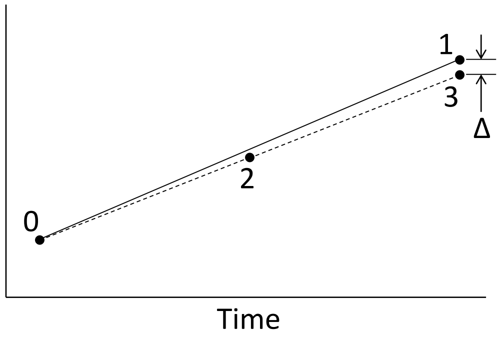


Figure 3: Timestep-doubling error estimation

If the error, Δ, exceeds the specified tolerance, a new timestep can be calculated and the step can be retried. If we assume the solver is at least fourth order accurate, the next best estimate can be approximated as:

$$h_{new} = h_{old} \left| \frac{\Delta_{new}}{\Delta_{old}} \right|^{\frac{1}{5}}, \tag{3}$$

where:

- h_{new} is the new timestep to try;
- h_{old} is the previous timestep;
- Δ_{new} is the desired error size; and
- Δ_{old} is the actual error size encountered previously.

The desired error size does not imply that it is desirable to have an error, but rather this is the maximum acceptable error size, also known as the *convergence criterion*. An error size of zero would be ideal, but this would require a timestep of zero.

Equation (3) is not limited to when the previous timestep exceeded an acceptable error size. It can be used after every timestep to optimize the timestep size. This allows the timestep to grow when the numerical model encounters a relatively stable region. Timestep-doubling can be incorporated into the inner-workings of a solver as shown in Figure 4. The optimum timestep depends not only on the maximum error size, but also on the timescale of the ODE system.

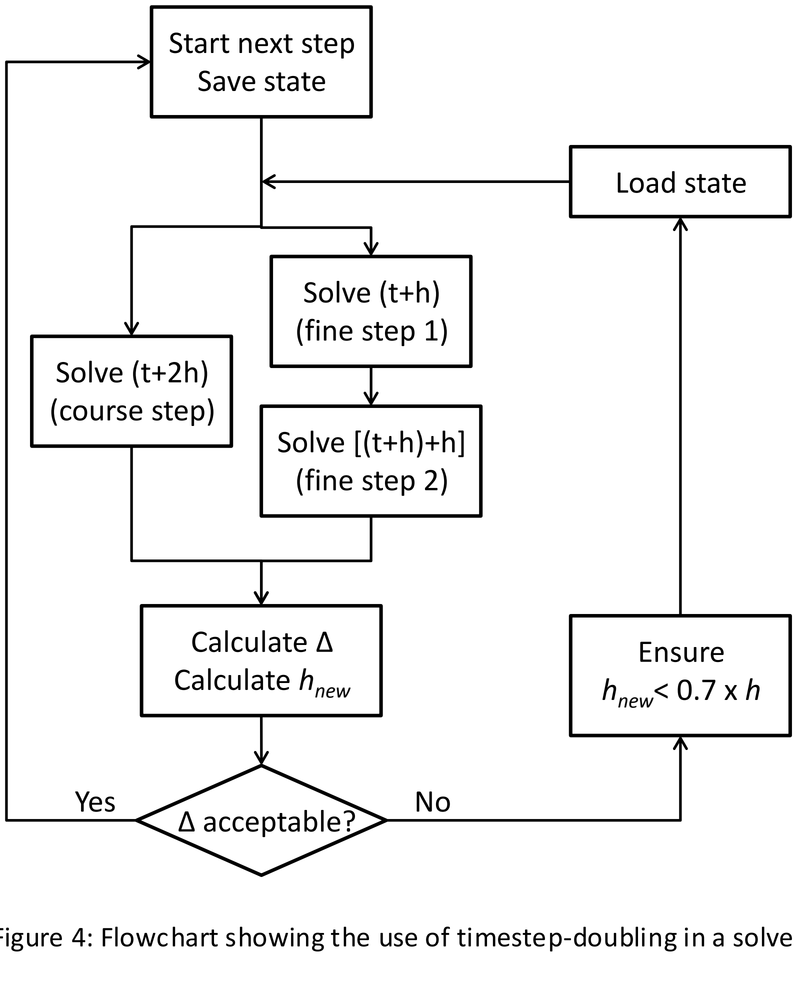


Figure 4: Flowchart showing the use of timestep-doubling in a solver

2.1.4 Timescales and stiffness

The ODE system is characterised by a variety of *timescales*. A timescale is the size of timestep necessary to produce an appreciable change in a model component. For instance, ADM1 models numerous reactions whose rates cover a broad spectrum: acid-base reactions have a short timescale, on the order of fractions of a second; whereas hydrolysis reactions have a long timescale, on the order of hours or days. An ODE system with such a broad range of timescales is said to be *stiff*, and is challenging to solve.

Solving stiff ODE systems requires numerically complex algorithms, such as the semi-implicit Bulirsch-Stoer method. Furthermore, as stiffness increases, the required numerical precision also increases. This becomes a problem when the requirements exceed the architecture of today's computers: double precision, or approximately fifteen significant figures. ADM1 suffers from this problem. Note that the high precision does not imply anything about the validity of the results; rather it is strictly a requirement for numerical stability: when the precision falls below the required threshold, the small timescale phenomena introduce perturbations that upset the solution.

There are two possible solutions to this problem: higher numerical precision may be emulated, a strategy with considerable impact on model performance; or a differential algebraic equation method may be used.

2.1.5 Differential algebraic equations

Stiff ODE systems may be solved by strategically removing short timescale phenomena and resolving their effects in separate algorithms, referred to herein as *algebraic routines*. Conventionally, this is done by assuming these phenomena occur sufficiently quickly that they arrive at equilibrium before their effects are appreciated by the rest of the ODE system. Therefore, rather than model the changes of these short timescale phenomena dynamically with time, a set of equilibrium equations are used. If no equilibrium equation is available, the ODE system itself can serve the purpose—a strategy herein referred to as *static equilibrium*: any one variable can be solved by assuming all other variables are static. In fact, this process can be performed sequentially for any number of variables, but it is only useful for variables that change on a very small timescale relative to the rest; otherwise, important dynamics will be lost in the model. In ADM1, the acid-base reactions and their associated variables are removed from the governing ODE system and solved using acid-base equilibrium. Hydrogen gas concentration can also be removed using static equilibrium to further reduce stiffness.

This strategy simplifies the ODE system, while introducing a set of algebraic equations. Collectively, the simplified ODEs and the algebraic equations constitute a differential algebraic equation (DAE) set. Strictly speaking, the general definition of a DAE set is:

$$0 = \mathbf{F}\left(t, \frac{d\mathbf{y}}{dt}, \mathbf{y}, \mathbf{z}\right), \tag{4}$$

where:

- **F** is a set of arbitrary functions;
- *t* is the independent variable;
- **y** are the dependent variables for which derivatives are defined; and
- **z** are a set of dependent variables for which no derivatives are defined.

In ADM-MDA, **y** and **z** are known as *standard variables* and *implicit variables*, respectively. In ADM1, **y** and **z** are known as *ODE variables* and *algebraic variables*, respectively.

An arbitrary DAE set is difficult to solve; however, in the context of ADM1 and ADM-MDA, all DAEs are have a simpler form, given by:

$$\frac{dy_i(t)}{dt} = f_i(t, y_0 \ldots y_M, z_0 \ldots z_N), \quad i = 0 \ldots M, \tag{5}$$

$$0 = g_j(t, \frac{dy_0}{dt} \dots \frac{dy_M}{dt}, y_0 \dots y_M, z_0 \dots z_N), \quad j = 0 \dots N, \tag{6}$$

where Equations (5) are the differential relations, and Equations (6) are the algebraic relations of the DAE set. This is referred to as *semi-explicit DAEs*. For simplicity, the term DAE will refer to this simpler form. Comparing ODE Equations (2) with DAE Equations (5) and (6), the distinction between these two types is obvious. Whereas the ODE set defines the derivatives of all variables with respect to the independent variable, the DAE set is missing some of these functions; however, in their place, there are additional algebraic relations that can solve for the variables directly.

Unlike general DAE sets, which are difficult to solve, the semi-explicit DAE set can be solved using a sequential solution approach. Here, DAE sets are solved within the framework of a standard ODE solver. The only difference is that whenever the variables are changed, the algebraic relations must be updated. This is achieved by performing the algebraic routines within the time derivative and Jacobian subroutines, as shown in Figure 5.

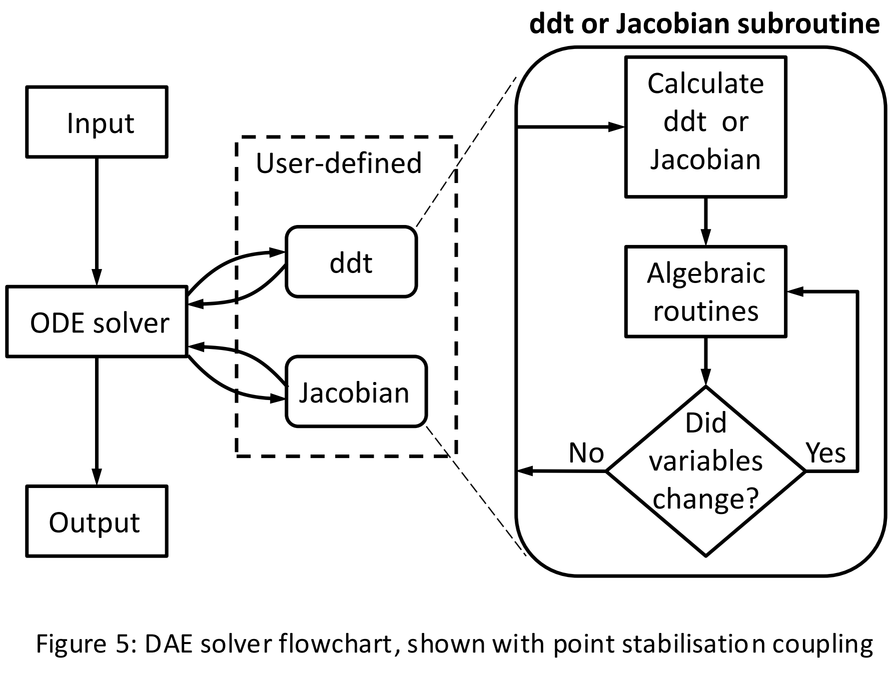


Figure 5: DAE solver flowchart, shown with point stabilisation coupling

This strategy treats algebraic routines independently, which leads to two important consequences. First, the ODE solver algorithm is unable to detect any errors introduced by the algebraic routines. Therefore, it is imperative that the numerical precision of the algebraic routines meets or exceeds that of the ODE solver algorithm, otherwise an error will creep into the solution, which will increase in severity with simulation time. Second, the algebraic routines may affect one another. To address this issue, a coupling strategy is required between the algebraic routines.

While the governing equations for ADM1 are an ODE set, the governing equations for CFD and ADM-MDA are a PDE set.

2.1.6 Partial differential equations

A partial differential equation (PDE) is a more general manifestation of an ODE. Whereas an ODE contains derivatives with respect to only one independent variable, a PDE involves multiple independent variables. A general PDE set may be expressed as:

$$0 = \mathbf{F}(\mathbf{x}, \mathbf{y}, \frac{\partial y_i}{\partial x_j}, \frac{\partial^2 y_i}{\partial x_j \partial x_k}, \ldots), \tag{7}$$

where:

- **F** is a set of arbitrary functions;
- **x** is a set of independent variables;
- **y** is a set of dependent variables; and
- *i*, *j*, and *k* are dummy indices, indicating that the functions, **F**, can include partial derivatives of any **y** with respect to any **x** to an arbitrary order.

Equation (7) is a general PDE set. The governing equations for ADM-MDA are of this type. Numerical solvers do exist for this type of PDE set, but they are based on extended implementations of solvers for a simpler type of PDE: those with only one dependent variable, as given by:

$$0 = f(\mathbf{x}, y, \frac{\partial y}{\partial x_j}, \frac{\partial^2 y}{\partial x_j \partial x_k}, \ldots). \tag{8}$$

These PDEs are often non-linear, and in order to solve them numerically, they must be linearized through a process known as *discretization*. There are several well-established discretization methods suited specifically to PDEs whose independent variables are the dimensions of time and space. These include:

- the finite difference method;
- the finite volume method; and
- the finite element method.

Since the independent variables in the governing equations for ADM-MDA are dimensions of space and time, these methods are applicable. ADM-MDA uses the finite volume method in its implementation. In the finite volume method, the physical geometry is subdivided into numerous contiguous control volumes in a process known as meshing. The method approximates partial derivatives of the PDE between nearby control volumes, and produces one linear equation for every control volume, given by:

$$a_p y_p + \sum_i a_i y_i = b_p, \tag{9}$$

where:

- *a* are the discretization coefficients;
- index *p* indicates the current control volume;
- index *i* are other local control volumes; and
- *b* is the source term coefficient.

These equations can be converted into a matrix form, such as:

$$\begin{bmatrix} a_{p,0} & a_{e,0} & 0 & 0 & 0 \\ a_{w,1} & a_{p,1} & a_{e,1} & 0 & 0 \\ 0 & a_{w,2} & a_{p,2} & a_{e,2} & 0 \\ 0 & 0 & a_{w,3} & a_{p,3} & a_{e,3} \\ 0 & 0 & 0 & a_{w,4} & a_{p,4} \end{bmatrix} \begin{Bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ y_4 \end{Bmatrix} = \begin{bmatrix} b_{p,0} \\ b_{p,1} \\ b_{p,2} \\ b_{p,3} \\ b_{p,4} \end{bmatrix} \tag{10}$$

This is a simple matrix—a one-dimensional problem with five control volumes; the coefficient matrix will typically have *N* x *N* elements, where *N* is the number of control volumes. This matrix relation can be solved using well established matrix solution methods.

Recall this method applies only to PDEs involving one dependent variable, with the form given in Equation (8). The governing equations for ADM-MDA and CFD have multiple dependent variables, and therefore have the form given in Equation (7). To solve these together, an equation coupling strategy is required.

PDE systems can also suffer from the same stiffness problems that can affect ODE systems. One response is to remove some relations from the PDE system and replace them with algebraic relations. The system becomes a partial differential algebraic equation system.

2.1.7 Partial differential algebraic equations

Partial differential algebraic equations (PDAE) are similar to DAEs, except they involve more than one independent variable. The general definition for a PDAE set is:

$$0 = \mathbf{F}(\mathbf{x}, \mathbf{y}, \frac{\partial y_i}{\partial x_j}, \frac{\partial^2 y_i}{\partial x_j \partial x_k}, \ldots, \mathbf{z}), \tag{11}$$

where:

- **F** is a set of arbitrary functions;
- **x** is a set of independent variables;
- **y** is a set of dependent variables for which partial derivatives are defined; and

- **z** is a set of dependent variables for which no partial derivatives are defined.

The PDAEs in ADM-MDA have four independent variables: the dimensions of space and time. A common technique for solving PDAEs with these independent variables is called the *method of lines*, or *semi-discretization*, and has been used routinely for decades (Jacob et al., 1996; Lucht et al., 1998; Simeon, 1998; Simeon, 2000; Wagner, 2000; and de Dieuvleveult, 2009). In this method, the spatial dimensions are discretized first, reducing the system to one independent variable, effectively converting it to a DAE. At that point, any conventional DAE solver can be used. An alternate method involves solving the PDE set using a block coupled solver, and using a segregated coupling strategy for the algebraic relations. All solution methods require an equation coupling strategy.

2.1.8 Equation coupling

All models involving more than one governing equation with shared variables must address the question of equation coupling. It may seem straightforward that achieving a solution is as simple as solving each equation one at a time, but this is insufficient as the equations may describe competing phenomena. Solving one equation may move the variables to one position, and solving the next one may move them back. For instance, in anaerobic digestion, the acetogenesis reaction produces free hydrogen, which are inhibitory and can cause other reactions to slow down; however, the concurrent methanogenesis reaction consumes this hydrogen as it is produced, always keeping the hydrogen levels low. If these reactions are solved one at a time, unrealistically high hydrogen concentrations will be calculated, and inhibition will come into effect when it should not.

Numerical solvers can be categorised according to how they handle this problem:

- coupled solvers; and
- segregated solvers.

2.1.8.1 *Coupled solvers*

It is sometimes possible to combine multiple governing equations into the same numerical framework, allowing them to be solved simultaneously. An algorithm that does this is a *coupled solver*. Coupled solvers account for all relations in a single iteration, and therefore they usually require fewer total iterations than their segregated counterparts. The flowchart of a single timestep for a coupled solver is trivially simple, as shown in Figure 6.

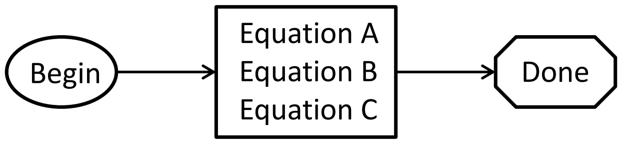


Figure 6: Coupled solver flowchart

The inner workings of a coupled solver, on the other hand, are often more numerically complicated than segregated solvers, and are often more taxing on computational resources. The trade-off is not always clear. The challenge in creating a coupled solver depends on whether the algorithm is *explicit* or *implicit*.

2.1.8.1.1 Coupling explicit algorithms

An explicit algorithm uses values from the previous step to solve the current step. Since the evaluations can be done separately, creating a coupled solver based on an explicit algorithm is trivially simple. The ODE solvers mentioned in Section 2.1.1 are explicit, and therefore can couple multiple equations easily.

2.1.8.1.2 Coupling implicit algorithms

On the other hand, an implicit algorithm uses values from the current step to solve the current step. In other words, all values must be calculated simultaneously. Coupling an implicit algorithm raises the complexity and computational demands of the solver by one level: a linear equation becomes a matrix, a matrix becomes a *block matrix*, and so on. Memory and computational demand also increase.

2.1.8.1.3 Coupled PDE solvers

Recall that solving a single PDE with one dependent variable can be achieved using the finite volume method, which produces a set of linear equations that are solved using matrix methods. When coupling multiple PDEs, the set of linear equations, becomes a set of linear *matrix* equations. That is, an uncoupled PDE solver produces a set of linear equations, repeated here:

$$a_p y_p + \sum_i a_i y_i = b_p, \tag{9}$$

whereas a coupled PDE solver produces a set of linear matrix equations, such as:

$$\begin{bmatrix} a_p^{y_0 y_0} & a_p^{y_0 y_1} & a_p^{y_0 y_2} \\ a_p^{y_1 y_0} & a_p^{y_1 y_1} & a_p^{y_1 y_2} \\ a_p^{y_2 y_0} & a_p^{y_2 y_1} & a_p^{y_2 y_2} \end{bmatrix} \begin{Bmatrix} y_{0_p} \\ y_{1_p} \\ y_{2_p} \end{Bmatrix} + \sum_i \begin{bmatrix} a_i^{y_0} & 0 & 0 \\ 0 & a_i^{y_1} & 0 \\ 0 & 0 & a_i^{y_2} \end{bmatrix} \begin{Bmatrix} y_{0_i} \\ y_{1_i} \\ y_{2_i} \end{Bmatrix} = \begin{bmatrix} b_p^{y_0} \\ b_p^{y_1} \\ b_p^{y_2} \end{bmatrix}. \tag{12}$$

In this example, the a_i coefficient matrix is a diagonal matrix. This signifies that the coupling between PDEs occurs only within a control volume. A system with this form is known as a *point-implicit coupled solver*, and is typical of reacting flows. ADM-MDA has this form, but CFD does not; CFD requires the a_i coefficient matrix to be full, which is known as a *full-coupled solver*.

The uncoupled PDE solver combines its linear equations, Equations (9), into a matrix that is solved using conventional matrix tools. A simple system would produce the matrix shown earlier:

$$\begin{bmatrix} a_{p,0} & a_{e,0} & 0 & 0 & 0 \\ a_{w,1} & a_{p,1} & a_{e,1} & 0 & 0 \\ 0 & a_{w,2} & a_{p,2} & a_{e,2} & 0 \\ 0 & 0 & a_{w,3} & a_{p,3} & a_{e,3} \\ 0 & 0 & 0 & a_{w,4} & a_{p,4} \end{bmatrix} \begin{Bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ y_4 \end{Bmatrix} = \begin{bmatrix} b_{p,0} \\ b_{p,1} \\ b_{p,2} \\ b_{p,3} \\ b_{p,4} \end{bmatrix} \tag{10}$$

The coupled PDE solver combines its linear matrix equations together to yield a block matrix equation, such as:

$$\begin{bmatrix} \begin{bmatrix} a^{y_0y_0}_{p,0} & a^{y_0y_1}_{p,0} & a^{y_0y_2}_{p,0} \\ a^{y_1y_0}_{p,0} & a^{y_1y_1}_{p,0} & a^{y_1y_2}_{p,0} \\ a^{y_2y_0}_{p,0} & a^{y_2y_1}_{p,0} & a^{y_2y_2}_{p,0} \end{bmatrix} & \begin{bmatrix} a^{y_0}_{e,0} & 0 & 0 \\ 0 & a^{y_1}_{e,0} & 0 \\ 0 & 0 & a^{y_2}_{e,0} \end{bmatrix} & 0 & 0 & 0 \\ \begin{bmatrix} a^{y_0}_{w,0} & 0 & 0 \\ 0 & a^{y_1}_{w,0} & 0 \\ 0 & 0 & a^{y_2}_{w,0} \end{bmatrix} & \begin{bmatrix} a^{y_0y_0}_{p,0} & a^{y_0y_1}_{p,0} & a^{y_0y_2}_{p,0} \\ a^{y_1y_0}_{p,0} & a^{y_1y_1}_{p,0} & a^{y_1y_2}_{p,0} \\ a^{y_2y_0}_{p,0} & a^{y_2y_1}_{p,0} & a^{y_2y_2}_{p,0} \end{bmatrix} & \begin{bmatrix} a^{y_0}_{e,0} & 0 & 0 \\ 0 & a^{y_1}_{e,0} & 0 \\ 0 & 0 & a^{y_2}_{e,0} \end{bmatrix} & 0 & 0 \\ 0 & \begin{bmatrix} a^{y_0}_{w,0} & 0 & 0 \\ 0 & a^{y_1}_{w,0} & 0 \\ 0 & 0 & a^{y_2}_{w,0} \end{bmatrix} & \begin{bmatrix} a^{y_0y_0}_{p,0} & a^{y_0y_1}_{p,0} & a^{y_0y_2}_{p,0} \\ a^{y_1y_0}_{p,0} & a^{y_1y_1}_{p,0} & a^{y_1y_2}_{p,0} \\ a^{y_2y_0}_{p,0} & a^{y_2y_1}_{p,0} & a^{y_2y_2}_{p,0} \end{bmatrix} & \begin{bmatrix} a^{y_0}_{e,0} & 0 & 0 \\ 0 & a^{y_1}_{e,0} & 0 \\ 0 & 0 & a^{y_2}_{e,0} \end{bmatrix} & 0 \\ 0 & 0 & \begin{bmatrix} a^{y_0}_{w,0} & 0 & 0 \\ 0 & a^{y_1}_{w,0} & 0 \\ 0 & 0 & a^{y_2}_{w,0} \end{bmatrix} & \begin{bmatrix} a^{y_0y_0}_{p,0} & a^{y_0y_1}_{p,0} & a^{y_0y_2}_{p,0} \\ a^{y_1y_0}_{p,0} & a^{y_1y_1}_{p,0} & a^{y_1y_2}_{p,0} \\ a^{y_2y_0}_{p,0} & a^{y_2y_1}_{p,0} & a^{y_2y_2}_{p,0} \end{bmatrix} & \begin{bmatrix} a^{y_0}_{e,0} & 0 & 0 \\ 0 & a^{y_1}_{e,0} & 0 \\ 0 & 0 & a^{y_2}_{e,0} \end{bmatrix} \\ 0 & 0 & 0 & \begin{bmatrix} a^{y_0}_{w,0} & 0 & 0 \\ 0 & a^{y_1}_{w,0} & 0 \\ 0 & 0 & a^{y_2}_{w,0} \end{bmatrix} & \begin{bmatrix} a^{y_0y_0}_{p,0} & a^{y_0y_1}_{p,0} & a^{y_0y_2}_{p,0} \\ a^{y_1y_0}_{p,0} & a^{y_1y_1}_{p,0} & a^{y_1y_2}_{p,0} \\ a^{y_2y_0}_{p,0} & a^{y_2y_1}_{p,0} & a^{y_2y_2}_{p,0} \end{bmatrix} \end{bmatrix} \begin{Bmatrix} \begin{Bmatrix} {y_0}_{p,0} \\ {y_1}_{p,0} \\ {y_2}_{p,0} \end{Bmatrix} \\ \begin{Bmatrix} {y_0}_{p,1} \\ {y_1}_{p,1} \\ {y_2}_{p,1} \end{Bmatrix} \\ \begin{Bmatrix} {y_0}_{p,2} \\ {y_1}_{p,2} \\ {y_2}_{p,2} \end{Bmatrix} \\ \begin{Bmatrix} {y_0}_{p,3} \\ {y_1}_{p,3} \\ {y_2}_{p,3} \end{Bmatrix} \\ \begin{Bmatrix} {y_0}_{p,4} \\ {y_1}_{p,4} \\ {y_2}_{p,4} \end{Bmatrix} \end{Bmatrix} = \begin{bmatrix} \begin{bmatrix} b^{y_0}_{p,0} \\ b^{y_1}_{p,0} \\ b^{y_2}_{p,0} \end{bmatrix} \\ \begin{bmatrix} b^{y_0}_{p,1} \\ b^{y_1}_{p,1} \\ b^{y_2}_{p,1} \end{bmatrix} \\ \begin{bmatrix} b^{y_0}_{p,2} \\ b^{y_1}_{p,2} \\ b^{y_2}_{p,2} \end{bmatrix} \\ \begin{bmatrix} b^{y_0}_{p,3} \\ b^{y_1}_{p,3} \\ b^{y_2}_{p,3} \end{bmatrix} \\ \begin{bmatrix} b^{y_0}_{p,4} \\ b^{y_1}_{p,4} \\ b^{y_2}_{p,4} \end{bmatrix} \end{bmatrix} \tag{13}$$

This numerical structure can be solved, although it requires specialized solution algorithms. The block matrix shown here is for a point-implicit coupled solver. The point-implicit coupled solver requires less memory and computations than the full-coupled solver, as the off-diagonal matrix elements can be stored and operated on as vectors.

The transport and reaction equations are coupled in ADM-MDA and solved using a block matrix solver. The flow field, however, is solved using a conventional segregated method. In practice, transport reaction equations dominate the simulation time.

2.1.8.2 Segregated solvers

An algorithm that solves equations separately is a *segregated solver*. These algorithms have to accommodate the fact that a single iteration of solving the governing equations will not likely produce a valid solution. Therefore, they must implement a feedback mechanism between the governing equations that allows them to affect one another, and a convergence test to ensure the solution is valid, as shown in Figure 7.

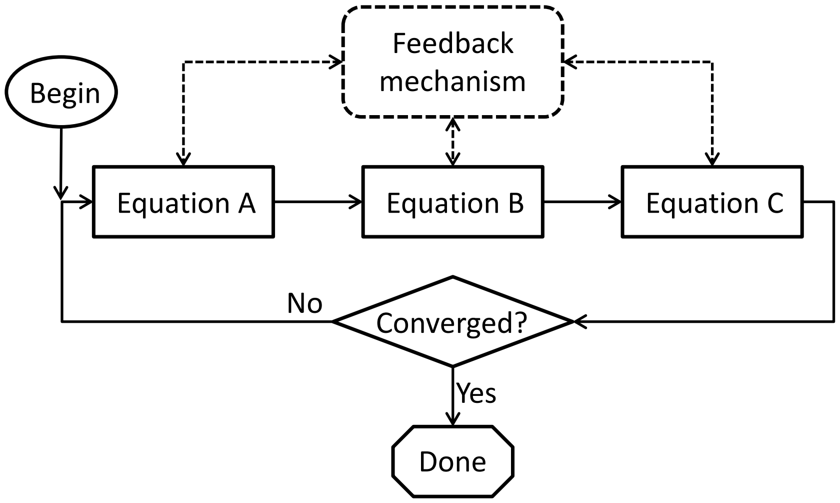


Figure 7: Segregated solver flowchart

Several sophisticated segregated algorithms exist that take advantage of some underlying features of their governing equations. These methods tend to be specific to a set of equations, such as the SIMPLE and PISO algorithms. These solve PDEs, which describe the conservation of mass and momentum in a continuum, to produce velocity and pressure distributions. Discrepancies in velocity between the two governing equations are used to derive changes in pressure, which serves as the feedback mechanism.

Simpler implementations exist, such as *point stabilisation coupling*, which solves the governing equations repeatedly until the solution stabilises, effectively using the system variables themselves as the feedback mechanism.

Unlike in coupled solvers, the feedback mechanism in a segregated solver may be unstable, leading to oscillations in the solution, as shown in Figure 8. To accommodate this, segregated solvers often use *relaxation*, wherein the latest solution is blended with the previous solution at each iteration. The relaxation equation is given by:

$$y_i^{next} = \lambda y_i^{new} + (1 - \lambda)\, y_i^{old}, \tag{14}$$

where:

- y_i^{next} are the values the solver uses for the next timestep or iteration;
- y_i^{new} are the latest values from the current timestep or iteration;
- y_i^{old} are the values from the previous timestep or iteration; and
- λ is the relaxation factor.

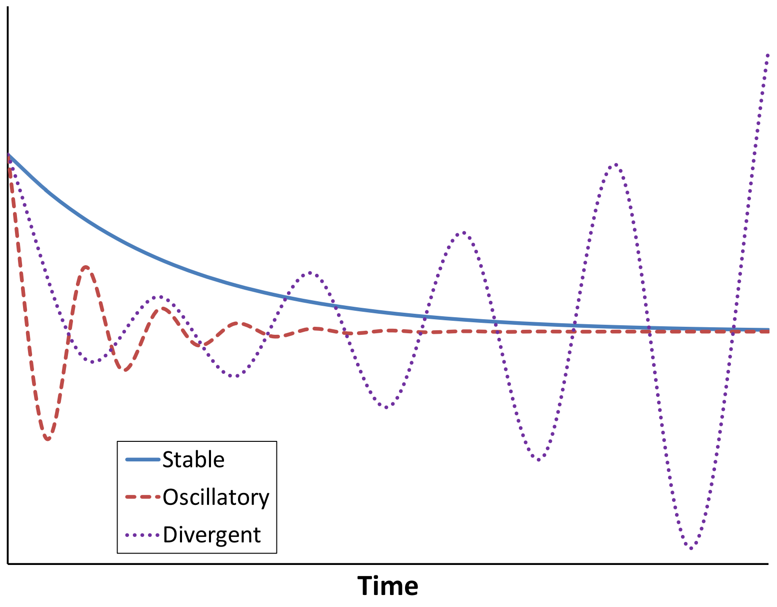


Figure 8: Oscillations characteristic of segregated solvers

Relaxation dampens the oscillations, but it also slows the progression of the solution. Furthermore, relaxation may introduce errors into the solution. Since the relaxed values are calculated without using the governing equations, there is no guarantee these values are consistent with the model. Therefore, it is advisable to test for convergence first, and relax only when convergence fails.

2.1.9 Stabilisation coupling

The term "stabilisation coupling" is assigned here to name a routinely-used numerical convention. In general, the method solves the governing equations repeatedly, while monitoring a predefined measure. When this measure "stabilises" between iterations, the solution is said to be converged. Stabilisation can be detected in various ways, but most commonly, the change in the measure between iterations is calculated. When this change falls below a threshold value, called the convergence criterion, it is said to have stabilised. There are several kinds of stabilisation coupling methods, which vary depending on the nature of the governing equations, the stabilisation measure, and the feedback method. These include:

- point stabilisation coupling;
- curve stabilisation coupling; and
- source term stabilisation coupling.

2.1.9.1 Point stabilisation coupling

Point stabilisation coupling is the most straightforward coupling method. The method solves the governing equations repeatedly, and monitors the solution for stabilisation, as shown in Figure 9. There are no special modifications made to the governing equations, nor to the variables, apart from relaxation. The only feedback mechanism is the effect the changing variables themselves have on the governing equations. Therefore, in this method, the solution itself is the stabilisation measure, and the feedback mechanism.

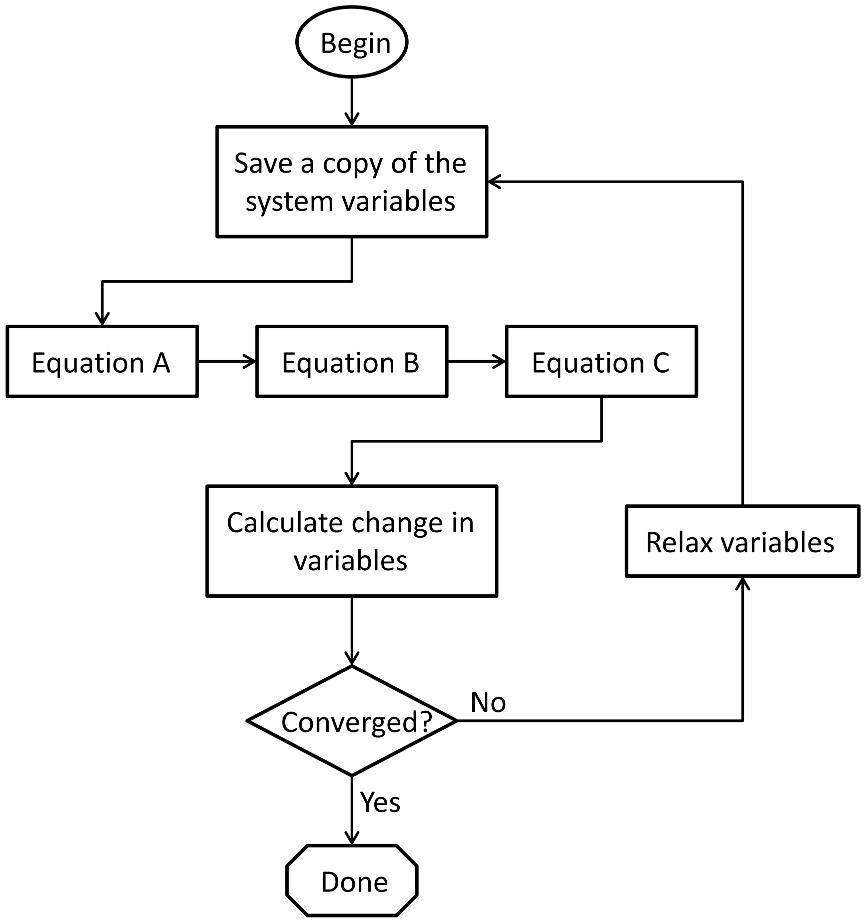


Figure 9: Point stabilisation coupling flowchart

Point stabilisation coupling only applies when the governing equations do not depend on any independent variables and hence have zero degrees of freedom. That is, each governing equation defines a single output value for every set of input values. For this reason, the equilibrium algebraic routines employed by ADM1 are well-suited to this method. ADM-MDA also employs *transient algebraic routines*, ones that depend on time, and these cannot be solved with point stabilisation coupling; *curve stabilisation coupling* must be used.

2.1.9.2 Curve stabilisation coupling

Curve stabilisation coupling is analogous to point stabilisation coupling, except the equation inputs and outputs are curves. This form of coupling works with governing equations that depend on one independent variable and therefore have one degree of freedom, such as a time-dependent process. The principle is the same as point stabilisation coupling, except instead of point values, the method must work with curves. To handle these numerically, the method must break the curves into a set of points, and must employ an interpolation / extrapolation method. Furthermore, the stabilisation measure becomes more complicated, as it has a set of curves as input. This can be calculated using statistical methods across the entire curve, such as root-mean square error (RMSE), or by choosing a characteristic position on the curve and calculating the change between iterations based on that.

Curve stabilisation coupling is not required in ADM1, as all the algebraic routines are based on equilibrium equations. On the other hand, ADM-MDA requires curve stabilisation coupling for its gas model. Only three points are used in ADM-MDA: previous time, current time, and next time, with linear interpolation between these. The stabilisation measure uses only the latest time values, as these are the most important.

One problem that arises with curve stabilisation coupling is that not all numerical frameworks can accommodate a curve as an input. For instance, an ODE solver uses single point values as inputs, called initial conditions, and outputs a set of curves. There is no straightforward way to accommodate a curve as an input. *Source term stabilisation coupling* is required.

2.1.9.3 Source term stabilisation coupling

Source term stabilisation coupling is a novel general numerical method that allows incompatible solver methods to be coupled, even when they share outputs. It is a powerful method that acts as a sort of all-purpose glue, holding complex models together. The method can be used to serve two purposes: first, if the output from one equation is the input for others, source term stabilisation coupling passively converts incompatible outputs into useable inputs; second, if multiple equations have the same output, source term stabilisation coupling acts as a feedback mechanism between equations, causing them to arrive at a universally agreed output. This method is used routinely throughout ADM-MDA.

Source term stabilisation coupling works by inserting controllable source terms into each affected governing equation, and iterating until the terms stabilise. Full implementation details of this method are shown in Appendix C.

2.2 ADM1 theory

As discussed in Section 1.6.2.2, there are numerous implementations to ADM1. The implementation employed here is the original model presented in the IWA's Scientific Technical Report no. 13 (Batstone et al., 2002), with some of the suggestions proposed by Rosen et al. (2006) for numerical stability.

2.2.1 Overview

ADM1 models a digester as two volumes: a liquid volume, representing the bulk fluid, and a gas volume, representing the headspace above the fluid. The gas volume is only used for ADM1's gas model, which it implements as a set of three differential equations, along with several analytical relationships. Most of the model dynamics takes place within the bulk fluid. Each volume contains a diversity of species, the concentrations or pressures of which are the model variables. Table 2 shows the distribution of variables. The full list of variables is included in the Appendix, Section D.1.

Table 2: Variable distribution in ADM1

Variable	Location	Number	Units	Description
S_{var}	Fluid volume	21	kg COD m^{-3}, or mol m^{-3}	Concentration of soluble species "var"
X_{var}	Fluid volume	12	kg COD m^{-3}, or mol m^{-3}	Concentration of particulate species "var"
$S_{g,var}$	Gas volume	3	kg COD m^{-3}, or mol m^{-3}	Concentration of gas "var"

The units used in ADM1 are in metric, but they are in forms deemed most common in industry, and as such, most are not fundamental SI units. Furthermore, some concentrations are expressed in terms of mass COD, whereas others are in terms of moles, depending on what is being measured. ADM-MDA is implemented in software that uses fundamental SI units, therefore ADM1 is herein converted to fundamental SI units for compatibility.

2.2.2 Kinetic model

The bulk fluid is modelled based on mass conservation. For an arbitrary species, *var*, the conservation of mass is:

$$\frac{dm_{var}}{dt} = \dot{m}_{var,in} - \dot{m}_{var,out} + \dot{m}_{var,reac}, \tag{15}$$

where:

- *m* is mass of species *var*;
- $\dot{m}$ is the mass flow rate of the species; and
- the subscript *reac* refers to mass produced in biochemical reactions.

ADM1 primarily uses concentrations. Assuming the fluid is incompressible and its volume does not change, Equation (15) can be rewritten as:

$$V_l \frac{dS_{var}}{dt} = Q_l S_{var,in} - Q_l S_{var,out} + V_l r_{var}, \tag{16}$$

where:

- V_l is the total volume of the liquid;
- S_{var} is the concentration of *var*;
- Q_l is the volumetric flow rate of the liquid; and
- r_{var} is the source term, the volumetric generation of *var* from biochemical processes.

Assuming the fluid is completely mixed, the concentration of the species in the fluid flowing out of the digester is equal to the concentration throughout the volume, and Equation (16) becomes:

$$\frac{dS_{var}}{dt} = \frac{Q_l}{V_l}\left(S_{var,in} - S_{var}\right) + r_{var}. \tag{17}$$

In this equation, $S_{var,in}$ is a boundary condition. The only unknowns are S_{var} and r_{var}, the latter of which is a function of all the variables and is represented by the kinetic model. A simple way to present r_{var} is known as a Petersen matrix, a simplified example of which is shown in Table 3. The full ADM1 Petersen matrix is given in the Appendix, in Section D.4.

Table 3: Petersen matrix for Disintegration and Hydrolysis

	Component →	**i**	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**Rate**
j	**Process ↓**		S_{su}	S_{aa}	S_{fa}	S_I	X_c	X_{ch}	X_{pr}	X_{li}	X_I	ρ_j
0	Disintegration					$f_{sI,xc}$	-1	$f_{ch,xc}$	$f_{pr,xc}$	$f_{li,xc}$	$f_{xI,xc}$	$k_{dis}X_c$
1	Hydrolysis of Carbohydrates		1					-1				$k_{hyd,ch}X_{ch}$
2	Hydrolysis of Proteins			1					-1			$k_{hyd,pr}X_{pr}$
3	Hydrolysis of Lipids		$1-f_{fa,li}$		$f_{fa,li}$					-1		$k_{hyd,li}X_{li}$

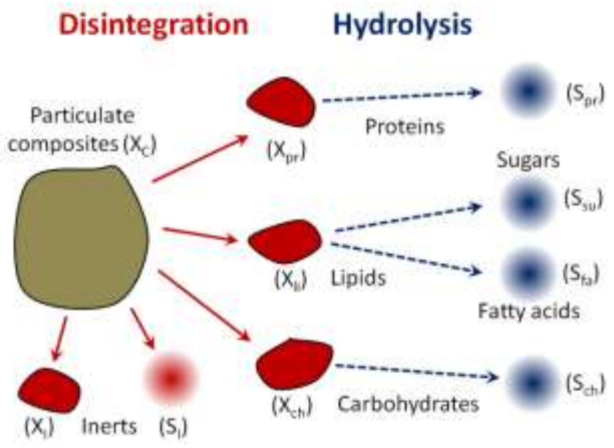


Figure 10: ADM1 disintegration and hydrolysis stages

The Petersen matrix above describes the processes of disintegration and hydrolysis, as pictured in Figure 10. Each row in the matrix represents a single process, showing the relative changes in mass, and the rate at which this reaction takes place. For instance, during the disintegration process, row 0, particulate composites, X_c, break down into particulate carbohydrates, X_{ch}, particulate proteins, X_{pr}, and particulate lipids, X_{li}, as well as soluble and particulate inerts, S_I and X_I. Disintegration is modelled as a first order relation; therefore, the rate is linear with X_c. The -1 under X_c indicates particulate composites are lost during this process, whereas the values in the remaining columns are all positive, indicating a gain. Generally, for mass conservation, the sum across a row must be zero, therefore we have:

$$f_{sI,xc} + f_{ch,xc} + f_{pr,xc} + f_{li,xc} + f_{xI,xc} = 1, \tag{18}$$

where f are empirical yield coefficients, which are determined from experimental results. The calculation of r_{var} is given by summing the mass contributions from all processes. For instance, assuming the Petersen matrix above is the complete system, the r_{su} for soluble sugars, S_{su}, is:

$$r_{S_{su}} = k_{hyd,ch} X_{ch} + (1 - f_{fa,li})\, k_{hyd,li} X_{li}, \tag{19}$$

where k are kinetic rate parameters. More generally, r_{var} is given by:

$$r_{var} = \sum_{j}^{N} \nu_{ij} \rho_j, \tag{20}$$

where:

- N are the total number of processes;
- v_{ij} are the Petersen matrix coefficients; and
- ρ_j are the reaction rates.

2.2.3 Inhibition

Each process has a range of acceptable conditions, such as temperature, or the concentration of a substance. As conditions become adverse, the process slows down, and may stop altogether. This is known as inhibition. There are several kinds of inhibition, such as competitive uptake inhibition where two processes require the same microorganisms. There are a number of suggested modelling strategies for inhibition:

- reduce the process rate by multiplying by a factor between 0 and 1;
- reduce the process rate by modifying its original formula to include built-in inhibition parameters;
- reduce the yields of the process; or
- increase the biological decay rate, k_{dec}.

ADM1 uses the first two methods, as the latter two are not well understood (Batstone et al., 2002). The third option would result in a dynamic Petersen matrix, which would further complicate the numerical system; however, implementations of ADM1 and ADM-MDA both support a dynamic Petersen matrix. Furthermore, ADM-MDA supports all inhibition methods mentioned here. Inhibition is presented in more detail in the Appendix, Section D.3.

2.2.4 Ion model

There are six acid-base pairs in ADM1 that must be solved. These are:

- valerate, S_{va-} / valeric acid, S_{hva};

- butyrate, S_{bu-} / butyric acid, S_{hbu};
- propionate, S_{pro-} / propionic acid, S_{hpro};
- acetate, S_{ac-} / acetic acid, S_{hac};
- bicarbonate, S_{hco3-} / carbon dioxide, S_{co2}; and
- ammonium, S_{nh3} / ammonia, S_{nh4+}.

In addition to all these pairs is the balance of hydrogen ions to liquid water, i.e. the pH of the solution. The hydrogen ion concentration also affects the process rate for these acid-base conversions. To determine the hydrogen ion concentration, ADM1 uses an overall charge balance:

$$S_{cat^+} + S_{NH_4^+} + S_{H^+} - S_{HCO_3^-} - \frac{S_{ac^-}}{0.064} - \frac{S_{pr^-}}{0.112} - \frac{S_{bu^-}}{0.160} - \frac{S_{va^-}}{0.208} - \frac{K_w}{S_{H^+}} - S_{an^-} = 0, \tag{21}$$

where:

- S_{cat+} and S_{an-} are inert cations and anions; and
- the numbers in the denominator are the number of kilograms of COD yielded from a mole of the corresponding species.

Batstone et al. (2002) suggest two ways to implement the ion model. The most straightforward way is to add the acid-base reactions to the Petersen matrix, which implements them as part of the overall ODE set, while using Equation (21) to solve for S_{H+}. Batstone et al. refer to this as the differential equation (DE) implementation. This method leads to a stiff system of equations that require specialized ODE algorithms such as the semi-implicit Bulirsch-Stoer method, and numerical precision that exceeds today's conventional computer architecture, double precision.

The alternate method is to remove the acid-base reactions from the ODE set, and solve them using acid-base equilibrium equations. This effectively assumes that the acid-base reactions occur instantaneously, thereby circumventing the numerical problems. With this method, the system of equations becomes a

DAE set, for which well-established solver algorithms exist. The model will not accurately capture the dynamic transitions of the acid-base system, but since they occur very quickly relative to the other biochemical processes, the deviation is negligible. To achieve this, the acid-base reactions are solved to equilibrium implicitly at every timestep. The equation set is:

$$S_{OH^-} - \frac{K_w}{S_{H^+}} = 0, \tag{22}$$

$$S_{va^-} - \frac{K_{a,va} S_{va}}{K_{a,va} + S_{H^+}} = 0, \tag{23}$$

$$S_{bu^-} - \frac{K_{a,bu} S_{bu}}{K_{a,bu} + S_{H^+}} = 0, \tag{24}$$

$$S_{pro^-} - \frac{K_{a,pro} S_{pro}}{K_{a,pro} + S_{H^+}} = 0, \tag{25}$$

$$S_{ac^-} - \frac{K_{a,ac} S_{ac}}{K_{a,ac} + S_{H^+}} = 0, \tag{26}$$

$$S_{HCO_3^-} - \frac{K_{a,CO_2} S_{IC}}{K_{a,CO_2} + S_{H^+}} = 0, \tag{27}$$

$$S_{NH_4^+} - \frac{S_{IN} S_{H^+}}{K_{a,NH_4} + S_{H^+}} = 0, \tag{28}$$

$$S_{IC} - S_{CO_2} - S_{HCO_3^-} = 0, \text{ and} \tag{29}$$

$$S_{IN} - S_{NH_3} - S_{NH_4^+} = 0, \tag{30}$$

along with Equation (21). These equations describe the full acid-base balance in ADM1; however, there are some redundant variables involved; the system can be reduced to seven independent variables.

2.2.5 Dissolved hydrogen gas concentration

To further improve model performance, the hydrogen gas reactions can be removed from the Petersen matrix and solved implicitly. Combining the implicit solution of S_{H2} with the acid-base equilibria leads to an equation set that can be solved using the framework of standard ODE algorithms such as the fourth-order Runge-Kutta method (Rosen et al., 2006).

2.2.6 Gas model

The gas volume has three types of gases, each with their own associated partial pressure and concentration:

- carbon dioxide (p_{CO2}, $S_{g,CO2}$);
- methane (p_{CH4}, $S_{g,CH4}$); and
- hydrogen (p_{H2}, $S_{g,H2}$).

The model also accounts for water vapour pressure, but this is assumed to be a function of temperature only. ADM1 uses an empirical curve fit, given by:

$$p_{H_2O} = 31.3 \exp\left(5290\left(\frac{1}{298} - \frac{1}{T}\right)\right), \tag{31}$$

where T is temperature. The pressures are related to the concentrations using the ideal gas law:

$$p_{H_2} = S_{g,H_2}\frac{RT}{0.016}, \tag{32}$$

$$p_{CH_4} = S_{g,CH_4}\frac{RT}{0.064}, \text{ and} \tag{33}$$

$$p_{CO_2} = S_{g,CO_2}RT, \tag{34}$$

where the numbers in the denominator are the number of kilograms COD yielded per mole of constituent. These variables are related to variables in the fluid volume through the differential relations:

$$\frac{dS_{g,H_2}}{dt} = -\frac{Q_g S_{g,H_2}}{V_g} + \rho_{T,H_2}\frac{V_l}{V_g}, \tag{35}$$

$$\frac{dS_{g,CH_4}}{dt} = -\frac{Q_g S_{g,CH_4}}{V_g} + \rho_{T,CH_4}\frac{V_l}{V_g}, \text{ and} \tag{36}$$

$$\frac{dS_{g,CO_2}}{dt} = -\frac{Q_g S_{g,CO_2}}{V_g} + \rho_{T,CO_2}\frac{V_l}{V_g}, \tag{37}$$

where:

- Q_g is the volumetric gas flow rate; and

- ρ_T are the gas transfer rates.

The gas transfer rates are given by:

$$\rho_{T,H_2} = k_L a \left(S_{H_2} - 0.016 K_{H,H_2} p_{H_2} \right), \tag{38}$$

$$\rho_{T,CH_4} = k_L a \left(S_{CH_4} - 0.064 K_{H,CH_4} p_{CH_4} \right), \text{ and} \tag{39}$$

$$\rho_{T,IC} = k_L a \left(S_{CO_2} - K_{H,CO_2} p_{H_2} \right), \tag{40}$$

where:

- $k_L a$ is the overall gas-liquid mass transfer coefficient; and
- K_H are Henry coefficients, constants from Henry's Law, which relates small liquid concentrations to vapour pressures.

There are two suggested ways to calculate the gas flow rate. The one used here is:

$$Q_g = K_p \left(p_{tot} - p_{atm} \right) \frac{p_{tot}}{p_{atm}}, \tag{41}$$

where K_p is the gas outlet pipe resistance, and p_{tot} is the sum of the constituent gases. However, in this case, if the pressure difference is negative, the gas would flow backward, and the digester would draw biogas from the atmosphere. To prevent this, ADM1 assumes there is a check valve in the plumbing, and Q_g is adjusted to zero. The total pressure is given by:

$$p_{tot} = p_{CO_2} + p_{CH_4} + p_{H_2} + p_{H_2O}. \tag{42}$$

2.3 CFD theory

Computational fluid dynamics (CFD) is the use of numerical methods to simulate fluid flow. Most CFD is based on the continuum assumption, which gives rise to a set of transport equations.

2.3.1 Transport equations – general form

The transport equations describe the motion of a continuum using conservation laws, including:

- the **conservation of mass**, which results in the **continuity equation**, given by:

$$\frac{\partial \rho}{\partial t} + \boldsymbol{\nabla} \bullet (\rho \mathbf{U}) = 0, \tag{43}$$

where:

- ρ is density, and
- **U** is the velocity field; and

- the **conservation of momentum**, which leads to the **momentum equation** (also known as the Navier-Stokes equation), given by:

$$\frac{D(\rho \mathbf{U})}{Dt} = -\boldsymbol{\nabla} p + \boldsymbol{\nabla} \bullet \boldsymbol{\tau} + \mathbf{S}_M, \tag{44}$$

where:

- *D/Dt* is the substantial derivative operator,
- *p* is the pressure field,
- $\boldsymbol{\tau}$ is the deviator stress tensor, and
- $\mathbf{S}_M$ is the momentum source (if any exists).

These equations are in vector form, and expand into a set of four non-linear PDEs, relating thirteen variables. With more variables than equations, the equation set is said to be open, and cannot be solved.

2.3.2 Viscosity models

Conventionally, the transport equation set is closed by developing a relationship between the deviator stress tensor, $\boldsymbol{\tau}$, and the velocity field, **U**. This relationship, known as the viscosity model, is usually expressed:

$$\boldsymbol{\tau} = f(\mathbf{S}), \tag{45}$$

where **S** is the strain rate tensor, given by:

$$\mathbf{S} = \frac{1}{2}\left[\boldsymbol{\nabla}\mathbf{U} + (\boldsymbol{\nabla}\mathbf{U})^T\right]. \tag{46}$$

The Newtonian viscosity model is the simplest viscosity model, given by:

$$\boldsymbol{\tau} = 2\mu\mathbf{S}, \tag{47}$$

where μ is the dynamic viscosity. Consequently, in a Newtonian fluid, the shear stress of a fluid element is directly related to the velocity gradient, according to:

$$\tau = \mu\left[\hat{\mathbf{n}} \bullet \nabla\left(\mathbf{U} \bullet \hat{\mathbf{s}}\right)\right], \tag{48}$$

where:

- $\hat{\mathbf{n}}$ is the direction perpendicular to the shearing plane; and
- $\hat{\mathbf{s}}$ is the direction parallel to the shearing plane.

In Cartesian coordinates, Equation (48) simplifies to:

$$\tau = \mu\frac{\partial U_x}{\partial y}. \tag{49}$$

Fluids that cannot be modelled adequately with the Newtonian viscosity model are called non-Newtonian fluids. Several alternate viscosity models exist, including:

- the Bird Carreau viscosity model;
- the cross power law viscosity model;
- the Herschel Bulkley viscosity model; and
- the power law viscosity model.

All of these models use the Newtonian formulation, which leads to well-established forms of the transport equations, but modify the dynamic viscosity term. Langner (2009) provides a detailed non-Newtonian model for anaerobic digesters.

2.3.3 Incompressible buoyancy model

Anaerobic digesters are liquid and experience negligible changes in density. Therefore, it is safe to assume the fluid is incompressible. This assumption greatly simplifies the transport equations and leads to faster numerical performance; however, this also eliminates the effects of buoyancy.

Buoyancy occurs when a gravitational body force is added as a momentum source ($\mathbf{S}_M$), and local differences in fluid density exist. Buoyancy gives rise to effects such as convective mixing, which exist in anaerobic digesters. In the case of buoyant liquids, density has negligible effect on the set of transport equations, with the exception of the gravitational force in the momentum source. Rather than discard the incompressible fluid assumption, buoyancy can be incorporated directly into the momentum source through a correlation with temperature, given by:

$$\rho - \rho_{\mathrm{ref}} = -\rho_{\mathrm{ref}}\beta\left(T - T_{\mathrm{ref}}\right), \tag{50}$$

where:

- ρ_{ref} is the density at a reference position;
- T_{ref} is the temperature at a reference position; and
- β is the coefficient of thermal expansion.

This is called Boussinesq buoyancy. Adding buoyancy also gives the fluid a hydrostatic pressure gradient, where the pressure depends on depth. Since the hydrostatic pressure is a predictable quantity, it can be removed from the transport equations and added later to the solution, a strategy that simplifies the momentum source. The exact hydrostatic pressure is given by:

$$p_{\mathrm{hyd}}(\mathbf{r}) = \int_{\mathbf{s}=\mathbf{r}}^{\infty} -\rho(\mathbf{s})\mathbf{g} \bullet d\mathbf{s}, \tag{51}$$

where:

- **r** is the position vector;
- **s** is a line starting at **r** and heading to infinity in the $-\hat{g}$ direction; and
- **g** is the gravitational vector.

This integral form is not practical for numerical purposes. Other approximate forms exist, which can be used with no loss in accuracy, provided they are used consistently before and after the numerical solution. One such approximation is:

$$p_{\mathrm{hyd}} = -\rho_{\mathrm{ref}}\ \mathbf{g} \bullet (\mathbf{r} - \mathbf{r}_{\mathrm{ref}}) \,, \tag{52}$$

where $\mathbf{r}_{\mathbf{ref}}$ is the reference position vector. When this form of hydrostatic pressure is used, the buoyancy momentum source becomes:

$$\mathbf{S}_{\mathbf{M}} = -\rho_{ref}\beta\,(T - T_{ref})\,, \tag{53}$$

where β is the coefficient of thermal expansion.

2.3.4 Transport equations – simplified form

The transport equations are modified with the assumptions detailed above, which are:

- the fluid is assumed to be Newtonian, with a constant viscosity (μ);
- the fluid is assumed to be incompressible, with a constant density (ρ); and
- a Boussinesq buoyancy momentum source is applied.

With these assumptions, the **continuity equation** becomes:

$$\boldsymbol{\nabla} \bullet \mathbf{U} = 0, \tag{54}$$

and the **momentum equation** becomes:

$$\rho\frac{D\mathbf{U}}{Dt} = -\boldsymbol{\nabla} p + \mu\boldsymbol{\nabla}^2\mathbf{U} - \rho_{ref}\beta\,(T - T_{ref})\,. \tag{55}$$

A general transport equation for an arbitrary quantity, ϕ, is given by:

$$\frac{\partial\phi_{var}}{\partial t} + \boldsymbol{\nabla} \bullet (\mathbf{U}\phi_{var}) - \boldsymbol{\nabla} \bullet \Gamma_{var}\boldsymbol{\nabla}\phi_{var} = S_{\phi}, \tag{56}$$

where Γ_{var} is its coefficient of diffusivity, and S_{ϕ} is the source term.

The temperature field can be resolved using a temperature transport equation:

$$\frac{\partial T}{\partial t} + \boldsymbol{\nabla} \bullet (\mathbf{U}T) - \boldsymbol{\nabla} \bullet \alpha\boldsymbol{\nabla} T = 0, \tag{57}$$

where α is thermal diffusivity. These equations are expanded in Cartesian coordinates in the Appendix, Section E.1. Note that there is no reaction source term in Equation (57); this is based on the assumption that energy change from biochemical reactions are negligible in anaerobic digesters. Section 2.4.7 on page 60 discusses this in detail.

2.4 CRAFTS and ADM-MDA theory

ADM-MDA is a multi-dimensional anaerobic digestion model; however, it is a *declarative* model. That is, it only describes the set of equations that need to be solved, but does not describe *how* to solve them. Since ADM-MDA is a PDAE, no existing numerical solvers are suitable. Therefore, this research requires the development of a novel solver: Coupled Reaction-Advection-Flow Transient Solver (CRAFTS). CRAFTS is an *imperative* model, in that it describes *how* to solve the equations. In fact, it is a general PDAE solver and can accept a broad range of equation sets, including ADM-MDA, but also including other reacting flow models such as the Activated Sludge Model (Gujer et al. 1999).

Several components must be considered during the development of CRAFTS and ADM-MDA. These are:

- process control;
- algorithm-switching;
- flow solution;
- coupled reaction model;
- acid-base reactions;
- gas transfer;
- the energy equation; and
- turbulence.

2.4.1 Process control

Anaerobic digesters contain many controlled components whose conditions affect the flow properties within the vessel. For instance, mixers and heaters may be switched on or off, influent may be injected, and gas may be withdrawn. In order to accommodate this, ADM-MDA must include some form of process control, such as a programmable logic controller (PLC) emulator.

2.4.2 Algorithm-switching

Initial bulk model simulations of ADM1 reveals that the biochemistry requires a timestep on the order of hours to days, whereas a flow simulation of the same digester requires a timestep on the order of fractions of a second. This led to the conclusion that solving the biochemistry and flow field together would require an impractically small timestep. Therefore, CRAFTS originally implemented algorithm-switching as a response to this problem. CRAFTS achieved this by creating two algorithms: one that solves the flow field while ignoring the reactions, and the other that solves the reactions, ignoring the flow field. Thus, whenever the conditions changed, such as when a heater is switched on or during a fluid injection event, the flow algorithm runs until the flow field stabilizes, at which point the reaction algorithm resumes. However, in practice, the biochemical reactions in ADM-MDA prove to have a much smaller timescale than ADM1, rendering the algorithm-switching unnecessary. Therefore, CRAFTS was rewritten to run both algorithms sequentially, providing an opportunity to couple the flow solution with the reaction solution. The effort spent developing the algorithm-switching was not lost, as it was used as the framework for the PLC controller emulator. Details of the original algorithm-switching strategy are presented in Appendix F.

2.4.3 Flow solution

CRAFTS must have an integrated flow field solver. The flow solver must:

- use an algorithm suitable to transient flow, such as the PISO algorithm;
- include buoyancy; and
- include turbulence.

2.4.4 Coupled reaction model

ADM1 has a set of 36 independent variables that must be handled in the CFD model. Since ADM1 is not strictly mass-conserving, a mass-fraction strategy will be omitted. There is a mass-conserving ADM1 variation known as the plant-wide model: ADM1-PWM (Huete et al., 2006; de Gracia et al., 2006; de Gracia et al., 2007; and Galí et al., 2009) that would be a good candidate for future studies involving mass fractions; however, at present, the ADM1 variables are treated as scalars. The governing equation for transporting and reacting these scalars is derived from Equation (56), and is given by:

$$\frac{\partial S_{var}}{\partial t} + \mathbf{\nabla} \bullet (\mathbf{U} S_{var}) - \mathbf{\nabla} \bullet \Gamma_{var} \mathbf{\nabla} S_{var} = r_{var}, \tag{58}$$

where r_{var} is the reaction source term. This is the multidimensional analogue of Equation (17):

$$\frac{dS_{var}}{dt} = \frac{V_l}{Q_l}(S_{var,in} - S_{var}) + r_{var}. \tag{17}$$

With 36 manifestations of Equation (58) and those in Section 2.3.4, ADM-MDA is defined by a set of 41 PDEs. The reaction term, r_{var}, in Equation (58) is an arbitrary function that may include any of the other variables. Therefore, the equations are coupled through the reaction term, and cannot be solved without a coupling strategy. This is herein referred to as the *coupled reaction model*. CRAFTS must be capable of solving this set of PDEs.

2.4.5 Algebraic routines

Recall that the implementation of ADM1 is complicated by short timescale phenomena, which renders it numerically impractical to solve. ADM1 resolves this by removing the acid-base reactions and solving these in their own independent implicit routines. There is no reason to expect that moving from an ODE framework to a PDE framework will alleviate these problems with ADM1. On the contrary, adding spatial variation only increases the model complexity. It is expected that ADM-MDA also must define a set of implicit algebraic equations in addition to the PDEs mentioned above, and CRAFTS must accommodate this. That is, ADM-MDA also must include Equations (21)-(30). This changes the nature of ADM-MDA from a set of PDEs to a more complex set of PDAEs.

2.4.6 Gas model

ADM1 has two distinct volumes and an interrelated set of equations shared between them that governs gas transfer. As a bulk model, this works well, as it is understood that gas develops in the liquid volume, and transfers across the interface into the gas volume. Numerically, the variables from each volume can be combined in the ODE set.

Extending this to three dimensions is challenging as CRAFTS is a finite volume solver and therefore it subdivides its liquid volume into an arbitrary number of smaller volumes, many of which will not have a gas-liquid interface. Furthermore, ADM-MDA still defines the gas volume using a bulk model. Coupling variables between these two disparate frameworks is not straightforward. CRAFTS must find a solution to this.

2.4.7 Energy equation

Similar to the continuity and momentum equations, the **energy equation** is based on conservation laws for energy. With reactions and buoyancy involved in the model, use of the energy equation might be expected, but it is omitted. Overall, the biochemical reactions contribute little to the energy balance in an anaerobic digester (Lübken et al., 2007). Furthermore, the total effect of these reactions is exothermic (Lübken et al., 2007; Lindorfer et al., 2006), although the opposite has been asserted, (Inoue et al., 1995), which is a testimony to its small energy contribution. Since the reactions are exothermic, energy use predictions from a model that ignores their contribution will be conservative.

Assuming a zero energy contribution from the reactions is a considerable simplification to the model not only because the energy equation can be omitted, but also the enthalpy calculations can be omitted, including all necessary interpolation to temperature and pH curves for each reaction, which would bring with it a host of experimentally determined model coefficients. The thermodynamic model can be reduced to scalar transport of temperature. This model still allows for heaters, wall heat loss, radiative heat loss, and buoyancy.

2.4.8 Turbulence

The flow in an anaerobic digester is usually laminar. There are some situations where turbulence might develop, such as during fluid injection events, or near mixing impellers. Including turbulence in the model enables a user to model relationships between biochemical reactions and turbulence, something that has not been previously reported. Therefore it is desirable to include turbulence in ADM-MDA. Doing so would not only affect the transient flow mode, but also the scalar transport equations where it acts as a mixing source. Although turbulence is included, detailed discussion on the principles involved are considered outside the scope of this thesis.

Chapter 3 Model development

This chapter provides an overview of the theory, programming environment, code structure, support structures and optimisation of the model.

3.1 Project size

A measure of project size in the software industry is *source lines of code* (SLOC), a simple tallying of the number of lines in the source files of the project. It is a poor measure, as clumsy code tends to be bloated, and elegant code tends to be concise; however it is a common measure, and is useful in terms of orders of magnitude.

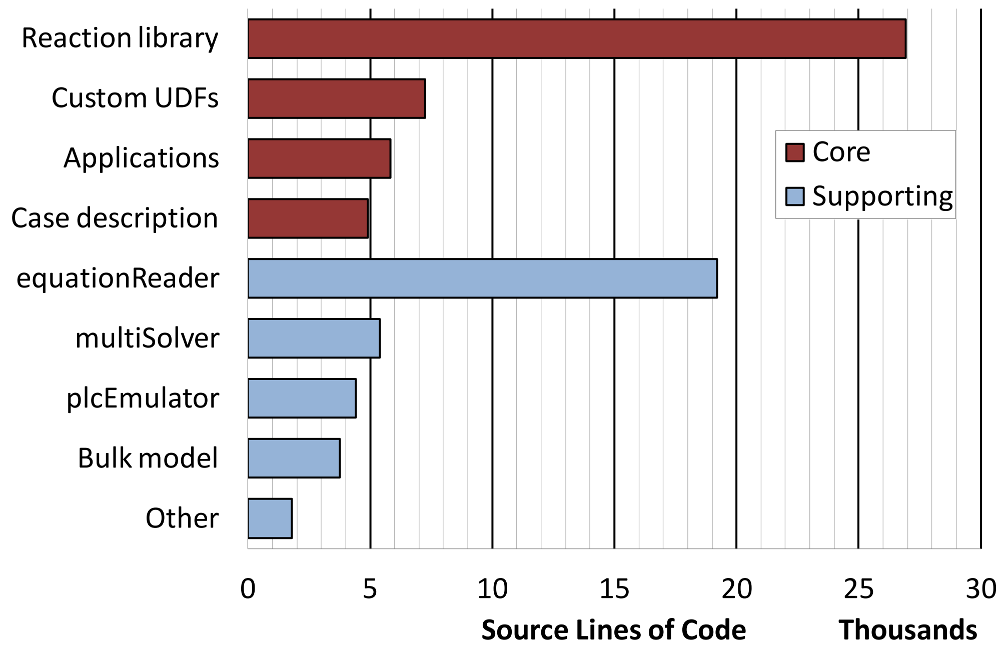


Figure 11: Project size break-down of ADM-MDA

The coding effort behind ADM-MDA took 2.5 years and the result measures approximately 80,000 SLOC, the break-down of which is shown in Figure 11. McConnell (2006) provides time estimation guidelines for the software industry, and suggests that a project measuring 80,000 SLOC would take on the order of

120 staff months. That is, the project would take 10 years for one full-time programmer, or 2.5 years for four full-time programmers. In short, ADM-MDA is a relatively large project.

3.2 Programming environment

The bulk model, a supporting structure used for code verification, is based in a different programming environment than CRAFTS. The bulk model uses Excel and Visual Basic macros; CRAFTS uses a fork of OpenFOAM, based on C++.

3.2.1 Bulk model programming environment

Excel was chosen for several reasons: first, it was incorrectly expected that ADM1 could be rapidly implemented in it; second, since ADM1 is a one dimensional model, the data output is conducive to being displayed in Excel spreadsheets; and third, data transfer between MATLAB and Excel is easy, allowing for rapid verification with the precompiled MATLAB implementation provided by Dr. Batstone. It was decided not to use MATLAB as its ODE solver worked internally, and the exercise of implementing an ODE solver in Excel was expected to be a valuable learning experience.

3.2.2 CRAFTS programming environment

CRAFTS requires an integrated CFD package. The task of building this from the ground up is not practical, nor is it beneficial as countless implementations already exist. Therefore an existing CFD software package was sought. The candidate packages, based on availability, were:

- CFX;
- Fluent; and
- OpenFOAM.

In terms of user customization, CFX has a common expression language (CEL), but this language does not allow user-defined algorithms. Fluent allows for user-defined functions (UDF), but the implementation limits their capability. Furthermore, Fluent does not have a block matrix solver. Ultimately, OpenFOAM was selected due to its flexibility. As the model development progressed, the software requirements of the solver became more specific. CRAFTS requires a block matrix solver, something that is only available in the OpenFOAM-extend fork. Furthermore, CRAFTS requires the more recent flexibility modifications to the VectorN library, a code modification that is currently only available in the git branch `feature/blockCoupledFVM` from the OpenFOAM-extend repository. Finally, incorporating the forward-looking adaptive timestepping with the flow model required a customization of the core libraries of OpenFOAM that is not publicly available. This fork of OpenFOAM will be made available following publication of this work. OpenFOAM is an open source CFD package written in C++. More details regarding OpenFOAM are included in Appendix O.

3.3 Design philosophy

The main goal of the design of CRAFTS is to make it accessible and useful to the ADM1 modelling community. To achieve this, it must be flexible, user-friendly and extensible.

3.3.1 Flexibility

The flexibility requirement is obvious given the diversity of ADM1 modifications that exist (Picioreanu et al., 2005; Blumensaat et al., 2005; Kleerebezem et al., 2006; Parker et al., 2006; Rodríguez et al., 2006; de Gracia et al., 2006; Huete et al., 2006; Batstone et al., 2006a; Batstone et al., 2006b; Brdjanovic et al., 2007; Jeppsson et al., 2007; Kauder et al., 2007; de Gracia et al., 2007; Lübken et al., 2007; Rodríguez et al., 2008; Nielsen et al., 2008; and Galí et al., 2009), with many more expected in the coming years. Therefore, in order to maximize the usefulness of ADM-MDA, the design of CRAFTS must be flexible.

CRAFTS achieves these objectives. For the first time, the user can specify model details without any programming or compiling, including:

- the general system control logic;

- the controllable components within the system, such as heaters and mixers;
- the variables to use in the reaction model;
- the reactions that take place within the model;
- the inhibitions that affect these reactions;
- the equations relating the variables and / or the reactions; and
- the coefficients of these equations.

These are all specified using text files alone.

For users that need to implement their own custom algorithms, CRAFTS is capable of running user-defined functions (UDFs), such as the Newton-Raphson algorithms that remove the ion model from the ODE in ADM1. UDFs require some programming and compiling, but only for a single module, not the entire solver. It is non-trivial to achieve this flexibility with a CFD code, including OpenFOAM.

The flexibility extends into the nature of the numerical methods themselves. Throughout the model development, anytime a setting was encountered, it was handed over to the user as an adjustable parameter, read from an input file.

3.3.2 User-friendliness

OpenFOAM is not a user-friendly program: it does not have a graphical user interface, it does not support Windows operating systems, and it usually requires programming knowledge to use. A considerable effort went into alleviating these disadvantages, however, no graphical user interface was created for CRAFTS: users are required to run their simulation by editing text files and enter console commands. Beyond this, every effort was made to facilitate the usability of CRAFTS.

One primary objective was to make CRAFTS useable without requiring any programming knowledge. This was achieved in the design of CRAFTS: users can specify all the physics of the model through text input files. To implement the design, OpenFOAM needed the capability of reading equations from text files, something it could not do. Therefore a general equation parser extension, **equationReader**, was created for OpenFOAM. Of course, CRAFTS could not avoid the programming requirements of users wishing to implement their own custom algorithms. This is done with UDFs, and several examples are available within the release.

The flexibility requirements lead to verbose input files, which can be intimidating to users. CRAFTS employs several strategies to alleviate this, including:

- thorough documentation of the input files, covered in Appendix K;
- default values for "advanced" settings, allowing simpler input files, and enabling users to familiarise themselves with the basic settings, and explore advanced settings when needed;
- tutorial examples included with the release, giving users a template from which to start;
- self-descriptive keywords for the settings, wherever possible, such as: "maxOuterLoopIterations", or "defaultConvergence"; and
- descriptive error messages and warnings, such as:

```
--> FOAM Warning:
From function admSimpleGas::readDict
in file daeWithGas/admGasModel/admSimpleGas/admSimpleGas.C:455

Coefficient p_gasOverPressure has dimensions [0 0 0 0 0 0 0],
should be [1 -1 -2 0 0 0 0].
```

This warning tells the user that the gas model's over-pressure parameter (see cracking pressure in Section 3.6.9.3 on page 100) is dimensionless when it should have the dimensions of pressure [kg m^{-1} s^{-2}].

3.3.3 Extensibility

Extensibility is the capacity for a software project to adapt to change in the future. CRAFTS is extensible: most of the classes defining the model, such as coefficients, reactions, reaction rates, etc., are built on a hierarchical framework that uses simple base classes as *interfaces*. Derived classes can be loaded dynamically at runtime from custom libraries using a programming idiom called the factory method of initialization. OpenFOAM's implementation of this is known as *runTimeSelection*, and it makes creating new derived classes easy, as these can be added without affecting the rest of the code. For instance, the coefficient class defines five types of coefficients:

- zeros;
- ones;
- constants;
- temperature dependent (based on an exponential relation); and
- custom.

To implement a new kind of coefficient dependent on shear forces, a user needs only to write the shearDependentCoefficient class and use it. None of the other parts of CRAFTS code needs to be modified, even those that work with coefficients. This would be useful to implement a shear-induced decay model. By way of contrast, if the coefficient class were hard-coded, implementing a shear-induced decay model directly into the coefficients would be impractical, as this would affect every part CRAFTS that uses coefficients, and they are used extensively. runTimeSelection is described in detail in the Appendix, Section N.8.

All runTimeSelection based classes also have a custom class implemented. The custom class allows the user to specify the behaviour of the class directly in an input file, something that vastly increases the flexibility of CRAFTS, while requiring no programming.

3.4 Process control and algorithm-switching

As mentioned earlier, anaerobic digesters are controlled devices, involving numerous components whose condition affects the flow properties within. In order to accommodate this, CRAFTS includes a programmable logic controller (PLC) emulator, implemented directly into the CFD software. The user specifies the controllable components and supplies the control logic, and CRAFTS will output a time-resolved simulation of the controlled process. This is achieved with the PLC emulator extension, **plcEmulator**, which is a support structure based on **multiSolver**. It is covered in more detail in Section 3.7.5 on page 108. This is the first implementation of a general control capability in CFD code (Gaden et al., 2012).

To accommodate the need to switch between steady-state and transient state flow modes, the PLC emulator also includes algorithm-switching. The required algorithm itself becomes a controlled output of the PLC emulator.

3.5 Reaction library

The reaction library constitutes the core of CRAFTS. It holds all the information about the model, its variables, reactions, and coefficients. It is an active library, and accommodates many numerical actions, such as:

- evaluating the latest value of each model component, such as reaction rates, inhibition values, and concentrations;
- taking the derivative of model components with respect to any dependent variable; and
- saving the current state of the simulation, should the algorithm require returning to that point.

Applications use these basic operations to implement solver algorithms. The reaction library is generic: it can be used by any application that needs user-defined reactions; CRAFTS is currently the only such application.

3.5.1 Components

The main components of the reaction library are:

- the time class and main database manager, `admTime`;
- the model class, `admModel`;
- its manager classes: `admVariableManager`, `admCoefficientManager`, `admReactionManager`; and
- the components of each of these classes, including:
  - the variables, `admVariable`;
  - the coefficients, `admCoefficient`;
  - the reactions, `admReaction`;
  - the reaction rates, `admReactionRate`; and
  - the reaction rate inhibitions, `admRateInhibition`.

3.5.2 Manager classes and the model class

The reaction library is constituted by a single model class, `admReactionReader`. This class subdivides its information into three parts, each of which is handled by a manager class. These are:

- model coefficients, handled by the **admCoefficientManager**;
- variables, handled by the **admVariableManager**; and
- reactions, handled by the **admReactionManager**.

The main purpose of the model class is to build these manager classes properly and provide access to them. It also stores some characteristics, such as index lists of non-zero yields, and it provides some other functionality, such as convergence test functions. The general structure of the manager classes and their respective data is presented in Figure 12.

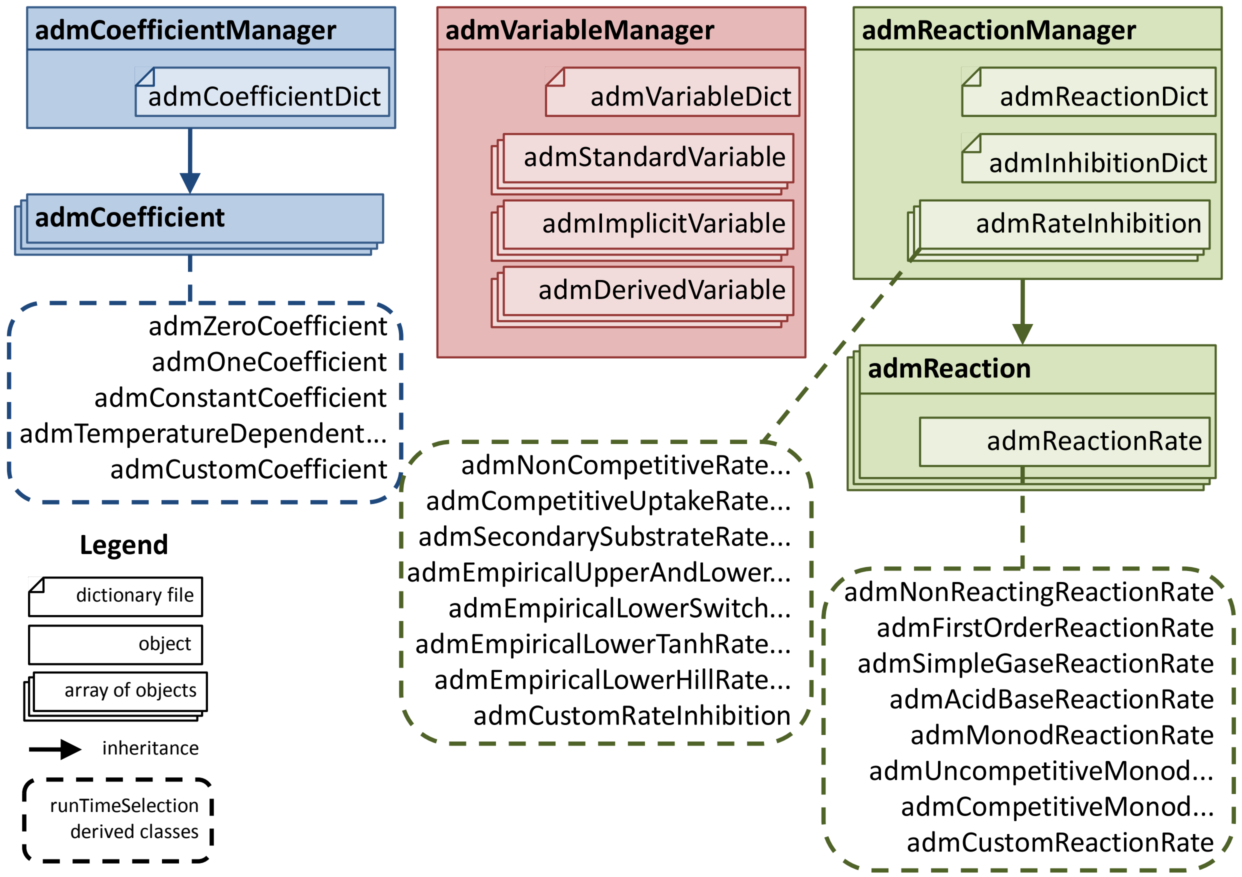


Figure 12: ADM-MDA management class structure

Each manager class reads from one or two user-created input files known as *dictionaries*. All together, these dictionaries define the physics of the reaction model.

3.5.3 Interfaces

The nature of communication between software components is an *interface*. In a large software project, uniformity is crucial. CRAFTS establishes a set of function names, parameter lists, and other conventions, and employs them uniformly. For example, some of the basic interface functions include:

- `evaluate`: return a scalar of the evaluation;
- `ddt`: return the time derivative of the evalution; and

- `ddy`: return the derivative of the evaluation with respect to another variable.

Expanding on these basic functions, suffixes indicate where the evaluation takes place. For instance:

- `evaluateField`: performs `evaluate` on the entire internal field;
- `ddtDims`: returns the dimensions associated with the time derivative; and
- `ddyNonZero`: returns true if the associated function is non-zero.

CRAFTS defines most of these interfaces in a set of interface classes, and others are carried by convention alone. The inheritance diagram for most of the interface classes in CRAFTS is shown in Figure 13.

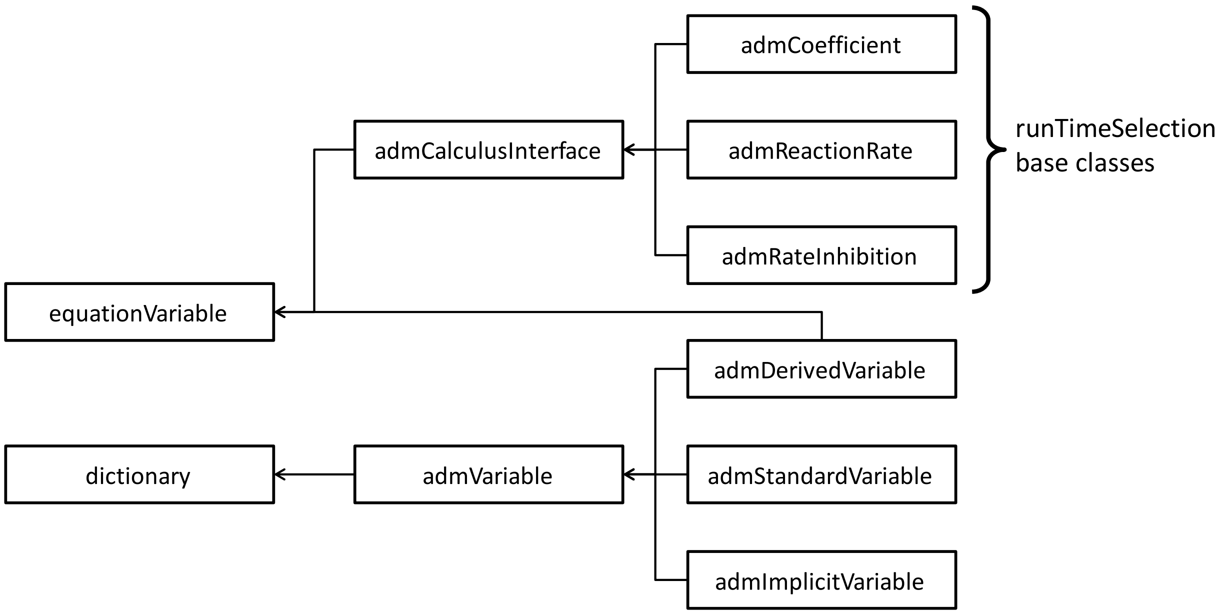


Figure 13: CRAFTS interface classes, inheritance diagram

The majority of the interface definition is in `admCalculusInterface` and `admVariable`. The variable classes store their data differently than the runTimeSelection base classes, which is why they need different inheritance, however, their interfaces are very similar. For efficiency, CRAFTS uses *lazy-evaluation*, a strategy where calculations are only performed when the result is needed. Lazy-evaluation is detailed in Section 3.8.2 on page 115. The `equationVariable` class makes lazy-evaluation possible for the **equationReader**. Standard and implicit variables are always up-to-date, and therefore lazy-evaluation does not apply to them. For this reason, these two classes do not inherit `equationVariable`. All variable classes inherit the `dictionary` class in order to allow users to define their own variable-specific settings in UDFs.

Fundamental objects, such as a coefficient, have simple calculus functions; more complex, derived objects, such as a reaction rate, have more complex calculus functions that call the calculus functions of the simpler objects and apply the chain rule. For instance, the reaction rate is given by:

$$\rho_j = \rho_j^* \prod_k I_k = \rho_j^* I_0 I_1 \ldots I_N, \tag{59}$$

where:

- ρ_j is the inhibited reaction rate;
- ρ_j^* is the uninhibited reaction rate; and
- I_k are all the inhibition factors applied to the reaction rate.

From the chain rule, the derivative of this with respect to an arbitrary variable is:

$$\begin{gathered}\frac{\partial \rho_j}{\partial y} = \left(\frac{\partial \rho_j^*}{\partial y}\right) I_0 I_1 \ldots I_N + \rho_j^* \left(\frac{\partial I_0}{\partial y}\right) I_1 I_2 \ldots I_N + \\ \rho_j^* I_0 \left(\frac{\partial I_1}{\partial y}\right) I_2 I_3 \ldots I_N + \ldots + \rho_j^* I_0 I_1 \ldots I_{N-1} \left(\frac{\partial I_N}{\partial y}\right)\end{gathered}. \tag{60}$$

The rate inhibition object has to call the `ddy` and `evaluate` functions numerous times on several rate inhibition objects. Usually the `ddy` is zero, since most functions have only one or two variables represented. Therefore, to save time calculating, an indexing operation is performed during initialization to record which `ddy`s are worth calling.

3.5.4 Variable classes

There are three classes of variables in CRAFTS:

- standard variables, `admStandardVariable`;
- implicit variables, `admImplicitVariable`; and
- derived variables, `admDerivedVariable`.

Variables are one of the few model components that are not based on runTimeSelection. Therefore the variable types are hard-coded into the model and it would be an involved process to add other variable types. Variables are defined in two places: the `admVariableDict` file contains most information about the variable; and the initial conditions directories contain the initial conditions and dimensions of the variable. The `admVariableDict` file is described in detail in the Appendix, Section K.3.

3.5.4.1 Standard and implicit variables

Standard and implicit variables are the main dependent variables of the system. Originally, the distinction was intended to be clear: the standard variables are solved by the coupled reaction model; the implicit variables are solved by the algebraic routines. However, flexibility requirements blurs the line between these variable types. Source term stabilisation coupling makes it possible to solve both implicit and standard variables in algebraic routines. The differences between these variable types does matter within the context of coupled reaction model: the coupled reaction model solves for the standard variables; whereas the implicit variables act as inputs through source term stabilisation coupling. Algebraic routines must provide a solution for all implicit variables.

3.5.4.2 Derived variables

Many quantities can be calculated whose values have important physical meaning, but are not necessary to achieve solution. For instance, the hydrogen ion concentration, S_{h+}, can also be expressed as pH. The concentration form is more meaningful for transport equations; whereas the pH is more meaningful for inhibition. Both these variables are not required as they are a simple function of one another. Therefore, the derived variables were introduced in order to allow the user to create these additional meaningful variables. These variables can be used to simplify post-processing, and can also be used directly in any of the equations entered in any of the dictionaries. A derived variable must be expressible as a function of the other variables and coefficients. CRAFTS does not store the value of a derived variable, as it is calculated on demand.

In terms of programming effort, the decision to include derived variables was costly. For example, since derived variables can be a function of other variables, calculating derivatives becomes complicated, as it is not immediately obvious if the “with repect to” variable is unrelated, giving zero, or is itself a function of a related variable, giving a value through the chain rule. This necessitates another level of abstraction in these functions. As a result, the model has to create a map of variable relationships, and the function that does this is complex and recursive. However, the implementation is complete, and affords more flexibility to the users. Furthermore, it allows CRAFTS to implement a form of ADM-MDA that more closely matches the implementation of ADM1.

3.5.5 Coefficient classes

Coefficients are numerical values that affect the behaviour of the model. Not only does this include all constants, it also includes some coefficients that vary spatially and temporally. For example, the acid-base equilibrium coefficient for water K_w varies with temperature, which is not spatially uniform.

Coefficients do not include:

- ρ_j - reaction rates;
- I_k - inhibition multipliers that act on reaction rates; or
- v_{ij} - yields in the Petersen matrix.

These are special cases and are handled separately. The five types of coefficients are:

- zero coefficients, which always return zero;
- one coefficients, which always return one;

- constant coefficients;
- temperature dependent coefficients (based on an exponential ratio formula); and
- custom.

The zero and one coefficients are primarily used internally, but they are available to the user.

Coefficients are defined in the `admCoefficientDict` file in the case directory. Coefficients are described in detail in the Appendix, Section K.4.

3.5.6 Reaction class

A single reaction in ADM-MDA represents a row in the Petersen matrix. A reaction consists of:

- a reaction rate, ρ_j;
- a set of yields, v_{ij}; and
- a set of reaction rate inhibitions, I_k.

Reactions are defined in the `admReactionDict` and `admInhibitionDict` in the case directory. These are described in detail in the Appendix, Section K.5.

3.5.6.1 Reaction rates

Reaction rates are the rates at which the reaction occurs. They are the ρ value in the rightmost column of the Petersen matrix. The eight kinds of reaction rates are:

- non-reacting reaction rate;
- first-order reaction rate;
- simple gas reaction rate;
- acid-base reaction rate;
- monod reaction rate;
- competitive monod reaction rate;
- uncompetitive monod reaction rate; and

- custom reaction rate.

Reaction rates are defined in the `admReactionDict` file. A detailed overview of the reaction rates is presented in the Appendix, Section K.6.

3.5.6.2 *Reaction rate inhibitions*

Reaction rate inhibitions are inhibition multipliers that are applied to reaction rates, according to Equation (59), repeated here:

$$\rho_j = \rho_j^* \prod_k I_k = \rho_j^* I_0 I_1 \ldots I_N, \tag{59}$$

where:

- ρ_j is the inhibited reaction rate;
- ρ_j^* is the uninhibited reaction rate; and
- I_k are the reaction rate inhibitions that apply to this reaction rate.

Some forms of inhibition do not conform to Equation (59), rather they modify the reaction rate equation itself. These forms of inhibition can only be modelled by creating a custom reaction rate. For example, the competitive monod reaction rate is a monod reaction rate with competitive inhibition. The eight kinds of reaction rate inhibitions are:

- non-competitive inhibition;
- competitive uptake inhibition;
- secondary substrate inhibition;
- empirical upper and lower inhibition;
- empirical lower (switch function) inhibition;
- empirical lower (tanh function) inhibition;
- empirical lower (Hill function) inhibition; and
- custom inhibition.

Reaction rate inhibitions are defined in the `admInhibitionDict` file. A detailed overview of reaction rate inhibitions is presented in the Appendix, Section K.7.

3.5.7 Build dependencies

The components of the reaction library depend on one another, making it important to build them in a specific order. The reaction library handles this automatically. Their build dependencies are shown in Figure 14. The dotted arrows indicate dependency, and vertical axis is build order; top is earliest. This is handled automatically during object creation.

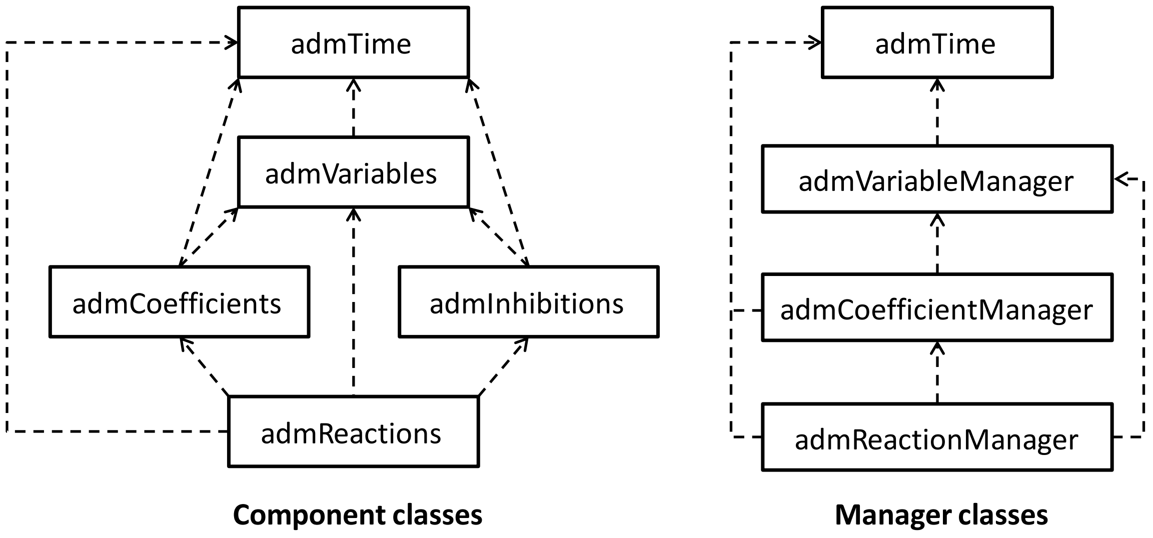


Figure 14: Reaction library build dependencies

3.6 Solver structures

The solver structures are not as generic as the reaction library. These are the code components that use the reaction library in a specific way to solve a PDAE set, such as a spatially resolved implementation of ADM1.

3.6.1 Components

The main components that constitute the solver structures of CRAFTS are:

- the solver applications: `craftsFoam` and `craftsPlcFoam`;

- the coupled reaction model class, `craftsModel`;
- the UDF framework, `craftsUdfs`; and
- the gas model, `craftsGasModel`.

3.6.2 Solver application

The solver applications, `craftsFoam` and `craftsPlcFoam` are the executables that launch CRAFTS. These combine a transient flow solver and PDAE solver together using point stabilisation coupling. `craftsPlcFoam` also incorporates a PLC emulator extension for handling process control.

3.6.3 Flow solver

CRAFTS uses runtime selection for the flow solver. That is, any flow solving algorithm can be implemented for CRAFTS. The three flow solver currently available are:

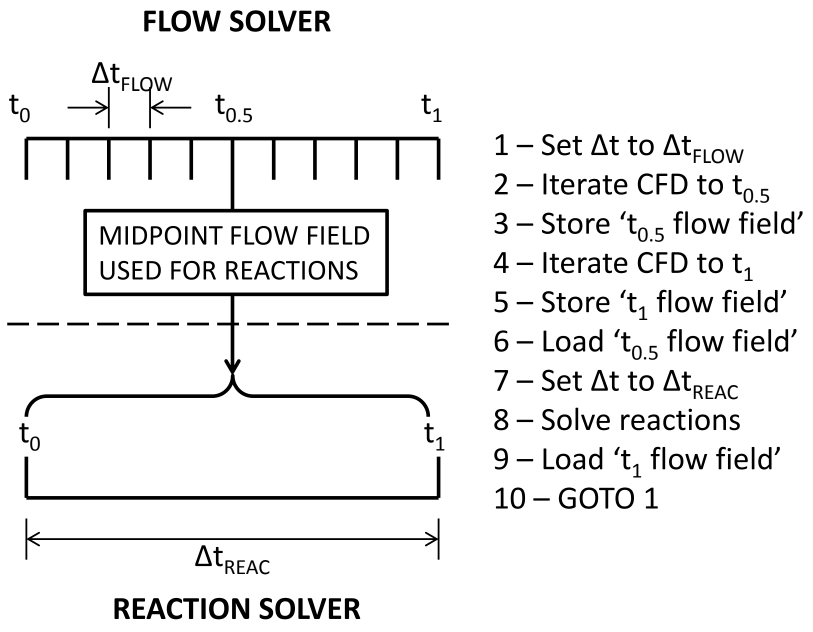
- `craftsSteadyStateFlow`: a model that does nothing, essentially disabling the flow solver;
- `craftsPisoFlow`: an implementation of the PISO algorithm, with a user-selectable source term; and
- `craftsPimpleFlow`: an implementation of the PIMPLE algorithm, with a user-selectable source term.

Optionally, the transient flow solver also incorporates *sub-stepping*, wherein the flow solver advances with a smaller timestep than the reaction solver.

3.6.3.1 Sub-stepping

Sub-stepping allows a smaller timestep to be used for the flow solver than the reaction solver. Not only does this accommodate problems with large differences in timescales, it is also necessary if CRAFTS is set to measure the error in the flow variables for its adaptive timestepping method, discussed later in Section 3.6.5 on page 84. For this, CRAFTS compares the error between one large step and two fine steps. The user can choose to include the flow solution variables in this calculation; however, in practice, CFD flow solvers have large errors in the first few iterations that dissipate quickly as the solution advances. That is, one large step of the PISO algorithm will always be significantly different from two small steps. Therefore sub-stepping was developed.

Sub-stepping chops up the reaction timestep into an even number of steps, and uses the solution at the midpoint as the flow field for the reaction solver, as shown in Figure 15.



FLOW SOLVER
Δt_{FLOW}
t_0
$t_{0.5}$
t_1
MIDPOINT FLOW FIELD USED FOR REACTIONS
t_0
t_1
Δt_{REAC}
REACTION SOLVER
1 – Set Δt to Δt_{FLOW}
2 – Iterate CFD to $t_{0.5}$
3 – Store '$t_{0.5}$ flow field'
4 – Iterate CFD to t_1
5 – Store 't_1 flow field'
6 – Load '$t_{0.5}$ flow field'
7 – Set Δt to Δt_{REAC}
8 – Solve reactions
9 – Load 't_1 flow field'
10 – GOTO 1

Figure 15: Flow model substepping

3.6.4 Coupled reaction model

The coupled reaction model is the solver component of CRAFTS that solves the species reaction and transport, governed by Equations (58), repeated here:

$$\frac{\partial S_{var}}{\partial t} + \nabla \bullet (\mathbf{U} S_{var}) - \nabla \bullet \Gamma_{var} \nabla S_{var} = r_{var}. \tag{58}$$

In the case of ADM-MDA, this is a coupled set of more that forty PDEs.

3.6.4.1 Coupled reaction model solver algorithms

A considerable amount of effort went into developing a suitable algorithm, during which time several coupling strategies were attempted, including:

- a full-field ODE solver;
- a volume-isolation ODE solver;
- a volume-isolation PDE solver; and
- a full-field block coupled PDE solver.

For a detailed review of the first three attempts, refer to Appendix G.

The solver method that succeeds is the full-field block coupled PDE solver. This method constructs and solves the block coupled matrix for the transport and reaction of all species. The solver then runs the algebraic routines, controlled with their own stabilisation coupling, and absorbs the result back into the PDE system using source term stabilisation coupling. The flowchart for this process is shown in Figure 16.

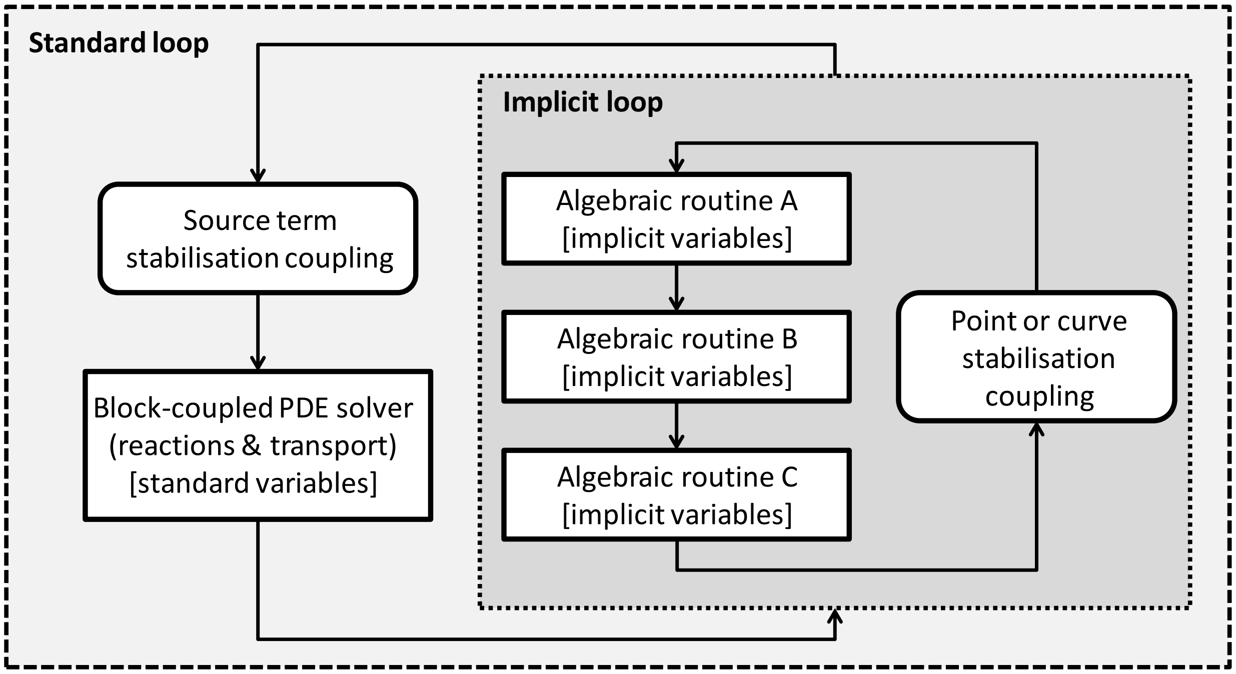

Standard loop
Implicit loop
Source term stabilisation coupling
Block-coupled PDE solver (reactions & transport) [standard variables]
Algebraic routine A [implicit variables]
Algebraic routine B [implicit variables]
Algebraic routine C [implicit variables]
Point or curve stabilisation coupling

Figure 16: Full-field block-coupled PDE solver

This solver requires the use of a general block matrix solver. A block matrix solver was recently released for a fork of OpenFOAM (Clifford et al., 2009; Kissling et al., 2010). CRAFTS uses this block matrix solver for the species transport and reaction kinetics.

3.6.4.2 Coupled reaction model class

The coupled reaction model class, `craftsModel`, is the core of the solver algorithm for the coupled reaction model. It extends the model class, `admModel`, by implementing the functions specific to a block coupled solver strategy, including the `step()` function, which performs one step solution of the species transport, reactions, and UDFs.

3.6.4.3 Block matrix construction

Normal matrix construction is handled automatically in OpenFOAM. Block matrix construction requires more manual manipulation. The linear equations resulting from the discretization process are matrix equations, given earlier in Equation (12) on page 37. There is one of these equations for every control volume in the mesh. The full block matrix concatenates all these equations together to form a very large matrix, such as the one given in Equation (13) on page 38. The process of combining these matrix equations together is handled by the block matrix tools support structure, which is detailed in Section 3.7.7 on page 114. The terms in the matrices consist of transport and reaction components. OpenFOAM has built-in functions that automatically fill discretization terms in standard matrices for transport equations. CRAFTS uses these functions to create a temporary set of standard matrices, and then uses the block matrix tools to absorb these into the block matrix.

CRAFTS must detail the process of filling the reaction terms into the individual linear equations, given in Equation (12). The code for this is contained in the file `craftsModelFillTerms.C`. Since CRAFTS is a point-implicit coupled solver, the reaction terms only contribute to two parts of Equation (12): the centre matrix, a_p, and the source term, b_p. The centre matrix is shown in Figure 17. All the matrices used in CRAFTS are larger than those shown in Equation (12). The matrices are square with side length equal to N + M, where N and M are the number of standard and implicit variables, respectively. CRAFTS indexes the standard variables in the upper left quadrant, and the implicit variables in the lower right quadrant.

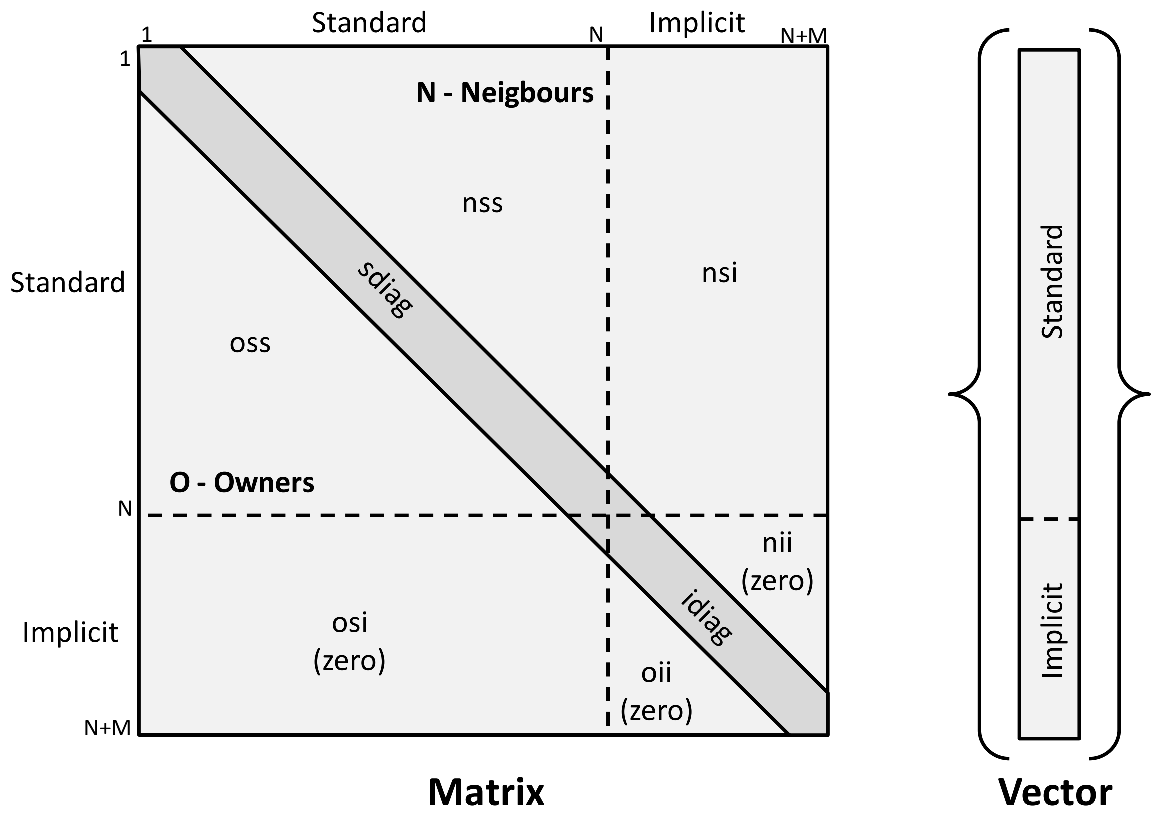


Figure 17: Block matrix indexing

OpenFOAM breaks down the matrix into three components: the diagonal, and the lower and upper triangles. It identifies off-diagonal terms according to their position: those in the lower triangle are *owner* terms, and those in the upper triangle are *neighbour* terms. The distinction is required because OpenFOAM handles unstructured grids, and direction is unknown. CRAFTS breaks down the matrix further, resulting in eight components:

- sdiag: the diagonal terms for standard variables;
- idiag: the diagonal terms for implicit variables;
- oss: owner terms relating standard variables to each other;
- nss: neighbour terms relating standard variables to each other;
- nsi: neighbour terms relating standard variables to implicit variables;

- osi: owner terms relating implicit variables to standard variables;
- oii: owner terms relating implicit variables to each other; and
- nii: neighbour terms relating implicit variables to each other.

Recall that implicit variables are only included in the matrix solution as part of source term stabilisation coupling with the algebraic routines. Although their solution is already known, it must be duplicated in the matrix by giving setting an appropriate source term. For this reason, the matrix terms differ considerably between standard variable and implicit variable rows (not columns). That is the horizontal dotted-line in Figure 17. For the implicit rows, all off-diagonal terms (osi, oii and nii) are zero. The diagonal term and source term are a linear approximation of the known solution. For standard variable rows, CRAFTS uses Newton-Raphson linearization for the reaction source terms. This consists of an implicit term added to the matrix and an explicit term added to the source term. These are given by:

$$\begin{aligned} a_{p_{i,j}} &= -\frac{\partial r_i}{\partial y_j} = -\nu_{i,k}\frac{\partial \rho_k}{\partial y_j} - \frac{\partial \nu_{i,k}}{\partial y_j}\rho_k, \text{ and} \\ b_{p_i} &= -\sum_k \frac{\partial r_i}{\partial y_k} y_k. \end{aligned} \tag{61}$$

CRAFTS first sets the transport terms, and subsequently adds these terms.

3.6.5 Adaptive timestepping

The theory behind adaptive timestepping is presented in Section 2.1.3 on page 26. CRAFTS uses a forward-looking adaptive timestepping method that was not previously available in OpenFOAM. The method employs timestep-doubling in order to estimate the error, and applies Equation (3), repeated here,

$$h_{new} = h_{old}\left|\frac{\Delta_{new}}{\Delta_{old}}\right|^{\frac{1}{5}}, \tag{3}$$

to the result at every step. The flowchart of this method is shown in Figure 4 on page 28. The user can specify the convergence levels for each variable, as well as the minimum error scaling factor.

Optionally, the flow solver can be included in the adaptive timestepping. In other words, the error levels of flow variables such as pressure, *p*; mass flux, *phi*; and turbulence, *R*, are included in the timestep calculation.

However, early trial and error proved that error-based timestep adjustment alone was insufficient. With error-based timestep adjustment, the objective is to find the maximum timestep that produces an acceptable error level. This usually translates into a faster simulation, since fewer timesteps are required; however, there are numerical systems where, beyond a threshold, larger steps take exponentially longer computation time than shorter ones, and consequently taking more, shorter timesteps will result in a faster simulation. The gas model is one such numerical system. To accommodate this problem, CRAFTS uses *performance feedback* to inform its timestep adjustment.

To implement performance feedback, CRAFTS measures the performance of the coarse step and the two fine steps and compares them. The user can choose the measure to be either CPU time, useful when each step takes more than a second; or iterations, useful when each step is too fast to register significant values of CPU time. There are two obvious extremes in designing performance feedback:

1. if the two fine steps were significantly faster than the single coarse step, then halving the timestep would be beneficial; and
2. if there was no appreciable difference in performance, then using the timestep calculated by the error-based method would be beneficial.

Performance feedback applies a blending factor between these two extremes, given by:

$$\lambda_p = \begin{cases} 1, & \gamma \geq 1 \\ \dfrac{1 - e^{B\gamma}}{1 - e^{B}}, & \gamma < 1 \end{cases}, \tag{62}$$

where:

- *B* is the bias, a dimensionless user-specified coefficient; and
- γ is the ratio of performance measures, given by:

$$\gamma = \frac{\text{fine 1} + \text{fine 2}}{\text{coarse}}. \tag{63}$$

Figure 18 shows the curve of Equation (62) for various bias values, *B*.

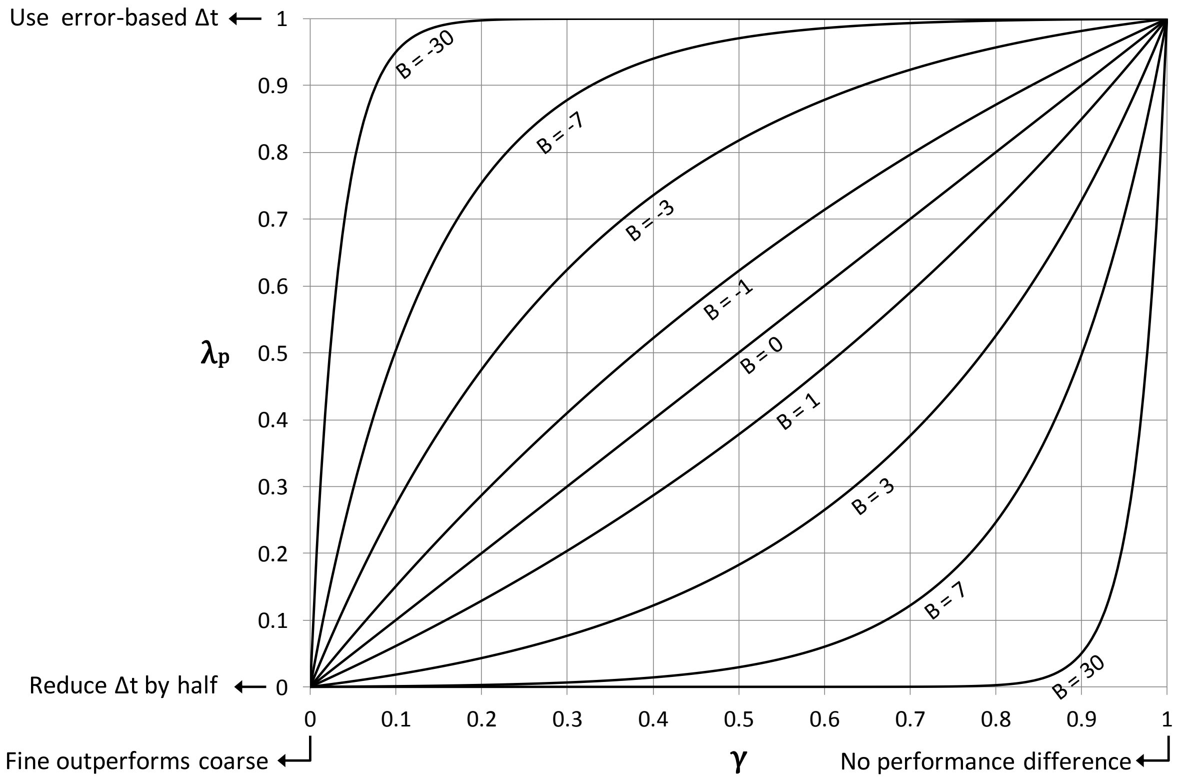


Figure 18: Performance feedback blend factor, λ_p as a function of γ

Performance feedback calculates the next timestep according to:

$$\Delta t_{\text{next}} = \lambda_p \Delta t_{\text{error}} + (1 - \lambda_p)\frac{\Delta t_{\text{old}}}{2}, \tag{64}$$

where:

- Δt_{error} is the next timestep provided by the error-based adaptive timestepping; and
- Δt_{old} is the last timestep used.

Performance feedback does not force a retry of the timestep should the performance be inadequate, therefore it is a backward-looking adaptive timestepping strategy. However, the error-based adaptive timestep control is forward-looking. Forward-looking adaptive timestepping cannot be used without implementing state control.

3.6.6 State control

State control is the ability for a software system to store and retrieve its transient data in order to return to a previous condition. As shown in Figure 4 on page 28, forward-looking adaptive timestepping has instances where the state is saved and loaded. Additionally, returning to a previous state is implied when performing the same step twice. OpenFOAM does not have state control. It is designed such that an advance in the timestep renders the previous timestep as a fait accompli. Therefore, many modifications had to be made to the core libraries of OpenFOAM to accommodate this, including:

- the Field and GeometricField classes, i.e. flow variables;
- the transport models;
- the viscosity models; and
- the turbulence models.

CRAFTS implements state control by overriding the main database class, `Time`, and by including state control accommodations throughout its architecture. Classes that implement state control in CRAFTS are:

- `admTime`;
- `craftsModel`;
- `craftsUdfs`;
- `reactionReader`; and

- `admVariableManager`.

Classes implementing state control have five additional functions:

- `saveState`: write the transient variables to a given save state slot;
- `clearState`: erase the transient variables from a given save state slot – invalidates the slot;
- `loadState`: read the transient variables in a given save state slot;
- `nStates`: report how many valid save state slots exist; and
- `validState`: returns true if the given state slot contains a valid save state.

Since several saves can be stored simultaneously, all saved data must be stored as lists. For instance, to add a scalar to state control, the class requires a scalar list; similarly, to add a scalar list to state control, the class requires a list of scalar lists, and so on.

One challenging aspect of implementing state control was in the variable manager: handling its built-in old timestep archives. OpenFOAM dynamically stores old timestep values for use with time derivative discretization schemes. It automatically detects how many old timesteps it requires, storing only what it needs. State control functions require a careful series of old timestep pointer reassignments for each variable.

3.6.7 User-defined functions

In order to accommodate the diverse modifications that researchers apply to ADM1, CRAFTS has a framework for user-defined functions (UDFs). This framework allows CRAFTS to isolate custom ADM1-specific code from the general coupled reaction model, which is otherwise applicable to many other processes beyond anaerobic digestion. All algebraic routines, including the ion model and the gas model, are implemented as UDFs.

The UDFs are encapsulated in a single user-created class that inherits `craftsUdfs`. Several examples exist in the release, and can be used as templates for creating new UDF objects. The examples include:

- `craftsEmptyUdfs`, which contains no algebraic routines, reducing ADM-MDA to a coupled PDE solver, as opposed to a coupled PDAE solver;
- `craftsSh2IonsUdfs`, which solves ADM1's ion model, and uses static equilibrium to solve S_{H2}; and
- `craftsSh2IonsGasUdfs`, which is the same as `craftsSh2IonsUdfs`, but it also solves the gas model.

3.6.7.1 Function hooks

A function hook is a position in the solver algorithm where a UDF is called. The six function hooks available in ADM-MDA are, in calling order:

- `initializeTimestep`, before the start of a timestep;
- `applyVariableLimits`, after the solution of the coupled reaction model (optional);
- `initializeImplicitLoop`, before the start of the implicit loop;
- `implicitLoop`, the implicit loop;
- `finalizeImplicitLoop`, at the end of a successful implicit loop; and
- `finalizeTimestep`, at the end of the timestep.

Figure 19 shows the relative position of each function hook from the scope of the `step()` function, which is one step for the reaction solver in CRAFTS. The implicit loop is unique, as this is what extends the reaction solver of CRAFTS from a coupled PDE solver to a coupled PDAE solver, as it contains all the algebraic routines for the problem. The initialize and finialize functions are required for transient algebraic routines.

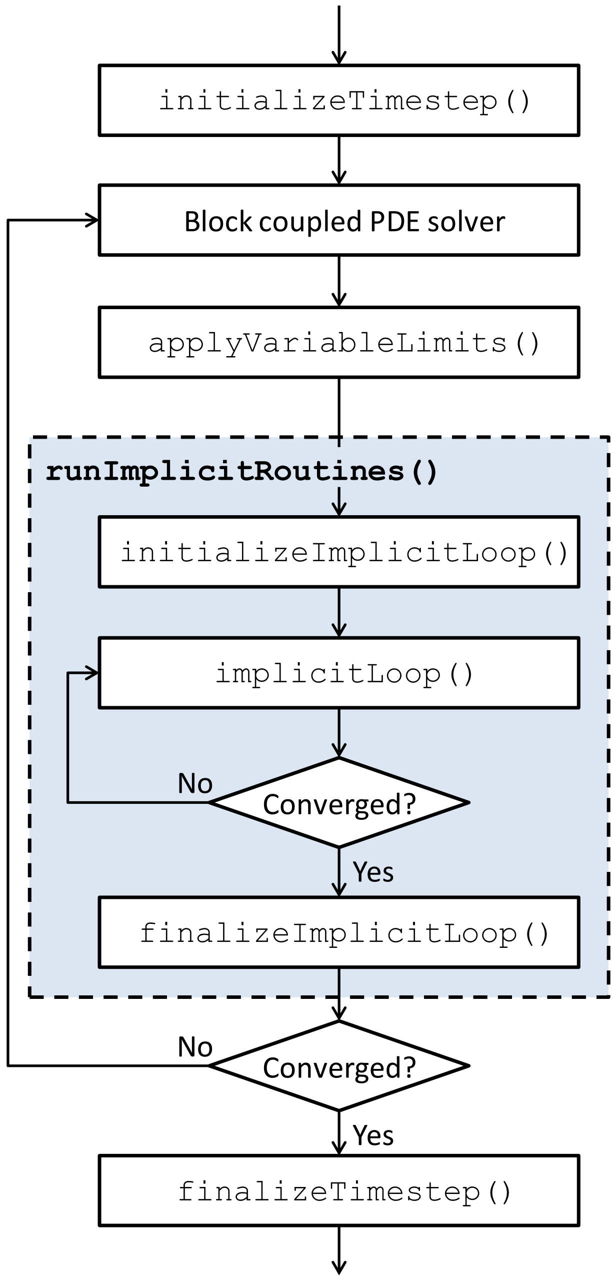


Figure 19: Function hooks flowchart

3.6.7.2 Customizing input files

Some UDF designs require user-specified settings. There are two mechanisms for taking input from the case text files. First, for global settings, the UDF input file admHooksDict is available. The UDF can lookup values using dictionary commands on the object, such as:

```
    convergence = readScalar(admHooksDict_.lookup("convergence"));
```

Settings that are associated with variables can be added to the `admVariableDict` and looked up using the variable object itself, such as:

```
    relaxation = readScalar(Svar.lookup("relaxation"));
```

3.6.7.3 Standard variables

Even though the coupled reaction model outputs standard variables, UDFs are free to change standard variables, provided the input file `admVariableDict` identifies each affected variable with a `changedByUdf` flag. CRAFTS automatically checks all `changedByUdf` standard variables for changes after a UDF, and accommodates the change using source term stabilisation coupling.

3.6.7.4 Implicit variables

Implicit variables can only be solved by UDFs or by CRAFTS' built-in static equilibrium solver, `implicitAutoSolve`.

3.6.7.5 Derived variables

Derived variables are not designed to have a value assigned to them, but it is possible. Rather than assigning the value directly to the derived variable, which is impossible, the other variables on which it depends must be changed. The user must first define a set of *reverse functions* in the `admVariableDict`. Reverse functions are the equations that the object uses to change the other variables on which it depends. For example, the concentration of carbon dioxide is a derived variable, given by:

$$S_{CO2} = S_{IC} - S_{HCO_3^-}.$$

Two possible reverse functions are:

$$S_{IC} = S_{CO2} + S_{HCO_3^-}, \text{ and}$$

$$S_{HCO_3^-} = S_{IC} - S_{CO2}.$$

Either or both may be used.

3.6.7.6 *Coupling within UDFs*

Since CRAFTS has no control over what users include in UDFs, the strongest coupling it can provide is point stabilisation coupling, a coupling strategy discussed in Section 2.1.9 on page 40. CRAFTS applies this type of coupling to the UDF routines. This coupling strategy works well for equilibrium routines, but may be inadequate for transient algebraic routines. Should a more sophisticated coupling be required, such as source term stabilisation coupling, the user must implement it directly into the UDF. CRAFTS provides an example of this in `craftsSh2IonsGasUdfs.C`.

3.6.7.7 *Return values*

The main functions involved in the reaction solver of CRAFTS are:

- `solve()` is the top-level function that manages the adaptive timestepping using the results from `coarseStep` and `doubleFineStep`;
- `coarseStep()` and `doubleFineStep()` solve the flow using `stepFlowDispatch` and the reactions using `stepReactionsDispatch` for one large step and two small steps respectively;
- `stepFlowDispatch(stepType)` is the dispatch function that calls the flow solver, and handles sub-stepping;
- `stepReactionsDispatch(stepType)` is the dispatch function that runs the reaction solver using `stepReactions`;
- `stepReactions()` performs one step in time, solving the coupled reaction model and the algebraic routines using `runImplicitRoutines`, and tests convergence using point stabilisation coupling with the standard variables;

- `runImplicitRoutines()` solves the algebraic routines using `implicitLoop`, and using point stabilisation coupling on the implicit variables for convergence;
- `implicitLoop()` is the UDF that performs all the algebraic routines.

There are many types of failures the solver algorithm can encounter, such as convergence failures, mass balance failures, and so on. For every detectable failure, the solver has a contingency. For example, should the mass balance fail within the gas model, the solver jumps back from the `implicitLoop()` function to the `stepReactions()` function in order to repeat the reaction solution. Similarly, should the Newton-Raphson solver in the ion model exceed its maximum iterations, the solver jumps from the `implicitLoop()` function all the way back to the `solve()` function, effectively declaring the entire timestep to be a failure, and retrying with a smaller step size.

With four levels of embedded functions, CRAFTS requires a clear, formal method of handling failures. It achieves this using integer return values, as shown in Figure 20. The dispatch functions are not shown, as they do not generate their own return values.

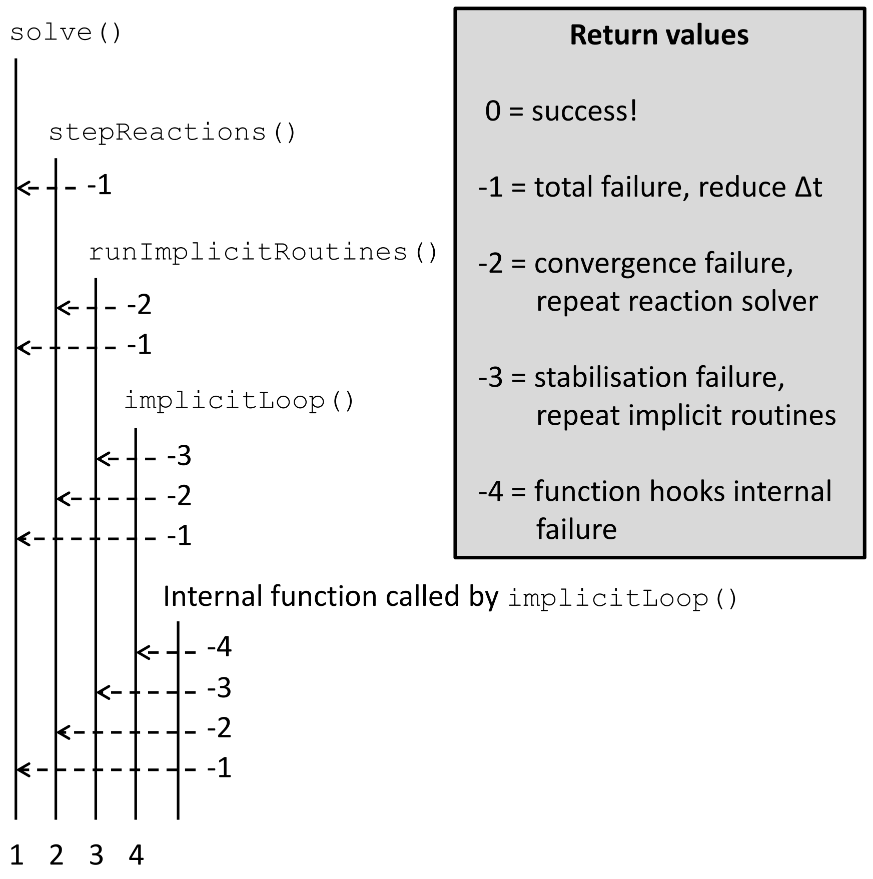

solve()
stepReactions()
-1
runImplicitRoutines()
-2
-1
implicitLoop()
-3
-2
-1
Internal function called by implicitLoop()
-4
-3
-2
-1
1
2
3
4
Return values
0 = success!
-1 = total failure, reduce Δt
-2 = convergence failure, repeat reaction solver
-3 = stabilisation failure, repeat implicit routines
-4 = function hooks internal failure

Figure 20: Steady-state flow solver return values

3.6.7.8 State control

Data stored by a UDF that changes between timesteps must be managed by the state control of the function hook object, otherwise, should a timestep be rejected, the UDF will not reset to the correct position. State control is covered in more detail in Section 3.6.6 on page 87.

3.6.7.9 Iterations

CRAFTS tracks its own performance in order to implement performance feedback adaptive timestepping, covered in Section 3.6.5 on page 84. The model requires UDFs to report their own iterations. This can be achieved using the code:

```
    model.lastStepIterations() += n;
```

where *n* is the number of iterations to add.

3.6.7.10 Ion model

Like ADM1, the ion model in ADM-MDA solves Equations (21)-(30) using the Newton-Raphson method. Unlike ADM1, it performs this solution simultaneously across the entire flow field. It is implemented directly as a UDF in `craftsSh2IonsGasUdfs.C` and `craftsSh2IonsUdfs.C`. Trial and error revealed that the ion model may be prone to failing due to excessive precision requirements. To accommodate this possibility, it detects precision failure conditions, and forces the model to retry the step with a smaller timestep value.

3.6.8 Static equilibrium

In the CRAFTS source code, static equilibrium is called *implicitAutoSolve* and is implemented in `craftsModelImplicitAutoSolve.C`. Static equilibrium finds the equilibrium value for a given variable, assuming all other variables are constant. It is the only “built-in” solver for implicit variables, in that it is not *implemented* in a UDF. However, it must be *called* by a UDF, which gives the user full control over when each static equilibrium function will operate. It can be used to solve any number of variables (one at a time), in any sequence, and at any point in a UDF. ADM-MDA uses it to solve for the dissolved gas concentration of hydrogen, S_{H2}.

Recall from Section 3.6.4.3 on page 81, the coupled reaction model does not insert the correct terms into the implicit variable rows; rather it uses source term stabilisation coupling. Static equilibrium works by solving one row for an implicit variable in the coupled reaction model. It inserts the correct terms and assumes all other variables are constant. It is based on the Newton-Raphson method, with modifications to account for species transport.

Static equilibrium solves the equation given by:

$$Ax = B, \tag{65}$$

where *A* is given by:

$$A = a_p - \frac{\partial r}{\partial x}, \tag{66}$$

and *B* is given by:

$$B = b_p - \sum_i a_i x_i + r - \frac{\partial r}{\partial x} x^*, \tag{67}$$

where:

- a_p is the diagonal transport coefficient;
- a_i are the neighbour transport coefficients;
- x_i are the neighbour variable values;
- r is the reaction source term; and
- x^* is the value from the previous iteration.

Equations (65) to (67) give:

$$x = \frac{b_p - \sum_i a_i x_i + r - \frac{\partial r}{\partial x} x^*}{a_p - \frac{\partial r}{\partial x}}. \tag{68}$$

Static equilibrium solves this equation iteratively.

3.6.9 Gas model

ADM-MDA models the gas volume above the fluid using a bulk model. On the other hand, the liquid volume is spatially resolved. These two volumes are intricately connected with their feedback effects: the gas partial pressures reduce the gas transfer from the liquid; and the gas transfer from the liquid increases the gas partial pressures. The gas partial pressures, in turn, are regulated by the flow rate of gas out of the gas volume, a quantity that, in practice, hovers very close to zero and introduces discontinuities into the numerical model. Furthermore, the principles governing gas transfer out of a liquid, known as Henry's Law, are understood from a bulk perspective; whereas the principles governing discrete mass transfer through a free surface in a spatially resolved model, known as mass diffusion, only relate mass transfer to concentration gradients and give no insight into how much mass should transfer. CRAFTS' implementation of ADM-MDA needs to find a stable feedback method between bulk and discrete volumes, and bridge the gap between Henry's Law and mass diffusion.

The gas model in CRAFTS comprises two components. First, to directly affect the coupled reaction model, it uses a custom boundary condition at the gas-liquid interface. The boundary condition sets the gradient of the dissolved gas concentration in the liquid by relating Henry's Law to diffusion mass transfer. Second, it performs the bulk of the gas model calculations in a transient algebraic routine. A set of tests are applied to ensure the coupled reaction model and the gas model agree before proceeding to the next timestep.

3.6.9.1 Gas transfer boundary condition

The boundary conditions affect the coupled reaction model directly. The mass transfer of gas across the boundary surface can be calculated in two ways, using different principles. These can be applied to each control volume on the liquid-gas boundary. From the gas volume side, Henry's Law can be arranged to state that the rate at which dissolved gas escapes a liquid is given by:

$$\dot{m}_i = k_L a V_l \left(\bar{S}_i - n_i K_{H,i} p_i \right) , \tag{69}$$

where:

- $\dot{m}_i$ is the mass transfer rate in dimensions [kg s^{-1}] or [mol s^{-1}], depending on the chosen measure* (see below);
- $k_L a$ is the overall gas-liquid transfer coefficient, in dimensions [s^{-1}];
- V_l is the volume of the liquid;

- $\bar{S}_i$ is the volumetric averaged concentration of gas specie *i*, dissolved in the liquid, in [kg m^{-3}] or [mol m^{-3}], depending on the chosen measure;
- n_i is the conversion factor from moles to kilograms for gas specie *i*, or unity, depending on the chosen measure;
- $K_{H,i}$ is Henry's coefficient for specie *i*, in dimensions [mol s^2 kg^{-1} m^{-2}], which is a function of temperature, as shown in the Appendix, Section D.2.2; and
- p_i is the partial pressure of specie *i* in the gas volume above the surface of the liquid.

*** NOTE:** The units used to measure the quantity of a substance can be either moles or kilograms. ADM1 uses both: hydrogen and methane are measured using kilograms; whereas carbon dioxide is measured using moles. CRAFTS' implementation of ADM-MDA follows this convention, which forces unexpected flexibility requirements into the gas model.

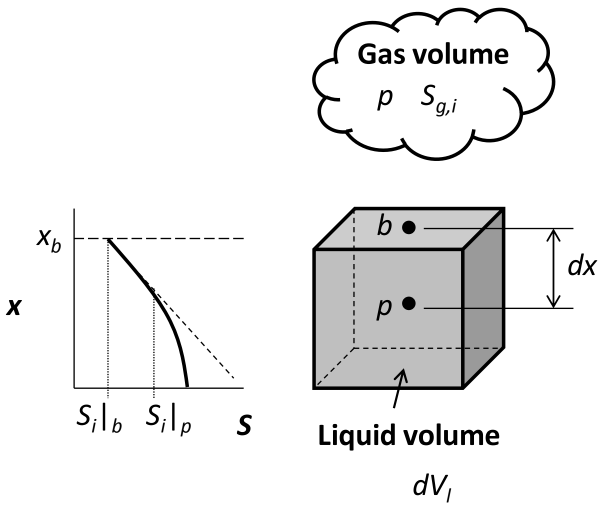


Figure 21: Gas boundary condition

From the liquid side, the mass transfer across the boundary is caused by diffusion. Diffusion relates the concentration gradient near the surface, as shown in Figure 21, to the mass transfer. The diffusion mass transfer from one control volume is given by:

$$d\dot{m}_i = -\Gamma \frac{\partial S_i}{\partial x} dA \approx -\Gamma \left(\frac{S_i|_b - S_i|_p}{dx} \right) dA, \tag{70}$$

where:

- Γ is the coefficient of diffusion, with dimensions [$m^2\ s^{-1}$];
- $S_i|_b$ and $S_i|_p$ are the dissolved gas concentration at the boundary surface and at the centre of the control volume, respectively; and
- dA is the boundary surface area.

Henry's Law, Equation (69), is an empirical relation that only applies to the bulk fluid. In order to use it in a finite volume boundary condition, the gas model assumes the mass transfer is uniform across the gas-liquid interface. With this assumption and Equations (69) and (70), the gradient at the boundary surface is:

$$\frac{S_i|_b - S_i|_p}{dx} = -\frac{k_L a V_l}{A\Gamma} \left(\bar{S}_i - n_i K_{H,i} p_i \right). \tag{71}$$

The partial pressure, p_i are treated as constant by the boundary condition. They are calculated in the gas transfer algebraic routine.

3.6.9.2 Gas transfer algebraic routine

A mass balance in the gas volume gives:

$$\frac{dm_{g,i}}{dt} = -S_{g,i} Q_g + \sum_{\text{surface}} d\dot{m}_{d,i} + \sum_{\text{volume}} d\dot{m}_{b,i}, \tag{72}$$

where:

- $m_{g,i}$ is the mass of specie *i* in the gas volume;
- $S_{g,i}$ is the concentration of specie *i* in the gas volume;
- Q_g is the volume flow rate of gas being withdrawn from the gas volume;
- $d\dot{m}_{d,i}$ is the mass flow rate of gas transferring into the gas volume through diffusion from the liquid; and

- $d\dot{m}_{b,i}$ is the mass flow rate of gas transferring into the gas volume through bubble formation.

Dividing by the gas volume gives the mass balance in terms of concentrations:

$$\frac{dS_{g,i}}{dt} = -\frac{S_{g,i}Q_g}{V_g} + \frac{1}{V_g}\left(\sum_{\text{surface}} d\dot{m}_{d,i} + \sum_{\text{volume}} d\dot{m}_{b,i}\right), \tag{73}$$

This is an ODE set with time as the independent variable and the gas concentration for each specie, S_{gi}, as dependent variables. From the relations presented in Section 2.2.6 on page 51, it is apparent that the gas flow rate, Q_g is a function of the dependent variables, and is given by:

$$Q_g = K_p\left(RT\sum_i \frac{S_{g,i}}{n_i} + p_{H_2O} - p_{atm}\right). \tag{74}$$

Equation (73) is coupled to all dependent variables through Equation (74), giving:

$$\frac{dS_{g,i}}{dt} = -S_{g,i}\frac{K_p}{V_g}\left(RT\sum_i \frac{S_{g,i}}{n_i} + p_{H_2O} - p_{atm}\right) + \frac{1}{V_g}\left[k_L a V_l\left(\bar{S}_i - n_i K_{H,i} p_i\right) + \sum_{\text{volume}} d\dot{m}_{b,i}\right]. \tag{75}$$

The resulting ODE set is very stiff, requiring high precision and unacceptably small timesteps. The source of this numerical instability is the gas flow rate.

3.6.9.3 Gas flow rate

The gas flow rate is given by Equation (41), repeated here:

$$Q_g = K_p\left(p_{tot} - p_{atm}\right)\frac{p_{tot}}{p_{atm}}. \tag{41}$$

Should the flow be negative, the digester would draw biogas in from the atmosphere, which is physically unrealistic. To prevent this, ADM1 assumes there is a check valve in the gas outlet, and sets Q_g to zero when it is less than zero. This assumption makes sense, but its implementation causes considerable numerical instability in the ODE set for the gas model. CRAFTS implementation of ADM-MDA addresses this by using a logistic curve function. The gas flow equation can be written more succinctly as:

$$Q_g = K_p\left(p_{tot} - p_{atm}\right)\frac{p_{tot}}{p_{atm}}H, \tag{76}$$

where *H* is the logistic function that depends on p_{tot} and p_{atm}. In ADM1, the *H* function is a step function, given by:

$$H = \begin{cases} 1, & p_{tot} - p_{atm} > 0 \\ 0, & p_{tot} - p_{atm} \leq 0 \end{cases}. \tag{77}$$

To improve the numerical performance, it is necessary to modify this function in two ways. First, the transition between on and off needs to be smooth. Second, and more important, the transition needs to begin at a non-zero pressure. This pressure offset can be interpreted as the *cracking pressure* of the check valve. This is achieved with the logistic function:

$$H = \left[1 + exp\left(1 - 2K_c\left(p_{tot} - p_{atm} - p_c\right)\right)\right]^{-1}, \tag{78}$$

where:

- K_c is the gas cut-off coefficient, a user-defined constant describing the sharpness of the cut-off curve; and
- p_c is the check valve cracking pressure.

The step function and logistic function are shown in Figure 22.

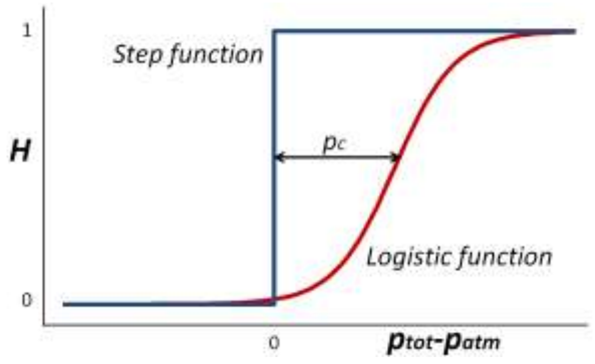


Figure 22: Step function and logistic function for gas flow rate

From the relations presented in Section 2.2.6 on page 51, it is apparent that Q_g is a function of the dependent variables, and is given by:

$$Q_g = \frac{\frac{K_p}{p_{atm}}\left(RT\sum_i \frac{S_{g,i}}{n_i} + p_{H_2O} - p_{atm}\right)\left(RT\sum_i \frac{S_{g,i}}{n_i} + p_{H_2O}\right)}{1 + exp\left(1 - 2K_c\left(RT\sum_i \frac{S_{g,i}}{n_i} + p_{H_2O} - p_{atm} - p_c\right)\right)}. \tag{79}$$

Compared with Equation (74) this modified form is far more complicated, but critically, also more stable.

When this is substituted into the ODE set, the result is:

$$\begin{aligned}\frac{dS_{g,i}}{dt} = &- \frac{\frac{K_p S_{g,i}}{p_{atm} V_g}\left(RT\sum_i \frac{S_{g,i}}{n_i} + p_{H_2O} - p_{atm}\right)\left(RT\sum_i \frac{S_{g,i}}{n_i} + p_{H_2O}\right)}{1 + exp\left(1 - 2K_c\left(RT\sum_i \frac{S_{g,i}}{n_i} + p_{H_2O} - p_{atm} - p_c\right)\right)} \\ &+ \frac{1}{V_g}\left(k_L a V_l\left(\bar{S}_i - n_i K_{H,i} p_i\right) + \sum_{\text{volume}} d\dot{m}_{b,i}\right)\end{aligned}. \tag{80}$$

This ODE set is still very stiff, requiring on the order of 100,000 ODE timesteps for every timestep the coupled reaction model takes. However, since the gas model is mostly independent of the mesh size, it is scalable.

The last term in Equation (80), $d\dot{m}_{b,i}$, is the bubble source term, and its use requires a bubble model.

3.6.9.4 Bubble model

The liquid volume can only hold a limited amount of dissolved gas before the gas comes out as bubbles. This is known as the *saturation point*, $S_{c,i}$, and is given by:

$$S_{c,i} = K_{H,i} R \frac{S_{g,i}}{n_i}. \tag{81}$$

Once the coupled reaction model completes a timestep, the dissolved gas concentrations may have moved above the saturation point. The bubble model removes the excess from the liquid volume and adds it to the gas volume. A simple way to handle this is with a first order empirical relation, giving the rate of bubble formation as:

$$d\dot{m}_{b,i} = k_b\left(S_i - S_{c,i}\right) dV_l, \tag{82}$$

where k_b is the coefficient of bubble formation, with units [s^{-1}]. Equation (82) only applies when $S_i > S_{c,i}$. The total bubble source term is determined by summing this equation over all control volumes.

3.6.9.5 Gas model coupling

The gas transfer boundary condition and gas transfer algebraic routine are coupled. This is managed by the gas transfer algebraic routine, which checks convergence in two ways, as shown in Figure 23. First, it uses point stabilisation coupling on the gas volume concentrations, initializing the first values with linear extrapolation based on previous values. Second, it compares the mass transfer from the liquid volume against the mass transferred to the gas volume. If these do not agree sufficiently, the coupled reaction model is repeated with an updated boundary condition.

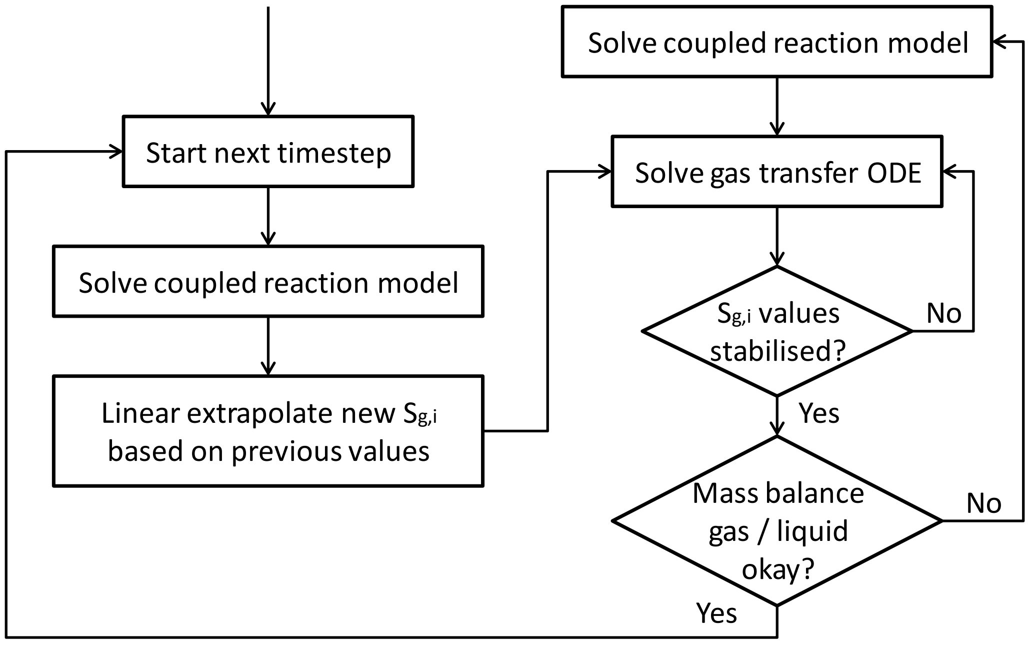

Start next timestep
Solve coupled reaction model
Linear extrapolate new $S_{g,i}$ based on previous values
Solve coupled reaction model
Solve gas transfer ODE
$S_{g,i}$ values stabilised?
No
Yes
Mass balance gas / liquid okay?
No
Yes

Figure 23: Gas model convergence flowchart

In addition to this, the bubble model modifies the dissolved liquid concentrations of the gas species, variables that are also outputs of the coupled reaction model. This is handled with source term stabilisation coupling, which is built directly into the model.

3.6.9.6 Pseudo Gas Model

Due to the numerical complexity of the gas model, a simplified gas model was created solely for verification purposes. The full gas model complicates verification for three reasons:

1. it is good practice to verify the liquid coupled reaction model without having the gas model confound the results;
2. since the gas model requires a gas-liquid interface on one boundary, it necessarily introduces spatial variation for all meshes with more than one control volume along the depth axis: this restricts the mesh size used for verification; and
3. the gas model requires the dissolved gas species to have a non-zero coefficient of diffusion, which can lead to diffusion mass loss through the inlet and outlet: this restricts the verification to cases with no flow.

Therefore, for verification purposes, an alternate implementation of ADM1 with a *pseudo gas model* has been developed and is implemented in ADM-MDA and the bulk model. Rather than requiring a gas volume, the pseudo gas model moves all gas variables into the liquid control volumes. That is, each control volume has its own independent set of gas pressures, gas flow rates, and so on. The pseudo gas model duplicates much of the numerical behaviour of the standard gas model, while removing the aspects that complicate verification against a bulk model.

3.7 Supporting structures

This section details the supporting code structures on which the development, or implementation of CRAFTS depends. These include:

- the bulk model;
- the **equationReader** extension;
- the **multiSolver** extension;
- the **plcEmulator** extension;
- the VectorN library; and
- the block matrix tool set.

3.7.1 Dependencies

Figure 24 shows the direct dependents of all the support structures in CRAFTS.

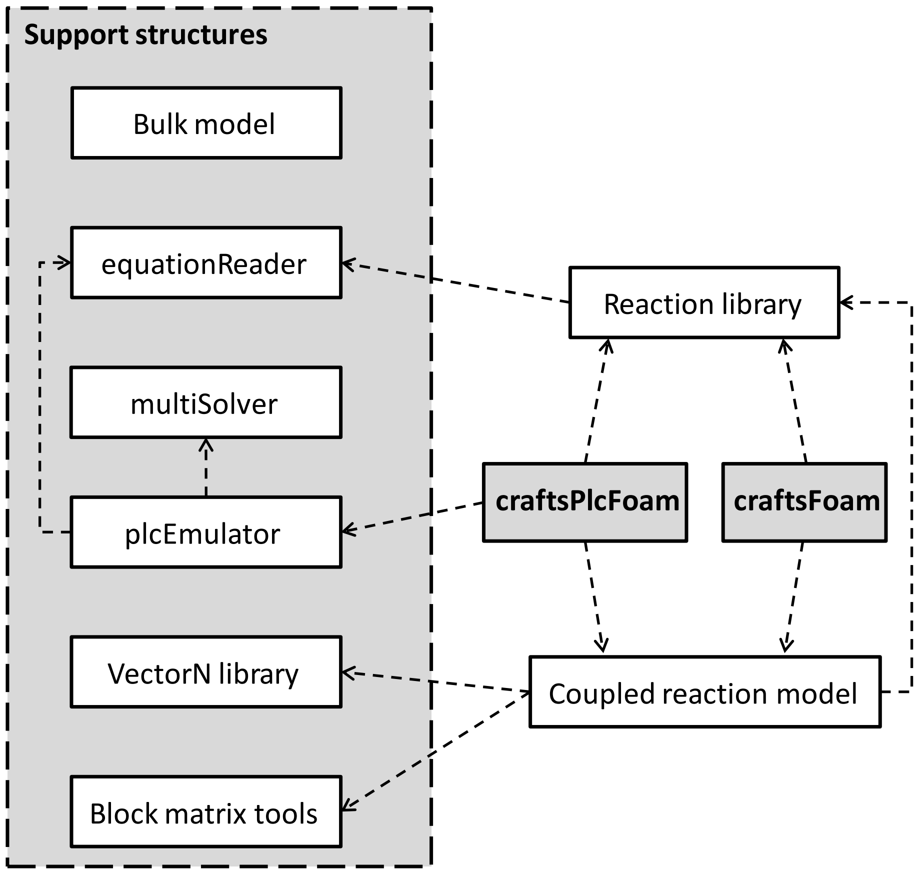


Figure 24: Support structure dependents

3.7.2 Bulk model

CRAFTS does not depend on the bulk model; however, it is an indispensible tool for code verification. The bulk model is implemented in Excel, and uses Visual Basic macros for its numerical algorithms. The implementation closely parallels that of the MATLAB implementation of ADM1. Two ODE solvers are written: Euler's method, and semi-implicit Bulirsch-Stoer. Parts of the numerical driver, as well as the ODE solver implementation adapt code from Press et al. (1992).

3.7.3 equationReader

In order to achieve its objectives on flexibility and user-friendliness, the design of CRAFTS demanded the ability to read and parse user-entered equations. OpenFOAM does not have this capability. Other CFD software suites do, for instance, CFX features an equation parser that it calls the common expression language (CEL). However, CRAFTS requires OpenFOAM to have this capability, and therefore, **equationReader** was created. CRAFTS depends on **equationReader** extensively. **equationReader** is capable of:

- dimension checking;
- working with all types, including scalars, vectors and tensors;
- working with all sizes, including single elements, fields, and GeometricFields, which are fields with the boundaries included;
- working with a variety of data sources, including native OpenFOAM variables, variables to be read from files, and its own custom equationVariable, used for lazy-evaluation; and
- automatically performing equation substitution to arbitrary depth, with built-in circular-dependency detection.

equationReader works by converting a user-entered equation into a set of operations with links to the appropriate data sources. For instance, it will convert the equation: “y0 + y1 * 2” into:

1. retrieve y1;
2. retrieve constant “2”;
3. multiply;
4. retrieve y0;
5. add; and
6. return the result.

This conversion process, known as parsing, is slow, but is performed only once. Equation evaluation is a for loop through all the operations. However, despite much optimization, evaluation is slower than it would be if the equations were hard-coded. For a simple Geometric Field calculation, **equationReader** takes approximately seven times longer than a hard-coded equation. For a single scalar, **equationReader** can take as much as 300 times longer. This is caused by the fact that the equations are interpreted, which means the path through the code cannot be predicted, and therefore cannot be optimised by the compiler. This extension is detailed in Appendix H.

3.7.4 multiSolver

The multiSolver extension was originally implemented to accommodate an earlier design feature in CRAFTS: algorithm-switching. This is where a solver changes the algorithm it is using in the middle of a simulation. Algorithm-switching is no longer necessary in CRAFTS, as the flow model is now solved in series with the reaction model, however it remains a viable option for numerical solvers.

OpenFOAM was not designed to handle algorithm-switching: it cannot switch solver algorithms in the middle of a simulation, nor can it change boundary-conditions in an arbitrary way. To accommodate this behaviour, **multiSolver** was created. The extension gives OpenFOAM the capability of performing multiple solver algorithms on the same dataset.

multiSolver faces the problem that each algorithm requires different objects in memory, many of which would be in conflict. When switching algorithms, it may be possible to convert most of these objects, but it is not safe, and cannot be used in a general way. The design, therefore, required the algorithm to terminate on its own and clean up its own memory, then launch the next algorithm, rebuilding all objects from scratch. This may seem inefficient, but it is the only safe solution. A second problem **multiSolver** faces is: each algorithm expects different features in the case directory, where the input files are stored. Changing the behaviour of OpenFOAM to allow for multiple case configurations would be too involved; instead, **multiSolver** addresses this problem by taking control of the case directory, and "morphing" it to what the next algorithm expects. That is, **multiSolver** automates what a normal human user would do if they were attempting to switch solver algorithms manually. In a sense, this is like designing a cruise control system for a car that, instead of operating internally, has a robotic foot physically pushing on the gas pedal in response to what it sees on the speedometer through a camera.

multiSolver refers to each configuration of the case directory as a *solver domain*. The user defines each solver domain using a special file format, termed *multiDict*. Like CRAFTS, **multiSolver** was designed for flexibility, and therefore, it has a vast array of options and settings. **multiSolver** writes its data to a different location in the file system than all the other OpenFOAM applications. Therefore, **multiSolver** includes a custom post-processor application of the same name, `multiSolver`, that copies the data to the required location. A detailed overview of **multiSolver** is presented in Appendix I.

Since **multiSolver** completely rebuilds the simulation variables at the start of each solver algorithm, this presents an opportunity to change the boundary conditions, source terms, and any other parameters. As a consequence, **multiSolver** can be used as the foundation of a process control system.

3.7.5 plcEmulator

Anaerobic digestion is a controlled process; however, the inputs, outputs and logic all depend on the specific application. In order for CRAFTS to achieve its objectives on flexibility, it must be able to accommodate a control system in a general way. OpenFOAM does not have a general control system; in fact, no software could be found to feature these capabilities, nor was any reported in the literature. Therefore, the **plcEmulator** extension was created for OpenFOAM. To accommodate the algorithm-switching required by CRAFTS, the PLC emulator provides the required algorithm name as an output.

Process control systems manage a set of outputs by applying a set of logic conditions that determines the system state based on a set inputs. A common process control system is a programmable logic controller (PLC). The **plcEmulator** is an implementation of a PLC directly within CFD software. This extension is detailed in Appendix J.

3.7.5.1 Outputs

The process has several controllable outputs, such as mixers, or heaters. The CFD simulation represents these through changing boundary conditions, source terms, and other simulation parameters. A control state is a specific configuration for each of these outputs. The **plcEmulator** uses the solver domains in **multiSolver** as control states. The user must define all possible control states in the case directory. To change outputs, the **plcEmulator** instructs **multiSolver** to switch solver domains. Additionally, the **plcEmulator** associates an algorithm with each solver domain, which enables algorithm-switching.

3.7.5.2 Inputs

Inputs are the sensors and timers that monitor the simulation. The user specifies the inputs through a text input file. In the source code, the inputs are called *triggers*. The input classes are based on runTimeSelection, and therefore it is easy to add additional input classes. The inputs available to the **plcEmulator** are:

- *equation*: when a user-defined equation is greater than or less than a given limit;
- *variable*: when a field variable is greater than or less than a given limit;
- *timer:* a timer that can repeat, and can be triggered by other inputs;
- *solverDomainGroup:* this input are true or false depending on the control state;
- *solver signal:* a signal sent from the solver to the PLC; and
- *conditional switch:* a switch that changes state depending on the conditions of the other inputs.

The solver signal input can only be used by applications that are specifically written to include it. CRAFTS no longer requires a solver signal input.

3.7.5.3 Logic

The user supplies the logic in a text input file. The logic input file is currently verbose: for every output switch, the file requires approximately 100 lines of user-input. Implementing a logic parser would reduce this considerably; however, informed by the development timeline of **equationReader**, an equation parser, it was decided that the logic parser should be omitted for the present time. The documentation for programming the logic is provided in the Appendix, Section J.4.

3.7.5.4 Sequence of events

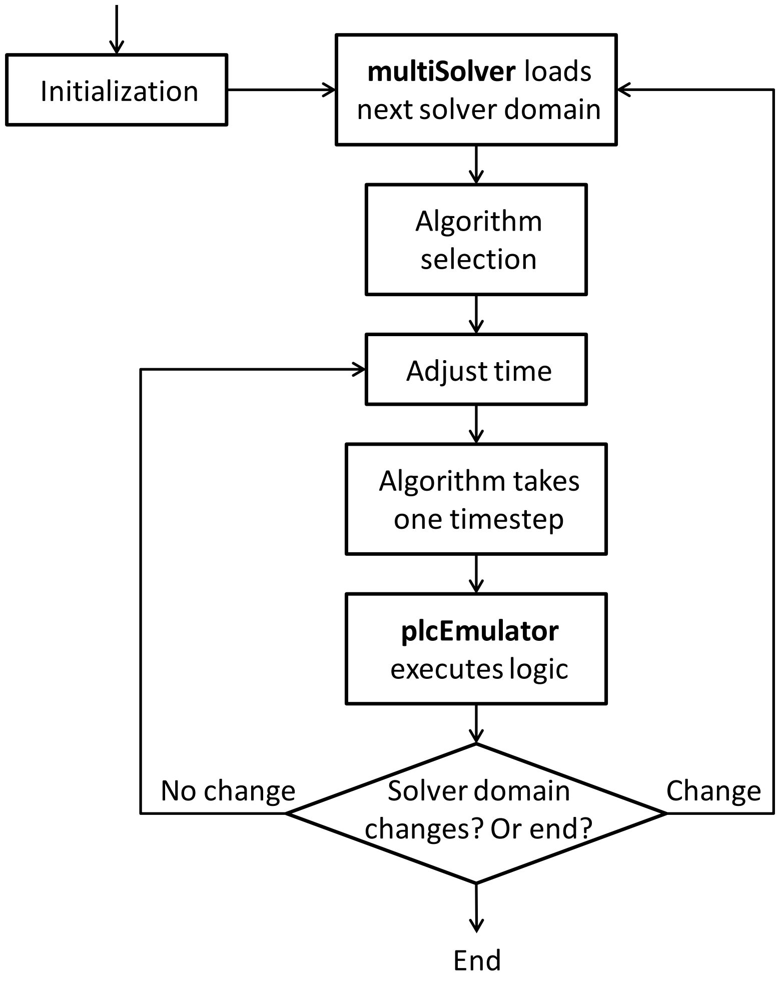

Initialization
multiSolver loads next solver domain
Algorithm selection
Adjust time
Algorithm takes one timestep
plcEmulator executes logic
Solver domain changes? Or end?
No change
Change
End

Figure 25: plcEmulator flowchart

Figure 25 presents a template for the intended use of the **plcEmulator**. Since the emulator uses timers as inputs, it sometimes must modify the end time and timestep of the current simulation in order to properly meet the end of the timer. This modification occurs in the "adjust time" step. Considerable effort went into resolving conflicts that arise when both the solver and the **plcEmulator** attempt to adjust the timestep.

3.7.6 VectorN library

The VectorN library problem was a significant challenge to CRAFTS, one that directly impacted its flexibility. Before being modified, it restricted the number of standard and implicit variables that the model could use. The solution achieves the flexibility requirements, allowing CRAFTS to have any number of variables, as originally designed, but it comes at a cost: when the user changes the number of variables, they may be required to edit an additional file, and recompile several components of CRAFTS.

The VectorN library is a component of OpenFOAM-extend, required by its block matrix solver. Whereas the normal vector type in OpenFOAM contains three elements, the VectorN type defines vector and tensor types of arbitrary size, along with all their derived data types, such as VectorN fields. This is critical for block matrices, since the size of their vectors depend on the number of coupled variables. When the block matrix solver was first released, the VectorN library was hard-coded for only a specific set of VectorN sizes: 2, 4, 6 and 8. In the case of CRAFTS, the VectorN size it requires is equal to the number of standard and implicit variables, a number that can vary arbitrarily. For instance, the "craftsSh2IonsGas" function hooks configuration requires a VectorN size of 34, while the "craftsSh2Ions" configuration requires a size of 37. Restricting the number of variables is unacceptable, given the flexibility requirements of CRAFTS.

The first attempted resolution was to define instantiations from Vector2 to Vector50. This attempt failed to compile. It turns out that compile effort increases exponentially with both the number of instantiations, and the VectorN number. Small distributions such as Vector2-8 and Vector34-35 are practical, but large distributions are unfeasible. The exact limitations depend on the platform and compiler.

The final solution was a collaborative effort between myself and Ivor Clifford, an OpenFOAM developer who parallelized the block matrix solver. Clifford worked on reducing the compile requirements by modifying its template metaprogramming. I worked on making the VectorN instantiations user-selectable. Clifford's modifications considerably improved compile time and performance of the VectorN library, but Vector2-50 remained out of reach.

Allowing the user to choose the distribution of the VectorN library instantiations would give CRAFTS its flexibility, but would impact its usability: it would require users to recompile the VectorN library and most CRAFTS components should the number of variables not be available. Furthermore, it is a challenging solution to implement because the flexibility must occur at compile-time, as opposed to runtime. Runtime flexibility is what software is designed for, and can be implemented in countless ways. Compile-time flexibility means the source code itself must be changeable, something that can only be achieved using static preprocessor directives. In early attempts to solve the problem, these preprocessor directives proliferated through the source code like an infectious disease, spreading throughout the VectorN library and outwards, affecting every piece of code that used the VectorN library. This resulted in bloated source code that was difficult to maintain: developers would be required to add 50-100 lines of preprocessor directives everywhere they use the VectorN library. The final solution came in designing a single file to contain all the preprocessor directives. The file is designed to behave like a "for loop", making it easy to use. With this, developers only require a `#define` and `#include` directive where they use the VectorN library.

These modifications were uploaded to the `feature/blockCoupledFVM` branch in the OpenFOAM-extend git repository.

3.7.7 Block matrix tools

Due to complications with the algorithm, the block matrix solver requires its data to be in a different format than conventional GeometricFields in OpenFOAM. The block matrix tools library provides functions that convert the data between the two formats. CRAFTS makes two modifications to the block matrix tools. First, it implements a set of functions that allows the matrix values to be incremented, rather than set. CRAFTS requires this as it sets the matrix values to those obtained from the transport equations, and subsequently increments them several times with reaction contributions. Second, CRAFTS optimizes the code using pointer loops, a strategy that can improve its execution time by more than a factor of ten.

3.8 Code optimisation

Code optimisation is modification of the source code to improve its overall performance. Optimisation can reduce execution time, and increase memory efficiency. Code optimisation should not be attempted until the software is near completion, and no major changes are expected. When CRAFTS was near completion, the entire library, including its support structures, was rewritten and restructured. Informed by the development process, the rewrite implemented several code optimisation strategies, including:

- indexing arrays;
- lazy-evaluation;
- pointer loops; and
- jump tables.

3.8.1 Indexing arrays

Flexibility often comes at the cost of efficiency. Flexible models have to accommodate many possibilities, which means shortcuts are not easy to find. On the other hand, hard-coded models have only one possibility and can be very efficient implementing it. For instance, derivatives in a hard-coded model are straightforward, as the developer can do the calculus ahead of time and learn which variables produce a non-zero derivative. Conversely, flexible models have to test each variable to find the non-zero derivatives.

CRAFTS approaches this problem using a significant number of index lists. It creates index lists for non-zero yields, non-zero evaluations, non-zero derivatives, uniform coefficients, and so on. These lists act as maps, charting a path around the deserts of zeroes and jungles of unnecessary calculations. The first timestep takes the longest, as it has to build these index lists, but subsequently, CRAFTS' performance approaches that of a hard-coded solution.

3.8.2 Lazy-evaluation

Lazy-evaluation is a method that delays calculations until the result is requested. The method stores the result once it is known, and prevents repeating the calculations should it be requested again. It must be able to detect when the inputs to the calculation have changed, warranting a recalculation. The method is beneficial for calculations that may be repeated when their inputs do not change, especially for time consuming calculations.

The **equationReader** necessitates the implementation of lazy-evaluation. As mentioned earlier, for a full field calculation, the **equationReader** takes up to seven times longer than a hard-coded solution. Each equation evaluation is costly, and therefore, it is necessary to minimize the use of **equationReader** evaluations. CRAFTS implements lazy-evaluation in all classes that directly require the **equationReader** evaluation. These are:

- derived variables;
- reaction rate, which stores the rate and associated total inhibition;

- reaction rate inhibition;
- custom coefficients; and
- exponential temperature ratio coefficients, which does not use the **equationReader**, but benefits from lazy-evaluation since it is available.

All these classes store the latest evaluation. To detect when the inputs change, CRAFTS defines a *milestone* variable, managed by the `admTime` class. It is a simple integer, that is incremented each time the inputs change. Lazy-evaluating objects also store the milestone value of their last evaluation. When an evaluation is requested, the object compares its milestone against the current value. If they differ, the object performs the calculation, otherwise it returns the stored value. CRAFTS automatically increments the milestone when it solves the coupled reaction model. The rest of the milestone increments occur within UDFs, and are up to the user:

```
model.setMilestone();
```

This should occur anytime the UDF updates the standard and implicit variables.

equationReader normally uses the values stored in memory for its calculation directly, however it cannot trust the values stored by lazy-evaluating objects. This necessitates a lazy-evaluating equation variable, `equationVariable`, which requests the calculation and returns it to the **equationReader**. The **equationReader** also uses lazy-evaluation for equation parsing, where it does not parse equations until the first time an evaluation is requested. This is not to improve performance, but for logistical reasons: if an equation depends on one that **equationReader** has not read yet, parsing will fail due to an unknown variable.

3.8.3 Pointer loops

Most of the data in CFD software is contained in arrays. Performing operations with these arrays requires iterative processing, or looping. The most straightforward way to code a loop is to use an indexing variable that counts through the elements in the array, such as:

```
for (register i = 0; i < x.size(); ++i)
{
        x[i] = y[i] + z[i];
}
```

Each index access operation, e.g.: `x[i]`, costs time in converting the index number to a memory offset, which is then used to slide past the start of the array to arrive at the correct element. The pointer loop method avoids this access cost, such as:

```
for (register i = 0; i < x.size(); ++i)
{
        *x = *y + *z;
        ++z;
        ++y;
        ++x;
}
```

This method adds three increment operators, e.g. `++x`, but the time savings from avoiding the index access operation, i.e. using `*x` instead of `x[i]`, is significant. Tests showed a tenfold improvement in execution time.

OpenFOAM implements pointer loops in all its field calculations using `#define` macros that change syntax depending on whether the processor is a standard scalar processor, or a high performance vector processor. CRAFTS uses pointer loops wherever possible, implementing them by:

1. using OpenFOAM's field functions when available;
2. when no suitable field function is available, using the macros behind the field functions; and
3. when the macros are not suitable, directly implementing the pointer loop.

3.8.4 Jump tables

A jump table is a table that contains pointers to functions: for each index value, the jump table responds with a function. Function pointers are rarely seen in C++ since virtual functions have largely replaced them; however, virtual functions require a set of derived classes, which is not suitable for all applications, especially jump tables. Jump tables are a fast, efficient way to handle a set of conditionals. The use of jump tables reduces the execution time of CRAFTS by 15% overall.

There are three main constructs for handling conditionals: the `if-elseif-else` method, the `switch-case` method, and the jump table. The jump table is the fastest of these, but also the most difficult to understand and maintain. Therefore, CRAFTS only uses jump tables where the benefits are most significant:

- throughout **equationReader**;
- in the derivative calculations for:
    - reaction rates; and
    - reaction rate inhibitions.

3.8.4.1 Jump tables in equationReader

The **equationReader** parses user-entered equations into a set of operations. When it evaluates an equation, its behaviour depends on the type of each operation. For instance, equationReader translates "y0 + y1 * 2" into:

1. retrieve y1;
2. retrieve constant "2";
3. multiply;
4. retrieve y0;
5. add; and

6. return result.

To evaluate an equation, **equationReader** cycles through all these operations, but at each one it must decide what to do. With an `if-elseif-else` construct, the code might look like:

```
if (operation == "add") { output = x + y; }
elseif (operation == "subtract") { output = x - y; }
elseif (operation == "multiply") { output = x * y; }
else (operation == "divide") { output = x / y; }
```

That is, to evaluate the equation, it must process a conditional at every operation. On the other hand, with a jump table construct, the table points to the correct functions. **equationReader** does not need to evaluate any conditionals; rather it only needs to follow the function pointers. **equationReader** uses jump tables for operations, data retrieval, and debugging switches.

3.8.4.2 Jump tables for derivative calculations

The derivative calculation for a reaction rate or rate inhibition changes behaviour depending on what the "with respect to" variable is. Since the calculation is complex, these objects use a jump table to find the correct function. To achieve this, the table has one entry for every variable in the model, with each entry pointing to the associated derivative function.

For instance, consider an inhibition object whose governing equation is:

$$I = y_2 + C_0(y_3)^2.$$

The possible derivatives with respect to an arbitrary variable, *y*, are:

$$\frac{\partial I}{\partial y_i} = 0,$$

$$\frac{\partial I}{\partial y_i} = 1, \text{and}$$

$$\frac{\partial I}{\partial y_i} = 2C_0 y_3.$$

To implement a jump table, the rate inhibition object creates one function for each possibility, and points the jump table to the appropriate function, as shown in Figure 26.

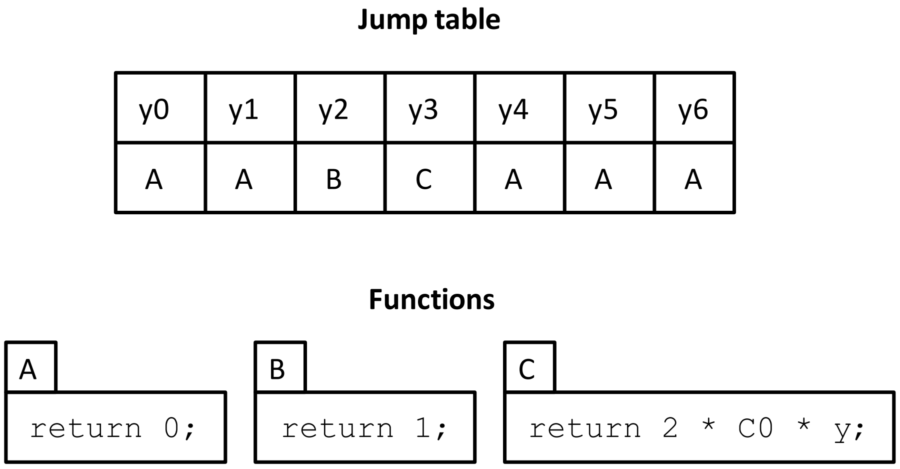


Figure 26: Sample jump table

Since reaction rate and inhibition objects are runtimeSelection objects that the user may extend, a set of macros defined in `admReactionJumpTable.H` streamlines the implementation, similar to OpenFOAM's macro-enabled implementation of runTimeSelection.

Chapter 4 Model verification and use

CRAFTS is a novel solver, and therefore must be subject to a comprehensive series of tests to ensure it is performing as expected before using it to produce data.

4.1 Verification and validation

Verification and *validation* are two important aspects of model development. *Verification* is the process of confirming the model implementation is correct and performs the algorithm as designed. This is typically performed by code developers during model development, and model output is usually expected to be correct to the round-off error of the machine. *Validation*, on the other hand, is the process of comparing model output against physical reality, usually through experimental results. This is typically performed by researchers before each modelling application, and model output is expected to be correct within a reasonable margin of error, such as 5%. All components of CRAFTS are verified; however, it is presently too premature to validate.

4.2 Verification method

The only practical way to achieve verification of a large model is to break it down into smaller components and verify each one with controlled tests. Given its size and scope, CRAFTS required an exhaustive number of these tests, including verifications for:

- basic behaviour, such as:
    - verifying that **equationReader** is properly mapping the parenthesis depth throughout the equation;
    - verifying that **multiSolver**'s file reading and writing is functioning correctly;
    - verifying the reaction library coefficient classes and reaction classes are producing the correct values;

- intermediate behaviours, such as:
    - verifying **equationReader** can substitute equations and detect circular references;
    - verifying that **multiSolver** is able to switch between solver domains on demand;
    - verifying that source term stabilisation is functioning properly and;
- advanced behaviours, such as:
    - verifying that **equationReader** is performing the operations expected and obtaining the correct values;
    - verifying that **multiSolver** is able to manage multiple solvers and solver domains in response to inputs from the **plcEmulator**;
    - verifying species transport in the coupled reaction model by proving that a non-reacting case setup produces identical results to scalar transport; and
    - verifying reactions kinetics in the coupled reaction model by duplicating bulk model results for a single control volume with flow, and for multiple control volumes with no flow.

Verification tests of the smaller components have not been documented. This chapter covers the verification of the larger components, presented in Table 4.

Table 4: Verification overview

Component	Benchmark	Gas model	Fluid injection?	Mesh
equationReader	Hand calculations	-	-	-
multiSolver	Model behaviour	-	-	-
plcEmulator	Model behaviour	-	-	-
Reaction library	Hand calculations	-	-	-
Flow solver	`pisoFoam`, `pimpleFoam`	-	-	-
Scalar transport	`scalarTransportFoam`	-	-	-
Bulk model	ADM1 - MATLAB	Full	Yes	N/A
Biochemical reactions	Bulk model	Pseudo	Yes	1 cell
Gas model	Bulk model	Full	No	1 cell
Coupled reaction model	Bulk model	Pseudo	No	10 x 10

4.3 Direct verification

Several components of CRAFTS can be verified in a straightforward manner. For these components, the expected results are easy to determine. For instance, the **equationReader** extension parses and evaluates user-entered equations. There is no need for statistical analyses, benchmarking, or interpreting percent errors: either it produces the correct result or it fails. For this reason, the direct verification simulations are almost completely arbitrary: they must be designed to ensure they test all the capabilities of the component in question; however, beyond that, they are arbitrary. Therefore, it is unnecessary to provide specific details of any of these verification simulations, such as the geometry, or boundary conditions. The components that can be verified in this manner are:

- the **equationReader** extension;
- the **multiSolver** extension;
- the **plcEmulator** extension;
- the reaction library;
- the flow solver; and
- scalar transport.

4.3.1 equationReader verification

A variety of verification tests were conducted to confirm the functionality of the **equationReader** extension. The **equationReader** has been verified for:

- correct order of operations (e.g. brackets, exponents, multiplication, etc.);
- all the functions it implements (e.g. sin, min, sqrt);
- handling parentheses to an arbitrary depth;
- equation substitution to an arbitrary depth and its circular dependency detection;
- handling a variety of data sources, (e.g. dictionary look-ups, CRAFTS variables, etc.);
- handling a variety of data types, (e.g. scalars, vectors, tensors, etc.); and
- calculating the result for numbers, dimensions, fields, and geometric fields.

4.3.2 multiSolver and plcEmulator verification

Verification of the **multiSolver** and **plcEmulator** extensions is accomplished by creating a simple test case with process control, and confirming the simulation executed the control as required. The results are detailed in Gaden et al. (2012), a copy of which is presented in Appendix L.

4.3.3 Reaction library verification

The reaction library contains all of the model's physical data, such as concentrations, yields, reaction rates, and their derivatives. Although its capabilities are extensive, all of the results it produces can also be calculated by hand. In this manner, all aspects of the reaction library have been verified.

4.3.4 Flow solver verification

There are two flow solvers implemented in CRAFTS: PISO and PIMPLE. Both of these are well-understood algorithms frequently used by the CFD modelling community. OpenFOAM has its own implementations of these algorithms: `pisoFoam` and `pimpleFoam`. Therefore, verification is achieved by comparing results from CRAFTS with these built-in implementations for identical case settings. The CRAFTS implementations produced identical results to the built-in OpenFOAM implementations of these algorithms. An in-depth discussion about the flow model verification as well as a validation of earlier implemented flow solvers can be found in Appendix N.

4.3.5 Scalar transport verification

CRAFTS uses scalar transport to calculate the non-reacting motion of its concentration variables. CRAFTS' scalar transport algorithm can be isolated by defining a non-reacting case with steady state flow. Results from this setup should be exactly identical to OpenFOAM's built-in solver for scalar transport: `scalarTransportFoam`. Therefore, verification of CRAFTS' scalar transport algorithm is achieved by comparing the results of these two solvers. CRAFTS produces identical results to OpenFOAM's built-in implementation of scalar transport.

4.4 Indirect verification

The remaining components require more extensive verification methods. The expected output for these components is not always easy to determine. For instance, it is not straightforward to use ADM1 as a benchmark for ADM-MDA because they are vastly dissimilar numerical platforms, and therefore a difference will be expected between the two outputs. The objective in indirect verification is ascertaining that the observed difference is solely caused by the differences between numerical platforms, and not a symptom of an underlying problem with the model. Components that must be verified in this manner include:

- the bulk model;
- the biochemical reaction solver;
- the gas model; and
- the coupled reaction model.

4.4.1 Bulk model verification

The bulk model, developed in Excel using Visual Basic macros, is verified against the precompiled MATLAB implementation of ADM1 provided by Dr. Batstone, an author of the original ADM1 publication (Batstone et al., 2002). Identical test simulations are performed on both the Excel model and the MATLAB model. The initial conditions, boundary conditions, and coefficient settings that fully define these simulations are based on the sample case file provided with the ADM1 distribution. These values are presented in Appendix D. Simulation results are shown in Figure 27 below.

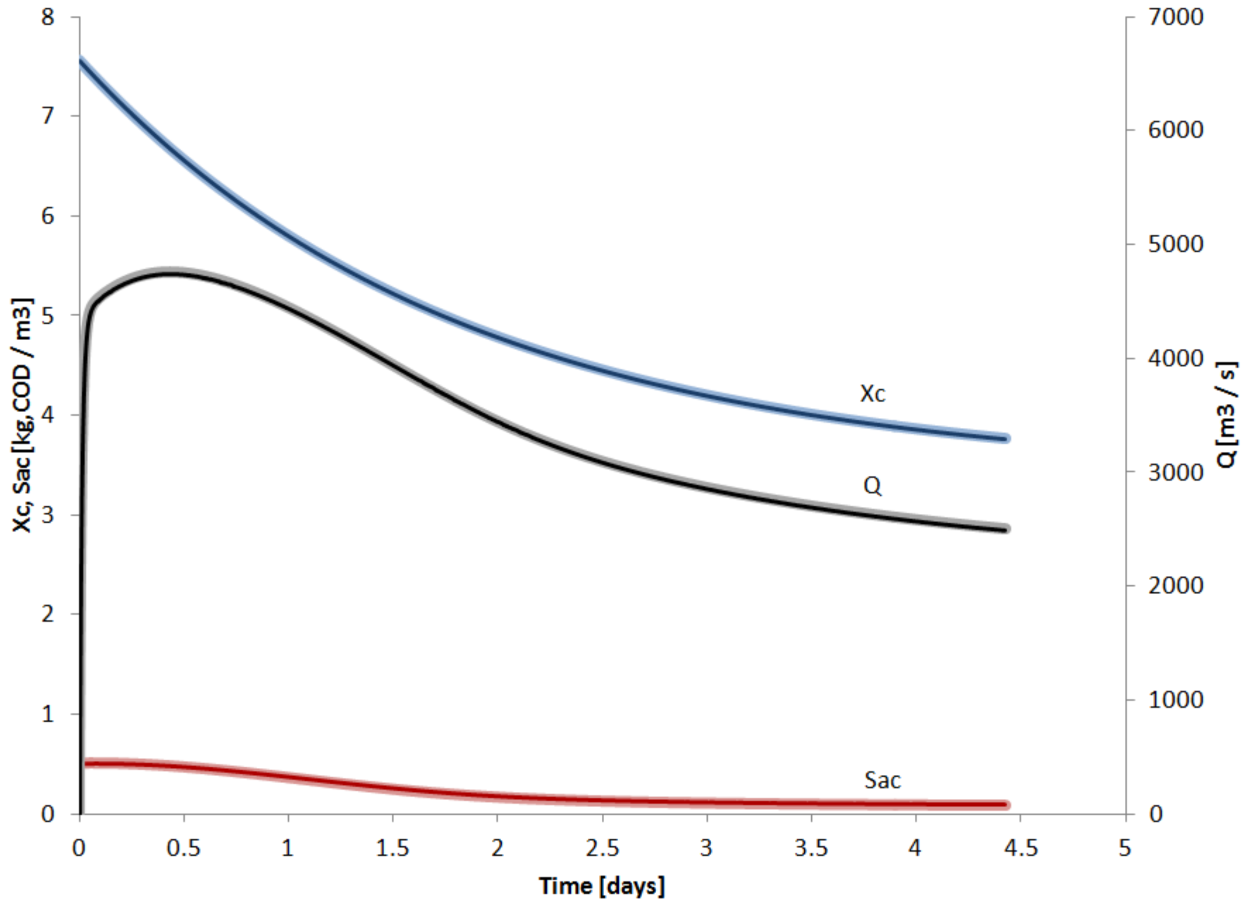


Figure 27: Verification of the Excel model

In this figure, there are three variables plotted. Each variable has two lines: the dark inner lines indicate MATLAB results, whereas the thick outer lines indicate Excel results. Since there is no discernible difference between these results, the lines appear on top of each other. The three variables shown above are:

- the concentration of particulate solids (X_c), representing fresh biomass that has yet to be digested;
- the concentration of solubilised acetate (S_{ac}), representing an intermediate stage in the digestion process; and
- the total gas flow (Q), representing the final output of the digestion process.

These three variables show no appreciable difference in the plot above, as is the case for all 36 model variables. The plot above shows only up to 4.5 days of simulation time; however, the verification was advanced to 1000 days, showing a maximum relative change of less than 5% between the two models. Compounded round-off error, or differences in ODE solvers may have contributed to this difference.

4.4.2 Biochemical reaction solver verification

The coupled reaction model simultaneously solves biochemical reactions and scalar transport. Scalar transport has already been verified; this section verifies the biochemical reactions. The bulk model provides a good standard against which the biochemical reaction solver can be compared; however, it is only a one-dimensional model. To verify against this, it is necessary to use case settings that remove spatial variation from the CRAFTS simulation. For this case, spatial variation is removed by reducing the mesh to a single control volume. The geometry for this case is shown in Figure 28.

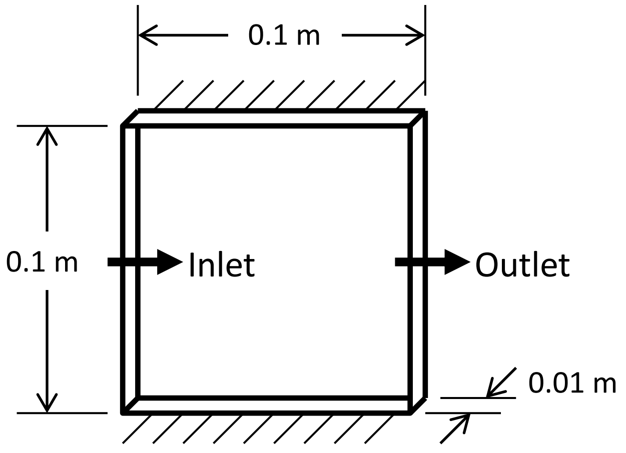


Figure 28: Simple cavity geometry

No flow solver is used in this simulation; rather, the flow is steady-state and uniform from left to right at 5.0×10^{-8} [m s^{-1}], corresponding to a hydraulic retention time of 23.1 days.

Diffusion introduces a problem that confounds the verification and is switched off by setting the diffusion coefficients of all concentration variables to zero. Diffusion is the tendency for a substance to spread out in a fluid. Since the simulation involves only one control volume, diffusion has no effect, with one exception: mass will transfer across any boundary with a non-zero concentration gradient. In this case, mass will be lost out the inlet. Typically, the inlet would have sufficient flow to overcome the diffusion velocity; however this case is different. ADM1 approximates periodic fluid injection in a digester by assuming a continuous, low velocity injection, and therefore the velocity is miniscule. To avoid this problem, diffusion is set to zero for verification purposes.

This simulation uses the pseudo gas model, which was created for verification purposes, as discussed in Section 3.6.9.6 on page 103. One hundred days of simulation time are recorded. The results show good agreement between the bulk model and CRAFTS for all variables. Figure 29 through Figure 31 present plots of X_c, S_{ac} and Q_{gas}, representative of various stages of anaerobic digestion. This verifies the function of the biochemical reaction solver.

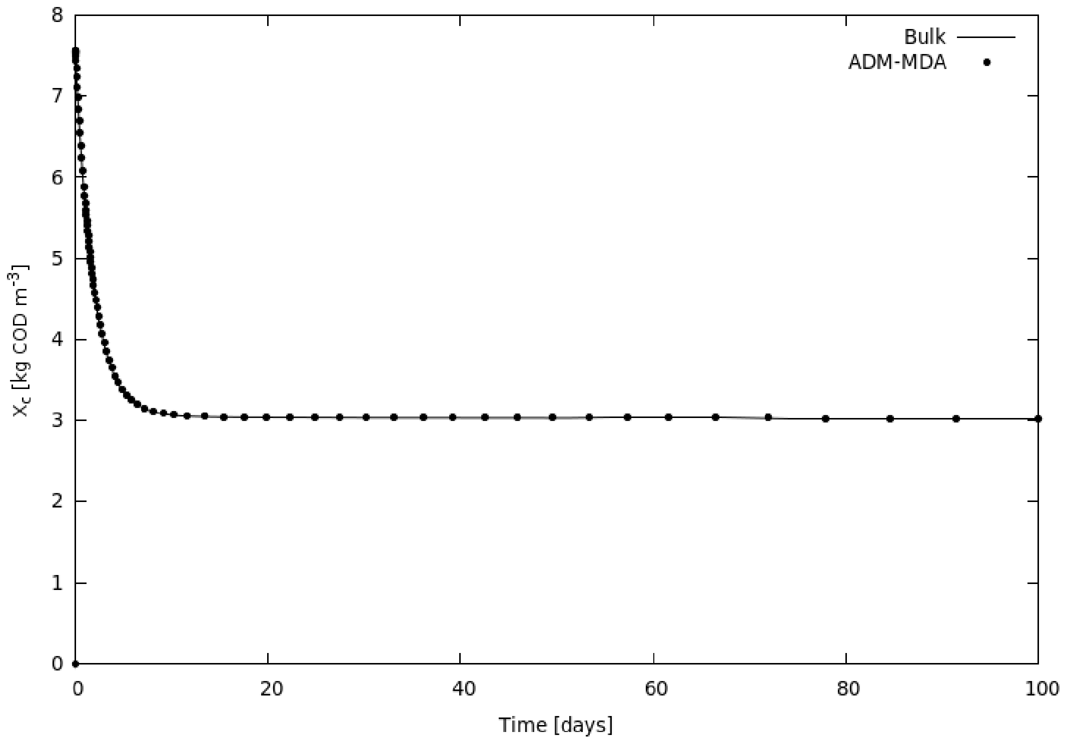


Figure 29: X_c verification, pseudo gas model with flow

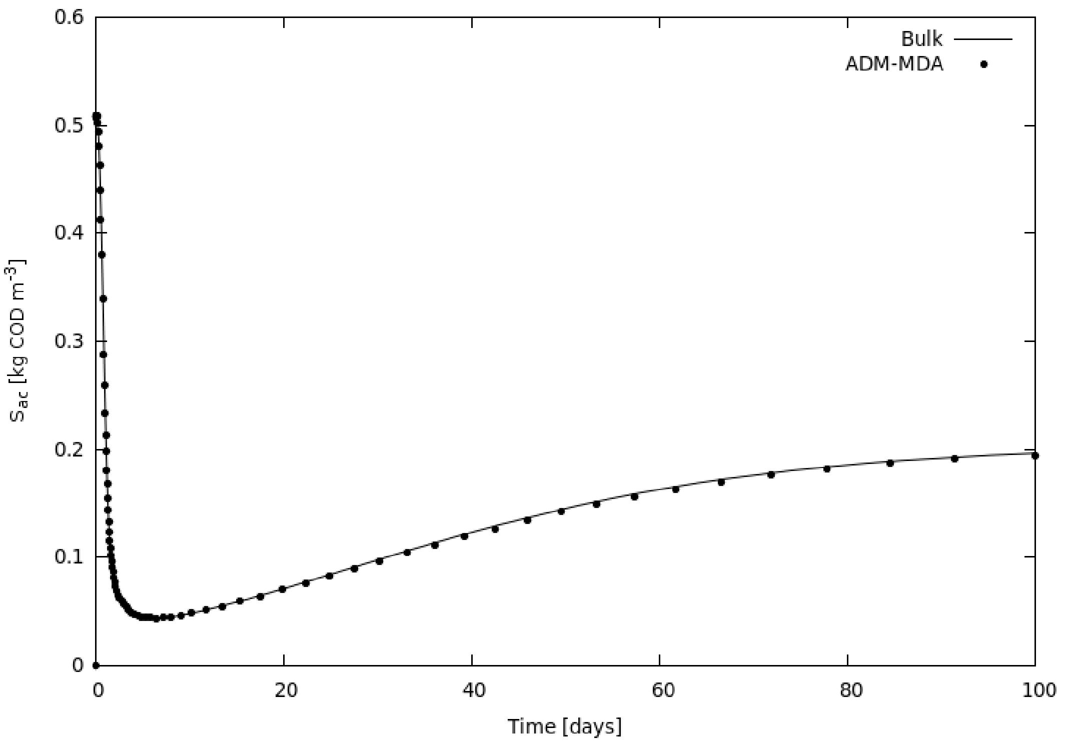


Figure 30: S_{ac} verification, pseudo gas model with flow

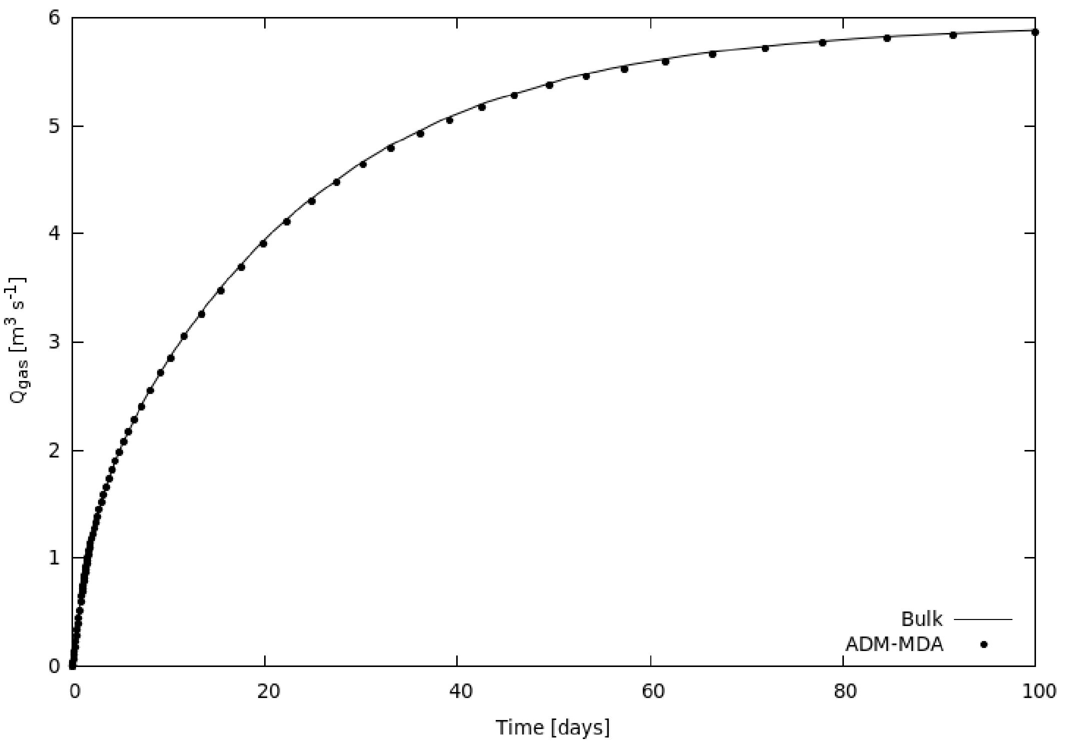


Figure 31: Q_{gas} verification, pseudo gas model with flow

4.4.3 Gas model verification

To verify the full gas model, diffusion has to be enabled for the concentration variables associated with each gas, S_{CH4}, S_{H2} and S_{IC}. As previously discussed, diffusion leads to mass loss through the inlet, which confounds the verification. To prevent this, the inlet is closed and fluid injection is disabled. This effectively simulates a starving digester, for verification purposes. Since this simulation also involves only one control volume, the value of the diffusion coefficients for these three variables have no effect; however, they must have a non-zero value. The geometry for this simulation is identical to the previous simulation, shown in Figure 28, but with the inlet and outlet set as walls. The initial conditions, boundary conditions, and coefficients that define the reaction physics are identical to the ADM1 case settings, presented in Appendix D. The results show good agreement with the bulk model. Figure 32 through Figure 34 present plots of variables representative of various stages of anaerobic digestion. The plots are shown with two different scales for detail. This verifies the function of the gas model.

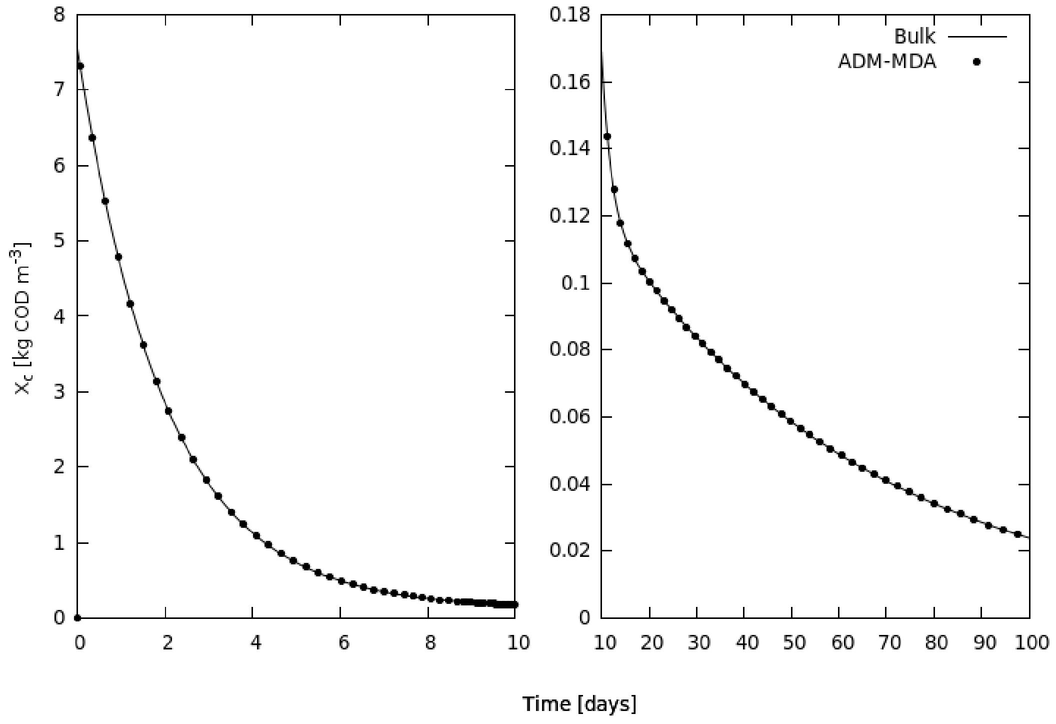


Figure 32: X_c verification, standard gas model without flow

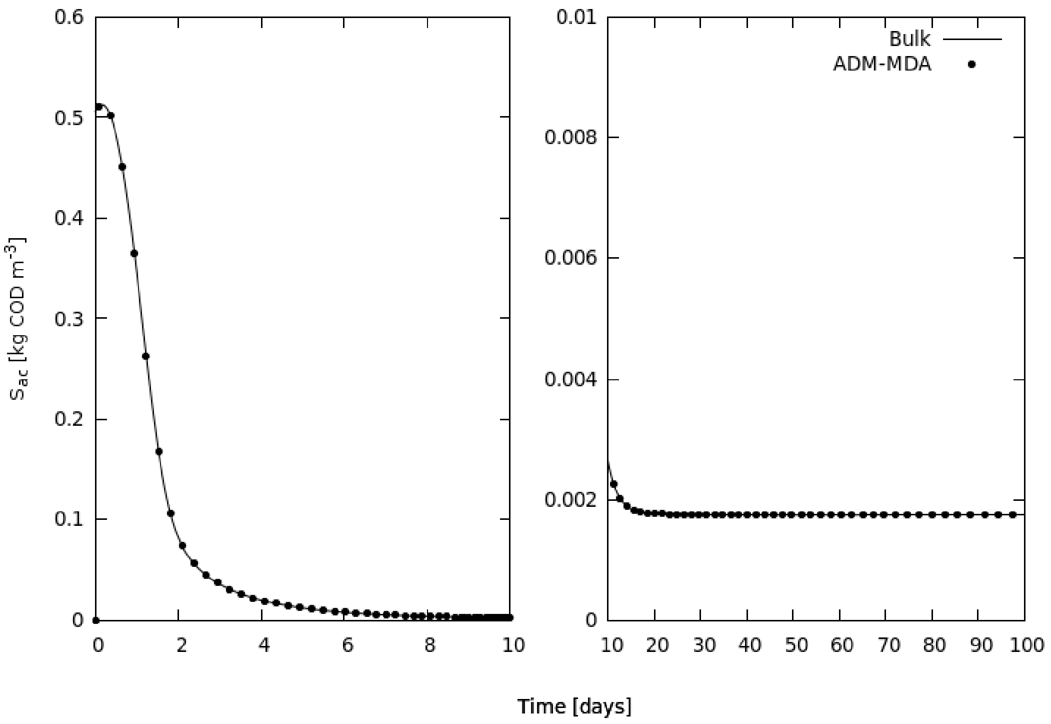


Figure 33: S_{ac} verification, standard gas model without flow

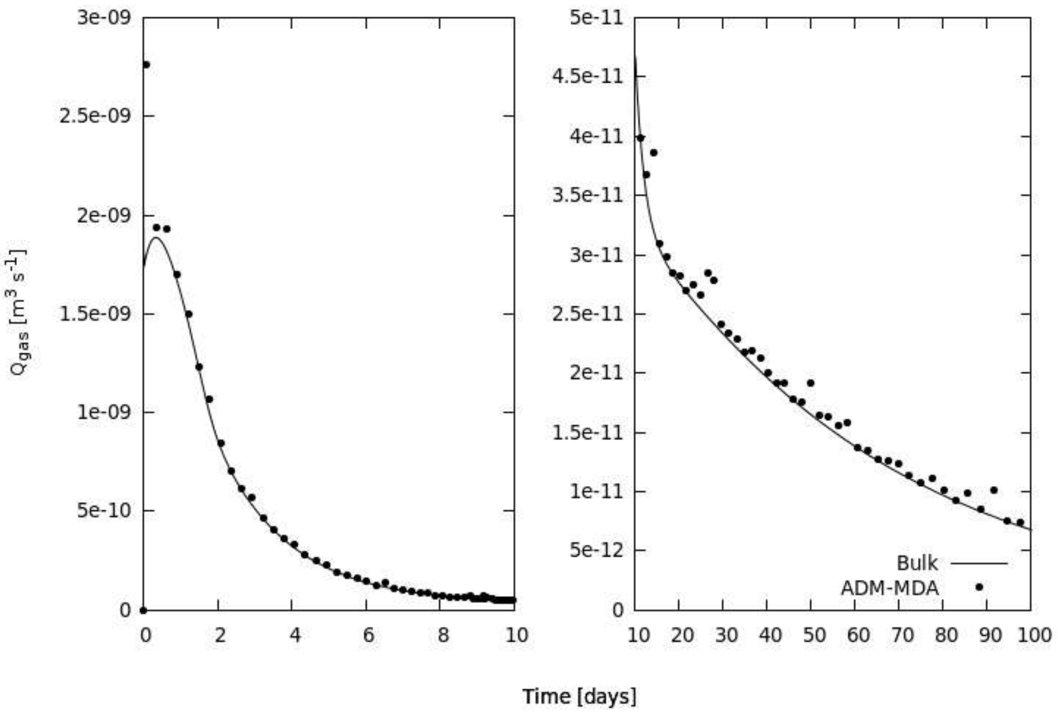


Figure 34: Q_{gas} verification, standard gas model without flow

4.4.4 Coupled reaction model verification

The main aspects of the coupled reaction model, namely the biochemical reactions and scalar transport, have been verified separately. What remains is a verification to confirm that the biochemical reactions and scalar transport function properly together. To accomplish this, the simulation uses a square cavity with a fixed swirling flow, and diffusion set to zero. The initial conditions are uniform throughout, all the boundary conditions are zero gradient, the pseudo gas model is used, and no fluid injection occurs. With these settings, all cells should produce identical results, and these results should correspond with the bulk model. Any spatial non-uniformities would suggest a problem with scalar transport, and any deviations from the bulk model would suggest a problem with the biochemical reactions. The geometry for this simulation is a 10 x 10 x 1 mesh, where each control volume has the same dimensions as the other verifications, Figure 28. Essentially this is one hundred of these smaller control volumes stacked together in a grid, as shown in Figure 35.

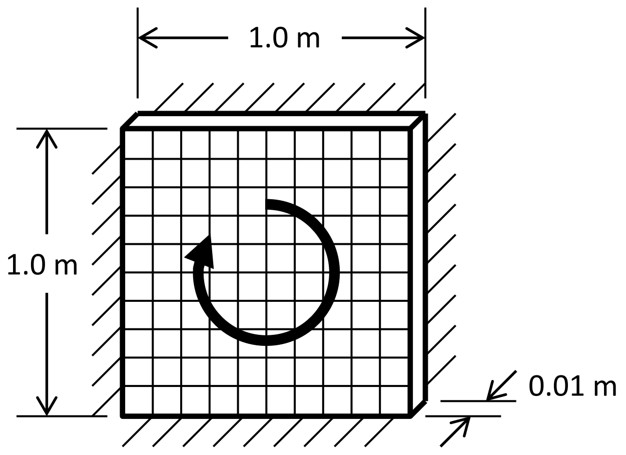


Figure 35: 10 x 10 cavity geometry

The flow in the cavity is a fixed, clockwise swirling flow, created using a modified version of OpenFOAM's built-in incompressible, steady-state solver, `simpleFoam`. The modification is the incorporation of a momentum source term, which drives the flow. The source is in the tangential direction, centred on the geometric midpoint of the cavity. The magnitude of the source term is parabolic, as shown in Figure 36.

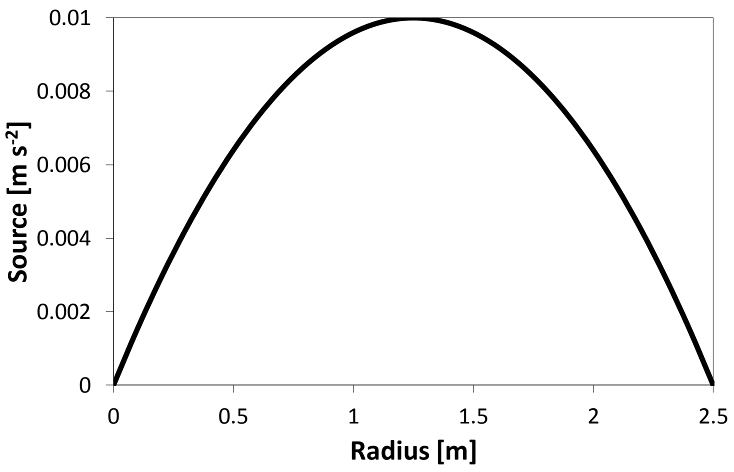


Figure 36: S_θ, mixing tangential source term, as a function of distance from vessel centre

After 100 days of simulation time, the maximum spatial deviation is 1.81%, and the volumetric average of the results show good agreement with the bulk model. Figure 37 through Figure 39 present plots of variables representative of various stages of anaerobic digestion. As before, the plots show two different scales for detail. This verifies the function of the coupled reaction model.

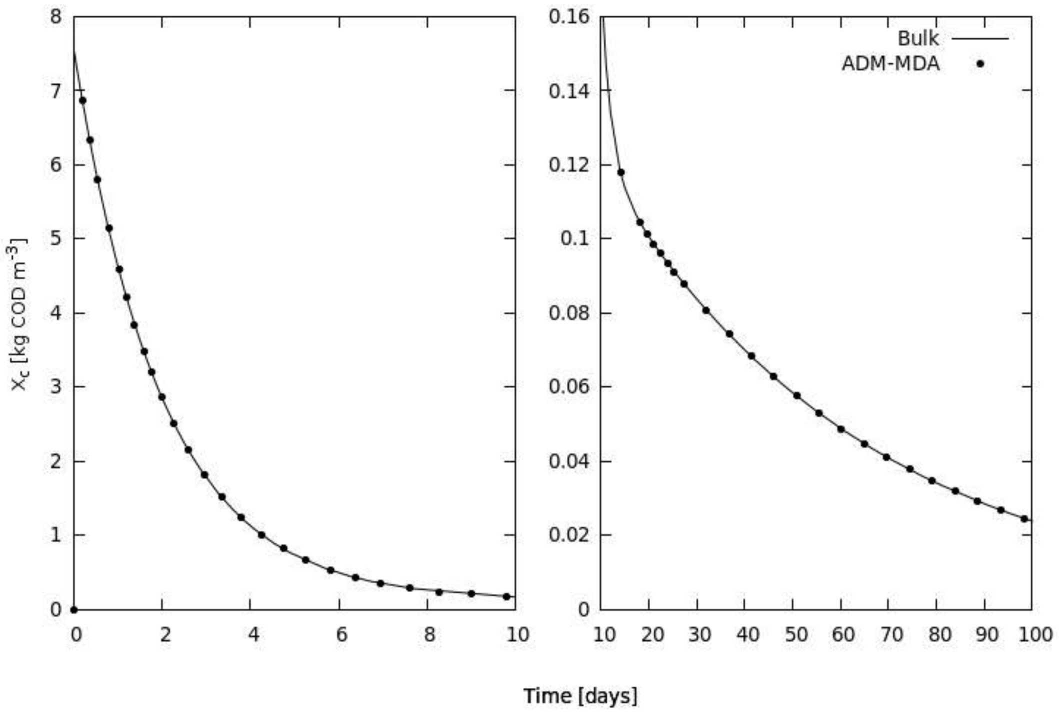


Figure 37: X_c verification, pseudo gas model, swirling cavity, no flow

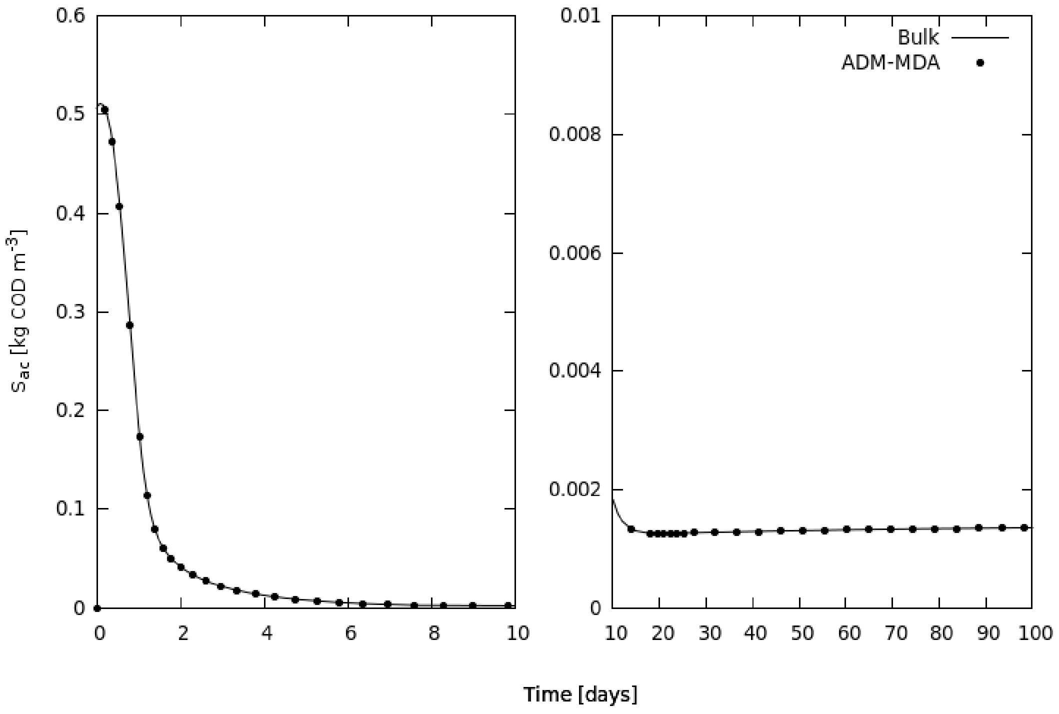


Figure 38: S_{ac} verification, pseudo gas model, swirling cavity, no flow

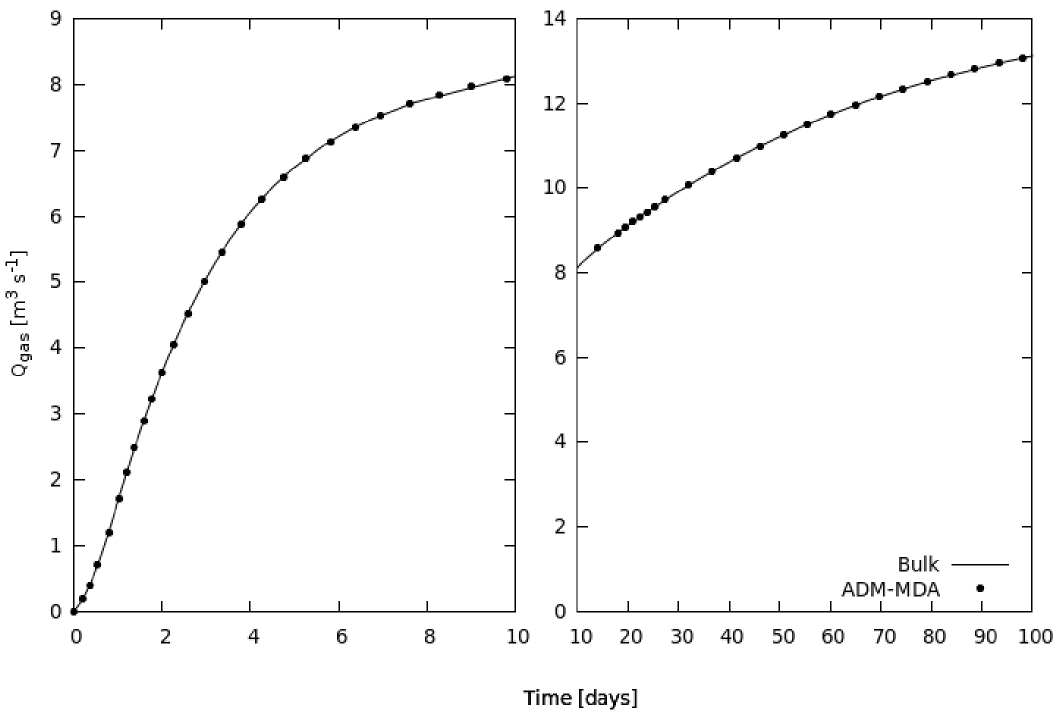


Figure 39: Q_{gas} verification, pseudo gas model, swirling cavity, no flow

4.5 Using the model

As with any numerical model, there are several aspects of CRAFTS that require careful attention. These involve numerical phenomena that, when improperly handled, will lead to invalid simulation results. They do not, however, indicate an error with the model. These are:

- the convergence criteria;
- the transport error; and
- variable limiters.

4.5.1 Convergence criteria

There are several convergence criteria in CRAFTS. These are:

- the matrix solver tolerances;
- the timestep doubling convergence criteria;

- the implicitAutoSolve convergence criteria; and
- the gas model convergence criteria.

4.5.1.1 Matrix solver tolerances

The matrix solver tolerances are the conventional convergence criteria for CFD. These are the RMS residuals of the current solution to the linearized equation set. These values can be adjusted in the `system/fvSolution` file. CRAFTS runs several matrix solvers, depending on the case setup. The CRAFTS simulations used the following tolerances:

- the flow model equations, e.g.:
    - pressure, `p`, and `pFinal`: 10^{-12} and 10^{-15}, respectively;
    - velocity, `U`, and `UFinal`: 10^{-12} and 10^{-15}, respectively;
    - turbulence kinetic energy, `k`: 10^{-12};
    - turbulence dissipation, `epsilon`: 10^{-12}; and
    - turbulence stress, `R`: 10^{-12};
- the scalar transport equations, one for every standard variable designed in the case setup: 10^{-15} for all scalars; and
- the coupled reaction matrix, `blockVar`: 10^{-12}.

These convergence criteria have no impact on the adaptive timestepping. When selecting values for these, the established body of literature regarding CFD convergence criteria is relevant. For the format of the `system/fvSolution` file, refer to the OpenFOAM documentation.

4.5.1.2 Timestep doubling convergence criteria

The error control for CRAFTS is founded on a timestep doubling method, as described in Section 3.6.5 on page 84. This method compares the results of a single coarse step with two fine steps. The differences are measured using RMS error, scaled by the average, for each standard and implicit variable. Unlike the matrix solver tolerances, these criteria are not related to an equation residual; they are based on the actual solution itself. Therefore, they will be up to several orders of magnitude larger than the matrix solver tolerances. The values can be adjusted in the `constant/admVariableDict` file. There are two extremities for the timestep doubling convergence criteria. If the criteria is too large (i.e. too loose), the errors in the system will destabilise the reaction solver, leading to erratic numerical behaviour. Conversely, if the criteria is too small (i.e. too tight), the automatically selected timestep will become too small. The criteria used in the CRAFTS simulations presented below are 10^{-4}.

4.5.1.3 implicitAutoSolve convergence criteria

CRAFTS has a function that automatically solves a variable by finding its equilibrium point, assuming all other variables are static. This is known as the *implicitAutoSolve*, as described in Section 3.6.8 on page 95. It is important that the convergence criteria for the implicitAutoSolve function are smaller (i.e. tighter) than the timestep doubling convergence criteria for the associated variables. The implicitAutoSolve convergence criteria can be set in the `constant/admVariableDict` using the `autoSolve` keywords. The criteria used in the CRAFTS simulations presented below are 10^{-4}.

4.5.1.4 Gas model convergence criteria

The gas model has two convergence tests: mass conservation, and stabilization. Each gas specie has a convergence criterion for both of these. Like the *implicitAutoSolve* function, it is important that the gas model convergence criteria are smaller (i.e. tighter) than the timestep doubling convergence criteria for the associated variables. The gas model convergence criteria can be adjusted in the `constant/craftsGasModelDict` file. The criteria used in the CRAFTS simulations presented below are 10^{-6} for both the mass conservation and stabilization tests.

4.5.2 Transport error

As with most other numerical methods, there are errors inherent in the numerical solution of scalar transport. If the simulations are not setup with careful attention, a transport error will dominate the solution. Furthermore, due to the fact that anaerobic digestion models have long simulation times, they are particularly sensitive to this problem. Therefore, this section demonstrates the effect of the transport error and how to avoid it, for the benefit of future model users. To demonstrate the effect, the simulation that verified the coupled reaction model, detailed in Section 4.4.4 on page 133, is repeated, but with one important difference: the velocity field is changed. Originally, the velocity field was obtained using a matrix solver tolerance of 10^{-12}, giving a high-quality mass-conserving velocity field. For this demonstration, the velocity field is obtained using a matrix solver tolerance of 10^{-4}. In theory, the results should be spatially uniform; however, initial simulations exhibited random spatial fluctuations in all variables, including the non-reacting species, S_{an} and S_{cat}. Figure 40 shows the uneven spatial distribution of S_{an} after 34,882 seconds.

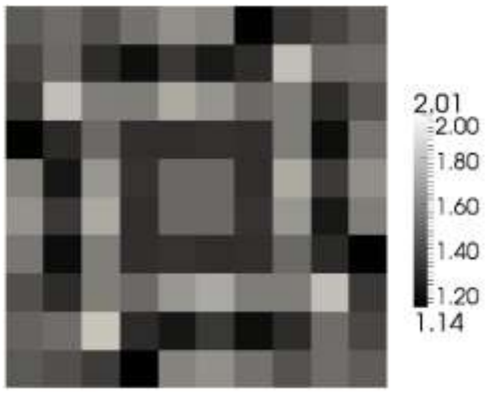


Figure 40: S_{an} spatial distribution caused by the transport error

Since the fluctuations existed in non-reacting species, the problem lies in scalar transport, and since scalar transport of CRAFTS has been verified, the underlying cause is due to errors inherent in CFD, and is not caused by any problems in CRAFTS. Several simulations were performed to establish the nature of the transport error. Case settings that affect the transport error are:

- **total simulation time** – the error propagates approximately linearly with time: a simulation of 1000 seconds produced a maximum error 987 times greater than a simulation of 1 second;
- **mesh density** – denser meshes producing greater errors: a 10 x 10 mesh produced a transport error of ±7.1x10^{-7} [% s^{-1}], a 40 x 40 mesh produced a transport error of ±2.4x10^{-4} [% s^{-1}], and a 160 x 160 mesh produced a transport error of ±1.7x10^{-2} [% s^{-1}]; and
- **quality of the velocity field** – the quality of the velocity field has a significant effect on the transport error: a velocity field created with a matrix solver tolerance of 10^{-4} produced a transport error of ±2.5x10^{-3} [% s^{-1}], whereas a tolerance of 10^{-9} produced a transport error of ±7.1x10^{-7} [% s^{-1}]; however, there is a limit to the accuracy gains obtained in this manner, as increasing the convergence criteria further did not show any significant improvement.

Other key factors likely affect the transport error, including the magnitude of the velocity and the discretization schemes. Case settings that have little or no effect on the transport error are:

- **volumetric average** – the volumetric average of the transported quantity did not show any deviation from the expected value;
- **timestep size** – this has negligible impact on the error: a single 1000 second step produced a nearly identical error to 100,000 steps of 0.01 seconds; and
- **scalar transport matrix solver tolerance** – the scalar transport matrix solver tolerance has negligible impact: a matrix solver tolerance of 10^{-6} produced an identical error as a matrix solver tolerance of 10^{-9}.

When designing a simulation, it is a good idea to test for a transport error. This can be achieved by performing a simulation that should produce a spatially uniform result, and checking for spatial differences. For the coupled reaction model verification simulation detailed in Section 4.4.4 on page 133, case settings were used to reduce the transport error to an estimated 7.1x10^{-7} [% s^{-1}], which would produce a total error of approximately 5%; however the final result was 1.81%.

4.5.3 Variable limiters

Most of the variables in ADM1 are concentration values. Negative concentrations do not make sense physically, and numerically they cause divergence. Therefore, it is desirable to restrict the variables; however, variable restrictions violate mass conservation. For instance, when the model sets a negative value to zero, it creates mass.

Finite volume scalar transport of a positive field can still lead to negative values, particularly when it uses simple discretization schemes in the presence of sharp gradients. In digester simulations, a fluid injection event can lead to steep concentration gradients. To assess the magnitude of mass creation in CRAFTS, a digester simulation is performed using the case settings detailed later, in Section 5.2 on page 147. Figure 41 shows the volumetric average of the concentration of solubilised anions, S_{an}, a non-reacting species. The first 400 seconds are shown, capturing a 120 second fluid injection event, and 280 seconds of additional mixing. The plot shows a gradual increase in mass over time.

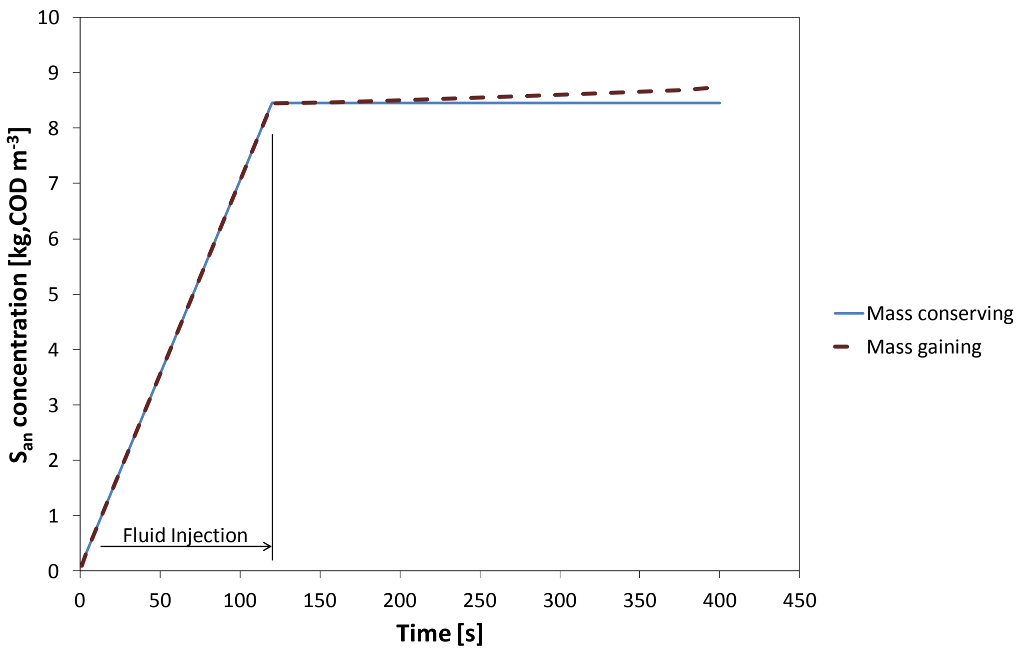


Figure 41: Mass gain in a non-reacting species

To correct for this, a number of mass conserving strategies were attempted. A simple summation gives the total amount of mass created or destroyed when the model applies the variable limits; the question is what to do with it. One strategy is to remove or add the difference evenly across all unlimited control volumes. This has the effect of removing fine detail resolution from the model. For instance, the effects of diffusion cause the concentration to spread out throughout the geometry, with many control volumes acquiring trace amounts of the species. Removing excess mass in this manner causes most of these control volumes to return to zero, thus eliminating the effects of diffusion. An alternative strategy, and the one that was selected, is to remove the mass proportionately based on the quantity of the species within each control volume. The mass conserving plot in Figure 41 is obtained from this method.

This is a questionable method. The preferred method is to never require variable limits, or that the mass created or destroyed is negligible. This may be achieved with improved discretization schemes, or higher density meshes. Therefore, the framework of CRAFTS was modified such that the variable limit algorithm can be designated by the user in a UDF. Should one not be designated, the default is the blind min / max algorithm that does not conserve mass. The CRAFTS implementation of ADM-MDA uses proportional mass corrections.

Chapter 5 Results and discussion

The practicality of the CRAFTS implementation of ADM-MDA is limited by the numerical stiffness of the biochemistry. However, an ADM-MDA simulation is performed within the constraints of CRAFTS to compare the performance of ADM1 with the spatially discretized version, ADM-MDA.

5.1 Numerical stiffness

The numerical stiffness of ADM1 is well-established and early verification attempts for the coupled reaction model show it is present in ADM-MDA. The more numerically stiff the model is, the greater the accuracy must be. However, increased accuracy requires smaller timesteps. Therefore, numerical stiffness can lead to two possible failure modes: divergence, caused by inadequate accuracy; or a timestep underrun, caused by an accuracy set too high, which necessitates a timestep size that is too small to detect. For a successful simulation, a balance between these two extremes must be found. Furthermore, if the problem is too numerically stiff, such a balance will not exist: the required accuracy will either lead to a vanishing timestep size, or will even exceed the capabilities of double precision. This is portrayed qualitative in Figure 42.

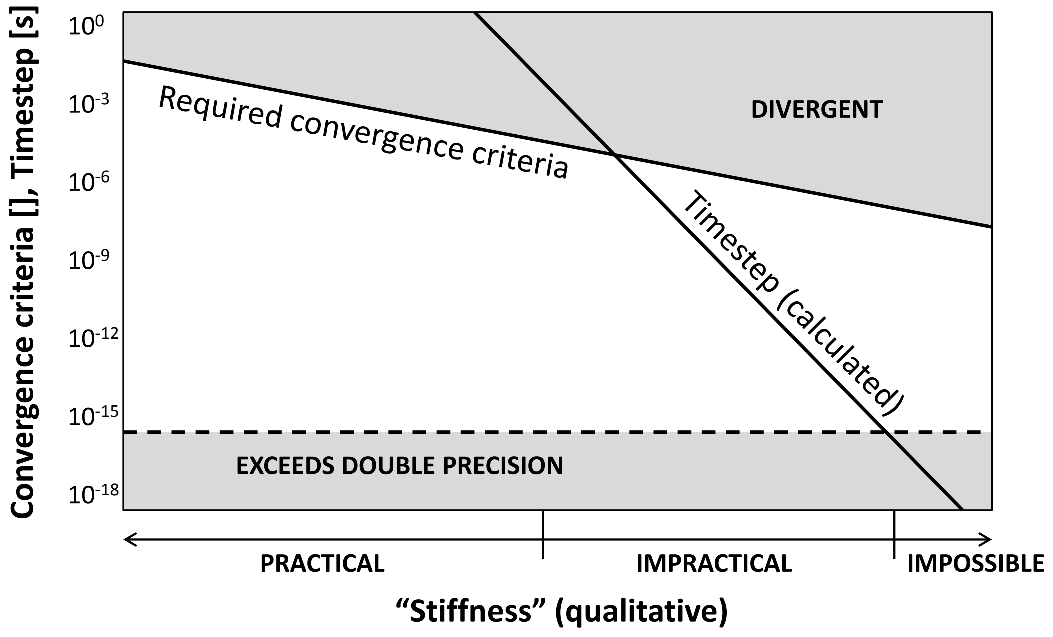


Figure 42: A qualitative portrayal of the effect of stiffness on convergence criteria and timestep size

In practice, it is very easy to specify an ADM-MDA problem that cannot be solved with the current CRAFTS framework. In terms of stiffness, even under ideal conditions, an ADM-MDA simulation is already close to the practical limits of CRAFTS. These conditions require timestep doubling convergence criteria of 10^{-3} or less, which gives an average timestep size of approximately 0.1 seconds. Conditions that increase stiffness, potentially pushing the problem beyond the capabilities of CRAFTS include:

- increasing the mesh density;
- high velocity gradients;
- high concentration gradients; and
- "upset" biochemistry.

The last one refers to the fact that the biochemical model itself has a wide range of numerical stiffness, which depends on the values of each variable, and therefore, it is possible for the reactions themselves to cause numerical failure.

A considerable effort went into reducing the model stiffness. Several components of the ADM-MDA implementation in CRAFTS were modified. These include:

- the gas model;
- the buoyancy; and
- the temperature inhibition.

5.1.1 Gas model

The gas model proves to be very volatile. To stabilise it, the near-zero behaviour of the gas flow has to be modified with a logistic function and a pressure offset, Section 3.6.9.3 on page 100. Furthermore, the gas model can exhibit exponential performance loss with increasing timestep size, which is addressed by implementing performance feedback, Section 3.6.5 on page 84. The bubble model source term was found to destabilise the gas model ODE, and it was necessary to switch it off. Consequently, the dissolved gas concentrations can increase above the saturation point without creating bubbles. Despite these modifications, the volatility in the gas model remains, as is evident in Figure 34. Section 8.2.1 on page 179 provides stiffness improvement suggestions.

5.1.2 Buoyancy

During verification simulations, the flow solver occasionally would exhibit divergence problems, caused by the pressure solution. Furthermore, with the case geometry and settings in use, the velocity solution would show recirculation at the outlet when buoyancy is enabled. It is likely that one of the reasons for the buoyancy problems is the fact that OpenFOAM's implementation of Boussinesq buoyancy does not allow for the pressure field to be fixed to an absolute value. PISO and PIMPLE solvers work with pressure differences, not the absolute pressure, therefore, it is necessary to constrain the absolute pressure at a boundary condition, or at a control volume to keep it from drifting too far. However, OpenFOAM's implementation of Boussinesq buoyancy does not permit fixing the pressure in a control volume. That is, it is missing:

```
        pEqn.setReference(pRefCell, pRefValue);
```

from the code for the pressure equation. Furthermore, the absolute pressure at a boundary condition cannot be determined due to the presence of a hydrostatic pressure gradient.

It was eventually decided to remove buoyancy from the model. In fact, it was insufficient to set gravity to zero: all traces of the buoyancy implementation had to be deleted. Buoyancy is an important force in an anaerobic digester, as there exists natural convective mixing. Its removal will prevent CRAFTS from capturing buoyant fluid motion. However, ADM1 does not capture any fluid motion, let alone buoyancy.

5.1.3 Temperature Inhibition

ADM1 includes temperature inhibition through temperature dependent coefficients. These are:

- the acid/base kinetic parameters for carbon dioxide, inorganic nitrogen, and water;
- Henry's coefficients for methane, carbon dioxide, and hydrogen; and
- the gas partial pressure of water.

When a low temperature inlet condition is used, the biochemical reactions become too stiff to solve. In other words, temperature inhibition leads to the "upset" biochemistry mentioned in Section 5.1 on page 143. Therefore, the case settings have to be changed to remove temperature variations from the problem. Temperature variation is also an important component of anaerobic digestion simulation; however, this problem is inherent in the biochemical model of ADM1. In fact, the MATLAB implementation of ADM1 does not even have a temperature variable: all of its temperature-dependant coefficients are hard-coded. Suggestions for future work to address the stiffness in the biochemistry model are detailed in Section 8.2.1 on page 179.

5.2 Full model simulation

This research is based on the hypothesis that adding spatial discretization to ADM1 will result in an improved anaerobic digestion model. This section performs the first test of that hypothesis by comparing ADM1 with CRAFTS' implementation of ADM-MDA for identical conditions. The problem is designed within the practicality constraints of CRAFTS.

5.2.1 Case setup

The case setup is a square two-dimensional vessel, shown in Figure 43, subdivided into a 10 x 10 orthogonal mesh. Although the CRAFTS solver is inherently a three-dimensional solver, a two-dimensional simulation can be performed by giving the *z*-direction a single mesh cell depth with a fractional distance, and giving the top and bottom surfaces "empty" boundary conditions, which disables their calculation.

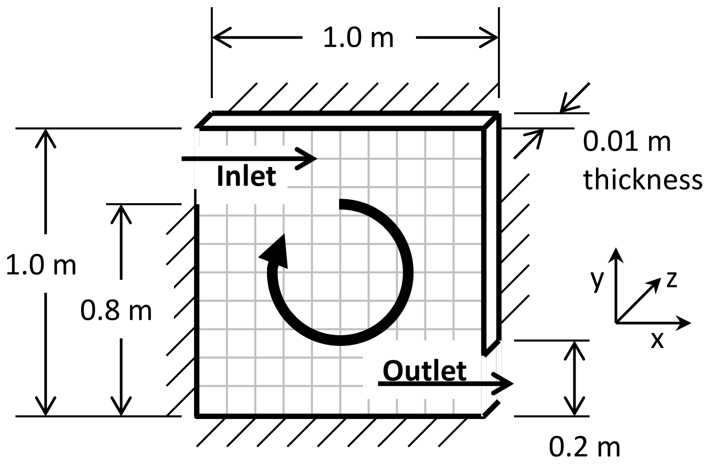


Figure 43: Case setup

Controllable elements include:

- fluid flow, on or off; and
- mixing, on or off.

Fluid injection events last two minutes at a time and occur on a daily basis. Fluid is injected at a velocity of 0.05 [m s^{-1}], giving the digester a hydraulic retention time of 8.33 days. When fluid is not being injected, the inlet and outlet surfaces become walls. This periodic fluid injection has to be manually duplicated in the bulk model, as it does not have a PLC emulator. The vessel undergoes a mixing operation for 10 minutes every hour. This is achieved with a parabolic source term that acts in the clockwise direction, presented earlier in Figure 36 on page 134. Diffusion is enabled, and each standard variable is given its own diffusion coefficient, shown in Table 5.

Table 5: Standard variable diffusion coefficients

i	Symbol	Γ (m^2 s^{-1})
0	S_{su}	6.60E-09
1	S_{aa}	8.00E-10
2	S_{fa}	8.00E-10
3	S_{va}	8.00E-10
4	S_{bu}	1.06E-09
5	S_{pro}	8.00E-10
6	S_{ac}	1.089E-09
7	S_{h2}	5.7555E-09
8	S_{ch4}	1.90571E-09
9	S_{ic}	2.45568E-09
10	S_{in}	1.88E-09
11	S_{i}	8.00E-10
12	X_{c}	5.00E-11
13	X_{ch}	5.00E-11
14	X_{pr}	5.00E-11
15	X_{li}	5.00E-11
16	X_{su}	5.00E-11
17	X_{aa}	5.00E-11
18	X_{fa}	5.00E-11
19	X_{c4}	5.00E-11
20	X_{pro}	5.00E-11

21	X_{ac}	5.00E-11
22	X_{h2}	5.00E-11
23	X_i	5.00E-11
24	S_{cat}	8.00E-10
25	S_{an}	8.00E-10

It is unnecessary to assign a diffusion coefficient to implicit or derived variables, as they are not subject to scalar transport in CRAFTS algorithms. The values for standard variables were determined qualitatively based on a preliminary literature search. The objective with diffusion is to ensure the effect is present and at the correct order of magnitude: diffusion will have negligible effect on the comparison between ADM1 and ADM-MDA; its inclusion is to ensure it is functional.

5.2.2 Runtime information

The simulation was stopped after 3.87 days of simulation time. This required 2.00 days of wall clock time to perform. The machine uses an AMD Phenom^{TM} 8550 Triple Core Processor with a clock speed of 2.2 GHz and has 4.0 GB of RAM. The simulation ran on a single core.

5.2.3 Time-resolved data

This section presents time resolved plots of all the variables in ADM-MDA compared against the bulk model. Variables from the liquid volume are volumetric averages. Plots are sorted approximately by stage of anaerobic digestion, or by physics phenomenon. Figure 44 is the legend, common to all plots.

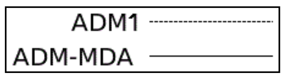


Figure 44: Legend for plots in the section

5.2.3.1 Disintegration stage

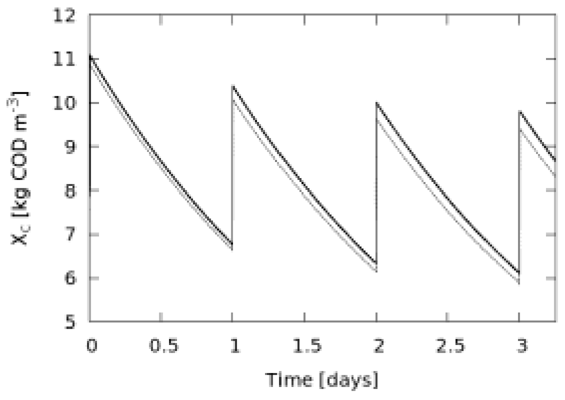


Figure 45: Composites

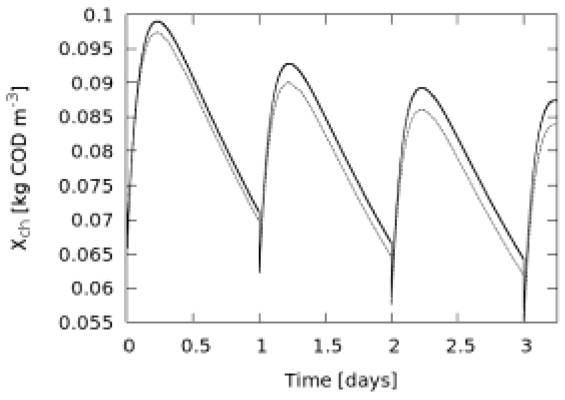


Figure 46: Carbohydrates

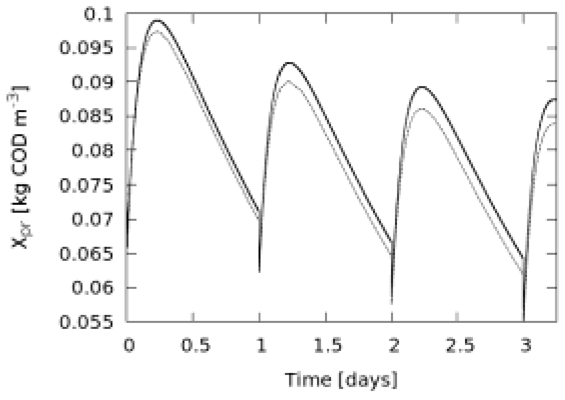


Figure 47: Proteins

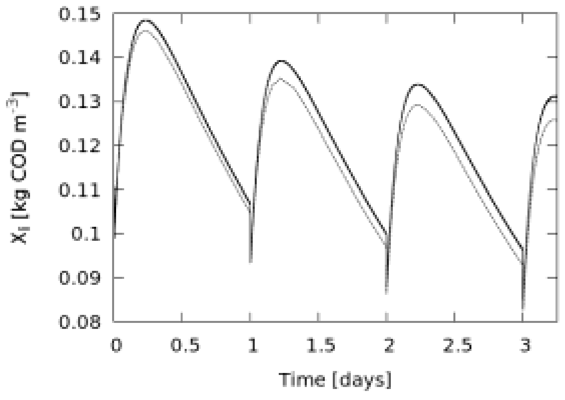


Figure 48: Lipids

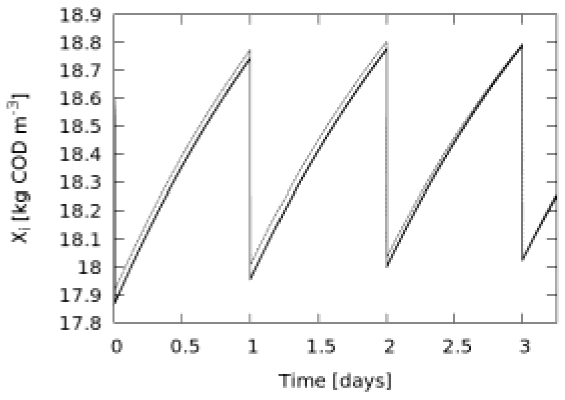


Figure 49: Particulate inerts

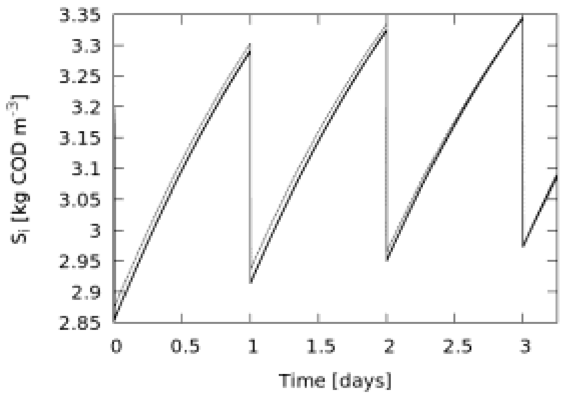


Figure 50: Soluble inerts

5.2.3.2 Hydrolysis stage

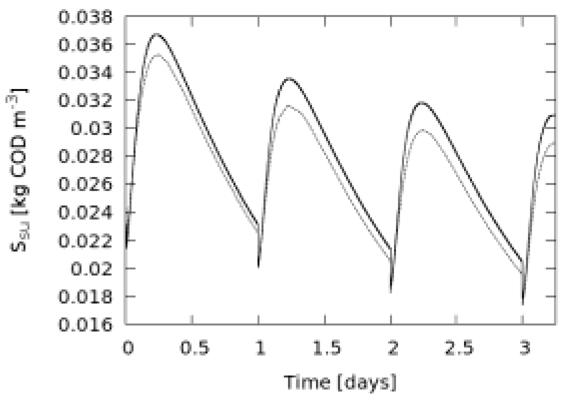


Figure 51: Monosaccharides

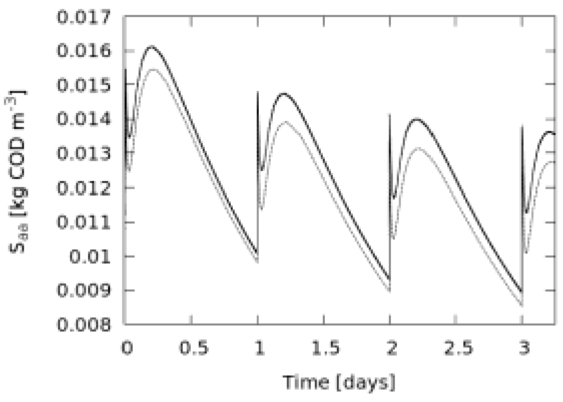


Figure 52: Amino acids

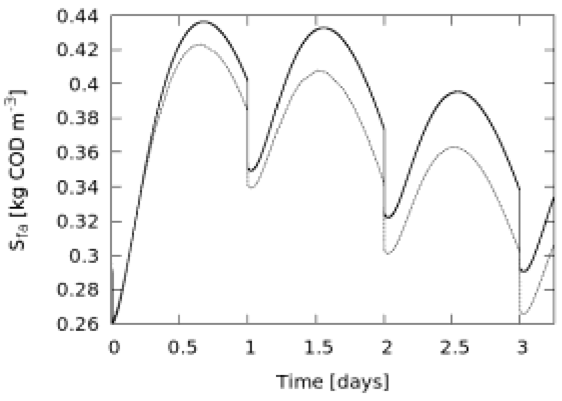


Figure 53: Long chain fatty acids

5.2.3.3 Acidogenesis, acetogenesis and methanogenesis

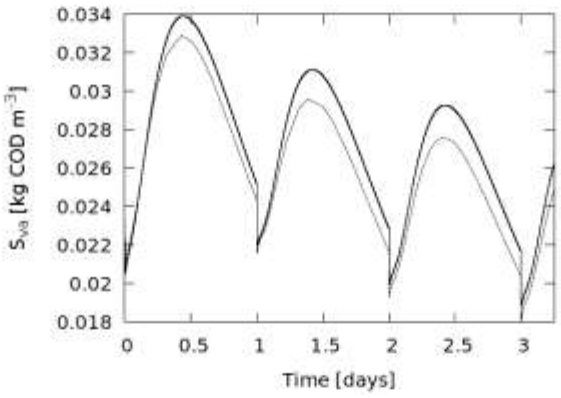


Figure 54: Total valerate

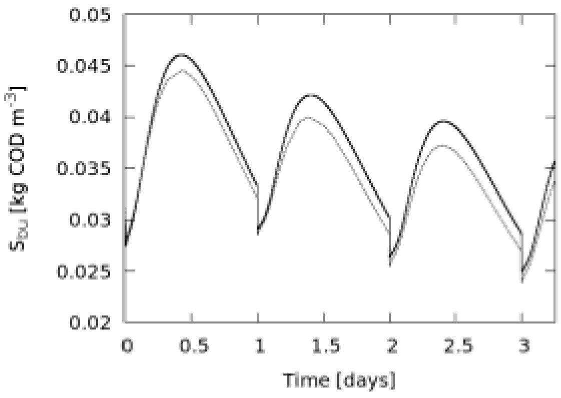


Figure 55: Total butyrate

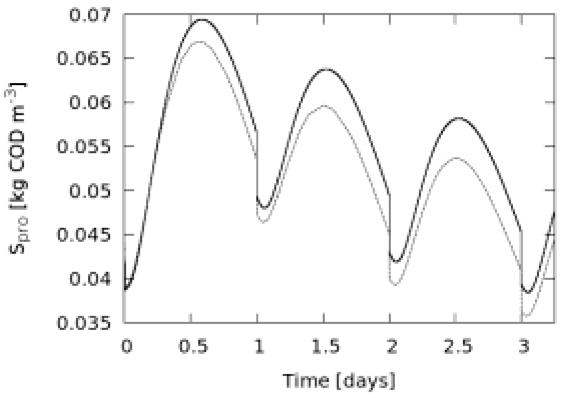


Figure 56: Total propionate

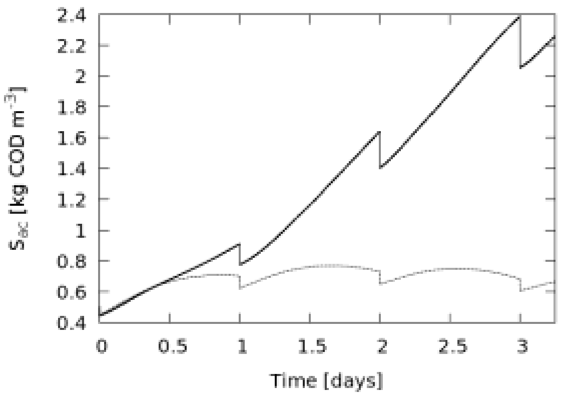


Figure 57: Total acetate

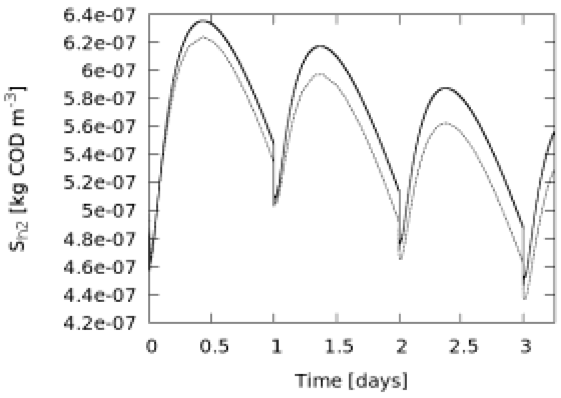


Figure 58: Dissolved hydrogen gas

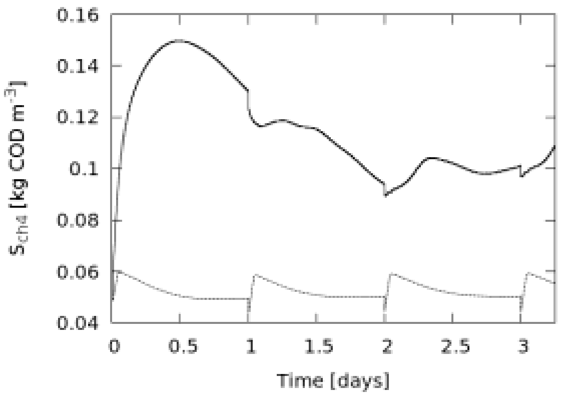


Figure 59: Dissolved methane gas

5.2.3.4 Carbon and nitrogen balances

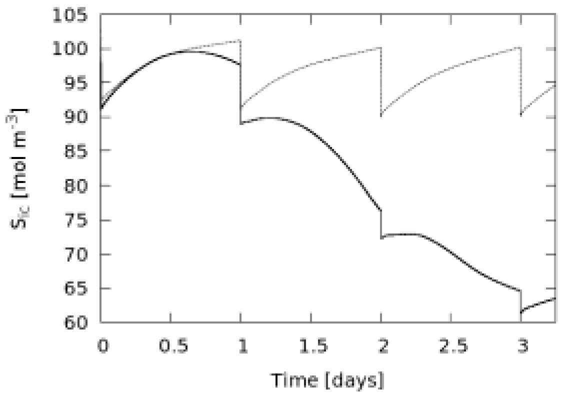


Figure 60: Inorganic carbon

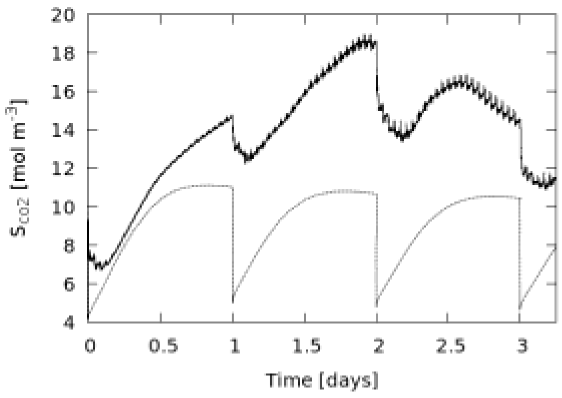


Figure 61: Dissolved carbon dioxide gas

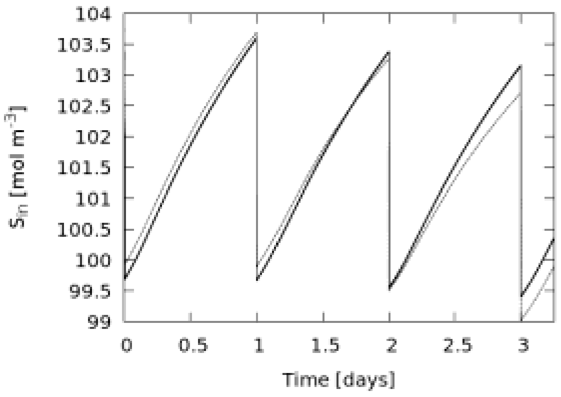


Figure 62: Inorganic nitrogen

5.2.3.5 Ion model

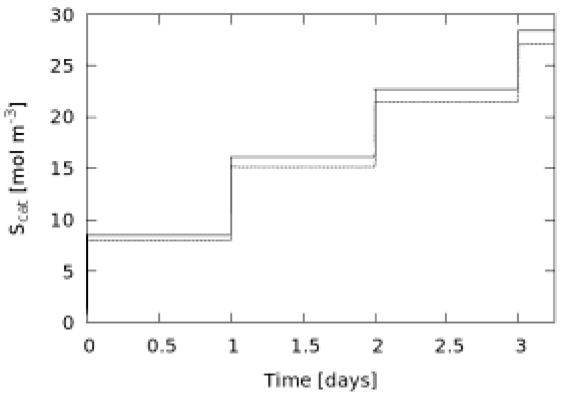


Figure 63: Cations

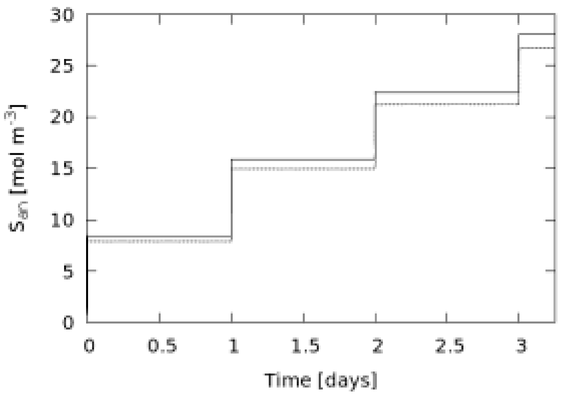


Figure 64: Anions

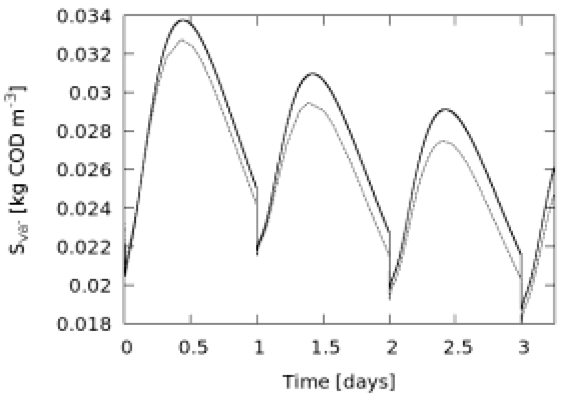


Figure 65: Valerate

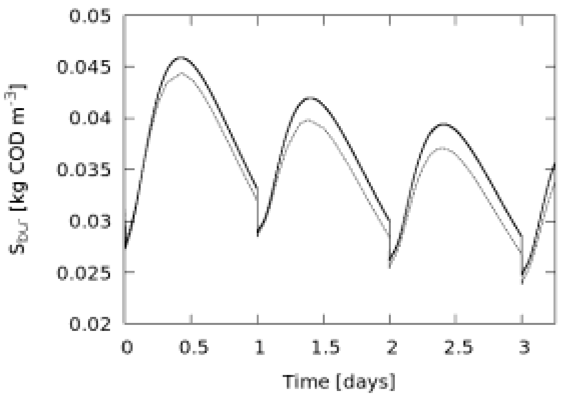


Figure 66: Butyrate

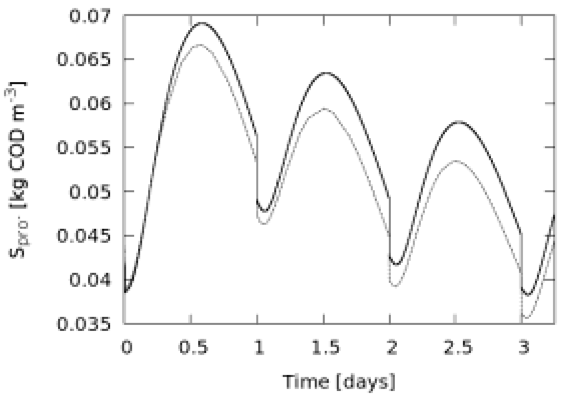


Figure 67: Propionate

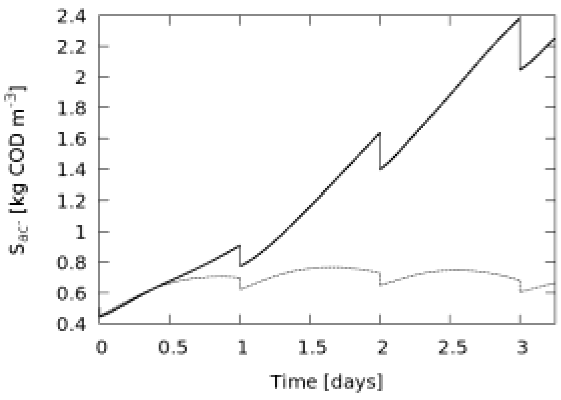


Figure 68: Acetate

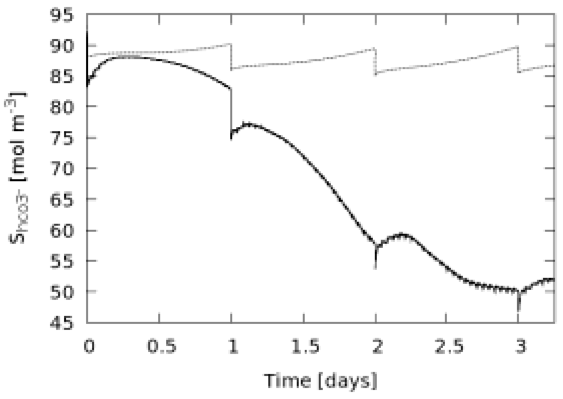


Figure 69: Bicarbonate

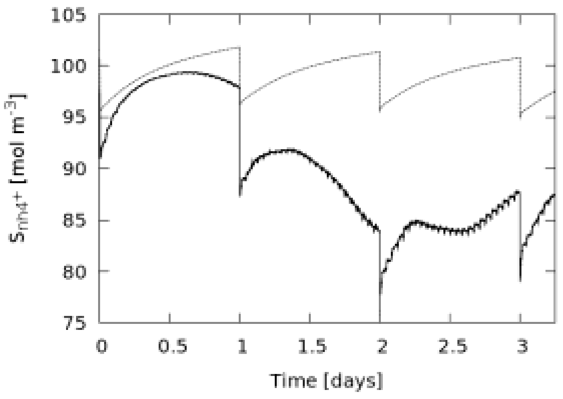


Figure 70: Ammonium

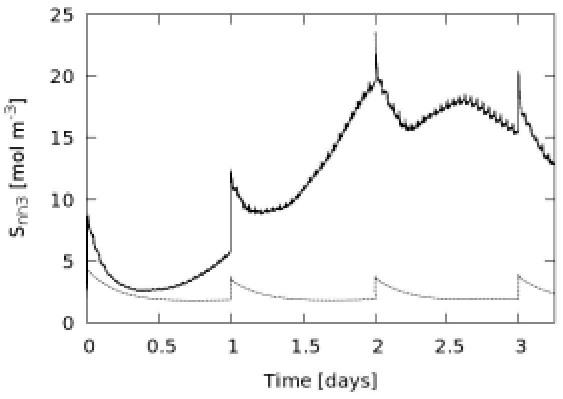


Figure 71: Ammonia

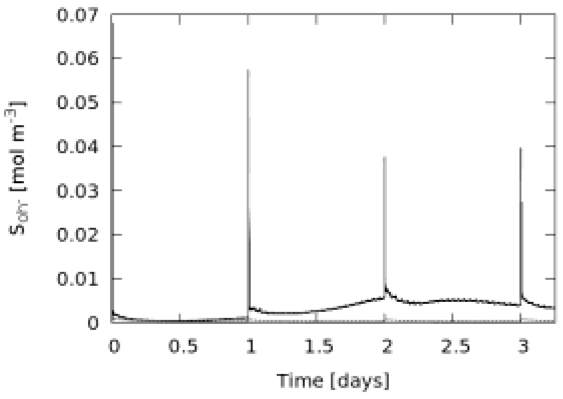


Figure 72: Hydroxide

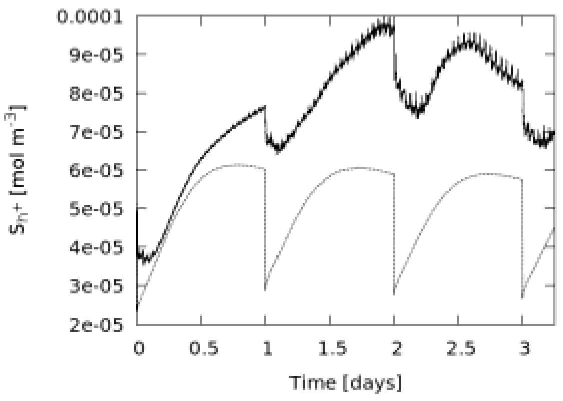


Figure 73: Hydrogen ions

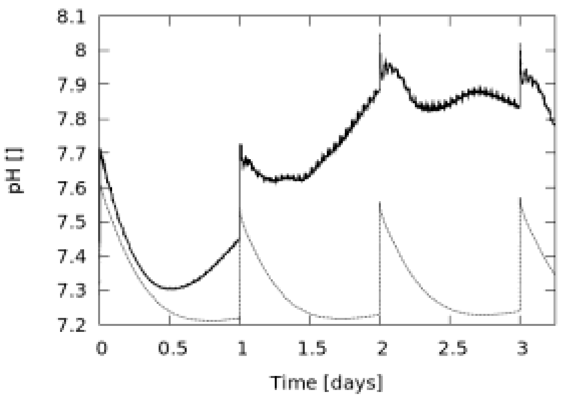


Figure 74: pH

5.2.3.6 Gas model

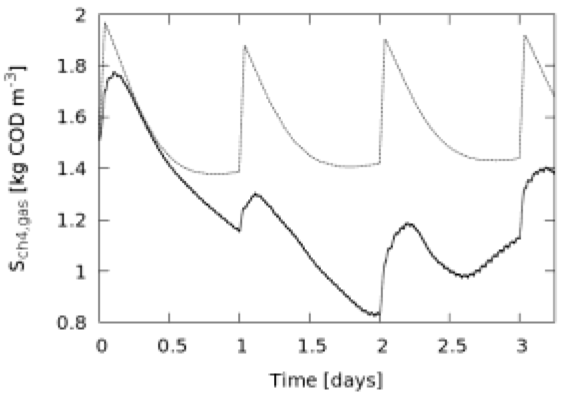


Figure 75: Methane concentration, gas volume

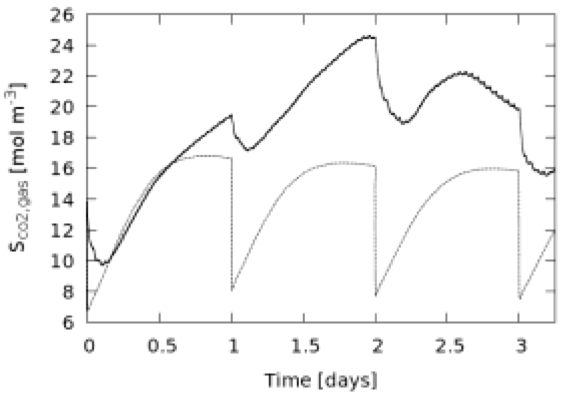


Figure 76: CO_2 concentration, gas volume

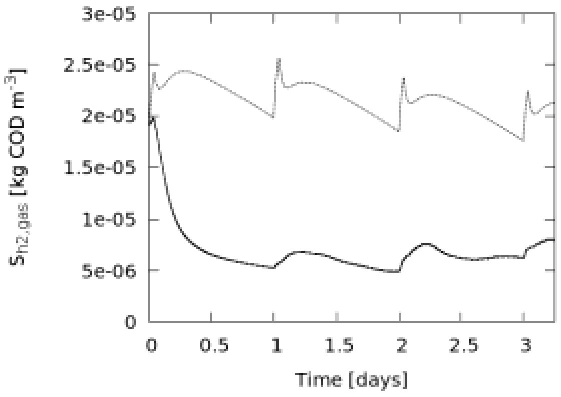


Figure 77: Hydrogen concentration, gas volume

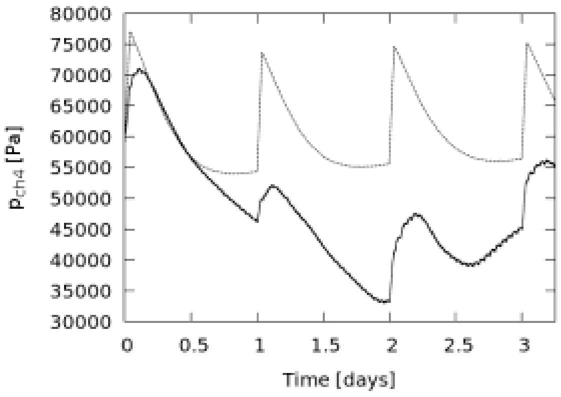


Figure 78: Methane partial pressure

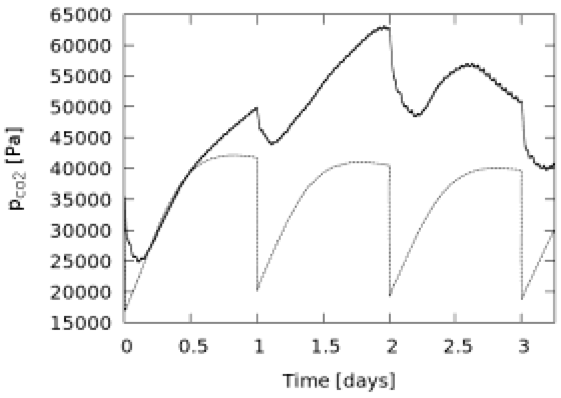


Figure 79: Carbon dioxide partial pressure

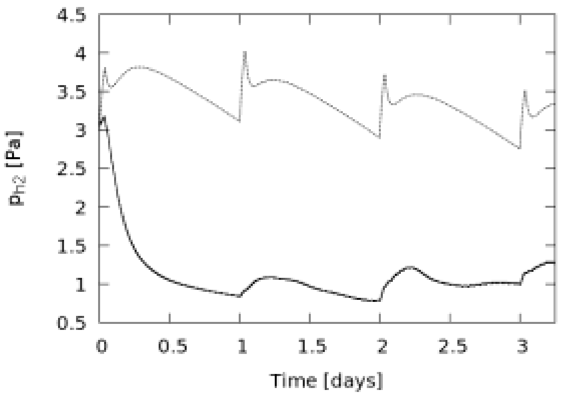


Figure 80: Hydrogen partial pressure

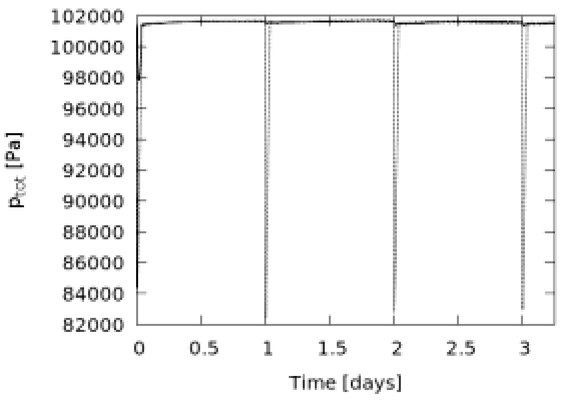


Figure 81: Total pressure

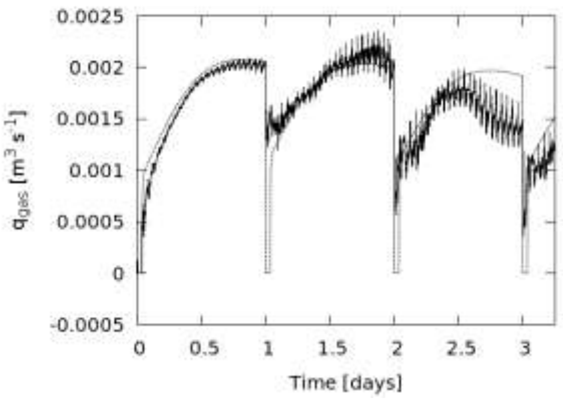


Figure 82: Gas flow rate

5.2.3.7 Microbial populations

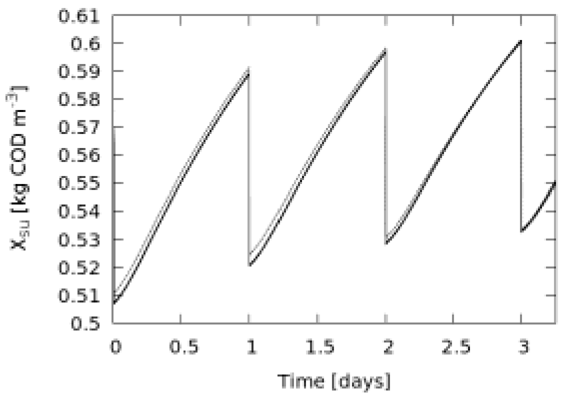


Figure 83: Sugar degraders

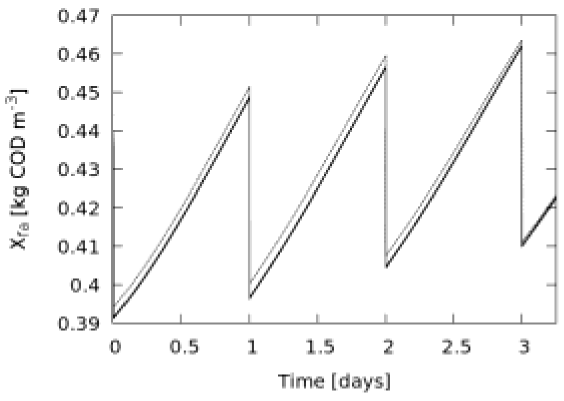


Figure 84: LCFA degraders

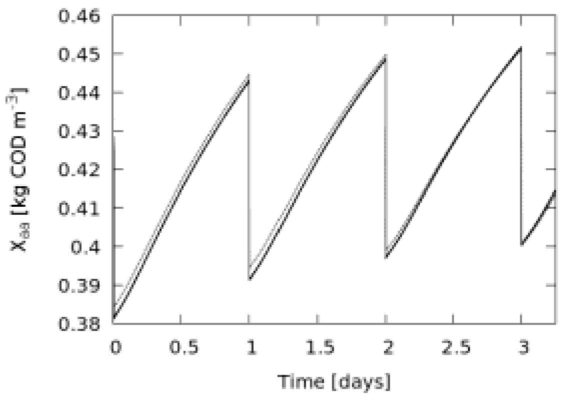


Figure 85: Amino acid degraders

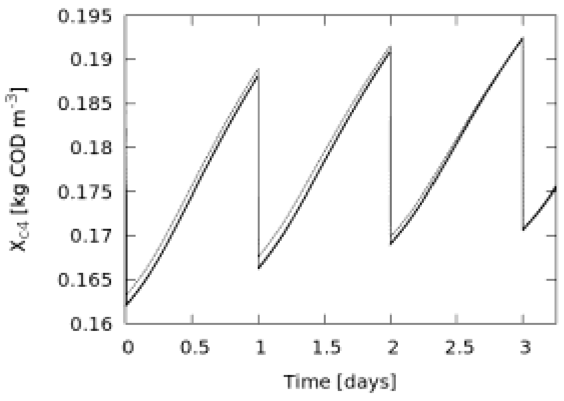


Figure 86: Valerate and butyrate degraders

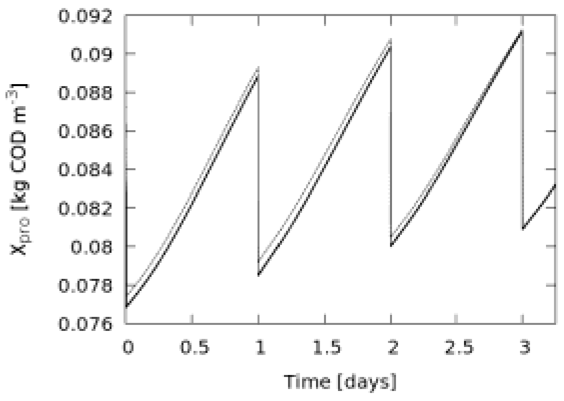


Figure 87: Propionate degraders

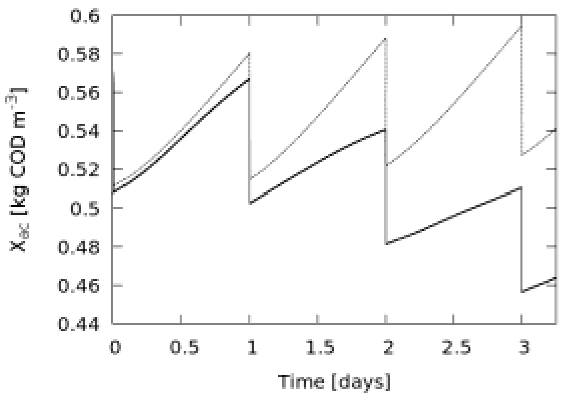


Figure 88: Acetate degraders

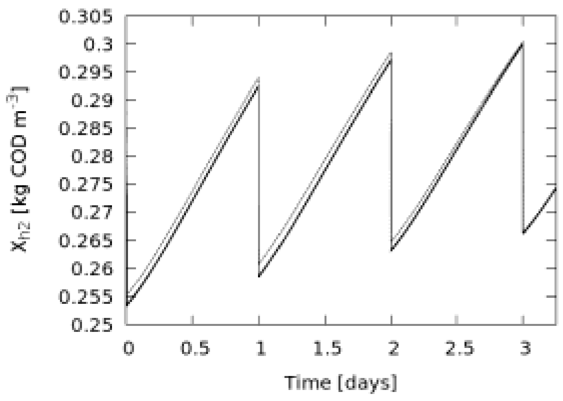


Figure 89: Hydrogen degraders

5.2.3.8 *Discussion*

The plots above show that obvious differences do exist between ADM-MDA and ADM1. This is not a modelling failure, but rather, these differences are caused by phenomena that can only manifest themselves in a spatially-resolved model. The fact that the two models show such significant differences is strong evidence that spatial resolution warrants further study for anaerobic digestion modelling.

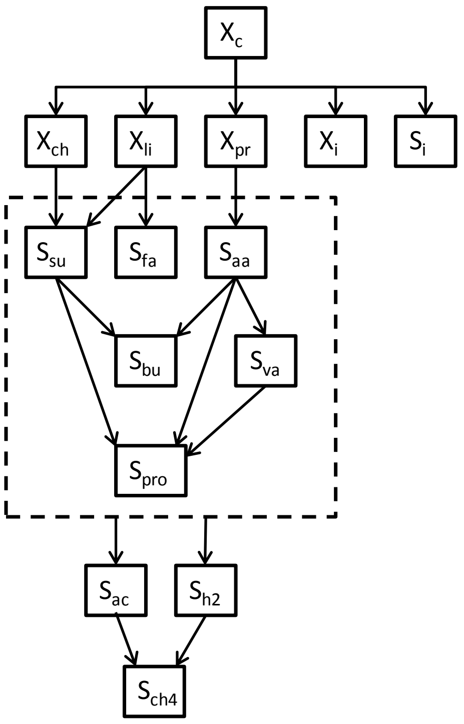


Figure 90: Simplified overview of pathways in an anaerobic digester

To help interpret the results, Figure 90 presents a simplified overview of the component break-down in a digester. At the top is particulate composites, X_c. This is the raw, undigested food that is fed to the digester. It contains large pieces of waste material, visible to the eye. The first step is disintegration, where X_c breaks up into smaller pieces characterized by five different variables. Two of these are inert, X_i and S_i, and contribute no further to the digestion process. The remaining three are X_{ch}, X_{li}, X_{pr}: carbohydrates, lipids, and proteins, respectively. These break down further through their own solubilisation process into S_{su}, S_{fa}, S_{aa}: sugars, fatty acids, and amino acids, which ferment into other products. All the products within the dotted box break down into acetate, S_{ac}, and hydrogen, S_{H2}, both of which form methane, S_{ch4}.

Turning to the results, all plots appear to show discontinuities at the start of each day. These actually are not discontinuities, but rather, fluid injection events that last two minutes each. Most species exhibit a drop in concentration as they are washed out during injection, but others, like X_c increase as new material is added to the digester. Figure 45 shows the profile of X_c, the particulate composites. The plot is a saw-toothed profile, with the upward spike representing the addition of new undigested material into the digester, and the gradual drop as this material is digested throughout the day. Although they have identical inlet conditions and flow rates, ADM-MDA consistently predicts higher peak values for X_c than ADM1. In this case, the ADM1 is in error, with its simplifying assumptions causing it to under-predict the peak value. During injection, fluid enters the inlet, while simultaneously, fluid is drawn out the outlet. There is insufficient time for the influent to travel from the inlet to the outlet during a single injection event. Therefore, the fluid flowing out the outlet is composed almost entirely of digested material from the previous day. On the other hand, the bulk model assumes the fluid is uniformly mixed at all times. Therefore, the food-rich flow at the inlet is instantly dispersed throughout the volume, allowing some of it to be removed through the outlet, resulting in an under-prediction of the peak value attained.

X_c has an initial value of 7.56 [kg COD m^{-3}], and an inlet value of 37 [kg COD m^{-3}]. The total fluid injection volume is 12 [m^3], and the digester volume is 100 [m^3]. Simple one-dimensional calculations based on these values show that X_c should be approximately 11.09 [kg COD m^{-3}] after the first fluid injection event. ADM-MDA predicts a value of 11.08 [kg COD m^{-3}], an under-prediction of 0.09%; whereas the bulk model predicts 10.88 [kg COD m^{-3}], an under-prediction of 1.9%.

The discrepancy in X_c between ADM1 and ADM-MDA is critical, as small differences in X_c cause large differences in the other species. The magnitude of these differences depends on yields, reaction rates, and the stage of digestion. For instance, the hydrolysis products S_{su} and S_{aa}, Figure 51 and Figure 52 respectively, show relatively small deviations, whereas S_{fa}, Figure 53, shows a larger deviation. This can be attributed to the fact that a larger proportion of mass decomposes through the pathway $X_c \rightarrow X_{li} \rightarrow S_{fa}$ than the others, based on the reaction yields.

The disintegration products, Figure 46 through Figure 48, all show the same pattern: sharp drops during fluid injection, as the inlet values of these are zero; followed by a sharp rise due to rapid disintegration, and a gradual fall as hydrolysis overtakes disintegration. Again, ADM-MDA predicts higher values than the ADM1, due to the primarily higher values of X_c. The inert disintegration products, Figure 49 and Figure 50 do not react further, therefore their profile is mostly linear, with its slope proportional to the disintegration reaction rate. The ADM-MDA and ADM1 predictions cross at day three. X_i and S_i are zero at the inlet, and therefore ADM1 overpredicts the amount that will wash out. However, ADM-MDA receives a small amount of extra mass when its greater X_c value disintegrates, and this soon overtakes ADM1's over-prediction.

The hydrolysis products are presented in Figure 51 through Figure 53. The amino acids, S_{aa} are the only hydrolysis products that also have mass coming in the inlet, and this gives its profile an extra spike and valley immediately following fluid injection. This is the uptake of the extra injected mass. The rounded maximum following this is the uptake of the mass generated from hydrolysis.

The first significant difference between ADM1 and ADM-MDA manifests itself with the total acetate concentrations, Figure 57. ADM-MDA shows an increasing trend, whereas ADM1 holds a relatively stable value. Since acetate is an interim product, a build-up of this species is an indicator of an unhealthy digester, potentially preceding a crash, or at the very least, representing a loss in the total productivity of the digester. ADM1 predicts a healthy digester, whereas ADM-MDA predicts an unhealthy digester. The only appreciable differences between the two simulations is that ADM-MDA incorporates spatial discretization. In other words, these results show that spatial discretization alone can make the difference between a model predicting a healthy digester, and one predicting an unhealthy digester. This is strong evidence that spatial discretization is an important component of anaerobic digestion modelling.

The question is: how could spatial discretization lead to such a significant difference in modelling outcome? To answer that, note that ADM-MDA predicts the acetate degraders, X_{ac}, Figure 88, are being washed out over the course of the digestion process. Spatial discretization alone can account for the discrepancy between ADM-MDA and ADM1 due to the latter's uniform mixture assumption. A reduction in acetate degraders leads to a slowdown of the aceticlastic methanogenesis, which reduces the rate at which acetate is consumed. Furthermore, much of the extra X_c mass in ADM-MDA breaks down into hydrogen and acetate, and from there the methanogens transform it into methane gas, either through hydrogenotrophic methanogenesis, or aceticlastic methanogenesis. Comparing the large discrepancy in acetate concentration, S_{ac}, Figure 57, with the small discrepancy in dissolved hydrogen gas concentration, S_{h2}, Figure 58, it is clear that the preferred pathway of the system as a whole is hydrogenotrophic methanogenesis. This may further exaggerate differences in acetate, S_{ac}.

The dissolved methane gas, S_{ch4}, Figure 59, also shows a significant discrepancy. This is likely caused by non-uniform mixing effects. Spatially, the dissolved methane gas collected near the centre of the reactor, away from the upper surface where gas transfer occurs. Therefore, less methane is available for transfer out of the digester, resulting in a build-up of the mean concentration within the digester liquid. Conversely, the concentration of methane gas in headspace at the top of the digester, $S_{ch4,gas}$, Figure 75, shows a reduction, which supports this hypothesis. With less mass transfer out of the bulk fluid, the dissolved methane gas, S_{ch4}, becomes supersaturated. The saturation concentration of dissolved methane gas is approximately 0.6 [kg COD m^{-3}]. Figure 59 shows the concentration exceeding this by a factor of 2.5. This is not surprising, as the bubble model was disabled (Section 5.1.1 on page 145); however, supersaturated gas concentrations have been reported in anaerobic digester effluent. This is a concern, as methane gas has greater impact on global warming than carbon dioxide. ADM-MDA has the potential to capture the gas physics with more detail than ADM1, as it will be able to resolve digesters with high sub-surface concentrations and low surface concentrations, as demonstrated here.

The gas flow rate, q_{gas}, Figure 82, is approximately equivalent between ADM1 and ADM-MDA, however the latter exhibits oscillatory behaviour, caused by differences in the modelling of the check valve. These oscillations propagate throughout the model, affecting many other variables, including most of the ion model species, Figure 69 through to Figure 74. ADM-MDA begins to show a relative drop-off in gas production starting midway through day two, most likely due to the acetate build-up. The composition of the biogas also differs significantly between models, with ADM-MDA producing less hydrogen, less methane, and more carbon dioxide, as seen in Figure 78 through Figure 80.

5.2.4 Spatially resolved data

Contour plots reveal details of the spatial distribution of various species throughout the digester. Close scrutiny of the results reveals a slight checkerboard pattern existing at times among some of the species. This pattern is very slight, with fluctuations less than 3%, as shown in Figure 91.

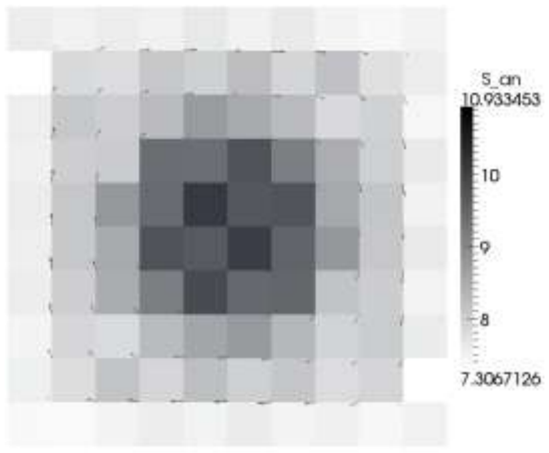
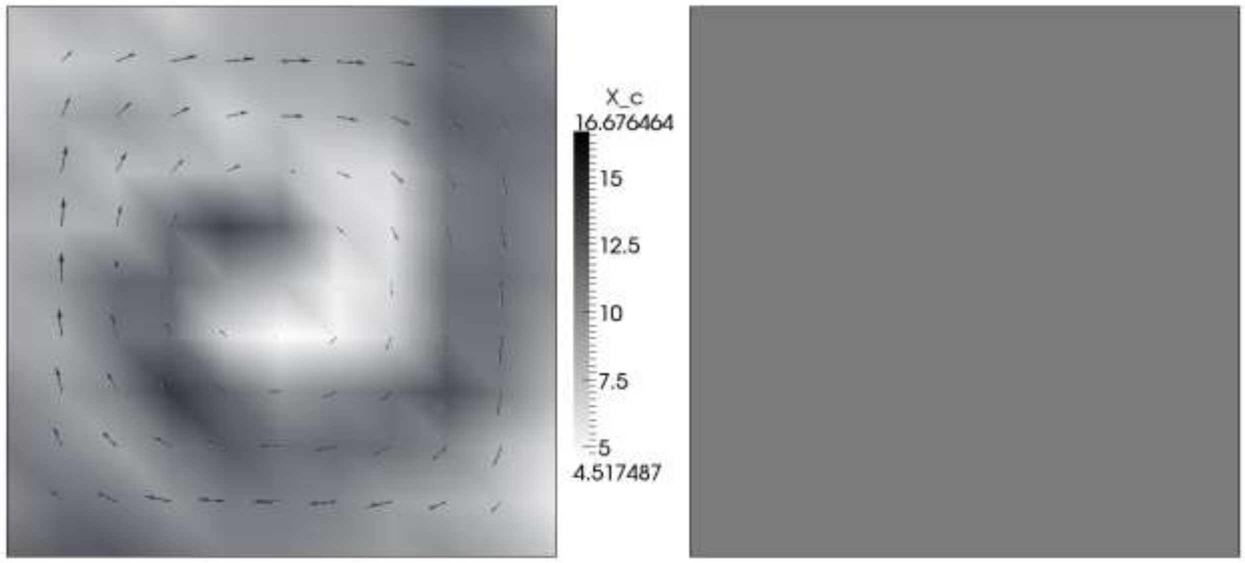


Figure 91: Soluble anion concentration, S_{an}, at t = 85185 [s]

Colour gradients disabled in this image in order to show the effect. The checkerboard pattern may be perceived along the boundaries. This is a numerical artefact caused by the use of a simple linear discretization scheme on a coarse mesh. As a test, a species with such checkerboarding, and the velocity field were used as initial values for OpenFOAM's scalar transport solver. With linear discretization, the checkerboarding remains; however, using a more sophisticated discretization scheme, such as QUICK, it instantly vanishes.

Figure 92 shows the particulate composites concentration, X_c, and velocity vectors at 1150 seconds into the first day. For comparison, the image on the right depicts the associated state of ADM1. Since there is no spatial resolution in this model, it appears as a single colour.



ADM-MDA **ADM1**

Figure 92: Particulate composites, X_c, spatial distribution at t = 1150 [s]

This is after the first fluid injection and mixing event, after giving the fluid approximately ten minutes to settle down. Steep concentration are still evident in the ADM-MDA. The food-rich fluid injected earlier can still be seen as a dark mass curling toward the centre of the digester. Similarly, Figure 93 shows the total acetate concentration, S_{ac}, at the same moment in time.

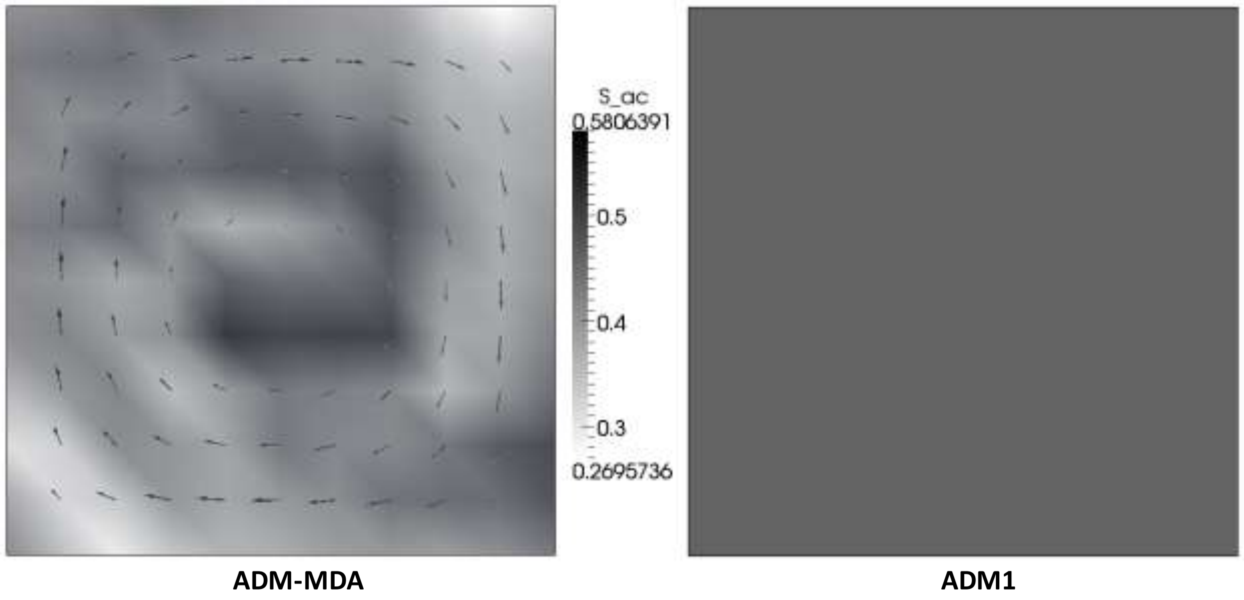


Figure 93: Total acetate concentration, S_{ac}, at t = 1150 [s]

Following a fluid injection event, the digester gradually stabilizes over the course of several hours, resulting in a spatial distribution with most mass concentrated in the centre of the reactor, observed in Figure 94.

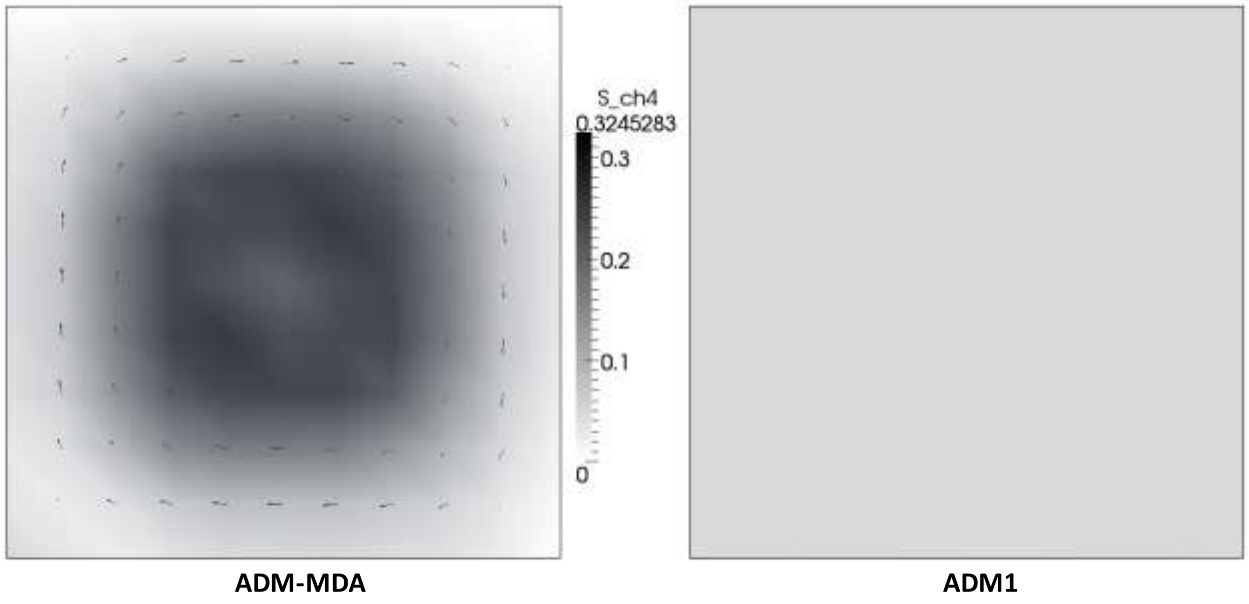


Figure 94: Dissolved methane concentration at t = 56,651 [s]

5.2.5 Animations

There is an animation accompanying this thesis. For readers who do not have access to this animation, Appendix P contains a series of screenshots from it. The animation shows a contour plot of the particulate composites, X_c, as it evolves through time, with red representing relatively high concentrations, and blue representing low concentrations. The video is a time-lapse, with one video second equal to approximately 379 simulation seconds, which means one day takes 3:48. Fluid injection events are obvious and occur at 0:00 and 3:48. There are frequent mixing events which occur approximately every 9.5 seconds, and last for 1.6 seconds in the video. In simulation time, these are ten minutes of mixing, spaced at hourly intervals. The mixing event gives sufficient momentum to the fluid that it never comes to a rest while the mixer is off, although it slows considerably.

Chapter 6 Conclusion

Despite significant research efforts into anaerobic digestion modelling, no practical model exists, which leaves the engineering process with fewer design tools. This state of affairs has contributed to the low adoption rate of anaerobic digesters, which are an environmental waste treatment and renewable energy option. The current state-of-the-art model, Anaerobic Digestion Model No. 1 (ADM1), is a bulk model that focuses on the biochemistry of the process, while largely ignoring the fluid dynamics. This thesis has improved on this model by adding an integrated fluid flow model, and changing its formulation to include spatial discretization. The new model, Anaerobic Digestion with Multi-Dimensional Architecture (ADM-MDA) cannot be solved using existing solvers, therefore this thesis creates a novel solver, Coupled Reaction-Advection-Flow Transient Solver (CRAFTS). CRAFTS is a novel partial differential algebraic equation (PDAE) solver, a general reacting liquid solver, and incorporates the first reported programmable logic controller (PLC) emulator directly integrated into a computational fluid dynamics (CFD) model. A rigorous set of tests and simulations verify all aspects of CRAFTS for proper function. Although CRAFTS is a capable general solver, this research finds that the numerical stiffness inherent in ADM1 limit the practicality of the CRAFTS-implemented ADM-MDA. In general, simulations are limited to:

- low mesh density;
- low velocity gradients;
- low concentration gradients; and
- “stable” biochemistry,

where the last one is to accommodate the fact that for some ranges of species concentrations, the biochemistry equations can become impractically stiff on their own. Despite this, the thesis performs an ADM-MDA simulation within the capabilities of CRAFTS, and compares the results to ADM1. The two models produce significantly different predictions about the outcome of the digestion process, with ADM-MDA predicting an unhealthy digester, and ADM1 predicting a healthy one. The spatial discretization of ADM-MDA allow it to resolve model physics that ADM1 cannot, and these reveal that for the selected conditions, the digester washes out its acetate degraders, and has a subsequent build-up of acetate. This drastic difference in modelling outcome underscores the importance of furthering study into spatial discretization for anaerobic digestion models, particularly if these models are to be used in the design process. A model that incorrectly predicts a productive digester would lead to design flaws. Therefore, the findings of this research suggest the ADM1 modelling community may benefit from exploring spatial discretization, and numerical methods that improve the practicality of ADM-MDA.

Chapter 7 Contribution

Several components of this work stand alone as contributions to the numerical or engineering sciences. This section summarizes these contributions, each of which will lead to a technical publication. These include:

- the source term stabilisation coupling method;
- the development and use of a novel PDAE solver;
- the implementation of **plcEmulator**, the first general PLC emulator within a CFD solver, and **multiSolver**, the formal implementation of algorithm-switching;
- the development of a general coupled liquid reacting solver; and
- a general three-dimensional implementation of ADM1.

This work also includes contributions to the research community that are not novel, but are useful. These include:

- **equationReader**: the implementation of an equation parser for OpenFOAM; and
- the publication of more than fifty wiki articles detailing the inner-workings of OpenFOAM, many of which are included in Appendix O.

7.1 Source term stabilisation coupling

Source term stabilisation coupling allows two disparate numerical frameworks to be coupled together. For instance, it permits an ODE solver and a PDE solver to operate independently, and simultaneously, collaborate on the result of a common output. It is a sort of all-purpose glue that can hold complex models together. No formal use of such a paradigm could be found in the open literature. The formal description and detailed example of its use in CRAFTS is a useful contribution to science and engineering.

7.2 PDAE solver

Conventional PDAE solvers, such as those that use semi-discretization, are based on ODE solution methods. They work by incorporating the full-field spatial discretization into the ODE solver's ddt and jacobian calculations, essentially reducing the problem at every time step to an ODE. There are two problems with this. First, this algorithm cannot be parallelized, which means it is unsuitable for large problems where multiple processors would be necessary. Secondly, ODE solver algorithms are optimized based on the assumption that the ddt and jacobian calculations are computationally cheap. Hence, they perform several such evaluations at interim points to arrive at a high quality estimate for the next timestep; some algorithms can perform upwards of forty interim evaluations. However, incorporating full-field discretization calculations into the ddt and jacobian routines makes their operation computationally expensive, and therefore ODE solution methods are not well-suited for solving PDAEs.

CRAFTS uses a novel algorithm for solving PDAEs based on PDE methods, which have evolved with the understanding that every evaluation is computationally expensive, and therefore are better suited for the demands of a PDAE system. CRAFTS works by joining a coupled PDE solver with the algebraic relations through source term stabilization coupling. This algorithm is parallelizable, and optimized for computationally intensive iterations.

No comparison has been made between these two different approaches. Regardless of the outcome of such a comparison, this novel PDE-based approach is a useful contribution to science and engineering.

7.3 Integrated PLC emulation and algorithm-switching

In the development of CRAFTS, a programmable logic controller (PLC) emulator was developed and integrated directly into OpenFOAM. Given the vast amount of process simulations performed in CFD, it is surprising that this has not been reported before, making CRAFTS the first CFD solver with a general integrated PLC emulator. Therefore, this represents a useful contribution to science and engineering.

7.4 General coupled liquid reacting solver

There are three components to a reaction solver:

- the flow solution;
- the advection of the species (species transport); and
- the reaction solution.

Conventional reaction solvers perform these three steps individually, as depicted in Figure 95.

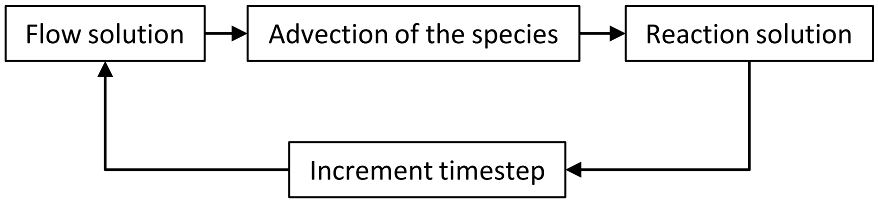


Figure 95: Flowchart of a segregated reaction solver

This method omits any non-linear effects between these phenomena. For instance, the model does not transport any quantity that may have been produced during the reaction stage of the current timestep. Therefore, such a solution method is theoretically incorrect, however, by imposing a limit on the *Courant number*, these errors can be reduced to a negligible size. The Courant number is given by:

$$C_o = \frac{\bar{u}\Delta t}{\Delta l}, \tag{83}$$

where:

- $\bar{u}$ is the mean velocity in the control volume;

- Δt is the timestep; and
- Δl is a characteristic length of the control volume.

The Courant number is the timestep divided by the residence time in a control volume. A Courant number of unity approximately indicates that the contents of the control volume will completely transport out of it in the next timestep. The Courant number should generally be much less than one for a valid solution. For a given mesh geometry, this acts as a restriction on the maximum timestep size.

CRAFTS, is named the Coupled Reaction-Advection-Flow Transient Solver because it couples these three steps together, as depicted in Figure 96.

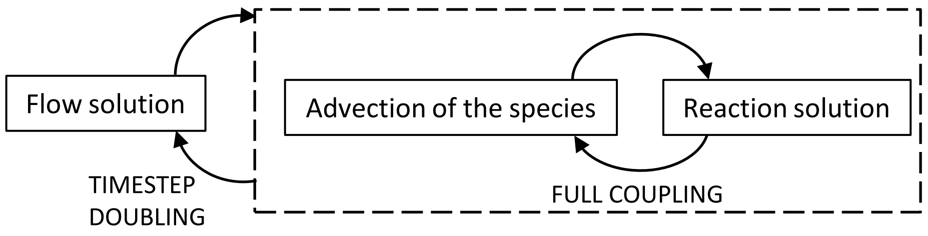


Figure 96: Flowchart of CRAFTS

The advection and reaction solution are directly coupled together within a block-matrix solver. The flow solution is loosely coupled through the adaptive timestep control, which will automatically reduce the timestep should the solution deviate. Therefore, CRAFTS is not strictly Courant number-limited, and therefore can take larger timesteps than conventional reaction solvers.

No such coupled reaction solver could be found reported in the open literature. This is a useful contribution to science and engineering. Furthermore, a segregated implementation of CRAFTS and ADM-MDA will be implemented for a comparative analysis.

7.5 General three-dimensional implementation of ADM1

Section 1.6.2.3 on page 22 establishes that no three-dimensional implementations of ADM1 exist, therefore CRAFTS implementation of ADM-MDA is a novel and useful contribution to engineering and science.

Furthermore, existing implementations of ADM1 are hard-coded, and require program modification to alter the biochemistry details. These include the MATLAB and Aquasim implementations reported in the literature. Therefore, CRAFTS is currently the only general ADM1-based solver available. A bulk model implementation of CRAFTS would allow researchers to take advantage of CRAFTS' flexibility, while not having to deal with a multi-dimensional model and the numerical stiffness problems it brings. Therefore a bulk model implementation of CRAFTS, along with data conversion utilities between OpenFOAM and MATLAB is planned.

Chapter 8 Suggested future work

The model presented above is complete and valid, but there are many opportunities to improve it and increase its capability. These opportunities can be divided into general model improvements, and ADM1-specific improvements.

8.1 General model improvements

General changes to the model include:

- finishing parallelisation;
- adding mesh motion;
- adding Lagrange particle tracking and settling; and
- implementing a Non-Newtonian model.

8.1.1 Parallelisation

Parallelisation is the act of modifying an algorithm, allowing it to work on several processors simultaneously. This is beneficial for algorithms that handle large amounts of data, as it reduces the total computation time. Parallelisation involves identifying when processors need to communicate, and handling this communication. OpenFOAM achieves parallelisation through geometry decomposition, wherein it subdivides the physical geometry into pieces and assigns each one to a processor.

The parallelisation efforts for ADM-MDA are incomplete, and therefore ADM-MDA cannot currently be run in parallel. The components that require parallel communications are:

- the solver applications, `craftsFoam` and `craftsPlcFoam`;
- the coupled reaction model;
- **multiSolver**;

- the **plcEmulator**; and
- the gas model.

Furthermore, users developing their own UDFs may have to implement parallelisation in the algebraic routines.

8.1.1.1 Parallelisation of the solver application

The parallelisation for the solver applications, `craftsFoam` and `craftsPlcFoam`, has not been designed. When running CRAFTS in parallel, all processors must be on the same timestep with the same delta *t* value. The timestep-doubling that CRAFTS uses has to be managed by the master processor, which then distributes the timestep information to the rest of the processors. Furthermore, convergence tests and next timestep size calculations are custom in CRAFTS, which require normalised root mean square calculations across the entire geometry. In parallel computing, a mechanism must be developed to handle this between processors. OpenFOAM has similar built-in functions, which can be used as templates.

8.1.1.2 Parallelisation of the coupled reaction model

The coupled reaction model has not been parallelised; however, Clifford (2011) has parallelized the block matrix solver on which it depends. Clifford's implementation can be used as a template for parallelising the coupled reaction solver.

8.1.1.3 Parallelisation of multiSolver

multiSolver is fully parallelised. Case directories in OpenFOAM differ between serial and parallel runs. **multiSolver** now accommodates these differences. Furthermore, the domain decomposition and recomposition applications were inadequate for **multiSolver**. To correct this, the post-processing application, `multiSolver`, now has functions that wrap the decomposition and recomposition utilities.

8.1.1.4 Parallelisation of the plcEmulator

The **plcEmulator** has not been parallelised. This component controls the simulation from a global perspective and cannot be made to run on multiple processors. Therefore, to parallelise the **plcEmulator**, it must reside entirely on the master processor. Its inputs must convert from multiple processors to one, and it must output globally to all processors. For example, the **plcEmulator** has an input that measures the volumetric average of a quantity. Each processor would measure the average in its own piece of the domain, not as a whole. Therefore, all processors must provide the volume-weighted sum and total volume, and the master processor can calculate the average.

8.1.1.5 Parallelisation of the gas model

The gas model has not been parallelised. The model has two components that require parallelisation. First, the gas boundary condition requires the volumetric average of the dissolved gas quantity, a calculation that should take place over the entire domain. However, it may be approximated by the local average, provided the processor's domain is sufficiently large. Second, the transient algebraic routine featuring the gas ODE solver must occur on only one processor. The gas volume model is a bulk model, and domain decomposition does not apply to it. Since the ODE calculations rely on values from the gas transfer boundary surface, this surface should reside on one processor, or communication needs to be set up to allow the other processors to share their values.

8.1.2 Mesh Motion

OpenFOAM has many built-in mesh motion capabilities, but CRAFTS and its components do not generally make use of them. In particular, **multiSolver** cannot handle mesh motion, and therefore the **plcEmulator** cannot implement control systems that modify the geometry of the problem, such as airfoil control surfaces. Implementing mesh motion in CRAFTS and its components should be relatively easy as most capabilities are already available in OpenFOAM. Furthermore, CRAFTS was implemented with the assumption that the mesh is not static, and therefore CRAFTS avoids the temptation to store mesh information locally, rather looking up the information from the mesh class when needed. This will help circumvent potential bugs when adding mesh motion to CRAFTS.

8.1.3 Lagrange particle tracking

CRAFTS is a three-dimensional coupled reaction solver. It differs from a bulk model in its treatment of the liquid volume, allowing for gradients to develop. However, CFD has many more capabilities that would give a reaction solver advantages over bulk models, the most obvious of which is particle tracking. With Lagrange particle tracking, CRAFTS would be able to resolve the disintegration stage of anaerobic digestion in detail. By implementing a settling model, CRAFTS would also be able to capture stagnation regions and volume loss caused by particle settling. By adding a particle reaction model based on previous work (Picioreanu et al., 2004; Picioreanu et al., 2005; Laspidou et al., 2005; Batstone et al., 2006b; and von der Schulenburg et al., 2009), CRAFTS would also be able to model biofilm reactors.

8.1.4 Non-Newtonian model

OpenFOAM has a thermophysical library that allows applications to solve Non-Newtonian fluid flow. ADM-MDA does not use this library, but implementing it should not be difficult.

8.2 ADM1-specific improvements

Changes to the model that are specific to ADM1 include:

- address the stiffness of ADM1;
- conducting an ADM1 coefficient and sensitivity analysis specific to ADM-MDA;
- implementing the ADM1-PWM mass-conserving model; and
- implementing a shear-induced decay model.

8.2.1 Addressing the stiffness of ADM1

The numerical stiffness inherent in ADM1 makes it very difficult to use in a practical way. Therefore, something must be done to address its stiffness. There are several possible solutions.

The gas model is the primary candidate for improving model stiffness. In all simulations exhibiting stiffness problems, numerical volatility was detectable earliest in the gas-related species. The pressure in the headspace affect the rate at which gas transfers out of the liquid, which in turn affects the pressure in the headspace. While these are trying to find equilibrium, the gas is flowing out of the headspace, reducing the pressure in the headspace. Altogether, this is a very stiff numerical system, and it is not clear what benefit is realized by resolving these short timescale dynamics. It may be possible to introduce some simplifications that remove these fluctuations, such as through an algebraic equilibrium routine. The gas model itself is implemented in an inelegant fashion, and modification of the algorithm will be challenging.

The ion model is another excellent candidate for improving model stiffness. After the gas model, ion model-related species showed the earliest volatility. ADM1 already suffered from ion model stiffness, and resolved it by removing the ion-related reactions from the kinetic model, and implementing them as an algebraic equilibrium routine. ADM-MDA was implemented in the same manner, and an earlier attempt to reverse this rendered ADM-MDA entirely useless. It may be necessary to simplify the ion model by reducing the number of species and reactions, or by eliminating it entirely.

It is worth noting that the single control volume simulations do not suffer from serious stiffness problems, and neither do multiple control volume simulations with flow disabled. The stiffness problems occur when the case uses multiple control volumes with fluid flow. The spatial transport of species apparently increases the model stiffness several orders of magnitude. This could be caused by the discretization schemes. The case settings for ADM-MDA always involved simple linear spatial discretization schemes. Furthermore, these schemes are responsible for the spatial checkerboarding seen in the final results. It is possible that higher order spatial discretization will yield improved numerical performance.

Should all these fail, the option is always available to remove components of ADM1 in an effort to produce a simpler, yet more numerically stable digester model.

8.2.2 Coefficient and sensitivity analysis

ADM1 has approximately 100 coefficients defined, most of which have to be adjusted to experimental data. Many published studies focus on this area (Jeong et al., 2004; Picioreanu et al., 2005; de Gracia et al., 2006; Batstone et al., 2006b; Tramšek et al., 2007; Page et al., 2008; Mu et al., 2008; Tartakovsky et al., 2008; Lee at al., 2009; and Batstone et al., 2009). Given the fact that even the simple case presented in Section 5.2 on page 147 showed differences between the bulk model and ADM-MDA, it is likely that the coefficients need to be modified for use in ADM-MDA. To streamline this process, in addition to trial and error, a sensitivity analysis can be conducted. A sensitivity analysis is where the coefficients are adjusted individually, and the effect is observed on the output. Comparing the change in coefficient with the change in output reveals to which coefficients the model is most sensitive.

8.2.3 Mass-conserving model

As mentioned earlier, ADM1 is not strictly mass conserving in the way it implements its carbon and nitrogen species. However, there is a mass-conserving variation known as the plant-wide model, ADM1-PWM (Huete et al., 2006; de Gracia et al., 2006; de Gracia et al., 2007; and Galí et al., 2009). Implementing this variation into ADM-MDA would make it possible to use mass fractions rather than scalars in species transport.

8.2.4 Shear-induced decay model

The effect of fluid dynamic characteristics on biochemical reactions is not well understood. High shear rates are known to be detrimental to micro-organisms (Yu et al. 2011; Hoffmann et al., 2008; Morales-Barrera et al., 2006). Therefore, a series of experiments are conducted to provide insights into the correlation between fluid flow and biochemical reactions. Dr. Cicek’s group at the University of Manitoba has performed a series of lab-scale anaerobic digestion experiments. A series of flow visualization experiments on identical equipment using particle image velocimetry (PIV) and acoustic Doppler velocimetry (ADV) was measured. These two datasets may make it possible to correlate the biochemical reactions with fluid flow. A draft paper detailing the PIV / ADV experiment is presented in Appendix L. As yet, no correlation has been made between these data, however, since these fluid dynamic characteristics are available to CRAFTS it is possible to implement a shear-induced decay model. Due to the flexible design of CRAFTS, this can be accomplished with the text input files alone, using custom coefficients in the decay reaction rates.

Bibliography

Alrawi, R. A.; Ahmad, A.; I., Norli; A. K., Mohd Omar. 2010. "Methane production during start-up phase of mesophilic semi-continues suspended growth anaerobic digester," International Journal of Chemical Reactor Engineering, v. 8: A89.

American Society of Agricultural Engineers. 2003. ASAE Standard: Manure Production and Characteristics. ASAE D384.1 FEB03. pp. 683-685.

Ampofo, F.; Karayiannis, T. G.. 2003. "Experimental benchmark data for turbulent natural convection in an air filled square cavity," International Journal of Heat and Mass Transfer, v. 46, pp. 3551-3572.

Ariesyady, H. D.; Ito, T.; Okabe, S.. 2007. "Functional bacterial and archaeal community structures of major trophic groups in a full-scale anaerobic sludge digester," Water Research, v. 41, pp. 1554-1568.

Axaopoulos, P.; Panagakis, P.; Tsavdaris, A.; Georgakakis, D.. 2001. "Simulation and experimental performance of a solar-heated anaerobic digester," Solar Energy, v. 70, n. 2, pp. 155-164.

Barros, P.; Ruiz, I.; Soto, M.. 2008. "Performance of an anaerobic digester-constructed wetland system for a small community," Ecological Engineering, v. 33, pp. 142-149.

Bernard, O.; Polit, M.; Hadj-Sadok, Z.; Pengov, M.; Dochain, D.; Estaben, M.; Labat, P.. 2001. "Advanced monitoring and control of anaerobic wastewater treatment plants: software sensors and controllers for an anaerobic digester," Water Science and Technology, v. 43, n. 7, pp. 175-182.

Batstone, D. J.; Hernandez, J. L. A.; Schmidt, J. E.. 2005. "Hydraulics of laboratory and full-scale upflow anaerobic sludge blanket (UASB) reactors," Biotechnology and Bioengineering, v. 91, n. 3, pp. 387-391.

Batstone, D. J.; Keller, J.; Angelidaki, I.; Kalyuzhni, S. V.; Pavlostathis, S. G.; Rozzi, A.; Sanders, W. T. M.; Siegrist, H.; Vavlin, V. A.. 2002. "Anaerobic Digester Model No.1 (ADM1)", Scientific and Technical Report No.13, IWA Task Group for Mathematical Modelling of Anaerobic Digestion Processes. IWA Publishing, London, United Kingdom.

Batstone, D. J.; Keller, J.; Steyer, J. P.. 2006a. "A review of ADM1 extensions, applications, and analysis: 2002-2005," Water Science and Technology, v. 54 n. 4, pp. 1-10.

Batstone, D. J.; Picioreanu, C.; van Loosdrecht, M. C. M.. 2006b. "Multidimensional modelling to investigate interspecies hydrogen transfer in anaerobic biofilms," Water Research, v. 40, pp. 3099-3108.

Batstone, D. J.; Tait, S.; Starrenburg, D. 2009. "Estimation of hydrolysis parameters in full-scale anaerobic digesters," Communication to the Editor, Biotechnology and Bioengineering, v. 102, n. 5., pp. 1513-1520.

Biswas, J.; Chowdhury, R.; Bhattacharya, P.. 2007. "Mathematical modeling for the prediction of biogas generation characteristics of an anaerobic digester based on food / vegetable residues," Biomass and Bioenergy, v. 31, pp. 80-86.

Blumensaat, F.; Keller, J.. 2005. "Modelling of two-stage anaerobic digestion using the IWA Anaerobic Digestion Model No. 1 (ADM1)," Water Research, v. 39, pp. 171-183.

Bourque, L.; Koroluk, R.. 2003. "Manure storage in Canada," Farm Environmental Management in Canada, v. 1, n. 1. Statistics Canada Catalogue no. 21-021-MIE. Available at http://www.statcan.gc.ca/pub/21-021-m/21-021-m2003001-eng.htm (accessed January 29, 2011).

Brdjanovic, D.; Mithaiwala, M.; Moussa, M. S.; Amy, G.; van Loosdrecht, M. C. M.. 2007. "Use of modelling for optimization and upgrade of a tropical wastewater treatment plant in a developing country," Water Science and Technology, v. 56, n. 7, pp. 21-31.

Britton, A.; Koch, F. A.; Mavinic, D. S.; Adnan, A.; Oldham, W. K.; Udala, B.. 2005. "Pilot-scale struvite recovery from anaerobic digester supernatand at an enhanced biological phosphorus removal wastewater treatment plant," Journal of Environmental Engineering and Science, v. 4, pp. 265-277.

Caretta, L. S.; Gosman, A. D.; Patankar, S. V.; Spalding, D. B., (1972). “Two calculation procedures for steady, three-dimensional flows with recirculation,” Proceedings of the Third International Conference on Numerical Methods in Fluid Dynamics, Paris.

Chen, S.-T.; Hsu, C.-Y.; Berthouex, P. M., 2006. "Fate and modeling of pentachlorophenol degradation in a laboratory-scale anaerobic sludge digester," Journal of Environmental Engineering, v. 132, n. 7, pp. 795-802.

Cicek, N.; Lambert, S.; Venema, H. D.; Snelgrove, K. R.; Bibeau, E. L.; Grosshans, R.. 2006. “Nutrient removal and bio-energy production from Netley-Libau Marsh at Lake Winnipeg through annual biomass harvesting,” Biomass and Bioenergy, v. 30, pp. 529-536. doi:10.1016/j.biombioe.2005.12.009

Clifford, I.. 2011. “Block coupled simulations using OpenFOAM,” 6th OpenFOAM Workshop. Penn State, State College, PA., USA. June 13-16.

Clifford, I.; Jasak, H.. 2009. "The application of a multi-physics toolkit to spatial reactor dynamics," American Nuclear Society - International Conference on Mathematics, Computational Methods and Reactor Physics 2009, M and C 2009, pp. 1585-1596.

Cline, T.; Thomas, N.; Shumway, L.; Yeung, I.; Hansen, C. L.; Hansen, L. D.; Hansen, J. C.. 2010. "Method for evaluating anaerobic digester performance," Bioresource Technology, v. 101, pp. 8623-8626.

Comerford, J. M.; Picken, D. J.. 1985. "Free-convective mixing within an anaerobic digester," Biomass, v. 6, pp. 235-245.

Comino, E.; Rosso, M.; Riggio, V.. 2009. "Development of a pilot scale anaerobic digester for biogas production from cow manure and whey mix," Bioresource Technology, v. 100, pp. 5072-5078.

de Dieuvleveult, C.; Erhel, J.; Kern, M.. 2009. "A global strategy for solving reactive transport equations," Journal of Computational Physics, v. 228, pp. 6395-6410.

Espinosa-Solares, T.; Valle-Guadarrama, S.; Bombardiere, J.; Domaschko, M.; Easter, M.. 2009. "Effect of heating strategy on power consumption and performance of a pilot plant anaerobic digester," Applied Biochemistry and Biotechnology, v. 156, pp. 465-474.

Estaben, M.; Polit, M.; Steyer, J. P.. 1997. "Fuzzy control for an anaerobic digester," Control Engineering Practice, v. 5, n. 98, pp. 1303-1310.

Fantozzi, F.; Buratti, C.. 2009. "Biogas production from different substrates in an experimental Continuously Stirred Tank Reactor anaerobic digester," Bioresource Technology, v. 100, pp. 5783-5789.

Ferrer, I.; Vázquez, F.; Font, X.. 2010. "Long term operation of a thermophilic anaerobic reactor: Process stability and efficiency at decreasing sludge retention time," Bioresource Technology, v. 101, pp. 2972-2980.

Fischer, J. R.. 1979. "Design criteria and operational guidelines for a pilot-scale anaerobic digester," Resource Recovery and Conservation, v. 4, pp. 1-11.

Fleming, J. G.. 2002. "Novel simulation of anaerobic digestion using computational fluid dynamics," Ph.D. Thesis, Faculty of Mechanical Engineering, North Carolina State University, North Carolina, United States.

Gaden, D. L. F.; Bibeau, E. L.. 2012. “A programmable logic controller emulator with automatic algorithm-switching for computational fluid dynamics using OpenFOAM,” Proceedings of The Canadian

Society for Mechanical Engineering International Congress 2012, CSME International Congress 2012, Winnipeg, Canada, June 4-6. Article submitted.

Galí, A.; Benabdallah, T.; Astals, S.; Mata-Alvarez, J.. 2009. "Modified version of ADM1 model for agro-waste application," Bioresource Technology, v. 100, pp. 2783-2790.

Gervardi, M. H.. 2003. The Microbiology of Anaerobic Digesters. John Wiley & Sons, Hoboken, New Jersey, United States.

de Gracia, M.; Huete, M.; Garcia-Heras, J. L.; Ayesa, E.. 2007. "Algebraic solution of the mass balanced ADM1 to predict the steady-state and to optimise the design of the anaerobic digestion of sewage sludge," Water Science and Technology, v. 56, n. 9, pp. 127-136.

de Gracia, M.; Sancho, L.; Garcia-Heras, J. L.; Vanrolleghem, P.; Ayesa, E.. 2006. "Mass and charge conservation check in dynamic models: application to the new ADM1 model," Water Science and Technology, v. 53, n. 1, pp. 225-240.

Gujer, W.; Henze, M.; Mino, T.; Van Loosdrecht, M.. 1999. “Activated sludge model no. 3,” Water Science and Technology, v. 39, n. 1, pp. 183-189.

Hansen, C. L.; West, G. T.. 1992. "Anaerobic digestion of rendering waste in an upflow anaerobic sludge blanket digester," Short Communication, Bioresource Technology, v. 41, pp. 181-185.

Harikishan, S.; Shihwu, S.. 2003. "Cattle waste treatment and Class A biosolid production using temperature-phased anaerobic digester," Advances in Environmental Research, v. 7, pp. 701-706.

He, P.. 2010. "Anaerobic digestion: an intriguing long history in China," Editorial, Waste Management, v. 30, pp. 549-550.

Hill, D. T.. 1983a. “Simplified monod kinetics of methane fermentation of animal wastes,” Agricultural Wastes, v. 5, pp. 1-16.

Hill, D. T.. 1983b. “Energy consumption relationships for mesophilic and thermophilic digestion of animal manures,” Transactions of the American Society of Agricultural Engineering, v. 26, n. 3, pp. 841-848.

Hoffmann, R. A.; Garcia, M. L.; Veskivar, M.; Karim, K.; Al-Dahhan, M. H.; Angenent, L. T.. 2008. "Effect of shear on performance and microbial ecology of continuously stirred anaerobic digesters treating animal manure," Biotechnology and Bioengineering, v. 100, pp. 38-48.

Hofmann, N.. 2008. “A geographical profile of livestock manure production in Canada, 2006”, EnviroStats, v. 2, n. 3. Statistics Canada Catalogue no. 16-002-x. Available at http://www.statcan.gc.ca/pub/16-002-x/2008004/article/10751-eng.htm (accessed 2011-01-28).

Huete, E.; de Gracia, M.; Ayesa, E.; Garcia-Heras, J. L.. 2006. "ADM1-based methodolgy for the characterisation of the influent sludge in anaerobic reactors," Water Science and Technology, v. 54, n. 4, pp. 157-166.

Hori, T.; Haruta, S.; Ueno, Y.; Ishii, M.; Igarashi, Y.. 2006. "Dynamic transition of a methanogenic population in response to the concentration of volatile fatty acids in a thermophilic anaerobic digester," Applied Environmental Microbiology, v. 72, n. 2, pp. 1623-1630.

Ike, M.; Inoue, D.; Miyano, T.; Liu, T. T.; Sei, K.; Soda, S.; Kodashin, S.. 2010. "Microbial population dynamics during startup of a full-scale anaerobic digester treating industrial food waste in Kyoto eco-energy project," Bioresource Technology, v. 101, pp. 3952-3957.

Illmer, P.; Gstraunthaler, G.. 2009. "Effect of seasonal changes in quantities of biowaste on full scale anaerobic digester performance," Waste Management, v. 29, pp. 162-167.

Inoue, S.; Sawayama, S.; Ogi, T.; Yokoyama, S., (1995). "Organic composition of liquidized sewage sludge," Biomass and Bioenergy, v. 10, pp. 37-40.

Issa, R. I., (1986). “Solution of implicitly discretized fluid flow equations by operator-splitting,” Journal of Computational Physics, v. 62., pp. 40-65

Jacob, J.; Le Lann, J; Pinguad, H.; Capdeville, B.. 1996. "A generalized approach for dynamic modelling and simulation of biofilters: application to waste-water denitrification," Chemical Engineering Journal, v. 65, pp. 133-143.

Jeong, H.; Suh, C.; Lim, J.; Lee, S., Shin, H.. 2004. "Analysis and application of ADM1 for anaerobic methane production," Bioprocess and Biosystems Engineering, v. 27, pp. 81-89.

Jeppsson, U.; Pons, M.-N.; Nopens, I.; Alex, J.; Copp, J. B.; Gernaey, K. V.; Rosen, C.; Steyer, J.-P.; Vanrolleghem, P. A. 2007. "Benchmark simulation model no 2: general protocol and exploratory case studies," Water Science and Technology, v. 56, n. 8, pp. 67-78.

Jones, R.; Parker, W.; Khan, Z.; Murthy, S.; Rupke, M.. 2008. "Characterization of sludges for predicting anaerobic digester performance," Water Science and Technology, v. 57, n. 5, pp. 721-726.

Jordaan, E. M.; Ackerman, J.; Cicek, N.. 2010. "Phosphorus removal from anaerobically digested swine wastewater through struvite precipitation," Water Science and Technology, v. 61, n. 12.

Kalia, A.. 1988. "Development and evaluation of a fixed dome plug flow anaerobic digester," Biomass, v. 16, pp. 225-235.

Karim, K.; Varma, R.; Vesvikar, M.; Al-Dahhan, M. H.. 2004. "Flow pattern visualization of a simulated digester," Water Research, v. 38, pp. 3659-3670.

Karim, K.; Hoffmann, R.; Klasson, K. T.; Al-Dahhan, M. H.. 2005a. "Anaerobic digestion of animal waste: effect of mode of mixing," Water Research, v. 39, pp. 3597,3606.

Karim, K.; Hoffmann, R.; Klasson, T.; Al-Dahhan M. H.. 2005b. "Anaerobic digestion of animal waste: waste strength versus impact of mixing," Bioresource Technology, v. 96, pp. 1771-1781.

Karim, K.; Klasson, K. T.; Hoffmann, R.; Drescher, S. R.; DePaoli, D. W.; Al-Dahhan, M. H.. 2005c. "Anaerobic digestion of animal waste: effect of mixing," Bioresource Technology, v. 96, pp. 1607-1612.

Kauder, J.; Boes, N.; Pasel, C.; Herbell, J.-D.. 2007. "Combining models ADM1 and ASM2d in a sequencing batch reactor simulation," Chemical Engineering Technology, v. 30, n. 8, pp. 1100-1112.

Keshtkar, A.; Meyssami, B.; Abolhamd, G.; Ghaforian, H.; Asadi, M. K.. 2003. "Mathematical modeling of non-ideal mixing continuous flow reactors for anaerobic digestion of cattle manure," Bioresource Technology, v. 87, pp. 113-124.

Khanal, S. K.. 2008. Anaerobic Biotechnology for Bioenergy Production. Wiley-Blackwell, Ames, Iowa, United States.

Kissling, K.; Springer, J.; Jasak, H.; Sch*u*tz, S.; Urban, K.; Piesche, M.. 2010. "A coupled pressure based solution algorithm based on the volume-of-fluid approach for two or more immiscible fluids," V European Conference on Computational Fluid Dynamics, ECCOMAS CFD 2010, Lisbon, Portugal, June 14-17.

Kleerebezem, R.; van Loosdrecht, M. C. M.. 2006. "Critical analysis of some concepts proposed in ADM1," Water Science and Technology, v. 54, n. 4, pp. 51-57.

Kunz, A.; Miele, M.; Steinmetz, R. L. R.. 2009. "Advanced swine manure treatment and utilization in Brazil," Bioresource Technology, v. 100, pp. 5485-5489.

Langner, J. 2009. “Investigation of non-Newtonian flow in anaerobic digesters,” Masters Thesis, Faculty of Mechanical Engineering, University of Manitoba, Winnipeg, Manitoba.

Laspidou, C. S.; Kungolos, A.; Samaras, P., (2005), "Advances in mathematical modelling of biofilm structures," WIT Transactions on Ecology and the Environment, v. 103, pp. 231-240.

Lee, M.-Y.; Suh, C.-W.; Ahn, Y.-T.; Shin, H.-S.. 2009. "Variation of ADM1 by using temperature-phased anaerobic digestion (TPAD) operation," Bioresource Technology, v. 100, pp. 2816-2822.

Leighton, I. R.; Forster, C. F.. 1996. "Mixing characteristics of a two-phase anaerobic digester," Transactions of the Institute of Chemical Engineers, v. 74 p. B, pp. 99-104.

Linder, G.. 2009. “Failure Analysis and Smart Grid Control Protocols for Anaerobic Digesters: Web Edition,” Masters Thesis, Department of Electrical Engineering, Clarkson University. Available at http://linderlabs.com/glinder/innovations/thesis/ (accessed 2011-02-01).

Lübken, M.; Wichern, M.; Schlattmann, M.; Gronauer, A.; Horn, H.. 2007. "Modelling the energy balance of an anaerobic digester fed with cattle manure and renewable energy crops," Water Research, v. 41, pp. 4085-4096.

Lucht, W.; Strehmel, K.. 1998. "Discretization based indices for semilinear partial differential algebraic equations," Applied Numerical Mathematics, v. 28, pp. 371-386.

Lusk, P.. 1998. Methane Recovery from Animal Manures: The Current Opportunities Casebook. National Renewable Energy Laboratory, Golden, CO, United States. Available at http://agrienvarchive.ca/bioenergy/download/methane.pdf (accessed 2011-01-31).

Lusk, P. D.. 1991. "Comparative economic analysis: anaerobic digester case study," Bioresource Technology, v. 36, pp. 223-228.

Ma, J.; Scott, N. R.; DeGloria, S. D.; Lembo, A. J.. 2005. "Siting analysis of farm-based centralized anaerobic digester systems for distributed generation using GIS," Biomass and Bioenergy, v. 28, pp. 591-600.

Marchiam, U.; Krause, C.. 1993. "Propionic to acetic acid ratios in overloaded anaerobic digestion," Bioresource Technology, v. 43, pp. 195-203.

Mattocks, R.; Wilson, R.. 2005. "Latest trends in anaerobic digestion in North America," BioCycle, v. 46, n. 2, pp. 60-63.

Mavinic, D. S.; Koch, F. A.; Huang, H.; Lo, K. V.. 2007. "Phosphorus recovery from anaerobic digester supernatants using a pilot-scale struvite crystallization process," Journal of Environmental Engineering Science, v. 6, pp. 561-571.

McConnell, S.. 2006. Software Estimation: Demystifying the Black Art. Microsoft Press, Redmond, WA, USA.

Meroney, R. N.; Colorado, P. E.. 2009. "CFD simulation of mechanical draft tube mixing in anaerobic digester tanks," Water Research, v. 43, pp. 1040-1050.

Morales-Barrera, L.; Cristiani-Urbina, E.. 2006. "Removal of hexavalent chromium by Trichoderma viride in an airlift bioreactor," Enzyme and Microbial Technology, v. 40, pp. 107-113.

Moser, M.; Mattocks, R.; Goldstein, J.. 2002. "Anaerobic digester innovations at a Colorado hog farm," BioCycle, v.43, n. 5, pp. 33-34.

Mu, S. J.; Zeng, Y.; Wu, P.; Lou, S. J.; Tartakovsky, B.. 2008. "Anaerobic digestion model no. 1-based distributed parameter model of an anaerobic reactor: I. Model development," Bioresource Technology, v. 99, pp. 3665-3675.

Nielsen, A. M.; Spanjers, H.; Volcke, E. I. P.. 2008. "Calculating pH in pig manure taking into account ionic strength," Water Science and Technology, v. 51, n. 11, pp. 1785-1790.

Ojolo, S. J.; Bamgboye, A. I.; Ogunsina, B. S.; Oke, S. A.. 2008. "Analytical approach for predicting biogas generation in a municipal solid waste anaerobic digester," Iranian Journal of Environmental Health, Science and Engineering, v. 5, n.3, pp. 179-186.

Ozkan-Yucel, U. G.; Gökçay, C. F.. 2010. "Application of ADM1 model to a full-scale anaerobic digester under dynamic organic loading conditions," Environment Technology, v. 31, n. 6, pp. 633-640.

Page, D. I.; Hickey, K. L.; Narula, R.; Main, A. L.; Grimberg, S. J.. 2008. "Modeling anaerobic digestion of dairy manure using the IWA Anaerobic Digestion Model no. 1 (ADM1)," Water Science and Technology, v. 58, n. 3, pp. 689-695.

Paquin, D.; Liang, T.. 2006. "Estimating the spatial variation of anaerobic digester heating potential," Biosystems Engineering, v. 95, n. 2, pp. 227-233.

Parker, W. J.. 2005. "Application of the ADM1 model to advanced anaerobic digestion," Bioresource Technology, v. 96, pp. 1832-1842.

Parker, W. J.; Wu, G.-H.. 2006. "Modifying ADM1 to include formation and emission of odourants," Water Science and Technology, v. 54, n. 4, pp. 111-117.

Picioreanu, C.; Batstane, D. J.; van Loosdrecht, M. C. M.. 2005. "Multidimensional modelling of anaerobic granules," Water Science and Technology, V. 52, n. 1-2, pp. 501-507.

Picioreanu, C.; Kreft, J.-U.; van Loosdrecht, M. C. M.. 2004. "Particle-based multidimensional multispecies biofilm model," Applied and Environmental Microbiology, v. 70, n. 5, pp. 3024-3040.

Press, W. H.; Teukolsky, S. A.; Vetterling, W. T.; Flannery, B. P.. 1992. Numerical Recipes in C, the Art of Scientific Computing, Second Edition. Cambridge University Press, New York, New York, USA.

Rodríguez, J.; Lema, J. M.; van Loosdrecht, M. C. M.; Kleerebezem, R.. 2006. "Variable stoichiometry with thermodynamic control in ADM1," Water Science and Technology, v. 54, n. 4, pp. 101-110.

Rodríguez, J.; Roca, E.; Lema, J. M.; Bernard, O.. 2008. "Determination of the adequate minimum model complexity required in anaerobic bioprocesses using experimental data," Journal of Chemical Technology and Biotechnology.

Rosen, C.; Jeppsson, U.. 2006. “Aspects on ADM1 implementation within the BSM2 framework,” Technical Report No. LUTEDX/(TEIE-7224)/1-35/(2006). Department of Industrial Electrical Engineering and Automation, Lund University, Lund, Sweden.

Rudniak, L.; Machniewski, P. M.; Milewska, A.; Molga, E.. 2004. "CFD modelling of stirred tank chemical reactors: homogeneous and heterogeneous reaction systems," Chemical Engineering Science, v. 59, pp. 5233-5239.

Sanchez, E.; Montalzo, S.; Travieso, L.; Rodriguez, X.. 1995. "Anaerobic digestion of sewage sludge in an anaerobic fixed bed digester," Biomass and Bioenergy, v. 9, n. 6, pp. 493-495.

Schindler, D. W.. 1977. "Evolution of phosphorus limitation in lakes," Science, New Series, v. 195, n. 4275, pp. 260-262.

von der Schulenburg, D. A. G.; Pintelon, T. R. R.; Picioreanu, C.; Van Loosdrecht, M. C. M.. 2009. "Three-dimensional simulations of biofilm growth in porous media," American Institute of Chemical Engineers Journal, Bioengineering, Food and Natural Products, v. 55, n.2, pp. 494-504.

Scruton, D. L.; Weeks, S. A.; Achilles, R. S.. 2004. "Strategic hurdles to widespread adoption of on-farm anaerobic digesters," ASAE Annual International Meeting 2004, pp. 5985-5991.

Shang, Y.; Johnson, B. R.; Sieger, R. 2005. "Application of the IWA Anaerobic Digester Model (ADM1) for simulating full-scale digestion," Water Science and Technology, v. 52, n.1-2, pp. 487-492.

Siegrist, H.; Vogt, D.; Garcia-Heras, J. L.; Gujer, W.. 2002. "Mathematical model for meso- and thermophilic anaerobic sewage sludge digestion," Environmental Science and Technology, v. 36, pp. 1113-1123.

Simeon, B.. 1998. "DAEs and PDEs in elastic multibody systems," Numerical Algorithms, v. 19, pp. 235-246.

Simeon, B.; Arnold, M.. 2000. "Coupling DAEs and PDEs for simulating the interaction of pantograph and catenary," Mathematical and Computer Modelling of Dynamical Systems, v. 6, pp. 129-144.

Statistics Canada. 2002-present – a. Cattle Statistics, 2010. Statistics Canada Catalogue no. 23-012-x, v. 9 n. 2. Available at http://www.statcan.gc.ca/pub/23-012-x/23-012-x2010001-eng.htm (accessed 2010-01-28).

Statistics Canada. 2002-present – b. Hog Statistics, Second Quarter 2010. Statistics Canada Catalogue no. 23-010-X. Available at http://www.statcan.gc.ca/pub/23-010-x/23-010-x2010003-eng.htm (accessed 2010-01-28).

Steinberg, L. M.; Regan, J. M.. 2008. "Phylogenetic comparison of the methanogenic communities from an acidic oligotrophic fen and an anaerobic digester treating municipal wastewater sludge," Applied Environmental Microbiology, v. 74, n. 21, pp. 6663-6671.

Tartakovsky, B.; Mu, S. J.; Zeng, Y.; Lou, S. J.; Guiot, S. R.; Wu, P.. 2008. "Anaerobic digestion model No. 1-based distributed parameter model of an anaerobic reactor: II. Model validation," Bioresource Technology, v. 99, pp. 3676-3684.

Terashima, M.; Goal, R.; Komatsu, K.; Yasui, H.; Takahashi, H.; Li, Y. Y.; Noike, T.. 2009. "CFD simulation of mixing in anaerobic digesters," Bioresource Technology, v. 100, pp. 2228-2233.

Tramšek, M.; Goršek, A.; Glavič, P.. 2007. "Methodology for determination of anaerobic digestion kinetics using a bench top digester," Resources, Conservation and Recycling, v. 51, pp. 225-236.

Türker, M.; Çelen, I.. 2007. "Removal of ammonia as struvite from anaerobic digester effluents and recycling of magnesium and phosphate," Bioresource Technology, v. 98, pp. 1529-1534.

Valle-Guadarrama, S.; Espinosa-Solares, T.; López-Cruz, I. L.; Domaschko, M.. 2011. "Modeling temperature variations in a pilot plant thermophilic anaerobic digester," Bioprocess and Biosystems Engineering, article in press, accepted November 6, 2010.

Vázquez-Padín, J.; Fernádez, I.; Figueroa, M.; Mosquera-Corral, A.; Campos, J.; Méndez, R.. 2009. "Applications of Anammox based processes to treat anaerobic digester supernatant at room temperature," Bioresource Technology, v. 100, pp. 2988-2994.

Vesvikar, M. S.; Al-Dahhan, M.. 2005. "Flow pattern visualization in a mimic anaerobic digester using CFD," Biotechnology and Bioengineering, v. 89, n. 6, pp. 719-732.

Voegeli, Y.; Zurbrügg, C.. 2008. "Decentralised anaerobic digestion of kitchen and market waste in developing countries - 'state-of-the-art' in South India," Proceedings Venice 2008, Second International Symposium on Energy from Biomass and Waste.

Waeger, F.; Delhaye, T.; Fuchs, W.. 2010. "The use of ceramic microfiltration and ultrafiltration membranes for particle removal from anaerobic digester effluents," Separation and Purification Technology, v. 73, pp. 271-278.

Wagner, Y.. 2000. "A further index concept for linear PDAEs of hyperbolic type," Mathematics and Computers in Simulation, v. 53, pp. 287-291.

Weld, R. J; Singh, R.. 2011. "Functional stability of a hybrid anaerobic digester / microbial fuel cell system treating municipal wastewater," Bioresource Technology, v. 102, pp. 842-847.

Wright, R. J.; Kemper, W.D.; Millner, P.D.; Power, J.F.; Korcak, R.F.. 1998. Agricultural Uses of Municipal, Animal, and Industrial Byproducts. U.S. Department of Agriculture, Agricultural Research Service, Conservation Research Report No. 44, 135 pp, ch. 4.

Wu B.; Bibeau E. L.; Kifie G.. 2006. “Development of 3-D Anaerobic digester heat transfer model for cold weather applications,” American Society of Agricultural and Biological Engineers ASABE, v. 49, n. 3, pp. 749-757.

Wu, B.; Chen, S.. 2008. "CFD simulation of non-Newtonian fluid flow in anaerobic digesters," Biotechnology and Bioengineering, v. 99, n. 3, pp. 700-711.

Young, P. J.. 1984. "Odours from effluent and waste treatment," Effluent and Water Treatment Journal, v. 24, n. 5, pp. 189-195.

Yu, L.; Ma, J.; Chen, S.. 2011. "Numerical simulation of mechanical mixing in high solid anaerobic digester," Bioresource Technology, v. 102, pp. 1012-1018.

Appendix A Additional literature review

A.1 Studies that focus on the overall process

Lusk (1991) compared the economic performance of two different farm-based anaerobic digesters: an earthen psychrophilic digester, and a mesophilic STR digester. The author concludes the psychrophilic digester is more economic due to a lower initial cost, and lower annual operational costs. Hansen et al. (1992) and Sanchez et al. (1995) performed lab-scale experimental investigations into the effect of changing the hydraulic retention time (HRT). Barros et al. (2008) implemented a treatment system that incorporated anaerobic digesters and constructed wetlands, with the goal of reducing wastewater treatment costs in rural areas. They evaluated the system performance based on effluent quality. Ojolo et al. (2008) studied household waste digestion and performed a regression analysis to produce an empirical model that predicts biogas output. Fantozzi et al. (2009) performed a lab-scale experiment that varied the substrate fed to the digester, and measured the resulting biogas output and composition. Comino et al. (2009) developed a pilot-scale anaerobic digester and reported on its methane output performance. Illmer et al. (2009) monitored the gas output and other parameters of a full-scale digester in Australia over the course of two years. The authors reported on an observed seasonal variation, and speculate the main cause was a seasonal variation in the composition of the wastes fed to the digester. Alrawi et al. (2010) performed a series of lab-scale experiments to evaluate the effect of different seeding compositions on the methane production during the critical start-up phase of a digester. They found that volatile suspended solids (VSS) and COD had a direct impact on the methane produced. Cline et al. (2010) present a new technique of measuring digester performance. They prescribe using several micro-reactors, each with an amount of fluid from the digester, and a prescribed pollutant or toxin. The performance of each micro-reactor contributed to the overall evaluation of the anaerobic digester. Ferrer et al. (2010) performed a series of lab-scale experiments aimed at finding the optimum solids retention time (SRT) on thermophilic digesters, evaluating methane production and other indicators in response to a change in SRT and organic loading rate (OLR). The authors concluded that 10 days was a minimum SRT, but 15 days yielded better quality effluent. Weld et al. (2011) evaluated the stabilizing effects of combining an anaerobic digester with a microbial fuel cell in a variety of configurations. The authors evaluated the system performance under stress caused by imposing a change in the pH level. They found that the anaerobic digester on its own was more stable than the hybrid system.

A.2 Studies focusing on the effluent

The treatment of wastewater is often the primary purpose of an anaerobic digestion system, in which case, the quality of the effluent is most important. Several studies focus on the effluent, such as quantifying it, or evaluating strategies to improve its quality. These studies are in a similar vein to the general studies above. Harikishan et al. (2003) evaluated the effluent quality from a temperature phased anaerobic digester at different loading rates. Several experiments investigated removal of nutrients using struvite precipitation (Britton et al., 2005; Mavinic et al., 2007; Türker et al., 2007; and Jordaan et al., 2010). Britton et al. and Mavinic et al. both looked at phosphorus removal; each reported a 90% recovery rate. Türker et al. focused on ammonia removal and the economics of the process. Jordaan et al. (2010) evaluated the conditions conducive to struvite precipitation, including pH and nutrient ratio. The authors achieved an 80% phosphorus recovery rate. Chen et al. (2006) evaluated the effectiveness of anaerobic digesters to dechlorinate pentachlorophenol, a pollutant commonly used as a wood preservative. They found that a semi-continuously fed digester performed better than a batch-fed digester. Vázquez-Padín et al. (2009) looked at nitrogen removal from digester effluents. Waeger et al. (2010) experimented with filtration techniques, which when applied to anaerobic digester effluents achieved an 85% drop in COD and a 20% drop in nitrogen.

A.3 Studies focusing on fluid flow

Several studies focus on the motion of fluid within the digester. These experiments include dye tracer studies, particle tracking, and characterization of mixing strategies. Comerford et al. (1985) evaluated the practicality of using a heater to mix the digester through free convection. They found it performed poorly: the area below the heater became a dead space, and solids tended to settle out. Leighton et al. (1996) performed residence time studies on a lab-scale digester comparing two different types of feed. The authors also developed a dimensionless parameter related to the degree of mixing. Karim et al. (2004) evaluated gas recirculation mixing using a draft tube. They found good mixing at the top of the digester, and total stagnancy at the bottom. Karim et al. (2005a, 2005b) compared three different mixing strategies at different waste concentrations, measured in percent total solids (TS). They found low waste concentrations (at 5% TS), there was no difference; however, above that the mixed digesters produced 10-30% more biogas than the unmixed. The authors also noted that mixing during start-up proved counter-productive as this reduced the overall pH, and led to process instability. Karim et al. (2005c) studied different draft tube mixing configurations. The authors found air leakage confounded the data, a testimony to the delicacy of the anaerobic process.

A.4 Studies focusing on design

Some papers pertain mostly to design, providing readers with advice, given the authors' experiences over the course of an anaerobic digester experiment. Fischer (1979) provides guidelines on the construction of a pilot-scale anaerobic digester, including a sizing strategy for an internal heat exchanger. On the other hand, Scruton et al. (2004) recommend against internal heat exchangers after rebuilding a farm-based digester at a demonstration site, owing to digester plugging and maintenance concerns. They also point out that insulation at the top of a flexible covered digester is expensive and unnecessary due to the insulative properties of the foam that forms in the digester. They further suggest that steam injection is an economic technique to pre-heat waste as it flows into the digester, a concern especially for cold climates. Kalia (1988) addressed some inadequacies with a conventional digester design for use in a cold climate in India. The author developed an improved digester that outperformed the conventional design, and detailed the design modifications. Axaopoulos (2001) describes the design of an anaerobic digester with solar panels on the cover. Wu et al. (2009) investigated how reactor designs affect digester heat loss in cold climates.

A.5 Studies focusing on control systems

Studies related to anaerobic digester control systems occupy a significant area in the literature. Esteban et al. (1997) developed a flow control system for an anaerobic digester using a fuzzy logic controller. The authors used a simplified anaerobic digester model to calibrate the system, and tested it on a pilot-scale digester. Bernard et al. (2001) developed two different controllers for alkalinity and flow rates of anaerobic digesters: a fuzzy logic controller, and an adaptive linearizing controller. The latter required the development of an anaerobic digester model. The authors tested the controllers on a pilot-scale digester and compared their performance. Espinosa-Solares et al. (2009) compared two different temperature control strategies: one with a 0.2 °C temperature range, the other with a 1.0 °C temperature range. The authors found improved output with the wider range, and speculate that this was caused by improved convective mixing.

A.6 Studies focusing on microbiology

Several studies focus on the microbiology of anaerobic digesters under various conditions. Hori et al. (2006) quantified the microbial populations as they transitioned before, during, and after an induced stressful event. Ariesyady et al. (2007) characterized the microbial populations and composition in a full-scale anaerobic sludge digester. Steinberg et al. (2008) quantified genetic differences between methanogens in a psychrophilic digester with those found in a natural wetland known as a fen. Ike et al.

(2010) studied the microbial composition during the start-up of a full-scale anaerobic digester for treating industrial food wastes. The authors were able to observe and characterize all stages of anaerobic digestion.

Appendix B ODE theory

B.1 ODE theory

All ordinary differential equations (ODEs) can be expressed as a set of first order ODEs, given by:

$$\frac{dy_i(x)}{dx} = f_i(x, y_0, y_1, \ldots, y_N), \quad i = 0, \ldots, N, \tag{A1}$$

where the functions f_i are known. There are numerous algorithms available to solve this system of equations, including:

- Euler's method;
- the fourth-order Runge-Kutta method; and
- the semi-implicit Bulirsch-Stoer method.

B.1.1 Euler's method

Euler's method is given by:

$$y_{n+1} = y_n + hf(x_n, y_n), \tag{A2}$$

where:

- *h* is the step size; and
- *y* is the dependent variable – concentration, S_{var}, in ADM1;
- *x* is the independent variable – time, *t*, in ADM1; and
- *f* is the rate of change of the dependent variable with respect to the dependent variable – the time derivative of concentration, dS_{var} / dt, in ADM1.

Press et al. (1992) point out this method is asymmetric, advancing the solution based on the value at only one end. The authors go on to recommend against using Euler's method:

There are several reasons that Euler's method is not recommended for practical use, among them, (i) the method is not very accurate when compared to other, fancier, methods run at the equivalent stepsize, and (ii) neither is it very stable. – Press et al. (1992), §16.1.

B.1.2 Fourth-order Runge-Kutta method

The fourth-order Runge-Kutta method is superior to Euler's method, and is popular for ODE integration among engineers and scientists. The method takes four trial steps before calculating the final value for each time step. It is given by:

$$k_0 = hf(x_n, y_n), \tag{A3}$$

$$k_1 = hf(x_n + \frac{h}{2}, y_n + \frac{k_0}{2}), \tag{A4}$$

$$k_2 = hf(x_n + \frac{h}{2}, y_n + \frac{k_1}{2}), \tag{A5}$$

$$k_3 = hf(x_n + h, y_n + k_2), \text{ and} \tag{A6}$$

$$y_{n+1} = y_n + \frac{k_0}{6} + \frac{k_1}{3} + \frac{k_2}{3} + \frac{k_3}{6} + O(h^5). \tag{A7}$$

Most problems are well suited for this method, but if the ODE system is stiff, or if high accuracy is required, other methods should be explored, such as the semi-implicit Bulirsch-Stoer method.

B.1.3 Semi-implicit Bulirsch-Stoer method (SIBS)

The semi-implicit Bulirsch-Stoer method (SIBS) makes use of Richardson extrapolation, a technique that accelerates the evaluation of a sequence. In SIBS, the end-point of an ODE integration is calculated numerous times, using increasingly finer steps. SIBS then extrapolates these results to the value that theoretically would be obtained using infinite steps. The formulation is based on the semi-implicit midpoint rule, given by:

$$\left[\mathbf{1} - h\frac{\partial \mathbf{f}}{\partial \mathbf{y}}\right] \bullet \mathbf{y}_{n+1} = \left[\mathbf{1} + h\frac{\partial \mathbf{f}}{\partial \mathbf{y}}\right] \bullet \mathbf{y}_{n-1} + 2h\left[\mathbf{f}\left(\mathbf{y}_n\right) - \frac{\partial \mathbf{f}}{\partial \mathbf{y}} \bullet \mathbf{y}_n\right]. \tag{A8}$$

SIBS, as with many ODE solution methods, requires calculation of the Jacobian, $\partial\mathbf{f}/\partial\mathbf{y}$, in addition to the derivative with respect to the independent variable, $\partial\mathbf{f}/\partial x$, or the time derivative for ADM1.

Appendix C Source term stabilisation coupling

Source term stabilisation coupling is a general numerical method that allows incompatible solver methods to be coupled, even when they share outputs. It is a powerful method that acts as a sort of all-purpose glue, holding complex models together. The method can be used to serve two purposes: first, if the output from one equation is the input for others, source term stabilisation coupling passively converts incompatible outputs into useable inputs; second, if multiple equations have the same output, source term stabilisation coupling acts as a feedback mechanism between equations, causing them to arrive at a universally agreed output. This method is used routinely throughout CRAFTS.

C.1 Terminology

To clarify discussion on source term stabilisation coupling, some terminology needs to be established, as represented in Figure A1.

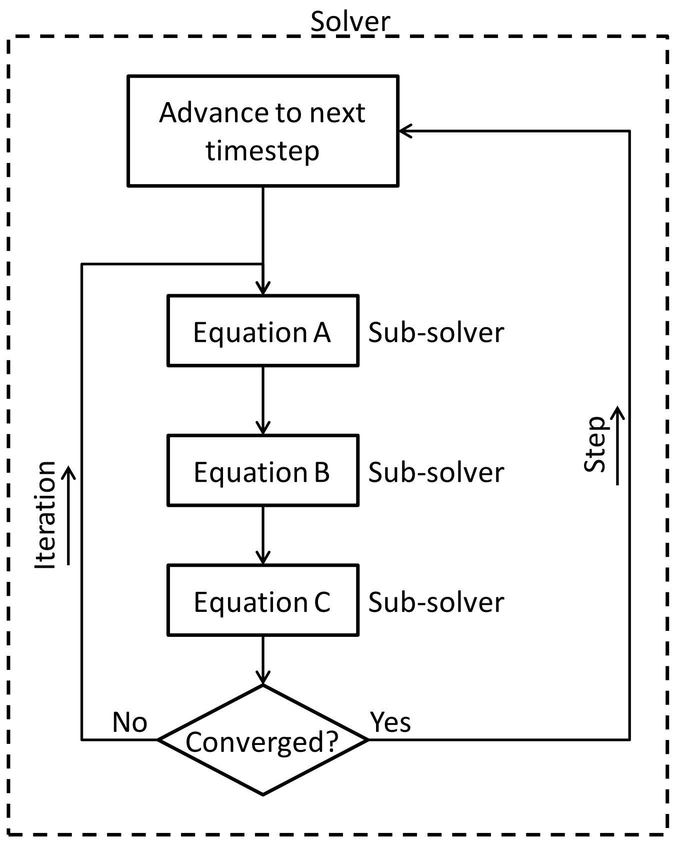


Figure A1: Terminology for stabilisation coupling

Sub-solver – an algorithm that solves a single governing equation.

Solver – encompasses all sub-solvers, and the coupling strategy. It solves all governing equations.

Iteration – a single cycle through all the sub-solvers. It takes many iterations to achieve a valid step.

Step – an advancement of the model. Usually these are timesteps, but in general they advance an independent variable.

C.2 Input to output

In its simplest implementation, source term stabilisation coupling converts incompatible output from one sub-solver into useable input for other sub-solvers. Consider a sub-solver, named Source Process, which produces the output, *T*. This output is required by a number of other sub-solvers, named

Destination Process 1, Destination Process 2, and so forth. To implement source term stabilisation coupling:

1. for each Destination Process:
    a. modify the numerical framework to calculate *T* as an *output*, instead of as an input;
    b. in the equation for *T*, assign a source term; and
    c. the source term must be controllable to produce any desired solution for *T*;
2. solve the Source Process to obtain the solution for *T*;
3. calculate a new source term for each Destination Process;
4. solve each Destination Process; and
5. compare the source terms against those from the previous iteration: if they have stabilised, this timestep is done; otherwise return to Step 2.

This method is written in a general way. Step 1 and Step 3 vary depending on the type of numerical analysis being undertaken, and not all of them are suitable to source term stabilisation coupling. For example, a numerical analysis method that cannot produce more than one output cannot be used here.

A more concrete implementation of this can be seen in CRAFTS where it uses this method to couple its coupled reaction model, a PDE solver, with the implicit variables, T_i, solved in the algebraic routines. The coupled reaction model requires these implicit variables as inputs, but the output from the algebraic routines may be time-dependent, and are therefore unsuitable as inputs into the PDE solver. To handle this, the implicit variables are added to the coupled reaction model as dependent variables, governed by this relation:

$$\frac{\partial T_i}{\partial t} = S_i, \tag{A9}$$

where S_i is the source term, given by:

$$S_i|_{t_n \dots t_n+1} = \frac{T_i|_{t_{n+1}} - T_i|_{t_n}}{t_{n+1} - t_n}, \tag{A10}$$

where:

- t_{n+1} and t_n are the next timestep and the previous timestep, respectively; and
- T_i are the values resulting from the algebraic routines at each timestep.

This is a simple forward-differencing linear interpolation scheme. Its guarantees that the solution for T_i calculated by the coupled reaction model will be identical to those from the algebraic routine, provided the time discretization scheme used by the PDE solver when evaluating Equation (A9) is the same as the Equation (A10). The coupled reaction model and the algebraic routines repeat until S_i stabilises.

This example uses linear interpolation, but it is possible to employ methods that are more complicated.

C.3 Shared output

In situations where more than one sub-solver share the same output, source term stabilisation coupling acts as a feedback mechanism that allows the sub-solvers to affect one another, and arrive at a universal output. This is achieved by adding a source term to the governing equations of all sub-solvers involved and establishing a relative measure of each sub-solver's effect on the output.

Consider several sub-solvers named Process 1, Process 2 and so on. Each process solves for the same variable, T, which may be a point, curve, surface, or any other form of output. The change in T across a step is given by:

$$\Delta T = T|_{t_{n+1}} - T|_{t_n}, \tag{A11}$$

where t_n and t_{n+1} are the current step and the next step, respectively. This change can be expressed as a sum of the individual contributions from each sub-solver:

$$\Delta T = \Delta T_1 + \Delta T_2 + \ldots + \Delta T_N = \sum_i T_i. \tag{A12}$$

Each one of these individual contributions, ΔT_i, can be added to a sub-solver as a source term, S_i. Therefore, when executing a sub-solver, the effects of all other sub-solvers can be present. Furthermore, the corresponding ΔT_i values can be subtracted from the result, giving the contribution of a single sub-solver.

To implement this method in a general way:

1. modify each sub-solver:
    a. add source terms and other terms necessary to the governing equation for T such that any given ΔT_i can be reproduced as an additive effect in the solution; and
    b. define a ΔT_i value in the solver, i.e. a global variable.
2. initialise all ΔT_i:
    a. to zero; or
    b. to the value from the previous step; or
    c. to a value using extrapolation from previous steps;
3. for each Process A:
    a. calculate the source terms S_i, for all $i = 1 \ldots N$, except for $i = A$;
    b. execute sub-solver A;
    c. from the result, T, subtract ΔT_i, for all $i = 1 \ldots N$, except for $i = A$;
    d. store the remaining result as ΔT_A; and
4. repeat Step 3, applying relaxation as necessary until all ΔT_i stabilise;
5. advance the timestep and return to Step 2.

The specific details of each step will vary depending on the numerical method of each sub-solver. Some numerical methods are unsuitable. Furthermore, the introduction of a source term into the governing equations of a sub-solver may destabilised the solution and render it divergent.

For a better understanding, consider two sub-solvers, Process A and Process B, both of which solve for T as a function of time. For simplicity, assume the governing equations are, for Process A:

$$\frac{dT}{dt} = f_A(t), \tag{A13}$$

and for Process B:

$$\frac{dT}{dt} = f_B(t). \tag{A14}$$

Solving these individually between time t_0 and t_1 produces two curves of T with respect to time, such as those shown in Figure A2. At this point, the two solutions are independent.

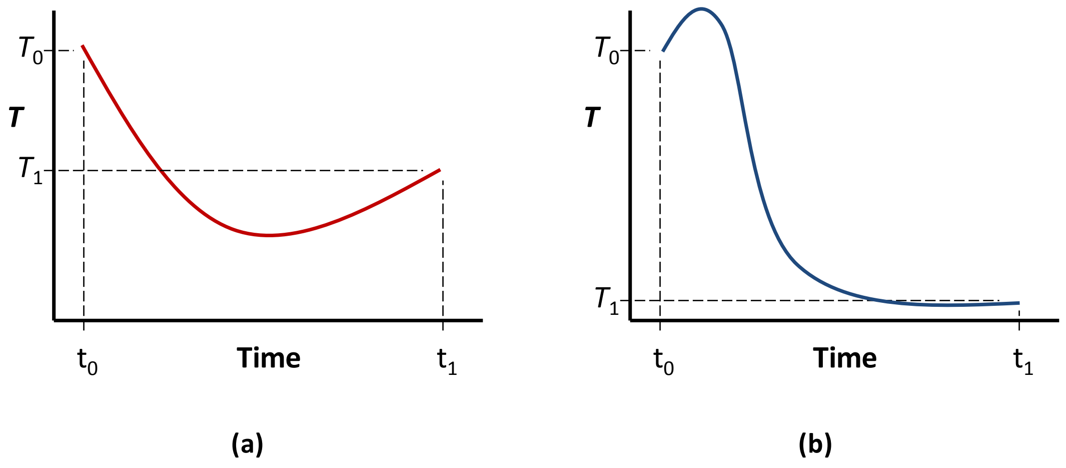


Figure A2: Example output curves for (a) Process A and (b) Process B

To implement the source term stabilisation coupling method, only Process A is solved independently. This initialises the method. Using standard curve-fitting techniques, the method creates an approximation of the solution from Process A, given by:

$$T^* = g_A(t). \tag{A15}$$

Since the relative effect of each process is more important, the method establishes a datum at time t_0, giving:

$$\Delta T_A = g_A(t) - T|_{t=t_0}. \tag{A16}$$

The method converts this into a source term by taking the derivative with respect to time:

$$S_A(t) = \frac{d\ g_A(t)}{dt}. \tag{A17}$$

Now, the method adds this source term to the governing equation of Process B:

$$\frac{dT}{dt} = f_B(t) + S_A(t). \tag{A18}$$

This has the effect of adding the solution from Process A into the sub-solver for Process B. The resulting solution will not be the same as adding together the two solutions, as shown in Figure A3. It is true that the source term acts as a form of linear addition, but since the sub-solver for Process B is deriving a new solution with it included, non-linear effects will be present.

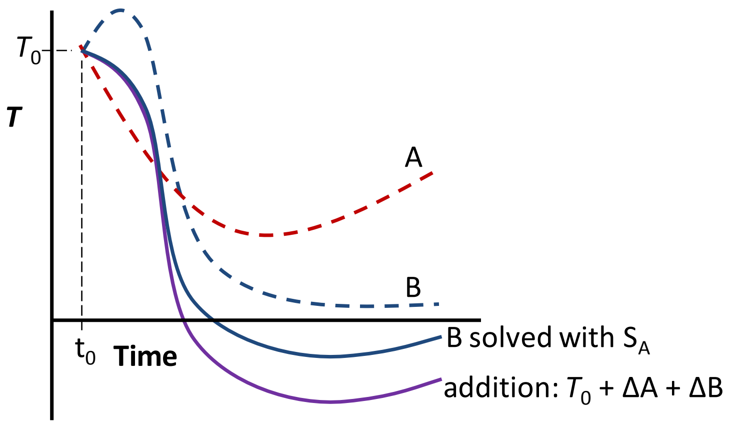


Figure A3: Comparison of the original curves, simple addition, and use of a source term

As before, the method uses standard curve-fitting techniques to find an approximation of the output, giving:

$$T^* = g_B(t). \tag{A19}$$

To derive the effect of Process B on the output, the method subtracts the change from other processes and the datum, yielding:

$$\Delta T_B = g_B(t) - \Delta T_A - T|_{t=t_0}. \tag{A20}$$

The method converts this into a source term for Process A by taking the derivative with respect to time:

$$S_B(t) = \frac{d\ g_B(t)}{dt}, \tag{A21}$$

which is added to the governing equation for Process A:

$$\frac{dT}{dt} = f_A(t) + S_B(t). \tag{A22}$$

The sub-solver for Process A produces a new solution. To derive the individual effect of Process A, Equation (A16) is modified to remove effects from Process B, according to:

$$\Delta T_A = g_A(t) - \Delta T_B - T|_{t=t_0}. \tag{A23}$$

This repeats until the solution stabilises.

Appendix D ADM1 theory details

D.1 Variables

There are 46 variables in the variation of ADM1 implemented here. These can be broken down into the categories of:

- ODE variables;
- algebraic variables; and
- derived variables.

D.1.1 ODE variables

Table A1 – ADM1 ODE variables

i	Symbol	Type	Loction	Initial	Inlet	Units	Description
0	S_{su}	ODE	Fluid	0.024309	0	[kg m^{-3}]	Monosaccharides
1	S_{aa}	ODE	Fluid	0.010808	0.05	[kg m^{-3}]	Amino acids
2	S_{fa}	ODE	Fluid	0.29533	0	[kg m^{-3}]	Long chain fatty acids
3	S_{va}	ODE	Fluid	0.02329	0	[kg m^{-3}]	Valerate
4	S_{bu}	ODE	Fluid	0.031123	0	[kg m^{-3}]	Butyrate
5	S_{pro}	ODE	Fluid	0.043974	0	[kg m^{-3}]	Propionate
6	S_{ac}	ODE	Fluid	0.50765	0	[kg m^{-3}]	Acetate
7	S_{h2}	ODE	Fluid	4.9652E-07	0	[kg m^{-3}]	Disolved hydrogen gas
8	S_{ch4}	ODE	Fluid	0.055598	0	[kg m^{-3}]	Disolved methane gas
9	S_{ic}	ODE	Fluid	102.58	6	[mol m^{-3}]	Inorganic carbon
10	S_{in}	ODE	Fluid	103.73	70	[mol m^{-3}]	Inorganic nitrogen
11	S_{i}	ODE	Fluid	3.2327	0.06	[kg m^{-3}]	Soluble inerts
12	X_{c}	ODE	Fluid	7.5567	37	[kg m^{-3}]	Composites
13	X_{ch}	ODE	Fluid	0.074679	0	[kg m^{-3}]	Carbohydrates
14	X_{pr}	ODE	Fluid	0.074679	0	[kg m^{-3}]	Proteins
15	X_{li}	ODE	Fluid	0.11202	0	[kg m^{-3}]	Lipids
16	X_{su}	ODE	Fluid	0.57565	0	[kg m^{-3}]	Sugar degraders

17	X_{aa}	ODE	Fluid	0.43307	0	[kg m^{-3}]	Amino acid degraders
18	X_{fa}	ODE	Fluid	0.44433	0	[kg m^{-3}]	LCFA degraders
19	X_{c4}	ODE	Fluid	0.18404	0	[kg m^{-3}]	Valerate and butyrate degraders
20	X_{pro}	ODE	Fluid	0.087261	0	[kg m^{-3}]	Propionate degraders
21	X_{ac}	ODE	Fluid	0.57682	0	[kg m^{-3}]	Acetate degraders
22	X_{h2}	ODE	Fluid	0.28774	0	[kg m^{-3}]	Hydrogen degraders
23	X_{i}	ODE	Fluid	18.6685	12	[kg m^{-3}]	Particulate inerts
24	S_{cat}	ODE	Fluid	3.3531E-39	71	[mol m^{-3}]	Soluble cations
25	S_{an}	ODE	Fluid	1.5293E-38	70	[mol m^{-3}]	Soluble anions
26	$S_{gas,h2}$	ODE	Gas	1.9096E-05	1.9096E-05	[kg m^{-3}]	Hydrogen gas
27	$S_{gas,ch4}$	ODE	Gas	1.5103	1.5103	[kg m^{-3}]	Methane gas
28	$S_{gas,co2}$	ODE	Gas	13.766	13.766	[mol m^{-3}]	Carbon dioxide gas

D.1.2 Algebraic variables

Table A2 – ADM1 algebraic variables

i	Symbol	Type	Loction	Initial	Inlet	Units	Description
0	S_{h2}	Algebraic	Fluid	4.9652E-07	n/a	[kg m^{-3}]	Disolved hydrogen gas (implicit)
1	S_{h+}	Algebraic	Fluid	5.3469E-05	n/a	[mol m^{-3}]	Hydrogen ions
2	S_{va-}	Algebraic	Fluid	0.023204	n/a	[kg m^{-3}]	Valeric acid
3	S_{bu-}	Algebraic	Fluid	0.031017	n/a	[kg m^{-3}]	Butyric acid
4	S_{pro-}	Algebraic	Fluid	0.043803	n/a	[kg m^{-3}]	Propionic acid
5	S_{ac-}	Algebraic	Fluid	0.50616	n/a	[kg m^{-3}]	Acetic acid
6	S_{hco3-}	Algebraic	Fluid	92.928	n/a	[kg m^{-3}]	Carbonic acid
7	S_{nh4+}	Algebraic	Fluid	0.0021958	n/a	[kg m^{-3}]	Nitric acid

***Note:** S_{h2} appears both as an ODE and algebraic variable. This variable may be solved with the ODE, or implicitly.

D.1.3 Derived variables

Table A3 – ADM1 derived variables

i	Symbol	Type	Loction	Initial	Inlet	Units	Description
0	$p_{gas,h2}$	Derived	Gas	n/a	n/a	[kg m^{-3}]	Hydrogen gas partial pressure
1	$p_{gas,ch4}$	Derived	Gas	n/a	n/a	[kg m^{-3}]	Methane partial pressure

2	$p_{gas,co2}$	Derived	Gas	n/a	n/a	[kg m^{-3}]	Carbon dioxide partial pressure
3	$p_{gas,tot}$	Derived	Gas	n/a	n/a	[kg m^{-3}]	Total gas pressure
4	q_{gas}	Derived	Gas	n/a	n/a	[kg m^{-3}]	Total gas flow rate
5	S_{co2}	Derived	Fluid	n/a	n/a	[mol m^{-3}]	Disolved carbon dioxide
6	S_{nh3}	Derived	Fluid	n/a	n/a	[mol m^{-3}]	Disolved ammonia
7	S_{oh-}	Derived	Fluid	n/a	n/a	[mol m^{-3}]	Disolved hydroxide
8	pH	Derived	Fluid	n/a	n/a	[kg m^{-3}]	pH

D.2 Coefficients

There are 109 coefficients in the variation of ADM1 implemented here. These can be broken down into the categories of:

- stoichiometric coefficients;
- kinetic rate coefficients;
- inhibition coefficients;
- yield coefficients; and
- other coefficients.

D.2.1 Stoichiometric coefficients

Table A4 – ADM1 stoichiometric Coefficients

Symbol	Value	Units	Description
C_{su}	31.3	[mol kg^{-1}]	Carbon content, monosaccharide
C_{aa}	30	[mol kg^{-1}]	Carbon content, amino acids
C_{fa}	21.7	[mol kg^{-1}]	Carbon content, fatty acids
C_{va}	24	[mol kg^{-1}]	Carbon content, valerate
C_{bu}	25	[mol kg^{-1}]	Carbon content, butyrate
C_{pro}	26.8	[mol kg^{-1}]	Carbon content, propionate
C_{ac}	31.3	[mol kg^{-1}]	Carbon content, acetate
C_{ch4}	15.6	[mol kg^{-1}]	Carbon content, methane
C_{sI}	30	[mol kg^{-1}]	Carbon content, soluble inerts

C_{xc}	30	[mol kg^{-1}]	Carbon content, carbon composites
C_{ch}	31.3	[mol kg^{-1}]	Carbon content, carbohydrates
C_{pr}	30	[mol kg^{-1}]	Carbon content, protein
C_{li}	22	[mol kg^{-1}]	Carbon content, lipids
C_{bac}	31.3	[mol kg^{-1}]	Carbon content, all microbial entities - i=17-23
C_{xI}	30	[mol kg^{-1}]	Carbon content, particulate inerts
N_{aa}	7	[mol kg^{-1}]	Nitrogen content, amino acids
N_{bac}	5.7143	[mol kg^{-1}]	Nitrogen content, all microbial entities - i=17-23
N_{I}	4.2857	[mol kg^{-1}]	Nitrogen content, particulate inerts
N_{xc}	2.6857	[mol kg^{-1}]	Nitrogen content, carbon composites
gpm_{ac}	0.064	[kg mol^{-1}]	Grams of COD yielded from a mole of acetate
gpm_{bu}	0.16	[kg mol^{-1}]	Grams of COD yielded from a mole of butyrate
gpm_{ch4}	0.064	[kg mol^{-1}]	Grams of COD yielded from a mole of methane
gpm_{h2}	0.016	[kg mol^{-1}]	Grams of COD yielded from a mole of hydrogen
gpm_{pro}	0.112	[kg mol^{-1}]	Grams of COD yielded from a mole of propionate
gpm_{va}	0.208	[kg mol^{-1}]	Grams of COD yielded from a mole of valerate

D.2.2 Kinetic rate coefficients

Table A5 – ADM1 kinetic rate coefficients

Symbol	Metric	Units	Description
$k_{dec,Xaa}$	2.315E-07	[s^{-1}]	First order decay rate of X_aa
$k_{dec,Xac}$	2.315E-07	[s^{-1}]	First order decay rate of X_ac
$k_{dec,Xc4}$	2.315E-07	[s^{-1}]	First order decay rate of X_c4
$k_{dec,Xfa}$	2.315E-07	[s^{-1}]	First order decay rate of X_fa
$k_{dec,Xh2}$	2.315E-07	[s^{-1}]	First order decay rate of X_h2
$k_{dec,Xpro}$	2.315E-07	[s^{-1}]	First order decay rate of X_pro
$k_{dec,Xsu}$	2.315E-07	[s^{-1}]	First order decay rate of X_su
k_{dis}	5.787E-06	[s^{-1}]	Rate of disintegration
$k_{hyd,ch}$	1.157E-04	[s^{-1}]	Rate of hydrolysis of carbohydrates
$k_{hyd,li}$	1.157E-04	[s^{-1}]	Rate of hydrolysis of lipids
$k_{hyd,pr}$	1.157E-04	[s^{-1}]	Rate of hydrolysis of proteins
$k_{m,aa}$	5.787E-04	[s^{-1}]	Maximum specific uptake of amino acids
$k_{m,ac}$	9.259E-05	[s^{-1}]	Maximum specific uptake of acetate
$k_{m,c4}$	2.315E-04	[s^{-1}]	Maximum specific uptake of valerate and butyrate
$k_{m,fa}$	6.944E-05	[s^{-1}]	Maximum specific uptake of fatty acids
$k_{m,h2}$	4.051E-04	[s^{-1}]	Maximum specific uptake of monosaccharide

$k_{m,pr}$	1.505E-04	[s^{-1}]	Maximum specific uptake of protein
$k_{m,su}$	3.472E-04	[s^{-1}]	Maximum specific uptake of monosaccharide
$k_{AB,ac}$	1E+10	[mol m^{-3}]	Base to acid coefficient, acetate
$k_{AB,bu}$	1E+10	[mol m^{-3}]	Base to acid coefficient, butyrate
$k_{AB,CO2}$	1E+10	[mol m^{-3}]	Base to acid coefficient, carbon dioxide
$k_{AB,IN}$	1E+10	[mol m^{-3}]	Base to acid coefficient, inorganic nitrogen
$k_{AB,pro}$	1E+10	[mol m^{-3}]	Base to acid coefficient, propionate
$k_{AB,va}$	1E+10	[mol m^{-3}]	Base to acid coefficient, valerate
$K_{a,ac}$	0.0174	[mol m^{-3}]	Acid-base equilibrium coefficient, acetate
$K_{a,bu}$	0.0151	[mol m^{-3}]	Acid-base equilibrium coefficient, butyrate
$K_{a,CO2}$	4.937E-04	[mol m^{-3}]	Acid-base equilibrium coefficient, carbon dioxide. ***Function of temperature, see below.***
$K_{a,IN}$	1.110E-06	[mol m^{-3}]	Acid-base equilibrium coefficient, inorganic nitrogen. ***Function of temperature, see below.***
$K_{a,pro}$	0.0132	[mol m^{-3}]	Acid-base equilibrium coefficient, propionate
$K_{a,va}$	0.0138	[mol m^{-3}]	Acid-base equilibrium coefficient, valerate
K_w	2.079E-08	[mol^2 m^{-6}]	Acid-base equilibrium coefficient, water. ***Function of temperature, see below.***
$K_{H,CH4}$	1.065E-05	[mol s^2 kg^{-1} m^{-2}]	Gas transfer coefficient (Henry's law), CH4. ***Function of temperature, see below.***
$K_{H,CO2}$	2.603E-04	[mol s^2 kg^{-1} m^{-2}]	Gas transfer coefficient (Henry's law), CO2. ***Function of temperature, see below.***
$K_{H,h2}$	7.288E-06	[mol s^2 kg^{-1} m^{-2}]	Gas transfer coefficient (Henry's law), H2. ***Function of temperature, see below.***
k_{La}	2.315E-03	[s^{-1}]	Overall gas-liquid mass transfer coefficient
k_P	5.787E-06	[m^4 s kg^{-1}]	Gas outlet pipe resistance coefficient
$K_{S,aa}$	0.3	[kg m^{-3}]	Half saturation value, amino acid uptake
$K_{S,ac}$	0.15	[kg m^{-3}]	Half saturation value, acetate uptake
$K_{S,c4}$	0.2	[kg m^{-3}]	Half saturation value, butyrate and valerate uptake
$K_{S,fa}$	0.4	[kg m^{-3}]	Half saturation value, fatty acid uptake
$K_{S,h2}$	7.00E-06	[kg m^{-3}]	Half saturation value, hydrogen uptake
$K_{S,pro}$	0.1	[kg m^{-3}]	Half saturation value, propionate uptake
$K_{S,su}$	0.5	[kg m^{-3}]	Half saturation value, sugar uptake

Coefficients that depend on temperature are:

$$k_{a,co_2} = 1000 \times 10^{-6.35} \exp\left[\frac{7646}{8.3145}\left(\frac{1}{T_{base}} + \frac{1}{T}\right)\right], \tag{A24}$$

$$k_{a,in} = 1000 \times 10^{-9.25} \exp\left[\frac{51965}{8.3145}\left(\frac{1}{T_{base}} + \frac{1}{T}\right)\right], \quad \text{(A25)}$$

$$K_{H,ch_4} = \frac{1.3 \times 10^{-3}}{101.325} \exp\left[\frac{-1.424 \times 10^{7}}{8314.5}\left(\frac{1}{T_{base}} + \frac{1}{T}\right)\right], \quad \text{(A26)}$$

$$K_{H,co_2} = \frac{0.034}{101.325} \exp\left[\frac{-1.941 \times 10^{7}}{8314.5}\left(\frac{1}{T_{base}} + \frac{1}{T}\right)\right], \quad \text{(A27)}$$

$$K_{H,h_2} = \frac{7.8 \times 10^{-4}}{101.325} \exp\left[\frac{-4.18 \times 10^{6}}{8314.5}\left(\frac{1}{T_{base}} + \frac{1}{T}\right)\right], \text{ and} \quad \text{(A28)}$$

$$k_w = 10^{-8} \exp\left[\frac{55900}{8.3145}\left(\frac{1}{T_{base}} + \frac{1}{T}\right)\right], \quad \text{(A29)}$$

where T_{base} is a base temperature, at 298.15 [K].

D.2.3 Inhibition coefficients

Table A6 – ADM1 inhibition Coefficients

Symbol	Value	Units	Description
$K_{I,IN}$	0.1	[mol/m3]	Inhibition parameter, inorganic nitrogen
$K_{I,nh3}$	1.8	[mol/m3]	Inhibition parameter, ammonium
$K_{I,h2,c4}$	1.0E-05	[kg/m3]	Inhibition parameter, valerate and butyrate
$K_{I,h2,fa}$	5.0E-06	[kg/m3]	Inhibition parameter, fatty acids
$K_{I,h2,pro}$	3.5E-06	[kg/m3]	Inhibition parameter, propionate
$pH_{LL,aa}$	4	[]	pH inhibition lower limit, amino acid uptake
$pH_{LL,ac}$	6	[]	pH inhibition lower limit, acetate uptake
$pH_{LL,h2}$	5	[]	pH inhibition lower limit, hydrogen uptake
$pH_{UL,aa}$	5.5	[]	pH inhibition upper limit, amino acid uptake
$pH_{UL,ac}$	7	[]	pH inhibition upper limit, acetate uptake
$pH_{UL,h2}$	6	[]	pH inhibition upper limit, hydrogen uptake

D.2.4 Yield coefficients

Table A7 – ADM1 yield coefficients

Symbol	Value	Units	Description
$f_{ac,aa}$	0.4	[]	Empirical yield, acetate from amino acids
$f_{ac,su}$	0.41	[]	Empirical yield, acetate from monosaccharides
$f_{bu,aa}$	0.26	[]	Empirical yield, butyrate from amino acids

$f_{bu,su}$	0.13	[]	Empirical yield, butyrate from monosaccharides
$f_{ch,xc}$	0.2	[]	Empirical yield, carbohydrates from composites
$f_{fa,li}$	0.95	[]	Empirical yield, fatty acids from lipids
$f_{h2,aa}$	0.06	[]	Empirical yield, hydrogen from amino acids
$f_{h2,su}$	0.19	[]	Empirical yield, hydrogen from monosaccharides
$f_{li,xc}$	0.3	[]	Empirical yield, lipids from composites
$f_{pr,xc}$	0.2	[]	Empirical yield, proteins from composites
$f_{pro,aa}$	0.05	[]	Empirical yield, propionate from amino acids
$f_{pro,su}$	0.27	[]	Empirical yield, propionate from monosaccharides
$f_{sI,xc}$	0.1	[]	Empirical yield, soluble inerts from composites
$f_{va,aa}$	0.23	[]	Empirical yield, valerate from amino acids
$f_{xI,xc}$	0.2	[]	Empirical yield, particulate inerts from composites
Y_{aa}	0.08	[]	Yield of biomass on amino acids
Y_{ac}	0.05	[]	Yield of biomass on acetate
Y_{c4}	0.06	[]	Yield of biomass on valerate and butyrate
Y_{fa}	0.06	[]	Yield of biomass on fatty acids
Y_{h2}	0.06	[]	Yield of biomass on hydrogen
Y_{pro}	0.04	[]	Yield of biomass on propionate
Y_{su}	0.1	[]	Yield of biomass on monosaccharides

D.2.5 Other coefficients

Table A8 – ADM1 other coefficients

Symbol	Value	Units	Description
P_{atm}	101325	[kg m^{-1} s^{-2}]	Atmospheric pressure
$p_{gas,h2o}$	5566.7745	[kg m^{-1} s^{-2}]	Water vapour partial pressure. ***Function of temperature, see below.***
q	1.928E-03	[m^3/s]	Digester liquid flow rate
R	8.3145	[kg m^2 s^{-2}]	Gas constant
T	308.15	[K]	Temperature
V_{gas}	300	[m^3]	Gas volume
V_{liq}	3400	[m^3]	Liquid volume

D.3 Inhibition

There are five types of inhibition modelled by ADM1. These are:

- hydrogen inhibition;
- nitrogen inhibition;

- ammonium inhibition;
- pH inhibition; and
- competitive uptake inhibition.

Inhibition is modelled using a function that gives a value between 0 and 1.

D.3.1 Hydrogen inhibition

High concentrations of free hydrogen are not tolerated by some of the organisms in an anaerobic digester, and consequently their respective processes suffer. Hydrogen inhibition is modelled as non-competitive inhibition according to:

$$I_{h_2} = \frac{K_I}{K_I + S_{h_2}}. \tag{A30}$$

Hydrogen inhibition affects:

- the uptake of valerate and butyrate ($I_{h2,c4}$ – processes 7 and 8);
- the uptake of long chain fatty acids ($I_{h2,fa}$ – process 6); and
- the uptake of propionate ($I_{h2,pro}$ – process 9).

D.3.2 Nitrogen inhibition

Nitrogen inhibition is modelled as a secondary-substrate inhibition, according to:

$$I_{in} = \frac{S_{in}}{S_{in} + K_{I,in}}. \tag{A31}$$

Nitrogen inhibition affects all uptake processes (index numbers 4 to 11).

D.3.3 Ammonium inhibition

Ammonium inhibition is modelled as a non-competitive inhibition, according to:

$$I_{nh_3} = \frac{K_{I,nh_3}}{K_{I,nh_3} + S_{nh_3}}. \tag{A32}$$

Ammonium inhibition affects the uptake of acetate (process 10).

D.3.4 pH inhibition

There are two kinds of pH inhibition:

- lower inhibition, where only low pH values cause inhibition; and
- upper and lower inhibition, where both high and low pH values cause inhibition.

There are several lower inhibition functions proposed, detailed by Rosen et al. (2006). All proposed models are implemented in the ADM1 code, and available as an option to the user.

In this ADM1 variation, pH inhibition affects three processes, all of which are modelled as lower inhibition functions:

- uptake of amino acids and sugars ($I_{pH,aa}$ – processes 4 and 5);
- uptake of acetate ($I_{pH,ac}$ – process 10); and
- uptake of hydrogen ($I_{pH,h2}$ – process 11).

The empirical switch functions for these are:

$$I_{pH} = \begin{cases} \exp\left[-3\left(\frac{pH-pH_{ul}}{pH_{ul}-pH_{ll}}\right)^2\right] & \text{if } pH < pH_{ul} \\ 1 & \text{if } pH \geq pH_{ul} \end{cases} \tag{A33}$$

where pH_{ul} and pH_{ll} are the upper and lower pH limits, respectively.

D.3.5 Competitive uptake inhibition

The uptake of valerate and butyrate are performed by the same microorganisms. Therefore, uptake of one of these will inhibit uptake of the other. This is modelled using competitive uptake inhibition, according to:

$$I_{va/bu} = \frac{S_{va}}{S_{va} + S_{bu}}, \tag{A34}$$

which acts on the uptake of valerate (process 7); and

$$I_{va/bu} = \frac{S_{bu}}{S_{bu} + S_{va}}, \tag{A35}$$

which acts on the uptake of butyrate (process 8).

D.4 Petersen matrix

Table A9 – ADM1 full Petersen Matrix, cols 0-9

Component →		i	0	1	2	3	4	5	6	7	8	9	Rate
j	Process ↓		S_{su}	S_{aa}	S_{fa}	S_{va}	S_{bu}	S_{pro}	S_{ac}	S_{h2}	S_{ch4}	S_{IC}	
0	Disintgration											$-\sum_{(i\neq 9)} C_i\nu_{i,0}$	$k_{dis}X_c$
1	Hydrolysis of carbohydrates		1									$-\sum_{(i\neq 9)} C_i\nu_{i,1}$	$k_{dis}X_c$
2	Hydrolysis of proteins			1								$-\sum_{(i\neq 9)} C_i\nu_{i,2}$	$k_{hyd,pr}X_{pr}$
3	Hydrolysis of lipids		$1-f_{fa,li}$		$f_{fa,li}$							$-\sum_{(i\neq 9)} C_i\nu_{i,3}$	$k_{hyd,li}X_{li}$
4	Uptake of sugars		-1				$(1-Y_{su})\,f_{bu,su}$	$(1-Y_{su})\,f_{pro,su}$	$(1-Y_{su})\,f_{ac,su}$	$(1-Y_{su})\,f_{h_2,su}$		$-\sum_{(i\neq 9)} C_i\nu_{i,4}$	$\frac{k_{m,su}S_{su}X_{su}}{K_{s,su}+S_{su}}I_{in}I_{ph,aa}$
5	Uptake of amino acids			-1		$(1-Y_{aa})\,f_{va,aa}$	$(1-Y_{aa})\,f_{bu,aa}$	$(1-Y_{aa})\,f_{pro,aa}$	$(1-Y_{aa})\,f_{ac,aa}$	$(1-Y_{aa})\,f_{h_2,aa}$		$-\sum_{(i\neq 9)} C_i\nu_{i,5}$	$\frac{k_{m,aa}S_{aa}X_{aa}}{K_{s,aa}+S_{aa}}I_{in}I_{ph,aa}$
6	Uptake of LCFA				-1				$(1-Y_{fa})\,0.7$	$(1-Y_{fa})\,0.3$		$-\sum_{(i\neq 9)} C_i\nu_{i,6}$	$\frac{k_{m,fa}S_{fa}X_{fa}}{K_{s,fa}+S_{fa}}I_{in}I_{ph,aa}I_{h2,fa}$
7	Uptake of valerate					-1		$(1-Y_{c_4})\,0.54$	$(1-Y_{c_4})\,0.31$	$(1-Y_{c_4})\,0.15$		$-\sum_{(i\neq 9)} C_i\nu_{i,7}$	$\frac{k_{m,c_4}S_{va}X_{c_4}}{K_{s,c_4}+S_{va}}I_{in}I_{ph,aa}I_{h2,c4}I_{va/bu}$
8	Uptake of butyrate						-1		$(1-Y_{c_4})\,0.8$	$(1-Y_{c_4})\,0.2$		$-\sum_{(i\neq 9)} C_i\nu_{i,8}$	$\frac{k_{m,c_4}S_{bu}X_{c_4}}{K_{s,c_4}+S_{bu}}I_{in}I_{ph,aa}I_{h_2,c_4}I_{bu/va}$
9	Uptake of propionate							-1	$(1-Y_{pro})\,0.57$	$(1-Y_{pro})\,0.43$		$-\sum_{(i\neq 9)} C_i\nu_{i,9}$	$\frac{k_{m,pr}S_{pro}X_{pro}}{K_{s,pro}+S_{pro}}I_{in}I_{ph,aa}I_{h_2,pro}$
10	Uptake of acetate								-1		$1-Y_{ac}$	$-\sum_{(i\neq 9)} C_i\nu_{i,10}$	$\frac{k_{m,ac}S_{ac}X_{ac}}{K_{s,ac}+S_{ac}}I_{in}I_{ph,ac}I_{nh_3}$
11	Uptake of hydrogen									-1	$1-Y_{h_2}$	$-\sum_{(i\neq 9)} C_i\nu_{i,11}$	$\frac{k_{m,h_2}S_{h_2}X_{h_2}}{K_{s,h_2}+S_{h_2}}I_{in}I_{ph,h_2}$
12	Decay of Xsu											$-\sum_{(i\neq 9)} C_i\nu_{i,12}$	$k_{dec,X_{su}}X_{su}$
13	Decay of Xaa											$-\sum_{(i\neq 9)} C_i\nu_{i,13}$	$k_{dec,X_{aa}}X_{aa}$
14	Decay of Xfa											$-\sum_{(i\neq 9)} C_i\nu_{i,14}$	$k_{dec,X_{fa}}X_{fa}$
15	Decay of Xc4											$-\sum_{(i\neq 9)} C_i\nu_{i,15}$	$k_{dec,X_{c_4}}X_{c_4}$
16	Decay of Xpro											$-\sum_{(i\neq 9)} C_i\nu_{i,16}$	$k_{dec,X_{pro}}X_{pro}$
17	Decay of Xac											$-\sum_{(i\neq 9)} C_i\nu_{i,17}$	$k_{dec,X_{ac}}X_{ac}$
18	Decay of Xh2											$-\sum_{(i\neq 9)} C_i\nu_{i,18}$	$k_{dec,X_{h_2}}X_{h_2}$
			Monosaccharides [kgCOD m-3]	Amino acids [kgCOD m-3]	Long chain fatty acids [kgCOD m-3]	Total valerate [kgCOD m-3]	Total butyrate [kgCOD m-3]	Total propionate [kgCOD m-3]	Total acetate [kgCOD m-3]	Total hydrogen gas [kgCOD m-3]	Methane gas [kgCOD m-3]	Inorganic carbon [mole m-3]	

Table A10 – ADM1 full Petersen Matrix, cols 10-23

	Component → i	10	11	12	13	14	15	16	17	18	19	20	21	22	23	Rate
j	Process ↓	S_{in}	S_i	X_c	X_{ch}	X_{pr}	X_{li}	X_{su}	X_{aa}	X_{fa}	X_{c4}	X_{pro}	X_{ac}	X_{h2}	X_i	
0	Disintgration		$f_{si,xc}$	-1	$f_{ch,xc}$	$f_{pr,xc}$	$f_{li,xc}$								$f_{xi,xc}$	$k_{dis}X_c$
1	Hydrolysis of carbohydrates				-1											$k_{dis}X_c$
2	Hydrolysis of proteins					-1										$k_{hyd,pr}X_{pr}$
3	Hydrolysis of lipids						-1									$k_{hyd,li}X_{li}$
4	Uptake of sugars	$-Y_{su}N_{bac}$						Y_{su}								$\frac{k_{m,su}S_{su}X_{su}}{K_{s,su}+S_{su}}I_{in}I_{ph,aa}$
5	Uptake of amino acids	$N_{aa}-Y_{aa}N_{bac}$							Y_{aa}							$\frac{k_{m,aa}S_{aa}X_{aa}}{K_{s,aa}+S_{aa}}I_{in}I_{ph,aa}$
6	Uptake of LCFA	$-Y_{fa}N_{bac}$								Y_{fa}						$\frac{k_{m,fa}S_{fa}X_{fa}}{K_{s,fa}+S_{fa}}I_{in}I_{ph,aa}I_{h2,fa}$
7	Uptake of valerate	$-Y_{c_4}N_{bac}$									Y_{c_4}					$\frac{k_{m,c_4}S_{va}X_{c_4}}{K_{s,c_4}+S_{va}}I_{in}I_{ph,aa}I_{h2,c4}I_{va/bu}$
8	Uptake of butyrate	$-Y_{c_4}N_{bac}$									Y_{c_4}					$\frac{k_{m,c_4}S_{bu}X_{c_4}}{K_{s,c_4}+S_{bu}}I_{in}I_{ph,aa}I_{h2,c4}I_{bu/va}$
9	Uptake of propionate	$-Y_{pro}N_{bac}$										Y_{pro}				$\frac{k_{m,pr}S_{pro}X_{pro}}{K_{s,pro}+S_{pro}}I_{in}I_{ph,aa}I_{h2,pro}$
10	Uptake of acetate	$-Y_{ac}N_{bac}$											Y_{ac}			$\frac{k_{m,ac}S_{ac}X_{ac}}{K_{s,ac}+S_{ac}}I_{in}I_{ph,ac}I_{nh_3}$
11	Uptake of hydrogen	$-Y_{h_2}N_{bac}$												Y_{h_2}		$\frac{k_{m,h_2}S_{h_2}X_{h_2}}{K_{s,h_2}+S_{h_2}}I_{in}I_{ph,h2}$
12	Decay of Xsu			1				-1								$k_{dec,X_{su}}X_{su}$
13	Decay of Xaa			1					-1							$k_{dec,X_{aa}}X_{aa}$
14	Decay of Xfa			1						-1						$k_{dec,X_{fa}}X_{fa}$
15	Decay of Xc4			1							-1					$k_{dec,X_{c_4}}X_{c_4}$
16	Decay of Xpro			1								-1				$k_{dec,X_{pro}}X_{pro}$
17	Decay of Xac			1									-1			$k_{dec,X_{ac}}X_{ac}$
18	Decay of Xh2			1										-1		$k_{dec,X_{h_2}}X_{h_2}$
		Inorganic nitrogen [mole m-3]	Soluble inerts [kgCOD m-3]	Composites [kgCOD m-3]	Carbohydrates [kgCOD m-3]	Proteins [kgCOD m-3]	Lipids [kgCOD m-3]	Sugar degraders [kgCOD m-3]	Amino acid degraders [kgCOD m-3]	LCFA degraders [kgCOD m-3]	Valerate and butyrate degraders [kgCOD m-3]	Propionate degraders [kgCOD m-3]	Acetate degraders [kgCOD m-3]	Hydrogen degraders [kgCOD m-3]	Particulate inerts [kgCOD m-3]	

Appendix E CFD theory details

This section provides additional details behind the CFD theory, including ADM-MDA.

E.1 Governing equations expanded in Cartesian coordinates

These are the continuity, momentum and energy equations in Cartesian coordinates, based on the assumptions that:

- the fluid is Newtonian – constant viscosity;
- the fluid is incompressible – constant density;
- the fluid is subject to a Boussinesq buoyancy force; and
- inertial reference frame.

E.1.1 Continuity

$$\frac{\partial U_x}{\partial x}+\frac{\partial U_y}{\partial y}+\frac{\partial U_z}{\partial z}=0 \tag{A36}$$

E.1.2 Momentum

The x-component is:

$$\begin{gathered}\rho\frac{\partial U_x}{\partial t}+\rho U_x\frac{\partial U_x}{\partial x}+\rho U_y\frac{\partial U_x}{\partial y}+\rho U_z\frac{\partial U_x}{\partial z}=-\frac{\partial p}{\partial x}+\\ \mu\left[\frac{\partial^2 U_x}{\partial x^2}+\frac{\partial^2 U_x}{\partial y^2}+\frac{\partial^2 U_x}{\partial z^2}\right]-\rho_{ref}\beta\left(T-T_{ref}\right)\end{gathered}, \tag{A37}$$

the y-component is:

$$\begin{gathered}\rho\frac{\partial U_y}{\partial t}+\rho U_x\frac{\partial U_y}{\partial x}+\rho U_y\frac{\partial U_y}{\partial y}+\rho U_z\frac{\partial U_y}{\partial z}=-\frac{\partial p}{\partial y}+\\ \mu\left[\frac{\partial^2 U_y}{\partial x^2}+\frac{\partial^2 U_y}{\partial y^2}+\frac{\partial^2 U_y}{\partial z^2}\right]-\rho_{ref}\beta\left(T-T_{ref}\right)\end{gathered}, \text{ and} \tag{A38}$$

the z-component is:

$$\rho\frac{\partial U_x}{\partial t}+\rho U_x\frac{\partial U_z}{\partial x}+\rho U_y\frac{\partial U_z}{\partial y}+\rho U_z\frac{\partial U_z}{\partial z}=-\frac{\partial p}{\partial z}+$$
$$\mu\left[\frac{\partial^2 U_z}{\partial x^2}+\frac{\partial^2 U_z}{\partial y^2}+\frac{\partial^2 U_z}{\partial z^2}\right]-\rho_{ref}\beta\left(T-T_{ref}\right). \quad \text{(A39)}$$

E.1.3 Energy equation

$$\frac{\partial T}{\partial t}+\frac{\partial U_x T}{\partial x}+\frac{\partial U_y T}{\partial y}+\frac{\partial U_z T}{\partial z}-\alpha\left(\frac{\partial^2 T}{\partial x^2}+\frac{\partial^2 T}{\partial y^2}+\frac{\partial^2 T}{\partial z^2}\right)=0. \quad \text{(A40)}$$

E.1.4 Scalar transport equation

This equation applies to ADM-MDA.

$$\frac{\partial S_{var}}{\partial t}+\frac{\partial U_x S_{var}}{\partial x}+\frac{\partial U_y S_{var}}{\partial y}+\frac{\partial U_z S_{var}}{\partial z}-\Gamma_{var}\left(\frac{\partial^2 S_{var}}{\partial x^2}+\frac{\partial^2 S_{var}}{\partial y^2}+\frac{\partial^2 S_{var}}{\partial z^2}\right)=0. \quad \text{(A41)}$$

Appendix F Algorithm-switching details

CRAFTS originally implemented algorithm-switching as a response to anticipated timescale differences between the biochemistry solution and the flow solution. However, in practice, the timescales do not differ significantly, rendering this design unnecessary, and it was removed from the model. Algorithm-switching remains a viable option for modelling phenomena with significant timescale differences, or with distinct changes in behaviour. The original details and rationale behind algorithm-switching are presented here.

Anaerobic digesters experience two disparate phenomena: the periodically changing flow conditions, and the biochemical reactions. A significant modelling challenge comes from the timescale differences between these two phenomena. Fluid flow happens quickly, as it is typically modelled using timesteps measured in seconds, or fractions of a second; on the other hand, biochemical reactions are slower, as ADM1 is typically modelled using timesteps measured in minutes or hours. It is impractical to reduce the biochemical model timestep to coincide with that of the fluid dynamic timestep, as this increases the computational effort by a factor of several thousand.

One solution is to recognize that the fluid flow in an anaerobic digester is most often steady-state. Although the digester periodically changes its control state, such as by switching on a heater, or during fluid injection, there is a steady-state solution for each control state. The transient flow conditions only exist as the digester transitions between these control states. When the flow is steady-state, the solver can focus exclusively on the biochemical reactions using their preferred timestep. Additionally, if the transient states occur sufficiently quickly, then the solver can ignore the biochemical reactions and focus exclusively on the fluid flow during transient flow conditions.

Formally applying this to the model, ADM-MDA distinguishes between *transient* flow and *steady-state* flow. During transient flow, ADM-MDA only updates the flow and temperature field, and performs non-reacting scalar transport of the species. Once the flow field stabilises, ADM-MDA switches to steady-state flow. In this mode, the flow field is not updated, and the biochemical reactions are active, as shown in Table A11.

Table A11: Transient and steady-state solver components

	Transient	**Steady-state**
Flow solver	✓	
Temperature solver	✓	✓
Species transport	✓	✓
Reaction solver		✓

ADM-MDA can make these simplifications if it assumes:

1. the biochemical reactions do not affect the flow field; and
2. the time it takes for the flow field to reach a steady state after a change of control is negligible compared to the timescale of the biochemical reactions.

The first assumption is valid for reactions that have relatively small changes in enthalpy, and do not affect the temperature distribution significantly. This is true in anaerobic digesters. The second assumption can be verified by observing the model output. If either of these are not valid, alternative methods must be employed to handle transient flow and biochemical reactions simultaneously.

The transition between transient and steady-state flow leads to a set of requirements. ADM-MDA must be able to:

- identify steady-state and transient conditions; and

- switch between different algorithms appropriate to the current conditions.

This can be achieved in a general way if *algorithm-switching* is included as a feature of the process control strategy ADM-MDA uses. Algorithm-switching is the ability of a numerical solver to change the algorithm it uses in response to the evolving simulation conditions.

Appendix G Coupled reaction model solver algorithms

A considerable amount of effort went into developing a suitable algorithm, during which time several coupling strategies were attempted, including:

- a full-field ODE solver;
- a volume-isolation ODE solver;
- a volume-isolation PDE solver; and
- a full-field block coupled PDE solver.

G.1 Full-field ODE solver

Finite volume discretization breaks down the reactor geometry into a set of contiguous control volumes. Each variable has a different value at each control volume. From one perspective, the control volumes can be treated as having their own independent set of variables. These can all be added to one large ODE set, where the governing ODEs are the relationships between control volumes, as determined by the discretization process. Once this is accomplished, the system can be solved using conventional DAE methods. The algebraic routines are performed during the evaluation of the ddt and Jacobian.

This strategy is equivalent to the existing PDAE solution method, method of lines. However, this method does not allow for parallelisation, leading to slow computational time, particularly for larger problems. Furthermore, the framework of OpenFOAM requires some complicated modifications in order to allow finite volume method discretization to occur at arbitrary positions in time, something that an ODE-based solver requires. The solution to this problem is to omit the discretization from the ODE component of the solver. The result is a volume-isolation ODE solver.

G.2 Volume-isolation ODE solver

This method is most similar to existing reaction solvers implemented in OpenFOAM, and is presented as a flowchart in Figure A4.

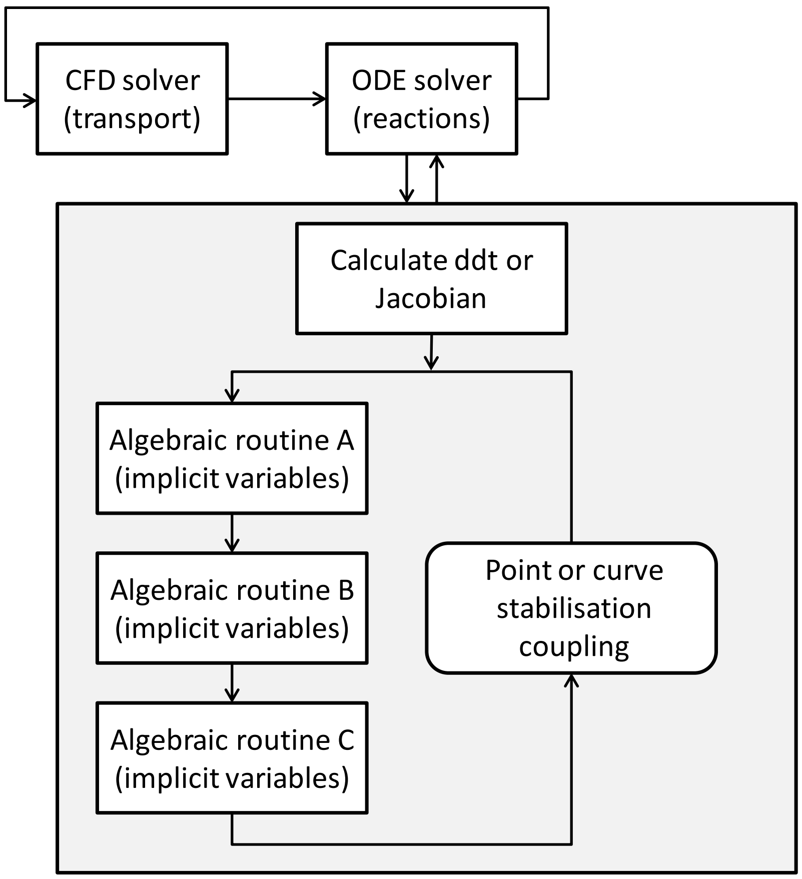

CFD solver (transport)
ODE solver (reactions)
Calculate ddt or Jacobian
Algebraic routine A (implicit variables)
Algebraic routine B (implicit variables)
Algebraic routine C (implicit variables)
Point or curve stabilisation coupling

Figure A4: Volume-isolation ODE solver flowchart

In this method, the species are transported first, with the reaction source term, r_{var} set to zero, according to:

$$\frac{\partial S_{var}}{\partial t} + \nabla \bullet (\mathbf{U}S_{var}) - \nabla \bullet \Gamma_{var} \nabla S_{var} = 0. \qquad \text{(A42)}$$

Without the reaction term, these equations are no longer coupled and can be solved individually using the finite volume method. The method then solves the reactions using an ODE solver on each control volume individually. The solver performs the algebraic routines within the ODE solver's ddt and

Jacobian routines, using point or curve stabilisation coupling. This solver overcomes the implementation problems with discretization, and can be parallelized. However, since this method isolates species transport from their reactions, it omits any non-linear effects between the two phenomena. More specifically, the model does not transport any quantity that may have been produced during the timestep, and does transport quantities that would have been consumed. This shortcoming imposes a limitation on the *Courant number*, which is given by:

$$C_o = \frac{\bar{u}\Delta t}{\Delta l}, \tag{A43}$$

where:

- $\bar{u}$ is the mean velocity in the control volume;
- Δt is the timestep; and
- Δl is a characteristic length of the control volume.

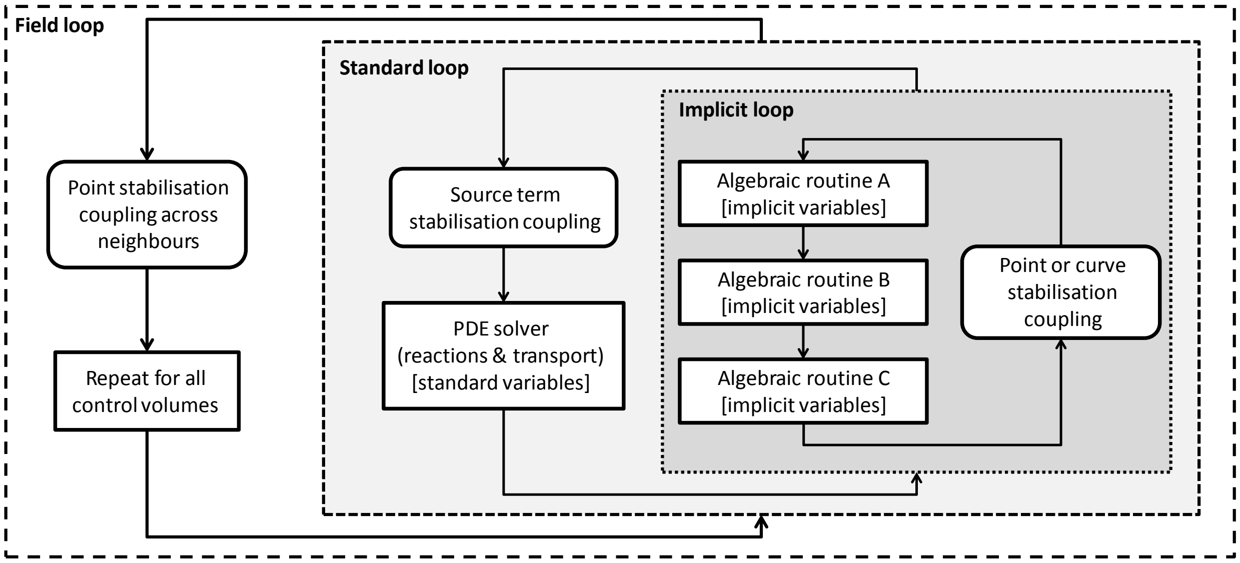
The Courant number is the timestep divided by the residence time in a control volume. A Courant number of unity approximately indicates that the contents of the control volume will completely transport out of it in the next timestep. The Courant number should generally be much less than one for a valid solution. For a given mesh geometry, this acts as a restriction on the maximum timestep size. For ADM-MDA whose biochemical reactions have long timescales, this is an unacceptable limitation. To avoid this problem, the species transport and biochemical reactions must be solved simultaneously.

G.3 Volume-isolation PDE solver

ADM-MDA is a large set of coupled PDEs. Recall that discretization of coupled PDEs yields a set of matrix equations, as shown in Equation (12), repeated here:

$$\begin{bmatrix} a_p^{y_0 y_0} & a_p^{y_0 y_1} & a_p^{y_0 y_2} \\ a_p^{y_1 y_0} & a_p^{y_1 y_1} & a_p^{y_1 y_2} \\ a_p^{y_2 y_0} & a_p^{y_2 y_1} & a_p^{y_2 y_2} \end{bmatrix} \begin{Bmatrix} y_{0_p} \\ y_{1_p} \\ y_{2_p} \end{Bmatrix} + \sum_i \begin{bmatrix} a_i^{y_0} & 0 & 0 \\ 0 & a_i^{y_1} & 0 \\ 0 & 0 & a_i^{y_2} \end{bmatrix} \begin{Bmatrix} y_{0_i} \\ y_{1_i} \\ y_{2_i} \end{Bmatrix} = \begin{bmatrix} b_p^{y_0} \\ b_p^{y_1} \\ b_p^{y_2} \end{bmatrix}. \tag{12}$$

Rather than massing these equations together and forming a block matrix, which requires a complex block matrix solver, these equations can be solved individually. To handle the neighbour terms, y_{n_i}, the solver uses point stabilisation coupling. The solver runs the algebraic routines after this, and absorbs the results back into the PDE system using source term stabilisation coupling. The flowchart for this process is shown in Figure A5.



Field loop
Standard loop
Implicit loop
Point stabilisation coupling across neighbours
Repeat for all control volumes
Source term stabilisation coupling
PDE solver (reactions & transport) [standard variables]
Algebraic routine A [implicit variables]
Algebraic routine B [implicit variables]
Algebraic routine C [implicit variables]
Point or curve stabilisation coupling

Figure A5: Volume-isolation PDE solver flowchart

Although theoretically sound, this solver proves to be numerically impractical. The field loop is difficult to converge, requiring exceptionally small relaxation factors and an unacceptably large number of iterations. The solution to this problem is to eliminate the field loop, which requires a block matrix solver.

G.4 Full field block-coupled PDE solver

The solver method that succeeds solves the transport and biochemical reactions across the entire fluid domain simultaneously. It functions identically to the volume-isolation PDE solver, except there is no need for a field loop, as shown in Figure A6.

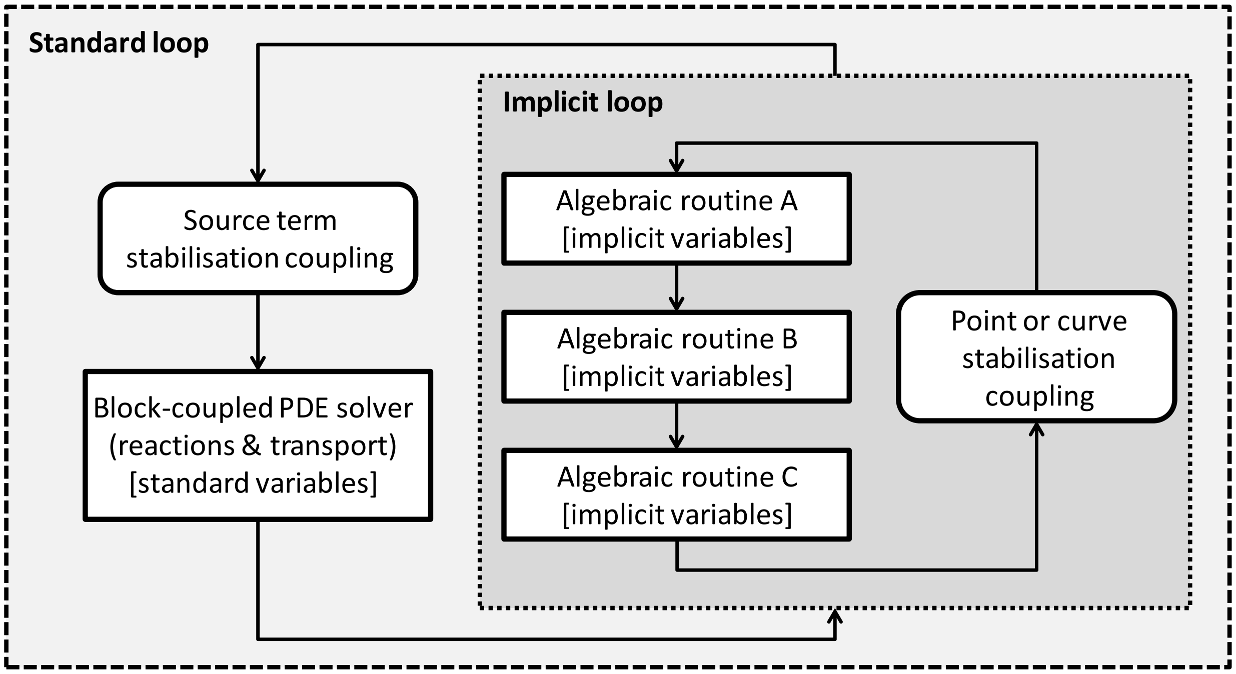

Standard loop
Implicit loop
Source term stabilisation coupling
Block-coupled PDE solver (reactions & transport) [standard variables]
Algebraic routine A [implicit variables]
Algebraic routine B [implicit variables]
Algebraic routine C [implicit variables]
Point or curve stabilisation coupling

Figure A6: Full-field block-coupled PDE solver

This solver requires the use of a general block matrix solver. A block matrix solver was recently released for a fork of OpenFOAM (Clifford et al., 2009; Kissling et al., 2010). ADM-MDA uses this block matrix solver for the species transport and reaction kinetics.

Appendix H equationReader extension

equationReader has been publically released to the *OpenFOAM-extend* project.
This section has also been published on the OpenFOAM wiki (openfoamwiki.net) and is therefore in the public domain.

H.1 What is it?

equationReader is an extension to OpenFOAM that allows you to work with user-entered equations. For example:

```
U.x         "sin(pi_ * t / 4)";
U.y         "rho * nut / L";
U.z         0;
nu          nu [0 2 -1 0 0 0 0] "1.2 + 3 * alpha^sin(pi_/6)";
aScalar     "nu / max(5, alpha)";
alpha       1.3;
```

H.2 Features

- **Works with most fields** - Now works with single elements, fields, DimensionedFields and GeometricFields;
- **Works with most Types** - Now works with scalars, vectors, and all kinds of tensors;
- **Flexible data sources** - In addition to these types, equations can also lookup values from dictionaries, and you can create anactiveVariable that derives its value on-the-fly.
- **Order of operations** - it is fully compliant with the conventional order of operations to an arbitrary parenthesis depth;
- **Equation dependency tracking** - equations can depend on one another to an arbitrary hierarchy depth;
- **Circular-reference detection** - it will halt computations when a circular reference is detected;
- **On-the-fly equation mapping** - it will automatically perform substitution on other equations when they are needed, even if they aren't specifically called for in the solver; and
- **Dimension-checking** - fully utilizes OpenFOAM's built-in dimension-checking, or you can force the outcome to a specific dimension-set to quickly disable it (if you are lazy).

Limitations: Although **equationReader** works with all Types, at its core, it is just a scalar calculator with dimensions. Therefore, you can't use vector operators, let alone tensor operators. Each equation must be expressed in scalar components.

H.3 Why would you need this?

Let the user define their own equations - this makes your application **more user-friendly**, and **more flexible**. But don't reinvent the wheel - if you are only working with *boundary conditions* or *initial conditions*, Bernhard's swak4Foam would probably be more suitable.

H.4 Update Info

- *2010-07-21*: Initial release
- *2010-08-05*: Bug-fix - differentiated versions for OpenFOAM-1.5.x/1.5-dev and OpenFOAM-1.6.x+
- *2010-08-12*: Major upgrade
    - Introducing `IOEquationReader` - EquationReader is now an `IOobject`. This enables automatic output
    - Added support for `scalarList` data sources - including `scalarField`, `volScalarField`, etc.
    - Removed the need for pointers for data sources
    - Cleaned up available functions
- *2010-10-16*: Bug fixes and added full support for fields
- *2011-04-06*: Major upgrade
    - Efficiency improvement - pointer functions have been implemented to increase computation speed by an order of magnitude (at least).
    - Improved dimension-checking on all functions.

    - Added a fieldEvaluate function for active equations whose output is to a scalar field.
    - Bug fix to get it working with 1.6-ext and higher.
- *2011-09-13:* Major upgrade
    - ***Now a stand-alone library***.
    - Now works with ***vectors and tensors***:
        - scalar;
        - vector;
        - tensor;
        - diagTensor;
        - symmTensor; and
        - sphericalTensor.
    - ***Now works with GeometricFields***
    - Dimension checking is now performed separately, improving efficiency of field and GeometricField calculations.
    - Interface changes:
        - Add data functions reorganized / changed.
        - Evaluate functions reorganized / changed.
        - Update functions removed.
- *2011-09-25:* ***Version 0.5.0***
    - Improved treatment of fields - ***now approximately 10x faster***.
    - Introduced version numbers to keep track of changes.

H.5 Efficiency

H.5.1 How fast is equationReader?

The most recent version of **equationReader** (*Version 0.5.0*, released September 25th, 2011) handles fields roughly 10x faster than the previous version. Overall, **equationReader** now takes approximately 5.87 times longer than a hard-coded solution when handling `GeometricFields`. That's for a simple equation. For more complex equations, **equationReader'**s performance improves.

Straight up `scalars` are still much slower. I haven't benchmarked the latest version, but previous versions were coming in at around 300 x slower.

H.5.2 Will it get faster?

Yes! The next plan is to have **equationReader** *compile* your equations at runtime. In theory, they will execute as fast as a hard-coded solution, less a small amount of overhead with the function call.

H.5.3 Parsing and evaluating

There is a difference between *parsing* and *evaluating*. When the equation is first read, it is a *human-readable* string expression. **equationReader** translates the *human-readable* form into an *operation list*. This is *parsing*. To calculate the result, **equationReader** does a `forAll(operations, i)`. This is *evaluating*.

Parsing happens only once, and is slow. *Evaluating* happens at every cell index, at every timestep (or however you've used it), and it is fast.

H.6 Installation

If you are running precompiled binaries, first ensure that you can compile your copy of OpenFOAM.

1. Download it from here, or copy and paste the link below:

   http://openfoam-extend.svn.sourceforge.net/viewvc/openfoam-extend/trunk/Breeder_1.6/libraries/equationReaderExtension/?view=tar

2. Unzip and unpack the files to where they need to be:

   a. If you trust me, do the "auto-unpack" commands (just open a terminal window, browse to the directory containing your download, and paste these lines in):

   ```
   tar --transform='s,equationReaderExtension,'$WM_PROJECT_DIR',' \
   -x -v -z -P -f openfoam-extend-equationReaderExtension.tar.gz
   ```

   b. If you want to see what's going on, just open the archive. It will be obvious where everything needs to go.

3. **If you have OpenFOAM 1.7.x or earlier** edit the file

   `src/equationReader/include/versionSpecific.H`:

   Comment out the second line:

   ```
   //#define ThisIsFoamVersion2
   ```

4. Compile the new library and demo project:

   ```
   cd $WM_PROJECT_DIR/src/equationReader
   wmake libso
   cd $WM_PROJECT_DIR/applications/solvers/equationReader/equationReaderDemo
   wmake
   ```

equationReader is now installed.

H.7 Testing

To test the installation, copy the new `tutorials/equationReader/` directory to your run directory, and run `equationReaderDemo`.

H.8 Using equationReader

H.8.1 Dictionary syntax

Any of these formats are acceptable to **equationReader**:

H.8.1.1 Standard equation

```
keyword     "equation";
keyword     scalar;
```

e.g.:

```
endTime     "2*pi_/360*60";
gamma       1.58e-6;
```

The **standard equation** format performs dimension checking for every operation. Use this if you want OpenFOAM to be strict about the dimensions you use. This has may unexpected consequences. For example:

- `sin(time)` is wrong because you can't have dimensions in any transcendental functions; and
- `max(deltaT, SMALL_)` is wrong because `SMALL` is dimensionless.

If this is too troublesome, you can also use:

H.8.1.2 Dimensioned equation

```
keyword     [dimensionSet] "equation";
keyword     [dimensionSet] scalar;
keyword     ignoredWord [dimensionSet] "equation";
keyword     ignoredWord [dimensionSet] scalar;
```

e.g.:

```
nu      [0 2 -1 0 0 0 0] "1 / (1e-5 + 2.3/4000 + SMALL_)";
rho     [1 -3 0 0 0 0 0] 1.235;
delta   delta [0 1 0 0 0 0 0] "sin(pi_ * t)";
alhpa   alpha [0 1 0 0 0 0 0] 3.2;
```

The **dimensioned equation** format disables dimension checking, and forces the final result to a given *dimensionSet*. Also note the optional *ignoredWord* - this allows **equationReader** to be compatible with *dimensionedScalar* formats.

H.8.2 Equation syntax

equationReader uses the conventional order of operations **BEDMAS**, then left to right:

- **B**rackets (and functions);
- **E**xponents;
- **DM** - division and multiplication; and
- **AS** - addition and subtraction.

It's just like Excel, except exponents 'a^b' don't work - use 'pow(a,b)' instead.

- you can use any amount of whitespace you want (use a backslash for a line break);
- multiplication is `*`, for example `2*3` is `6`;
- there is no implied multiplication - you must explicitly use `*`. For example:

  `2 sin(theta)` **INCORRECT**

  `2 * sin(theta)` **CORRECT**

  and

  `2(3 + 4)` **INCORRECT**

  `2 * (3 + 4)` **CORRECT**

H.8.2.1 Mathematical constants

equationReader recognizes all the mathematical constants I could find in the OpenFOAM library. To specify a mathematical constant, append the regular OpenFOAM format with an underscore '_'. The available constants are:

- `e_` (Euler's number);
- `pi_`;
- `twoPi_`;
- `piByTwo_`;
- `GREAT_`;

- `VGREAT_`;
- `ROOTVGREAT_`;
- `SMALL_`;
- `VSMALL_`; and
- `ROOTSMALL_`.

H.8.2.2 Functions

Functions available to **equationReader** are:

- pow(x)
- sign(x)
- pos(x)
- neg(x)
- mag(x)
- limit(x, y)
- minMod(x, y)
- sqrtSumSqr(x, y)
- sqr(x)
- pow3(x)
- pow4(x)
- pow5(x)
- pow6(x)
- inv(x)
- sqrt(x)
- cbrt(x)

- hypot(x, y)
- exp(x)
- log(x)
- log10(x)
- sin(x)
- cos(x)
- tan(x)
- asin(x)
- acos(x)
- atan(x)
- atan2(x, y)
- sinh(x)
- cosh(x)
- tanh(x)
- asinh(x)
- acosh(x)
- atanh(x)
- erf(x)
- erfc(x)
- lgamma(x)
- j0(x)
- j1(x)
- jn(x, y)
- y0(x)

- y1(x)
- yn(x, y)
- max(x, y)
- min(x, y)
- stabilise(x, y)

H.8.3 Troubleshooting your equations

Your equations may cause you trouble, such as:

- Giving you a `SIGFPE`; or
- Failing dimension checks.

If this happens and you don't know why, **equationReader** has a detailed set of debug switches to help.

To change the debug switch, edit the `OpenFOAM/etc/controlDict` file and add:

```
        equationReader      integerValue;
```

to the `DebugSwitches` list.

The debug switches available are:

0. silent mode;
1. scalar logging (light);
2. scalar logging (verbose);
3. dimension logging (light);
4. dimension logging (verbose);
5. scalar & dimension logging (light); or
6. scalar & dimension logging (verbose).

The scalar logging will report scalar-related operations to the console. The dimension logging, relates to dimensionSet operations. *verbose* reports operation-by-operation, so it can be overwhelming.

H.9 Programming with equationReader

Most of the programming features can be gleaned from the **equationReader** demo application. Please also refer to that.

H.9.1 Creating an equationReader object

To add **equationReader** to an application:

- Put `#include "IOEquationReader.H"` at the top of your main source file;
- Put `#include "createEquationReader.H"` somewhere after `createTime`;

H.9.2 Adding data sources

You need to add data sources - this is where **equationReader** looks for its variables.

H.9.2.1 Beware of duplicate sources

Currently, **equationReader** does not check if you are adding multiple variables of the same name. When this happens, you never know which source will be used. I didn't add it because it didn't occur to me until I started writing this paragraph. Expect it in the future.

H.9.2.2 Is the data permanent?

Data must be permanently available. For instance, `mesh.C()` is a valid data source because it returns a &reference. But `turbulence->R()` is not valid because it returns an object (or `tmp<object>`), and hence is *derived* from other permanent sources.

To use *derived* data sources, there are two options:

1. Create a permanent copy, and update it at every timestep. This is demonstrated in the **equationReaderDemo** application.
2. Create an *activeVariable*.

H.9.2.3 Active variables (advanced developers)

An *activeVariable* is one that does not permanently store its data, and provides values *on-demand* to **equationReader**. The key to this is it must be able to calculate a single cell value on-demand, and not the entire field at once. The interface is given in the `equationReader/equationVariable/equationVariable.H` file.

H.9.2.4 Functions to add data sources

To add data sources:

- for `scalars`, `dimensionedScalars`, `scalarFields`, `GeometricScalarFields`, etc.:

```
 eqns.scalarSources().addSource(scalarObject);
```

  or if it doesn't have its own name (i.e. `scalars` and `scalarFields`) or you want to assign it a different name:

```
eqns.scalarSources().addData(scalarObject, name);
```

- for `vectors`, `dimensionedVectors`, `vectorFields`, `GeometricVectorFields`, etc.:

```
eqns.vectorSources().addSource(vectorObject);
```

  or if it doesn't have its own name (i.e. `scalars` and `scalarFields`) or you want to assign it a different name:

```
eqns.vectorSources().addSource(vectorObject, name);
```

- and so on for other Types (`tensor`, `diagTensor`, `symmTensor`, and `sphericalTensor`);
- for `dictionaries` or `activeVariables`:

```
 eqns.addSource(dataObject);
```

H.9.3 Reading in the equations

To read equations from a dictionary use:

```
eqns.readEquation(dictionaryName, equationName);
```

H.9.4 Searching the equations

equationReader allows you to search its equations. Similar to the `dictionary` object, this will return `true` if *`equationName`* exists:

```
        eqns.found(equationName);
```

The *evaluate* functions below call for `eqnNameOrIndex`. This means you can either use a `word` (the `equationName`), or a `label` (the `equationIndex`). The `equationIndex` is faster, as **equationReader** doesn't have to perform its own lookup. ***Never assume the `equationIndex` is equal to the order in which the equations were read.*** If the equations depend on one another, they may not always be in the same index. To learn the `equationIndex`, use:

```
        equationIndex = eqns.lookup(equationName);
```

H.9.5 Evaluating equations

Once you are done adding data sources, and reading equations, you can start evaluating equations.

H.9.5.1 All data sources required

When evaluating an equation, **equationReader** needs access to all the possible variables and other equations that equation might depend on. If that variable or equation isn't found, **equationReader** produces a **FatalError**. ***Therefore it is a mistake to try adding more data sources after the first evaluation.***

H.9.5.2 No mesh available

equationReader doesn't care about the mesh... all it cares about are the sizes of the the fields. ***The size of the variable fields must match.*** Index checking is expensive, so it is only available in `FULLDEBUG` mode. These rules apply:

- a single-element variable (e.g. a `scalar`, or a `dimensionedVector`) is assumed *uniform throughout the entire domain*, and can be used in any equation;

- a field variable (e.g. a `scalarField`, or a `DimensionedVectorField`) does not have a boundary field, therefore it is only available to equations of other fields or internal fields. Attempting to use it in a `GeometricField` is a mistake; and
- a `GeometricField` variable can be used with any equation.

There are two indices to indicate field / boundary field position:

- `cellIndex` - this is the position within a field (e.g. cell number in the internal field, or face number on a boundary patch);
- `geoIndex`:
    - 0 = the internal field;
    - greater than 0 = the boundary patches. The `geoIndex` is therefore 1-indexed on the boundaryField: `patchI = geoIndex - 1`.

If you omit either of these in the evaluation equations, they are assumed equal to zero.

H.9.5.3 Evaluation functions

- for single element types:

```
scalarA = eqns.evaluateScalar
(
    eqnNameOrIndex,
    [cellIndex],
    [geoIndex]
);
vectorA.x() = eqns.evaluateScalar
(
    xEqnNameOrIndex,
    [cellIndex],
    [geoIndex]
); // and so on for all its components
tensorA.xx() = eqns.evaluateScalar
(
    xxEqnNameOrIndex,
    [cellIndex],
    [geoIndex]
); // and so on for all its components
```

- for `dimensionedScalars`:

```
dimensionedScalarA = eqns.evaluateDimensioned
(
    eqnNameOrIndex,
    [cellIndex],
    [geoIndex]
);
```

- for other `dimensionedTypes` - there is no elegant dimensionChecking... use this hack:

```
vectorA.x() = eqns.evaluateScalar
(
    xEqnNameOrIndex,
    [cellIndex],
    [geoIndex]
);
vectorA.y() = eqns.evaluateScalar
(
    yEqnNameOrIndex,
    [cellIndex],
    [geoIndex]
);
vectorA.z() = eqns.evaluateScalar
(
    zEqnNameOrIndex,
    [cellIndex],
    [geoIndex]
);
vectorA.dimensions() = eqns.evaluateDimensions(xEqnNameOrIndex);
vectorA.dimensions() = eqns.evaluateDimensions(yEqnNameOrIndex);
vectorA.dimensions() = eqns.evaluateDimensions(zEqnNameOrIndex);
```

- for `scalarFields`:

```
eqns.evaluateScalarField(resultScalarField, eqnNameOrIndex, [geoIndex]);
```

  or

```
eqns.evaluateTypeField
(
    resultScalarField,
    dummyWord,
    eqnNameOrIndex,
    [geoIndex]
);
```

- for `vectorFields`:

```
eqns.evaluateTypeField
(
    resultVectorField,
    "x", // this is the component name
    xEqnNameOrIndex,
    [geoIndex]
); // and so on for the "y" and "z" components
```

- and so on for other `typeFields`;

- for `DimensionedScalarFields`:

```
eqns.evaluateDimensionedScalarField
(
    resultDimensionedScalarField,
    eqnNameOrIndex,
    [geoIndex]
);
```

  do not use `evaluateDimensionedTypeField` - this will fail for scalars;

- for `DimensionedVectorFields`:

```
eqns.evaluateDimensionedTypeField
(
    resultDimensionedVectorField,
    xEqnNameOrIndex,
    "x",
    [geoIndex]
); // and so on for the "y" and "z" components
```

- and so on for other `DimensionedTypeFields`;

- for `GeometricScalarFields`:

```
eqns.evaluateGeometricScalarField
(
    resultGeometricScalarField,
    eqnNameOrIndex
);
```

  do not use `evaluateGeometricTypeField` - this will fail for scalars;

- for `GeometricVectorFields`:

```
eqns.evaluateGeometricTypeField
(
    resultGeometricTypeField,
    "x",
    xEqnNameOrIndex
); // and so on for the "y" and "z" components
```

- and so on for other `GeometricTypeFields`;

H.10 Uninstallation

H.10.1 The stand-alone (new) version

H.10.1.1 Am I running the stand-alone version?

Does `src/equationReader/` exist? If so, then you have the stand-alone version.

H.10.1.2 How do I uninstall the stand-alone version?

Enter the following commands into your console:

```
rm -rf $WM_PROJECT_DIR/src/equationReader
rm -rf $WM_PROJECT_DIR/applications/solvers/equationReader
rm -rf $WM_PROJECT_DIR/tutorials/equationReader
```

The stand-alone version of **equationReader** has been uninstalled.

H.10.2 The integrated (old) version

Since the integrated version of **equationReader** is compiled into the core of OpenFOAM, uninstallation requires file editting and recompiling of OpenFOAM.so.

H.10.2.1 Am I running the integrated version?

Does `src/OpenFOAM/db/dictionary/equation/` exist? If so, then you have the integrated version.

H.10.2.2 How do I uninstall the integrated version?

To uninstall the integrated **equationReader**:

1. Edit the `src/OpenFOAM/Make/files` file:

   Find and delete the bold lines below:

   ```
   functionEntries = $(dictionary)/functionEntries
   $(functionEntries)/functionEntry/functionEntry.C
   $(functionEntries)/includeEntry/includeEntry.C
   $(functionEntries)/includeIfPresentEntry/includeIfPresentEntry.C
   $(functionEntries)/inputModeEntry/inputModeEntry.C
   $(functionEntries)/removeEntry/removeEntry.C

   equation = $(dictionary)/equation
   $(equation)/equationReader/equationReader.C
   $(equation)/equationReader/equationReaderIO.C
   $(equation)/equation/equation.C
   $(equation)/equation/equationIO.C
   $(equation)/equationOperation/equationOperation.C

   IOEquationReader = db/IOobjects/IOEquationReader
   $(IOEquationReader)/IOEquationReader.C
   $(IOEquationReader)/IOEquationReaderIO.C

   IOdictionary = db/IOobjects/IOdictionary
   $(IOdictionary)/IOdictionary.C
   $(IOdictionary)/IOdictionaryIO.C
   ```

2. Edit the `src/OpenFOAM/primitives/Scalar/Scalar.C` file:

   Near the top, delete this line:

   ```
   #include "equationReader.H"
   ```

   Near line 84, delete the bold section below:

   ```
      if (t.isNumber())
      {
          s = t.number();
      }
      else if (t.isString())
      {
          // DLFG 2010-07-21 Modifications for equationReader
          equationReader eqn;
          eqn.readEquation
   ```

```
        (
            equation
            (
                "fromScalar",
                t.stringToken()
            )
        );
        s = eqn.evaluate(0).value();
    }
    else
    {
        is.setBad();
        FatalIOErrorIn("operator>>(Istream&, Scalar&)", is)
            << "wrong token type - expected Scalar found " << t.info()
            << exit(FatalIOError);

        return is;
    }
```

From the terminal, enter the following commands:

```
rm -rf $WM_PROJECT_DIR/src/OpenFOAM/db/dictionary/equation
rm -rf $WM_PROJECT_DIR/src/OpenFOAM/db/IOobjects/IOEquationReader
rm -rf $WM_PROJECT_DIR/src/OpenFOAM/lnInclude
rm -rf $WM_PROJECT_DIR/applications/solvers/equationReader
rm -rf $WM_PROJECT_DIR/tutorials/equationReader
rm $FOAM_USER_APPBIN/equationReader*
rm $FOAM_APPBIN/equationReader*
cd $WM_PROJECT_DIR/src/OpenFOAM
rmdepall
wmake libso
```

The integrated version of **equationReader** has been uninstalled.

Appendix I multiSolver extension

multiSolver has been publically released to the *OpenFOAM-extend* project.
This section has also been published on the OpenFOAM wiki (openfoamwiki.net) and is therefore in the public domain.

I.1 What is it?

multiSolver is a master control class that allows you to create a superSolver composed of multiple solvers within a superLoop. All solvers operate on the same dataset in sequence. For example:

1. `icoFoam` - runs to completion;
2. data is handed over to `scalarTransportFoam`;
3. `scalarTransportFoam` - runs to completion;
4. data is handed back to `icoFoam`, and the superLoop repeats.

I.2 Features

- **Multiple solvers** - multiple solvers can be used in sequence on the same data set.
- **Changing boundary conditions** - the boundary conditions can change at distinct time intervals.
- **Independent time** - each solver can operate with an independent time value, although universal time can still be used.
- **Single case directory** - the settings for all solvers are stored within a single case directory using a "multiDict" dictionary format.
- **Easy data management** - All the data output is sorted into subdirectories corresponding to the solver, and can be loaded / unloaded using the multiPost utility.
- **Store fields** - To save memory and hard drive space, not all solvers have to use all the fields. Rather, they can "store" any unneeded fields, leaving more memory and disk space. The next solver retrieves all stored fields, and no data is lost.

I.3 Why would you need this?

A fundamental assumption in the design of OpenFOAM is the existence of a universal time. Therefore the time object is the top-level objectRegistry (i.e. **runTime** hosts the database for your simulation). This design works for nearly all simulations imaginable, except for those that require more than one time frame. For these situations, **multiSolver** will come in handy.

I.4 When would you need this?

The capabilities of **multiSolver** are useful for:

- multi-step processes to be modelled within a single application, e.g. fluid injection, followed by settling;
- modelling of a flow problem characterized by two different timescales, e.g. stirring with biochemical reactions; and
- changing boundary conditions mid-run.

Basically, if you find yourself:

- frequently copying data between case directories;
- frequently stopping and changing the simulation details, then restarting; or
- using `runTime++` more than once in your solver,

then **multiSolver** might help you.

I.5 Mesh motion not fully supported

NOTE: At this time, multiSolver allows for full mesh motion, provided the mesh returns to its original position between solvers. This functionality is planned for the future.

I.6 Update info

- *2010-07-23*: Initial import

- *2011-03-29*: Minor bug fix for 1.6-ext and 1.7.1
- *2011-04-05*: Major upgrade - now works for parallel simulations
- *2011-06-03*: Minor bug fix - decompose nolonger omits the initial directories
- *2011-07-01*: Overhauled the documentation on the wiki page (no change to the code)

I.7 Installation

To install **multiSolver**:

1. If you are running pre-compiled binaries, first make sure you can compile your copy of OpenFOAM.
2. Download the code from here:

```
http://openfoam-extend.svn.sourceforge.net/viewvc/openfoam-
extend/trunk/Breeder_1.5/libraries/multiSolverExtension/?view=tar
```

3. Open a terminal window and browse to the folder with your download.
4. Execute the following commands. You should be able to just copy and paste all 8 lines into your terminal:

```
tar --transform='s,multiSolverExtension,'$WM_PROJECT_DIR',' \
-x -v -z -P -f openfoam-extend-multiSolverExtension.tar.gz
cd $WM_PROJECT_DIR/src/multiSolver
wmake libso
cd $FOAM_APP/utilities/postProcessing/multiSolver
wmake
cd $FOAM_APP/solvers/multiSolver/multiSolverDemo
wmake
```

5. multiSolver should now be installed.

I.7.1 Testing the installation

The installation comes with a demo application and test case. First copy the test case into your `run/tutorials` directory:

```
cp -rf $WM_PROJECT_DIR/tutorials/multiSolver $FOAM_RUN/tutorials
```

To run the test case:

```
cd $FOAM_RUN/tutorials/multiSolver/multiSolverDemo/teeFitting2d
blockMesh
multiSolverDemo
```

To view the results:

```
multiSolver -load all
multiSolver -set icoFoam1
paraFoam
```

I.7.1.1 About the test case

The demo application is:

1. `icoFoam1` - i.e. `icoFoam` with boundary conditions 1;
2. `scalarTransportFoam`;
3. `icoFoam2` - i.e. icoFoam with boundary conditions 2;
4. `scalarTransportFoam` (again);
5. repeat.

The test case is a 2-dimensional tee fitting. The boundary conditions are:

1. `icoFoam1`:

```
        Inlet
         ||
         ||
Closed ====== Outlet
```

2. `scalarTransportFoam`:

```
        T = 1
         ||
         ||
 zeroG ====== zeroG
```

3. `icoFoam2`:

```
        Outlet
         ||
         ||
Outlet ====== Inlet
```

4. `scalarTransportFoam`:

```
        T = 1
         ||
         ||
 zeroG ====== zeroG
```

The test case also has storeFields defined to demonstrate their use:

- `icoFoam` doesn't need the `T` field, so it *stores* this field; and
- `scalarTransportFoam` doesn't need the `P` field.

Since `icoFoam1` is the first to run, it must have all fields defined in its `initial/0` directory, even though it is storing the `T` field.

I.8 Using multiSolver

I.8.1 Case directory structure

The case directory for a *superSolver* is different than a standard solver. The directory transforms itself while the solver runs. It contains transient files that are continuously modified by **multiSolver**, and static files that you use to describe the change.

I.8.1.1 Data

Now, you don't have to worry about this as it is handled automatically, but the most notable difference between a *superSolver* and a standard solver is where they keep their data.

A *typical* normal solver

```
[caseName]
|-system
|-constant
|-0
|-0.2 [timeDirectory]
|-0.4 [timeDirectory]
... more [timeDirectories]
```

A *typical* superSolver

```
[caseName]
|-system
|-constant
'-multiSolver
  |-[solverDomain1]
  | |-initial
  | |-0 [superLoop]
  | | |-0 [timeDirectory]
  | | |-0.2 [timeDirectory]
  | | |-0.4 [timeDirectory]
  | | ... more [timeDirectories]
  | |-1 [superLoop]
  | | |-1.2 [timeDirectory]
  | | |-1.4 [timeDirectory]
  | | |-1.6 [timeDirectory]
  | | ... more [timeDirectories]
  | ... more [superLoops]
  |-[solverDomain2]
  | |-initial
  | |-0 [superLoop]
  | | |-0 [timeDirectory]
  | | |-0.2 [timeDirectory]
  | | |-0.4 [timeDirectory]
  | | ... more [timeDirectories]
  | |-1 [superLoop]
  | | |-1.2 [timeDirectory]
  | | |-1.4 [timeDirectory]
  | | |-1.6 [timeDirectory]
  | | ... more [timeDirectories]
  | ... more [superLoops]
  ... more [solverDomains]
```

In short, standard solvers keep their data in:

```
        [caseName]/[timeValue]
```

whereas *superSolvers* sort their data into superLoop subdirectories within subdirectories named after the solverDomain:

```
        [caseName]/multiSolver/[solverDomainName]/[superLoopIndex]/[timeValue]
```

The standard location of `case/[timeValue]` is used as a temporary loading area, mostly for post-processing. You don't have to worry about this, as it is handled automatically by **multiSolver** and its associated post-processing application.

I.8.1.2 Boundary changes

Changes to the mesh are not yet supported by **multiSolver**, but boundary changes between `patch` and `wall` are supported. To do this, create create multiple copies of the `constant/boundary` file with their respective solver domain names as extensions. For instance:

```
-constant
 |-boundary
 |-boundary.domain1
 |-boundary.domain2
 |-boundary.domain3
 |-faces
 |-neighbour
 |-owner
 `-points
```

I.8.1.3 multiDicts

A regular solver reads information from various dictionary files, and these affect its behaviour. When using **multiSolver**, there will be dictionaries whose contents need to be *different* for each solverDomain. To specify this behaviour, a *multiDict* is used.

multiDicts:

- sit in the same directory as the dictionary they are managing;
- have the prefix `multi`, followed by the name of their child dictionary; and
- are not required if the dictionary doesn't need to change between solvers.

For example, a typical `constant` directory might look like:

```
constant
|-reactionProperties          standard dictionary (does not change)
|-multiTransportProperties   multiDictionary (describes change)
'-transportProperties        auto-generated dictionary (changes)
```

In this directory, there are three files:

- `reactionProperties` - this is a standard dictionary. (You can tell because there's no `multiReactionProperties`.) It is user-editable and will not change during a run. **multiSolver** ignores these files;
- `multiTransportProperties` - this is a multiDict. It is also user-editable and will not change during a run. **multiSolver** recognizes it by its prefix "`multi`". It describes how the dictionary `transportProperties` should change during a run;
- `transportProperties` - this dictionary is automatically generated by **multiSolver**. Its content changes during a run (and during post-processing). Editing this file is only useful for runTime modification.

A multiDict has the following structure:

```
dictionaryName   fvSchemes;

multiSolver
{
    sovlerDomainName1 // this is the solverDomain name
    {
         // settings for the first solver go here
    }
    solverDomainName2 // another solverDomain name
    {
        // settings for the second solver go here
    }
    solverDomainName3 // etc..
    {
    }
    default // optional
    {
        // default settings go here
```

```
                // these are loaded first, then overwritten by solverName (above)
                // solvers whose names are not listed above inherit only these settings,
                //    or none at all if default is absent
            }
        }
```

Sometimes, two or more solverDomains will have identical dictionaries. Rather than write out their settings several times, the `sameAs` keyword is available:

```
        solverDomainName1
        {
            // settings for the first solver go here
        }
        solverDomainName2
        {
            sameAs   solverDomainName1;
        }
```

Exception! The `multiControlDict` contains the settings for **multiSolver**. It is not a standard multiDict. See the next section for details.

I.8.1.4 MultiControlDict

The `multiControlDict` is the **multiSolver** analogue of the `controlDict`. The `controlDict` is auto-generated based on the content of this file. The `multiControlDict` is the main control dictionary and therefore it does not conform to the format of a regular multiDict. It is located in `case/system` and is used to automatically generate the `controlDict`.

Since **multiSolver** has so many optional settings, the full list of settings for the `multiSolverControlDict` is long, but most of these keywords are not required. The full list is shown below:

```
multiSolverControl
{
    initialStartFrom
        firstTime
        firstTimeInStartDomain // *1
        firstTimeInStartDomainInStartSuperLoop // *1 *2
        startTime // *3
        startTimeInStartDomain // *1 *3
        startTimeInStartDomainInStartSuperLoop // *1 *2 *3
        latestTime // (default)
        latestTimeInStartDomain // *1
        latestTimeInStartDomainInStartSuperLoop // *1 *2
    startTime // (required with *3)
    startDomain // (required with *1)
    startSuperLoop // (required with *2)
    finalStopAt
        endTime // (default) *4
        endTimeInEndDomain // *4 *5
        endTimeInEndDomainInEndSuperLoop // *4 *5 *6
        superLoopEnd // *6
        writeNow
        noWriteNow
        nextWrite
    endTime // (required with *4)
    endDomain // (required with *5)
    endSuperLoop // (required with *6)

    multiDictsRunTimeModifiable // (default true)
    timeFormat
    timePrecision
}

solverDomains
{
    solverDomainName1
    {
        startFrom
            firstTime
            startTime // *7
            latestTimeThisDomain
            latestTimeAllDomains // (default)
        startTime // (required with *7)
        stopAt
            endTime // *8 (default)
            writeNow
            noWriteNow
            nextWrite
            iterations // *9 (cannot be used with adjustableTimeStep)
            solverSignal
            elapsedTime // *10
        endTime // (required with *8; default 0)
        iterations // (required with *9)
        elapsedTime // (required with *10)
        storeField
        purgeWriteSuperLoops // (default 0)
// * The rest are the standard controlDict entries, i.e.:
//        writeControl
//            timeStep
//            runTime
//            adjustableRunTime
//            cpuTime
//            clockTime
//        writeInterval
//        purgeWrite
//        writeFormat
//        writePrecision
//        writeCompression
//        runTimeModifiable
//        graphFormat
//        deltaT
//        maxCo
//        adjustTimeStep
//        maxDeltaT
// * Anything else entered here will automatically be merged verbatim into the controlDict
    }
    solverDomainName2
    {
        timeValueStartFrom
        // etc..
    }
    default // (optional, but all prefixes must be defined)
    {
        // values here are loaded first, then overwritten by the solverDomainName
    }
}
```

multiSolverControl

The multiSolverControl subdictionary contains all the settings that affect the superSolver globally.

Initial start settings

- **initialStartFrom** - this setting determines where the superSolver reads the initial data from and begins its run at:
    - **firstTime** - load data from `case/multiSolver/currentSolverDomainName/0/0`
    - **firstTimeInStartDomain** - load data from `case/multiSolver/startDomain/0/0`
    - **firstTimeInStartDomainInStartSuperLoop** - load data from `case/multiSolver/startDomain/startSuperLoop/0`
    - **startTime** - search `case/multiSolver/[allSolverDomains]/[allSuperLoops]` for the closest *globalTime* to *startTime*; load this data
    - **startTimeInStartDomain** - search `case/multiSolver/startDomain/[allSuperLoops]` for the closest *localTime* to *startTime*; load this data
    - **startTimeInStartDomainInStartSuperLoop** - search `case/multiSolver/startDomain/startSuperLoop` for the closest *localTime* to *startTime*; load this data
    - **latestTime** - search `case/multiSolver/[allSolverDomains]/[allSuperLoops]` for the latest *globalTime*; load this data

    - **latestTimeInStartDomain** - search `case/multiSolver/startDomain/[allSuperLoops]` for the latest *localTime*; load this data
    - **latestTimeInStartDomainInStartSuperLoop** - search `case/multiSolver/startDomain/startSuperLoop` for the latest **localTime**; load this data
- **startTime** - only if required (see above)
- **startDomain** - only if required (see above)
- **startSuperLoop** - only if required (see above)

Final stop at settings

- **finalStopAt** - this setting determines when the **multiSolver** will stop the simulation
    - **endTime** - stop when *globalTime* reaches *endTime*
    - **endTimeInEndDomain** - stop when *localTime* reaches *endTime* in sovlerDomain *endDomain*
    - **endTimeInEndDomainInEndSuperLoop** - stop when *localTime* reaches *endTime* in sovlerDomain *endDomain* at superLoop *endSuperLoop*
    - **endSuperLoop** - stop after the superLoop number reaches *endSuperLoop*
    - **writeNow** - stop and write after the next solver starts
    - **noWriteNow** - stop without writing after the next solver starts
    - **nextWrite** - stop at the next designated write time after the next solver starts
- **endTime** - only if required (see above)
- **endDomain** - only if required (see above)
- **endSuperLoop** - only if required (see above)

Other global settings

- **multiDictsRunTimeModifiable** - when set to *on*, **multiSolver** will scan and reread any changed multiDicts
- **timeFormat** - *fixed*, *scientific*, or *general*, the same as in a regular `controlDict`. The timeFormat must be applied globally - there cannot be two solvers using different timeFormats.
- **timePrecision** - again, the same as in the regular **controlDict**.

Solver domains

The solverDomains subdictionary contains all the solverDomainNames as subdictionaries, and also can include a *default*, but *all* solverDomainNames must be included. These subdictionaries contain all the settings that apply *locally* to a single solverDomain.

Local start settings

- **startFrom** - this setting determines the localTime value to start from each time this solverDomain is initialized in each superLoop.
    - **firstTime** - start from localTime = 0
    - **startTime** - start from localTime = *startTime*
    - **latestTimeThisDomain** - start from the localTime it left off at when it was last in this solverDomain
    - **latestTimeAllDomains** - start from the latest globalTime (i.e. localTime and globalTime are equal for this solverDomain)
- **startTime** - only if required (see above)

Local stop settings

- **stopAt** - this setting determines where the solver will stop within each superLoop.
    - **endTime** - stop when localTime = *endTime*
    - **writeNow** - stop now and write out the results
    - **noWriteNow** - stop now without writing out results
    - **nextWrite** - stop at the next scheduled write time
    - **iterations** - stop after *endIterations* have been achieved. This setting cannot be used if *adjustableTimeStep* is enabled for this solverDomain
    - **solverSignal** - leave it up to the solver to give the stop signal. This essentially sets *endTime* to a very large number
    - **elapsedTime** - stop after elapsedTime has passed for this solverDomain
- **endTime** - only if required (see above)
- **iterations** - only if required (see above)
- **elapsedTime** - only if required (see above)

Other local settings

- **storeField** - a wordList of any fields that this solverDomain doesn't need. This saves the field from having to be carried in memory and written out at every write time, but also allows the latest values of the field to be passed on to the next solverDomain. This works by copying the field to the last write time of the solverDomain in question.
- **purgeWriteSuperLoops** - this works the same as the standard *purgeWrite*, except instead of overwriting `[timeValue]` directories, `[superLoop]` directories are overwritten.

Any other values placed in the `solverDomainName` subdictionary will be merged verbatim into its `controlDict`.

Default solver domain

Any values placed in the default solverDomain will be loaded first, and written over by any values specific to a solverDomain. Although this is a **default** subdictionary, **all solverDomainNames must be present in the solverDomains subdictionary**.

I.8.1.5 Boundary conditions and initial values

The boundary conditions and initial values are located in:

```
        case/multiSolver/[solverDomainName]/initial/0
```

This is the analogue of the `case/0` directory, except there is one for every solverDomain. Unlike in a regular simulation, **multiSolver** will always refer to the boundary conditions located in `initial/0`.

I.8.1.5.1 Changing boundary conditions

multiSolver allows the boundary conditions to change between solverDomains. To accomplish this, the latest internalField is combined with the boundaryField from `initial/0`.

I.8.1.6 Advanced boundary condition settings

I built in some functionality into **multiSolver** that I don't think is actually necessary. It basically allows you to have the updated boundary field values skip solverDomains. The feature is also untested. I'm commenting out the documentation for this section.

I.8.2 Store fields

Some solvers may not need to use all the fields created by other solvers. On the other hand, these other solvers need the latest values for these fields. There are two options for handling this:

1. add the unneeded fields to the `createFields.H` of the solver. The extra fields will be carried in memory, and written out at every time step.

2. declare these fields as storeFields in the `multiControlDict`. With this option, for the solver that doesn't need them, these fields will not be loaded in memory, and will be written only to the first timestep in each run.

Note: The *first* solverDomain to run must have all fields from all solvers defined in its `initial/0` directory.

I.8.3 Local time and Global time

A fundamental principle of **multiSolver** is that time is independent between solverDomains. (If this causes you apprehension, don't worry, the default behaviour uses a standard global time.) Therefore there are two defined time values:

- *localTime* - the value known to the solver; and
- *globalTime* - the universal time, known only by **multiSolver**.

I.8.3.1 Initial start

The `multiControlDict` has settings for an initial start defined gloablly (`initialStartAt`), and an initial start defined for each solverDomain (`startAt`). Sometimes these settings may appear to come into conflict. What happens when `initialStartAt` is set to `latestTimeAllDomains`, but `startAt` is set to `startTime = 0`?

The `initialStartAt` settings are where the initial data is read from. The `startAt` settings are where the local time value starts from. Sometimes you may be trying to resume from the middle of a run, and the initial time value should pick up from where it left off. **multiSolver** tries to determine when this is the case. The rules are:

- `startAt` time is equal to the *localTime* of the `initialStartAt` data source;

- if the `initialStartAt` is from a different solverDomain than the initial solverDomain, the `startAt` time specified in the `multiControlDict` is used instead.

I.8.3.2 Switching domains

When switching domains:

- *globalTime* stays the same (i.e. time does not step when switching domains); and
- *localTime* is set to the value specified by the `startAt` settings in the `multiControlDict`.

I.8.3.3 End time

The local `stopAt` settings are always compared with the global `finalStopAt` settings. If the `finalStopAt` value occurs before the local `stopAt` value, the `finalStopAt` value is used, and the end condition is set.

I.8.3.4 Initial superLoop

The initial superLoop value is determined by the `initialStartAt` settings. It is set to be equal to the superLoop value of the `initialStartAt` data source. If the `initialStartAt` data source is from a different solverDomain than the initial solverDomain, the next superLoop is used.

I.8.4 Runtime Modification

There are two levels of runtime modification: within a solverDomain, and globally.

- Editting a standard dictionary (e.g. `controlDict`, or `transportProperties` applies to a solverDomain. Its effect depends on whether that solver has `runTimeModifiable` enabled. This happens at the end of a solver iteration. However, these changes will be lost in the next superLoop when the same solverDomain is initialized.
- Editting a multiDict dictionary applies globally. This is goverened by `multiDictsRunTimeModifiable` setting in the `multiControlDict`, but these

modifications do not take place until the next solverDomain is initialized. However, these changes are permanent.

I.9 Parallel

Parallel processing is now available with **multiSolver**.

I.9.1 decomposePar

To decompose the case directory:

1. Set it up as a usual multiSolver-enabled case directory;
2. Create a `system/decomposeParDict` file as you would with a regular parallel solver;
3. Instead of `decomposePar`, use:

```
multiSolver -preDecompose && decomposePar && multiSolver -postDecompose
```

I.9.2 reconstructPar

To reconstruct the case directory, instead of `reconstructPar`, use:

```
multiSolver -preReconstruct && reconstructPar && multiSolver -postReconstruct
```

I.9.3 Aliasing

If you are going to be doing this regularly, it might be a good idea to create a shorter alias for these two commands. To do this, add:

```
alias multiDecomposePar='multiSolver -preDecompose && decomposePar && multiSolver -
postDecompose'
alias multiReconstructPar='multiSolver -preReconstruct && reconstructPar && multiSolver -
postReconstruct'
```

To the end of your `OpenFOAM-version/etc/aliases.sh` or `OpenFOAM-version/etc/aliases.csh` file.

I.9.4 Parallel post processing

Apparently post processors are available that work with the data split across the processor directories / drives. **multiSolver** can be post processed in this way as well. To achieve this, use the commands as described on the post processing page, except run them in parallel.

For example, instead of:

```
multiSolver -load all
```

use:

```
mpirun -n 4 multiSolver -load all -parallel
```

substituting the correct options for mpirun. Then run your fancy parallel post processor.

I.10 Programming

I.10.1 Programming basics

Programming a multiSolver-enabled application is almost as simple as pasting two solvers together.

I.10.1.1 Simple example

Often a simple example is enough to get started. Here's a simple multiSolver-enabled application, or "superSolver":

```
/*---------------------------------------------------------------------------*\
                             ... STANDARD HEADER ...
\*---------------------------------------------------------------------------*/

#include "fvCFD.H"
#include "multiSolver.H"

// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

int main(int argc, char *argv[])
{

#   include "setRootCase.H"
#   include "createMultiSolver.H"

// * * * * * * * * * * * * * * * * icoFoam  * * * * * * * * * * * * * * * * //

    Info << "*** Switching to icoFoam ***\n" << endl;
    solverDomain = "icoFoam";
#   include "setSolverDomain.H"

// Paste everything from icoFoam.C, starting with #include "createTime.H",
// and ending just before (but not including) return 0;

// * * * * * * * * * * * * * scalarTransportFoam * * * * * * * * * * * * * * //

    Info << "*** Switching to scalarTransportFoam ***\n" << endl;
    solverDomain = "scalarTransportFoam";
#   include "setSolverDomain.H"

// Paste everything from scalarTransportFoam.C, again, starting with
// #include "createTime.H", and ending just before (but not including)
// return 0;

#   include "endMultiSolver.H"
    return(0);
}

// ************************************************************************* //
```

I.10.1.2 Basic strategy

- Write (or choose existing) solvers that you intend to use with your superSolver;
- The `createFields.H` files (and associate #include statements) have to be renamed if they differ between solvers;
- Add `#include "multiSolver.H"` to the top of the solver;
- Just after `#include "setRootCase.H"`, add: `#include "createMultiSolver.H"`
- Between solvers use:

```
    solverDomain = "nextSolverDomain";
#   include "setSolverDomain.H"
```

- End with:

```
#include "endMultiSolver.H"
return (0);
```

I.10.2 Advanced concepts

If the basic framework described above doesn't suit your needs, read on. This section also covers some semantics that may be useful to know.

I.10.2.1 More on solverDomains

A solverDomain is an individual solver loop. It is assigned a name, and the list of names is static. A solverDomainName cannot be:

- `all;`

- `constant`;
- `default`; or
- `root`.

All solverDomains must appear in the `case/system/multiControlDict` file, although declaring additional names is not a problem.

I.10.2.2 Order of execution

In the simple example above, all the solverDomains execute in sequence, once per superLoop. This is not necessary: you can enclose them in conditionals; they can execute in any order; they can miss entire superLoops; however, they cannot execute more than once per superLoop. Use: `multiRun++` between solvers to force the superLoop number to increment if necessary.

Note: Using a `multiRun++` statement may lead to user-confusion with the `endSuperLoop` condition for `finalStopAt`.

I.10.2.3 runTime must go out of scope

Looking at the include files specified in the simple example, you will notice that the entire solver loop is enclosed in its own set of braces { }, starting before `#include "createTime.H"`. This is necessary because runTime, the mesh, and all fields must go out of scope before **multiSolver** initializes another solverDomain.

I.10.2.4 End condition

multiSolver will detect the end condition automatically during the `setSolver` function. It will archive the last `case/time` directory, and exit the superLoop. This is achieved using the `#include` framework described in the simple example. If you are deviating from this framework, the requirements for correctly ending the **multiSolver** are:

- `setSolver` will automatically detect an end condition;
- to force an end condition, use `setFinished();`
- once the end condition is met, the function `setSolverDomain` must be encountered at least once more (although it may be encountered any number of times) to perform the final data clean-up;
- the function `multiRun()` returns true if another `setSolverDomain` still must be encountered; false means the run has finished, and `setSolverDomain` has completed the final clean-up;
- the inidivual solverDomain loops cannot be encountered after the final clean-up has taken place. Enclose them each in:

```
if (multiRun.run())
{
    // solverDomain loop
}
```

I.10.2.5 #undef directives

Sometimes there will be conflicts with `#define` directives across solverDomains. For instance, if you have more than one solver using `#include "createPhi.H"`, only the first solver will recognize it. This is caused by the fact that the `createPhi.H` file has this structure:

```
#ifndef createPhi_H
#define createPhi_H

// createPhi code

#endif
```

These preprocessor directives ensure the code for `createPhi` is read only once, regardless of how many times it is included. This prevents the compiler from complaining that something is being redefined. The problem is, when we switch solver domains, `phi` goes out of scope, and it is not recreated. To overcome this, use an `#undef` directive between solver domains: `#undef createPhi.H`

This, and a few other `#undef` directives are already included by default in the `setSolverDomain.H` file. You may encounter others that need to be undefined. Please let me know if you do, and I will add it to the `setSolverDomain.H` file. In the mean time, to add an `#undef` directive:

- using `#include "setSolverDomain.H"`:

```
    Info << "*** Switching to icoFoam ***\n" << endl;
#   undef [conflicting definition]
    solverDomain = "icoFoam";
#   include "setSolverDomain.H"
```

- without using `#include "setSolverDomain.H"`:

```
} // previous solver domain goes out of scope
multiRun.setSolverDomain(solverDomain);

#undef [conflicting definition]

if (multiRun.run())
{ // next solver domain comes into scope
```

where **`[conflicting definition]`** is the definition that needs to be removed.

I.10.3 How does it work?

OpenFOAM is incredibly flexible, and easily extensible, but implementing a change of this kind challenged its founding assumptions. Therefore, the flexibility was not there on level it needed to be, leaving little option but to use a top-level wrapper implementation.

A wrapper encloses the targeted object in a class that gives it the environment it expects to operate, while simultaneously presenting a different environment to other objects interfacing with it. At the top-level, the "other objects" are users. (Strictly speaking, at the code-level, multiSolver is not a true wrapper since it doesn't include an "OpenFOAM solver" as a member variable, but it is in principle.) **multiSolver** works by mutating the case directory into what each solver requires. A transient solver will see the correct `ddtSchemes` setting in `fvSchemes`; likewise a steady state solver will see `steadyState` for `ddtSchemes`. This is the purpose of the multiDict dictionary format.

The data output and input are hard-coded to the `case/[timeValue]` directory. Therefore, when **multiSolver** initializes the next solverDomain, it archives the existing output into the correct directory at `case/multiSolver/[solverDomainName]/[superLoopIndex]/[timeValue]`, and copies the latest field values to the initial time the next solver expects.

I.10.4 Reference

This reference section gives an overview of the functions available to you. You don't need to know any of this. It might be better just to look at the source.

I.10.4.1 Solver interface functions

Functions designed for use within a solver.

setSolverDomain

This function mutates the case directory into what the next solverDomain expects. It:

- rereads any modified multiDicts;
- archives the existing data to `case/multiSolver/solverDomain/superLoop/time`;
- copies the current field data to the `case/time/` directory, swapping the boundary conditions if necessary;
- creates and writes the new `controlDict`;
- swaps all multiDicts to the next solverDomain; and
- checks for the end condition.

setFinished

This function tells multiSolver that once the current solverDomain is finihsed, the full superSolver run is finished. (Technical, it tells multiSolver to save the last data and clean-up at the next `setSolverDomain()`, then the full superSolver run is finished.)

operator++

This function increments the superLoop number. It must be used if the same solverDomain is visited more than once in the same superLoop (otherwise it will overwrite its previous data). It can be used once between any pair of `setSolverDomain()` functions.

run and end

The `run()` and `end()` functions are analogous to those of the same name in the `Time` class. The first is true when the run should continue; the latter is true when the run should end.

I.10.4.2 Post processing functions

There are several functions designed for post-processing. Many of them depend on the use of the `timeCluster` and `timeClusterList` classes.

setSolverDomainPostProcessing

This mutates the case directory to that expected by the specified solverDomain. This is necessary for post-processors that read the controlDict.

timeFunctions

There are several searching / cataloging functions available for post-processing. Their function is described in the **multiSolver.H** header file. These include:

- **`findSuperLoops`** - list all the superLoop directories in a given path;
- **`findClosestGlobalTime`** - find the closest globalTime to a given value in a `timeClusterList`;
- **`findClosestLocalTime`** - find the closest localTime to a given value in a `timeClusterList`. If timeClusters are overlapping, this function only uses the those from the latest superLoop;
- **`findInstancePath`** - return the path to a given `timeCluster` and index;
- **`findMaxSuperLoopValue`** - maximum superLoop by value;
- **`findMaxSuperLoopIndices`** - return a labelList of the `timeClusters` with the maximum superLoop value;
- **`nonOverlapping`** - if the `timeClusters` overlap in time, return false. `timeClusters` that share starting points, or share ending points are non-overlapping;
- **`readSuperLoopTimes`** - catalog the time directories in `case/multiSolver/givenSolverDomain/givenSuperLoop`;
- **`readSolverDomainTimes`** - catalog the time directories in `case/multiSolver/givenSolverDomain/allSuperLoops`;
- **`readAllTimes`** - catalog the time directories in `case/multiSolver/allSolverDomains/allSuperLoops`;
- **`loadTimeClusterList`** - copy / move time directories in a `timeclusterList` to `case/time`;
- **`archiveTimeDirectories`** - copy / move time directories from sourcePath to destinationPath; and
- **`purgeTimeDirectories`** - delete all time directories in a given path.

I.10.4.3 Support classes

There are a few additional classes that were written to support **multiSolver**. These include:

- `tuple2List` class;
- `timeCluster` class;
- `timeClusterList` class; and
- `dummyControlDict` class.

tuple2List

This is a sortable list of paired values was created. It is sortable by first or second value, and currently can be any combination of `scalar` or `label`.

timeCluster

`timeCluster` is to **multiSolver** what `instant` is to runTime. This object holds all the information necessary to catalog the data within a single `solverDomain/superLoop` directory. It holds:

- the solverDomain name;
- the superLoop number;
- the globalTimeOffset; and
- an instantList, cataloging all the time directories within the directory.

Sometimes, a `timeCluster` is used to identify a single time directory within a solverDomain/superLoop. This is useful for functions such as: `findClosestGlobalTime`, which needs to identify a single time directory. To assist in this operation, `operator()` has been defined. It creates just such a `timeCluster`, given the index of the instant.

(This is a bit messy, it might have been smarter to create a different class to distinguish between single time directories, and time directory lists.)

timeClusterList

Again, similar to `instantList`, for **multiSolver**. A `timeClusterList` can catalog all the data in the case directory. Unlike `instantList`, this class has some functions of its own:

- **`append`** - add another `timeCluster` or `timeClusterList` to its collective;
- **`globalSort`** - sort its constituent `timeClusters` by their minimum globalTime;
- **`purgeEmpties`** - remove any `timeClusters` that have an empty `instantList`. Returns false if none remain. Many functions depend on a non-empty `instantList`.
- **`selectiveSubList`** - returns a `timeClusterList` that is composed of a subList of the original `timeClusterList`. The sublist is seleted by index using a labelList. The sublist is not a true sublist like in other classes; rather it is simply another timeClusterList.

Note: Use **`purgeEmpties`** to ensure there are no empty `timeClusters`. Many of the timeFunctions will throw a fatal error if passed an empty `timeClusterList`.

dummyControlDict

In order to allow runTimeModification of **multiSolver**'s multiDicts, **multiSolver** required an objectRegistry that doesn't dissappear between solverDomains, when runTime goes out of scope. Therefore it needed its own objectRegistry. Hence, `multiDictRegistry_` is a `Time` object. `Time` was never intended to be a member variable, therefore its constructors do not allow initialization without a `controlDict`. The object `dummyControlDict` was introduced as a self-initializing, minimal `controlDict`.

`dummycontrolDict` also has constructors that take a `multiControlDict`; or its name. These constructors look for the `timeFormat` and `timePrecision` keywords. If found, it includes these in its settings. These settings are static variables owned by `Time`; and to simplify implementation, it was made universal - i.e. they can only be set once (at initialization) in **multiSolver**.

Ultimately, the `dummyControlDict` was necessary for global runTimeModification.

I.11 Post processing

multiSolver has a post processing tool with the same name, which is required to make data available to the standard post processors.

I.11.1 Overview

OpenFOAM is hard-coded to look for data in the `case/[time]` directories. In order to post-process (including sampling, and data conversion) the data needs to be there. To accomplish this, a post processing utility is available. Typing the command `multiSolver` in the terminal accesses this. It has four main commands:

- `-load` - copy data files from their storage location to `case/time` (i.e. load the data for post-processing);
- `-purge` - delete data files;
- `-list` - display the contents found in the case directory storage location; and
- `-set` - make the case directory appear as required for the given solver name.

I.11.2 Syntax

```
multiSolver -command 'options' [-global] [-local] [-noPurge] [-noSet] [-noStore]
```

I.11.3 -command

Choose one of:

- **list** - list data available
- **load** - copy specified data to case/time
- **purge** - delete specified data
- **set** - change case directory to match supplied solver domain

I.11.3.1 list

Takes no options. Giving options will produce an error.

```
multiSolver -list
```

I.11.3.2 load

Loads the data specified in options. Additional options:

- `-global`, `-local` - By default, `load` will copy by *localTime*, unless the times overlap, in which case it loads by *globalTime*. These options force it to load by the specified time.
- `-noPurge` - By default, load will purge the `case/time` directories before copying the new data in. This option disables this behaviour.
- `-noSet` - By default, if all the specified load data is from a single solverDomain, load will automatically set the case directory to this solverDomain. This option disables this behaviour.
- `-noStore` - By default, if any *storeFields* exist, they will be copied into every time directory in which they are missing - this is for ease of post-processing. This option disables this copying, leaving only the data that actually exists in the `multiSolver` directory.

Load all data:

```
multiSolver -load all
```

Load all data from icoFoam solverDomain, using globalTime.

```
multiSolver -load icoFoam -global
```

Load all data from scalarTransportFoam, superLoops 1, 2, and 5 through 9, but do not delete the `case/time` directories first:

```
        multiSolver -load 'scalarTransportFoam 1 2 5:9' -noPurge
```

I.11.3.3 purge

Purges the data specified in options.

Delete all the `case/time` directories:

```
        multiSolver -purge root
```

Delete superloops 5 through 9 in all solverDomains:

```
        multiSolver -purge '5:9'
```

Delete all data in all solverDomains (except for initial directories):

```
        multiSolver -purge all
```

I.11.3.4 set

Sets the case directory to a given solverDomain. That is, all multiDicts are changed to the correct solverDomain; and a `controlDict` is written. The `controlDict` doesn't have all the data for the given solverDomain, but it (importantly) points to the first `case/time` directory as its `startFrom`. `paraFoam` uses this to initialize.

Set can only take a single solverDomain in options. Any superLoop specifications, or additional solverDomain names will result in error.

I.11.4 'options'

The options to the command. If the options have whitespace characters (i.e. are more than 1 word), they must be enclosed in apostrophes. Options come in two parts: solverDomains, followed by superLoops. Each can have any number of entries, including zero, but options cannot be empty unless using `-list`.

I.11.4.1 solverDomainNames

A simple word list, space delimited. List any solverDomains you want to load data from. e.g.:

```
        scalarTransportFoam icoFoam customSolver
```

or

```
        icoFoam
```

Omitting solverDomainNames indicates that *all* solverDomains will be used.

I.11.4.2 superLoop numbers

A space delimited number list in any order. Can include ranges using a : character. A value of `-1` indicates the initial directory. e.g.:

```
        3 5 6 9:12
```

or

```
        -1 5
```

Omitting superLoop numbers entirely indicates that *all* superLoops will be used.

I.11.4.3 Special words

There are two special words reserved for some of the commands:

- **`all`** - indicates all solverDomains and all superLoops. Useable by `-load` and `-purge`.
- **`root`** - indicates all case/time directories. Useable by -purge only.

e.g.:

```
        multiSolver -load all
        multiSolver -purge root
```

I.11.5 Examples

Show all available data:

```
        multiSolver -list
```

Load all available data:

```
        multiSolver -load all
```

Load all data from solverDomain icoFoam:

```
        multiSolver -load icoFoam
```

Load all data from superLoops 8 and 9 from all solverDomains:

```
multiSolver -load '8 9'
```

Load all data from superLoops 4, 6, 7, 8, and 9 from solverDomain scalarTransportFoam, but do not delete any existing case/time directories.

```
multiSolver -load 'scalarTransportFoam 4 6:9' -noPurge
```

Load all data from superLoops 4, 5, 7, 8 and 9 from solverDomains icoFoam and scalarTransportFoam, but force the directory names to globalTime:

```
multiSolver -load 'icoFoam scalarTransportFoam 4 5 7:9' -global
```

Delete all case/time directories:

```
multiSolver -purge root
```

Delete all time directories in case/multiSolver/allSolverDomains/allSuperLoops, but do not delete any initial directories:

```
multiSolver -purge all
```

Delete superLoops 5, 6, 7, 8, and 9 from all solverDomains:

```
multiSolver -purge '5:9'
```

This instruction says to delete the initial directory from solverDomain icoFoam, but the initial directory is never deleted, therefore nothing is done:

```
multiSolver -purge 'icoFoam -1'
```

Set the case directory to scalarTransportFoam's settings:

```
multiSolver -set scalarTransportFoam
```

Appendix J plcEmulator extension

This section contains documentation for a project that has not yet been released publicly. It is intended to be released at a later date.

J.1 What is it?

The **plcEmulator** extension is a programmable logic controller (PLC) emulator that is built directly into OpenFOAM. Not only can it control the components in the system, it also can control the algorithm the solver uses: **plcEmulator** allows for *algorithm-switching*. It uses **multiSolver** to manage a process control system for your CFD simulations.

J.2 Why would you need this?

Many engineering processes involve control systems. **plcEmulator** allows you to produce time-resolved simulations of these processes, simultaneously evaluating the fluid flow and the control system. For applications characterised by distinct changes in the simulation physics, algorithm-switching allows the solver to change behaviour as required. For example, in spacecraft re-entry simulations, the simulation begins with a rarefied medium flow, best handled with Lagrangian methods; and ends with a continuum flow, best handled with Eulerian methods. Applications include:

- HVAC systems;
- chemical processes; and
- processes with distinct steps.

J.3 How do you use it?

I recommend that you first familiarise yourself with **multiSolver**. Once you are comfortable with that, you need an application that makes use of **plcEmulator**, such as the demo that comes with the release. Programming your own custom application is covered later. Ultimately, using **plcEmulator** amounts to managing the solverDomains in your case directory, and editing the `plcControllerDict` file in `[case]/constant`.

J.4 Case directory

plcEmulator is an extension of **multiSolver**, so the case directory has to conform to the oddities that **multiSolver** requires. Also, there's an extra file you need to create: the `plcControllerDict` file, located in `[case]/constant`. This file is most important as it lists all the inputs, outputs and user-supplied logic.

J.4.1 Outputs

plcEmulator uses **multiSolver**'s solverDomains as outputs. Say there are three controllable outputs: a heater (on / off), a mixer (a swirling source term, or zero), and fluid flow (high / low). This gives eight possible system states:

- heater off, mixer off, fluid flow low;
- heater off, mixer off, fluid flow high;
- heater off, mixer on, fluid flow low;
- heater off, mixer on, fluid flow high;
- heater on, mixer off, fluid flow low;
- heater on, mixer off, fluid flow high;
- heater on, mixer on, fluid flow low; and
- heater on, mixer on, fluid flow high.

To establish the outputs of the system, you have to define a solverDomain for each of these possibilities. In the **multiSolver**-enabled case directory, you give each one the appropriate set of boundary conditions / source terms. For instance, solverDomains with the heater on has a positive temperature gradient set on the heater boundary patch. For the mixer, say you are using a solver algorithm that looks up a sourceTerm setting from `transportProperties` or some other file. To actuate the mixer, you setup the `transportProperties` file as a multiDict, and have it declare a swirling source term for solverDomains with the mixer on, and zero for the others.

J.4.2 plcControllerDict file syntax

The basic syntax of this file is:

```
algorithmGroups
{
    // algorithm groups go here
}

triggers
{
    // triggers (inputs) go here
}

logic
{
    // logic goes here
}
```

J.4.3 Algorithm groups

If your solver doesn't use algorithm-switching, just list all your solver domains in the `algorithmGroups` section of the `plcControllerDict` file. For example:

```
algorithmGroups
{
    default
    (
        heaterOnMixerOn heaterOnMixerOff heaterOffMixerOn heaterOffMixerOff
    );
}
```

However, if your solver does use algorithm-switching, then you need to treat the algorithm as another controllable output, and define a set of solverDomains for each algorithm. For example, say your solver switches between Lagrangian and Eulerian algorithms. You need to create solverDomains for each type, and list them by algorithm in the `plcControllerDict` file:

```
algorithmGroups
{
    Lagrangian
    (
        LheaterOnMixerOn LheaterOnMixerOff LheaterOffMixerOn LheaterOffMixerOff
    );
    Eulerian
    (
        EheaterOnMixerOn EheaterOnMixerOff EheaterOffMixerOn EheaterOffMixerOff
    );
    }
```

J.4.4 Inputs

(In the source code, inputs are called "triggers".) The inputs are the sensors and timers in the equipment you are simulating. They can include things like thermocouples, flow meters, and so on. You specify these entirely in the `plcControllerDict` file in the `triggers` section. Inputs can be:

- a `conditionalSwitch`;
- an `equationLimit`;
- a `solverDomainGroup`;
- a `solverSignal`;
- a `timer`; and
- a `volScalarFieldLimit`.

The inputs are based on runTimeSelection, and therefore are extensible.

J.4.4.1 Conditional switch

The `conditionalSwitch` input is **TRUE** or **FALSE** depending on the condition of the other inputs. Its syntax is:

```
[triggerName]
{
    type    conditionalSwitch;
    initialValue    true/false; // default false
    trueWhen
    {
        triggered       ( [triggerName0] [triggerName1] ... );
        notTriggered    ( [triggerName0] [triggerName1] ... );
    }
    falseWhen
    {
        triggered       ( [triggerName0] [triggerName1] ... );
        notTriggered    ( [triggerName0] [triggerName1] ... );
    }
}
```

- `[triggerName]` is a unique name.
- Any of the lists of triggers can be empty.
- Should the conditional switch not have a valid solution, it holds its previous value.

- If you leave both associated `triggered` and `notTriggered` lists empty, they will always be valid. In this case, the switch will only take the opposite value when its conditions are met. This means:
    - if `falseWhen` has empty trigger lists, the switch is always **FALSE** unless `trueWhen` is met; and
    - if `trueWhen` has empty trigger lists, the switch is always **TRUE** unless `falseWhen` is met.

J.4.4.2 Equation limit

The `equationLimit` trigger is **TRUE** or **FALSE** depending on the result of a user-defined equation. This depends on the **equationReader** extension. Its syntax is:

```
[triggerName]
{
    type    equationLimit;
    initialValue    true/false; // default false
    equationName    [equationName];
    equationReaderName  [equationReaderName]; // default eqns
    equationReaderObjectRegistry    ( [pathFromRunTime] );
    greaterThan     [scalar]; // optional if lessThan is defined
    lessThan        [scalar]; // optional if greaterThan is defined

    // And any one of the evaluation methods, with its associated settings:
    evaluation      point;
    geoIndex        [label]; // default 0
    cellIndex       [label];

    evaluation      patchFieldAverage;
    patchIndex      [label];
    meshName        [word]; // default region0
    meshObjectRegistry  ( [pathFromRunTime] );

    evaluation      patchFieldMaximum;
    patchIndex      [label];
    meshName        [word]; // default region0
    meshObjectRegistry  ( [pathFromRunTime] );

    evaluation      patchFieldMinimum;
    patchIndex      [label];
    meshName        [word]; // default region0
    meshObjectRegistry  ( [pathFromRunTime] );

    evaluation      internalFieldAverage;
    meshName        [word]; // default region0
    meshObjectRegistry  ( [pathFromRunTime] );

    evaluation      internalFieldMaximum;
    meshName        [word]; // default region0
    meshObjectRegistry  ( [pathFromRunTime] );

    evaluation      internalFieldMinimum;
    meshName        [word]; // default region0
```

```
        meshObjectRegistry   ( [pathFromRunTime] );
    }
```

The **plcEmulator** changes control states (solverDomains) by ending simulations and automatically starting a new one with the new control state. Since it does this, OpenFOAM's in-memory database is constantly being rebuilt. This means that the plcEmulator cannot hold onto any data in the database. Instead, it searches OpenFOAM's in-memory database each time it needs something. This in-memory database is called the objectRegistry. The equationLimit switch must be able to find an **equationReader** in the objectRegistry, and if the evaluations depend on the mesh, it has to find the mesh too. Therefore you have to specify their names in the objectRegistry with `equationReaderName` and `meshName`. These are usually "`eqns`" and "`region0`" respectively. You also have to specify *which* objectRegistry to search with `pathFromRunTime`. The syntax for this is:

- `.` = runTime
- `subRegistry` or `./subRegistry` = one objectRegistry under runTime
- `./subRegistry0/subRegistry1/...` = the general case

You can specify multiple objectRegistry paths. This searches in `runTime` and `runTime/region0`:

```
    equationReaderObjectRegistry    ( . ./region0 );
```

The point evaluation method requires a `geoIndex` and a `cellIndex`:

- `geoIndex` is `0` for the internal field, or `(patchIndex + 1)` for boundary patches; and
- `cellIndex` is the element number of the associated field.

J.4.4.3 Solver domain group

The `solverDomainGroup` switch is **TRUE** if the current solverDomain is contained in a list you specify. The syntax is:

```
    [triggerName]
    {
        type    solverDomainGroup;
        initialValue    true/false; // default false
        solverDomains   ( [solverDomainName0] [solverDomainName1] ...);
```

```
}
```

J.4.4.4 Solver signal

The solver can send signals to **plcEmulator**. For instance, consider a solver that runs icoFoam until it reaches steady state, then it switches to pisoFoam. Once it detects steady state, it sends a signal to **plcEmulator**, which then switches the algorithm to pisoFoam. The syntax of this is:

```
[triggerName]
{
    type    solverSignal;
    initialValue    true/false; // default false
}
```

In the source code of the solver, it is:

```
control.signal("[signalName]");            // this toggles the signal
control.signal("[signalName]", true);      // this sets the signal to true
control.signal("[signalName]", false);     // this sets the signal to false
```

J.4.4.5 Timer

Timers can be used as inputs. The syntax is:

```
[triggerName]
{
    type    timer;
    initialValue        true/false; // default false
    duration            [scalar];
    valueWhenActive     [true/false]; // default true
    roundOff            [scalar];
    nRepeat             [label]; // default -1
    startAt             runStart;
    startAt             trigger;
    trigger             [triggerName]; // required if startAt is trigger
}
```

- When the timer is running, it holds the value specified by `valueWhenActive`.
- When the timer is not running, it holds the opposite value.
- Since floating point arithmetic is never 100% accurate, you have to tell it how much `roundOff` error to use when determining whether the timer has started / stopped.
- `nRepeat` is how many times the timer can repeat, where:
  - `-1` = indefinitely;
  - `0` = run only once; and
  - `N` = the number of repetitions (i.e. `1` means it will run twice).

- The timer can be set to start at the beginning of the simulation: `runStart`, or be triggered by another input: `trigger`.
- Timers set to `startAt runStart`:
  - Regardless of the settings in the `controlDict`, this timer starts at t=0.
  - If set to repeat, this timer will repeat the instant it expires.
- Timers set to `startAt trigger`:
  - When inactive, this timer waits until its start trigger (specified by the `trigger` keyword) becomes **TRUE**, then it starts running.
  - Once running, this timer will continue running until the time expires, regardless of what happens with its start trigger.
  - If this timer is set to repeat, it will wait until its start trigger becomes **TRUE** again before repeating.
  - I think you can set this trigger to start at the beginning of a run: set `initialValue` to the same as `valueWhenActive`.
  - You may have to set `initialValue` to the opposite of `valueWhenActive` to ensure it doesn't start running at the beginning of a simulation.
- When a timer expires, the **plcEmulator** handles it carefully:
  - Even if it instantly repeats, the timer sends a pulse of its inactive value to the plcEmulator.
  - This timer pulse is valid for one change in control state: if it changes the control state, the pulse is used up.

J.4.4.6 volScalarFieldLimit

A volScalarField can be used as a trigger. The syntax is:

```
[triggerName]
```

```
    {
        type    volScalarFieldLimit;
        initialValue    true/false; // default false
        variableName    [equationName];
        objectRegistry  ( [pathFromRunTime] );
        greaterThan     [scalar]; // optional if lessThan is defined
        lessThan        [scalar]; // optional if greaterThan is defined

        // And any one of the evaluation methods, with its associated settings:
        evaluation      point;
        geoIndex        [label]; // default 0
        cellIndex       [label];

        evaluation      patchFieldAverage;
        patchIndex      [label];

        evaluation      patchFieldMaximum;
        patchIndex      [label];

        evaluation      patchFieldMinimum;
        patchIndex      [label];

        evaluation      internalFieldAverage;

        evaluation      internalFieldMaximum;

        evaluation      internalFieldMinimum;
    }
```

The `volScalarFieldLimit` input is very similar to the `equationLimit` input. Refer to the `equationLimit` input for more details.

J.4.5 Logic

The logic has the following syntax:

```
logic
{
    order   ([testName] [testName] [testName] ...); // optional

    [testName]
    {
        solverDomain    [solverDomainName];
        triggered       ( [triggerName] [triggerName] ... ); // optional
        notTriggered    ( [triggerName] [triggerName] ... ); // optional
    }

    [testName]
    ...
}
```

- The logic is a set of tests: you specify which inputs must be true (`triggered`) and which must be false (`notTriggered`).
- If these conditions are met, **plcEmulator** switches to the `solverDomain` you specify.
- If the returned `solverDomain` is the same as the current one, the simulation continues.
- The input lists `triggered` and `notTriggered` may be empty (or entirely absent).

- **plcEmulator** cycles through the tests one at a time, and stops the instant a valid solution is found.
- The tests cannot be guaranteed to happen in the order you wrote them in your input file. If order matters, use the `order` keyword.
- When using the `order` keyword, *all* your logic tests must be included in the list.

This constitutes a form of programming. It may take some time to get the feel for this "language". For instance, if you leave "`triggered`" and "`notTriggered`" both empty, the test will always be true. You can program it by absolute logic: every possible permutation of inputs (on / off) is represented in your logic tree. Or, you can program it by relative logic: only the conditions that cause the state to change are represented. If you use relative logic, you have to specify the initial state using the `multiControlDict` (`startSolverDomain`).

J.5 Programming

Figure A7 shows the intended use of **plcEmulator** within a solver application. Since **plcEmulator** involves timers, occasionally it must modify the end time, or timestep. That is the purpose of the "adjust time".

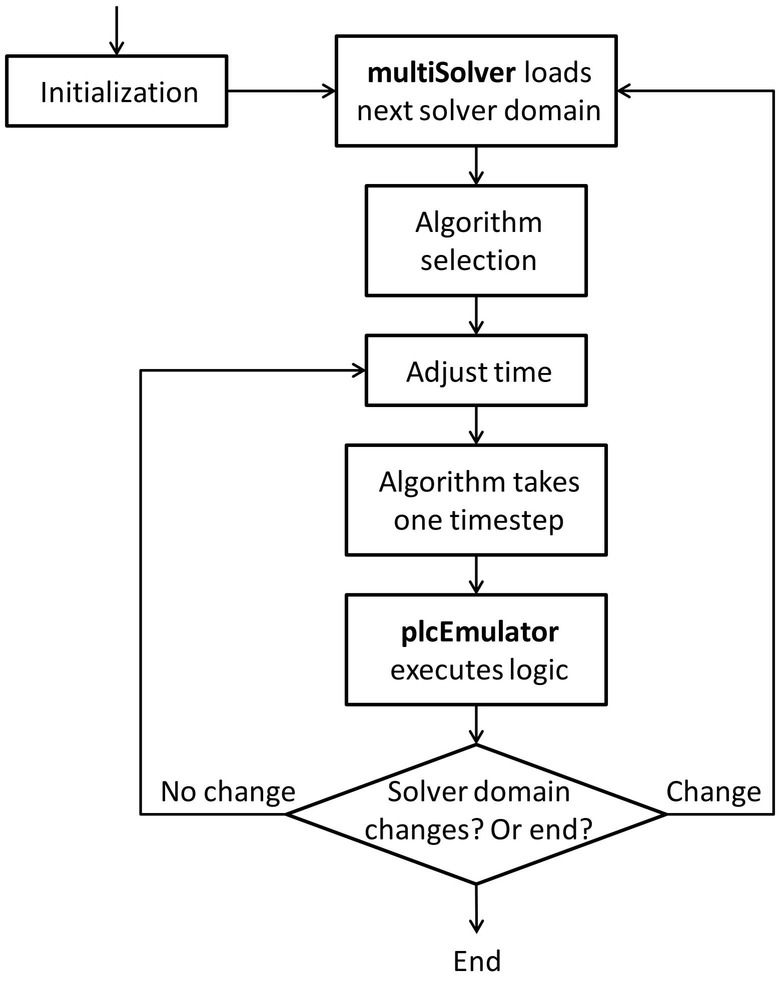

Initialization
multiSolver loads next solver domain
Algorithm selection
Adjust time
Algorithm takes one timestep
plcEmulator executes logic
Solver domain changes? Or end?
No change
Change
End

Figure A7: plcEmulator flowchart

A skeleton template for **plcEmulator** main application is shown below. The implementation is a little rough around the edges. For example, it requires a `bugWorkAround` Boolean.

```
#include "argList.H"
#include "multiSolver.H"
#include "plcController.H"

using namespace Foam;

// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

int main(int argc, char *argv[])
{
#   include "setRootCase.H"

    // Create multiSolver object
    multiSolver multiRun
    (
        multiSolver::multiControlDictName,
        args.rootPath(),
        args.caseName()
    );

    word solverDomain;
    word algorithm; // only required if using algorithm-switching

    // Find initial solver domain
    timeCluster tcSource(multiRun.initialDataSource());

    solverDomain = tcSource.solverDomainName();
    if (solverDomain == "default")
    {
        solverDomain = multiRun.startDomain();
        if (solverDomain == "default")
        {
            FatalIOErrorIn("main", multiRun.multiControlDict())
                << "Cannot determine start solverDomain. Use keyword "
                << "'startDomain' or change 'initialStartFrom'."
                << exit(FatalIOError);
        }
    }

    // Initialize solver domain
    multiRun.setSolverDomain(solverDomain);

    // Apparently I can't use setSolverDomain twice without running the solver
    bool bugWorkAround(true);

    // Create controller, reads current solverDomain and triggers
    plcController control(multiRun);

    // Main loop (superLoop)
    while (multiRun.run())
    {
        algorithm = control.currentAlgorithmName(); // optional
        solverDomain = control.currentSolverDomainName();

        if (!bugWorkAround)
        {
            multiRun.setSolverDomain(solverDomain);
        }
        bugWorkAround = false;

        if (multiRun.run())
        {
            Info << "Switching to solverDomain " << solverDomain
                << " using " << algorithm << " ***\n" << endl; // optional

            // If not algorithm-switching, go straight into the solver body
            // from here; otherwise, use a conditional to start the right one

            if (algorithm == "steadyStateSolver")
            {
#               include "steadyStateSolver.H"
            }
            else if (algorithm == "transientSolver")
            {
#               include "transientSolver.H"
            }
            else
            {
                FatalErrorIn("main")
                    << "Unrecognized algorithm, '" << algorithm << "'.  "
                    << "Expecting 'steadyStateSolver' or 'transientSolver'."
                    << abort(FatalError);
            }
            multiRun++;
        }
    }

    Info<< "End\n" << endl;
    return(0);
}
```

The skeleton template for each algorithm is:

```
#   include "createTime.H"
#   include "createMesh.H"

    // * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

    // Allow control to adjust the timestep if necessary, also, checks for
    // changes in solverDomain - this will intercept when a run is resumed
    // and switch to the correct solverDomain.
    control.initializeAndAdjustTime(runTime);

    while (runTime.run())
    {

        // Paste the main solver loop code here

        // To demonstrate solverSignal triggers, not required
        if ( /* some conditional */ 0)
        {
            control.signal("steadyStateAchieved", true);
        }

        // control.updateAndWrite does this:
        //  * update all triggers
        //  * apply logic
        //      * if solverDomain changes, model.time().writeAndEnd();
        //      * if not, model.time().write();
        //  * check for new timers that may interrupt end time
        //      * adjust end time if necessary
        control.updateAndWrite(runTime);
        runTime++;
    }

    // Reset control signal - not required
    control.signal("steadyStateAchieved", false);
```

Refer also to the demo application included in the release.

Appendix K CRAFTS input files

This section details the use of objects specific to CRAFTS.

K.1 Input file syntax overview

This is a key for the syntax of the input files.

- Anything in square brackets means you must supply the value.
- Anything not in square brackets is verbatim:

```
variableName    [variableName];

// For example:

variableName    rho;
variableName    S_ac;
```

- `[word]` is a word. It is allowed to have alphanumeric characters as well as round parentheses (), bar |, and underscore _;
- `[label]` is an integer:

```
chainRuleSearchDepth    [label];

// For example:

chainRuleSearchDepth    3;
```

- `[scalar]` can be any floating point number:

```
convergence     [scalar];

// For example:

convergence     10e-6;
convergence     3.68e-4;
convergence     10;
```

- `[symmtensor]` is a space-delimited list of six scalars, all of which are enclosed by rounded parentheses. They correspond to the XX, XY, XZ, YZ and ZZ components of a symmetric tensor.
- `[dimensionSet]` is a space-delimited list of seven `[scalar]`, all of which are enclosed in a single set of square brackets. They correspond to the powers applied to the dimensions: [Mass

Length Time Temperature Moles Current LuminosityIntensity], with standard SI units: [kg m s K mol A cd]:

```
dimensions      [dimensionSet];

// For example:

dimensions      [1 -3 0 0 0 0 0]; // density, kg/m3
dimensions      [1 -1 -2 0 0 0 0]; // pressure, kg/ms2
```

- `[dimensionedScalar]` is `[word] [dimensionSet] [scalar]`:

```
diffusion       [dimensionedScalar];

// For example:

diffusion       gamma [0 2 -1 0 0 0 0] 1.2e-4;
```

- `[equation]` is a string whose content is an expression. It may also be preceeded with a dimensionSet to disable dimension checking in the equation, and force the outcome to the given dimensions. You can also have a word before the dimensionSet, which it ignores:

```
function    [equation];

// For example:

function    "2 * rho / U";                        // equation
function    [dimensionSet] "2 * rho / U"          // dimensioned equation
function    banana [dimensionSet] "2 * rho / U"  // dimensioned equation and ignored word
```

- `[dimensionedScalarOrEquation]` is either:
  - `[scalar]`;
  - `[dimensionSet] [scalar]`;
  - `[dimensionedScalar]`; or
  - `[equation]`:

```
value   [dimensionedScalarOrEquation];

// In other words:

value   [scalar];
value   [dimensionSet] [scalar];
value   [dimensionedScalar];
value   [equation];

// For example:

value   2.4;
value   "2 * rho / U";
value   [0 0 0 0 0 0 0] 2.4;
value   [0 2 -1 -2 0 0 0] "2 * rho / U";
value   banana [0 0 0 0 0 0 0] 2.4;
value   banana [0 2 -1 -2 0 0 0] "2 * rho / U";
```

- `[scalarOrEquation]` accepts all formats of `[dimensionedScalarOrEquation]` except the resulting dimensions (calculated or assigned) must be dimensionless.
- `[varName]` is the name of a variable defined in `admVariableDict`.
- `[coefficient]` can be either the name of a coefficient already defined in `admCoefficientDict`, or a `[dimensionedScalarOrEquation]`.

K.2 Global settings

Global settings can be found in two places: the `controlDict` file (or `multiControlDict` if the solver uses **multiSolver**) and the `admSettingsDict` file.

K.2.1 controlDict

The `controlDict` is a file used by all OpenFOAM solvers. ADM-MDA adds a few extra settings to the `controlDict` file. These are:

```
{
    outputEquations              [yes/no];
    outputEquationDataSources    [yes/no];
    minDeltaT                    [scalar]; (default 0)
    maxDeltaT                    [scalar]; (default VGREAT)
    resumeDeltaT                 [yes/no]; (default no)
}
```

Where:

- **outputEquations** is a switch that enables output from the equation reader. An equations file will appear in the time directory;
- **outputEquationDataSources** adds detail to the equations file;
- **minDeltaT** is the minimum time step permitted (with adaptive time step control);
- **maxDeltaT** is the maximum time step permitted; and
- **resumeDeltaT** allows the solver to continue an interrupted run, using the same deltaT value it had previously been using. There is a bug with this setting, if **multiSolver** is being used (e.g.

admmda), this will set the deltaT value to 0 and cause the program to crash. If **multiSolver** isn't being used (e.g. admFullCoupledSolver), it works properly.

K.2.2 admSettingsDict

The admSettingsDict is found in the constant directory. It contains:

```
chainRuleSearchDepth        [label];
functionHooks               [functionHooksName];
flowModel
{
    type        [flowModelName];
    subStepping
    {
       // The entries differ depending on substepping type - choose only 1
        type            off;

        type            fixedNSteps;
        nSteps          [label];  (must be even)

        type            fixedTimestep;
        maxDeltaT       [scalar]; (default VGREAT)
        targetDeltaT    [scalar]; (default maxDeltaT/2 + minDeltaT/2)
        minDeltaT       [scalar]; (default VSMALL)
        maxNSteps       [label];  (must be even, default 2000000000)
        minNSteps       [label];  (must be even, default 2)

        type            adaptiveTimestep;
        maxDeltaT       [scalar]; (default VGREAT)
        minDeltaT       [scalar]; (default VSMALL)
        maxNSteps       [label];  (must be even, default 2000000000)
        minNSteps       [label];  (must be even, default 2)
        initialNSteps   [label];  (must be even, default 2)
    }
    adaptiveTimestepping
    {
        enableWithFlowModel     [yes/no]; (must be yes if req'd by subStepping)
        ignore ( phi p R );     (include fields to ignore)
        tolerance
        {
            phi     [scalar];     (if required)
            p       [scalar];     (if required)
            R       [symmTensor]; (if required)
        }
        minErrorScale
        {
            phi     [scalar];     (default SMALL)
            p       [scalar];     (default SMALL)
            R       [symmTensor]; (default SMALL symmtensor)
        }
    }
}
outerLoopMaxIterations      [label];
innerLoopMaxIterations      [label];
adaptiveTimeStepping
{
    useAdaptiveTimeStepping [yes/no];
    convergenceFactor       [scalar]; (required if useAdaptiveTimeStepping)
    overClockFactor         [scalar]; (optional - default = 1.0)
    maxIncreaseFactor       [scalar]; (required if useAdaptiveTimeStepping)
    maxReductionFactor      [scalar]; (required if useAdaptiveTimeStepping)
    minReductionFactor      [scalar]; (required if useAdaptiveTimeStepping)
    performanceFeedback
    {
        usePerformanceFeedback  [yes/no]; (required if useAdaptiveTimeStepping)
        measure                  [iterations/cpuTime]; (required if usePerformanceFeedBack)
        bias                     [scalar];             (required if usePerformanceFeedBack)
    }
}
outputFlags
{
    timeDetails                     [yes/no]; (default = no)

    reactionSummary                 [yes/no]; (default = yes)
    reactionAverages                [yes/no]; (default = yes)
    reactionSolverPerformance       [yes/no]; (default = no)
    reactionResidualSummary         [yes/no]; (default = yes)
    reactionResidualDetails         [yes/no]; (default = no)
    reactionErrorScales             [yes/no]; (default = no)
    reactionTimestepEstimate        [yes/no]; (default = yes)

    implicitLoopSummary             [yes/no]; (default = no)
    implicitLoopDetails             [yes/no]; (default = no)
    implicitAutoSolvePerformance    [yes/no]; (default = no)

    functionHooksSummary            [yes/no]; (default = no)
    ionSolverPerformance            [yes/no]; (default = no)
    gasSolverPerformance            [yes/no]; (default = yes)

    flowSolverPerformance           [yes/no]; (default = no)
    flowResiduals                   [yes/no]; (default = yes)
    flowErrorScales                 [yes/no]; (default = no)
    flowTimestepEstimate            [yes/no]; (default = yes)
    flowContinuityErrors            [yes/no]; (default = yes)
    flowSubStepProgress             [yes/no]; (default = yes)
}
```

Where:

- **chainRuleSearchDepth**: this is the number of derived variables to search through when calculating the derivative;
- **functionHooks** is the runTimeSelected function hooks to use - these are user-created functions that perform actions:
    - at the start of a timestep (`initializeTimestep`);
    - after the solution of the coupled reaction model (`applyVariableLimits`);
    - at the start of the implicit loop (`initializeImplicitLoop`);
    - during the implicit loop (`implicitLoop`);
    - at the end of the implicit loop (`finalizeImplicitLoop`);
    - at the end of the timestep (`finalizeTimestep`); and
- **flowModel** is a sub-dictionary detailing the flow model:

- **type** is the name of the flow model, currently either `steadyState`; `simple`; or `pimple`;
- **subStepping** is a sub-dictionary detailing the way the flow model handles substepping:
  - **type** is one of four different options:
    - `off` causes substepping to be disabled;
    - `fixedNSteps` causes the flow solver to take the same number of substeps at every timestep;
    - `fixedTimeStep` causes the flow solver to adjust the number of substeps in order to keep its timestep approximately constant, regardless of the timestep of the reaction solver;
    - `adaptiveTimestep` is useful when the flow variables are included in the adaptive timestepping calculations. Should the flow variables be the bottleneck (i.e. the flow model requires the smallest timescale), then the `adaptiveTimestep` setting causes it to increase the number of substeps it takes in order to keep the reaction solver at its desired timestep.
  - `nSteps` is the number of substeps it will used, which must be an even number;
  - `maxDeltaT` is the largest timestep the flow solver will allow before throwing an error and halting the program;
  - `targetDeltaT` is the desired timestep that the flow solver will try to reach by adjusting the number of substeps;
  - `minDeltaT` is the minimum timestep the flow solver will allow before throwing an error and halting the program;

        - `maxNSteps` is the largest number of substeps the flow solver is allowed to use; and
        - `minNSteps` is the smallest number of substeps the flow solver is allowed to use.
    - **adaptiveTimeStepping** is a sub-dictionary that describes which flow variables, if any, should be used for the timestep doubling adaptive timestepping:
        - **enableWithFlowModel** is whether or not to include the flow model in the adaptive timestepping algorithm;
        - **ignore** is a list of flow model variable names that should not be included in the calculations;
        - **tolerance** is the convergence criteria, listed by variable; and
        - **minErrorScale** is the minimum scaling factor to use, should the average value of the flow variable be too close to zero;
- **outerLoopMaxIterations** is the maximum number of iterations to attempt for the outer loop;
- **innerLoopMaxIterations** is the maximum number of iterations to attempt for the inner loop, also known as the implicit loop;
- **adaptiveTimeStepping** is a sub-dictionary that describes the adaptive timestepping behaviour of CRAFTS:
    - **useAdaptiveTimeStepping**: if you set this to `no`, the solver will halt if it determines it requires adaptiveTimeStepping;
    - **convergenceFactor**: you may want the adaptive timestepping convergence criteria to differ from the solver convergence criteria. A tighter criteria leads to smaller timesteps. The criteria used by the adaptive timestepping routine is the solver's convergence criteria multiplied by this factor;

- **overClockFactor**: the adaptive timestepping routine makes a "guess" at what the next timestep *should* be. If you think it is giving you excessively small (or large) timesteps, use this factor. The timestep the solver winds up using is the adaptive timestepping routine's suggested timestep multiplied by the overClockFactor. Ultimately, the adaptive timestepping routine tests convergence, and if you force it to go too big, it will adjust.
- **maxIncreaseFactor**: this is the maximum amount that the adaptive timestepping routine can increase the timestep. Must be greater than 1. For example, 2.0 means the timestep can double at most;
- **maxReductionFactor**: this is the most that the adaptive timestepping routine is allowed to reduced the timestep by. Must be less than 1. For example, 0.5 means the timestep can be cut in half at the most;
- **minReductionFactor**: if a timestep fails, the adaptive timestepping routine will calculate a new "optimum" timestep, but sometimes it may choose a value very close to the one it just attempted. The minReductionFactor forces it to reduce the timestep by a minimum amount. It only applies when the timestep fails. Must be less than 1. For example, a value of 0.7 means: if a timestep fails, the next timestep can be, at most, 70% of the previous timestep;
- **performanceFeedback**: up until now, adaptive timestepping is based on the error level, not on the performance of the solver. But, some numerical systems perform excessively slowly with larger timesteps. Therefore, performance feedback allows the solver to also be aware of how long the steps are taking, and to reduce the timestep should larger values be slowing it down excessively. Settings include:
    - **usePerformanceFeedBack**: self-explanatory;

        - **measure** is the performance measure and can either be `iterations`, which is a poor measure, but necessary if each timestep is too fast to register on the CPU clock; and `cpuTime`, which is more accurate; and
        - **bias** is a factor that determines how aggressively it slows down the timestep;
- **outputFlags** is a sub-dictionary that controls the information that the solver reports to the console. Each posted line is preceded by the name of the output flag, so if you wish to suppress some lines, you know the source of it. Flags include:
    - **timeDetails** will announce any changes in timestep, increments in the time value, and save / load states;
    - **reactionSummary** posts at the end of an iteration in the reaction solver, reporting the timestep, delta t, iteration number, highest residual, and implicit loop result;
    - **reactionAverages** does not actually post anything to the console; rather it creates files in the output directories with the volumetric averages for all the variables defined in `constant/admVariableDict`;
    - **reactionSolverPerformance** prints the performance details of the coupled block matrix solver after each solution;
    - **reactionResidualSummary** provides the name and values for the worst residuals among the reaction variables at each timestep;
    - **reactionResidualDetails** provides a list of all the residuals of all variables at each timestep, and before the first timestep lists the names of the variables;
    - **reactionErrorScales** provides the residual scaling factors for each variable at each timestep;

- **reactionTimestepEstimate** gives the variable name and timestep requirements of the bottleneck variables (i.e. those requiring the smallest time step) for the reactionvariables at each timestep;
- **implicitLoopSummary** announces whether the implicit loop failed or succeeded and the number of iterations it required;
- **implicitLoopDetails** reports the residuals of the implicit variables at each inner loop;
- **implicitAutoSolvePerformance** gives the iterations and convergence details for each variable solved by the implicit auto-solver;
- **functionHooksSummary** is available to users who write user-defined functions (UDFs), and for the existing UDFs, reports success / failure and which phase of the UDF caused the failure;
- **ionSolverPerformance** gives the final convergence and iterations for the ion solver, applicable only to UDFs that have an ion solver;
- **gasSolverPerformance** gives the number of iterations, mass residual, and delta residual details for each iteration of the gas solver, applicable only to UDFs that use the gas model;
- **flowSolverPerformance** reports the performance details of the matrix solver for the flow solver;
- **flowResiduals** reports the residuals of the flow model variables;
- **flowErrorScales** gives the error scaling factors for the flow model variables;
- **flowTimestepEstimate** gives the timestep sizes required by each flow model variable at each timestep;
- **flowContinuityErrors** gives the details of the continuity errors in the flow field after the flow solver iterates, and provides the Courant number; and

- **flowSubStepProgress** gives a count of how many substeps are completed out of how many in total, for each flow model iteration.

- **defaultStandardConvergence**: you can specify the convergence criterion for each standard variable individually in `admVariableDict`. Those that don't have a convergence criterion defined get this default value. If you omit this, every variable must have a convergence criterion defined;
- **defaultImplicitConvergence**: similar to defaultStandardConvergence;

K.3 Variables

There are 3 kinds of variables in CRCAFTS:

- standard;
- implicit; and
- derived.

Variables are defined in two places: the `admVariableDict`, and the initial conditions directory. The `admVariableDict` has the syntax:

```
defaults // optional subdictionary, all settings within are optional
{
    standard
    {
        convergence     [scalar]; // if missing, must be set individually
        minErrorScale   [scalar]; // default SMALL
    }
    implicit
    {
        convergence     [scalar]; // if missing, must be set individually
        minErrorScale   [scalar]; // default SMALL
    }
}

[variableName1]
{
    // variable sub-dictionary
}

[variableName2]
{
    // variable sub-dictionary
}

[variableName3]... etc.
```

Where the variable sub-dictionaries have the syntax:

```
[variableName]
{
    // The entries differ depending on variable type - choose only 1
    type            standard;
    diffusion       [dimensionedScalar];
    convergence     [scalar]; // optional if default exists
    minErrorScale   [scalar]; // optional if default exists
    changedByUdf    [yes/no]; // default no
    upperLimit      [scalar]; // default VGREAT
    lowerLimit      [scalar]; // default 0
    // other dictionary entries can be entered here and looked up
    // independently using varName.lookup(keyword, entry);

    type            implicit;
    diffusion       [dimensionedScalar];
    convergence     [scalar]; // optional if default exists
    minErrorScale   [scalar]; // optional if default exists
    upperLimit      [scalar]; // default VGREAT
    lowerLimit      [scalar]; // default 0
    autoSolve               [yes/no]; // default no
    autoSolveConvergence    [scalar]; // default convergence (above)
    autoSolveMaxIter        [label]; // default 10,000
    // other dictionary entries can be entered here and looked up
    // independently using varName.lookup(keyword, entry);

    type            derived;
    suppressOutput  [yes/no]; //default no
    function        [equation];
    ddt             [equation];
    reverseFunctions // this subdictionary is optional
    {
        order       ( [varName] [varName] ... ); // optional
        varName     [equation];
        varName     [equation];
        ...
    }
    Jacobian // this subdictionary is optional
    {
        varName     [equation];
        varName     [equation];
        ...
    }
    // other dictionary entries can be entered here and looked up
    // independently using varName.lookup(keyword, entry);
}
```

Where:

- **defaults** is an optional sub-dictionary that describes the default convergence criteria and scaling factors for **standard** and **implicit** variables. If no default is defined, or if this sub-dictionary is missing, then these convergence and error scale details must be included in each variable sub-dictionary:
    - **convergence** is the default convergence criterion used by the solver and adaptive timestep calculations;

    - **minErrorScale**: the errors are scaled by the volumetric average of each variable, but if the average is too close to zero this causes numerical problems, therefore the minErrorScale allows you to limit the minimum value it can attain;

And in the variable sub-dictionary:

- **type** is the variable type, which can be:
    - **standard**;
    - **implicit**; or
    - **derived**; and
- **diffusion** is the diffusion constant (often denoted Γ) for the variable;
- **convergence** is the convergence criterion. This is optional if you have defined a default convergence in `admSettingsDict`;
- **minErrorScale** is the minimum error scaling factor, as described in the defaults section above;
- **changedByUdf**: if the *implicitLoop* changes the value of a *standard* variable, set this to `yes`. It enables source term stabilisation in the coupled reaction model;
- **upperLimit** is the upper limit restriction placed on *standard* or *implicit* variables;
- **lowerLimit** is the lower limit restriction placed on *standard* or *implicit* variables;
- **autoSolve** is for *implicit* variables, and triggers the generic Newton-Raphson solver based on static equilibrium;
- **autoSolveConvergence** is the convergence criterion for the *autoSolve* routine on this variable;
- **autoSolveMaxIter** is the maximum number of iterations for the *autoSolve* routine on this variable;
- **suppressOutput** is for *derived* variables - if enabled, the variable is not written to file at each output;
- **function** is the equation that defines a derived variable;

- **ddt** is the time derivative of the function;
- **reverseFunctions** is `function` solved for various variables, only used if the algorithm needs to set the value of the *derived* variable;
- **Jacobian** contains a list of the non-zero derivatives of function;

K.4 Coefficients

Coefficients are numerical values that affect the behaviour of the model. Not only does this include all constants, it also allows some coefficients to vary spatially and temporally. For example, the acid-base equilibrium coefficient for water K_w varies with temperature. Coefficients do not include:

- ρ_j - reaction rates;
- I - inhibition multipliers that act on reaction rates; or
- v_{ij} - yields in the Petersen matrix.

These are special cases and are handled separately.

There five types of coefficients are:

- zero coefficients;
- one coefficients;
- constant coefficients;
- temperature dependent coefficients (based on an exponential ratio formula); and
- custom.

Coefficients are defined in the `admCoefficientDict` file in the case directory.

K.4.1 Zero coefficients

These coefficients always evaluate to zero. They are more useful internally so that functions that return a coefficient can return a coefficient that is guaranteed to be zero.

K.4.1.1 Syntax

They are specified in the `admCoefficientDict` file according to:

```
coefficientName
{
     type        zero;
     dimensions  [dimensionSet]; // default dimensionless
}
```

K.4.2 One coefficients

Similar to zero coefficients, except with a value of unity.

K.4.2.1 Syntax

They are specified in the `admCoefficientDict` file according to:

```
coefficientName
{
     type        one;
     dimensions  [dimensionSet]; // default dimensionless
}
```

K.4.3 Constant coefficients

Constant coefficients are invariant both spatially and temporally.

K.4.3.1 Theory

These are true constants. To conserve memory, ADM-MDA stores these as a single `dimensionedScalar`.

K.4.3.2 Syntax

They are specified in the `admCoefficientDict` file according to:

```
[coefficientName]  [dimensionedScalarOrEquation];
```

There is a longer format available, but it is not necessary:

```
[coefficientName]
{
     type    constant;
     value    [dimensionedScalarOrEquation];
}
```

K.4.4 Temperature dependent coefficients

The **temperature dependent exponential ratio coefficient** uses a temperature dependency that is common to several coefficients in anaerobic digesters.

K.4.4.1 Theory

A **temperature dependent exponential ratio coefficient** is given by:

$$C = A \exp\left(\frac{B}{T_{base}} - \frac{B}{T}\right), \tag{A44}$$

where:

- *A* and *B* are numerical constants; and
- T_{base} is the temperature base value.

Note: Temperature is assumed not to be a standard variable. If this is not the case, using this type of coefficient will erroneously give:

$$\frac{\partial C}{\partial T} = 0. \tag{A45}$$

Use a **custom coefficient** instead.

K.4.4.2 Syntax

A **temperature dependent exponential ratio coefficient** is specified in the `admCoefficientsDict` file by:

```
[coefficientName]
{
    type         temperatureDependentExponentialRatio;
    A            [scalarOrEquation];
    B            [scalarOrEquation];
    T_base       [scalarOrEquation];
    T_var        [variableName];
    dimensions   [dimensionSet]
}
```

Where:

- **A**, **B**, and **T_base** are coefficients; and

- dimensions are the dimensions of the coefficient (if omitted it is assumed dimensionless).

K.4.5 Custom coefficient

A **custom coefficient** can exhibit any kind of variable dependency. For instance, a **custom coefficient** makes inhibition through yield reduction or increased biological decay rate possible.

K.4.5.1 Theory

Custom classes use the **equationReader**.

K.4.5.2 Syntax

A **custom coefficient** is specified in the `admCoefficientsDict` file by:

```
[coefficientName]
{
    type        custom;
    uniform     [yes/no]; // default no
    function    [dimensionedScalarOrEquation];
    ddt         [dimensionedScalarOrEquation];
    Jacobian
    {
        [variableName]  [dimensionedScalarOrEquation];
        [variableName]  [dimensionedScalarOrEquation];
        ...
    }
}
```

Where:

- **uniform** - when set to no the coefficient will allow for spatial variation;
- **function** is the function describing the variable dependence of the coefficient;
- **ddt** - if the coefficient depends directly on time, include its time derivative here (if not, omit this entry); and
- **Jacobian** - if the coefficient depends on one or more standard variables, put its derivative with respect to each (non-zero) one here. Again, omit if it does not. The customCoefficient does not use the chain rule to search through other variables to find non-zero derivatives. Therefore, put all the desired non-zero derivatives here.

K.5 Reactions

A single reaction in ADM-MDA represents a row in the Petersen matrix. A reaction consists of:

- a reaction rate (ρ);
- a set of yields; and
- a set of reaction rate inhibitions.

K.5.1 Syntax

Reactions are defined in the `admReactionDict` according to:

```
reactionName
{
    rate
    {
        reactionRate details ...
    }
    rateInhibitions (inhibitionName, inhibitionName, ...);
    yields
    {
        variableName    "equation";
    }
}
```

where:

- **reactionRate details** are the entries for a reaction rate;
- **rateInhibitions** (optional) if there are inhibitions for this reaction, list them by name – see section;
- **yields** is a list of variables and their yield – these are the values in the Petersen matrix (v_{ij}).

K.6 Reaction rates

Reaction rates are the rates at which the reaction occur. They are the ρ value in the rightmost column of the Petersen matrix. The eight kinds of reaction rates are:

- non-reacting reaction rate;
- first-order reaction rate;
- simple gas reaction rate;
- acid-base reaction rate;

- monod reaction rate;
- competitive monod reaction rate;
- uncompetitive monod reaction rate; and
- custom reaction rate.

Reaction rates are defined in the `admReactionDict` file.

Note: All the derivatives expressed in this section assume coefficients have zero derivatives. The code now takes derivatives of the coefficients as well. Therefore the derivatives in the code are more complex than those shown below.

K.6.1 Non-reacting reaction rate

The **non-reacting reaction** rate sets the rate to zero. This is a quick way to turn-off reactions in a simulation.

K.6.1.1 Theory

A non-reacting reaction rate is given by:

$$\rho = 0. \tag{A46}$$

K.6.1.2 Syntax

A **non-reacting reaction rate** can be specified in the `admReactionDict` file by:

```
[reactionName]
{
    rate
    {
        type        nonReacting;
    }
    yields
    {
        ...
    }
}
```

K.6.2 First-order reaction rate

The **first-order reaction rate** is a reaction rate that correlates with one independent variable.

K.6.2.1 Theory

A first-order reaction rate is given by:

$$\rho = kS_{var}, \tag{A47}$$

where:

- *k* is a coefficient; and
- S_{var} is the variable on which it depends.

The **first order reaction rate** has a non-zero derivative, with the non-zero term given by:

$$\frac{\partial \rho}{\partial S_{var}} = k. \tag{A48}$$

K.6.2.2 Syntax

A **first-order reaction rate** can be specified in the `admReactionDict` file by:

```
[reactionName]
{
    rate
    {
        type        firstOrder;
        k           [coefficient];
        var         [variableName];
    }
    yields
    {
        ...
    }
}
```

Where:

- *k* is a coefficient; and
- *var* is the variable on which the reaction rate depends.

K.6.3 Simple gas reaction rate

The **simple gas reaction rate** is an approximation reaction rate for simple gas models.

K.6.3.1 Theory

A **simple gas reaction rate** is given by:

$$\rho = k_L a \left(S_{var} - n \; K_H p_{var} \right) , \tag{A49}$$

where:

- $k_L a$ is the overall gas transfer coefficient;
- S_{var} is the dissolved component of the gas;
- n is the COD equivalent fraction of the gas;
- K_H is Henry's law coefficient; and
- p_{var} is the gas partial pressure.

The **simple gas reaction rate** has a non-zero derivative, with non-zero terms given by:

$$\frac{\partial \rho}{\partial S_{var}} = k_L a, \text{ and} \tag{A50}$$

$$\frac{\partial \rho}{\partial p_{var}} = -k_L a \; n \; K_H . \tag{A51}$$

K.6.3.2 Syntax

A **simple gas reaction rate** can be specified in the `admReactionDict` file by:

```
[reactionName]
{
    rate
    {
        type         simpleGas;
        k_L_a        [coefficient];
        n            [coefficient];
        K_H          [coefficient];
        S_var        [variableName];
        p_var        [variableName];
    }
    yields
    {
        ...
    }
}
```

Where:

- **k_L_a**, **n**, and **K_H** are coefficients;
- **S_var** is the variable for the soluble dissolved component of the gas; and
- **p_var** is the variable for the gas partial pressure.

K.6.4 Acid-base reaction rate

The **acid-base reaction rate** is a reaction rate for acid-base equilibrium. There are two forms depending on which fundamental variables are being used:

- soluble ions (S_{var-}) and their associated acid (S_{hvar}); or
- soluble ions (S_{var-}) and the total concentration (S_{var}).

K.6.4.1 Theory

An **acid-base reaction rate** is given by:

Form 1

$$\rho = k_{AB}\left(S_{var^-}S_{H^+} - K_a S_{hvar}\right), \tag{A52}$$

or:

Form 2

$$\rho = k_{AB}\left[S_{var^-}\left(K_a + S_{H^+}\right) - K_a S_{var}\right], \tag{A53}$$

where:

- k_{AB} is the acid-base kinetic parameter;
- S_{var-} is the concentration of the compound's negative ion form;
- S_{H+} is the concentration of hydrogen ions;
- K_a is the acid-base equilibrium coefficient;
- S_{hvar} is the concentration of the compound's acid form; and
- S_{var} is the total concentration of the compound.

The **acid-base reaction rate** has a non-zero Jacobian, with the non-zero terms given by:

Form 1

$$\frac{\partial \rho}{\partial S_{var^-}} = k_{AB} S_{H^+}, \tag{A54}$$

$$\frac{\partial \rho}{\partial S_{hvar}} = k_{AB} K_a, \text{ and} \tag{A55}$$

$$\frac{\partial \rho}{\partial S_{H+}} = k_{AB} S_{var-}, \tag{A56}$$

or:

Form 2

$$\frac{\partial \rho}{\partial S_{var-}} = k_{AB} \left(K_a + S_{H+} \right), \tag{A57}$$

$$\frac{\partial \rho}{\partial S_{var}} = k_{AB} K_a, \text{ and} \tag{A58}$$

$$\frac{\partial \rho}{\partial S_{H+}} = k_{AB} S_{var-}. \tag{A59}$$

K.6.4.2 Syntax

An **acid-base reaction rate** can be specified in the `admReactionDict` file by:

```
[reactionName]
{
    rate
    {
        type        acidBase;
        k_AB        [coefficient];
        K_a         [coefficient];
        S_H_p       [variableName];
        S_var_m     [variableName];
        S_hvar      [variableName]; // * see note below
        S_var       [variableName]; // * see note below
    }
    yields
    {
        ...
    }
}
```

Where:

- **k_AB** and **K_a** are numerical constants;
- **S_H_p** is the variable for dissolved hydrogen ion concentration;
- **S_var_m** is the variable for dissolved negative ions;
- **S_hvar** is the variable for the acid concentration; and
- **S_var** is the total concentration (ions and acids included) of the compound in question.

***Note:** Either **S_hvar** or **S_var** must be chosen, not both. It is preferable to choose the one that is a standard variable if possible.

K.6.5 Monod reaction rate

The **monod reaction rate** is a reaction rate based on monod kinetics.

K.6.5.1 Theory

A **monod reaction rate** is given by:

$$\rho = \frac{k_m S_{var} X_{var}}{K_s + S_{var}}, \tag{A60}$$

where:

- k_m is the kinetic rate parameter;
- K_s is the half-saturation value;
- S_{var} is the soluble dissolved concentration of the compound; and
- X_{var} is the particulate concentration of the compound.

The monod reaction rate has a non-zero derivative, with non-zero terms given by:

$$\frac{\partial \rho}{\partial S_{var}} = \frac{K_s k_m X_{var}}{(K_s + S_{var})^2}, \text{ and} \tag{A61}$$

$$\frac{\partial \rho}{\partial X_{var}} = \frac{k_m S_{var}}{K_s + S_{var}}. \tag{A62}$$

K.6.5.2 Syntax

A monod reaction rate can be specified in the admReactionDict file by:

```
[reactionName]
{
    rate
    {
        type        monod;
        k_m         [coefficient];
        K_s         [coefficient];
        S_var       [variableName];
        X_var       [variableName];
    }
    yields
    {
        ...
        }
}
```

Where:

- **k_m** and **K_s** are numerical constants;
- **S_var** is the variable for the soluble dissolved concentration of the compound; and
- **X_var** is the variable for the particulate concentration of the compound.

K.6.6 Competitive monod reaction rate

Unlike other forms of inhibition, competitive inhibition changes the form of the reaction rate (ρ), and it cannot be modelled as an inhibition rate multiplier. Therefore the **competitive monod reaction rate** was created as a monod reaction rate with integrated competitive inhibition.

K.6.6.1 Theory

A **competitive monod reaction rate** is given by:

$$\rho = \frac{k_m K_I S_{var} X_{var}}{K_s \left(K_I + S_I\right) + S_{var} K_I}, \tag{A63}$$

where:

- k_m is the kinetic rate parameter;
- K_s is the half-saturation value of the compound;
- K_I is the half-saturation value of the inhibitor;
- S_{var} is the soluble dissolved concentration of the compound;
- S_I is the soluble dissolved concentration of the inhibitor; and
- X_{var} is the particulate concentration of the compound.

The competitive monod reaction rate has a non-zero derivative, with non-zero terms given by:

$$\frac{\partial \rho}{\partial S_{var}} = \frac{K_s K_I k_m X_{var} \left(K_I + S_I\right)}{\left[K_s \left(K_I + S_I\right) + S_{var} K_I\right]^2}, \tag{A64}$$

$$\frac{\partial \rho}{\partial X_{var}} = \frac{k_m K_I S_{var}}{K_s\left(K_I + S_I\right) + S_{var} K_I}, \text{ and} \quad (A65)$$

$$\frac{\partial \rho}{\partial S_I} = \frac{-k_m K_I K_s S_{var} X_{var}}{\left[K_s\left(K_I + S_I\right) + S_{var} K_I\right]^2}. \quad (A66)$$

K.6.6.2 Syntax

A **competitive monod reaction rate** can be specified in the `admReactionDict` file by:

```
[reactionName]
{
    rate
    {
        type         competitiveMonod;
        k_m          [coefficient];
        K_s          [coefficient];
        K_I          [coefficient];
        S_var        [variableName];
        S_I          [variableName];
        X_var        [variableName];
    }
    yields
    {
        ...
    }
}
```

Where:

- **k_m**, **K_s** and **K_I** are numerical constants;
- **S_var** is the variable for the soluble dissolved concentration of the compound;
- **S_I** is the inhibitor variable; and
- **X_var** is the variable for the particulate concentration of the compound.

K.6.7 Uncompetitive monod reaction rate

Similar to the competitive monod reaction rate, the **uncompetitive monod reaction rate** was created to model uncompetitive inhibition. It is a monod reaction rate combined with uncompetitive inhibition.

K.6.7.1 Theory

An **uncompetitive monod reaction rate** is given by:

$$\frac{k_m X_{var} S_{var} S_I}{K_s S_I + S_{var}\left(S_I + K_I\right)}, \quad (A67)$$

where:

- k_m is the kinetic rate parameter;
- K_s is the half-saturation value of the compound;
- K_I is the half-saturation value of the inhibitor;
- S_{var} is the soluble dissolved concentration of the compound;
- S_I is the soluble dissolved concentration of the inhibitor; and
- X_{var} is the particulate concentration of the compound.

The competitive monod reaction rate has a non-zero derivative, with a non-zero term givens by:

$$\frac{\partial \rho}{\partial S_{var}} = \frac{k_m K_s X_{var} S_I^2}{\left[K_s S_I + S_{var}\left(S_I + K_I\right)\right]^2}, \tag{A68}$$

$$\frac{\partial \rho}{\partial X_{var}} = \frac{k_m S_{var} S_I}{K_s S_I + S_{var}\left(S_I + K_I\right)}, \quad \text{and} \tag{A69}$$

$$\frac{\partial \rho}{\partial S_I} = \frac{k_m X_{var} K_I S_{var}^2}{\left[K_s S_I + S_{var}\left(S_I + K_I\right)\right]^2}. \tag{A70}$$

K.6.7.2 Syntax

An **uncompetitive monod reaction rate** can be specified in the `admReactionDict` file by:

```
[reactionName]
{
    rate
    {
        type        uncompetitiveMonod;
        k_m         [coefficient];
        K_s         [coefficient];
        K_I         [coefficient];
        S_var       [variableName];
        S_I         [variableName];
        X_var       [variableName];
    }
    yields
    {
        ...
    }
}
```

Where:

- **k_m**, **K_s** and **K_I** are numerical constants;
- **S_var** is the variable for the soluble dissolved concentration of the compound;

- **S_I** is the inhibitor variable; and
- **X_var** is the variable for the particulate concentration of the compound.

K.6.8 Custom reaction rate

A **custom reaction rate** can have any relation defined.

K.6.8.1 Theory

Custom classes use the **equationReader**.

K.6.8.2 Syntax

A **custom reaction rate** is specified in the `admReactionsDict` file by:

```
[reactionName]
{
    rate
    {
        type        custom;
        function    [dimensionedScalarOrequation];
        ddt         [dimensionedScalarOrequation];
        Jacobian
        {
            [variableName]  [dimensionedScalarOrequation];
            [variableName]  [dimensionedScalarOrequation];
            ...
        }
    }
    yields
    {
        ...
    }
}
```

Where:

- **function** is the function describing the variable dependence of the coefficient;
- **ddt** - if the coefficient depends directly on time, include its time derivative here (if not, omit this entry); and
- **Jacobian** - if the coefficient depends on one or more standard variables, put its derivative with respect to each (non-zero) one here. Again, omit if it does not.

K.7 Reaction rate inhibitions

Reaction rates inhibitions are inhibition multipliers that are applied to reaction rates, according to:

$$\rho_{eff} = \rho I, \tag{A71}$$

where:

- ρ_{eff} is the inhibited reaction rate;
- ρ is the uninhibited reaction rate; and
- I is the reaction rate inhibition.

The eight kinds of reaction rates inhibitions are:

- non-competitive inhibition;
- competitive uptake inhibition;
- secondary substrate inhibition;
- empirical upper and lower inhibition;
- empirical lower (switch function) inhibition;
- empirical lower (tanh function) inhibition;
- empirical lower (Hill function) inhibition; and
- custom inhibition.

Reaction rate inhibitions are defined in the `admInhibitionDict` file.

Note: All the derivatives expressed in this section assume coefficients have zero derivatives. The code now takes derivatives of the coefficients as well. Therefore the derivatives in the code are more complex than those shown below.

K.7.1 Non-competitive inhibition

Non-competitive inhibition models the situation in which a process is inhibited as the concentration of an inhibitor goes up. This form appears in ADM1 to model free ammonia and hydrogen inhibition.

K.7.1.1 Theory

Non-competitive inhibition is given by:

$$I = \frac{K_I}{K_I + S_I}, \tag{A72}$$

where:

- K_I is a coefficient; and
- S_I is the concentration of the inhibitor.

Non-competitive inhibition has a non-zero derivative, with non-zero terms given by:

$$\frac{\partial I}{\partial S_I} = -\frac{K_I}{\left(K_I + S_I\right)^2}. \tag{A73}$$

K.7.1.2 Syntax

Non-competitive inhibition is specified in the `admReactionsDict` by:

```
[inhibitionName]
{
    type     nonCompetitive;
    K_I      [coefficient];
    S_I      [variableName];
}
```

Where:

- **K_I** is the inhibition coefficient; and
- **S_I** is the inhibitor variable.

K.7.2 Competitive uptake inhibition

Competitive uptake inhibition is used to model the situation when two processes compete for the same micro-organisms. As the ratio of the inhibitor concentration to the process concentration increases, the inhibition effect increases.

K.7.2.1 Theory

Competitive uptake inhibition is given by:

$$I = \frac{S_{var}}{S_{var} + S_I}, \tag{A74}$$

where:

- S_{var} is the substrate for the inhibited process; and
- S_I is the inhibitor.

Competitive uptake inhibition has a non-zero derivative, with non-zero terms given by:

$$\frac{\partial I}{\partial S_{var}} = \frac{S_I}{\left(S_{var} + S_I\right)^2}, \text{ and} \tag{A75}$$

$$\frac{\partial I}{\partial S_I} = -\frac{S_{var}}{\left(S_{var} + S_I\right)^2}. \tag{A76}$$

K.7.2.2 Syntax

Competitive uptake inhibition is specified in the `admReactionsDict` by:

```
[inhibitionName]
{
    type        competitiveUptake;
    S_var       [variableName];
    S_I         [variableName];
}
```

Where:

- **S_var** is the variable for the substrate for the inhibited process; and
- **S_I** is the inhibitor variable.

K.7.3 Secondary substrate inhibition

Secondary substrate inhibition models the situation when the drop in an inhibitor leads to a corresponding drop in the inhibition. This form is used in ADM1 for modelling the drop in uptake processes as inorganic nitrogen concentrations drop.

K.7.3.1 Theory

Secondary substrate inhibition is given by:

$$I = \frac{S_I}{S_I + K_I}, \tag{A77}$$

where:

- S_I is the inhibitor; and
- K_I is a coefficient.

Secondary substrate inhibition has a non-zero derivative, with a non-zero term given by:

$$\frac{\partial I}{\partial S_I} = \frac{K_I}{(S_I + K_I)^2}. \tag{A78}$$

K.7.3.2 Syntax

Secondary substrate inhibition is specified in the `admReactionsDict` by:

```
[inhibitionName]
{
    type        secondarySubstrate;
    K_I         [coefficient];
    S_I         [variableName];
}
```

Where:

- **K_I** is a numerical constant; and
- **S_I** is the inhibitor variable.

K.7.4 Empirical upper and lower inhibition

Empirical upper and lower inhibition models the situation in which a process requires the pH to be within a narrow window. Deviations above and below this result in an increase in inhibition. ADM1 uses this to model pH inhibition on uptake processes. If free ammonia inhibition is used, it is recommended to use an empirical lower inhibition instead.

K.7.4.1 Theory

Empirical upper and lower inhibition is given by:

$$I = \frac{1 + 2 \times 10^{\frac{1}{2}(pH_{LL} - pH_{UL})}}{1 + 10^{(pH_{var} - pH_{UL})} + 10^{(pH_{LL} - pH_{var})}}, \tag{A79}$$

where:

- pH_{LL} is the lower pH limit;
- pH_{UL} is the upper pH limit; and
- pH_{var} is a variable.

Empirical upper and lower inhibition has a non-zero derivative, with non-zero terms given by:

$$\frac{\partial I}{\partial pH_{var}} = \frac{\ln(10)\left[10^{(pH_{LL} - pH_{var})} - 10^{(pH_{var} - pH_{UL})}\right]\left[1 + 2 \times 10^{\frac{1}{2}(pH_{LL} - pH_{UL})}\right]}{\left[1 + 10^{(pH_{var} - pH_{UL})} + 10^{(pH_{LL} - pH_{var})}\right]^2}. \tag{A80}$$

In terms of S_{H+}, it is given by:

$$\frac{\partial I}{\partial S_{H+}} = \frac{\partial I}{\partial pH_{var}} \frac{\partial pH_{var}}{\partial S_{H+}} = \frac{\partial I}{\partial pH_{var}} \left[\frac{-1000 \log_{10}(e)}{S_{H+}}\right], \tag{A81}$$

where e is Euler's number.

K.7.4.2 Syntax

Empirical upper and lower inhibition is specified in the `admReactionsDict` by:

```
[inhibitionName]
{
    type        empiricalUpperAndLower;
    pH_LL       [coefficient];
    pH_UL       [coefficient];
    pH_var      [variableName]; // * see note below
    S_Hp_var    [variableName]; // * see note below
}
```

Where:

- **pH_LL** and **pH_UL** are coefficients;
- **pH_var** is the **pH** variable; and
- **S_Hp_var** is the S_{H+} variable.

***Note** – Use only one of pH_var or S_Hp_var, not both. Choose the one that is a standard variable. If neither are standard type, just choose only one.

K.7.5 Empirical lower (switch function) inhibition

The **empirical lower inhibitions** model the situation when a process requires a minimum pH level to take place. A drop in pH results in a drop in process reaction rate. ADM1 uses this inhibition to model pH inhibition on uptake processes. There are several forms of this inhibition implemented, including the switch function. The switch function was the original form described by Batstone et al. (2002).

K.7.5.1 Theory

Empirical lower (switch function) inhibition is given by:

$$I = \begin{cases} \exp\left[-3\left(\frac{pH_{var}-pH_{UL}}{pH_{UL}-pH_{LL}}\right)^2\right] & \text{if } pH_{var} < pH_{UL} \\ 1 & \text{if } pH_{var} \geq pH_{UL} \end{cases}, \qquad \text{(A82)}$$

where:

- pH_{LL} is the lower pH limit;
- pH_{UL} is the upper pH limit; and
- pH_{var} is a variable.

Empirical lower (switch) inhibition has a non-zero derivative, with non-zero terms given by:

$$\frac{\partial I}{\partial pH_{var}} \begin{cases} \frac{6pH_{UL}(pH_{var}-pH_{UL})}{(pH_{UL}-pH_{LL})^2} \exp\left[-3\left(\frac{pH_{var}-pH_{UL}}{pH_{UL}-pH_{LL}}\right)^2\right] & \text{if } pH_{var} < pH_{UL} \\ 0 & \text{if } pH_{var} \geq pH_{UL} \end{cases}. \qquad \text{(A83)}$$

In terms of S_{H+}, it is given by:

$$\frac{\partial I}{\partial S_{H+}} = \frac{\partial I}{\partial pH_{var}} \frac{\partial pH_{var}}{\partial S_{H+}} = \frac{\partial I}{\partial pH_{var}} \left[\frac{-1000\log_{10}(e)}{S_{H+}}\right], \qquad \text{(A84)}$$

where e is Euler's number.

K.7.5.2 Syntax

Empirical lower (switch function) inhibition is specified in the `admReactionsDict` by:

```
[inhibitionName]
{
    type        empiricalLowerSwitch;
    pH_LL       [coefficient];
    pH_UL        [coefficient];
    pH_var       [variableName]; // *see note below
    S_Hp_var     [variableName]; // *see note below
}
```

Where:

- **pH_LL** and **pH_UL** are coefficients;
- **pH_var** is the **pH** variable; and
- **S_Hp_var** is the S_{H+} variable.

***Note** – Use only one of pH_var or S_Hp_var, not both. Choose the one that is a standard variable. If neither are standard type, just choose only one.

K.7.6 Empirical lower (tanh function) inhibition

The **empirical lower inhibitions** model the situation when a process requires a minimum pH level to take place. A drop in pH results in a drop in process reaction rate. ADM1 uses this inhibition to model pH inhibition on uptake processes. There are several forms of this inhibition implemented, including the tanh function. The tanh function was suggested as a modification to the original ADM1 model to improve numerical performance (Rosen et al., 2006).

K.7.6.1 Theory

Empirical lower (tanh function) inhibition is given by:

$$I = \frac{1}{2}\left[1 + \tanh\left(a\frac{pH_{var}}{pH_{avg}} - 1\right)\right], \tag{A85}$$

where:

- pH_{avg} is given by:

$$pH_{avg} = \frac{pH_{LL} + pH_{UL}}{2}, \tag{A86}$$

- a is a coefficient;
- pH_{LL} is the lower pH limit;
- pH_{UL} is the upper pH limit; and

- *pH_{var}* is a variable.

Empirical lower (tanh) inhibition has a non-zero derivative, with non-zero terms given by:

$$\frac{\partial I}{\partial pH_{var}} = \frac{a}{2pH_{avg}}\text{sech}^2\left(a\left(\frac{pH_{var}}{pH_{avg}} - 1\right)\right). \tag{A87}$$

In terms of S_{H+}, it is given by:

$$\frac{\partial I}{\partial S_{H+}} = \frac{\partial I}{\partial pH_{var}}\frac{\partial pH_{var}}{\partial S_{H+}} = \frac{\partial I}{\partial pH_{var}}\left[\frac{-1000\log_{10}(e)}{S_{H+}}\right], \tag{A88}$$

where *e* is Euler's number.

K.7.6.2 Syntax

Empirical lower (tanh function) inhibition is specified in the `admReactionsDict` by:

```
[inhibitionName]
{
    type        empiricalLowerTanh;
    a           [coefficient];
    pH_LL       [coefficient];
    pH_UL       [coefficient];
    pH_var      [variableName]; // *see note below
    S_Hp_var    [variableName]; // *see note below
}
```

Where:

- **a**, **pH_LL** and **pH_UL** are coefficients;
- **pH_var** is the **pH** variable; and
- **S_Hp_var** is the S_{H+} variable.

***Note** – Use only one of pH_var or S_Hp_var, not both. Choose the one that is a standard variable. If neither are standard type, just choose only one.

K.7.7 Empirical lower (Hill function) inhibition

The **empirical lower inhibitions** model the situation when a process requires a minimum pH level to take place. A drop in pH results in a drop in process reaction rate. ADM1 uses this inhibition to model pH inhibition on uptake processes. There are several forms of this inhibition implemented, including the

Hill function. The Hill function was suggested as a modification to the original ADM1 model to improve numerical performance (Rosen et al., 2006).

K.7.7.1 Theory

Empirical lower (Hill function) inhibition is given by:

$$I = \frac{pH_{var}^{n}}{pH_{var}^{n} + pH_{avg}^{n}}, \tag{A89}$$

where:

- pH_{avg} is given by:

$$pH_{avg} = \frac{pH_{LL} + pH_{UL}}{2}, \tag{A90}$$

- n is a coefficient;
- pH_{LL} is the lower pH limit;
- pH_{UL} is the upper pH limit; and
- pH_{var} is a variable.

Empirical lower (Hill function) inhibition has a non-zero derivative, with non-zero terms given by:

$$\frac{\partial I}{\partial pH_{var}} = \frac{n pH_{avg}^{n} pH_{var}^{(n-1)}}{\left(pH_{var}^{n} + pH_{avg}^{n}\right)^{2}}. \tag{A91}$$

In terms of S_{H+}, it is given by:

$$\frac{\partial I}{\partial S_{H+}} = \frac{\partial I}{\partial pH_{var}} \frac{\partial pH_{var}}{\partial S_{H+}} = \frac{\partial I}{\partial pH_{var}} \left[\frac{-1000 \log_{10}(e)}{S_{H+}}\right], \tag{A92}$$

where e is Euler's number.

K.7.7.2 Syntax

Empirical lower (Hill function) inhibition is specified in the `admReactionsDict` by:

```
[inhibitionName]
{
    type        empiricalLowerHill;
    n           [coefficient];
    pH_LL       [coefficient];
    pH_UL       [coefficient];
    pH_var      [variableName]; // *see note below
    S_Hp_var    [variableName]; // *see note below
}
```

Where:

- **n**, **pH_LL** and **pH_UL** are coefficients;
- **pH_var** is the **pH** variable; and
- **S_Hp_var** is the S_{H+} variable.

***Note** – Use only one of pH_var or S_Hp_var, not both. Choose the one that is a standard variable. If neither are standard type, just choose only one.

K.7.8 Custom inhibition

To model any other form of inhibition that acts as a multiplier on the reaction rate, **custom inhibition** is available. To model a form of inhibition that acts differently than a simple reaction rate multiplier, use a custom reaction rate or a custom yield coefficient.

K.7.8.1 Theory

Custom objects use the equation reader.

K.7.8.2 Syntax

Custom inhibition is specified in the `admReactionsDict` by:

```
[inhibitionName]
{
    type        custom;
    function    [dimensionedScalarOrEquation];
    Jacobian    // * see note, below
    {
        variableName     [dimensionedScalarOrEquation];
        variableName     [dimensionedScalarOrEquation];
        ...
    }
}
```

Where:

- **function**, is an equation that results in a dimensionless quantity;
- **Jacobian** is a sub-dictionary containing all the non-zero Jacobian functions.

***Note** – If none of the variables appearing in function are standard variables, omit the Jacobian entry.

****Note** – The model never requires a time derivative of a rate inhibition; therefore it is not necessary to add a ddt entry.

K.8 Gas model settings

Most settings for the gas model are in the `admSimpleGasDict` file. The syntax of this file is:

```
universal
{
    odeSolver                    [odeSolverName];
    odeSolverMaxIterations       [label];         // default 10000
    epsilon                      [scalar];
    defaultDeltaConvergence      [scalar];        // optional
    defaultMassConvergence       [scalar];        // optional
    defaultNearZeroDeltaScale    [scalar];        // default SMALL
    defaultNearZeroMassScale     [scalar];        // default SMALL
    outputDictionary             [word];          // default gasModel
    surfacePatch                 [boundaryName];
    gasVolume                    [coefficient];   // Must be constant & uniform
    R                            [coefficient];   // Must be constant & uniform
    T                            [variableName];
    k_p                          [coefficient];   // Must be constant & uniform
    kGasCutOff                   [coefficient];   // Must be constant & uniform
    p_atm                        [coefficient];   // Must be uniform
    p_liq                        [coefficient];
}

species
{
    [gasName]
    {
        liquidVariable       [variableName];
        conversionToMoles    [coefficient]; // Must be constant & uniform
        k_L_a                [coefficient]; // Must be uniform
        k_H                  [coefficient];
        k_b                  [coefficient];
        formBubbles          [yes/no];
        upperLimit           [scalar]; // default VGREAT
        lowerLimit           [scalar]; // default 0
        relaxation           [scalar]; // default 1.0
        deltaConvergence     [scalar]; // required if no default
        massConvergence      [scalar]; // required if no default
        nearZeroDeltaScale   [scalar]; // default defined above
        nearZeroMassScale    [scalar]; // default defined above
    }
    ...
}
```

K.8.1 Universal settings

- **odeSolver** is the name of the ODE solver algorithm, which can be any of:
    - Euler;
    - KRR4 (a fourth-order Kaps-Rentrop scheme);

  - RK (a fifth-order Cash-Karp embedded Runge-Kutta scheme); and
  - SIBS (a semi-implicit Bulirsch-Stoer method);

- **odeSolverMaxIterations** is the most iterations before the ODE solver fails and reduces the timestep;
- **epsilon** is a very small number used by the ODE solver, and is critical in determining the error size;
- **defaultDeltaConvergence**: the gas model compares the change in gas concentrations between iterations. This is the default convergence criterion for the comparison;
- **defaultMassConvergence**: the gas model compares the mass transfer out of the liquid volume (diffusion) with the mass transfer into the gas volume (Henry's law). This is the default convergence criterion for the comparison;
- **defaultNearZeroDeltaScale**: when the gas concentrations are close to zero, there is no suitable factor by which to scale the error. Therefore, this is the default scale applied in these circumstances;
- **defaultNearZeroMassScale**: same thing for the mass convergence test;
- **outputDictionary**: the gas model prints its data to a dictionary file – this is where you decide on its name;
- **surfacePatch**: this is the name in the mesh assigned to the liquid-gas interface boundary;
- **gasVolume** is the volume of the gas region, with dimensions [m^3];
- **R** is the universal gas constant, with dimensions [$kg\ m^2\ s^{-2}\ K^{-1}\ mol^{-1}$];
- **T** is the name of the temperature variable;
- **k_p** is the gas outlet pipe resistance coefficient, with dimensions [$m^4\ s\ kg^{-1}$];
- **kGasCutOff**: the gas flow out of the gas volume shuts off when pressure is too low to prevent back flow. To simulate this shut off, this logistic function is used:

$$H = \left[1 + exp\left(1 - 2K_c\left(p_{tot} - p_{atm} - p_c\right)\right)\right]^{-1}, \tag{93}$$

where K_c is the gas cut-off coefficient (dimensionless).

- **p_atm** is the atmospheric pressure. It is assumed that the atmospheric pressure is modelled as a coefficient.
- p_liq is the vapour pressure of the bulk liquid. This is also assumed to be a coefficient.

K.8.2 Species settings

The gas model can have any number of species. They each are defined in the `species` sub-dictionary.

- **liquidVariable** is the name of the variable for the dissolved gas concentration of the species in the liquid volume;
- **conversionToMoles** is misnamed in this version. It actually should be "conversionFromMoles". This is a conversion factor that you multiply moles of the species with to get kilograms. If the liquidVariable is already in [mol/m^3] (as opposed to [kg/m^3]), conversionToMoles should be dimensionless 1.0.
- **k_L_a** is the overall gas-liquid mass transfer coefficient, with units [s^{-1}];
- **k_H** is Henry's coefficient for gas transfer, with units [kg^{-1} s^{-2} m^2 mol];
- **k_b** is the coefficient of bubble formation, with units [s^{-1}];
- **formBubbles** allows you to enable or disable the bubble source term model;
- **upperLimit** and **lowerLimit** allow you to put limits on the concentration of this species in the gas volume;
- **relaxation** is for convergence control. Generally this should be a fraction, less than 1.0. Should the solution be oscilliatory or divergent, reduce this;
- **deltaConvergence** allows you to specify the convergence criterion for this species, where the error is calculated by comparing iterations;

- massConvergence allows you to specify the convergence criterion for this species, where the error is calculated by comparing the mass transfer out of the liquid volume (diffusion) with the mass transfer into the gas volume (Henry's law); and
- **nearZeroDeltaScale** and **nearZeroMassScale**: when the values get close to zero, error calculations become unstable. This allows you to specify the scaling factor for near-zero error scaling.

Appendix L PLC emulator paper

Proceedings of The Canadian Society for Mechanical Engineering International Congress 2012
CSME International Congress 2012
June 4-6, 2012, Winnipeg, Manitoba, Canada

A PROGRAMMABLE LOGIC CONTROLLER EMULATOR WITH AUTOMATIC ALGORITHM-SWITCHING FOR COMPUTATIONAL FLUID DYNAMICS USING OPENFOAM

David L. F. Gaden and Eric L. Bibeau
Department of Mechanical Engineering, University of Manitoba
Winnipeg, Canada
david_gaden@umanitoba.ca, eric_bibeau@umanitoba.ca

Abstract—A programmable logic controller (PLC) emulator is implemented into OpenFOAM, a computational fluid dynamics (CFD) software suite. The controller is generic and allows users to study the fluid dynamics and the control system of engineering processes simultaneously. The user specifies their own inputs, outputs and logic, and the software will subsequently simulate the entire time dependent process. The controller also allows the simulation to dynamically change the algorithm it uses, making it suitable for a wider range of processes, including those with distinct stages characterised by different physics. This is the most versatile PLC emulator ever implemented and successfully demonstrated for CFD.

Keywords-PLC;control;CFD;algorithm-switching;OpenFOAM

I. INTRODUCTION

Computational fluid dynamics (CFD) is used to simulate engineering processes involving fluid flow. Many such processes involve control systems, and these complicate the use of CFD. For instance, the heating, ventilating, and air-conditioning (HVAC) system of a building has various components, such as fans and heaters, that may be on or off depending on the state of the system. This can be accommodated by performing CFD simulations for all possible states, or by incorporating the control system into the simulation itself.

Control systems sometimes use a previously created library of simulation data to inform their operation [1-6]. This is achieved by performing CFD simulations for all possible states, and for a variety of expected conditions. However, this strategy can become impractical with system complexity, as the number of states increases exponentially with the number of switchable outputs. Furthermore, each simulation begins with a predetermined initial state, which will not accurately capture transitions between states, nor will the set of simulations be adequate in evaluating the control strategy. A better solution is to incorporate the control system directly into the CFD simulation itself, handling system changes internally based on logic provided by the user.

There is a large body of literature involving CFD simulations with built-in control systems [3,7-9], and these provide insight into their individual control strategy. However, these systems are custom-written, application-specific implementations. A CFD simulation with a general framework for process control would be more versatile. A programmable logic controller (PLC) emulator for would achieve this goal.

Once a general process control framework is implemented, a CFD solver becomes applicable to a wider set of conditions, due to the changes in the system state. As a direct consequence of this, the system may enter a state where the original solver algorithm is inadequate. This includes engineering processes with distinct stages characterized by differing physics, or simulations that require an initialization algorithm. For instance, a chemical batch process may begin with a transient heating and mixing stage, followed by a steady-state reacting stage. A solver for such a process must be capable of *algorithm-switching*.

Algorithm-switching is the ability of a CFD solver to dynamically change the algorithm during a simulation. Although not formally defined, some algorithm-switching has previously been reported. Gains in efficiency have been realized by dynamically changing the discretization schemes mid-run [10,11]. Kobolov et al. [12,13] developed a solver that switches from a rarefied medium algorithm to a continuous medium algorithm for spacecraft re-entry studies. However, a general algorithm-switching framework for CFD has not been reported.

The purpose of this paper is to develop a proof-of-concept PLC emulator and an algorithm-switching framework for CFD simulations, verify the emulator, and release the resulting source code to the CFD community as free and open source software.

II. Theory

A PLC has a set of inputs, outputs, and logic.

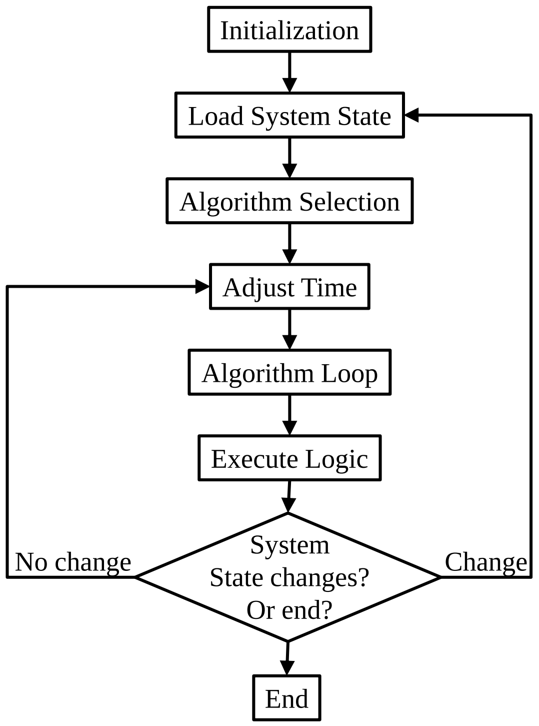
- **Inputs** are any measurable quantity within the simulation, including time. These represent actual sensors, such as flow meters, thermocouples, or timers in the physical system being simulated.
- **Outputs** are changeable entities within the model that can alter the course of the simulation. This can include boundary conditions, modelling constants, source terms, and even the algorithm to use, dynamically changing the sets of non-linear equations to solve. The outputs represent physical components such as valves, heaters, and fans.
- **Logic** is a user-supplied set of conditions that describes the state of the outputs for all given sets of inputs; or, when provided with an initial state, describes only the conditions that cause a change of state. Ladder logic is the most common form of PLC logic descriptor.

III. Code Development

OpenFOAM, a free and open source CFD suite, is used as a platform for this project.

A. Overview

The sequence of operations for the proposed emulator is shown in Figure 1.



PLC emulator process diagram

Since the PLC emulator uses timers as inputs, it sometimes must modify the end time and timestep of the current simulation in order to properly meet the end of the timer. That is the purpose of the “adjust time” step in the emulator.

B. Algorithm-switching

OpenFOAM does not have algorithm-switching capabilities, therefore an extension was written, called *multiSolver*. multiSolver was publicly released as free and open source software in July of 2010 [14], and has subsequently been used by the OpenFOAM community [15].

Since each algorithm requires its own set of objects in memory, the only safe and practical way to switch between algorithms is to let the first algorithm terminate normally and subsequently initialize the next algorithm using the latest values for each variable. Therefore, there is considerable overhead in switching between algorithms. However, this gives an opportunity to change the case settings, including boundary conditions, modelling constants, and source terms. Thus multiSolver can be used in a general way to control the system state.

C. Outputs

To control the simulation, the PLC emulator must be capable of changing:

- the boundary conditions;
- the source terms;
- the modelling constants; and
- the algorithm.

multiSolver has the ability to address these requirements, and therefore can be used universally; however, each time it makes a change to the system state, multiSolver deletes and reinitializes all objects in memory. This overhead needs to be considered if the control scheme requires frequent changes to the outputs and for larger simulations; however, this cost of using a PLC emulator must also be balanced with its benefits, which include:

- increased simulation quality assurance by minimizing user errors when manually manipulating input and output variables between control states,
- reduced time for the CFD user when running a complete simulation case, and
- making simulations more representative of the physical world.

D. Inputs

The inputs available to the PLC are:

- **variable limits** – when a given variable is greater than or less than a given limit, evaluated either at a specific position, or averaged over the entire geometry;
- **equation** – when a user-supplied equation is greater than or less than a given limit;
- **timers** – these can repeat, and can be triggered by other inputs;
- **system state groups** – these inputs are true or false depending on the state of the system;
- **solver signals** – signals sent from the solver to the PLC; and
- **conditional switches** – these are true or false depending on the conditions of other inputs; and

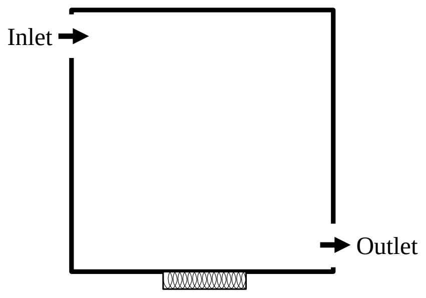
The PLC emulator uses OpenFOAM's *runtime selection* idiom, wherein it uses a set of generic base classes as interfaces between model components. This gives it a modular, extensible design. Therefore, if a user wishes to create additional custom inputs, this can be achieved without modifying any of the existing code, thus providing a complete environment in which to easily define the control logic.

E. Logic

The logic is user-specified in a text-based input file, the format for which is documented with the software release. The logic input file is currently verbose: a case with four controllable switches requires close to 400 lines of user-input as a text parser has not yet been implemented.

IV. Case Setup

A proof of concept case study and verification is conducted. The case study simulates a semi-continuous chemical process in the reactor vessel, as shown in Figure 2.



Reactor vessel

The process is:

- a relatively cold fluid (323 K) flows into a reactor vessel for sixty seconds, at which point the vessel is sealed and heated;
- once the heat reaches a set temperature (473 K), the reactions begin;
- the temperature is controlled by a thermostat that switches on below 513 K, and off above 523 K;
- after 1,000 seconds (close to sixteen minutes), the inlet and outlet are opened again; and
- the products flow out, cold fluid flows in, and the process repeats.

The case study does not have actual reactions; rather it uses the concept of reactions to demonstrate algorithm-switching. There are two algorithms: one with reactions, and one without. They are both identical, except the one with reactions also defines a secondary variable, *S*, which follows a sinusoidal trajectory.

There are three outputs managed by the PLC in this model. These are:

- fluid flow, on or off;
- heater, on or off; and
- reactions, on or off.

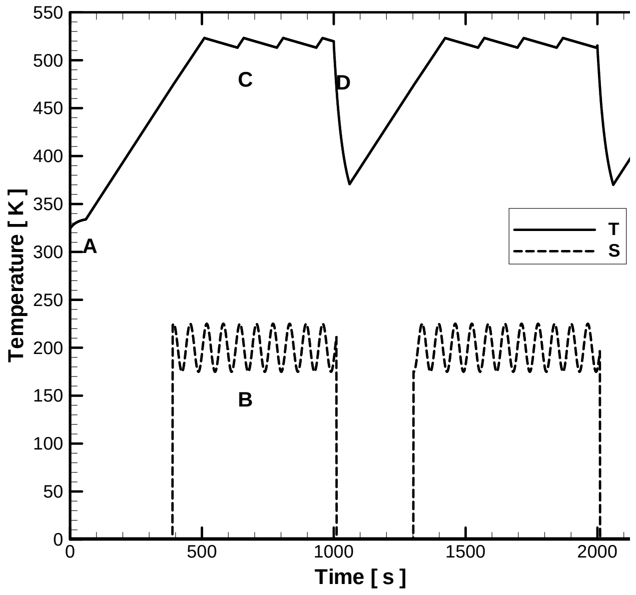
The inputs are:

- temperature; and
- time.

For simplicity, the case study is a two-dimensional, incompressible simulation. Boussinesq buoyancy is used, thus providing a mixing effect when the flow is switched off. In order to demonstrate thermostat function, the heat loss through the vessel walls is exaggerated.

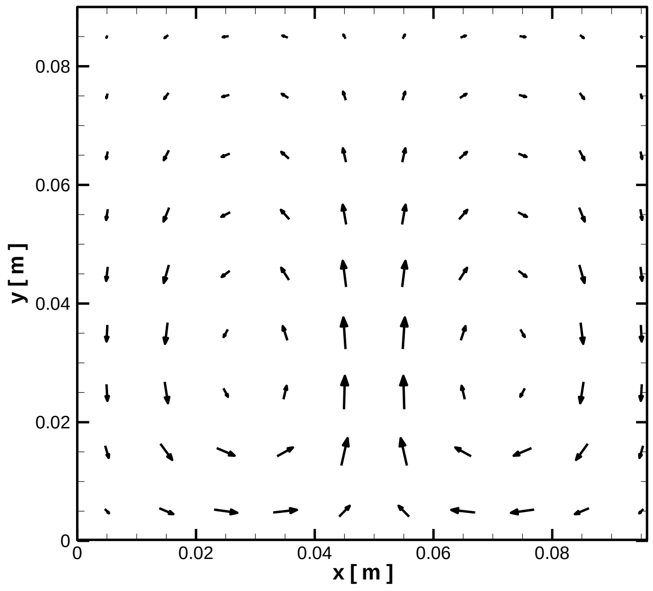
V. Results and Discussion

The temperature and reaction variable profiles are a good proxy for visualising the operation of the controller. A plot of these variables is shown in Figure 3. The simulation begins with the flow and heater on. The sharp change in the temperature rise at 60 seconds (feature A) occurs when the process flow stops. At approximately 390 seconds, as the temperature passes 200 °C, the first reactions begin and the reaction variable begins registering oscillations (feature B). This demonstrates algorithm-switching is occurring. The saw-toothed profile of the temperature during these reactions (feature C) is the thermostat controlling the heater. The sharp drops at 1,000 and 2,000 seconds (feature D) occur when the flow is switched on, allowing cold fluid to enter the reactor vessel.



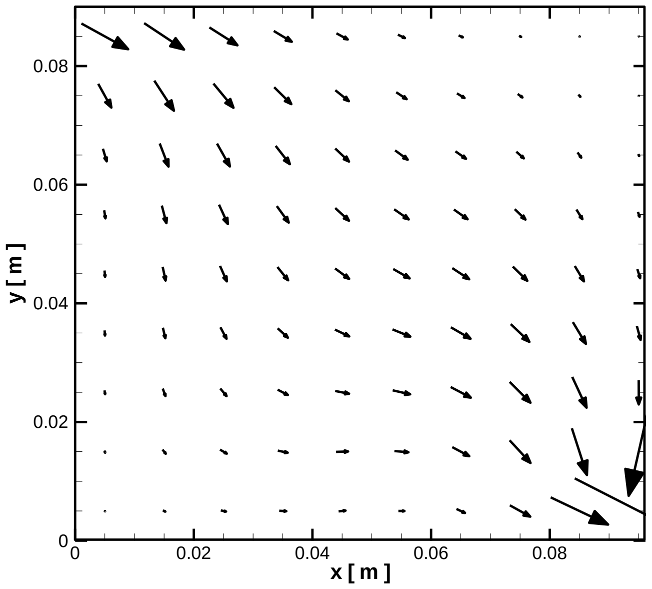
The average temperature and reaction variable, as a function of process time.

The flow profiles also show proper controller function. When the flow is switched off, buoyancy forces dominate, giving the profile shown in Figure 4.



Velocity vectors with flow switched off.

Similarly, when the flow is switched on, a cross-flow is seen traveling from inlet to outlet, as shown in Figure 5.



Velocity vectors with flow switched on.

The case study shows proper function of the PLC emulator, and the expected response within the CFD solution. Algorithm-switching for a simple case study has been verified.

VI. Conclusion

A general PLC emulator with algorithm-switching was implemented within the framework of a CFD simulation. A proof-of-concept case study verified its proper function and demonstrated its usefulness. With this PLC emulator, a user can evaluate complex 3-D processes using CFD, from the perspectives of fluid flow and process control simultaneously. Algorithm-switching makes this tool more versatile as it can apply to a wider range of physics and boundary conditions.

VII. Future Work

The PLC emulator does not currently support parallel processing. Once parallel processing has been implemented, the PLC emulator will be made publicly available as free and open source software.

Other suggested improvements include:

- a logic parser would make the entering the PLC logic easier from a user's perspective, as the current format requires approximately 100 lines of user-input per output switch;
- mesh motion needs to be incorporated into the multiSolver component in order to allow control systems that manipulate geometry; and
- a mechanism through which the PLC emulator can directly change boundary conditions would improve model performance.

Contributions from the open source community are welcomed.

Acknowledgment

Funding for this project was provided by Manitoba Hydro, and the Natural Sciences and Engineering Research Council of Canada (NSERC). Technical assistance from Hrvoje Jasak was gratefully appreciated.

REFERENCES

[1] R. Zhang, C. Zhang, J. Jiang, "A new approach to design a control system for a FGR furnace using the combination of the CFD and linear system identification techniques," Combustion Theory and Modelling, Vol. 15, No. 2, pp. 183-204, 2011.

[2] Y. Liu, Hanchang Shi, Huiming Shi, Z. Wang, "Study on a discrete-time dynamic control model to enhance nitroten removal with fluctuation of influent in oxidation ditches," Water Research, Vol. 44, No. 18, pp. 5150-5157, 2010.

[3] M. Milovanovic, O. M. Aamo, "Attenuation of vortex shedding in CFD simulations by model-based output feedback control," 49th IEEE Conference on Decision and Control, Atlanta, USA, December 15-17, 2010.

[4] T. Plikas, L. Gunnewiek, T. Gerritsen, M. Brothers, A. Karges, "The predictive control of furnace tapblock operation using CFD and PCA modeling," JOM, October 2005.

[5] Y. Yang, M. A. Reuter, D. T. M. Hartman, "CFD modelling for control of hazardous waste incinerator," Control Engineering Practice, Vol. 11, No. 1, pp. 93-101, 2003.

[6] Stropky D., Bibeau E.L., Yuan J., Salcudean M., “Advanced process modelling saves money,” 2001 Joint Engineering/F&C Conference, San Antonio, Texas, December, 2001.

[7] Z. Sun, S. Wang, "A CFD-based test method for control of indoor environment and space ventilation," Building and Environment, Vol. 45, No. 6, pp. 1441-1447, 2010.

[8] S. Wong, W. Zhou, J. Hua, "Designing process controller for a continuous bread baking process based on CFD modelling," Journal of Food Engineering, Vol. 81, No. 3, pp. 523-534, 2007.

[9] S. B. Choi, H. Xu, M. D. Mirmirani, "LQG control of a CFD-based aeroelastic wing model," Proceedings of the 42nd IEEE Conference on Decision and Control, Maui, USA, December 2003.

[10] R. Winkelmann, J. Häuser, R. D. Williams, "Strategies for parallel and numerical scalability of CFD codes," Computer Methods in Applied Mechanics and Engineering, Vol. 174, No. 3-4, pp. 433-456, 1999.

[11] R. Löhner, H. Luo, J. D. Baum, D. Rice, "Improvements in speed for explicit, transient compressible flow solvers," International Journal for Numerical Methods in Fluids, Vol. 56, No. 12, pp. 2229-2244, 2008.

[12] V. I. Kolobov, H. Q. Yang, S. A. Bayyuk, V. V. Aristov, A. Frolova, S. Zabelok, "Unified methods for continuum and rarefied flows," 42nd AIAA Aerospace Sciences Meeting and Exhibit, Reno, USA, January 5-8, 2004.

[13] V. Kobolov, V. Aristov, R. Arslanbekov, S. Bayyuk, A. Frolova, S. Zabelok, "Construction of a unified continuum/kinetic solver for aerodynamic problems," Journal of Spacecraft and Rockets, Vol. 42, No. 4, pp. 598-606, 2005.

[14] D. L. F. Gaden, “multiSolver released,” CFD-Online Forums, message posted to www.cfd-online.com/Forums/openfoam-news-announcements-other/78502-multisolver-released.html, July 23, 2010.

[15] K. Gott, A. Kulkarni, “Construction and application of a unified flow solver for use with physical vapour deposition (PVD) modeling,” poster presented at the 6th OpenFOAM Workshop, State College, USA, June 13-16, 2011.

Appendix M Stirplate experiment

A draft paper titled "fluid flow characterization into the stirring action of a laboratory stir plate," detailing the stirplate experiment that was conducted is presented below.

M.1 Abstract

Magnetic stir plates and hexagonal stir bars are frequently used in the laboratory environment but there are few studies that report fluid dynamic shear stresses of the stirring action. To quantify the mixing action, a Particle Image Velocimetry (PIV) and Acoustic Doppler Velocimetry (ADV) experimental investigation is performed on a cylindrical vessel undergoing a stirring action imparted by a hexagonal stir bar. The fluid characterization is performed at six horizontal cross-sections at four different stirring velocities. The tangential and radial components of the mean velocities are studied as well as the shear stress profile. The ADV experiment is taken at a single spatial point in the vessel, for all four stir speeds. The PIV and ADV results are compared, and the mean tangential velocities are found to be in good agreement. The PIV dataset is found to have large errors in the fluctuating velocity components and the correlation coefficient.

M.2 Introduction

Magnetic stirring is a frequently used mixing process in laboratories, and a variety of automated stirrers are commercially available. Often all that is necessary from the perspective of practical experimentation is to have the fluid in question thoroughly mixed. However, there are applications where it is important to understand the motion of the fluid and shear stresses generated as it is undergoing the stirring action. For instance, the rate of a chemical or biological interaction is related to the mixing within the fluid, especially important in the hydrolysis stage for anaerobic digestion. The purpose of this study is to characterize the stirring action of a laboratory magnetic stir plate with a six inch diameter (152.4 mm) cylindrical vessel. This will provide insight into studies that have been performed using this type of equipment. Scaling laws are introduced to generalize the results.

Most investigations of stirring have been experimental, although numerical models have advanced to the point that they show promise as a practical method [1]. Several investigations focus on the mixing characteristics, such as liquid-liquid and two or three phase flows. Lamberto et al. [2], Hall et al. [3] and Montante et al. [4] study liquid-liquid mixing; and Hadjiev et al. [5] studied liquid-solid mixing. Several studies have focused on the flow field and transient effects near Rushton impellers [6-9], pitched bladed impellers [3, 10] and other types of impellers [11-13], but there have been no studies in the open literature on the stirring action of a hexagonal stir bar actuated by a magnetic stir plate, which is a common piece of equipment in a laboratory environment.

In this study, the stirring motion of a fluid in a cylindrical vessel is investigated using a Particle Image Velocimetry (PIV) system and an Acoustic Doppler Velocimetry (ADV) system. Due to geometric limitations of the ADV equipment, a single spatial point was characterized in the vessel for the ADV experiment. Therefore emphasis is placed on the PIV results.

M.2.1 Scaling considerations

A characteristic length and velocity scale are required to make the results non-dimensional and extend the applicability of the results. For the velocity scale, the maximum radial component of velocity, ϑ, will be used; for the length scale, the vessel diameter, R, will be used: $L = R$, and $U = U_{\vartheta,max}$. These scaling parameters are then employed to produce dimensionless quantities: $r^* = r / R$; $U_{\vartheta,max}{}^* = U_\vartheta / U_{\vartheta,max}$; and $U_r{}^* = U_r / U_{\vartheta,max}$. To non-dimensionalize shear stress, the kinematic viscosity is proposed:

$$\tau^* = \tau \left(\mu \frac{U_{\theta,\max}}{R} \right)^{-1} , \tag{A94}$$

where shear stress is given by:

$$\tau = \mu \frac{\partial U_\theta}{\partial r} . \tag{A95}$$

M.2.2 Experimental parameters

The goal of the experiment is to capture the stirring motion induced by a Cimarec magnetic stir plate within a cylindrical vessel of the dimensions shown in Figure A8. The principal fluid motion is in the ϑ direction; therefore the fluid profile is captured using horizontal cross sections rather than vertical.

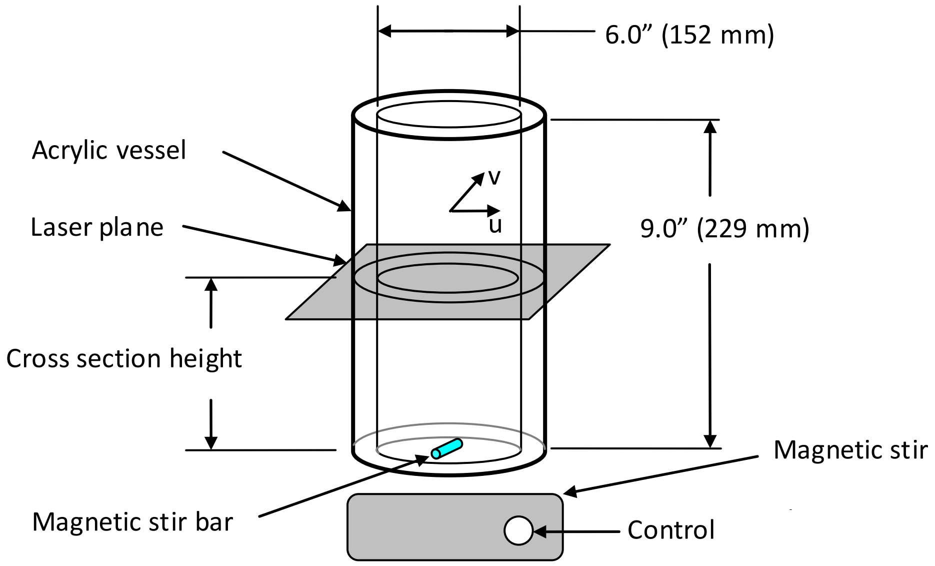


Figure A8: Stir vessel geometry

Four different stirring velocities were used and investigated at six different cross-sectional heights. The experimental parameters are summarized in Table A12, where U_{tip} is the tip speed of the stir bar. The stir plate setting was controlled with an analogue knob, therefore the mixing speed varied slightly between experiments. Experimental trials are denoted by height, H, and stir plate setting, S. For instance, H3S4 refers to the trial taken at plane 3 and setting 4.

Table A12 – Experimental trials

Trial	**Height** [mm]	**Setting**	**RPM**	U_{tip} [mm s^{-1}]	$U_{\vartheta,max}$ [mm s^{-1}]
H1S3	43.1	3	80.86	322.63	99.03
H1S4	43.1	4	122.45	488.55	166.64
H1S5	43.1	5	210.53	839.96	310.67
H1S6	43.1	6	310.88	1240.36	475.27

H2S3	69.1	3	76.14	303.79	98.70
H2S4	69.1	4	139.53	556.72	192.55
H2S5	69.1	5	209.06	834.11	302.34
H2S6	69.1	6	311.69	1243.58	489.57
H3S3	96.1	3	80.18	319.90	103.83
H3S4	96.1	4	131.29	523.83	182.60
H3S5	96.1	5	200.67	800.63	302.98
H3S6	96.1	6	302.27	1205.99	482.68
H4S3	121.1	3	77.12	307.70	97.29
H4S4	121.1	4	132.45	528.45	184.23
H4S5	121.1	5	211.27	842.92	320.26
H4S6	121.1	6	310.88	1240.36	495.08
H5S3	145.1	3	74.81	298.49	96.69
H5S4	145.1	4	129.31	515.93	187.65
H5S5	145.1	5	206.19	822.64	307.96
H5S6	145.1	6	311.11	1241.28	500.79
H6S3	169.1	3	69.77	278.36	89.22
H6S4	169.1	4	130.72	521.55	185.34
H6S5	169.1	5	202.73	808.86	305.93
H6S6	169.1	6	305.34	1218.27	497.50

M.2.3 PIV Experimental method and apparatus

All components of the stirring vessel were constructed of acrylic to enable imagery. The stirring action was produced by a 3” (76.2 mm) long by $^3/_8$” (9.53 mm) diameter hexagonal magnetic stir bar. A 120 volt 7” x 7” Cimarec magnetic stir plate actuated the stir bar. The velocity and shear stress data was collected using a PIV system. The images were captured from a position above the vessel. To prevent image distortion caused by the motion of the free surface, the top was enclosed and air was removed. The vessel was filled with water that had been sitting still for several days to ensure no dissolved oxygen would precipitate out and obstruct the camera view. The water was embedded with polyamide seeding particles with a mean diameter of 20 microns. The fluid was illuminated with a New Wave Gemini double pulsed Nd:YAG laser, frequency doubled to emit light at a wavelength of 532 nm. To reduce diffraction of the laser plane through the cylinder, the vessel was place in a rectangular tank that was also filled with water. Diffraction was minimal as the laser plane filled the entire cross section of the vessel. A Kodak Megaplus ES1.0 CCD camera was used to record images with a resolution of 1008 x 1016 pixels. The horizontal alignment of the laser plane was achieved using a levelled target. The camera position was adjusted using a printed target to ensure the desired field of view was acquired. The anti-diffraction tank, vessel and magnetic stir plate were positioned on a vertically adjustable platform to facilitate data acquisition at various heights without the need to realign the laser for each position. The effective optical distance from the camera to the laser plane was a function of fluid depth, and therefore the camera required focal adjustments at each position. This also impacted the data reduction, as the scale factors changed. The rotational speed of the stir bar was measured using acoustic tachometry.

M.2.4 ADV Experimental method and apparatus

The same acrylic vessel and fluid from the PIV experiment was used for the ADV experiment. The lid had to be altered to allow the ADV probe into the vessel - a ½” (12.7 mm) slot was cut from the centre to one of the edges. This also had the consequence of altering the flow field, as a free surface was now produced. Owing to spatial limitations, only a single point in the vessel was deemed acceptable for the ADV.

M.3 Data reduction

The dataset for each PIV trial consisted of 1200 paired images. Vector arrays were created from the image pairs using the cross-correlation and moving average validation algorithms included with the PIV data analysis software. For each trial, these vector maps were reduced to a single array with statistical information including mean velocities U_x and U_y, fluctuating components u_x' and u_y', and the correlation coefficient of the two velocities, C_{xy}. Interrogation areas measuring 32 x 32 pixels were employed with a 50% overlap, resulting in arrays of 62 x 62 vectors. Due to rotational symmetry, only a portion of the vessel cross-section was necessary, thus enabling a higher resolution to be attained. The sides of the interrogation areas varied between 3.17 mm and 3.32 mm depending on the scale factor for each experimental trial. With the 50% overlap, this resulted in vector spacing of approximately 1.6 mm.

The data is converted to polar co-ordinates. Mean velocity is calculated using:

$$\begin{aligned} U_r &= U_x \cos(\theta) - U_y \sin(\theta) \\ U_\theta &= U_y \cos(\theta) + U_x \sin(\theta) \end{aligned} . \tag{A96}$$

The fluctuating components are given by:

$$\begin{aligned} u_r' &= \left[u_x'^2 \cos^2(\theta) + u_y'^2 \sin^2(\theta) + 2C_{xy} u_x' u_y' \sin(\theta)\cos(\theta) \right]^{\frac{1}{2}} \\ u_\theta' &= \left[u_y'^2 \cos^2(\theta) + u_x'^2 \sin^2(\theta) - 2C_{xy} u_x' u_y' \sin(\theta)\cos(\theta) \right]^{\frac{1}{2}} \end{aligned} . \tag{A97}$$

Lastly, the correlation coefficient between the U_r and U_ϑ is determined using:

$$C_{r\theta} = \frac{\left(u_y'^2 - u_x'^2 \right)\cos(\theta)\sin(\theta) + C_{xy} u_x' u_y' \left[\cos^2(\theta) - \sin^2(\theta) \right]}{u_r' u_\theta'} . \tag{A98}$$

Owing to radial symmetry, the fluid dynamic quantities were averaged along lines of constant radius to produce one-dimensional profiles for each experimental trial. To facilitate this, the vector arrays were interpolated to a polar grid using Kriging, a cubic interpolation algorithm, as shown in Figure A9.

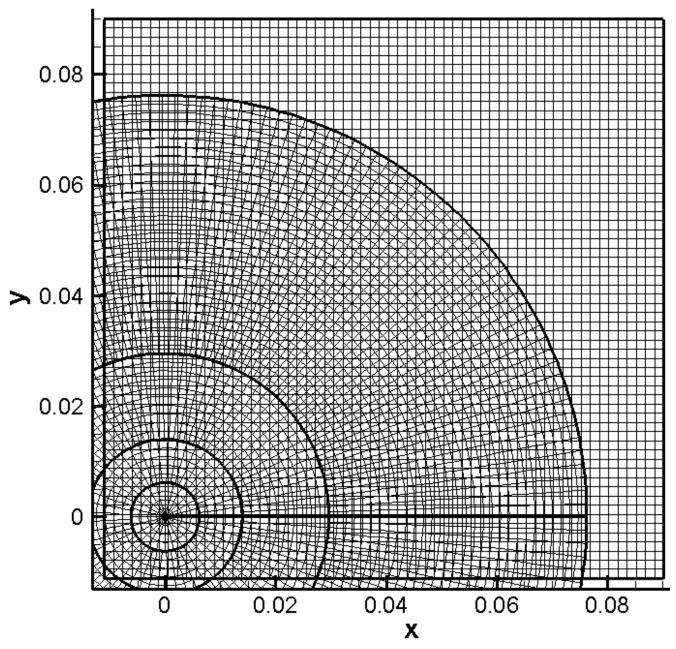


Figure A9: Interpolation grid

M.3.1 Monte Carlo study

A Monte Carlo study was performed to evaluate the variability in the PIV data, a strategy recommended by Keane et al. [14]. Monte Carlo simulations comparing subsets of a large data set have been used in previous studies to establish uncertainty [15]. The H5S5 dataset was selected for this analysis. From its 1200 vector maps, two data subsets of equal size were randomly selected, with no duplications permitted. Rejected vectors were removed, and statistical data was calculated for each dataset. The two resulting datasets were compared against one another and the differences were calculated using root-mean-square error:

$$\phi_{RMSE} = \frac{\sqrt{\frac{1}{N}\sum_{k}\sum_{l}\left[\phi_1(k,l)-\phi_2(k,l)\right]^2}}{\phi_s}, \qquad \text{(A99)}$$

where ϕ_1 and ϕ_2 are variables in datasets 1 and 2 respectively; *k* and *l* are indices on the vector map, and ϕ_s is a scaling standard. The root mean square of the variable over the full field and over all vector maps was used. This comparison was performed ten times for each subset size ranging from one to 600. The results are presented in Figure A10. The points represent all data; the lines represent a moving average that includes two hundred points.

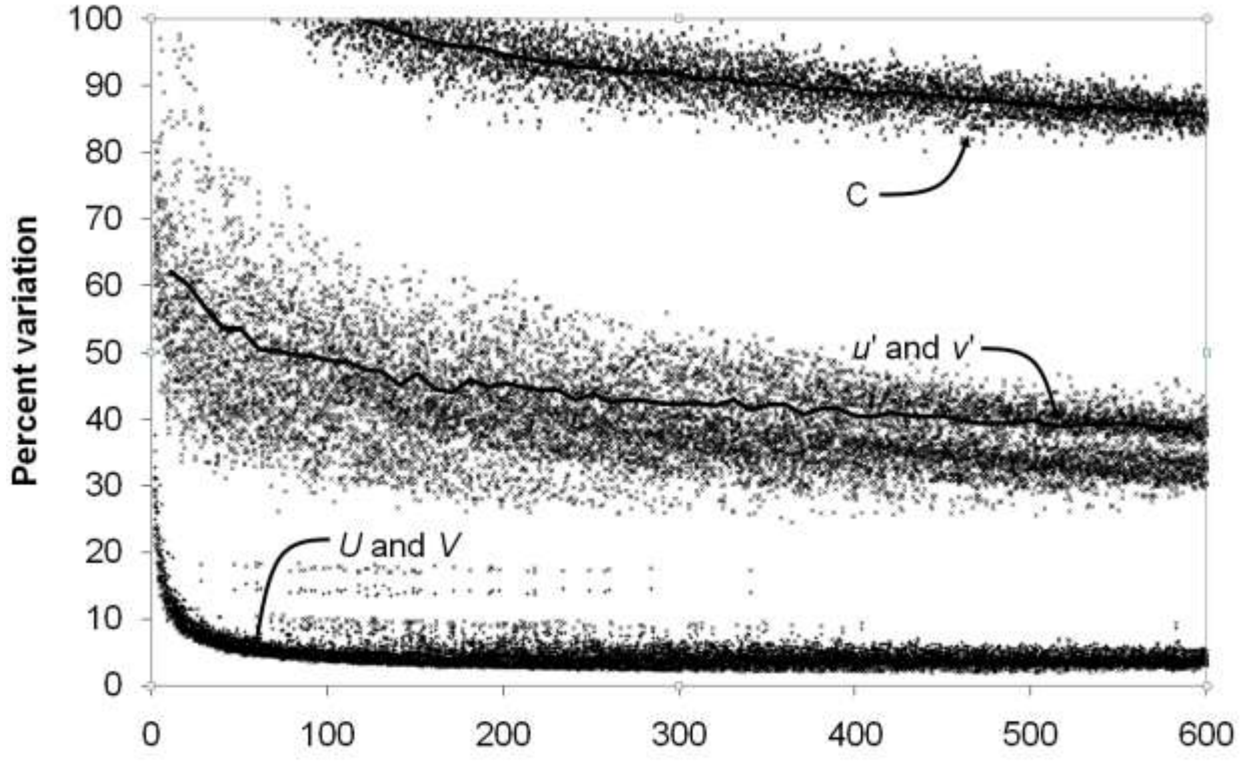


Figure A10: Monte Carlo simulation results showing velocities and fluctuations

This analysis reveals the consistency of the data, not its accuracy; a systemic error in the data will not show up here. If steady state conditions are expected, then variation in the data can be interpreted as a minimum to the error range. The results are summarized in Table A13.

Table A13 – Monte Carlo simulation summary, dataset size N = 600

Parameter	*U*	*V*	*u'*	*v'*	C_{xy}
Percent Variation	3.47	4.20	39.02	32.65	85.39

From Table A13, we can see that the minimum error in velocities is less than 5%, and the fluctuating components and correlation coefficient are each greater than 35%. The high level of inconsistency in the dataset suggests the presence of considerable noise resulting from the relatively large interrogation areas chosen.

M.4 Results and discussion

Velocity contours and streamlines for the full PIV field of view are shown in Figure A11 and Figure A12.

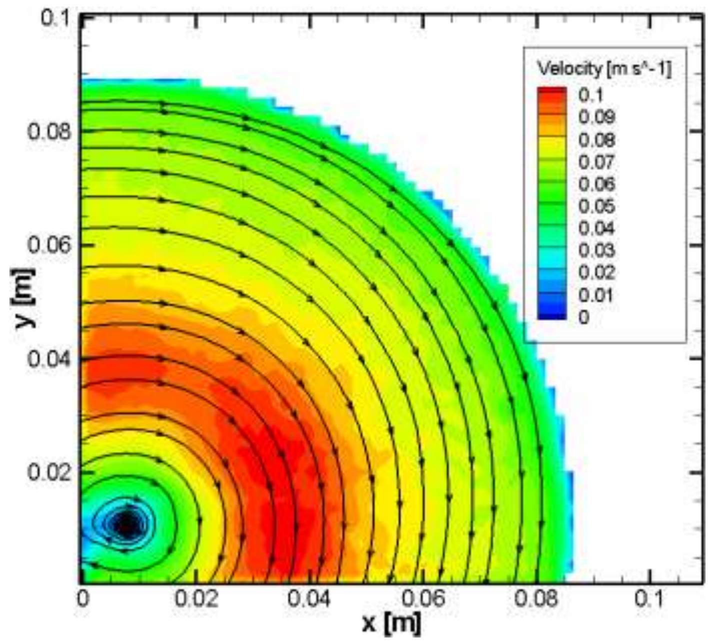


Figure A11: Velocity contour and streamline plot for trial H1S3

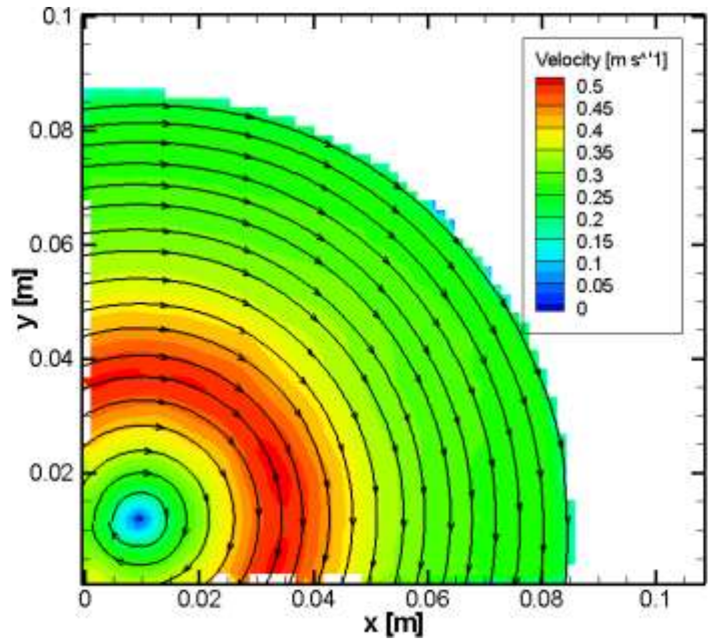


Figure A12: Velocity contour and streamline plot for trial H5S6

The maximum tangential velocity, $U_{\vartheta,max}$ was used as a characteristic scaling velocity. This quantity is not readily available from the experimental parameters. Another option would have been the tip speed of the stir bar, U_{tip}; however, scaling by the tip speed shows trends in the dataset less discernibly. These two velocities are strongly correlated by a power law, as shown in Figure A13.

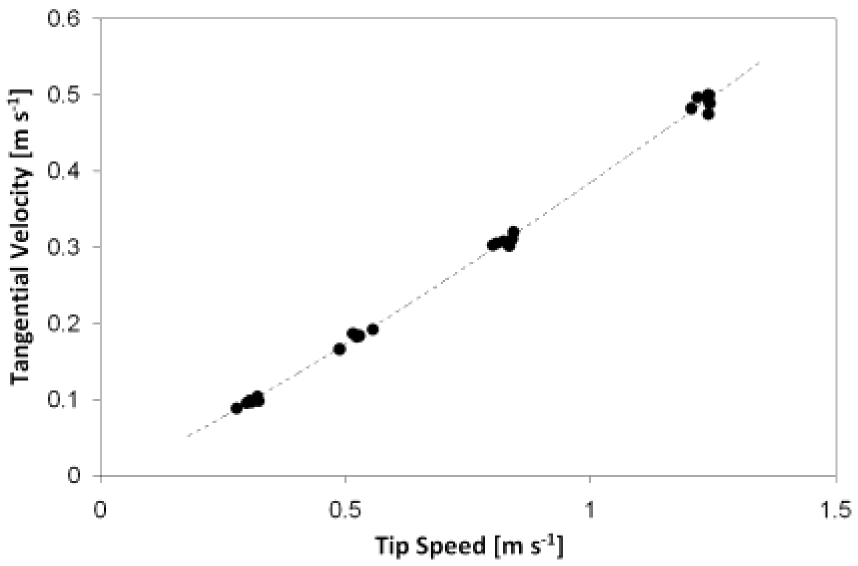


Figure A13: Tangential velocity versus stir bar tip speed

The trend is given by:

$$U_{\theta,\max} = (0.38551)\ U_{tip}^{1.1549}. \qquad \text{(A100)}$$

This relation is applicable to the specific conditions in this experiment. A general function would likely contain other variables, such as fluid viscosity and geometry. The cross-sectional height had no effect on the relation.

The tangential velocity profile was consistent throughout the experimental trials, as shown in Figure A14. It was characterised by a strong central vortex, followed by a gradual velocity dampening with increasing radius, until the wall effect dominates and velocity drops precipitously.

The main difference between trials was in the near-wall region. This was largely affected by the cross sectional height, as shown in Figure A15. Cross sections closer to the stir bar had a more even velocity, with a sharper slope at the wall, whereas cross sections further from the stir bar had a more pronounced peak. However, heights H1 and H2 were reversed from the general trend. This is possibly the effect of higher turbulence closer to the stir bar. The stir setting had a similar effect on the profile, although it was less pronounced, as shown in Figure A16. Higher stir settings had a more pronounced peak. The peak velocity was located between *r** = 0.306 and 0.367. The peak shifted slightly inwards with decreasing stir setting, and also with increasing cross section height.

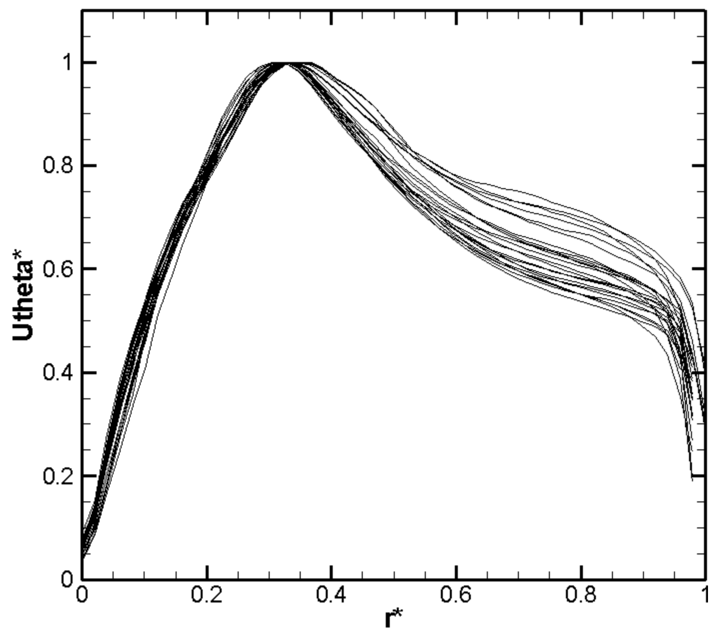


Figure A14: Mean tangential velocity, all trials

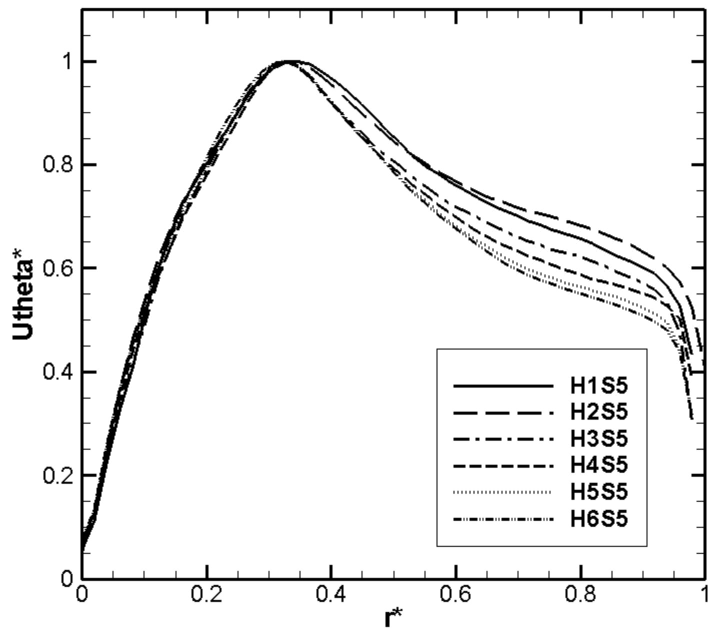


Figure A15: Tangential velocity at stir setting S5, for all cross-sectional heights

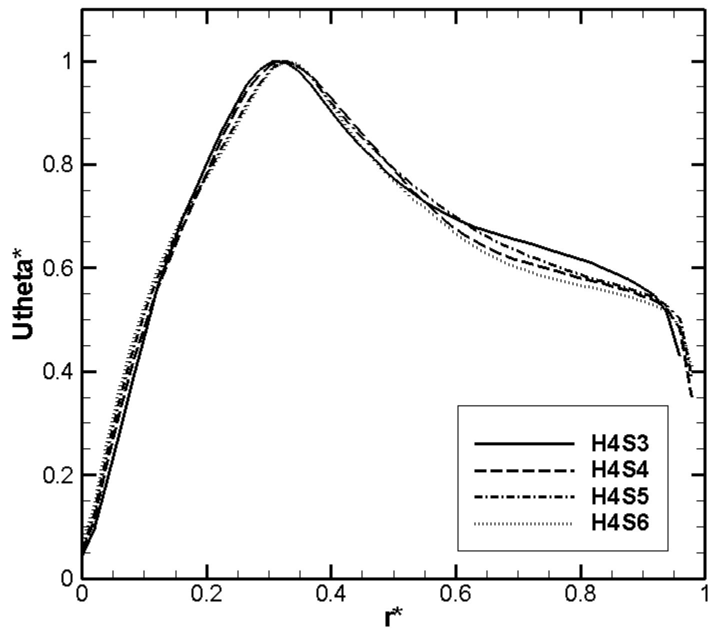


Figure A16: Tangential velocity, cross section H4 for all stir settings

Radial velocity, U_r, was several orders of magnitude smaller than the tangential velocity, U_ϑ, therefore the signal to noise ratio is much higher, as is evident in Figure A17. Despite the high level of noise, the general trends are clear. The radial velocity shows a significant negative quantity at the lower cross sectional heights H1 and H2, as shown in Figure A18 and Figure A19, respectively. At cross sectional heights of H3 and H4 in Figure A20 and Figure A21, the radial velocity shows a general positive trend. Finally, the H5 and H6 profiles are close to zero. Furthermore, there is a general trend of increasing radial velocity toward the vessel walls. Toward the centre of the vessel, all profiles are noisy, and close to zero. Indirectly, the axial velocity can be inferred from the radial velocity, as any radial component must be accompanied by an axial component due to continuity. These profiles suggest a recirculating vortex with divergent flow at the bottom and convergent flow at the top.

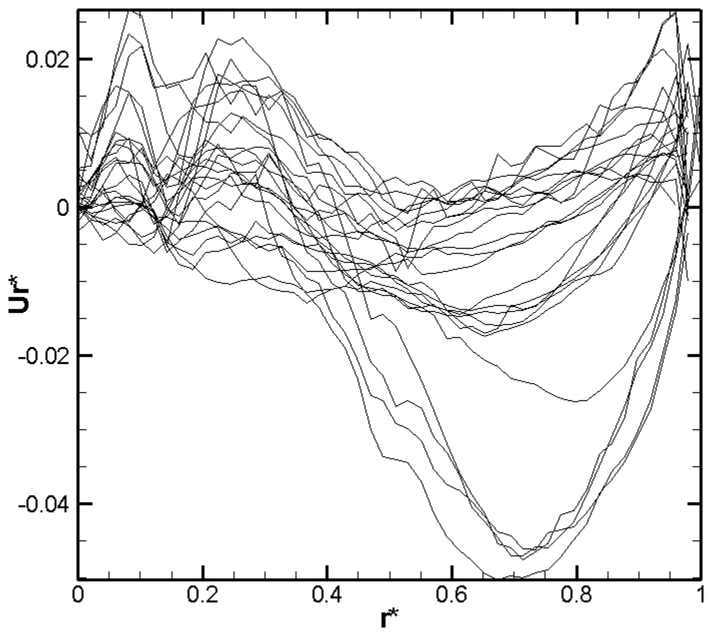


Figure A17: Radial velocity, all trials

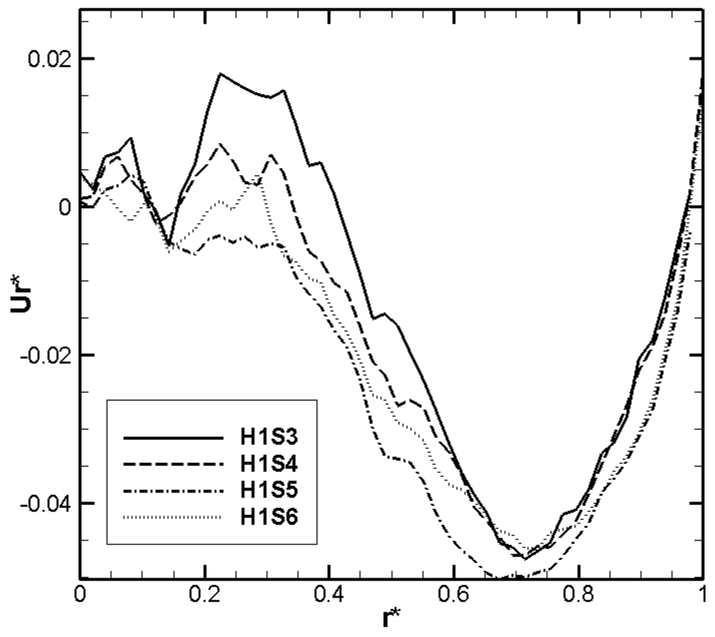


Figure A18: Radial velocity, cross section height H1, all stir settings

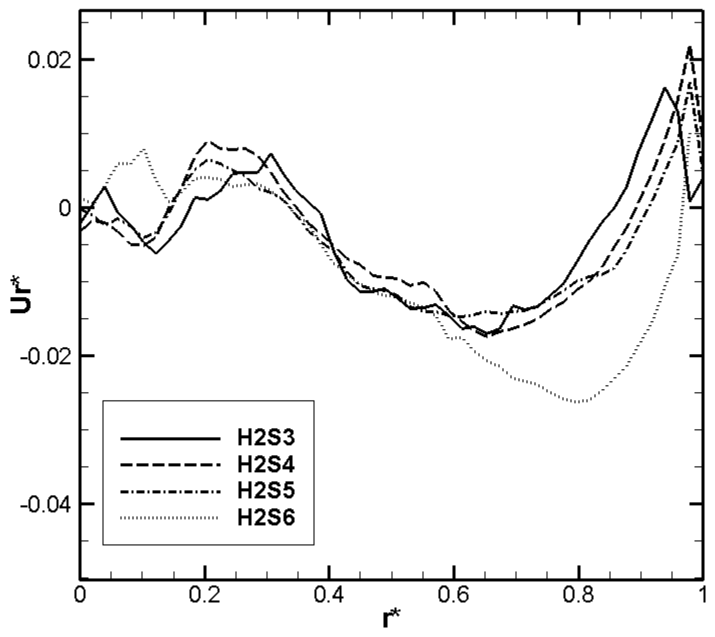


Figure A19: Radial velocity, cross section height H2, all stir settings

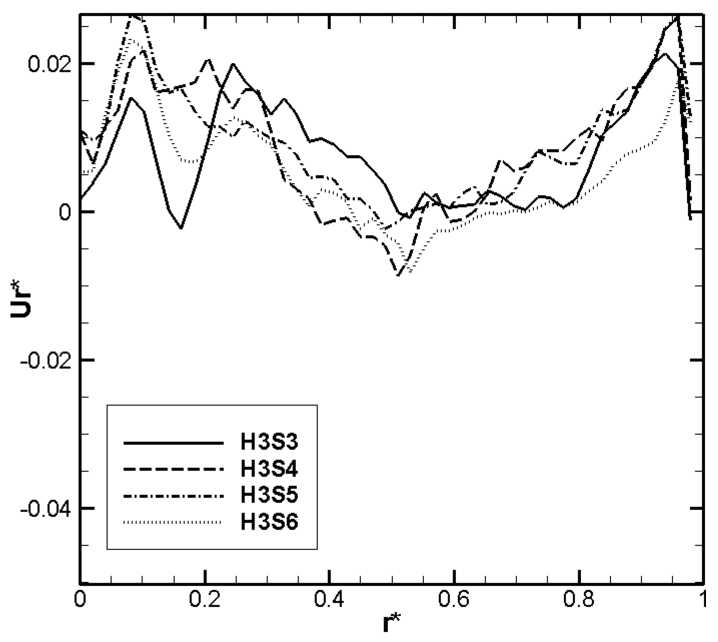


Figure A20: Radial velocity, cross section height H3, all stir settings

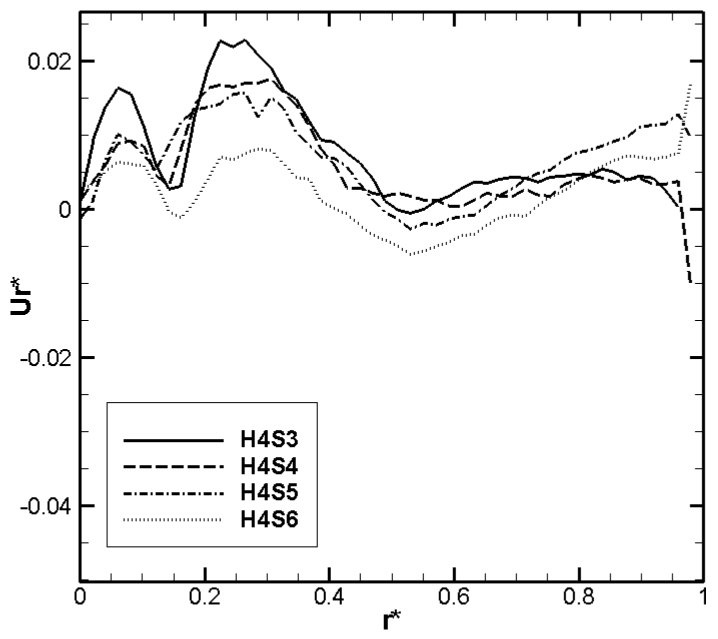


Figure A21: Radial velocity, cross section height H4, all stir settings

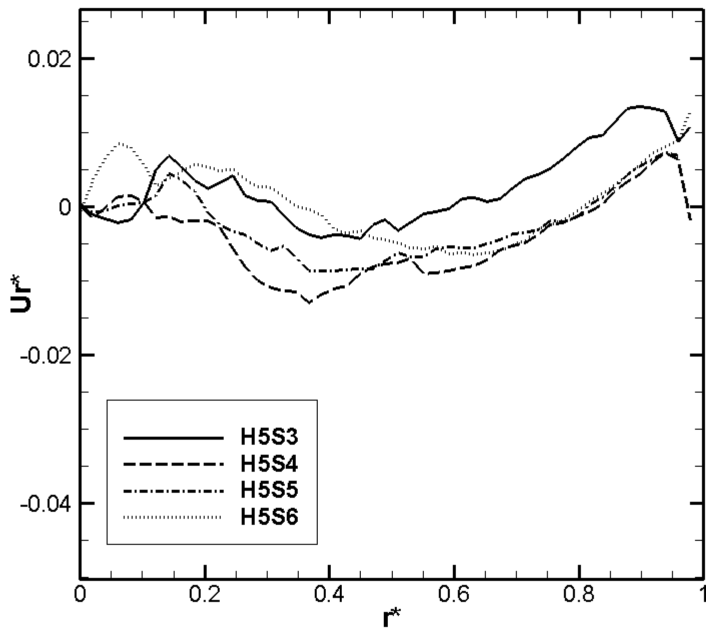


Figure A22: Radial velocity, cross section height H5, all stir settings

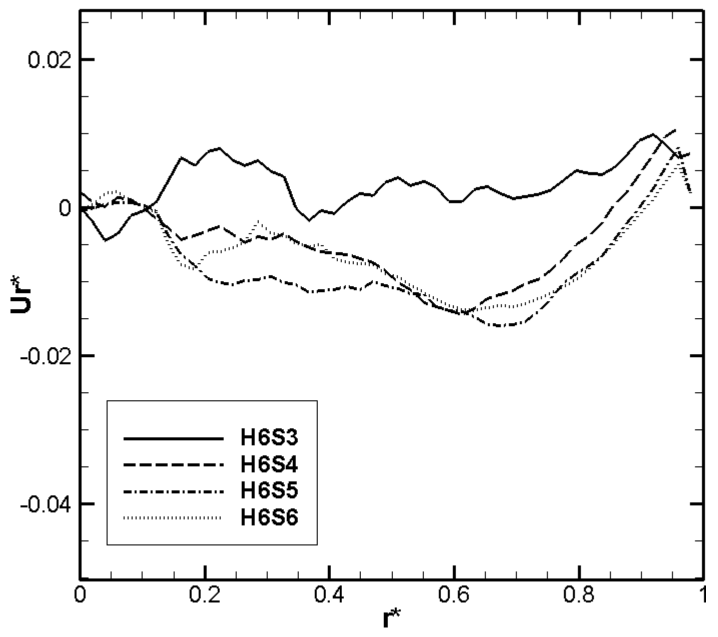


Figure A23: Radial velocity, cross section height H6, all stir settings

The shear stress profile was derived from the tangential velocity profile. As shown in Figure A24, there was little variation between trials. The profiles all had a sharp peak very close to the centre of the vessel, corresponding to the steep velocity slope passing through the axis. The shear stress dropped, passing through zero near *r** = 3.2 until levelling off at approximately *r** = 0.4. Finally, a sharp downward drop is seen from the wall effects. The stir setting had the most pronounced effect. Larger stir settings increased both the near-centre maxima and the near-wall minima. A secondary peak was visible in some of the profiles at approximately *r** = 0.27. This peak was evident at all cross sectional heights with the maximum stir setting, S6 (Figure A25), and for all stir settings at the lowest cross sectional height, H1 (Figure A26). Otherwise, this secondary peak was less pronounced, or not present. Noting that the stir bar radius is *r** = 0.5, it is interesting to observe that the vast majority of shear stress occurs within this region (excluding wall effects). This may suggest that larger or smaller vessels would not show a much different profile within the stir bar region; and may exhibit a shallower or steeper shear slope outside of this.

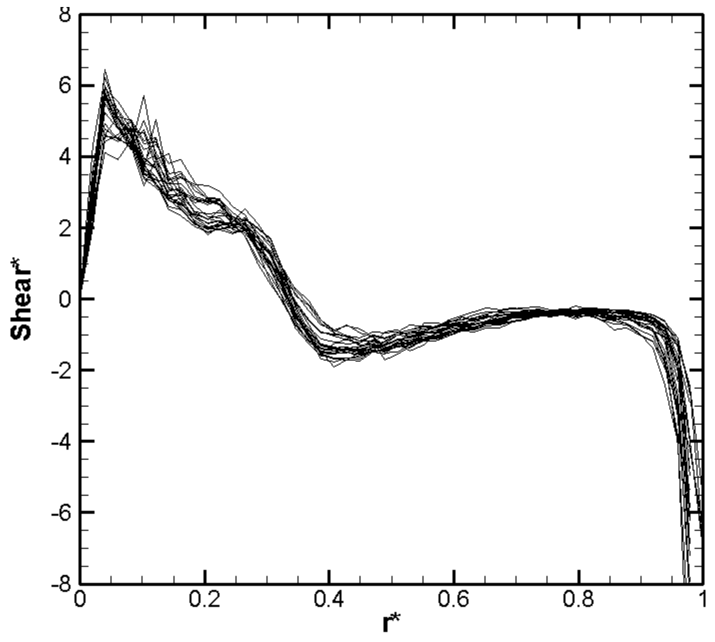


Figure A24: Shear stress profiles, all trials

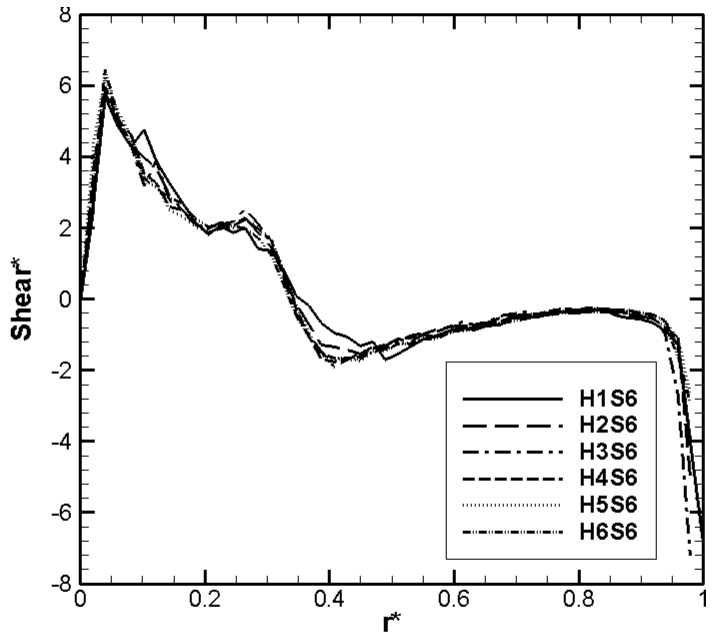


Figure A25: Shear stress, stir setting S6, all cross section heights

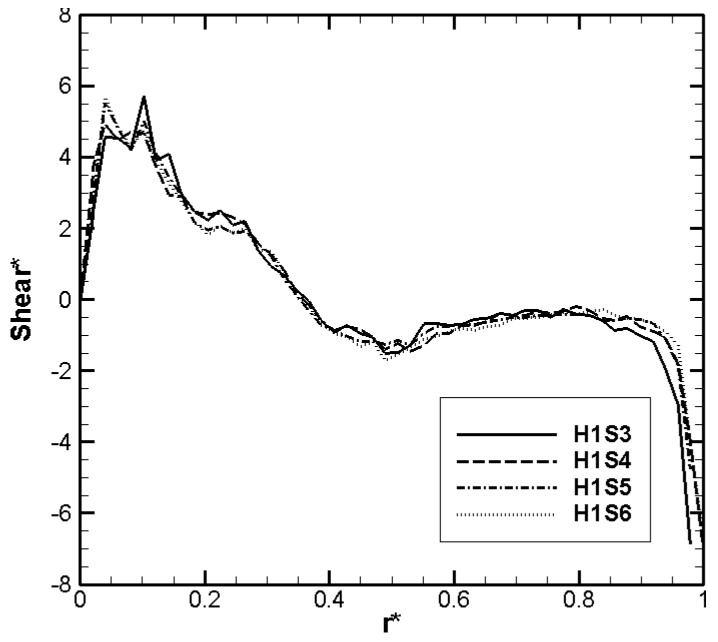


Figure A26: Shear stress, cross section height H1, all stir settings

M.4.1 PIV and ADV comparison

The ADV acquired data at the half radius point on the H5 plane. The PIV data was interpolated at this point to facilitate comparison. There were some notable differences between these two experiments.

The PIV and ADV trials were conducted with slightly different mixing speeds; and the required slot in the lid of the ADV experiment allowed for a free surface to form, whereas the PIV experiment required a closed top with no air. No attempt was made to account for these differences. Table A14 shows the result of the comparison between the ADV and PIV. The tangential ϑ component of velocity showed an excellent correlation between the two sensing systems, with errors ranging from 1.3% to 8.6%. The large difference in the S6 dataset can be partly accounted for with the difference in mixing speed. This flow quantity was expected to perform best as it had the largest signal to noise ratio for both systems. The *r* component of velocity faired the worst with the PIV sensing as much as two orders of magnitude smaller values than the ADV. This could be caused by the presence of the free surface in the ADV dataset. The fluctuating components were expected to compare poorly, given the high degree of variability in the PIV data. The ϑ RMS fluctuating component faired poorly, with the PIV detecting a value two to three times greater. Surprisingly the *r* component was close between the two datasets. It is possible that the *r* fluctuating component was so small that both systems simply represented the background noise level, which would be similar.

Table A14 – ADV and PIV comparison

Trial	**Data**	**RPM**	**Uθ** [cm/s]	**Ur** [cm/s]	**uθ'** [cm/s]	**ur'** [cm/s]
H5S3						
	ADV	70	7.21	-1.40	0.87	1.41
	PIV	75	7.46	-0.02	1.60	1.72
	% error	**7.14**	**3.48**	**98.30**	**83.19**	**21.69**
H5S4						
	ADV	130	15.15	-3.16	1.47	2.34

	PIV	129	14.82	-0.14	2.43	2.53
	% error	**0.77**	**2.20**	**95.53**	**64.73**	**8.10**
H5S6						
	ADV	205	24.54	-5.30	2.00	3.39
	PIV	206	24.23	-0.21	3.36	3.68
	% error	0.49	1.28	96.11	67.88	8.63
H5S7						
	ADV	305	35.33	-7.57	2.56	4.24
	PIV	311	38.37	-0.23	4.19	4.36
	% error	**1.97**	**8.62**	**96.98**	**63.36**	**2.92**

M.5 Conclusion

A PIV and ADV experimental study into the stirring action in a cylindrical vessel was performed. The PIV study characterised the fluid motion at six cross sections for four stirring velocities. The experiment was repeated using an ADV system, and a single point within the fluid was characterised. The PIV data exhibited large errors for the fluctuating velocity components and correlation coefficients; however the mean velocity error was less than 5%. The tangential velocity, radial velocity and shear stress profiles were studied. The tangential velocity showed a strong central vortex peaking at approximately r^* = 0.32. The radial velocity showed an inward motion at the bottom of the vessel, and an outward motion at the middle of the vessel, suggesting a vertically recirculating flow. The shear stress was positive within the radius of the stir bar, and near-zero outside of this, with a large negative drop near the vessel wall. The PIV data and ADV data were compared and good agreement was found in the ϑ velocities, as well as the r fluctuating components. The r velocities and ϑ fluctuating components were in significant disagreement. The experimental efforts herein produced a reliable characterisation of the principle fluid motion and shearing forces within a cylindrical vessel under the stirring action of a magnetic stir plate and hexagonal stir bar.

M.6 References

[1] Sommerfeld, M.; Decker, S. (2004). "State of the art and future trends in CFD simulation of stirred vessel hydrodynamics", Chemical Engineering and Technology, v. 27, pp. 215-224.

[2] Lamberto, D. J.; Alvarez, M. M.; Muzzio, F. J., 1999. "Experimental and computational investigation of the laminar flow structure in a stirred tank", Chemical Engineering Science, v. 54, pp. 919-942.

[3] Hall, J. F.; Barigou, M.; Simmons, M. J. H.; Stitt, E. H., 2004. "Mixing in unbaffled high-throughput experimentation reactors", Industrial Engineering Chemical Research, v. 43, pp. 4149-4158.

[4] Montante, G.; Moštěk, M.; Jahoda, M.; Magelli, F., 2005. "CFD simulations and experimental validation of homogenisation curves and mixing time in stirred Newtonian and pseudoplastic liquids", Chemical Engineering Science, v. 60, pp. 2427-2437.

[5] Hadjiev, D.; Sabiri, N. E.; Zanati, A., 2006. "Mixing time in bioreactors under aerated conditions", Biochemical Engineering Journal, v. 27, pp. 323-330.

[6] Yoon, H. S.; Sharp, K. V.; Hill, D. F.; Adrian, R. J.; Balachandar, S.; Ha, M. Y.; Kar, K., 2001. "Integrated experimental and computational approach to simulation of flow in a stirred tank", Chemical Engineering Science, v. 56, pp. 6635-6649.

[7] Dianrong, G.; Yiqun, W., 2004. "Two-dimensional PIV experimental investigation of mean flow field in stirred tank with Rushton turbine", Chinese Journal of Mechanical Engineering, v. 40, n. 12, pp. 192-198.

[8] Dhainaut, M.; Tetlie, P.; Bech, K, 2005. "Modeling and experimental study of a stirred tank reactor", International Journal of Chemical Reactor Engineering, v. 3, article A61

[9] Wu, Y.; Min, J.; Li, Z.; Gao, Z., 2007. "Particle image velocimetry (PIV) study of the flow structure in a stirred tank", Journal of Beijing University of Chemical Technology, v. 34, n. 6, pp. 561-565.

[10] Chung, K. H. K.; Barigou, M.; Simmons, M. J. H., 2007. "Reconstruction of 3-D flow field inside miniature stirred vessels using a 2-D PIV technique", Transactions of IChemE, Part A, Chemical Engineering Research and Design, v. 85 (A5), pp. 560-567.

[11] Kilander, J.; Rasmuson, A., 2005. "Energy dissipation and macro instabilities in a stirred square tank investigated using an LE PIV approach and LDA measurements", Chemical Engineering Science, v. 60, pp. 6844-6856.

[12] Torré, J.; Fletcher, D. F.; Lasuye, T.; Xuereb, C., 2007. "Single and multiphase CFD approaches for modelling partially baffled stirred vessels: comparison of experimental data with numerical predictions", Chemical Engineering Science, v. 62, pp. 6246-6262.

[13] Xiao, H.; Takahashi, K (2007). "Mixing characteristics in the horizontal non-baffled stirred vessel in low viscosity fluid", Journal of Chemical Engineering of Japan, v. 40, pp. 679-683.

[14] Keane, R. D.; Adrian, R. J., 1992. “Theory of cross-correlation analysis of PIV images,” Applied Scientific Research, v. 49, pp. 191-215.

[15] Oerlemans, J., 2005. “Extracting a climate signal from 169 glacier records,” Science, v. 308, pp. 675-677.

Appendix N Flow solver verification and validation

As mentioned earlier in Chapter 4, verification and validation are distinct concepts. Verification is the process of ensuring the algorithm is functioning properly. Validation is the process of comparing the algorithm output against experimental data. The former is used during algorithm and model development. The latter is used prior to numerical experimentation.

N.1 Verification

CRAFTS employs two flow solver algorithms: PISO and PIMPLE. These are well-established, standard algorithms in the CFD knowledgebase, which are in regular use today. The PISO algorithm was published in 1986 (Issa), and the PIMPLE algorithm is a hybrid between the PISO algorithm and the SIMPLE algorithm, the latter of which was published in 1972 (Caretto et al.).

Algorithm verification is used to ensure the algorithm is performing correctly. There are no comparisons against experiment. There are no statistical analyses. There are no interpretations of errors. Either the algorithm performs correctly, or it does not.

The easiest way to verify the algorithm is to compare the output from the new implementation against a known good implementation. Since PISO and PIMPLE are such common algorithms, there is a plethora of implementations available; in fact, OpenFOAM has its own built-in implementations of these algorithms. Therefore the flow solver verification becomes the trivial task of performing identical simulations using both the CRAFTS implementations and the OpenFOAM implementations, and comparing the output. There is no room for error in this kind of verification. The output from each implementation must be identical in every way. If the algorithm is correct, every digit of every number in the output will be identical between implementations. Therefore errors are easy to detect.

Since the goal is ensuring two implementations of the same algorithm produce the same output, the details of the simulation used to test these algorithms is largely inconsequential. For example, a viscosity of 8.90 x 10^{-4} [kg m^{-1} s^{-1}] will give results similar to the behaviour of water; a viscosity of 1.983 x 10^{-5} [kg m^{-1} s^{-1}] will give results similar to the behaviour of air; and a viscosity of 1,000,000 [kg m^{-1} s^{-1}] will give non-physical results. Regardless of the viscosity setting, both implementations of the algorithm must still produce identical results, and verification can be achieved.

The simulation details do matter as far as testing all aspects of the algorithm. For instance, setting gravity to 0 will not test the buoyancy source terms in a buoyant solver. Therefore, it is important to choose a case that will actuate all the numerical machinery of the algorithm under consideration.

The test case used to verify these algorithms is a lid-driven flow within a cubic cavity measuring 0.1 [m] on each side. The mesh is a 20x20x20 orthogonal grid. The case uses no-slip wall boundary conditions for all surfaces, with the top surface having a velocity of 1 [m s^{-1}] in the *x*-direction. The case uses a *k*-*ε* turbulence model. The fluid conditions use a dynamic viscosity of 1.00 x 10-5 [m^2 s^{-1}], corresponding approximately to air. The case uses a fixed timestep of 0.005 [s] and runs for 10 iterations.

The PISO and PIMPLE algorithms in CRAFTS exactly duplicated the output from OpenFOAM's built-in PISO and PIMPLE solvers in all cases. Therefore, the CRAFTS flow solver algorithms have been verified.

N.2 Validation

Before performing any numerical experimentation in CFD, there are three prerequisites that must be met:

1. model validation;

2. grid independence; and
3. convergence independence.

Should this also be a transient simulation, a fourth must also be met:

4. timestep independence.

If these steps are not accomplished, the numerical experiments cannot be trusted to produce valid results. This is always necessary regardless of the CFD software being used, including all commercially available packages.

Model validation is performed to ensure the model is producing reasonably realistic results. Validations require experimental data, or in rare cases, direct numerical simulation data. Validations involve statistical analyses. Since no model captures reality perfectly, there will be some level of error, and validations involve interpreting these errors.

The research described in this dissertation advances the state-of-the-art for anaerobic digestion modelling. It does not, however, arrive at a practical model that can be used for numerical experimentation. Therefore, it is premature to perform a validation on the flow model; however, early in the development of CRAFTS, before the numerical stiffness problems became evident, a validation was performed. Before presenting the results, it should be emphasized that the flow model algorithms are standard CFD algorithms, and they have already been verified against the implementations that ship with OpenFOAM, as detailed in Section N.1 on page 375. Therefore, should there be any concerns with the results presented below, they do not indicate a problem with the CRAFTS implementation; rather, they only indicate a problem with the validation. There is a vast array of possible problems that could affect a validation, such as poor mesh quality, incorrect fluid properties, or inadequate discretization schemes.

The validation compares the flow solver output against published experimental results (Ampofo et al., 2003). The case involves natural convection in a rectangular air-filled cavity with horizontally opposed hot and cold walls. The geometry is shown in Figure A27. The remaining walls are adiabatic.

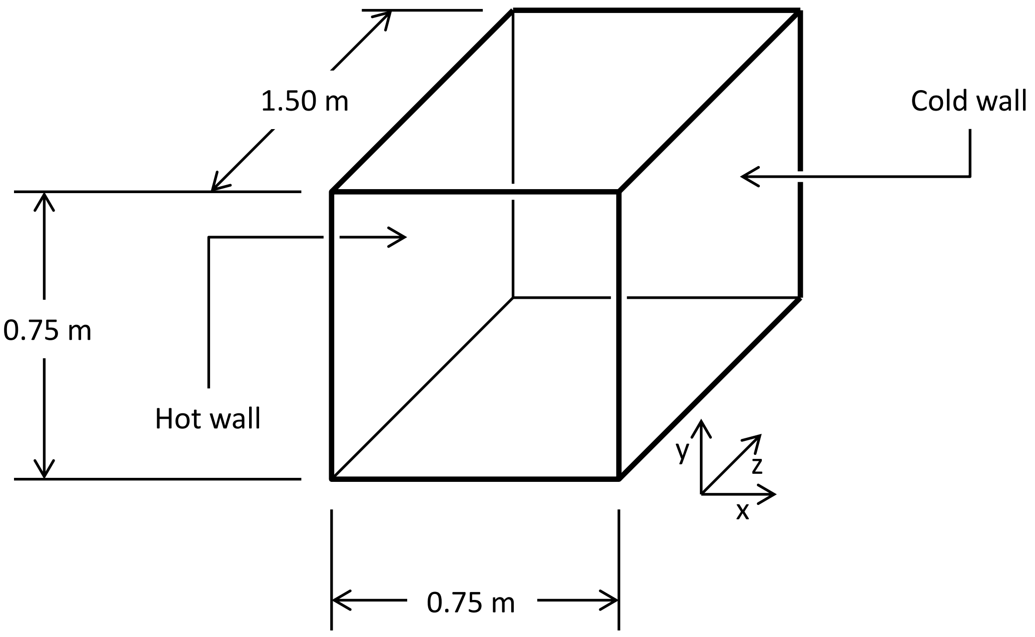


Figure A27: Cavity geometry

The experiment maintains the hot and cold walls at 50 °C and 10 °C, respectively. Ampofo et al. (2003) also provide the temperature profiles of the top and bottom horizontal walls, which are incorporated into the boundary conditions of the simulation. The published results are from a plane slicing the 1.50 [m] depth dimension in half. The simulation uses an orthogonal grid with 100 x 100 x 20 cells in the *x*, *y*, and *z* direction, respectively. The mesh has boundary-layer grading, wherein each cell height is 50% greater than the previous, moving away from the nearest wall. The simulation uses a k-ε turbulence model. Fluid properties include:

- laminar viscosity, $\nu = 1.568 \times 10^{-5}$ [$m^2 s^{-1}$];
- thermal expansion coefficient, β = 3.315 x 10-3 [K^{-1}];

- reference temperature, $T_{ref} = 303.15$ [K];
- laminar Prandtl number, $Pr_l = 0.712$; and
- turbulent Prandtl number, $Pr_t = 0.85$.

The simulation uses a timestep of 0.01 seconds, and runs until steady state. The results are shown along a line along the *x*-axis at the centre of the cavity. The *x*- and *y*-components of velocity are shown in Figure A28 and Figure A29 respectively. The former shows some differences between experiment and simulation, whereas the latter shows an excellent match. These plots break the velocity down into components, and the *y*-component is up to two orders of magnitude greater than the *x*-component. The *x*-component, on its own, seems to suggest a poor match; however, this is misleading. To clarify, a combined total magnitude profile, shown in Figure A30, shows how close the velocity field matches the experimental values.

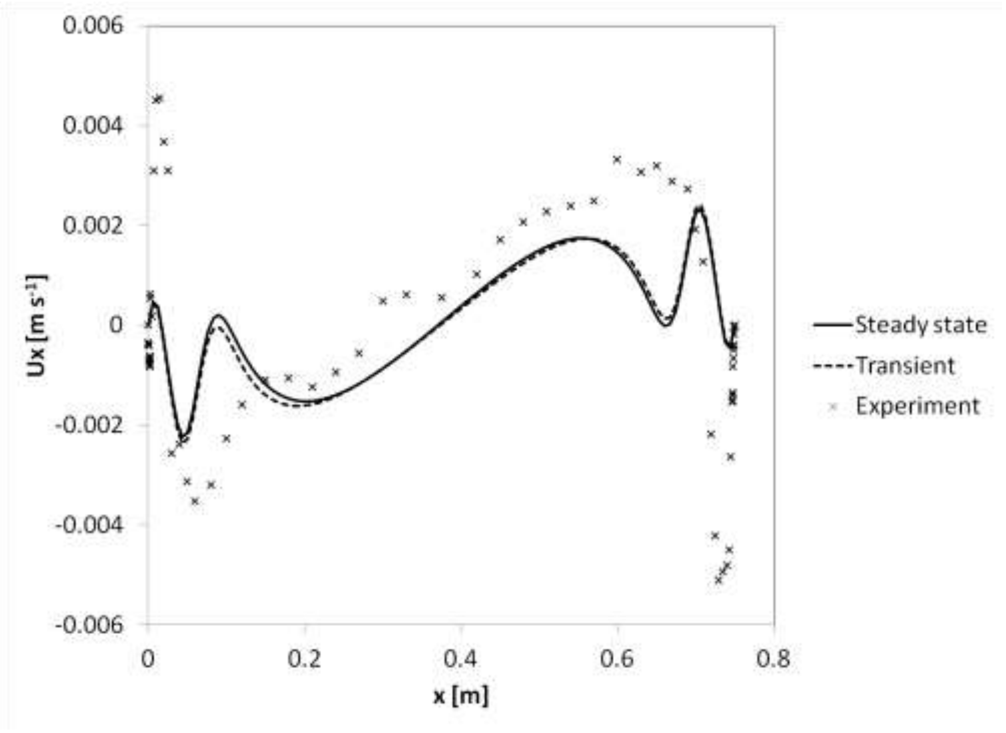


Figure A28: *x*-component of velocity

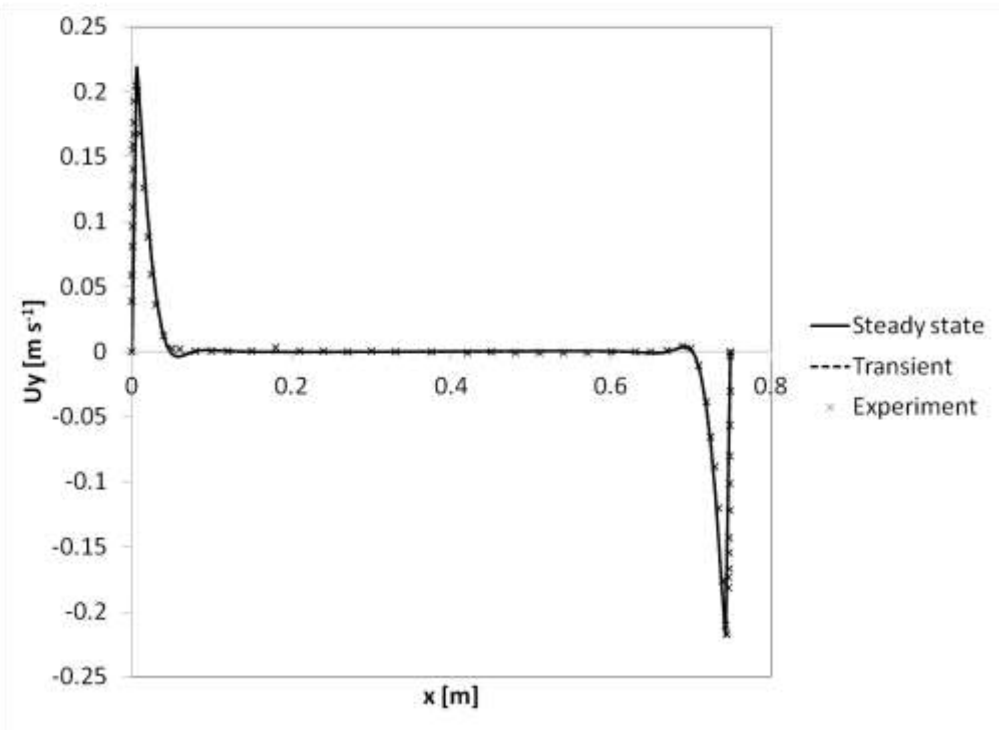


Figure A29: *y*-component of velocity

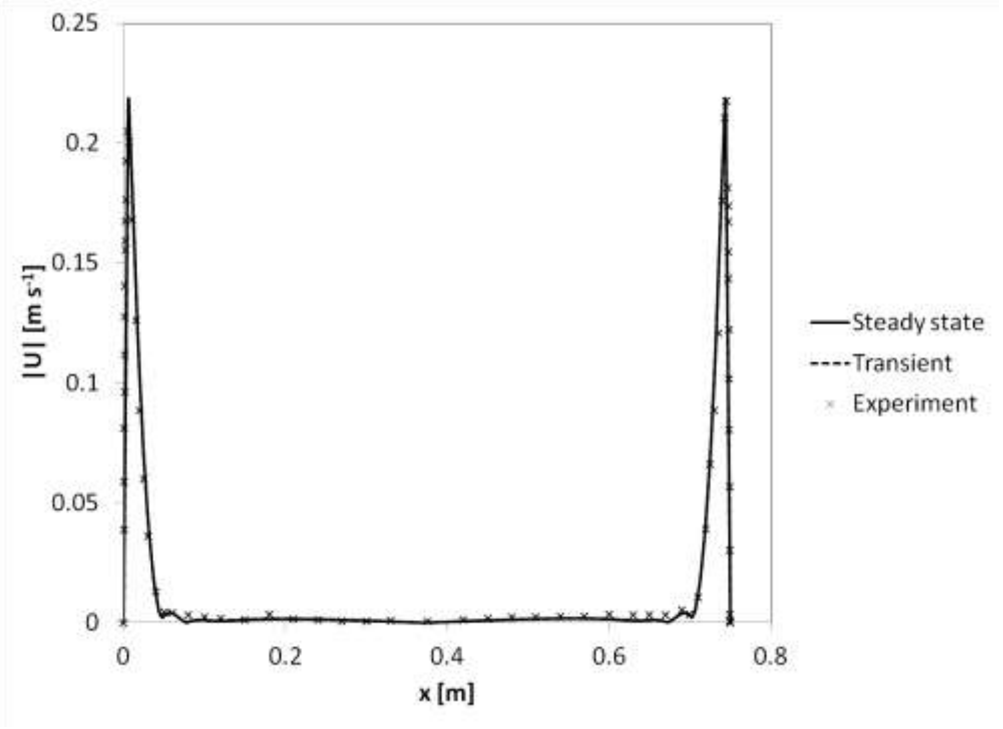


Figure A30: Velocity magnitude profile

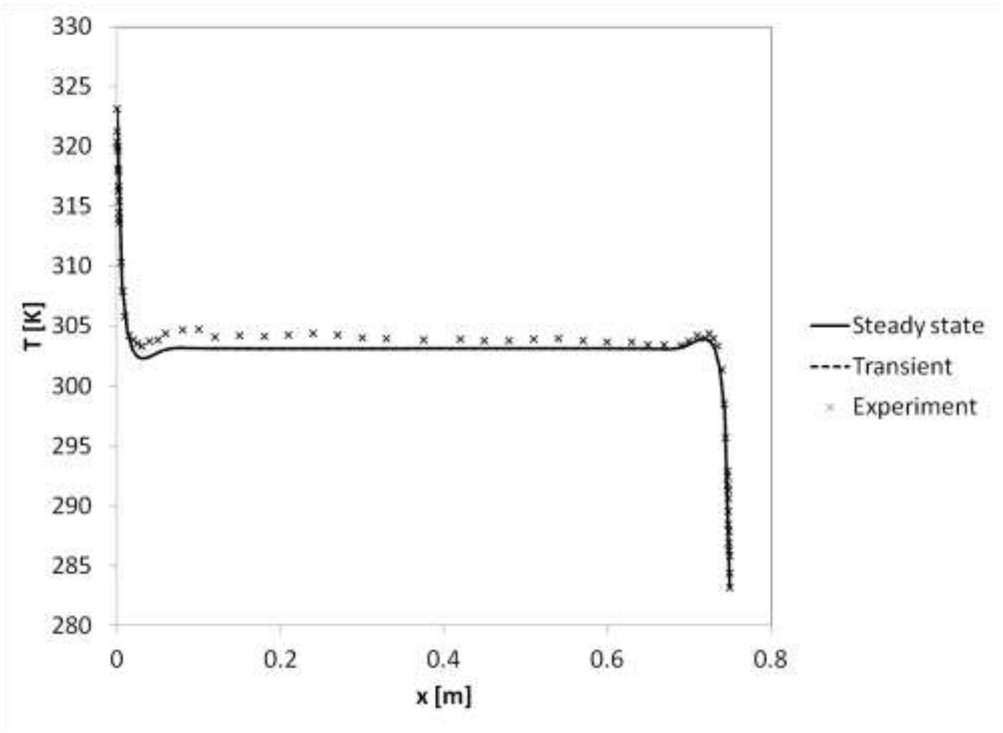


Figure A31: Temperature profile

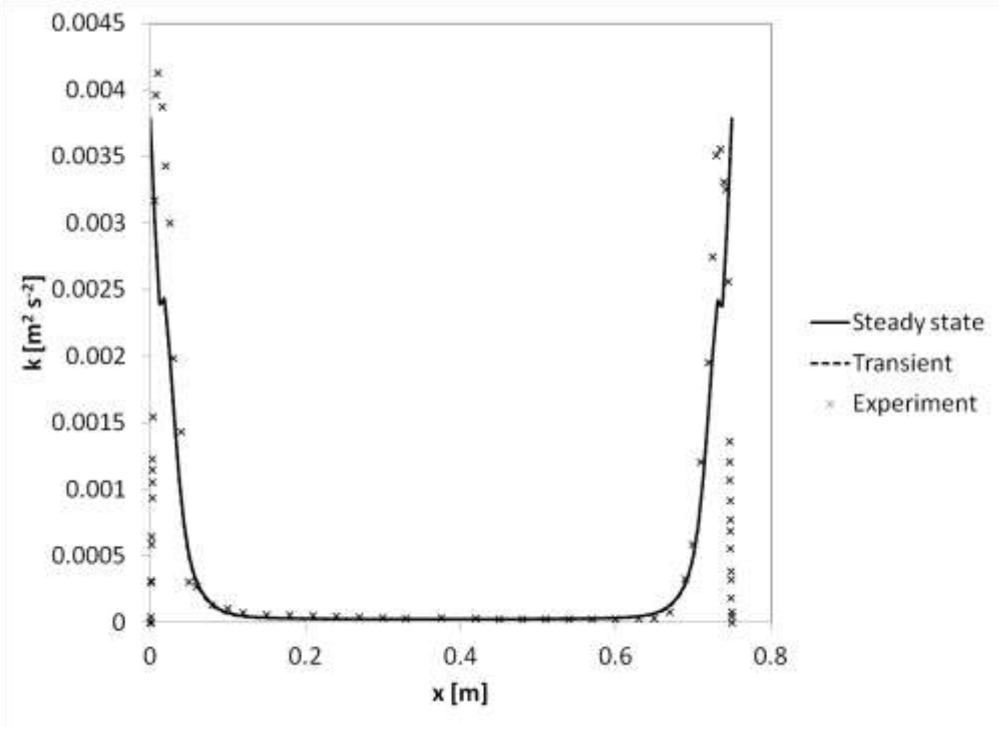


Figure A32: Turbulence kinetic energy profile

The temperature profile, Figure A31, shows a close match between experiment and simulation. The simulation under-predicts the temperature in the middle of the cavity, but the boundary layer is accurately predicted. The turbulence kinetic energy profile, Figure A32, shows a good match throughout the volume, including the boundary layers. However, simulations do not capture the sudden drop to zero at the wall. An adjustment in the *k* boundary conditions may improve this issue. The plot of the *x*-component of velocity shows slight differences between the transient and steady state simulations; in all other plots they are on top of one another.

In conclusion, the steady state and transient flow models are in good agreement with experimental values, and have therefore been validated for natural convection flow conditions inside a buoyant cavity.

Appendix O OpenFOAM tutorial

This section has also been published on the OpenFOAM wiki (openfoamwiki.net) and is therefore in the public domain.

O.1 What is OpenFOAM?

OpenFOAM is an open source software suite written in C++, designed primarily around three-dimensional continuum mechanics, although its capabilities extend beyond this. The library headers for OpenFOAM use their own syntax and operators to build a framework for fully-realized vector field mathematics in three-dimensional Cartesian coordinates. Tensors are also fully supported in these operations, although tensor calculus is not included.

O.1.1 What's FOAM got to do with it?

In this world, many acronyms cleverly spell what they are about. Make no mistake, OpenFOAM is not one of those acronyms. Indeed several oddly chosen acronyms can be found within OpenFOAM's reach. For instance, the combined PISO and SIMPLE solver is has the unfortunate name "pimpleFoam". FOAM itself stands for **F**ield **O**peration **a**nd **M**anipulation, referring to the mathematical nature of the code, not bubbly frothiness. So it really has nothing to do with foam... unless, of course, you are using it to model foam. Perhaps the developers were more concerned about what went into the code, rather than what to call it.

O.1.2 Origins

OpenFOAM, originally called FOAM, was proprietary software developed by Nabla Ltd.. In 2004 Nabla Ltd. ceased operation and released FOAM as open source under GNU General Public License, under the name "OpenFOAM". At this point two independent companies were created:

- OpenCFD, Ltd. (maintains official OpenFOAM); and
- Wikki, Ltd. (maintains OpenFOAM-extend).

Both of these companies were started by some of FOAM's original developers and have nothing to do with one another[1]. Each company maintains their own variation of OpenFOAM.

O.1.3 OpenFOAM variations

Although there are several variations to OpenFOAM, the two main ones are those maintained by Wikki, Ltd. and OpenCFD, Ltd.:

- Official OpenFOAM; and
- OpenFOAM-extend.

O.1.3.1 Official OpenFOAM

The Official OpenFOAM project is maintained by OpenCFD, Ltd., the trademark holder of the name OpenFOAM. This variation is likely the most installed. Some features are available in this variation that are not available in the OpenFOAM-extend variation. However, OpenFOAM-extend has consistently incorporated these changes into its source code, albeit one or two years later. At present, features that only appear in the Official OpenFOAM include:

- runtime selectable turbulence wall functions - alternate wall functions can be used without rewriting the turbulence models;
- density-based thermophysical model - allows for the full energy equation with liquids; and
- Boussinesq buoyancy SIMPLE solver.

For the full list, read the release notes[2] for each version number that exceeds OpenFOAM-extend.

O.1.3.2 OpenFOAM-extend

The OpenFOAM-extend project is maintained by Wikki, Ltd.. It is usually referred to by the latest version number, followed by 'dev' (for example 1.6-dev).

[1] http://www.cfd-online.com/Forums/openfoam-code-other/64825-differences-between-1-5-x-1-5-dev.html#4
[2] http://www.openfoam.com/docs/release-notes.php

NOTE: Usually *dev* refers to a *developmental version* which will eventually become part of the release.

In OpenFOAM, this is not the case: 'dev' is the name of the core component of the OpenFOAM-extend project.

This project is open to community contributions, none of which get adapted into the Official OpenFOAM. Therefore the dev-version has a vast library of capabilities unmatched by the Official OpenFOAM project, including:

- GGI interfaces - useful for rotating reference frames, and stitching together multiple meshes;
- groovyBC - allows arbitrary boundary conditions to be specified, without requiring coding / compiling;
- newer rectangular matrix solvers; and
- pyFOAM - a front end that makes managing OpenFOAM cases easier, i.e. it makes available many features that users of other software suites take for granted, such as:
    - easily changing the case setup; and
    - plotting the residuals on a graph as the solver is running.

Many of the contributions are compatible with both variations of OpenFOAM, and can be manually installed as desired.

The Official OpenFOAM and OpenFOAM-extend are two different open source projects. The only reason they have not diverged is because OpenFOAM-extend continues to adapt the new releases of Official OpenFOAM into its own code. Therefore, new features appearing in Official OpenFOAM eventually become part of OpenFOAM-extend. These adaptations can take one or two years. This practice continues[3].

[3] http://www.cfd-online.com/Forums/openfoam-bugs/67274-openfoam-1-6-ggi-mergemeshes.html#post254554

O.1.3.3 Other variations

Other variations include:

- FreeFOAM, a project with the intent of making OpenFOAM platform independent; and
- OpenFOAM MS Windows binary release.

O.2 OpenFOAM file structure

OpenFOAM is not a single standalone executable. It has a set of libraries, and a set of applications, including solvers and utilities. This architecture allows the libraries to be used independently. It also means there is not a single consistent interface or case structure.

O.2.1 Directory structure

The directory structure is:

- **applications** - contains the source code for all the executables
- bin
- doc
- etc
- lib
- **src** - contains the source code for all the libraries
- tutorials
- wmake

The most important directories are in bold - these contain all the source code.

O.2.1.1 Applications

Applications include solvers and utilities. There are dozens of each of these. Some standard solvers include:

- scalarTransportFoam - solves laminar, incompressible transport of a passive scalar;

- simpleFoam - steady-state, incompressible solver with RAS turbulence; and
- pisoFoam - transient, incompressible solver with RAS turbulence.

O.2.1.2 Libraries

The full OpenFOAM library contains close to one million lines of code. To keep it organized, it is subdivided into dozens of smaller libraries. These include:

- **OpenFOAM** - the main library;
- **finiteVolume** - contains classes specific to the finite volume method;
- **themophysicalModels** - classes specific to themodynamics, such as enthalpy, mixtures, and associated functions;
- **transportModels** - classes related to transportProperties, such as viscosity models;
- **turbulence models**; and
- many others.

Some directories within the libraries are named `lnInclude`. These directories contain automatically generated links to other files. These directories can be ignored - they are for compiling purposes, to greatly reduce the size of the include directory tree.

O.3 How to use OpenFOAM

OpenFOAM does not have a graphical user interface (GUI). Everything is done in text files and console commands. OpenFOAM does't even have a 'case file'. Instead it has a 'case directory'. A typical case directory structure might contain:

- **system** directory containing:
    - **controlDict** - file containing the main settings such as start time, end time, time step, and so on;

    - **fvSchemes** - file specifying which discretization schemes to be used, can be broken down by terms in the equations;
    - **fvSolution** - file specifying which solver / smoother / preconditioners are to be used for the matrix solutions;
- **constant** directory containing:
    - **transportProperties** - file specifying fluid properties such as viscosity;
    - **polymesh** - directory containing mesh files (this is the initial mesh, in the case of mesh motion);
- **0** - directory containing the initial conditions; and
- **[time value]** - data is output into time directories whose names correspond to the timestep.

Each solver requires different information in the case directory. Most solvers have a sample case in the tutorial section. An easy way to set up a case is to:

1. Locate the tutorial associated with your desired solver;
2. Copy the directory contents into your run folder; and
3. Rename and modify as required.

O.3.1 Meshes

OpenFOAM has some meshing capabilities, but it also has mesh import capabilities. A good way to start is to create a mesh in another format and convert it over using the appropriate convert utility. It supports mesh conversion from:

- STAR-CDPROSTAR format (starToFoam);
- GAMBIT .neu format (gambitToFoam);
- I-DEAS mesh in ANSYS .ans format (ideasToFoam); and
- CFX .geo mesh files (cfxToFoam).

O.3.2 Post-processing

OpenFOAM uses Paraview as a post-processor, which does have some capabilities. However, it also supports conversion to other formats, including:

- Fluent (foamMeshToFluent; foamDataToFluent);
- FieldView (foamToFieldView);
- Ensight (foamToEnsight); and
- Tecplot (OpenFOAM 1.6.X only).

Furthermore there are means to sample 1D and 2D slices of data and export it to a variety of formats.

O.4 Matrices in OpenFOAM

Linear equations resulting from the discretization method are solved in OpenFOAM using matrix methods. OpenFOAM has a specialized strategy for handling these matrices.

O.4.1 Overview

The discretization process results in the equation:

$$\mathbf{A}\boldsymbol{\Psi} = \mathbf{B}, \tag{A101}$$

where:

- **A** is the discretization matrix, which holds the matrix coefficients;
- $\boldsymbol{\Psi}$ is the vector array of any variables being solved for; and
- **B** is the discretization source term.

The A component is an *N* x *N* square matrix, where *N* is the number of equations produced by the discretization process. OpenFOAM has developed a specialized strategy to handle this matrix. Important components of this strategy include:

- lduMatrix (lower - diagonal - upper - matrix storage class);
- lduAddressing (an indexing class); and

- fvMatrix (contains components specific to the finite volume method).

0.4.2 lduMatrix

The lduMatrix class stores the matrix coefficients in three arrays:

- `scalarField *lowerPtr_;`
- `scalarField *diagPtr_;`
- `scalarField *upperPtr_;`

These arrays are Fields that contain all the non-zero entries in the coefficient matrix.

0.4.2.1 Diagonals

Every cell has one and only one diagonal coefficient, none of which are zero. Therefore the diagonal list contains N values, where N is the number of cells in the mesh. These are denoted:

- as discretization coefficients, $a_p \neq 0$; or
- as matrix coefficients, $a_{ii} \neq 0$.

0.4.2.2 Upper and lower triangles

The upper and lower triangles have (*N* - 1)*N* possible values, however most are zero. The non-zero values correspond to cell-pairs that influence one another. OpenFOAM uses a small computational molecule, therefore only adjacent cells influence one another. In other words, there are two non-zero coefficients for every *internal face* in the mesh: one that appears in the lower triangle; and one that appears in the upper triangle. For instance, given a simple 3 x 3 mesh:

1	2	3
6	5	4
7	8	9

Figure A33: 3 x 3 mesh

The non-zero coefficients for OpenFOAM would be:

i	1	2	3	4	5	6	7	8	9
1	X	N				N			
2	O	X	N		N				
3		O	X	N					
4			O	X	N				N
5		O		O	X	N		N	
6	O				O	X	N		
7						O	X	N	
8					O		O	X	N
9				O				O	X

Figure A34: Non-zero coefficients

Some mathematical operations require distinguishing between the two coefficients as they are sometimes opposite in sign (for instance when fluxes are involved). OpenFOAM's mesh design anticipated this requirement, which is why every face has an *owner cell* and a *neighbour cell*. Owner, neighbour and diagonal coefficients are defined as:

$$a_{ij} = \begin{cases} \text{owner}, & i > j \\ \text{diagonal}, & i = j \\ \text{neighbour}, & i < j \end{cases} \tag{A102}$$

where:

- *i* is the matrix row number (corresponds to cell index); and
- *j* is the matrix column number (corresponds to cell index).

With this definition, the *lower triangle* has the *owner* coefficients; and the upper triangle has the *neighbour* coefficients. They are denoted:

- as discretization coefficients, $a_{r,i}$, where *r* indicates "related[4] cells";
- as matrix coefficients:
  - owner coefficients are: $[a_{ij}]_{i>j} = a_o$; and
  - neighbour coefficients are: $[a_{ij}]_{i<j} = a_o$.

With this consistently applied across the mesh, a direction for positive surface-normal vectors is defined: a positive surface-normal vector (such as fluxes) points away from the *owner* cell, and towards the *neighbour* cell.

0.4.2.3 Diagonal, symmetric and asymmetric matrices

It is mandatory to define the diagonal; the others are not required. If only one of the lower or upper triangles are defined, the matrix is assumed to be symmetric. If both are defined, it will assume the matrix is asymmetric, even if they are equal.

Table A15 – Matrix pointers

lowerPtr	diagPtr	upperPtr	Result
Yes/No	No	Yes/No	Invalid matrix – error thrown
No	Yes	No	Diagonal matrix
Yes	Yes	No	Symmetric matrix
No	Yes	Yes	Symmetrix matrix
Yes	Yes	Yes	Asymmetric matrix

CAUTION! Accessing the opposite pointer without a `const` modifier will convert the matrix to an asymmetric matrix. To modify the off-diagonal of a symmetric matrix, first test which pointer is active using `hasUpper()` and `hasLower()`.

[4] What I'm calling *related cells* is conventionally called *neighbours*. But OpenFOAM has a different meaning for *neighbours*, so the term *related cells* is used.

O.4.2.4 OpenFOAM code example

For example, in calculating the diffusion of a quantity φ, the governing equation is:

$$\nabla \bullet \Gamma \nabla \mathbf{U} = 0. \tag{A103}$$

This is the Laplacian, which is calculated in the finite volume method using fvm::laplacian. The discretization coefficients are:

$$a_r = -\frac{\Gamma S_f}{\delta}, \text{ and} \tag{A104}$$

$$a_p = \sum_r a_r. \tag{A105}$$

To implement this in OpenFOAM:

```
GeometricField<scalar, fvsPatchField, surfaceMesh> gammaMagSf ( gamma*mesh.magSf() );
fvMatrix<Type>& fvm;
fvm.upper() = deltaCoeffs.internalField() * gamma*mesh.magSf().internalField();
fvm.negSumDiag();
```

Note:

- `fvm.lower` is never mentioned - this is because it is equal to `fvm.upper` and a symmetric matrix is being created;
- `fvm.upper` is opposite in sign to a_r above - this is because `fvm.lower` are matrix coefficients, whereas a_r are discretization coefficients.

O.4.3 lduAddressing

The lower, diagonal and upper coefficients are all scalarFields (basically a list), but they are indexed differently. The diagonal coefficients are indexed by *cell index*; whereas the upper and lower triangles are indexed by *face index*. In order to get the indices to match, an addressing array is provided for the lower and upper triangles to translate their face index into a corresponding cell index. This addressing array is called an *lduAddressing* array.

O.4.3.1 When do you need it?

lduAddressing is needed anytime you need to perform an operation on all rows of an `lduMatrix` that involves both *diagonal* and *off-diagonal* coefficients.

O.4.3.2 How do you use it?

To use *lduAddressing*:

- separate operations involving the lower triangle from operations involving the upper triangle;
- loop over all the faces in the mesh;
- upper and lower triangle coefficients use the loop index - e.g. `upper[facei] =` ...
- diagonal coefficients use the lduAddressing coefficient of the face index:
    - o `diag[l[facei]] -= lower[facei]`, where `l` is the *lduAddressing* for the lower triangle; and
    - o `diag[u[facei]] -= upper[facei]`, where `u` is the *lduAddressing* for the upper triangle.

O.4.3.3 Example

For example, the `lduMatrix::negSumDiag` operation is frequently required. This is where the diagonal is the negative sum of its neighbours:

$$a_p = -\sum_r a_r. \tag{A106}$$

The guts of its code is:

```
for (register label face=0; face<l.size(); face++)
{
    Diag[l[face]] -= Lower[face];
    Diag[u[face]] -= Upper[face];
}
```

Where:

- `Diag` is the diagonal coefficient;
- `l` is the *lduAddressing* for the lower triangle;

- `u` is the *lduAddressing* for the upper triangle;
- `Lower` is the lower triangle; and
- `Upper` is the upper triangle.

0.4.4 fvMatrix

Since OpenFOAM implements multiple numerical strategies such as finite difference, finite area, finite element, and so on, everything that is specific to any one of these strategies is stripped out of the base class lduMatrix. This facilitates code reuse.

fvMatrix is the finite volume extension of lduMatrix. It includes the source, **B**, as well as a reference to the field, $\mathbf{\Psi}$ from $\mathbf{A\Psi} = \mathbf{B}$. It handles what happens on the boundaries, and also implements specific operations that occur frequently in finite volume algorithms, such as fvMatrix::H.

0.5 Compile environment

OpenFOAM is designed for Linux or Unix, and is not platform independent; however there has been some success porting it to Mac and Windows. OpenFOAM uses `wmake`, a custom compile environment that automatically generates compile commands (e.g. g++) and linker commands (e.g. ld) based on the contents of a `Make` directory, and environment variables.

0.5.1 Using wmake

To use wmake, create a `Make` directory in the root of your library or application that needs to be compiled. Within that directory, create two text files: `files` and `options`. Once these files have all the required information, the `wmake` commands are available:

- `wclean all` (usesd to delete compiled objects - for a rebuild); or
- `wmake` (for an application); or
- `wmake libso` (for a library).

0.5.1.1 files

The `files` file contains a list of compile targets, as well as the destination binary name. The compile target list is a list of file names that either:

- have a path relative to the directory where `Make` resides; or
- have symbolic links within one of the `lnInlude` directories that are included in the `options` file.

The destination binary name is specified using either:

- "`LIB = path and filename`" for libso compiles; or
- "`EXE = path and filename`" for application compiles.

OpenFOAM's convention for these destinations is:

- for libso: `LIB = $(FOAM_LIBBIN)/libraryName`, where `libraryName` is the name of the library prefixed with `lib`. For instance, for `finiteVolume` the name is `libfiniteVolume`; and
- for applications: `EXE = $(FOAM_APPBIN)/applicationName`, where `applicationName` is the name of the application, no prefixes are necessary.

Custom applications and libraries

The system variables `FOAM_LIBBIN` and `FOAM_APPBIN` point to file paths that are intended to hold only files associated with the OpenFOAM release. Therefore if you have a custom library or custom application, use `FOAM_USER_LIBBIN`, and `FOAM_USER_APPBIN`.

Note: If you have created a custom library, you must add `-L $(FOAM_USER_LIBBIN)` to the `EXE_LIB` section of the options file for any application that wants to use it.

0.5.1.2 options

The `options` file includes the command line options that are given to the compiler and linker. There are

- `EXE_INC` - any additional include paths are specified here using `–I[path name]`. E.g.:

```
EXE_INC = \
    -I$(LIB_SRC)/dynamicMesh/lnInclude \
    -I$(LIB_SRC)/finiteVolume/lnInclude
```

- `LIB_LIBS` - used when compiling a library. Specifies any additional libraries that are to be linked in, (without the lib prefix). E.g.:

```
LIB_LIBS = \
    -ldynamicMesh \
    -lfiniteVolume
```

- `EXE_LIBS` - used when compiling an application. Specifies any libraries that will be needed during the compile, (without the `lib` prefix). E.g.:

```
EXE_LIBS = \
    -lfiniteVolume \
    -llduSolvers
```

Any other options supported by your compiler or linker can also be used.

Note: The `OpenFOAM` library (i.e. `src/OpenFOAM`) and `OSspecific` library (i.e. `src/OSspecific`) are included by default. Libraries and `lnInclude` directories for these do not need to be specified in the `files` or `options`.

0.5.2 Using an IDE

There is no easy to use integrated development environment. Trying to incorporate OpenFOAM into an IDE is complicated:

- the OpenFOAM file structure:
  - composed of dozens of stand alone applications and libraries;
  - some dependency on third-party source; and
  - files and options files contain inconsistent parameters for compiling;
- file inclusions:
  - the include directory tree is very large (with nearly 5000 files); and
  - there are dozens of filename collisions within the include tree.

Despite this, some users have reported success in using Eclipse[5].

O.6 objectRegistry

The `objectRegistry` is a hierarchical database that OpenFOAM uses to organize its model-related data. It is complemented by `IOobject`, and `regIOobject`. `IOobject` is a class that provides standardized input / output support, as well as giving access to `runTime`, the root of the `objectRegistry`. `regIOobject` automatically manages the registration and deregistration of objects to the `objectRegistry`.

O.6.1 Why is it needed?

It is needed to simplify the communication between the solvers and their data. OpenFOAM implements many modelling solution methods such as:

- finite volume method;
- finite differencing method;
- finite area method; and
- Ordinary Differential Equation solvers.

It achieves this through abstraction and templating - or in other words - by chopping up its code into many small pieces that can be used by all these solution methods. To make this possible, all common elements of the modelling data are wrapped up into a single universal framework. This is the `objectRegistry`.

O.6.2 When do you need it?

Anytime you are working with modelling data. For example, this includes:

- physical models;
- field data;

[5] http://openfoamwiki.net/index.php/Howto_Use_OpenFOAM_with_Eclipse

- dictionaries; and
- the mesh itself.

It is hard to imagine a situation where you will not need it.

O.6.3 How does it work?

O.6.3.1 Overview

The main components of the objectRegistry are:

- the `IOobject`;
- the `regIOobject`; and
- the `objectRegistry`.

The `objectRegistry` is a hash table with a few additional member variables and methods. It catalogues `regIOobject` pointers by `IOobject::name_`, and helps manage their read and write operations. Every object it catalogues exists in-memory and usually on-disk in some form, the nature of which is governed by read and write options carried out by the individual `regIOobjects`.

O.6.3.2 Hierarchy

There are actually two hierarchies in effect with the `objectRegistry`:

- the hierarchy of objects held *in-memory*; and
- the hierarchy of objects *on-disk*.

Hierarchy *in-memory*

An `objectRegistry` is itself a `regIOobject`, which means it can be a registered member of another `objectRegistry`. This leads to hierarchies. For example, `scalarTransportFoam` creates this `objectRegistry` hierarchy when it initializes (in order of appearance):

```
runTime                  //Time              (objectRegistry)
|-controlDict            //IOdictionary      (regIOobject)
```

```
`-mesh                  //fvMesh            (objectRegistry)
 |-fvSchemes            //IOdictionary      (regIOobject)
 |-fvSolution           //IOdictionary      (regIOobject)
 |-points               //pointIOField      (regIOobject)
 |-faces                //faceIOlist        (regIOobject)
 |-owner                //labelIOlist       (regIOobject)
 |-neighbour            //labelIOlist       (regIOobject)
 |-boundary             //polyBoundaryMesh  (regIOobject)
 |-pointZones           //pointZoneMesh     (regIOobject)
 |-faceZones            //faceZoneMesh      (regIOobject)
 |-cellZones            //cellZoneMesh      (regIOobject)
 |-T                    //volScalarField    (regIOobject)
 |-U                    //volVectorField    (regIOobject)
 `-transportProperties  //IOdictionary      (regIOobject)
```

In-memory hierarchy of `scalarTransportFoam` `objectRegistry`*, in order of appearance.*

Every objectRegistry has:

- pointers to all its child `regIOobjects`;
- a reference to its parent `objectRegistry`; and
- a reference to the master `objectRegistry`;
    - the master is always `runTime`.

Hierarchy *on-disk*

The on-disk hierarchy depends on the read / write behaviour of the objects. `objectRegistry` objects have different read and write behaviour than "regular" `regIOobjects`. During a typical write operation, a `regIOobject` only has to write its data to a file, whereas an `objectRegistry` cycles through all its child objects, and tells them each ot write themselves. Read operations are performed on an as-needed basis.

A `regIOobject` writes its file to

```
instance_/local_/name_
```

where instance_ can be:

```
* timeName()       normally [caseDirectory]/[timeName]     e.g. reactor/0.045
* system()         normally [caseDirectory]/system         e.g. reactor/system
* constant()       normally [caseDirectory]/constant       e.g. reactor/constant
* caseSystem()     normally [caseDirectory]/system, becomes ../system in parallel runs
* caseConstant()   normally [caseDirectory]/constant, becomes ../constant for parallel
```

Functions above are given local to `runTime`

`local_` can be anything, and is optional; and

`name_` can be anything.

Similarly, *read-only* files placed in any of these paths can be read by the solver, provided it knows to look for it. The *on-disk* hierarchy written by `scalarTransportFoam` is:

```
caseDirectory           // directory
|-system                // directory
| |-controlDict         // read-only
| |-fvSchemes           // read-only
| `-fvSolution          // read-only
|-constant              // directory
| |-transportProperties // read-only
| `-polyMesh            // directory
|   |-boundary          // read-only
|   |-faces             // read-only
|   |-neighbour         // read-only
|   |-owner             // read-only
|   `-points            // read-only
`-[timeName, eg 1.5]    // directory
  |-uniform             // directory
  | `-time              // read / write
  |-phi                 // read / write
  |-T                   // read / write
  `-U                   // read / write
```

On-disk hierarchy of `scalarTransportFoam objectRegistry`*.*

0.6.4 Read-write functions

The `objectRegistry` employs virtual functions to achieve a universal framework. When `runTime.writeNow()` is called, this is approximately what happens:

1. `runTime` calls `regIOobject::write()`
2. `regIOobject` calls the virtual function `writeObject`. Normally this would call `objectRegistry::writeObject`, but `Time` has overriden this:
3. `runTime::writeObject` first writes an `IOdictionary` called time to the `[caseName]/[timeName]/constant` directory before calling the regular `objectRegistry::writeObject`
4. `objectRegistry::writeObject` cycles through all its child `regIOobjects` and calls `writeObject` on each of them.

5. If the child object is a "regular" `regIOobject`, `regIOobject::writeObject` will be called.
6. `regIOobject::writeObject` will open the `OFstream` and call `writeData`, a function that must be defined in all `regIOobjects`.

Important write functions are:

- `writeObject`:
  - o `objectRegistry` uses this to call `writeObject` on all its child `regIOobjects`;
  - o `regIOobject` uses this to call `writeData` on itself;
  - o some derived `objectRegistry` classes override this one, such as `Time`.
- `writeData`:
  - o `objectRegistry` implements this as an error-throw. Should not be called.
  - o `regIOobject` purely virtual function (i.e. = 0). All derived classes must override this.

Important read functions are:

- `objectRegistry` can scan the case directory to find any modified files that need to be reread. It accomplishes this with:
  - o `readIfModified`; and
  - o `readModifiedObjects`.
- `regIOobject`
  - o `read` - opens file stream with `readStream`, reads data with `readData`, closes the file stream;
  - o `readData` - must be overridden by the derived class, otherwise returns a fail;
  - o `readStream` - opens file stream and checks its type.

`IOobject` handles other read and write operations such as reading and writing the header.

0.6.5 IOobject versus regIOobject

An `IOobject` can be thought of as a dormant `regIOobject`. It doesn't have data input / output, and it doesn't automatically register / deregister itself from its `objectRegistry`. In fact, `IOobject`s do not really exist - there isn't a single `IOobject` class that isn't also a `regIOobject` (apart from `IOobject` itself). The separation between these two objects only comes into play when passing them as parameters:

- objects are passed as `IOobject&` almost exclusively; whereas
- `regIOobject&`s are passed only to register themselves in the `objectRegistry` (and in the constructor parameters for themselves).

An `IOobject` is a class that gives an object all the information that `objectRegistry` needs to catalogue it. This includes:

- `name_` - this serves as both:
- its filename on the hard drive; and
- its key in the `objectRegistry` hash table;
- `instance_` - the path to its file, not including the final backslash;
- `local_` - an optional local path;
- `db_` - the `objectRegistry` with which it will register or has registered;
- `rOpt` - its read option; and
- `wOpt` - its write option.

On the other hand, a `regIOobject` is an `IOobject` that also automatically registers and deregisters itself from its parent `objectRegistry`. It achieves this with its constructors and its destructor.

O.7 Memory management with tmp

`tmp` is a wrapper class that allows large local objects to be returned from functions / methods without being copied. It also allows a program to quickly clear a large object out of memory, which can be used to reduce the peak memory. `tmp` works in the background and is not usually visible at the top-level code.

O.7.1 Why is it needed?

It isn't needed at all, but it reduces the effort and increases the efficiency of your program.

When C++ returns a local object, it copies the object, returns the copy, and deletes the original. This behaviour is safe memory management; however, if you are dealing with large objects (such as a 10 million element geometric tensor field), it can be a costly process. Passing a reference won't work because the data will still be going out of scope. The challenge is getting the large dataset to survive the change in scope, something that `tmp` accomplishes.

O.7.2 When do you need it?

If you are using OpenFOAM, you are already using `tmp` without seeing it. You can explicitly use it anytime you are returning a large local object. Furthermore, `tmp` is useful anytime you can delete a large object in the middle of an algorithm to reduce peak memory.

O.7.3 How do you use it?

All objects above a single field size can use `tmp`. They all can convert from `tmp<object>` to `object`, so `tmp` is transparent. To return a large object using `tmp`, simply enclose the return type with `tmp< >`:

```
 // Original:
largeClass myFunc ()
    {
        largeClass returnMe(new largeClass);
        // Somehow fill the data in returnMe
        return returnMe;
    }

// Becomes:
tmp<largeClass> myFunc()
    {
        tmp<largeClass> tReturnMe(new largeClass);
        largeClass& returnMe = tReturnMe();
        // Somehow fill the data in returnMe
        return tReturnMe;
    }
```

To explicitly unwrap the object, use `operator()`.

Reducing peak memory is less important now than it was in the 70's, but sometimes it may be necessary. To use `tmp` for reducing memory:

- First, the algorithm must have a point where you can throw a large amount of data away, and this action reduces peak memory.
- Wrap the object with the discardable data in `tmp`:

  ```
  tmp<largeClass> discardableDataObject;
  ```

- Use the object as usual.
- When it comes time to throw away the data:

  ```
  discardableDataObject.clear();
  ```

- The memory is released.

See `simpleFoam` as an example.

0.7.4 How does it work?

0.7.4.1 Summary

Normally when returning a local object, the compiler will first call the `copy constructor` to create the return copy, then call the `destructor` on the original object to clean it up. `tmp` overrides both these functions. When the compiler calls the `copy constructor`, `tmp` only copies a pointer and a reference, leaving the data where it is. When the compiler calls the destructor, the original `tmp` redirects its pointer to `NULL`. Since the data has a new reference to it, the compiler leaves it alone, and the data survives. In practice, it gets a bit trickier.

0.7.4.2 Implementation

Overriding the `copy constructor` and `destructor` is not enough for this to work. If `tmp` is to be transparent, it must behave the way an unwrapped object would normally behave. This means sometimes `tmp` will need to be deleted properly, or its data will need to be duplicated properly. This mens it needs to detect when it is a *temporary local object*, when it is a *duplicated object*, and when it is just a *regular object* on the stack. To achieve this:

- `tmp` holds a flag to determine whether it is a temporary object.
    - The flag is `true` if `tmp` it is constructed from a pointer to an object as it would be with the `new` keyword.
    - The flag is set to `false` if `tmp` is constructed from a reference, as would happen if the object already exists, or `tmp` is pushed onto the stack.
- See `autoPtr` for a similar wrapper class suitable to pointers.
- All objects usable by `tmp` keep track of how many references point to them.
- This is achieved with the `refCount` class, and is one of the costs of implementing `tmp`.
- When being deleted, `tmp` asks its object how many references are pointing to it.
- If there is more than one, `tmp` redirects its pointer upon deletion.
- If there is only one, `tmp` does not redirect its pointer, and the data is finally deleted.

0.8 runTimeSelection

OpenFOAM's runTimeSelection mechanism is a templated implementation of an idiom in C++ known as a "virtual constructor" or "Factory Method of initialization". Most of its implementation takes place within macros, so it is hidden from the programmer.

0.8.1 Why is it needed

In fully object-oriented programming, an object will often interface with a generic base class to manipulate derived classes. This greatly simplifies top-level code, and makes adding future derived classes easy. Functions called through their base classes are known as virtual functions. C++ allows for these types of functions, *with the exception of constructors*. Therefore, if an OpenFOAM object wants to construct a derived class using a base class as its interface, it cannot.

For instance, a solver interfaces with a generic boundary condition class (`fvPatchField` for finite volume). The solver will start by creating the fields (`createFields.H`), but it does not have access to the derived boundary condition classes. So there are two options:

- make the solver interface with the individual boundary conditions; or
- implement a "virtual constructor" work-around.

There are dozens of boundary conditions and dozens of solvers, so the preference would be the virtual constructor.

0.8.2 When do you need it?

Anytime you have an object that interfaces with a base class, and you need to construct a derived class, you will need it. Either you are working with a set of OpenFOAM classes that already have runtime selection implemented (such as writing a custom boundary condition), or you are creating an entirely new set of classes and need to use runtime selection.

0.8.3 How do you use it?

It is easiest to find a derived class that most closely matches what you intend, copy it, rename it, search and replace the old name with the new name in all the files, then modify the code as necessary. This way you reduce the risk of "breaking" the runtime selection. However, it helps to understand what the components of runtime selection are.

O.8.3.1 runTimeSelection components

Say we have a base class, **Base**, and a set of derived classes, including **Derived1**, **Derived2** and **Derived3**. Also, say you want two constructors to act as virtual constructors:

```
Base(dictionary& dict);
Base(label& place, scalar& magnitude, Istream& is);
```

To keep track, we give them informal names. Call the first one MrConstructor and the second, MrsConstructor.

Runtime selection works by creating a hash table of **Derived** constructor pointers in **Base**. To accomplish this, **Base** requires hash table `creator` and `destroyer` functions as well as a subclass `AddToTable` that **Derived** uses to register its constructor. (The details of these functions are in the Detailed implementation section.)

The main components of runtime selection are:

- `declareRunTimeSelectionTable` - macro invoked in `Base.H`
- `defineRunTimeSelectionTable` - macro invoked in `Base.C`
- `addToRunTimeSelectionTable` - macro invoked in `Derived.C`
- `New` - "Selector" function added to **Base**.

declareRunTimeSelectionTable

Think of this as the header for the runtime selection changes to **Base**.

What it does

This macro adds to **Base.H**:

- `typedef`s for the constructor pointer and hash table data types;
- the hash table pointer;
- prototypes for the hash table `creator` and `destroyer` functions; and
- the full definition of the `AddToTable` subclass.

How to use it

For every constructor type (e.g. MrConstructor, MrsConstructor, etc..) there should be a `declareRunTimeSelectionTable` entry in **Base.H**:

```
declareRunTimeSelectionTable(autoPtr,baseType,argNames,argList,parList);
```

- `autoPtr` - either `tmp` (for large objects) or `autoPtr`. This is the type of wrapper class to be used to return the constructed **Derived**.
- `baseType` - this is **Base**.
- `argNames` - this is a small name you gave to the parameter list (i.e. MrConstructor and MrsConstructor above).
- `argList` - this is the full parameter list with modifiers, types and names - enclose in ()'s.
- `parList` - this is the parameter list with names only.

Example: **fvPatchField.H**

```
declareRunTimeSelectionTable
(
    tmp,
    fvPatchField,
    patch,
    (
        const fvPatch& p,
        const DimensionedField<Type, volMesh>& iF
    ),
    (p, iF)
);
```

defineRunTimeSelectionTable

There are some variations to this macro depending on the extent of templating. Think of this as the body for runtime selection changes to **Base.C**.

What it does

This macro adds to **Base.C**:

- initialization for the hash table pointer (necessary as it is `static`)
- the body for the the hash table creator and destroyer functions

How to use it

For every constructor type (e.g. MrConstructor, MrsConstructor, etc..) there should be a `defineRunTimeSelectionTable` entry in **Base.C**:

```
defineRunTimeSelectionTable(baseType,argNames)
```

The arguments are the same as above:

- `baseType` - this is **Base**.
- `argNames` - this is a small name you gave to the parameter list (i.e. MrConstructor and MrsConstructor above).

Example: **RASModel.C**

```
defineRunTimeSelectionTable(RASModel, dictionary);
```

addToRunTimeSelectionTable

This is a tiny snippet of code that adds a **Derived** constructor to the hash table held by **Base**. There are a variety of implementations of this macro depending on the templating.

What it does

Adds an `AddToTable` member variable to **Base**, which in turn adds **Derived**'s constructor to the hash table.

How to use it

For every constructor type, an `addToRunTimeSelectionTable` call should be made in **Derived.C**:

```
addToRunTimeSelectionTable(baseType,thisType,argNames)
```

The arguments are:

- `baseType` - this is **Base**.
- `thisType` - this is **Derived1**, **Derived2** or **Derived3**.
- `argNames` - this is a small name you gave to the parameter list (i.e. MrConstructor and MrsConstructor above).

Selectors

Selectors act as the virtual constructor function itself. They are declared in **Base**, and in OpenFOAM they are named `New`.

What they do

The selector looks through the function parameters to determine the `typeName` of the **Derived** to be constructed. It uses the `typeName` to look up the constructor in the hash table, and returns the constructor pointer it finds.

How to use them

There are no macros available for this. You need to fully implement the selector functions yourself. For every constructor type, you need to declare a `New` in **Base.H** and implement it in **Base.C**. Follow examples of other selectors already written.

Complications

In some situations, the runtime selection mechanism is further complicated with:

- additional templating; and
- class hierarchy.

Templating

Some of the macros may be hidden within other macros depending on the level of templating. For instance, if **Base** is itself a templated function `Base<Type>`, then the derived functions will have to:

- set their own `typeName` for all expected `Type`'s; and
- register all expected Type's to the constructor table.

This can be accomplished with a `typedef` macro in **Derived.H** and a macro to amass all the `addToRunTimeSelectionTable` entries in **Derived.C**.

Another common occurance involves the `typeName` macros. All the classes affected by runtime selection must have a `typeName`, accomplished with the `TypeName` macro in the header, and one of the `defineTypeNameAndDebug` macros in the body. To streamline the process, the `addToRunTimeSelectionTable` and `defineTypeNameAndDebug` macros can (and often are) called together in another macro.

Both of the situations above mean the `addToRunTimeSelectionTable` will not be directly visible in **Derived.C**.

Class hierarchy

Some situations arise where two runtime selection groups overlap in the class hierarchy. This can lead to a case where a **Derived** class of one group is also the **Base** of another group. In this case, a class will have both:

- macros belonging to a **Base**; and
- macros belonging to a **Derived**.

Just focus on the macros you expect to see. The other macros will not interfere with it.

O.8.4 How does it work?

In this section, a closer look is taken at what OpenFOAM's runtime selection is doing. A pared-down version of runtime selection is implemented in detail.

O.8.4.1 Virtual functions

Any function defined in a base class and redefined in its derived class is a virtual function. An object can use a base class pointer to manipulate a derived class object. This is accomplished behind the scenes through the use of pointers and hash tables. A hash table (called a `vtable`) is created for each class that has at least one virtual function, and each class has a pointer (called a `vpointer`) to its `vtable`. The entries in the vtable are pointers to all the virtual functions owned by that class. When a virtual function is called, the correct function is found by following the `vpointer` and looking it up on the `vtable`.

Unfortunately, a `vtable` cannot be constructed until the object in question is fully manifested. In other words, no virtual constructors.

0.8.4.2 Solution overview

The solution is to create your own vtable's and vpointer's:

1. Create a hash table in **Base** that contains constructor pointers indexed by `typeName`.
2. Have all the **Derived** add themselves to the list:
    a. Create a subclass `AddToTable` within **Base** whose constructor adds an entry to the hash table
    b. Add a line in **Derived.C** that creates an instance of `Base::AddToTable`
3. Create a "selector" function in **Base** called `New` that:
    a. is called like a regular **Base** constructor function;
    b. first looks through the parameters to determine the `typeName` of the **Derived** that should be constructed;
    c. uses the `typeName` to look up the correct **Derived** constructor pointer in the hash table; and
    d. returns the constructor pointer.
4. Wrap all these changes into macros so the programmer doesn't see them, and only has to worry about calling `New` instead of new.

0.8.4.3 Detailed implementation

This section shows the changes to the code that need to be made.

General changes

To implement a runtime selection mechanism:

- Decide on one or more sets of parameter lists for the virtual constructors.
- If there is more than one, give them each a short, informal name. For example:
- **Base** or **Derived**(`dictionary&`) is MrConstructor, and
- **Base** or **Derived**(`label&, scalar&, Istream&`) is MrsConstructor.
- Data within the parameters must reveal the `typeName` of the intended **Derived** type.
- Implement these as regular constructors in **Base** and **Derived**.

Adding a `vtable` and `vpointer` to Base

Most of the changes happen to **Base**. In **Base** you need to:

- create a data type that is a pointer to each constructor type:

  In **Base.H**:

```
typedef Base (*MrConstructorPtr)( const dictionary& dict)

typedef Base (*MrsConstructorPtr)
        (
            const label& place,
            const scalar& magnitude,
            const Istream& dataFlow
        );
```

- create a static pointer to a hash table of these function pointers:

  In **Base.H**:

```
// Showing only MrConstructor for now on
typedef HashTable<MrConstructorPtr, word, string::hash>
    MrConstructorTable;

static MrConstructorTable* MrConstructorTablePtr_;
```

  In **Base.C**:

```
//This line is needed because it is static
Base::MrConstructorTable* Base::MrConstructorTablePtr_ = NULL;
```

- Lastly, add `static` functions that create and delete the hash table:

  In **Base.H**:

```
static void constructMrConstructorTables();
static void destroyMrConstructorTables();
```

  In **Base.C**:

```
void Base::constructMrConstructorTables()
{
    static bool constructed = false;
    // this is necessary because it will be called many times
    if (!constructed)
    {
        Base::MrConstructorTablePtr_ = new Base::MrConstructorTable;

        constructed = true;
    }
}

void Base::destroyMrConstructorTables()
{
    if (Base::MrConstructorTablePtr_)
    {
        delete Base::MrConstructorTablePtr_;
        Base::MrConstructorTablePtr_ = NULL;
    }
}
```

Getting derived classes to register

To force **Derived** to register to the constructor table, OpenFOAM uses a clever trick:

- First **Base** is given an extra subclass whose constructor inserts new entries into the hash table, templated on the **Derived** type:

  In **Base.H**:

```
template<class DerivedType>
class addMrConstructorToTable
{
public:

    static Base New ( const dictionary& dict)
    {
        return Base(new DerivedType (dict));
    }

    addMrConstructorToTable
    (
        const word& lookup = DerivedType::typeName
    )
    {
        constructMrConstructorTables();
        MrConstructorTablePtr_->insert(lookup, New);
    }

    ~addMrConstructorToTable()
    {
        destroyMrConstructorTables();
    }
};
```

- Lastly, all **Derived** classes have one extra line in their **Derived.C**:

```
Base::addMrConstructorToTable<DerivedType>
    addDerivedTypeMrConstructorToBaseTable_;
```

Although this is in **Derived.C**, it is adding an instance of `Base::AddToTable` to the **Base** class. When **Base** is fully read, there will be a complete `vtable` with all the **Derived** constructors.

Selector – the virtual constructor function

The last step is to create a function in **Base** that takes constructor calls, looks up the correct constructor in the table, and returns a pointer to it. They are declared in **Base**, and in OpenFOAM they are named `New`. A simple selector might look like this:

In **Base.H**:

```
//- Return a pointer to a new Derived created on freestore
//  from dictionary
static tmp<Base> New(const dictionary&);
```

In **Base.C** (or **BaseNew.C**):

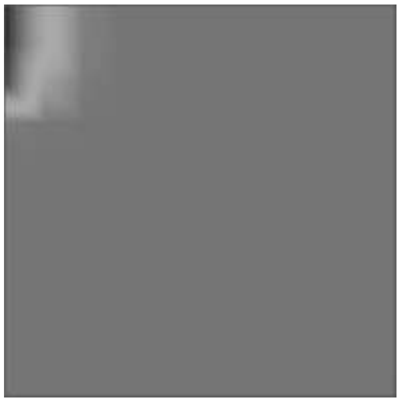
```
Foam::tmp<Foam::Base> Foam::Base::New(const dictionary& dict)
{
    // omitting error catching and debug statements

    word DerivedType(dict.lookup("type"));

    typename MrConstructorTable::iterator cstrIter
        = MrConstructorTablePtr_->find(DerivedType);

    return cstrIter()(dict);
```

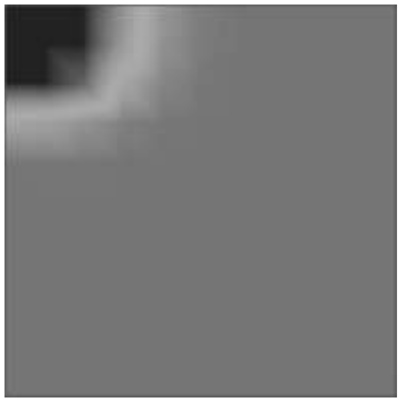
When does this all happen?

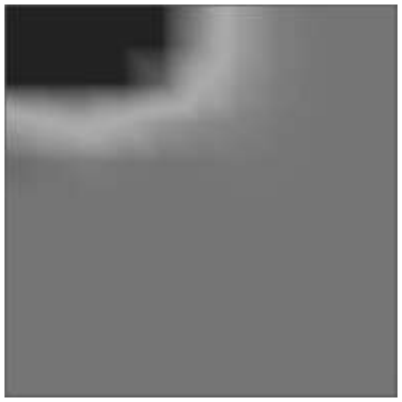
The "virtual constructor" table is fully built and loaded into memory when the include list is fully read, and before the first line of code is executed. This is achieved using the `static` keyword. All `static` members of a class exist prior to any instance of that class.

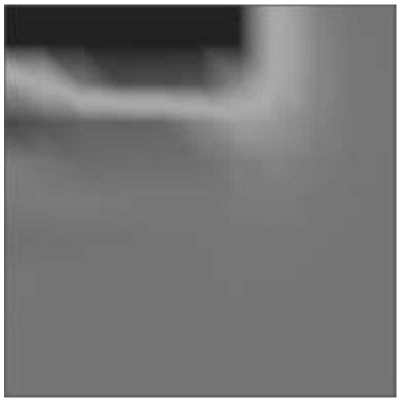
Appendix P Animation screenshots

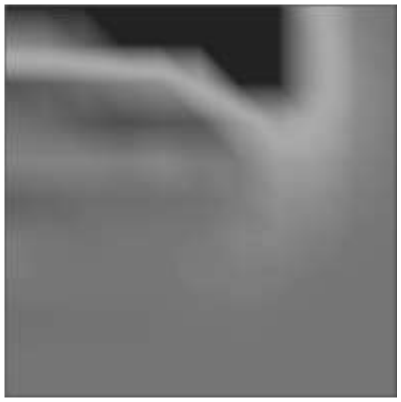
For the benefit of readers who do not have access to the animation discussed in Section 5.2.5, this section presents a series of timelapse screenshots from the first fluid injection and mixing event from the animation. The time values are simulation time.

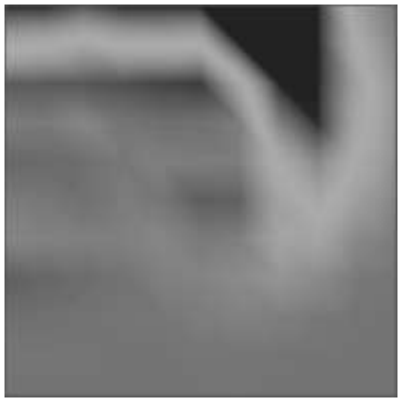

Frame 1, *t* = 11 [s]

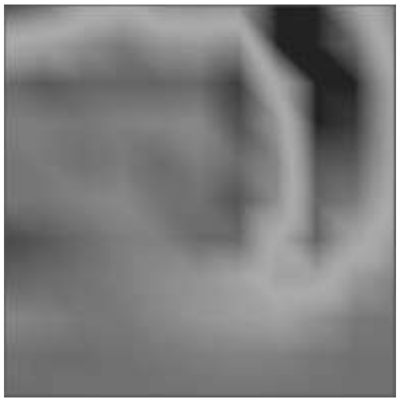

Frame 6, *t* = 64 [s]

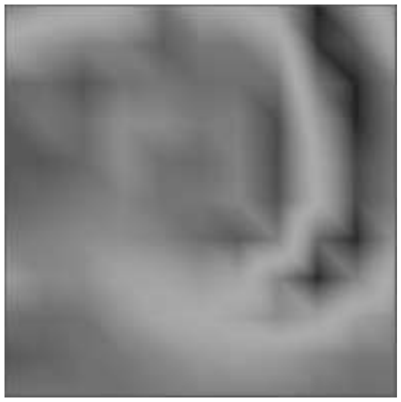

Frame 11, *t* = 120 [s]

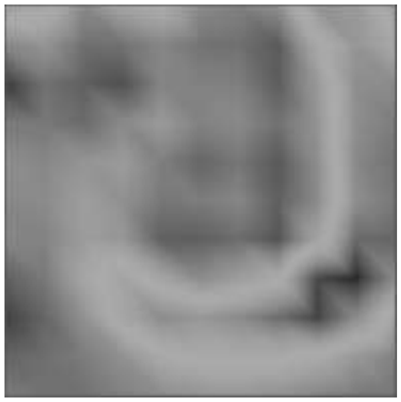

Frame 16, *t* = 177 [s]

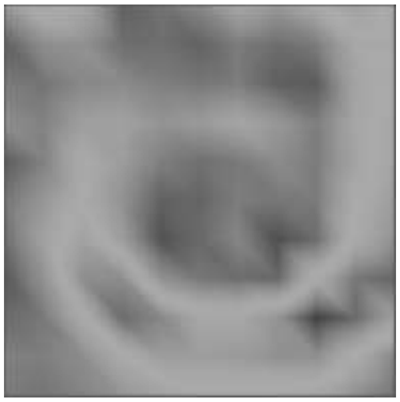

Frame 21, *t* = 237 [s]

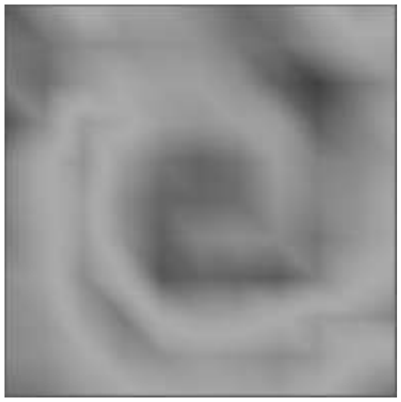

Frame 26, *t* = 294 [s]

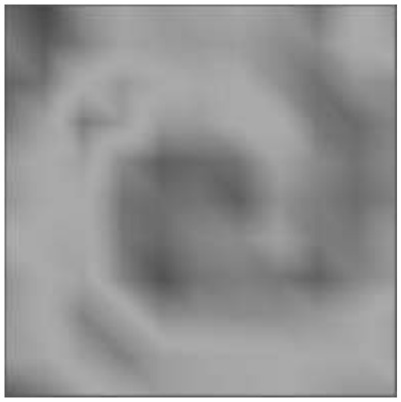

Frame 31, *t* = 352 [s]

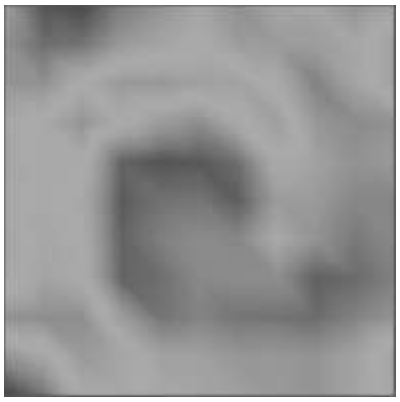

Frame 36, *t* = 410 [s]

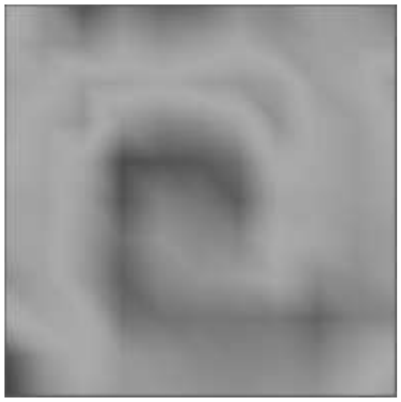

Frame 41, *t* = 466 [s]

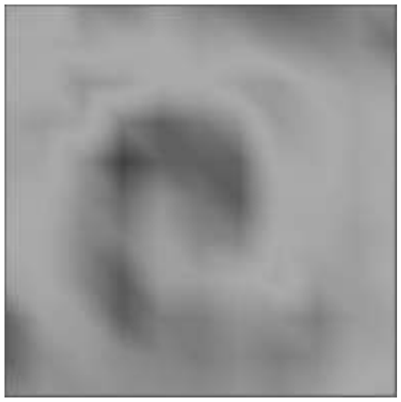

Frame 46, *t* = 522 [s]


Frame 51, *t* = 574 [s]

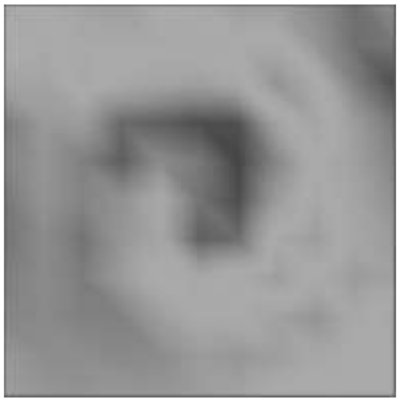

Frame 56, *t* = 623 [s]

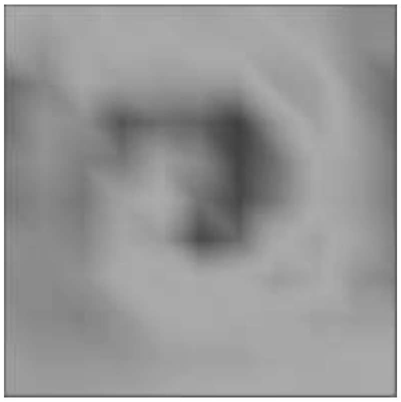

Frame 61, *t* = 682 [s]


Frame 66, *t* = 742 [s]


Frame 71, *t* = 802 [s]

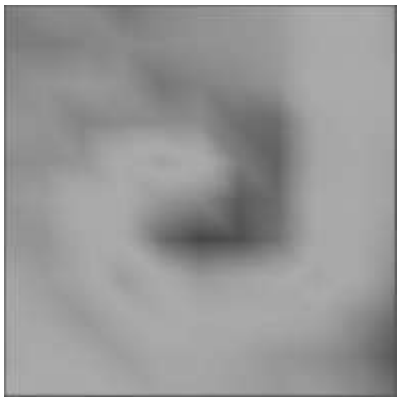
Frame 76, t = 862 [s]

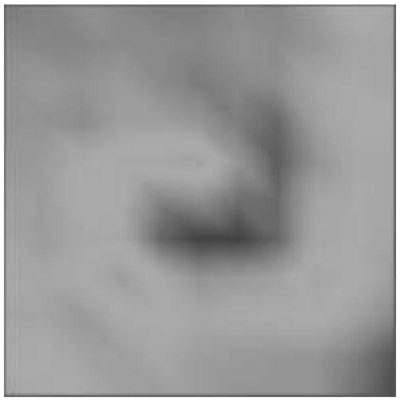

Frame 81, t = 922 [s]


Frame 86, t = 982 [s]

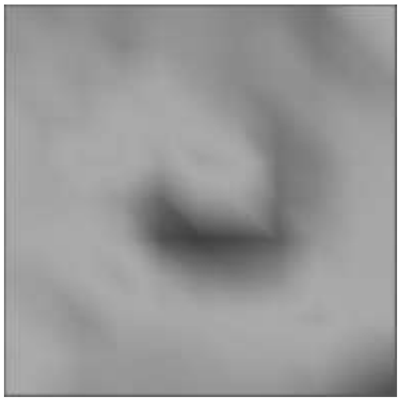
Frame 91, t = 1042 [s]

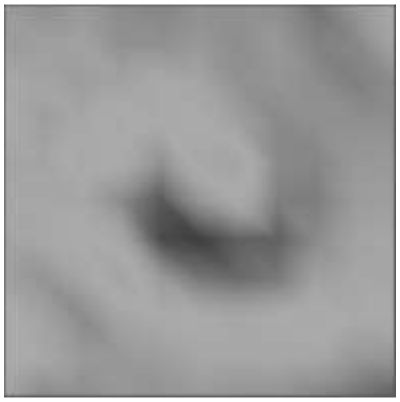
Frame 96, t = 1102 [s]


Frame 101, t = 1150 [s]

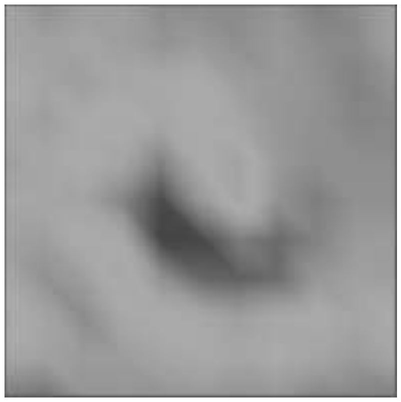

Frame 106, t = 1210 [s]

Frame 111, t = 1270 [s]


Frame 116, t = 1330 [s]



Frame 121, *t* = 1390 [s]



Frame 126, *t* = 1450 [s]

Frame 131, *t* = 1510 [s]

Frame 136, *t* = 1570 [s]

Frame 141, *t* = 1630 [s]

Frame 146, *t* = 1690 [s]

Figure A35: Animation screenshot series