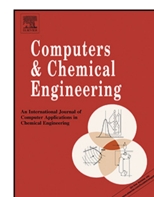
Computers and Chemical Engineering 62 (2014) 139–160

Contents lists available at ScienceDirect

Computers and Chemical Engineering

journal homepage: www.elsevier.com/locate/compchemeng



From discretization to regularization of composite discontinuous functions

Tareg M. Alsoudani*, I.D.L. Bogle

Centre for Process Systems Engineering, University College London, Department of Chemical Engineering, Torrington Place, London, WC1E 7JE, U.K.

ARTICLE INFO

Article history:
Received 1 October 2012
Received in revised form
24 November 2013
Accepted 30 November 2013
Available online 18 December 2013

Keywords:
Discontinuity detection
Discontinuity resolution
Solving differential equations.

ABSTRACT

Discontinuities between distinct regions, described by different equation sets, cause difficulties for PDE/ODE solvers. We present a new algorithm that eliminates integrator discontinuities through regularizing discontinuities. First, the algorithm determines the optimum switch point between two functions spanning adjacent or overlapping domains. The optimum switch point is determined by searching for a "jump point" that minimizes a discontinuity between adjacent/overlapping functions. Then, discontinuity is resolved using an interpolating polynomial that joins the two discontinuous functions.

This approach eliminates the need for conventional integrators to either discretize and then link discontinuities through generating interpolating polynomials based on state variables or to reinitialize state variables when discontinuities are detected in an ODE/DAE system. In contrast to conventional approaches that handle discontinuities at the state variable level only, the new approach tackles discontinuity at both state variable and the constitutive equations level. Thus, this approach eliminates errors associated with interpolating polynomials generated at a state variable level for discontinuities occurring in the constitutive equations.

Computer memory space requirements for this approach exponentially increase with the dimension of the discontinuous function hence there will be limitations for functions with relatively high dimensions. Memory availability continues to increase with price decreasing so this is not expected to be a major limitation.

© 2013 Elsevier Ltd. All rights reserved.

1. Introduction

A process can be thought of as a complex system that is described by, mostly, continuous mathematical functions (algebraic or differential). Solution of these differential equations, usually through integration, brings an insight into the behaviour of the process under study. However, the continuity of these mathematical functions is sometimes broken by internal or external influences. Breakage of continuity occurs because of the tendency of scientists to treat each process condition with differing constitutive equations and/or boundary conditions. Once simulation shifts from one condition to another, the underlying equations change, usually with no reservation to mathematical continuity. A rapid phase shift or flow reversal represents an example of an internally generated discontinuity in ODE/DEA system whereas switching a pump on/off can be considered as an external influence that raises a mathematical discontinuity in the modelled system.

Handling discontinuity through ODE/DAE solvers is performed through two steps: discontinuity detection and discontinuity resolution; although some solvers combine the two steps (Mao & Petzold, 2002).

The literature refers to the problem of locating a discontinuity as discontinuity detection (Javey, 1988). Process simulators usually couple their integrators with the modelling language. This coupling eases detection of jump discontinuities.

Regardless of the form or source of discontinuity, it needs to be resolved either before starting to integrate the ODE/DAE system (if possible) or whenever it is encountered during the evolution of integration process. Methods for the resolution of discontinuities arising during integration of differential equations can be divided into two types:

1. Type I tries to handle discontinuities using methods that are usually integrated with the solver (integrator) of the ODE/DAE system. Those methods are usually generic, irrespective of the system to be modelled and handle discontinuities at the time they are encountered during integration (or simulation). Most literature on discontinuity detection and resolution covers this

* Corresponding author. Tel.: +966 50 634 7994; fax: +44 0 20 7383 2348.
E-mail addresses: t.alsoudani@live.ucl.ac.uk (T.M. Alsoudani), d.bogle@ucl.ac.uk (I.D.L. Bogle).

0098-1354/$ – see front matter © 2013 Elsevier Ltd. All rights reserved.
http://dx.doi.org/10.1016/j.compchemeng.2013.11.019

Nomenclature

a_p	specific area of the pellet
C_T	dimensionless total concentration
C_t^{ref}	total molar concentration
D_z	axial thermal conductivity
k_{gl}	overall mass transfer coefficient
L	column length
N_g^m	number of fluid film mass-transfer units = $\frac{1-\varepsilon}{\varepsilon}\frac{a_p k_{gl} L}{u_{ref}}$
Pe_m	mass Peclet number = $\frac{u_{ref} L}{D_z}$
ρ_s	solid density
q_i^{ref}	maximum adsorbence of adsorbate i in adsorbent pellet
Q_i	dimensionless adsorbence of adsorbate i in adsorbent pellet
u_{ref}	reference velocity
U	dimensionless velocity
$\vec{v}_i$	vector dimension at time instant i of simulation run
x	dimensionless axial distance or x-dimension
y	dimensionless concentration (mole fraction) or y-dimension
$\langle y \rangle$	adsorbate dimensionless concentration (mole fraction) in solid phase

Greek Letters

ε	void fraction
ζ_{m_i}	mass capacity factor = $\frac{1-\varepsilon}{\varepsilon}\frac{\rho_s q_i^{ref}}{C_t^{ref}}$
τ	dimensionless time

Sub/superscripts:

f	feed
p	purge
i	component index or simulation time instant
m	mass
s	solid

class (eg. Ellison, 1981; Javey, 1988; Mao & Petzold, 2002; Park & Barton, 1996).

2. Type II handles discontinuities using knowledge about the process to be modelled. It remodels the ODE/DAE system in a way that eliminates discontinuities. Literature is very sparse in this area (e.g. Borst, 2008; Brackbill, Kothe, & Zemach, 1992; Carver, 1978; Helenbrook, Martnelli, & Law, 1999).

Borst (2008) refers to the two types as discretization and regularization, respectively (Fig. 1). He also points out that internal model discontinuities are better handled using type II methods irrespective of the solver integration routine. Surprisingly, both classes use some form of an interpolation to convert a discontinuous region into a continuous one when dealing with internally generated discontinuities. Externally generated discontinuities are usually handled by re-initialization of the model equations and their respective new initial and boundary conditions. In the foregoing discussion, we will briefly touch on recent literature covering each of the categories.

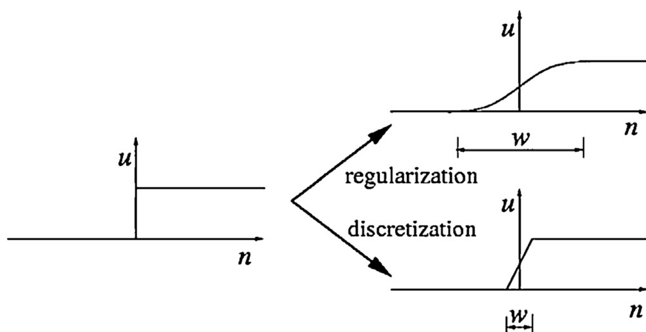


Fig. 1. Transformation of a discontinuity into a regularization or a discretization problem. (Borst, 2008).

1.1. Type I—Integrator based discontinuity resolution

Cellier (1979) demonstrated that the most efficient approach to locating a state event is through *discontinuity locking*. In discontinuity locking the system of ODE/DAE is locked until the end of the integration step regardless of the existence of a state event during the step. After completion of the integration step that involves a state event, the exact location of the state event is detected. Several event location algorithms that use discontinuity locking mechanism are reported and for a comprehensive review of state event detection algorithms the reader may refer to Park and Barton (Park & Barton, 1996). Mao and Petzold (2002) have introduced an event detection algorithm that is based on regulating the integration step size based on discontinuity functions that are appended to the DAE system. Recently, Archibald, Gelb, and Yoon (2008) introduced a state event detection algorithm that is based on polynomial annihilation techniques. Their method relies on the difference of the Taylor series expansions behaviour between continuous and non-continuous intervals of the tested function.

Once a discontinuity is detected, it needs to be resolved before the integrator passes it. Javey (1988) reports three methods for resolving discontinuities. In all methods, the integrator checks the sign change of a discontinuity-function after each integration step as indication of having located a discontinuity:

1. Once the discontinuity is located, the integrator switches modelling equations to those after the discontinuity and starts at the end of the current step. This procedure is inaccurate as it accumulates error each time a discontinuity is encountered. Mao and Petzold (2002) warn about mere stepping over discontinuities without carefully handling them with some rigour.
2. Once the discontinuity is located, the integrator halves the step and repeats the last integration step in a hope to resolve the discontinuity. Resolution is generally achieved if the function is continuous but the integrator may fail to resolve the discontinuity due to the use of a large integration step. Thus, repeating the integration step with smaller step sizes, where the discontinuity is detected should eventually reveal the continuity of the function. This solution, although better than the first one, is still considered inefficient because the integrator needs to iterate at the discontinuity until an acceptable error tolerance is achieved. If the acceptable error tolerance is not achieved after repeated step-halving (usually because of an instantaneous discontinuity), the integrator aborts integration. The method is then unable to resolve the discontinuity (Carver, 1978).
3. Once the discontinuity is located, the integrator reinitializes the differentiable variables using post discontinuity conditions after interpolating differential and algebraic variable at the discontinuity using a *discontinuity function* (an interpolating polynomial). It should be noted that this method implies mathematical continuity of differential equations through the discontinuity domain regardless of the validity of the resulting solution, as demonstrated by Cellier (1979). This method is the most commonly adopted in recent integration routines used for process simulation. The mismatch between the results obtained using the interpolating polynomial and those obtained when reinitializing the ODE/DAE system after crossing a

discontinuity sometimes creates what is known as a *sticky discontinuity*. Sticky discontinuities occur because sometimes after reinitializing the ODE/DAE system, the state of the differential variables returns to the value it had before triggering the discontinuity resolution resulting in an infinite loop: locating the discontinuity, interpolating to conditions after the discontinuity, reinitializing ODE/DAE after the discontinuity, re-evaluating discontinuity trigger and falling back to the same discontinuity, interpolating to conditions after discontinuity, etc.

Two problems arise from Type I discontinuity resolution:

- Re-initialization effort is directly proportional to the number of DAE/ODE equations. Even if discontinuity is encountered in one equation of the system, the integrator still needs to reinitialize the entire system. This procedure is computationally expensive. What is needed is an approach that detects and eliminates localized discontinuities leaving the rest of the system's continuous functions intact.
- Some integration routines use interpolating polynomials to bridge discontinuous domains. The use of integrator-based interpolating polynomials can produce inaccurate results at and/or after the discontinuous region. Park and Barton (1996) demonstrate that sticky discontinuities arise because the interpolating polynomial used by the integrator to overcome a ODE/DAE discontinuity may land the ODE system at a point before the discontinuity. This is mainly due to the difference in behaviours between the ODE/DAE system and the interpolating polynomial that is used to approximate its behaviour at the discontinuity although the ODE/DAE system and the interpolating polynomial share the same initial conditions at the location immediately preceding the discontinuity. We may easily deduce that even if the interpolating polynomial has managed to cross the discontinuity, it will probably land at a location post the discontinuity that is different from that corresponding to the course of the ODE/DAE system. So, even when discontinuities are resolved using integrator-based interpolating polynomials, the solution post discontinuity loses accuracy. The error accumulates with every discontinuity that is resolved.

1.2. Type II—System impeded discontinuity resolution

In this section, we shed light on resolution of discontinuities using bridging functions that are derived from laws surrounding the physical system or their approximation. The first published attempt was by Carver (1978). He appended the discontinuous functions to the ODE system after a slight transformation. Then, he applied Gears algorithm (Gear, 1970) to detect discontinuities. Carver's attempt was the only encountered attempt to generalize a solution using Type II although the problem was still left discretized (i.e. no regularization functions used).

Brackbill et al. (1992) resolved a discontinuity resulting from the contact of two fluids at an interface point by a smooth interpolation between discontinuities using the following function:

$$P(x) = \begin{cases} C_1 & (FLUID1) \\ C_2 & (FLUID2) \\ 0.5 * (C_1 + C_2) & (INTERFACE) \end{cases}$$

Helenbrook et al. (1999) criticized Brackbill's approach as introducing an error that is linearly proportional to the formed grid. Instead they recommended replacing discontinuities with moving boundaries that retain the interface region between the two fluids.

Borst (2008) emphasized that the use of regulating functions derived from the physics of the problem (Type II) will better eliminate discontinuities than the sole use of Type I discretization techniques. He attributes the enhancement to the increase in length (or time) scale over that resulting from the use of discretization techniques; as illustrated in Fig. 1. He illustrated the concept by modelling fractures of solid material at their break points.

In this paper, we provide a generic approach to Type II problems that is problem independent. Once included within a simulation package, this approach will eliminate the need for the simulator integrator routine to reinitialize state variables whenever a discontinuity is located. In addition, since the approach tackles discontinuities at their appropriate level, interpolating polynomials resulting from this approach more resemble the accurate simulation path than those generated by integration routine that resolve discontinuities at state variable level only. The resolution is generic enough to be adopted in:

1. implicitly defined discontinuities arising from discontinuous constitutive equations but usually appear in state variables.
2. explicitly defined discontinuities that are formulated as boundary conditions.

Although examples demonstrating the concept are drawn from the field of chemical engineering, the concept applies to any mathematically developed model that involves the use of logical expressions in any field of science. Other examples, at which this work might prove beneficial, include start-up and shut-down of process units such as those reported by (Gani, Ruiz, & Cameron, 1986) and (Ruiz, Cameron, & Gani, 1988). The next section describes the one-dimensional detection and resolution methods. This is then generalized to two-dimensional problems which requires explanation of overlapping regions, and then to *n*-dimensions (shown in 3D using mesh-grid), followed by two examples: one involving a fluid flow problem with two variables and the other involving the regularization of initial and boundary conditions of a Pressure-Swing-Adsorption (PSA) column.

2. One-dimensional discontinuity detection and resolution

Let us assume that we have a composite function f that is defined by two separate sub-functions $f_1(x)$ and $f_2(x)$ that span two adjacent domains $[a', b]$ and $[a, b']$, respectively:

$$f(x) = \begin{cases} f_1(x), & x \in [a', b] \\ f_2(x), & x \in [a, b'] \end{cases} \tag{1}$$

For demonstration purposes, we will assume that $a > a'$. The ideal situation for the modeller is to have a continuous composite function across the entire simulation domain regardless of the sub-domains defining the respective sub-functions. To achieve this situation, the switch between f_1 and f_2 has to occur at a changeover (switch) location g satisfying the following condition (Fig. 2a):

$$f_1(g) = f_2(g) \tag{2}$$

However, switch point g is seldom searched for, or even considered, when modelling. Instead the modeller usually opts for the selection of a point g' based usually on a widely adopted convention. Such an arbitrary selection often raises a discontinuity between sub-domains at any arbitrary switch point g' as illustrated in Fig. 2a. In such a case, the objective is to eliminate a discontinuity between two intersecting functions spanning overlapping domains. In this case, functions intersect and function domains overlap. Thus, there exists a point g that satisfies (2).

When (2) is not satisfied, functions are said to be non-intersecting as illustrated in Fig. 2b and c. For non-intersecting functions, there is usually a location g, along the dimension of the independent variable that minimizes the distance between the two

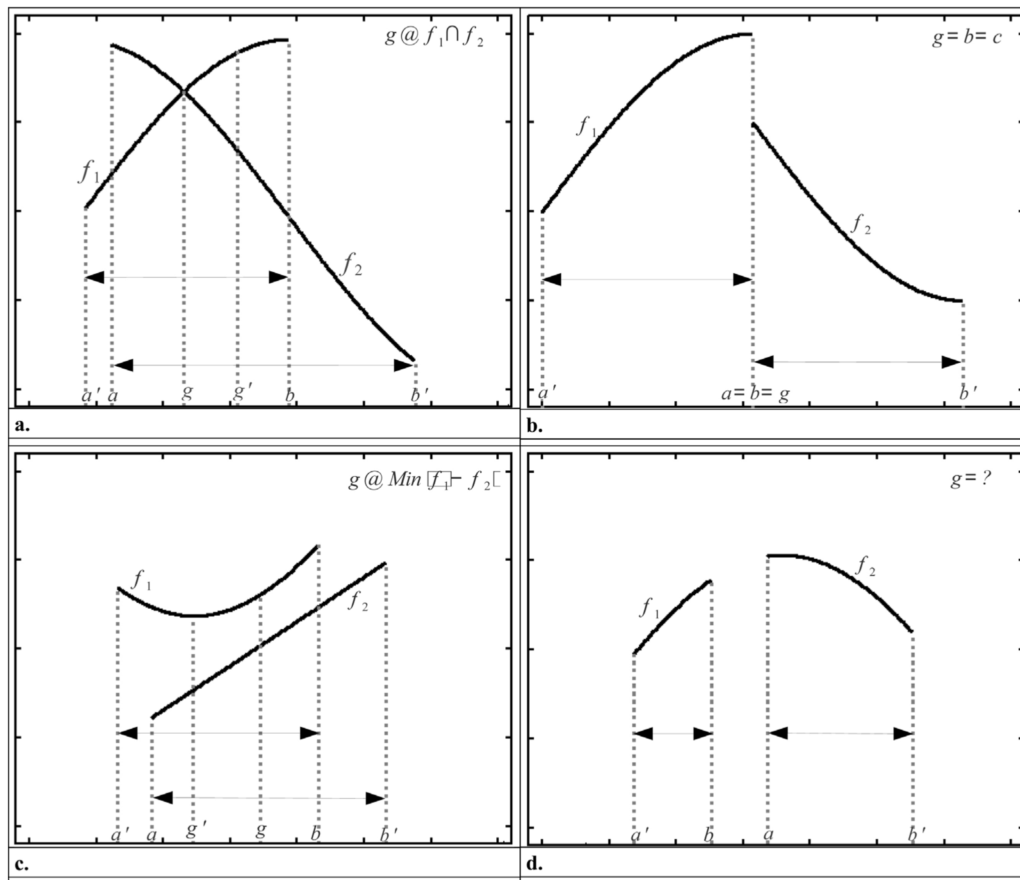


Fig. 2. Forms of domain switch points between two functions and types of discontinuities between two adjacent domains.

functions and hence allows for a smoother jump. Jumping between the two functions at any point other than g would result in an extra effort by the integration routine to resolve the discontinuity. Thus, in such cases, the objective of this work is to minimize jump effort between two overlapping but non-intersecting functions spanning overlapping domains. In this case respective functions' domains overlap ($a<b$). However, unlike case I, functions do not intersect.

The first objective of this work is to find the best switch point g for any given set of two overlapping functions, whether intersecting or non-intersecting. The second objective is to eliminate discontinuities in non-intersecting functions by devising an interpolating polynomial at the location of the discontinuity between the two functions. To achieve both objectives, the method is decomposed into discontinuity detection and discontinuity resolution sub-problems.

2.1. One-dimensional discontinuity detection

First, we must sort the ranges for the respective functions using their starting points in an ascending or descending order and then compare the location of the domain end of one function, (e.g. b for f_1), with the domain start of its successor, (e.g. a for f_2). If the end and start domain limits of two respective successive functions are equal (i.e. $a=b$), the discontinuity is said to be non-overlapping. Point g is immediately identified for non-overlapping domains as $g=a=b$ *as* illustrated in Fig. 2b. Sorting and comparison will also immediately detect if sub-functions f_1 and f_2 do not satisfy the continuity assumed for the main function f spanning $[a', b']$ as illustrated in Fig. 2d.

Having identified an overlap domain, to find g for overlapping discontinuous functions, we will transform the problem into an optimization problem. As an example, the overlap domain for Fig. 2a and Fig. 2c is $[a,b]$. We define a difference function:

$$e(x) = \left| f_1(x) - f_2(x) \right| \tag{3}$$

Our objective is to find a point g that minimizes $e(x)$ over the domain $[a,b]$. Since this is a fairly simple optimization problem, it can be solved using any of the commercially available optimisation routines. A side advantage of this approach is that it would accurately detect g for cases where functions intersect and overlap (Fig. 2a) as the point where $e(g)=0$.

Once g is detected, it can be immediately inserted into the logical expression of the composite function; replacing any arbitrary selected g' by the modeller. For example, if the detection algorithm resulted in locating a minimum jump effort point g between two discontinuous functions f_1 and f_2, g can easily be inserted into the final conditional statement as illustrated in (4):

$$\begin{array}{lll} \textit{If}\ (x<g)\ \textit{then} & (\textit{Domain I}) & \\ \quad f=f_1(x) & & \\ \textit{Else if}\ (x>=g)\ \textit{then} & (\textit{Domain II}) & \\ \quad f=f_2(x) & & \end{array} \quad \textit{or} \quad \begin{array}{ll} \textit{If}\ (x<=g)\ \textit{then} & (\textit{Domain I}) \\ \quad f=f_1(x) & \\ \textit{Else if}\ (x>g)\ \textit{then} & (\textit{Domain II}) \\ \quad f=f_2(x) & \end{array} \tag{4}$$

For cases where functions intersect and overlap (Fig. 2a), a discontinuity detection algorithm is sufficient to grant at least smooth continuity between the discontinuous functions but not their respective first and second derivatives. For cases where functions touch or overlap but do not intersect (Fig. 2b and c, respectively), discontinuity detection algorithm might be sufficient if the simulation integrator routine is able to jump between the functions without the need for reinitializing the state variables. As indicated by Borst (2008), resolution of discontinuity using Type I discontinuity handlers might not always be appropriate because of the

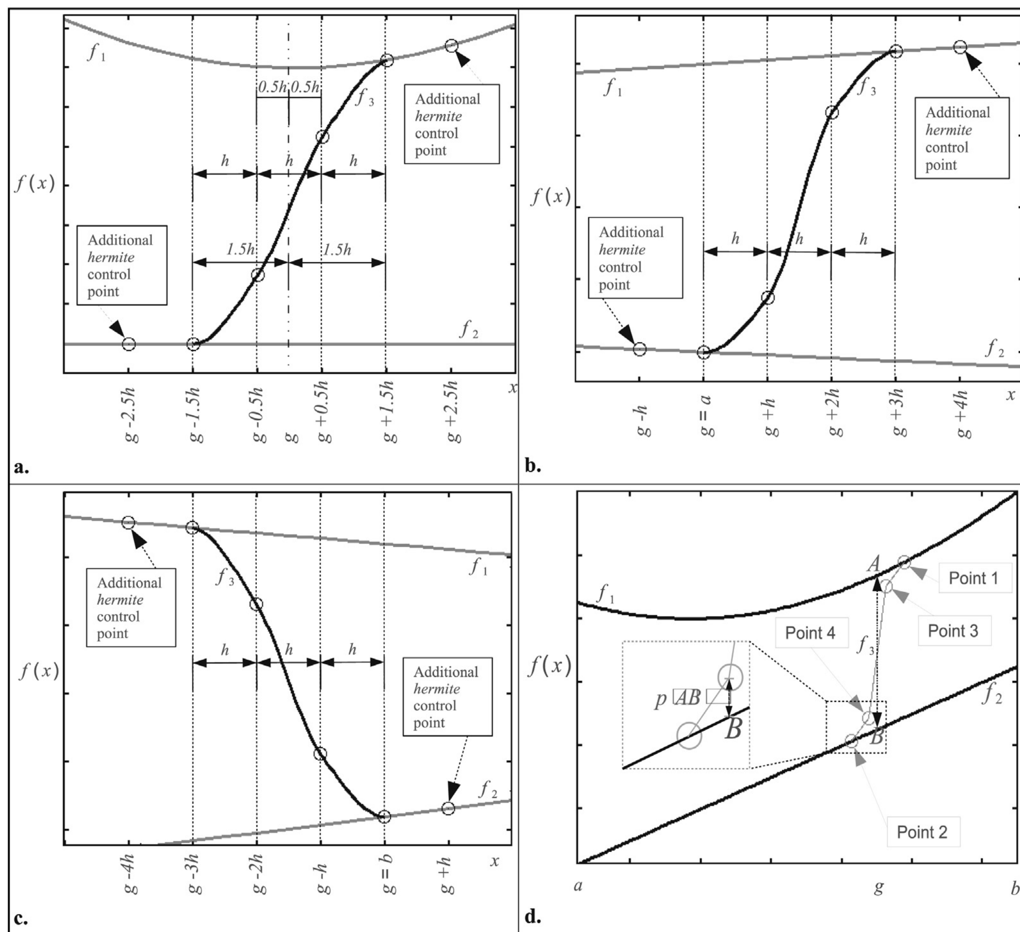


Fig. 3. Location of mesh control points relative to the minimum jump-effort point g.

exhaustive need to reinitialize state variables and the fact that, in some cases, re-initialization might alter the solution path. Thus, we propose a discontinuity resolution algorithm to avoid falling into state-variable re-initialization.

2.2. One-dimensional discontinuity resolution

Discontinuity resolution takes the form of bridging the two discontinuous domains through an interpolating polynomial, f_3. Linear interpolation requires at least two points. However, we will attempt to link functions using a smooth interpolating polynomial preferably to the third degree. Linking functions with a third degree interpolating polynomial ensures continuity up to the second derivative of the interpolating function. To construct any smooth polynomial, we need at least three points. One would think that three points are sufficient to construct the polynomial around the discontinuity point. However, as we will demonstrate later, at least four points are required in order to minimize first and second derivatives' discontinuities at the junction points between the interpolating polynomial f_3 and the corresponding discontinuous functions f_1 and f_2.

To simplify computations, we will evenly separate the points by an interval h from each other. Their exact locations will be relative to the location of the discontinuity location (g) in the independent variable dimension. The location of the mesh control points, relative to g, takes one of three forms depending on whether the function has a minimum in the overlap domain ($[a,b]$):

- If a minimum $g \in (a, b)$ exists, mesh control points will be located at distances $g-1.5h$, $g-0.5h$, $g+0.5h$ and $g+1.5h$ as illustrated in Fig. 3a. This selection of points' locations ensures even distribution of the interpolating points on both sides of the point g.
- If the minimum $g \notin (a, b)$, then g must reside at one end of the domain. If g is located at the start of the overlap domain ($g=c$), mesh control points will be located at g, $g+h$, $g+2h$ and $g+3h$ as illustrated in Fig. 3b.
- If g is located at the end of the overlap domain ($g=b$), mesh control points will be located at g, $g-h$, $g-2h$ and $g-3h$ as illustrated in Fig. 3c.

To perform a smooth transition, we need at least one point to lie on each of the functions' curves at the respective sides of the discontinuity location. Let us call these points *point 1* and *point 2*. Taking Fig. 3a as an example for the case where $g \in (a, b)$, the respective locations of points 1 and 2 will be $(g-1.5h, f_2(g-1.5h))$ and $(g+1.5h, f_1(g+1.5h))$, respectively. Of course, one can argue that we could also position the points at $(g-1.5h, f_1(g-1.5h))$ and $(g+1.5h, f_2(g+1.5h))$. However, we should bear in mind that the sorting algorithm, explained earlier, decides on the order of the functions based on their span over the independent variable dimension.

For the case where g is located at the start of the overlap domain ($g=a$), the respective locations of points 1 and 2 will be $(g, f_2(g))$ and $(g+3h, f_1(g+3h))$. For the case where g is located at the end of the overlap domain ($g=b$), points 1 and 2 will be located at $(g-3h, f_1(g-3h))$ and $(g, f_2(g))$, respectively. Respective examples of both cases are illustrated in Fig. 3b and c. Of course, the detection algorithm argument still holds.

For the last two points (*points* 3 and 4), of the four point set, we utilized the length of the line segment $|AB|$, defined by Eq. (5) and

illustrated in Fig. 3d, to shift f_3 function values at these points from the respective discontinuous functions values. Since the location of g, on the independent variable dimension, corresponds to the point that exhibits minimum distance between the two functions f_1 and f_2 within the overlap domain, the length of the line segment $|AB|$ corresponds to that minimum distance.

As an example, let us take the case where $g \in (a, b)$. The y-axis values of the points located at distances $-0.5h$ and $+0.5h$ from the point g will be calculated as the values of the functions at these respective points after adding or subtracting a fraction p of $|AB|$. For lower valued functions (e.g. f_2), Point 3 would have the coordinates $(g-0.5h, f_2(g-0.5h)+p|AB|)$. For higher valued functions (e.g. f_1), Point 4 would have the coordinates $(g+0.5h, f_1(g+0.5h)-p|AB|)$.

$$|AB| = \left|f_1(g) - f_2(g)\right| \tag{5}$$

Fritch and Carlson (1980) detail the necessary and sufficient conditions to ensure monotonicity of the interpolating polynomial control points. Basically, they prove that in order to ensure a monotonically increasing or decreasing function, slopes of control points should have the same sign or a value of zero. To emphasise the same concept, the value of p should satisfy the condition in (6):

$$0 \le \frac{p}{|AB|} \le 0.5 \tag{6}$$

Naturally, providing a separate p value for each of the functions f_1 and f_2 would add to the degrees of freedom as long as they satisfy the condition in (6). These two p values can act as tuning parameters to smooth the transition between f_3 and the discontinuous functions f_1 and f_2. In addition, the original formulation of *hermite* interpolating polynomials (to be discussed later) uses a tension parameter (t) that extends between 0 and 1. We could use either t or p to perfect the resulting interpolation curve. However, we intend to keep both parameters in order to smooth the transition between the interpolating polynomial and the discontinuous functions.

To demonstrate the effect of the interpolation algorithm on the logical expression, let us consider the example in (4) and assume $g \in (a, b)$. After generating the four-point interpolating polynomial, the logical statements above will be transformed into (7).

We should also note that, because of the uniqueness of the solution for one-dimensional functions, the devised procedure can be run off-line prior to the start of the simulation run. Indeed, we recommend embedding the algorithm into the modelling language compiler to automate generation of polynomials and their respective additional conditional expressions.

$$\begin{aligned}
&\textit{If}\ (x < g - 1.5h)\ \textit{then} && (\textit{Domain I})\\
&\quad f = f_1(x) &&\\
&\textit{Else if}\ (|x - g| \le 1.5h)\ \textit{then} && (\textit{Interpolating polynomial Domain})\\
&\quad f = f_3(x) &&\\
&\textit{Else if}\ (x > g + 1.5h)\ \textit{then} && (\textit{Domain II})\\
&\quad f = f_2(x) &&\\
&\textit{End if} &&
\end{aligned} \tag{7}$$

The algorithm can easily be extended to account for complex logical expressions such as (8) by solving for x.

$$\begin{aligned}
&\textit{If}\ (w(x) = 0)\ \textit{then} && (\textit{Domain I})\\
&\quad f = f_1(x) &&\\
&\textit{Else if}\ (w(x) > 0)\ \textit{then} && (\textit{Domain II})\\
&\quad f = f_2(x) &&\\
&\textit{End if} &&
\end{aligned} \tag{8}$$

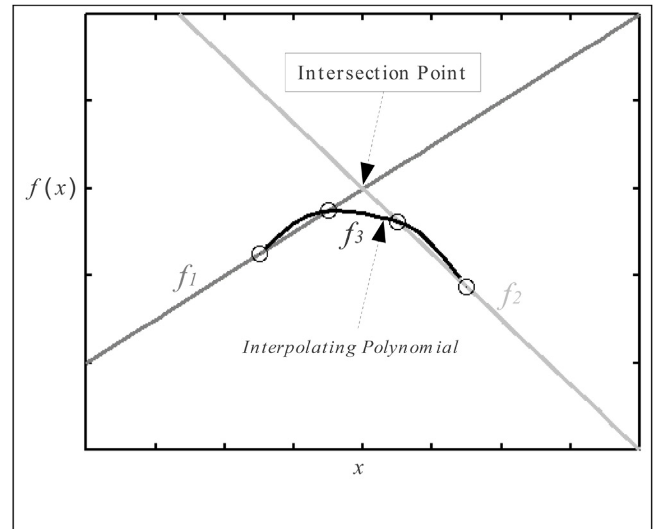


Fig. 4. A four-point *hermite* interpolating polynomial between two intersecting uni-dimensional functions using *tension* (t) = 0.

Lastly, an additional side benefit resulting from the use of the line segment $|AB|$ to locate the intermediate points at $g-0.5h$ and $g+0.5h$ is that the locations of these points automatically coincide with the locations of the respective functions f_1 and f_2 if f_1 and f_2 posses a common intersection point since $|AB| = 0$ in this case regardless of the value of p. This benefit indicates that detection and resolution algorithms can be integrated seamlessly without the need to treat intersecting functions separately. Fig. 4 illustrates the resulting interpolating polynomial of two intersecting functions.

2.3. Perfecting the connection and the bounding box problem

The smoothing of the transition between the interpolating polynomial and the discontinuous functions can be transformed into an optimization problem that minimizes first or second derivative differences between the interpolating polynomial and the discontinuous functions at Point 1 and Point 2. The optimization problem can be formulated as

$$\begin{array}{ll}
\min: \left|f'^{-}_{P1} - f'^{+}_{P1}\right| + \left|f'^{-}_{P2} - f'^{+}_{P2}\right| & \min: \left|f''^{-}_{P1} - f''^{+}_{P1}\right| + \left|f''^{-}_{P2} - f''^{+}_{P2}\right| \\
s.t. = \begin{cases} 0 \le p_i < 0.5 \\ 0 \le t \le 1 \end{cases} & s.t. = \begin{cases} 0 \le p_i < 0.5 \\ 0 \le t \le 1 \end{cases} \\
\text{a. first order derivative optimization} & \text{b. second order derivative optimization}
\end{array} \tag{9}$$

If the derivatives of the discontinuous functions, appearing in the cost function, are readily available, they can be directly evaluated through the available expressions. Otherwise, any derivative estimation numerical technique (e.g. secant method) can be used to evaluate the required derivatives.

Once the position of the points is determined, we need to connect them with a continuous interpolating function that is preferably second order smooth to aid in calculation of Jacobian and Hessian matrices when required by the numerical ODE/DAE solver. Two interpolation methods satisfy our criteria: cubic splines and cubic *hermite* interpolating polynomials. However, we selected *hermite* interpolating polynomials for the following reasons:

1. For the same set of interpolating points, cubic spline interpolating polynomials exhibit more overshoot than their cubic *hermite* counterparts (Fritsch & Carlson, 1980).
2. Cubic *hermite* interpolating polynomials have one more degree of freedom to better control the shape of the interpolating polynomial (Bartels, Beatty, & Barsky, 1987; Kochanek & Bartels, 1984). This degree of freedom is granted by the extra tension

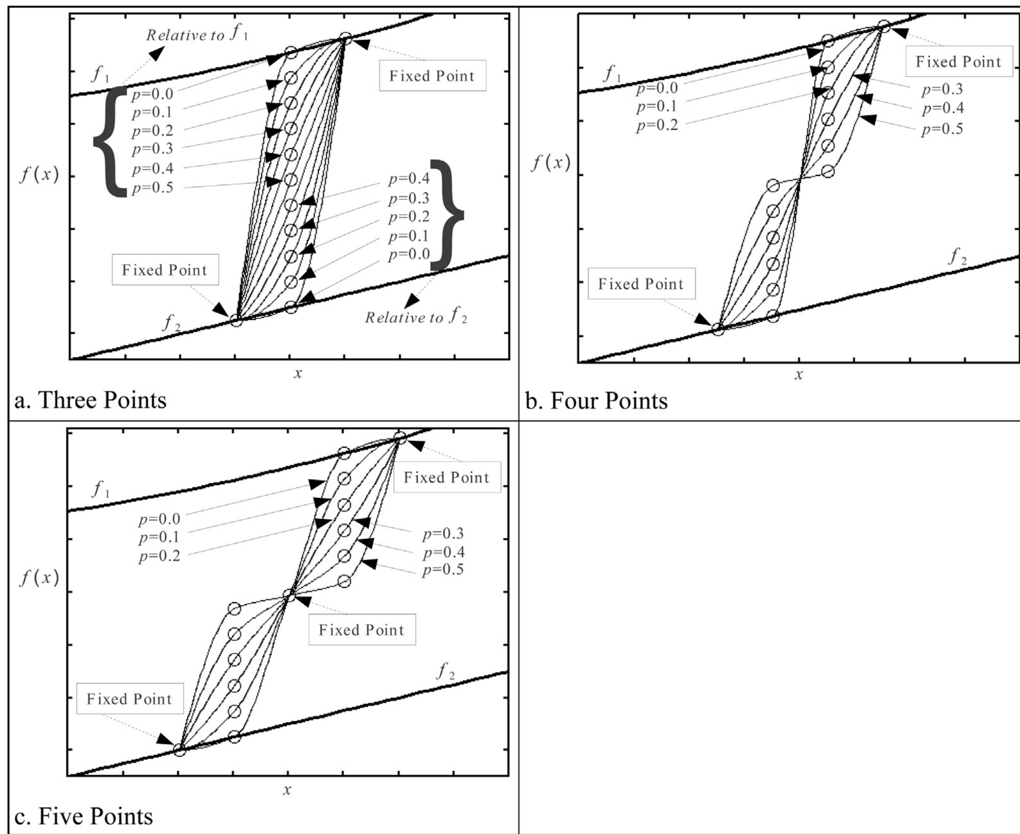


Fig. 5. Comparison between 3, 4 and 5 control points using a *hermite* interpolating polynomial.

parameter (t). As the name implies, t is roughly a measure of how stretched or lose is the connecting polynomial between the mesh control points. Assuming that mesh control points are connected through a thread, a $t=0$ indicates a loose thread while a $t=1$ indicates a tightly wrapped thread. We encourage using *hermite* interpolating polynomials for the extra degree of freedom they provide. The discussion from this point onward will assume the utilization of *hermite* interpolating polynomials.

Nevertheless, the reader should note that *hermite* interpolating polynomials require two more additional mesh control points over cubic splines as illustrated in Fig. 3a–c. Interpolation will still occur between the four control points discussed earlier. The additional two points only aid in forming the shape of the curve.

Let us now turn our attention to an issue that will further constrain the value of the p parameter. At lower values of p and/or tension parameter (t), the bounds of the interpolating polynomial tend to cross the maximum function boundaries set by the control points as illustrated in Fig. 5. This situation might not create an issue for most discontinuous functions. However, certain types of discontinuous functions mandate proper bounding of the interpolating polynomial to the upper and lower limits set by the control points. For example, if x denotes valve opening and $f(x)$ represents flow, then it would not be expected for the flow to arrive at its maximum value until valve opening reaches 100% ($x=1$). An interpolating polynomial that is not properly bounded will result in the undesirable situation leading to either a maximum flow before reaching 100% valve opening or worse leading to a negative flow before the valve is fully closed. This problem is known as the *bounding-box* problem in computational geometry (Filip, Magedson, & Markot 1986).

To resolve the problem, we need to bound the maximum and minimum values of the interpolating polynomial to the values set by control points 1 and 2 so that

$$f_1\left(x_{P_1}\right) \leq f_3(x) \leq f_2\left(x_{P_2}\right) \quad \textit{for} \quad x_{P_1} \leq x \leq x_{P_2} \tag{10}$$

The solution to the problem comes straight forward from calculus. To do so, the optimization routine needs to identify the maximum and minimum values of $f_3(x)$, compare them to those of control points 1 and 2, and finally, reject or accept the pair of (p_i, t) values based on adherence to condition (10).

2.4. Are four control points enough?

The discussion, so far, has assumed that we need at least four points to properly interpolate. However, we need a good justification to favour four points over three or five. This can be demonstrated by considering the plots of the *hermite* interpolating polynomial for three, four and five interpolating points shown in Fig. 5a–c.

When using a three-point interpolating polynomial, two of the points lie on the respective discontinuous functions. The x coordinate of the third point corresponds to the minimum jump effort location (g). The only degree of freedom available to tune the curvature, excluding the *hermite* tension parameter, is through the manipulation of the function value at the minimum jump effort point g. We varied $p/|AB|$ values from 0 to 0.5 relative to f_1 and f_2 in the upper and lower sections of the figure, respectively. As illustrated in Fig. 5a, the drawback of a three-point interpolating polynomial is that it always favours better closure towards one of the discontinuous functions over the other.

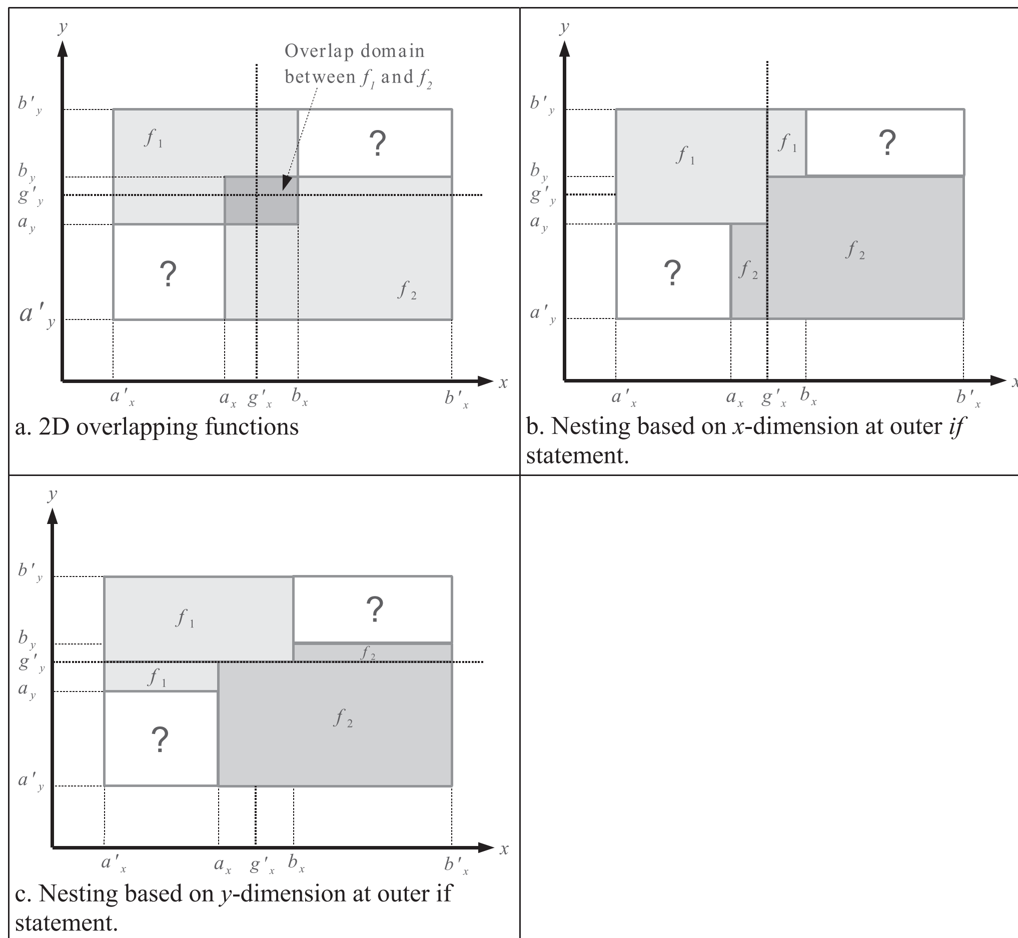


Fig. 6. An example illustrating applicability domains of two-dimensional overlapping functions f_1 and f_2 and the effect of conditional nesting on boundaries segregation.

For the case of four control points, we omitted the g point and relied only on two points separated by a distant h from each of the sides of the minimum jump effort location g. The interpolating function values, at the junctions with f_1 and f_2 are fixed at the values of their respective functions f_1 and f_2. We used equal values of p to distance interpolating function values at points 4 and 5. Thus, we have one degree of freedom (again excluding *hermite* tension) to smooth the transition between the interpolating polynomial f_3 and the functions f_1 and f_2, namely, p. The common intersection point between all generated curves is purely curvature related and has no relation to the g point discussed earlier.

For the case of five control points, we made use of the minimum jump effort location (g) to add the fifth point. The value of the interpolating polynomial f_3, at this point, is calculated and fixed at the mean of the two discontinuous functions f_1 and f_2 (i.e. $f_3(g) = 0.5[f_1(g) + f_2(g)]$). The values of the control points at the junctions with f_1 and f_2 are assigned the respective values of the functions. The values of these two points are also fixed. We also used constant values of p to distance the points located at $g-h$ and $g+h$ from their respective functions f_1 and f_2. We plotted the resulting interpolating values of $p/|AB|$ ranging from 0 to 0.5 in Fig. 5c.

The resulting curves for four-point interpolating polynomials (Fig. 5b) provide similar degrees of curvature to those obtained using five-point interpolating polynomial (Fig. 5c). Thus, we may comfortably conclude that a four-point interpolating polynomial is sufficient to provide good closure between the interpolating polynomial and the discontinuous functions.

2.5. Regularizing boundary and initial conditions

Discontinuities in boundary conditions usually take the form presented in Fig. 2b (i.e. $g = a = b$). Because the overlap domain is so small, any regularization will force f_3 to lie outside the overlap region. Moreover, since the switch between logical expressions (f_1 to f_3) or (f_2 to f_3) can be space, time or state variable dependent, we cannot evenly distribute f_3 span between f_1 and f_2. Even distribution could violate state variable dependency. Thus, the solution would be to insert an additional time interval to accommodate f_3 between f_1 *and* f_2. This makes sense since the set of boundary conditions at the overlap region does not coincide with any of the sets of boundary conditions belonging to the discontinuous functions.

Regularizing the form in Fig. 2b can take one of the forms in Fig. 3a–c. Using the forms presented in Fig. 3a and c would require calculation of more control points at locations before f_3 (points at the left side of the g point when replacing the *x-axis* with a time axis). The use of the form presented in Fig. 3b reduces the number of points located to the left of the g point to only one point, namely the additional control point required by the *hermite* interpolating polynomial. We should mention that accurate estimation of the value of the state variable at this point is not very important. This is due to the fact that the additional *hermite* control points are used to adjust the shape of the resulting curve bounded by the four points discussed earlier. The algorithm would work with any arbitrary value of the state variable at that point. However, accurate determination provides a better initial interpolation curve. After optimizing the shape of the curve through (9), the final curve would have

better closure at both ends of the interpolation region than a curve optimized with an arbitrary selection of the additional *hermite* control point. To accurately calculate the value of f_1 at this point, the integrator needs to pass through the control point and record a snapshot of the boundary condition values at that point. For time events, the event can be marked in integrator time-line. For state events, the integrator needs to switch to the branch of the logical expression containing the regularization function before realizing the existence of a shift in boundary conditions. Then, it needs to return back an interval h in time to record the snapshot. In both time and state event cases, such approaches add an extra unnecessary burden on the integrator. To mask the problem from the integrator, we allowed the integration routine to freely control integration step-size while taking snapshots of the time steps taken by the integrator. Once the regular expression shifts to the regularizing function, the location of that *hermite* control point is calculated through approximating past integration steps with an interpolating polynomial. Although computationally exhaustive, we think this approach provides a better estimation of the past value of the state variable. To avoid such computations, we can assume the value of the state variable at the left *hermite* control point to be equal to that at the g point. This assumption is used to calculate the additional *hermite* control point located to the right of the g point.

3. Two-dimensional functions

So far, we have discussed tackling the problem for one dimensional functions. What if z is a function of two variables (e.g. $z = f(x,y)$), where z poses one or more discontinuities along each of the dimensions. The discontinuous function may take a form like

$$f(x, y) = \begin{cases} f_1(x, y), x \in [a'_x, b_x], y \in [a'_y, b_y] \\ f_2(x, y), x \in [a_x, b'_x], y \in [a_y, b'_y] \end{cases} \tag{11}$$

Assuming $a'_x < a_x \leq b_x < b'_x$ and $a'_y < a_y \leq b_y < b'_y$ (Fig. 6a), if g'_x and g'_y are arbitrary selected as discontinuity boundaries along the x and y dimensions, respectively, a possible pseudo code of (11) could be written as either of the forms in (12).

When dealing with two dimensional relations, discontinuities are present as planes as illustrated in Fig. 6a. We can deduce some conclusions from projecting the domains of f_1 and f_2 into the x–y plane. The discontinuity planes formed by using form (12a) are illustrated in Fig. 6b. Similarly, the discontinuity planes formed by using form (12b) are illustrated in Fig. 6c. Notice that the difference in nesting of conditional statements only affects the resulting output within the overlap domain that is illustrated in Fig. 6a.

The solution strategy remains the same as for one dimension: the problem is still decomposed into discontinuity detection and discontinuity resolution sub-problems.

3.1. Two-dimensional discontinuity detection

Before elaborating on the approach to handle discontinuity detection and resolution in 2D, let us look at the how functions overlap in two dimensional space. Fig. 6a illustrates the case where there are overlaps between the two functions in both domains. In such cases the detection algorithm will detect an optimum switch point for each of the domains respective overlap intervals. When functions are adjacent to each other in one dimension and overlap in the other, the overlap domain in Fig. 6a reduces to a line. In such cases, the detection algorithm will only have one degree of freedom; that is to find the optimum switch point for the domain where overlap exists. When functions are adjacent to each other in both domains, the overlap domain reduces to a point in the projected 2D space. The detection algorithm has zero degrees of freedom in this case and the resulting discontinuity locations will correspond to the intersection point between the two functions.

It should be noted that, in 2D problems, detection of optimum switch points does not guarantee passage of the simulation trajectory through these points. It only helps in formulating the logical expression around the minimum jump effort point to aid in minimizing discontinuity while switching. This conclusion stimulates us to questioning the credibility of the obtained conventional simulation results when simulation trajectory does not pass through an overlapping domain (shown as question marks in Fig. 6). When not passing through an overlap domain, conditional expressions will extrapolate the use of discontinuous functions regardless of extrapolation applicability. This statement holds for all logical expressions involving the use of functions bounded by specified intervals. Since conventional modelling packages do not provide an apparent fix to this problem, it becomes the responsibility of the modeller to either ensure that the selected functions cover the intended unknown simulation bath, or to insert as many functions as possible (with differing domains) to cover a wider area to, hopefully, minimize extrapolation. Thus, we think it is essential to include the applicability domains of each logical branching expression as part of the model input file. Then, the simulation package would check whether the solution falls within the specified applicability domains and flags an alert (or stops simulation execution) when the simulation trajectory deviates from the applicable domains of the branched logical expressions.

The detection of an optimum jump points for 2D functions can be formulated as an extension of the 1D problem. For two discontinuous functions overlapping at $[a_x, b_x]$ and $[a_y, b_y]$ in x and y dimensions, respectively; the optimum switch point $g(x,y)$ is found through solving the optimization problem:

$$\min e(x, y) = |f_1(x, y) - f_2(x, y)|$$
$$s.t. = \begin{cases} x \in [a_x, b_x] \\ y \in [a_y, b_y] \end{cases} \tag{13}$$

```
If (a'x < x < g'x) then
    If (ay < y < b'y) then
        f(x,y) = f1(x,y)
    Else if [(ax < x) and (a'y < y < ay)] then
        f(x,y) = f2(x,y)
    End if
Else if (g'x < x < b'x) then
    If (a'y < y < by) then
        f(x,y) = f2(x,y)
    Else if [(x < bx) and (by < y < b'y)] then
        f(x,y) = f1(x,y)
    End if
End if

(12a)
```

```
If (a'y < y < g'y) then
    If (a'x < x < ax) and (y > ay) then
        f(x,y) = f1(x,y)
    Else if [(ax < x < b'x)] then
        f(x,y) = f2(x,y)
    End if
Else if (g'y < y < b'y) then
    If (a'x < x < bx) then
        f(x,y) = f1(x,y)
    Else if [(bx < x < b'x) and (y < by)] then
        f(x,y) = f1(x,y)
    End if
End if

(12b)
```

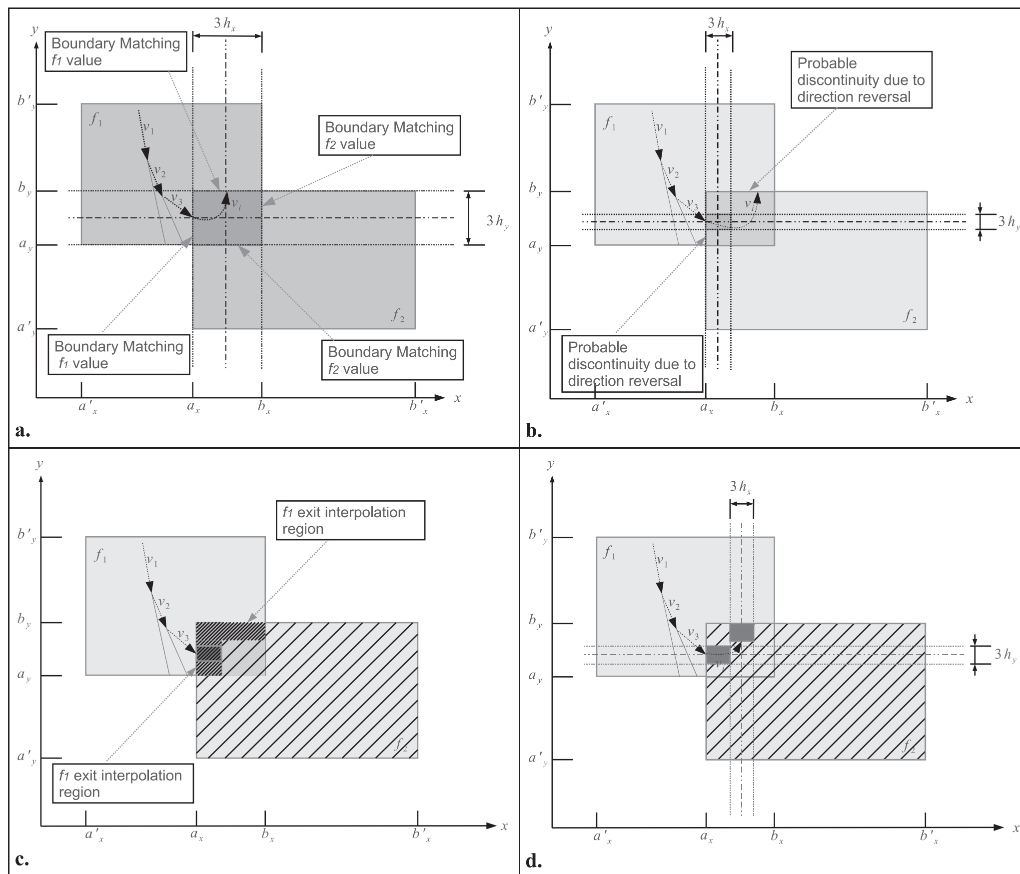


Fig. 7. Approaches I and II to resolving discontinuity.

As we indicated in the 1D case, once the g_x and g_y locations are determined, their values can be directly substituted into the constructed logical expression to minimize jump effort between the two adjacent discontinuous functions. The model can, then, be solved using any of the available integration packages. Nevertheless, since detection of optimum switch points does not always guarantee elimination of re-initialization of the ODE/PDE model at the switch point or accuracy of integrator-based interpolated solution afterwards, the need arises for a discontinuity resolution algorithm.

3.2. Two-dimensional discontinuity resolution

Once overlap boundaries between the discontinuous functions are determined through the detection algorithm, we need to interpolate between the discontinuous functions in order to eliminate discontinuity. We propose two approaches and highlight their pros and cons.

The simplest approach (approach I) is to cover the entire overlap domain with an interpolating polynomial. Boundaries of the interpolating polynomial will correspond to those of the continuous function at the boundary location as illustrated in Fig. 7. The fact that the values of the interpolating polynomial at its boundaries match that of the neighbouring functions facilitates smooth transition in all directions.

However, this approach comes at a cost. For a fixed number of control points per dimension, interpolation mesh size is overlap-domain size dependent. This means that mesh resolution will decrease as the size of the overlap domain increases and vice versa. Of course, increasing the number of control points for large overlap domains will resolve this problem but at heavy computational cost. Thus, we recommend adopting this approach for a relatively small overlap domain size. A typical *if* structure using this approach (based on Fig. 7a) is illustrated in (14).

Note that the logical expression well encapsulates the bounding domains of the discontinuous functions. Thus, the last *Else* statement is needed to indicate to the user that simulation trajectory is deviating from the specified functions' boundaries.

An alternative approach (approach II) would be to track a two dimensional trajectory vector $\vec{v}_n$ as simulation progresses and generate grid points of the interpolating polynomial once the logical expression shifts to the branch containing the interpolating polynomial as illustrated in Fig. 7b. The $\vec{v}_n$ vector tracks the coordinates of the independent variables of the composite function as simulation progresses. Full derivation of the underlining equations is presented in the appendix.

In this approach, the mesh is constructed at the intersection point between $\vec{v}_n$ and the overlap domain. The aim of the constructed mesh is to facilitate transition from the currently active discontinuous function to the function towards which $\vec{v}_n$ is heading. Once transition to the destination discontinuous function is complete the rest of the overlap domain is considered as a seamless part of the destination discontinuous function. This approach allows generation of variable grid sizes that are independent of the size of the overlap domain and with a fixed number of control points.

$$
\begin{aligned}
&\textit{If } \left[\left\{\left(a'_x \le x < a_x\right) \wedge \left(a_y < y < b'_y\right)\right\} \vee \left\{(a_x \le x \le b_x) \wedge \left(b_y < y \le b'_y\right)\right\}\right] \textit{ then} \\
&\quad f(x,y) = f_1(x,y) \\
&\textit{Else if } \left[\left\{\left(b_x < x \le b'_x\right) \wedge \left(a'_y \le y \le b_y\right)\right\} \vee \left\{(a_x \le x \le b_x) \wedge \left(a'_y \le y < a_y\right)\right\}\right] \textit{ then} \\
&\quad f(x,y) = f_2(x,y) \\
&\textit{Else if } \left[(a_x \le x \le b_x) \wedge \left(a_y \le y \le b_y\right)\right] \textit{ then} \\
&\quad f(x,y) = \textit{interpolate} \\
&\textit{Else} \\
&\quad \text{Print } \text{``}\textit{Illegal extrapolation}\text{''} \\
&\textit{End if}
\end{aligned}
\tag{14}
$$

The approach works well with one exceptional situation. This situation will arise when $\vec{v}_i$ changes direction, within the overlap domain, and returns back to the discontinuous function where it originally came from as illustrated in Fig. 7b. Since the overlap domain, with exception of the interpolation region, has been replaced with the values of the destination discontinuous function a discontinuity would probably occur at the boundaries of the overlap domain with the function where the vector has originally come from. Such a situation is solvable through formulating an additional exit interpolating polynomial with the original function as illustrated in Fig. 7c. Note that even the entry region (cross-hatched) is treated as a possible interpolating region to move back to f_1 from the overlap region. The fine-hatched region resembles the entire area at which interpolation might occur. However, the generated mesh will only cover the portion where $\vec{v}_i$ is headed as illustrated in Fig. 7d. Note that this problem would never occur if approach I is used because $\vec{v}_i$ will always fall in the region of the interpolating polynomial once it is inside the overlap region as illustrated in Fig. 7a. Two advantages arise from using approach II:

1. It allows for variable size mesh, i.e. h_x and h_y can be arbitrary selected as long as the resulting mesh does not cross the overlap domain.
2. Only four points are needed per interpolation dimension regardless of the size of the overlap domain.

However, more checks are needed in this approach over the approach I. A typical conditional structure pseudo code is illustrated in (15).

3.3. How legal is "illegal" extrapolation?

As we discussed earlier, extrapolation occurs when trying to join the two discontinuous functions by a polynomial that lies outside their designated domains. This is illustrated in Figs. 2d and 7 (domains marked by question marks) for 1D and 2D functions, respectively. There are two reasons (cases) behind alerting the modeller about illegal extrapolation:

1. The extrapolation domain might be defined by a function exhibiting a behaviour that is different from the behaviour of the functions to be extrapolated. In such cases, extrapolation will result in erroneous simulation output.
2. Either or both of the functions to be linked might not be mathematically defined in the extrapolation region (e.g. division by zero). In such cases control points 3 and 4 cannot be calculated due to unavailability of function values at the location of these points.

The modeller will obtain a less than accurate result in the first case. However, if the modeller is confident about the consistency of the behaviour between the extrapolation region and the functions to be extrapolated, he or she can simply alter domain boundaries of the functions to append the extrapolated region to one of them, divide it between the two functions or, even better, append it to both functions and rely on the detection optimizer to locate the best transition point g.

As for the second case, the integrator will simply stop integrating because the values of the functions at points 3 and 4 are dependent on the respective values of functions 1 and 2. However, the dependency can be broken by eliminating function evaluations at these two points. We should recall that function evaluations at points 3 and 4 are needed to calculate the amount of dip based on p parameter. If some curvature smoothness at the junction points between the interpolating polynomial and the discontinuous functions can be sacrificed in quest for continuity, then the integrator can extrapolate between the values of the two discontinuous functions using their respective boundaries that are adjacent to the extrapolation domain.

As we might expect, the second solution will work for cases 1 and 2. However, it will not eliminate errors associated with the first extrapolation case. So, it still becomes the modeller responsibility to tackle the first case by inserting an appropriate function to define the region that might otherwise be erroneously extrapolated.

3.4. Mesh generation

In order to interpolate, a mesh needs to be generated. For one-dimensional problems, the mesh reduces to a one-dimensional set of points. The 2D+ problems require an elaboration on mesh generation methods.

Mesh generation is approach dependent. Generating the mesh using approach I is a fairly easy task since the mesh will cover the entire overlap region. The values of the boundary points surrounding the overlap region will always correspond to the neighbouring continuous functions adjacent to the overlap domain as illustrated in Fig. 7a.

```
If (active_point_coordinate∉overlap_domain) then
    entry_completed = false; first_overlap_entry = true; first_exit_attempt = true
    If (active_point_coordinate∈f1 range) then
        f= f1(x,y)
        Active_function = f1
    Else if (active_point_coordinate∈f2 range) then
        f= f2(x,y)
        Active_function = f2
    Else
        Print "Illegal extrapolation"
    End if
Else if (active_point_coordinate∈overlap_domain) then
    detect_entry_intersection_plain;
    If first_overlap_entry = true then
        construct_entry_mesh;
        first_overlap_entry = false
    End if
    If (active_point_coordinate∈entry_interpolation)∧(not entry_completed) then
        f = entry_interpolate
    Else
        entry_completed = true
        If (active_point_coordinate∈exit_interpolation) then
            If first_exit_attempt = true then
                construct_exit_mesh
                first_exit_attempt = false
            End if
            f = exit_interpolate
        Else
            f = f_Destination_function(x,y)
        End if
    End if
End if
```
(15)

For approach II, mesh generation is more complex. The extra complication arises from the tracking of $\vec{v}_i$. We will discuss four methods to construct the mesh around the intersection of the $\vec{v}_n$ with the discontinuity plane. We will briefly explain each method and provide our reasoning for selecting one of them. For

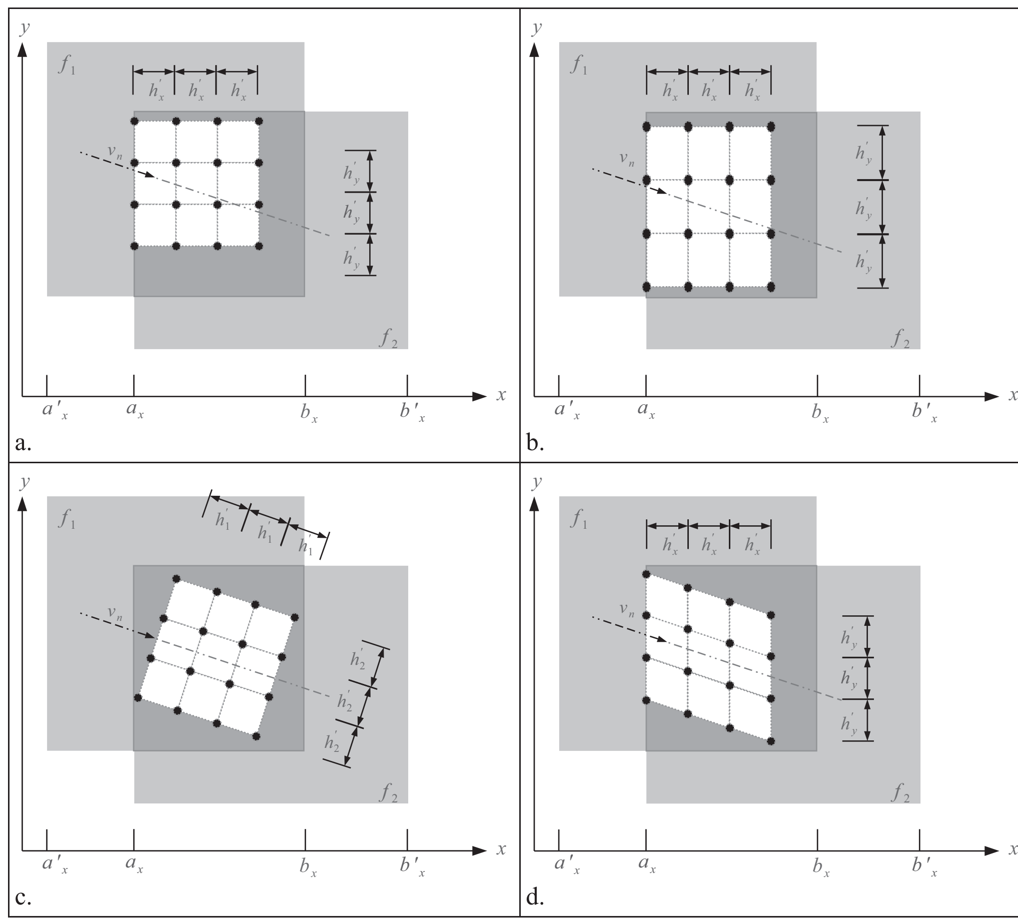


Fig. 8. Four ways to construct a mesh around a vector-plane intersection point.

simplicity, we will demonstrate examples using a discontinuity plane orthogonal to *x-axis*. However, the concept applies to discontinuities orthogonal to either of *x*- or *y*-axis.

The first method constructs a squared mesh around the discontinuity point as illustrated in Fig. 8a. Values of h'_x and h'_y are measured with respect to their respective *x*- and *y*-axes. The size of the mesh is fixed. The distribution of the mesh control points along the sides of $\vec{v}_n$ is dependent on the slope of the $\vec{v}_n$.

The second method is similar to the first one with the exception that the size of the mesh is expandable in the direction that is perpendicular to the discontinuity plane. The advantage of this method is that it allows for better distribution of the control points along each side of the $\vec{v}_n$ vector as illustrated in Fig. 8b. As can be deduced from the figure, vector $\vec{v}_n$ is almost always leaning towards one set of the mesh control points over the other.

The third method aligns the grid with the direction of $\vec{v}_n$. This method better distributes grid points along the sides of $\vec{v}_n$, compared to the former two methods as illustrated in Fig. 8c. Note that h'_1 and h'_2 are measured parallel and orthogonal to $\vec{v}_n$, respectively, but not relative to *x*- and *y*-axes. Since the grid is aligned to $\vec{v}_n$ while the logical expression is based on a discontinuity that is orthogonal to either *x*- or *y*-axis, logical statements around interpolation region become functions of the direction of $\vec{v}_n$. Since the generated mesh is not aligned with overlap domain, it becomes a difficult task to superimpose the mesh on the logical expression.

The fourth method relies on fixing h' along both dimensions while shifting lines parallel to the continuous domain to align grid with $\vec{v}_n$. Fig. 8d illustrates the concept for the case where the *x*-dimension being the discontinuous one. The fourth method resolves the drawbacks of the previous three methods. Thus, we opted for implementing this method in grid construction.

4. N-dimensional functions

4.1. N-dimensional discontinuity detection

To generalize, for two *n*-dimensional discontinuous functions, discontinuity detection detects the overlap region between the two discontinuous functions. It also detects the optimum switch point between the two discontinuous functions. The position of the two functions, relative to the overlap region and the location of the optimum switch point, assists in formulating the logical expression. If functions do not overlap in any of the dimensions, the algorithm flags an error and simulation execution stops.

4.2. N-dimensional discontinuity resolution

Discontinuity resolution takes the form of an interpolating polynomial that connects the two discontinuous functions. For one-dimensional discontinuous functions, the interpolating polynomial is best formulated around the minimum jump effort point.

For discontinuous functions of dimensions greater than one, the solution can follow one of two approaches:

1. The first approach relies on constructing an interpolating polynomial that covers the entire overlap region. This path is suitable for relatively small overlap regions. For large overlap regions, interpolating polynomial mesh resolution can be enhanced by

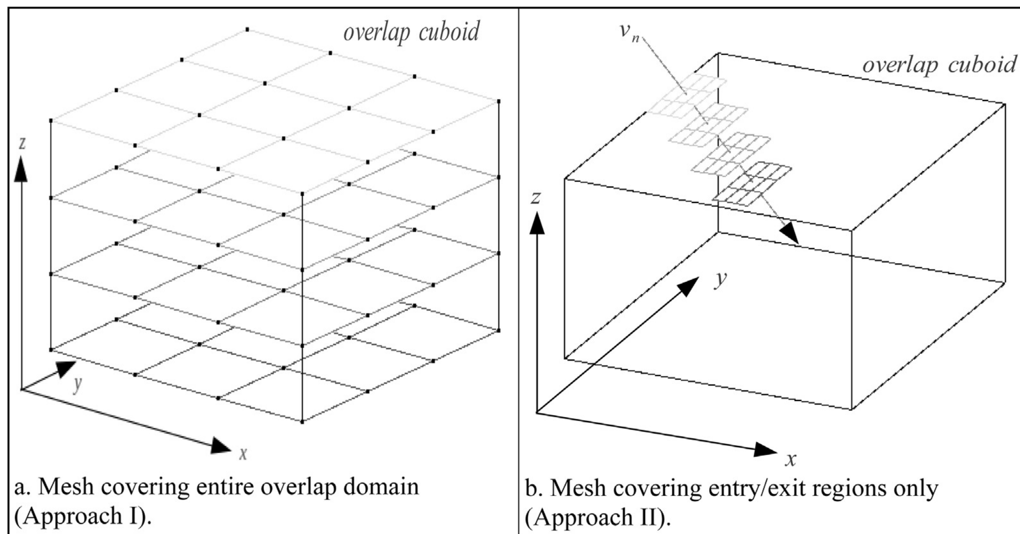


Fig. 9. Representation of the two types of generated meshes in a 3D cuboid overlap domain.

increasing the number of control points at heavy computational cost.

2. The second approach constructs one mesh and possibly a second one. The first mesh is constructed at entry to the overlap region. It facilitates smooth transition between the active discontinuous functions, at the entry point of the overlap region, and the destination one. Once transition occurs, the rest of the overlap region is treated as if it were part of the discontinuous function towards where the simulation vector is heading. In situations where the simulation vector reverts back to the function where it originally came from, an exit mesh is constructed to resolve discontinuity at exit location. This path has the advantage of varying the mesh size based on user specification while maintaining a fixed number of control points.

Fig. 9a and b illustrate mesh generation for an overlap-domain between two 3D discontinuous functions using approaches I and II to discontinuity resolution, respectively.

The total required number of mesh points is an exponential function of the dimensions of the composite function and can be calculated as

$$\textit{number of mesh points} = m^n \tag{16}$$

where m is the number of control points per dimension and n is the number of dimensions.

To ensure smooth transition between the two discontinuous functions, at least four control points are needed per dimension. In case of *hermite* interpolating polynomials, six control points are needed per dimension. Fig. 10 illustrates the relationship between the number of control points needed and the dimensions of the composite function. Although computational power and capacity are machine dependent, we can deduce from the plot the existence of a threshold beyond which computational power and/or machine space (memory or hard disk) becomes prohibitive. For example, for a tenth dimension discontinuous function, a cubic spline would require a mesh composed of 1,048,576 points. That is a megabyte of memory/disk space per discontinuity. The problem becomes worse when using *hermite* interpolating polynomials. For a tenth dimension discontinuous function, the *hermite* interpolating polynomial requires 60,466,176 mesh points. This is about 58 megabytes of memory/disk space (1 MB = 2^{20} bytes) per discontinuity encountered.

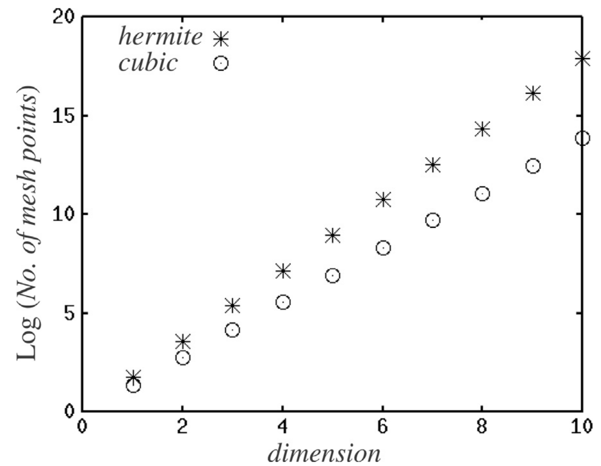


Fig. 10. A semi-log plot of number of mesh points required versus discontinuous function dimension.

One might think that we could use sparse matrix algebra to conserve memory. However, this is not possible since we only have four or six points per dimension, all of which contribute to the shape of the interpolation curve, resulting in a very dense matrix. Yet, some solutions can help reducing the implications of this problem or eliminating it. For example, the number of dimensions can be reduced if any dimension exhibiting constant values throughout the interpolation region is omitted from the interpolation mesh. Also, since usually hard disk space is more abundant than memory, the entire mesh can be saved in a computer hard drive using binary files to accelerate simulation program access to these mesh-point files. Lastly, instead of generating the mesh once at the first entry to the interpolation region and saving it, the simulation routine can opt to generate the mesh at each interpolation run and dispose it immediately after the composite function value is computed to free memory/hard disk space. The latter resolution saves a tremendous amount of disk space by dynamically allocating mesh space to compute function values and freeing the space once the function value is computed. However, additional CPU time is required to construct the exact same mesh at every function evaluation within the interpolation region. Of course, a combination of one or more of the above resolutions will result in a more efficient and/or robust algorithm by taking into consideration CPU speed and the amount

of available memory and hard disk spaces for each machine the algorithm is implemented on. For example, the simulation routine can be programmed to:

1. Generate interpolation mesh only once in memory when memory space is abundant.
2. Once memory occupied space reaches a specified maximum, the simulation routine switches to storing a one-time generated mesh in the machine hard drive.
3. If hard drive space is limited or has reached a critical level, the routine shifts to dynamically creating and destroying meshes at each function evaluation inside the interpolation region.

To further enhance efficiency, the routine can be programmed to optimize memory utilization by loading lower dimension functions' meshes into memory while saving higher dimension ones to hard disk. The prior knowledge of the dimension of each composite function will assist the simulation routine in calculating the maximum amount of occupied hard disk/memory space beyond which dynamic allocation and destruction of interpolation meshes (bullet 3) should be used instead of a single-time generated mesh (bullets 1 or 2).

Such a resolution is hardware dependent. Thus, below certain machine hardware specifications and based on computed mesh size for each interpolating polynomial in a simulation model, the simulation routine can flag an error message prior to starting simulation run indicating the inability to run the model on a specified machine. However, we think modern hardware capabilities extend far beyond such minimum specifications.

Last, it is good to shed some light on whether this work eliminates the need for implicit integrators. The answer is no. Taking Fig. 3 as an example, we notice that slope changes are very evident between each of the sub-functions and their respective interpolating polynomial. An explicit integration routine with a fixed integration step size can easily overlook these slope changes, even in a regularized composite function, resulting in sever simulation errors. Of course, minimizing integration step length might resolve the issue but at the cost of increased simulation run-length. The use variable integration step-size in implicit integrators ensures the adjustment of the step-size as and when required. Larger integration steps are used when integration error is within bounds. Whenever integration error exceeds the bounds, integration step is halved and error is recalculated. The implicit integration routine adjusts integration step size when moving between discontinuous sub-functions and their respective interpolating polynomial. Thus, the use of implicit integration routines is still favoured even after model regularization.

5. The algorithm

Algorithm implementation is programming language dependent as it involves either modification of conditional statements or a complete rewrite of the discrete composite function to regularize it. In compiler-based modelling languages such as gPROMS (PSE Enterprise, 2012), we recommend impeding the code within language compiler. However, this solution might not be feasible for general purpose modelling languages such as MATLAB or GNU Octave or even general purpose imperative languages such as C++, FORTRAN or Pascal. In such cases, the programmer can write his/her custom code to iterate through discretized composite functions and transform them to their regularized counterparts.

Fig. 11 illustrates a simplified flowchart diagram of the algorithm. A simplified step-by-step procedure that should be executed by the modelling language follows:

STEP-01: Start simulation run
STEP-02: Check for the availability of any functions containing logical expressions or standalone logical expressions involving continuous variables (i.e. of real or float types) inside original model code.
STEP-03: Search for an optimum switch point that minimizes the difference in values between any two sub-functions within their overlap domain.
STEP-04: Adjust the standalone logical expression or the one within the composite function to account for the new switch point.
STEP-05: If resolution is enabled by the modeller, reconstruct a regularized logical expression from the discretized one (recommended).
STEP-06: Repeat STEP 2 and STEP 3 until all logical expressions within modeller's code are handled.
STEP-07: Start Integration and Initialize variables.
STEP-08: Integration routine advances integration step if final integration step is not reached.
STEP-09: Update $\vec{v_i}$ for each composite regularized function.
STEP-10: If composite regularized function parameters are not within the interpolation region, calculate function f value using the provided discontinuous sub-function that lies within the active domain. If parameters are within the overlap domain, check if this is the first entry to the overlap region in order to generate the interpolation grid. If the grid is already generated, use interpolating polynomial f_3 to calculate f.
STEP-11: Repeat STEPS 8–11 until simulation completes.

6. Examples and discussion

6.1. Example 1: Regularizing discontinuity in heat transfer coefficient calculation

We implemented the detection and resolution algorithms in a C++ code. Then, we linked the compiled code to a gPROMS (PSE Enterprise, 2012) reactor model through gPROMS Foreign Object Interface (FOI). The gPROMS reactor model is a simplified model representing the reactor unit from the Patent by Minkkinen, Mank, and Jullian (1993). Catalyst reaction and adsorption constants are obtained from Barrer and Sutherland (1956). Simplified one-dimensional *hermite* interpolation code is presented by Bourke (2011). Breeuwsma (2011) presented a general C++ and Java codes for multidimensional interpolation that can be used in conjunction with any one-dimensional interpolation method. We combined the codes of Bourke and Breeuwsma to formulate our C++ multidimensional *hermite* interpolation routines.

We tested the effect of transition from laminar to turbulent flow regimes on the wall heat transfer coefficient. For laminar flow, we used the simplified constant heat-flux equation of $Nu_d = 4.364$. We assumed that *Reynolds* number ranges from 0 to 2310. For turbulent flow we used the Gnielinski correlation (Kreith, 2000):

$$Nu_d = \frac{(f/2)(Re_d - 1000)Pr}{1 + 12.7(f/2)^{1/2}(Pr^{2/3} - 1)} \left[1 + \left(\frac{d}{L}\right)^{2/3}\right] \tag{17}$$

where:

$$f = [1.58 \ln(Re_d) - 3.28]^{-2}$$

$$2,300 < Re_d < 10^6$$

$$0.6 < Pr_d < 2,000$$

$$0 < d/L < 1$$

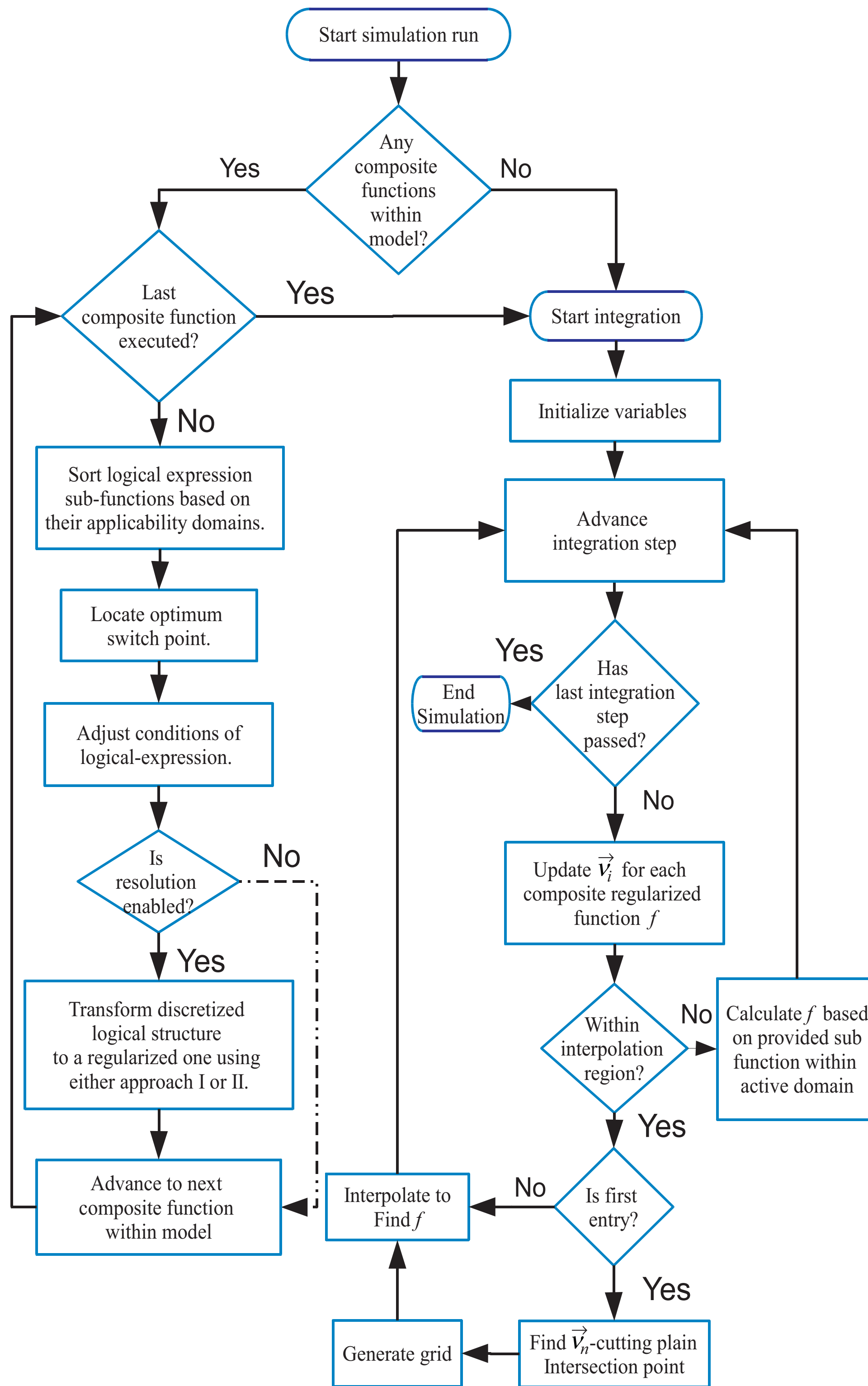


Fig. 11. A simplified flowchart illustrating flow of the algorithm presented in this work. Solid lines represent the more preferred path while the dashed line represents the less preferred one.

Thus, *Nusselt* number can be expressed as

$$Nu_d = \begin{cases} 4.34 & 1 < Re_d < 2310 \\ \dfrac{(f/2)\,(Re_d - 1000)\,Pr}{1 + 12.7(f/2)^{1/2}\,(Pr^{2/3} - 1)} \left[1 + \left(\dfrac{d}{L}\right)^{2/3}\right] & 2300 < Re_d < 10^6, 0.6 < Pr_d < 2000 \end{cases} \quad (18)$$

A plot of Nu_d versus Re and Pr for laminar and turbulent flow regimes is illustrated in Fig. 12.

Table 1
Reported simulation time for several runs using varying discretization nodes.

No. of discretization nodes	Time (s)	Time (s)	Time (s)	Time (s)	Time (s)
	Base case	Conventional Absolute	Conventional Above base	This work Absolute	This work Above base
10	37	38	1	42	5
20	4	7	3	8	4
50	8	11	3	11	3
100	9	20	11	17	8
200	14	34	20	28	14
300	21	50	29	41	20
400	29	69	40	55	26
500	35	82	47	66	31

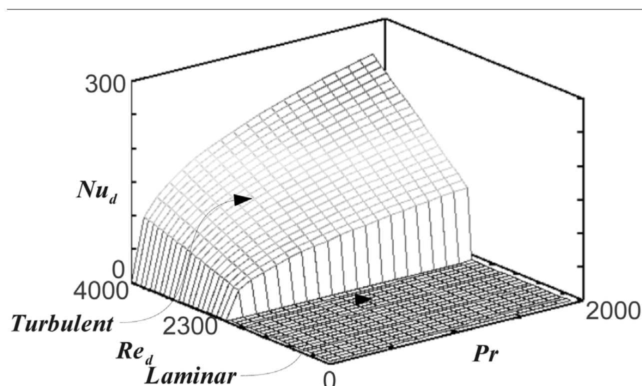


Fig. 12. A plot of *Nusselt* number versus *Prandtl* and *Reynolds* numbers for Laminar and Turbulent flow regimes.

We expect to observe a decline in the time required to perform a simulation run, when using the approach presented in this work when compared with conventional simulation reinitialization procedures. Since the developed reactor model discretizes axial space to convert PDEs to ODEs, we intend to use the number of discretization points as a variable to test our theory. We expect our code to best perform at large numbers of discretization points. The performance should approach that of conventional simulation techniques as the number of discretization points is reduced. This is due to the fact that the number of equations requiring initialization is directly proportional to the number of discretization points.

To establish a baseline for our analysis and to eliminate the bias introduced by every simulation run on the analysis, we recorded machine time taken to complete a constant velocity simulation that does not pass through any discontinuities for a set of axial discretization nodes that span from 10 to 500 as outlined in Table 1. To eliminate any variance in reported data (due to interfering machine background tasks) we repeated each run three times and reported the average outcome of the three runs on the table. We should also mention that the reported base case is based on conventional simulation runs. We noticed a consistent additional one second when using FOI to report base case results. We think the additional one second is attributed to initiation and termination of the link between gPROMS and the FOI. We should also mention that results on Table 1 are generated using a single lumped heat transfer coefficient that is based on feed conditions for the entire reactor length. Also, simulation runs were performed on a machine equipped with an Intel i5 processor, 4 GB RAM and running a Linux operating system.

After establishing the base case, we applied a sinusoidal input to the feed velocity that crosses Reynolds boundary of 2300 between the two correlations ten times. Results obtained are plotted in Fig. 13. With the exception of the reported time using ten discretization nodes, the rest of the points closely resemble straight lines. Excluding the point corresponding to ten discretization nodes

Table 2
Regression results for correlating simulation run length with number of discretization nodes.

	Slope	Intercept	Correlation coefficient
Conventional	0.15869	3.40857	0.9992
This work	0.12263	4.78063	0.9993

(explained later) and applying regression analysis between the number of discretization nodes and the absolute simulation run length for the conventional case and this work yields the tabulated results in Table 2. The slopes resulting from the regression analysis represent the run length time per discretization node. Dividing the slope resulting from this work (0.12263) by the slope resulting from conventional runs (0.15869) provides the fractional run length time elapsing from this work per elapsed run length of conventional runs (0.7728). The results show that using the approach provided in this work results in about 23% saving in run length time over conventional discontinuity handling techniques at least for 2D discontinuous functions. Of course, the same conclusion would have been achieved had we directly regressed run length time for conventional discontinuity handlers against the results obtained in this work bypassing the inclusion of discretization nodes in regression analysis.

As it appears from the figures and supported by the computational results, there is a consistent drop in the reported simulation time when using the new approach for two dimensional discontinuous functions. Also, the new approach becomes more attractive as the number of the state variables, to be initialized, increases.

As the number of state variables decreases, both approaches to resolving discontinuity report closer simulation times. However, since initialization itself introduces errors in the solution, the new approach still holds the advantage of not reinitializing any state variables.

As illustrated in Fig. 13a, there is a sudden increase in the reported time when using ten discretization points. This sudden increase in simulation time is mainly attributed to the decline in discretization resolution. As the number of space discretization points decreases, the integrator is forced to take smaller integration steps in order to meet the specified error tolerance criteria for a successful integration step.

6.2. Example 2: Regularizing boundary and initial conditions of a PSA unit

Pressure swing adsorption (PSA) processes are considered among few of the processes that exhibit continuous dynamics from the moment they are started until they are shut down. Any PSA column undergoes a sequence of steps whereby inlet and exit valves are automatically opened and closed or products are redirected through switch valves. Feeds are introduced at some steps and products are collected at either the same step or at different steps. Ruthven, Farooq, and Knaebel (1994) and Yang (1997)

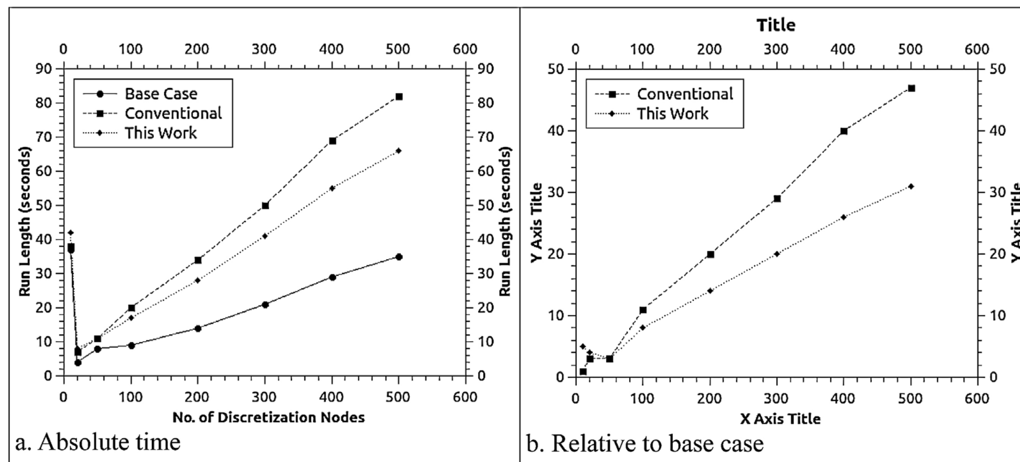


Fig. 13. Simulation Run Length versus number of internal discretization nodes.

discuss general concepts of PSA units. We modelled the PSA unit presented by Minkkinen et al. (1993) for the separation of iso-from normal paraffins. The underlying differential equations for the dynamic distributed parameter system are outlined and discussed by Silva, Silva, and Rodrigues (2000). A simplified isothermal set of model equations is used to demonstrate the concept. The gas phase component mass balance is presented in (19) and the solid phase component mass balance is presented in (20). Velocity distribution across the vessel is obtained by solving the overall mass balance equation assuming that total concentration is a function of time only. The underlying ODE is presented in Eq. (21). All presented equations are in the normalized form.

$$-\frac{1}{Pe_m}\frac{\partial^2 y_i}{\partial x^2}+\frac{\partial(Uy_i)}{\partial x}+\frac{\partial y_i}{\partial \tau}+\frac{y_i}{C_T}\frac{\partial C_T}{\partial \tau}+\frac{\zeta_{m_i}}{C_T}\frac{\partial Q_i}{\partial \tau}=0 \tag{19}$$

$$\frac{\partial Q_i}{\partial \tau}=\frac{N_g^m}{\zeta_{m_i}}C_T\langle y_i-y_i\rangle \tag{20}$$

$$C_T\frac{\partial U}{\partial x}+\frac{\partial C_T}{\partial \tau}+\sum_{i=1}^{n}\zeta_{m_i}\frac{\partial Q_i}{\partial \tau}=0 \tag{21}$$

No mass exchange is assumed between adjacent solid phase pellets (adsorbent). The adsorbent only exchanges mass with the gas phase. Thus, Eq. (20) can simply be substituted into (19) to obtain an overall mass balance around the vessel. The gas phase component mass balance is a PDE that requires two boundary conditions in addition to the initial condition. Boundary conditions at both ends of the vessel change from Nuemann to Robin and vice versa depending on the active PSA step as illustrated in Fig. 14.

At any time instant during the simulation, a velocity profile is obtained through solving a one dimensional Dirichtlet boundary ODE in space only. However, the location of the boundary condition is PSA step dependent as illustrated in Fig. 14.

Each step undergone by a PSA column possesses differing boundary conditions that uniquely identifies the step from its sister steps as illustrated in Fig. 14. The switch from one step to the other is either time dependent (e.g. adsorption and desorption steps) or state variable dependent (e.g. pressurization and de-pressurization). Regardless of the solver used, conventional solution of PSA column differential equations requires re-initialization of the ODE/DAE system at the start of each step in the sequence. The model repeats the cycles until a desired maximum number of cycles is reached or an error tolerance is reached on exit concentrations at the end of either depressurization or desorption step signifying the reach of a cyclic steady state. A typical conventional *if* structure that controls transitions between steps is illustrated in (22).

Cycle = 0
Repeat
    Cycle Time = 0
    If (Cycle Time >= 0) and (Cycle Time <= T_1) *then*
        Step = Pressurization; reinitialize model equations based on Pressurization BCs;
        run simulation for step length
    Else if (Cycle Time > T_1) and (Cycle Time < T_2) *then*
        Step = Adsorption; reinitialize model equations based on Adsorption Bcs;
        ; run simulation for step length
    Else if (Cycle Time >= T_2) and (Cycle Time <= T_3) *then*
        Step = Depressurization; reinitialize model equations based on Depressurization Bcs;
        run simulation for step length
    Else if (Cycle Time > T_3) and (Cycle Time < T_4) *then*
        Step = Desorption; reinitialize model equations based on Desorption Bcs;
        run simulation for step length
    End if
    Cycle = Cycle + 1
Until (Cycle = Max Cycles) or (| $Y_{Cycle} - Y_{Cycle-1}$ | <= Tolerance)

Where: $T_1 = Time_{PressurizationStep}$, $T_2 = T_1 + Time_{AdsorptionStep}$, $T_3 = T_2 + Time_{DepressurizationStep}$,
$\boldsymbol{T_4 = T_3 + Time_{DesorptionStep}}$ (22)

In this work, we linked the boundary conditions for any two consecutive steps through the use of 1D *hermite* interpolating

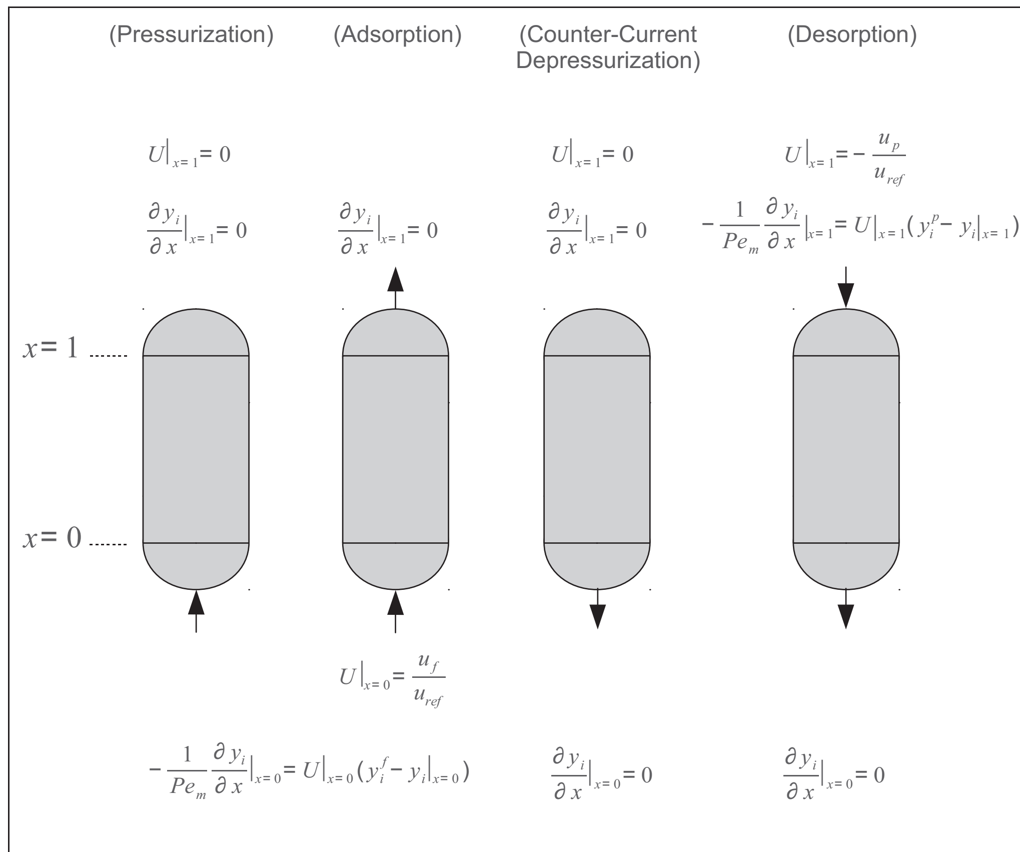


Fig. 14. Velocity and component balance boundary conditions for each of Skarstrum PSA cycles.

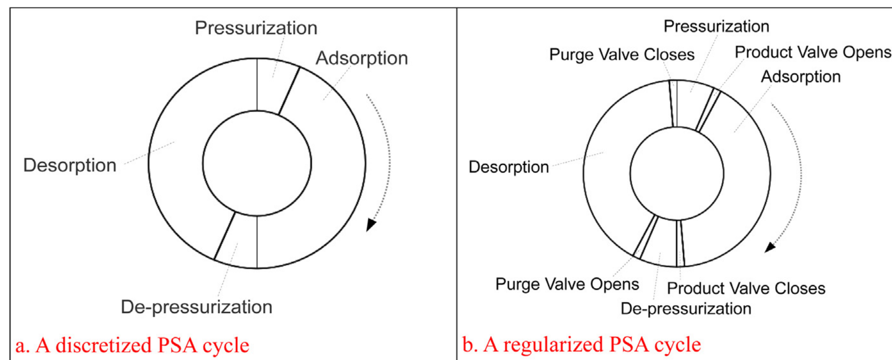


Fig. 15. Comparison between a discretized and a regularized PSA cycle illustrating relative time span for each of the cycle steps and valve opening/closure span for $w = 10$ s. The arrows indicate cycle direction.

polynomials as illustrated in (23) for velocity boundaries and (24) and (25) for concentration boundaries at each side of the PSA column.

At each time step, a velocity profile is obtained through solving an ODE equation with one boundary condition. However, the location of the boundary condition is PSA cycle step dependent. So, in order to regularize velocity boundaries, we had to calculate the entire velocity profile in the FOI and then pass it to gPROMS model. Otherwise, we would be forced to discretize velocity distribution in gPROMS model. The velocity profile is calculated using an ODE solver provided by GNU Scientific Library (GSL, 2011) and then transferred to gPROMS model through gPROMS FOI. All Foreign Object and Foreign Object Interface codes are programmed using C++ programming language.

$$U|_{x=0 \, or \, x=1} = f\left(Time_{Cycle}\right) = \begin{cases} U|_{x=1} = 0 & 0 \le Time_{Cycle} \le T_1 \\ Interpolate & T_1 < Time_{Cycle} < T_2 \\ U|_{x=0} = \left(U_f/U_{ref}\right) & T_2 \le Time_{Cycle} \le T_3 \\ Interpolate & T_3 < Time_{Cycle} < T_4 \\ U|_{x=1} = 0 & T_4 \le Time_{Cycle} \le T_5 \\ Interpolate & T_5 < Time_{Cycle} < T_6 \\ U|_{x=1} = -\dfrac{U_p}{U_{ref}} & T_6 \le Time_{Cycle} \le T_7 \\ Interpolate & T_7 < Time_{Cycle} < T_8 \end{cases} \tag{23}$$

$$\frac{\partial y_i}{\partial x}|_{x=0} = f\left(Time_{Cycle}\right) = \begin{cases} U|_{x=0}(y_i^f - y_i|_{x=0}) & 0 \le Time_{Cycle} \le T_1 \\ Interpolate & T_1 < Time_{Cycle} < T_2 \\ U|_{x=0}(y_i^f - y_i|_{x=0}) & T_2 \le Time_{Cycle} \le T_3 \\ Interpolate & T_3 < Time_{Cycle} < T_4 \\ 0 & T_4 \le Time_{Cycle} \le T_5 \\ Interpolate & T_5 < Time_{Cycle} < T_6 \\ 0 & T_6 \le Time_{Cycle} \le T_7 \\ Interpolate & T_7 < Time_{Cycle} < T_8 \end{cases} \quad (24)$$

$$\frac{\partial y_i}{\partial x}|_{x=1} = f\left(Time_{Cycle}\right) = \begin{cases} 0 & 0 \le Time_{Cycle} \le T_1 \\ Interpolate & T_1 < Time_{Cycle} < T_2 \\ 0 & T_2 \le Time_{Cycle} \le T_3 \\ Interpolate & T_3 < Time_{Cycle} < T_4 \\ 0 & T_4 \le Time_{Cycle} \le T_5 \\ Interpolate & T_5 < Time_{Cycle} < T_6 \\ U|_{x=1}(y_i^p - y_i|_{x=1}) & T_6 \le Time_{Cycle} \le T_7 \\ Interpolate & T_7 < Time_{Cycle} < T_8 \end{cases} \quad (25)$$

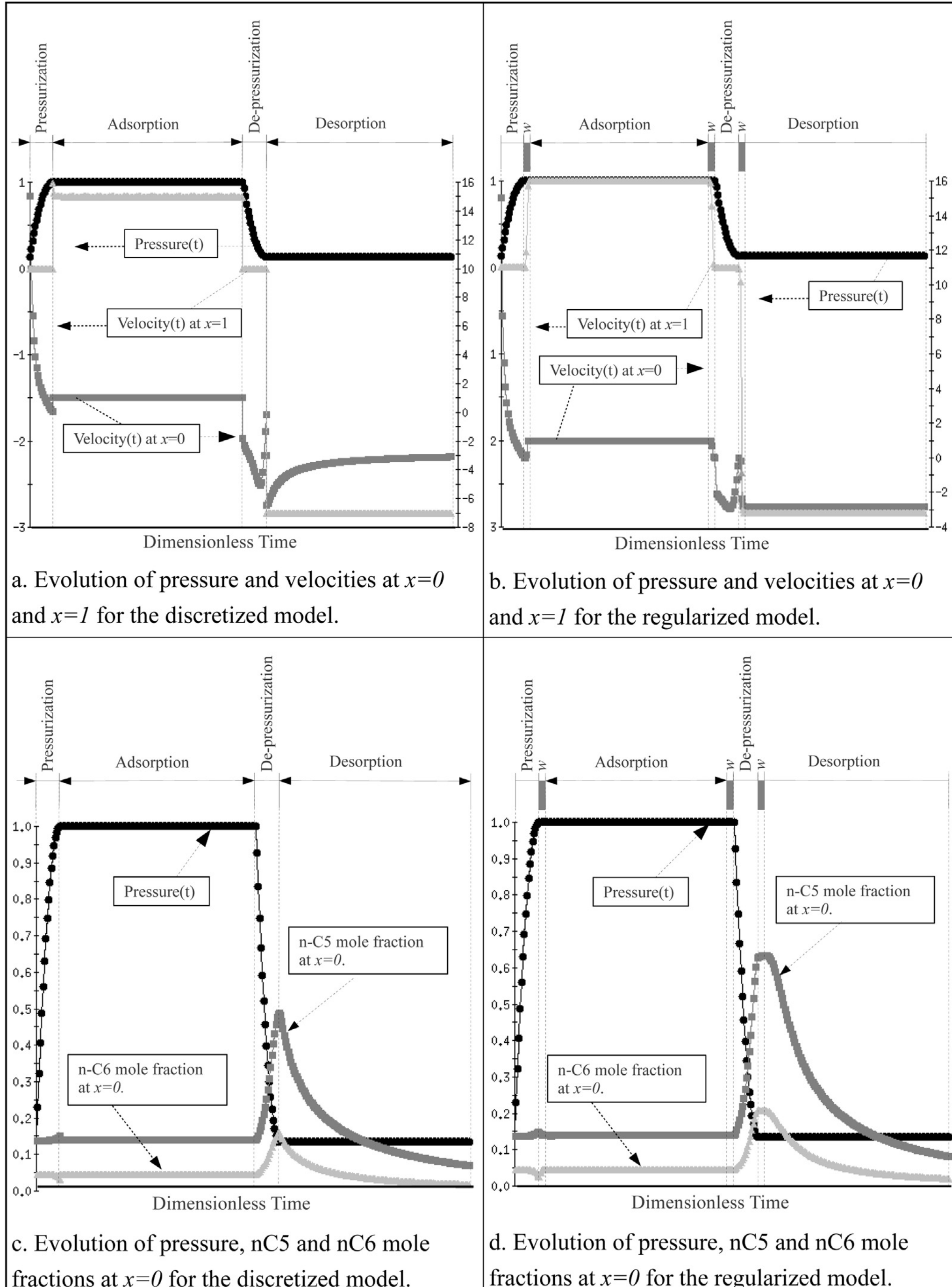


Fig. 16. Evolution of pressure, velocity and concentration curves over time for discretized and regularized PSA models.

where:

T_1 = Time$_{\text{Pressurization Step}}$
$T_2 = T_1 + w$
$T_3 = T_2$ + Time$_{\text{Adsorption Step}}$
$T_4 = T_3 + w$
$T_5 = T_4$ + Time$_{\text{Depressurization Step}}$
$T_6 = T_5 + w$
$T_7 = T_6$ + Time$_{\text{Desorption Step}}$
$T_8 = T_7 + w$

Borst (2008) refers to the length of the regularization function with the symbol w as illustrated in Fig. 1. Since the overlap domain is small enough to apply approach I to discontinuity resolution, one can easily relate w to h with the formula in (26).

$$w = 3h \tag{26}$$

There is always a physical meaning to the length (time span) of the regularizing function. In the PSA example, w refers to the amount of time it takes the valve to move from fully closed (0%) to fully open (100%) or vice versa. The valve travel speed can easily be calculated as:

$$v = \frac{100\%}{w} \tag{27}$$

From (27), we can easily deduce that $w = 0$ (a discretized model) corresponds to a valve exhibiting an infinite speed. This is unrealistic. Moreover, with a regularized model, the modeller can study the effect of valve speed on process performance by varying w and possibly optimizing process performance through manipulating w. Thus, with regularization, we are able to add one more parameter to the PSA unit optimization problem. This addition couldn't have been brought into the optimisation problem had we used a discretized model.

We ran the PSA column for one cycle using $w = 10$ s and compared the output with the discretized model. To equalize the length of the cycle for the discretized model with that of the regularized model, we appended the first two regularization periods to the adsorption step of the discretized model. The third and fourth regularization periods are appended to the desorption step of the discretized model. Fig. 15a illustrates the contribution of each step time span to the total cycle time when using conventional PSA modelling techniques. In our example, each of pressurization and depressurization steps takes about 49 s while each of adsorption and desorption steps takes about 331 s including the appended 20 s to equalize total cycle time for conventional work with that in this work. Fig. 15b adds the contribution of valve opening/closure time of 10 s to the total cycle time.

Pressure and boundary velocity evolution curves are illustrated in Fig. 16a and b for the discretized and regularized models, respectively. Inlet concentration profiles for normal pentane and normal hexane are illustrated in Fig. 16c and d for the discretized and regularized models, respectively. Pressure evolution curves are added to all Fig. 16a–d because it is a main characteristic in separating steps of a PSA cycle. The span of the regularization functions is highlighted in grey in Fig. 16b and d.

The evolution of concentration curves clearly indicates the existence of a difference between the discretized and the regularized models of the PSA column.

7. Summary and conclusions

A new approach to resolving discontinuities in dynamic simulation is presented. The method has two parts: discontinuity detection and discontinuity resolution. The approach uses *hermite* polynomials to bridge the discontinuities and it is shown that four interpolating points give smoother curvature than three while more than four give no extra benefit. The approach is shown to work in problems with many dimensions. It is generic enough to be adopted in solving any ODE/DAE system involving discontinuities in either state variables and/or their respective constitutive equations.

Discontinuity resolution completely eliminates re-initialization of state variables because it treats and bridges discontinuities at their local origin whether the origin is a state variable or a constitutive equation. Elimination of reinitialization reduces simulation run length by 23%. The reduction in simulation run-length is attributed to the localized treatment of the discontinuity at its origin instead of reinitializing entire model equations to resolve a local discontinuity. Nevertheless, this reduction is not the major achievement of work. This work achieves two other goals that were not present in previous works in this field:

1. Regularization more resembles reality than mere re-initialisation of variables because it takes into account the time and/or spacial factors between state changes. States transit through time and space from their initial to final values. Failing to take this fact into account jeopardises model accuracy. This failure is clearly evident in conventional model variables' re-initialization as we presented in PSA unit example.
2. Sticky discontinuities result from the use interpolating polynomials that do not represent the model to bridge model discontinuities as outlined earlier. Even if the integration routine manages to overcome sticky discontinuities, the generated error between the equations representing the actual model and those used by the approximating interpolating polynomial might lead to misleading simulation results. This work completely eliminates the use of integrator-based polynomials to bridge discontinuities by relying on interpolating polynomials that are derived from model equations with strict adherence to bounds that match both ends of interpolating polynomial to its adjacent discontinuous sub-functions.

To reduce computational time, it is preferable to construct interpolation mesh only once and save the mesh in computer memory or disk. However, as the number of dimensions increase, more memory/disk space is needed to save the location of the mesh interpolating points. Thus, a system programmer might be forced to compromise computational efficiency in order to accommodate a model in the available machine space by reconstructing meshes when interpolating and destroying them immediately afterwards.

Type I discontinuity resolution is the conventional attempt to resolving a discontinuity. However, it does not bring the pieces back together. It either jumps over a discontinuity through reinitialization of entire model equations or approximates the model at the discontinuity location with an interpolating polynomial that is not properly bound by model equations and is probably created at the wrong model level. This work resolves these deficiencies by bringing some of the pieces back together. The interpolating polynomial is derived from the mathematical model, properly bounded by model bounds, generated at the exact discontinuity level and localized to resolve discontinuity at its origin leaving other model equations intact. An extra step beyond the scope of this work would be to entirely eliminate the use of discretized models and regularizing functions as illustrated in the example by Abadpour and Panfilov (2009).

Appendix A. Three-D vector tracking and mesh generation equations for approach II

Although the discussion is illustrated using a 3D function, the approach is applicable to functions of any dimension.

A.I Three-D vector tracking

Let us assume that at time t_0, the 3D function f initializes at x_0, y_0 and z_0 coordinates of their respective axes in a region bounding f_1. The resulting starting point is $P_0(x_0, y_0, z_0, f(x_0, y_0, z_0))$. Since $f(x_0, y_0, z_0)$ can be calculated at any $P(x, y, z)$, we do not need to track function values. As the simulation advances by one step to t_1, the coordinates of another point $P_1(x_1, y_1, z_1)$ are identified. The locations of these two points are sufficient to determine the trajectory vector v_1 that is accurate to time t_1 only. Using linear algebra notation, vector v_1 can be written as:

$$\vec{v}_1 = \vec{P}_0 P_1 = \begin{bmatrix} x_1 - x_0 \\ y_1 - y_0 \\ z_1 - z_0 \end{bmatrix} \quad \text{(A.1)}$$

Now, let us transform the logical expression into a discontinuity plane. A plane can be uniquely identified through either:

1. a point inside the plane and a vector orthogonal to that plane,
2. or through three non-collinear points inside the plane. In this case the vector in case 1 is calculated using the three non-collinear points.

We will define the plane using the second case. To start, we need to locate arbitrary points $P_A(x_A,y_A,z_A)$, $P_B(x_B,y_B,z_B)$ and $P_C(x_C,y_C,z_C)$ located inside the discontinuity plane. We will demonstrate the procedure for the discontinuity plane cutting the x dimension. Since the plane is cutting the x dimension at $x = x_n$, the x-coordinates of the three points will take the value of x_n. The discontinuity plane is extending infinitely in all coordinates. This extension allows us to select arbitrary values for the y coordinates y_A,y_B and y_C and the z coordinates z_A, z_B and z_C. So, the coordinates of the points become:

$$\begin{aligned} &P_A(x_A, y_A, z_A) \\ &P_B(x_B, y_B, z_B) \\ &P_C(x_C, y_C, z_C) \end{aligned} \quad \text{(A.2)}$$

where $x_A = x_B = x_C = x_n$. A check for non-co-linearity needs to be performed before proceeding to the next step. If the points are identified as collinear, then another set of arbitrary values for y_A, y_B, y_C, z_A, z_B and z_C needs to be assumed and the above procedure is to be repeated. Once points pass the non-collinearity test, $\vec{v}_p$ that is orthogonal to the discontinuity plane is obtained via multiplying vectors $\vec{P}_A P_B$ with $\vec{P}_A P_C$ (or any similar combination) as vector cross product. Thus,

$$v_p = \vec{P}_A P_B x \vec{P}_A P_C = \begin{bmatrix} x_B - x_A \\ y_B - y_A \\ z_B - z_A \end{bmatrix} x \begin{bmatrix} x_C - x_A \\ y_C - y_A \\ z_C - z_A \end{bmatrix} = \begin{bmatrix} (y_B - y_A)(z_C - z_A) - (z_B - z_A)(y_C - y_A) \\ (z_B - z_A)(x_C - x_A) - (x_B - x_A)(z_C - z_A) \\ (x_B - x_A)(y_C - y_A) - (y_B - y_A)(x_C - x_A) \end{bmatrix} = \begin{bmatrix} a_{vp} \\ b_{vp} \\ c_{vp} \end{bmatrix} \quad \text{(A.3)}$$

Since the general equation of any plane passing through point $P_0(x_0, y_0, z_0)$ and orthogonal to $\vec{v}\begin{pmatrix} a \\ b \\ c \end{pmatrix}$ is:

$$a(x - x_0) + b(y - y_0) + c(z - z_0) = 0 \quad \text{(A.4)}$$

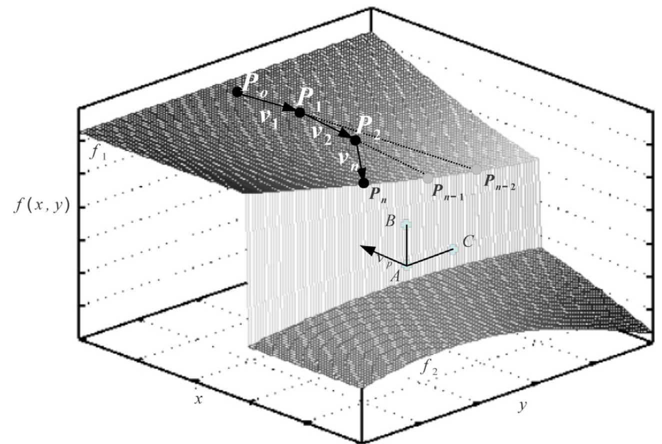


Fig. A.1. Progression of $\vec{v}_i$ towards a discontinuity plane.

we could easily formulate the equation of the discontinuity plane as one of the equations in (A.5):

$$a_{vp}(x - x_A) + b_{vp}(y - y_A) + c_{vp}(z - z_A) = 0 \quad \text{(A.5a)}$$

$$a_{vp}(x - x_B) + b_{vp}(y - y_B) + c_{vp}(z - z_B) = 0 \quad \text{(A.5b)}$$

$$a_{vp}(x - x_C) + b_{vp}(y - y_C) + c_{vp}(z - z_C) = 0 \quad \text{(A.5c)}$$

using points $P_A(x_A, y_A, z_A)$, $P_B(x_B, y_B, z_B)$ or $P_C(x_C, y_C, z_C)$ as an example.

Next, we need to find the intersection point of the line, directed by v_1 that is passing through P_0 and P_1, with the discontinuity plane defined by Eq. (A.5). To do this, we need to write the equation for this line in the form

$$x = x_0 + (x_1 - x_0)\tau \quad \text{(A.6a)}$$

$$y = y_0 + (y_1 - y_0)\tau \quad \text{(A.6b)}$$

$$z = z_0 + (z_1 - z_0)\tau \quad \text{(A.6c)}$$

Substituting (A.6) into (A.5), we get:

$$a_{vp}(x_0 + (x_1 - x_0)\tau - x_A) + b_{vp}(y_0 + (y_1 - y_0)\tau - y_A) + c_{vp}(z_0 + (z_1 - z_0)\tau - z_A) = 0 \quad \text{(A.7)}$$

Equation (A.7) has only one unknown (τ). Solving for τ and substituting the resulting value into (A.6), we obtain the intersecting point of the line P_0P_1 with the discontinuity plane. Since the vector will intersect the plane at time t_n, we will call the intersection point $P_n(x_n, y_n, z_n)$. The discussion is illustrated in Fig. A.1.

A.2 Mesh generation using approach II

Next, we need to construct the coordinates of the 64-point interpolating polynomial. To do so, we will rely on the direction of the $\vec{P}_0 P_1$ vector. The idea is to generate 4 planes that are parallel to the discontinuity dimension and separated by a distance h along the discontinuous dimension as illustrated in Fig. 9B for an intersection at z plane. Since we assumed intersection at x-plane, the planes will be separated by a distance h_x. Hence, the x dimensions

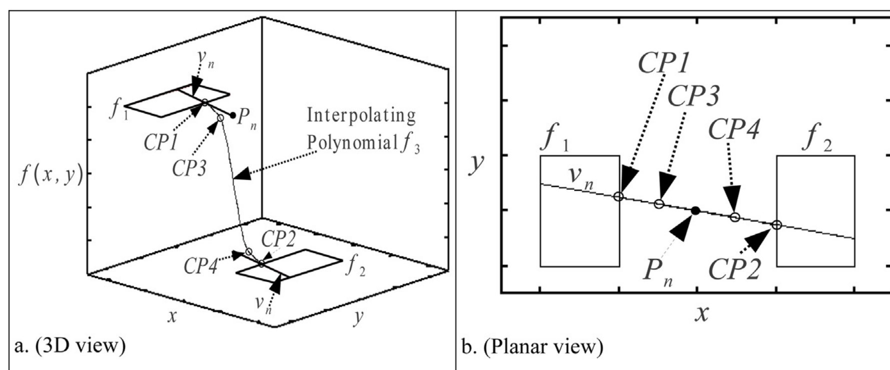


Fig. A.2. The behaviour of a 2D interpolating polynomial demonstrating the continuity of the polynomial along the continuous coordinate while interpolating along the discontinuous axis. (*CP* = Control Point).

of the four discontinuous planes become: x_n, x_n+h_x, x_n+2h_x and x_n+3h_x if $\vec{v}_n$ is entering the overlap domain from the left end. If $\vec{v}_n$ is entering the overlap domain from the right end, the x dimensions of the 4 discontinuous planes become: x_n, x_n-h_x, x_n-2h_x and x_n-3h_x. Since we are aiming for a symmetrical distribution of control points around the $\vec{v}_n$ vector, we need to calculate the coordinates of the other dimensions (y and z) for the points lying on $\vec{v}_n$ vector and having the 4x-coordinates mentioned above. To do so, we will calculate a new τ for each of the newly generated x-values:

$$\begin{aligned}
\tau_{x_n+1h_x} &= \frac{(x_n+1h_x-x_0)}{(x_1-x_0)}\ (a) \qquad \tau_{x_n-1h_x} = \frac{(x_n-1h_x-x_0)}{(x_1-x_0)}\ (a) \\
\tau_{x_n+2h_x} &= \frac{(x_n+2h_x-x_0)}{(x_1-x_0)}\ (b) \ \text{or}\ \tau_{x_n-2h_x} = \frac{(x_n-2h_x-x_0)}{(x_1-x_0)}\ (b) \\
\tau_{x_n+3h_x} &= \frac{(x_n+3h_x-x_0)}{(x_1-x_0)}\ (c) \qquad \tau_{x_n-3h_x} = \frac{(x_n-3h_x-x_0)}{(x_1-x_0)}\ (c)
\end{aligned} \tag{A.8}$$

Next we substitute the newly obtained τ values into Eq. (A.6) to get the other coordinates of the points at which $\vec{v}_n$ intersects with other planes. Last, we construct a mesh of sixteen points surrounding each of the four newly calculated points on $\vec{v}_n$. Fig. A.2 illustrates the concept when applied to 2D discontinuous functions.

References

Abadpour, A., & Panfilov, M. (2009). Method of negative saturations for modeling two-phase compositional flow with oversaturated zones. *Transport in Porous Media*, *79*, 197–214.

Archibald, R., Gelb, A., & Yoon, J. (2008). Determining the locations and discontinuities in the derivatives of functions. *Applied Numerical Mathematics*, *58*, 577–592.

Barrer, R. M., & Sutherland, J. W. (1956). *Inclusion complexes of faujasite with paraffins and permanent gases*. London: Physical Chemistry Laboratories, Imperial College.

Bartels, R. H., Beatty, J. C., & Barsky, B. A. (1987). *An introduction to splines for use in computer graphics and geometric modeling*. Morgan Kaufmann.

Borst, R. (2008). Challenges in computational materials science: Multiple scales, multi-physics and evolving discontinuities. *Computational Materials Science*, *43*, 1–15.

Bourke, Paul (2011). *Interpolation methods*, http://paulbourke.net/miscellaneous/interpolation

Brackbill, J. U., Kothe, D. B., & Zemach, C. (1992). A continuum method for modelling surface tension. *Journal of Computational Physics*, *100*, 335–354.

Breeuwsma (2011). *Cubic interpolation*, http://www.paulinternet.nl/?page=bicubic, 2011.

Carver, M. B. (1978). Efficient integration over discontinuities in ordinary differential equation simulations. *Mathematics and Computers in Simulation*, *XX*, 190–196.

Cellier, F.E, (1979). *Combined continuous/discrete system simulation by use of digital computers*. A dissertation submitted to Swiss Federal Institute of Technology Zurich for the degree of Doctor of Technical Sciences, Zurich.

Ellison, D. (1981). Efficient automatic integration of ordinary differential equations with discontinuities. *Mathematics and Computers in Simulation*, *XXIII*, 12–20.

Filip, D., Magedson, R., & Markot, R. (1986). Surface algorithms using bounds on derivatives. *Computer Aided Geometric Design*, *3*, 296–311.

Fritsch, F., & Carlson, R. (1980). Monotone piecewise cubic interpolation. *SIAM Journal on Numerical Analysis*, *17*(2), 238–246.

Gani, R., Ruiz, C. A., & Cameron, I. T. (1986). A generalized model for distillation columns—I: Model description and applications. *Computers and Chemical Engineering*, *10*(3), 181–198.

Gear, C. W. (1970). The automatic integration of ordinary differential equations. *Commun. ACM*, *14*(3), 176–190.

PSE Enterprise (2012). *gPROMS modelling language*, Copyright © 1997–2012.

GSL. (2011). GNU Scientific Library.

Helenbrook, B. T., Martnelli, L., & Law, C. K. (1999). A numerical method for solving incompressible flow problems with a surface of discontinuity. *Journal of Computational Physics*, *148*, 366–396.

Javey, S. (1988). A language construct for the specification of discontinuities. *Journal of Systems and Software*, *8*(5), 409–417.

Kochanek, D. H. U., & Bartels, R. H. (1984). *Interpolating splines with local tension, continuity and bias control. Computer graphics, SIGGRAPH-84 conference proceedings* (18), ed 3

Kreith, F. (2000). *CRC handbook of thermal engineering*. CRC Press.

Mao, G., & Petzold, L. R. (2002). Efficient integration over discontinuities for differential-algebraic systems. *Computers and Mathematics with Applications*, *43*, 65–79.

Minkkinen, A., Mank, L., & Jullian, S. (1993). Process for the isomerization of of C5/C6 normal paraffins with recycling of normal paraffins, U.S. Patent 5233120.

Park, T., & Barton, P. I. (1996). State event location in differential-algebraic models. *ACM Transactions on Modelling and Computer Simulation*, *6*(2), 137–165.

Ruiz, C. A., Cameron, I. T., & Gani, R. (1988). A generalized model for distillation columns—III: Study of startup operations. *Computers and Chemical Engineering*, *12*(1), 1–14.

Ruthven, D. M., Farooq, S., & Knaebel, K. S. (1994). *Pressure swing adsorption*. Wiley-VCH.

Silva, J. A. C., Silva, F. A. D., & Rodrigues, A. E. (2000). Separation of n/iso paraffins by PSA. *Separation and Purification Technology*, *20*, 97–110.

Yang, R. T. (1997). *Gas separation by adsorption processes*. Imperial College Press.