Sādhanā (2021)46:206
https://doi.org/10.1007/s12046-021-01724-y
© Indian Academy of Sciences

GSEIM: a general-purpose simulator with explicit and implicit methods

MAHESH B PATIL*, RUCHITA D KORGAONKAR and KUMAR APPAIAH

Department of Electrical Engineering, Indian Institute of Technology, Bombay, Mumbai, India
e-mail: mbpatil@ee.iitb.ac.in

MS received 15 April 2021; revised 2 July 2021; accepted 7 August 2021

Abstract. A new simulation package, GSEIM (General-purpose Simulator with Explicit and Implicit Methods), for solving a set of ordinary dierential equations (ODEs) is presented. A novel feature of GSEIM is the provision for solving a set of ODEs using explicit or implicit schemes. The organisation of the program is illustrated with the help of a block diagram. Various features of GSEIM are discussed. Two ways of incorporating new elements in GSEIM, viz., as a template and as a subcircuit, are explained by taking a specific example. The flexibility provided to the user to incorporate new elements together with the open-source nature of GSEIM is expected to make it a viable alternative for simulation of practical systems involving ODEs. Simulation examples are described, which validate GSEIM and bring out its capabilities.

Keywords. Numerical solution; ordinary differential equations; user-defined elements.

1. Introduction

A wide variety of engineering applications require numerical solution of a set of ordinary differential equations (ODEs), satisfying some given initial conditions. This need is currently addressed by commercial [1, 2] as well as open-source [3] software packages. Although the numerical methods for solving ODEs are well known (see, e.g., [4–9]), different packages have different strengths and weaknesses, based on their performance, ease of use, capability of adding new library elements, cost, user support, and legacy issues. The purpose of this paper is to present a new ODE solver called GSEIM (General-purpose Simulator with Explicit and Implicit Methods) and illustrate its working process through examples. The open-source nature of GSEIM [10], the flexibility offered to the user for incorporating new elements, and the possibility of using explicit or explicit methods are expected to make GSEIM an attractive alternative for various applications.

The paper is organised as follows. In Section 2, we briefly review the advantages and limitations of explicit and implicit methods for solving ODEs. We then describe, in Section 3, the block-level organisation of GSEIM. A key feature of GSEIM is the flexibility with which the user can add new elements to the library. We describe this aspect in Section 4 where we point out, using a few examples, how computations related to explicit and implicit methods are incorporated in the element templates. In Section 5, we look at how a subcircuit (hierarchical block) can be added to GSEIM, using the example of an induction machine model. One important requirement in many engineering applications is accurate handling of abrupt changes. We describe in Section 6 how that is implemented in GSEIM. In Section 7, we present two simulation examples to illustrate the capabilities of the new platform. Finally, in Section 8, we present our conclusions and comments on future directions.

2. Explicit and implicit methods

There are several well-known explicit and implicit methods for solving ODEs (see, e.g., [5]). In order to illustrate the advantage of an explicit method over an implicit method, let us consider a single ODE of the form

$$\frac{dx}{dt} = f(t, x) \tag{1}$$

The discretised form of this ODE using the improved Euler method (an explicit method) is given by

$$x_{n+1} = x_n + \frac{h}{2}\left[f(t_n, x_n) + f(t_n + h, x_n + hf(t_n, x_n))\right] \tag{2}$$

where x_n and x_{n+1} correspond to the numerical solutions at times t_n and t_{n+1}, respectively, and $h = t_{n+1} - t_n$ is the time step. Since t_n and x_n are known, computing x_{n+1} involves only *evaluation* in this case.

Consider now the discretised form of equation (1) when the backward Euler method (an implicit method) is used:

*For correspondence
Published online: 09 October 2021

$$x_{n+1} = x_n + hf(t_{n+1}, x_{n+1}) \tag{3}$$

Because x_{n+1} is also involved on the right-hand side, obtaining x_{n+1} in this case requires the *solution* of equation (3).

For a system of ODEs, an explicit method would still involve function evaluations, i.e., the process of updating the system variables by evaluating functions of their past values. An implicit method, on the other hand, would give rise to a system of equations that has to be *solved*. If the system of equations is nonlinear, an iterative procedure such as the Newton–Raphson method would be required with the associated complication of convergence difficulties. Clearly, from the perspective of work per time step, an explicit method would be advantageous over an implicit method of the same order. However, implicit methods are superior in terms of stability, as illustrated by the following example.

Consider an *RC* circuit, as shown in figure 1. We are interested in the variation of V_1 and V_2 when a step input voltage $V_s(t)$ is applied.

For solving the circuit equations numerically, we first rewrite them as a system of ODEs:

$$\begin{aligned} \frac{dV_1}{dt} &= \frac{1}{R_1C_1}(V_s - V_1) - \frac{1}{R_2C_1}(V_1 - V_2) \\ \frac{dV_2}{dt} &= \frac{1}{R_2C_2}(V_1 - V_2) \end{aligned} \tag{4}$$

We expect V_1 and V_2 to start changing as the input step is applied and eventually settle down to their steady-state values. For this problem, rather than using constant time steps for the entire interval of interest, it is far more efficient to use small time steps when the variations are rapid and large time steps when they are slow. We consider two methods which employ such adaptive time step computation: (a) the Runge–Kutta–Fehlberg (RKF45) method, an explicit method which employs Runge–Kutta methods of order 4 and 5 in each time step [8], and (b) the trapezoidal–backward difference formula (TR–BDF2) method, an implicit method which employs the TR and BDF2 methods in each time step [11].

In each of these methods, an estimate of the local truncation error (LTE) is obtained in each time step. If the LTE is small, the current time step is accepted, and the next time step is allowed to be larger. If the LTE is larger than a specified value, the current time step is rejected and a smaller time step is tried. As the circuit approaches steady state, the LTE tends to zero, allowing the algorithm to take larger time steps, limited eventually only by an upper limit set by the user.

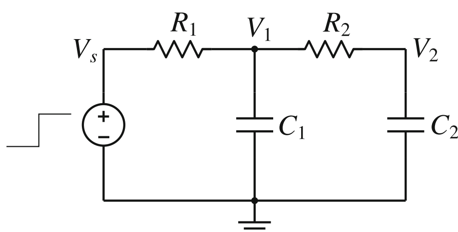

Figure 1. *RC* circuit with two time constants.

Figure 2 shows the results for $R_1 = R_2 = 1$ kΩ and $C_1 = C_2 = 1$ μF. Both RKF45 and TR–BDF2 methods perform as expected. As the circuit approaches steady state, they make the time steps larger, leading to fewer time steps overall and therefore a faster simulation.

When C_2 is changed from 1 to 0.1 μF, the two methods show very different behaviour (see figure 3). The TR–BDF2 method continues to increase the time step as the circuit approaches the steady state. The RKF45 method does increase the time step up to a certain point, but at $t \approx 5.1$ ms, it forces a small time step. After that, it once again starts increasing the time step, but only up to $t \approx 10.8$ ms, and so on. This behaviour is related to the stability of the RKF45 algorithm. The explicit Runge–Kutta methods employed in the RKF45 algorithm are conditionally stable. They require the time step to be smaller than a certain multiple of the smallest time constant in the system [9]. With $R_1 = R_2 = 1$ kΩ, $C_1 = C_2 = 1$ μF, the time constants are $\tau_1 = 2.6$ ms and $\tau_2 = 0.38$ ms. With $C_2 = 0.1$ μF, the time constants are $\tau_1 = 1.1$ ms and $\tau_2 = 0.09$ ms, and the largest time step allowed by the RKF45 algorithm is correspondingly reduced. For the TR–BDF2 method, which is A-stable, there is no such restriction, and therefore it allows large time steps as steady state is approached, thus reducing the computation time significantly.

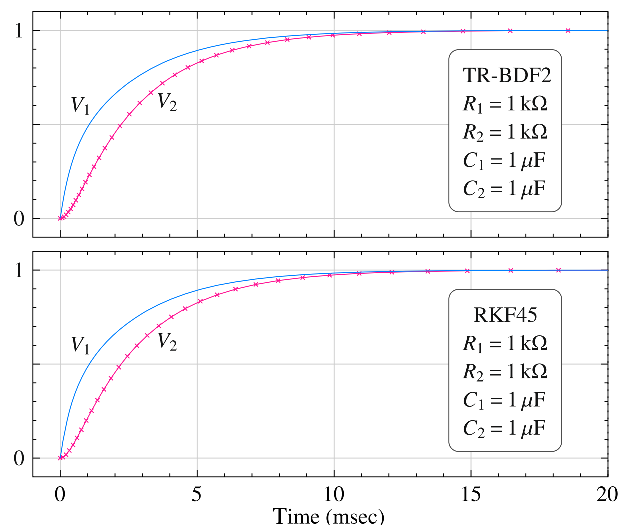

Figure 2. Numerical solution of equation (4) using the TR–BDF2 and RKF45 methods. The parameter values are $R_1 = R_2 = 1$ kΩ and $C_1 = C_2 = 1$ μF. Crosses show the simulator time points.

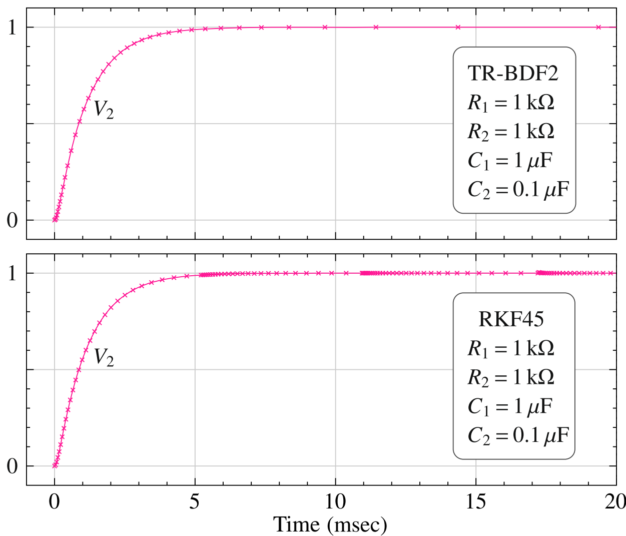

Figure 3. Numerical solution of equation (4) using the TR–BDF2 and RKF45 methods. The parameter values are $R_1 = R_2 = 1$ kΩ, $C_1 = 1$ μF, and $C_2 = 0.1$ μF. Crosses show the simulator time points.

From the above discussion, it is clear that, from the efficiency perspective, the choice of the method (explicit or implicit) would depend on the problem being solved. For this reason, GSEIM incorporates explicit as well as implicit methods. The implementation details can be found in [10].

3. GSEIM organisation

The block diagram of the GSEIM program is shown in figure 4. The schematic entry graphical user interface (GUI) block is adapted from the GNURadio package [12]. It enables the user to prepare a schematic diagram of the system of interest and produces a high-level netlist. A parser program takes the high-level netlist as an input, performs parsing of node names and computation of element parameters (if required). As its output, the parser program produces a low-level netlist. The solver takes the low-level netlist as its input, and using information from the library, it prepares and solves the set of ODEs corresponding to the user's system, creating output files requested by the user. Finally, the plotting program reads the output files and displays the plots in an interactive manner.

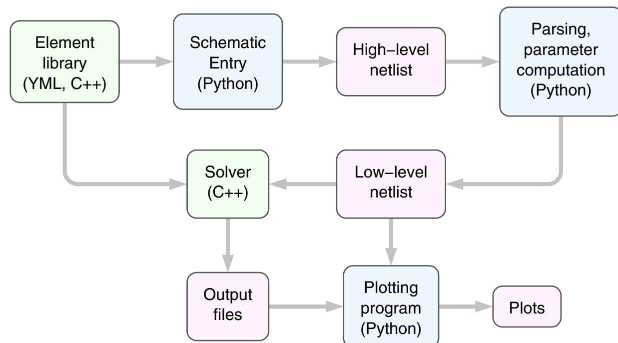

Figure 4. Block diagram of the GSEIM program.

GSEIM has been designed to completely decouple the element library from the solver, which makes it possible for the user to add new elements to the library (if required). For each element (say, `xyz`), the library contains two files: (a) `xyz.xbe` which contains information about the variable names, parameter names and values, and equations related to that element and (b) `xyz.yml` which specifies how the element would appear in the schematic entry GUI. A detailed description of these files would be presented in the GSEIM manual. In Section 4, we will take a brief look at the `xbe` files for a few elements.

The salient features of the GSEIM package can be described as follows:

(a) The solver, which handles the most intensive computation, viz., numerical solution of the ODEs, is written in C++ because of its high performance.
(b) For all other purposes, viz., schematic capture, parsing, parameter computation, and plotting, python is used because of the flexibility and ease of programming it offers.
(c) Output parameters, which determine what data gets stored in the output files during simulation, are specified without having to add extra elements—such as the 'scope' in Simulink [1] and Xcos [3]—to the schematic diagram. This helps in avoiding clutter.
(d) Subcircuits (hierarchical blocks) can be used for simplifying the schematic.
(e) Explicit as well as implicit numerical methods are incorporated in the solver. Currently, the following methods are made available:

    (i) explicit (fixed time step): improved Euler, Heun, and Runge–Kutta (4th order);
    (ii) explicit (auto time step): RKF45, Bogacki, and Shampine (2,3);
    (iii) implicit (fixed time step): backward Euler and TR;
    (iv) implicit (auto time step): backward Euler-auto, TR-auto, and TR–BDF2.

    The backward Euler-auto and TR-auto methods are used only for nonlinear problems. In these methods, the time step is adjusted depending on the number of Newton–Raphson iterations required at a given time point.

(f) GSEIM provides a GUI for plotting the variables specified by the user. The plotting GUI, shown in figure 5, allows the user to select the output file of interest, the x-axis variable (typically time), and the y-

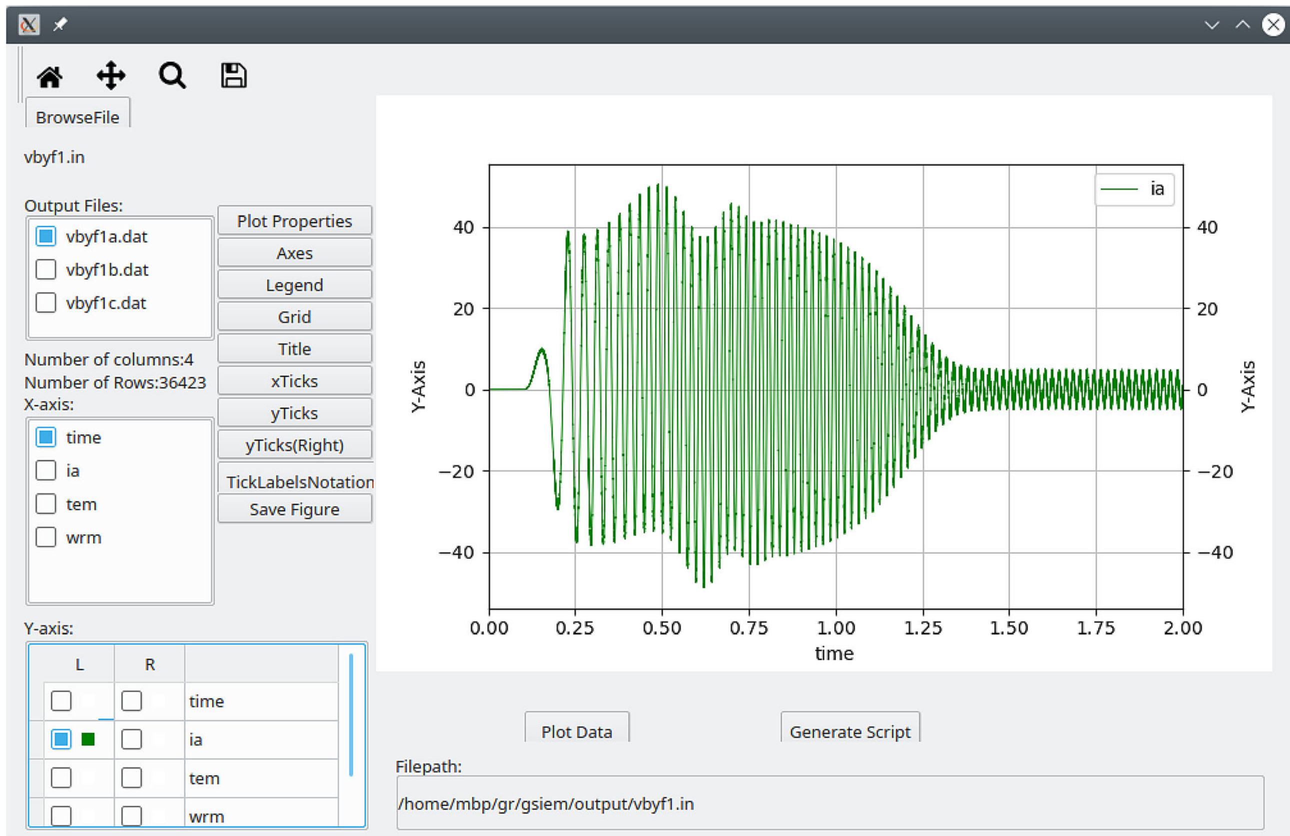

Figure 5. Snapshot of the plotting GUI provided as a part of the GSEIM package.

axis variable(s) to be included in the plot. It also gives the user control over plot attributes such as line colour, line width, and symbol type, with appropriate values specified by default. Using this information, the plotting GUI displays the plot requested by the user. It also produces python code associated with the plot. If required, the user can edit this code in order to make the plot more suitable for a report or a presentation. This way, the user can benefit from a wide range of plotting capabilities offered by python.

4. Library element templates

A very important feature of GSEIM is that it allows the user to add new functionality in the form of library elements, by writing a suitable 'template'. In this section, we look at the syntax of element templates with the help of some examples. We start with a few remarks.

(a) An element template has three types of variables in general: input, output, and auxiliary. Only the input and output variables are made available in the schematic capture GUI for connection to other elements.

(b) Two types of elements are allowed:

(i) `evaluate` type in which the element equations are of the form $y = f(x_1, x_2, \ldots)$, where y is an output and x_1, x_2, etc. are inputs. These elements do not involve time derivatives;

(ii) `integrate` type with equations of the form $\frac{dy}{dt} = f(x_1, x_2, \ldots)$, where y is an output or auxiliary variable, and x_1, x_2, etc. can be input, output, or auxiliary variables.

4.1 *Adder*

As our first example, we consider the `sum_2` element which gives $y = k_1 x_1 + k_2 x_2$, where x_1 and x_2 are input variables, y is the output variable, and k_1 and k_2 are real parameters. This is an `evaluate` type element, i.e., its output can be written as a function of its input, and it does not involve time derivatives. Figure 6 shows the overall structure of `sum_2.xbe`.

The following features may be noted:

(a) The element name is specified by the keyword `name`.

```
xbe name=sum_2 evaluate=yes
Jacobian: constant
input_vars: x1 x2
output_vars: y
rparms: k1=1  k2=1
outparms: x1 x2 y
n_f=0
n_g=1
g_1: x1 x2 y
C:
 ...
endC
endxbe
```

Figure 6. sum_2.xbe template (partial).

(b) The assignment evaluate=yes specifies the element type.
(c) The assignment Jacobian: constant indicates that when the element equation $y - k_1x_1 - k_2x_2 = 0$ is differentiated with respect to the variables involved in the equation, we get constants.
(d) The lines input_vars and output_vars specify the input and output variables of the element, respectively.
(e) The names and default values of the real parameters are given by the rparms statement. (GSEIM also allows integer and string parameters; they are not used in sum_2.)
(f) The outparms statement specifies the names of output parameters which will be made available by this template for saving to the user's output files during simulation (if requested by the user).
(g) The n_f and n_g statements specify the number of f and g functions for this element. (This aspect is described in detail in the GSEIM manual [10].)
(h) The g_1 statement indicates the variables involved in the function g_1.
(i) The C++ part of the template, to be described separately, appears between the C and endC statements.

Before we look at the C++ part of sum_2.xbe, let us see where it fits in the overall scheme. The GSEIM library preprocessor embeds the C++ part of each element template in the C++ function corresponding to that element. This function receives objects X and G from the GSEIM main program and is expected to compute various quantities such as function values, output parameters, etc. The object G is a global object and is used to pass information about the current time point, type of method being used (implicit or explicit), etc. It also conveys to the element routine, through the flags array, what computation the main program is expecting from the element routine in the present call. The object X is specific to the element being treated, and it contains variables and parameter values related to that element. With this background, we can make the following points about the C++ part of sum_2.xbe, as shown in figure 7:

(a) If an explicit method is being used, the template only needs to evaluate y in terms of x1 and x2.
(b) If an implicit method is being used, the template needs to supply information about the equation it satisfies, which in this case is

$$g_1 \equiv y - k_1x_1 - k_2x_2 = 0 \tag{5}$$

If the program is requesting the function value, $g_1(x_1, x_2, y)$ is evaluated; if it is requesting the derivatives, then $\frac{\partial g_1}{\partial x_1}, \frac{\partial g_1}{\partial x_2}, \frac{\partial g_1}{\partial y}$ are evaluated.
(c) If the program is requesting assignment of output parameters, the parameters listed in the outparms statement (see figure 6) are assigned.

4.2 *Integrator*

Next, we consider an element of type integrate, viz., the integrator, which satisfies $y = k \int x\,dt$, where x and y are the input and output variables, respectively, and k is a real parameter. Since GSEIM expects the equations to be written in the general form $\frac{dy}{dt} = f(x_1, x_2, \ldots)$, we rewrite the

```
variables:
source:
   k1 = X.rprm[nr_k1];
   k2 = X.rprm[nr_k2];

   x1 = X.val_vr[nvr_x1];
   x2 = X.val_vr[nvr_x2];

   if (G.flags[G.i_trns]) {
     if (G.flags[G.i_explicit]) {
       X.val_vr[nvr_y] = k1*x1 + k2*x2;
     } else if (G.flags[G.i_implicit]) {
       if (G.flags[G.i_function]) {
         y = X.val_vr[nvr_y];
         X.g[ng_1] = y-k1*x1-k2*x2;
       }
       if (G.flags[G.i_jacobian]) {
         J.dgdvr[ng_1][nvr_y ] =  1.0;
         J.dgdvr[ng_1][nvr_x1] = -k1;
         J.dgdvr[ng_1][nvr_x2] = -k2;
       }
     }
   }
   if (G.flags[G.i_outvar]) {
     X.outprm[no_x1] = X.val_vr[nvr_x1];
     X.outprm[no_x2] = X.val_vr[nvr_x2];
     X.outprm[no_y ] = X.val_vr[nvr_y ];
   }
```

Figure 7. C++ part of the sum_2.xbe template (partial).

integrator equation as $\frac{dy}{dt} = kx$. For `integrate` type elements, we also need to specify the initial or 'start-up' value of the state variable(s). For the integrator, we will denote that by y_0.

The integrator template without the C++ part is shown in figure 8. The C++ part is shown separately in figure 9. The start-up parameter `y_st` corresponds to y_0 mentioned above. The fact that time derivative of `y` is involved in the element equation is indicated by the `f_1` statement.

In the C++ part of the template (figure 9), we have different sections for start-up and transient simulation. In the start-up section, the equation $y = y_0$ is handled. In the transient section, if the method is explicit, only the function $f_1 = kx$ is evaluated; if it is implicit, the function $g_1 = kx$ as well as its derivative $\frac{\partial g_1}{\partial x}$ are computed.

4.3 *Induction motor*

We now look at a more complex element of type `integrate`, viz., `indmc1.xbe`, which implements the induction machine model given by:

$$\frac{d\psi_{ds}}{dt} = v_{ds} - r_s i_{ds} \tag{6}$$

$$\frac{d\psi_{qs}}{dt} = v_{qs} - r_s i_{qs} \tag{7}$$

$$\frac{d\psi_{dr}}{dt} = -\frac{P}{2}\,\omega_{rm}\psi_{qr} - r_r i_{dr} \tag{8}$$

$$\frac{d\psi_{qr}}{dt} = \frac{P}{2}\,\omega_{rm}\psi_{dr} - r_r i_{qr} \tag{9}$$

$$\frac{d\omega_{rm}}{dt} = \frac{1}{J}\left(T_{em} - T_L\right) \tag{10}$$

where

```
xbe name=integrator integrate=yes
Jacobian: constant
input_vars: x
output_vars: y
rparms: k=1
stparms: y_st=0
outparms: x y
n_f=1
f_1: d_dt(y)
n_g=1
g_1: x
C:
 ...
endC
endxbe
```

Figure 8. `integrator.xbe` template (partial).

```
if (G.flags[G.i_startup]) {
  y_st = X.stprm[nst_y_st];
  if (G.flags[G.i_explicit]) {
    X.val_vr[nvr_y] = y_st;
  } else if (G.flags[G.i_implicit]) {
    X.h[nf_1] = X.val_vr[nvr_y]-y_st;
  }
}
k = X.rprm[nr_k];
x = X.val_vr[nvr_x];

if (G.flags[G.i_trns]) {
  if (G.flags[G.i_explicit]) {
    X.f[nf_1] = k*x;
  } else if (G.flags[G.i_implicit]) {
    if (G.flags[G.i_function]) {
      X.g[ng_1] = k*x;
    }
    if (G.flags[G.i_jacobian]) {
      J.dgdvr[ng_1][nvr_x] = k;
    }
  }
}
```

Figure 9. C++ part of the `integrator.xbe` template (partial).

```
xbe name=indmc1 integrate=yes
Jacobian: variable
input_vars: vqs vds tl
output_vars: wrm
aux_vars: psids psidr psiqs psiqr
iparms: poles=4
sparms:
rparms:
+  rs=0.435  lls=0.002  lm=0.0693  llr=0.002
+  rr=0.816  j=0.089  ls=0  lr=0  le=0
+  l1=0  l2=0  l3=0  x1=0  x2=0
stparms: psids0=0 psiqs0=0 psidr0=0 psiqr0=0
+  wrm0=0
outparms: wrm tem vds vqs ia ib ic
n_f=5
f_1: d_dt(psids)
f_2: d_dt(psiqs)
f_3: d_dt(psidr)
f_4: d_dt(psiqr)
f_5: d_dt(wrm)
n_g=5
g_1: vds psids psidr
g_2: vqs psiqs psiqr
g_3: wrm psiqr psids psidr
g_4: wrm psidr psiqs psiqr
g_5: tl psids psidr psiqs psiqr
C:
 ...
endC
endxbe
```

Figure 10. `indmc1.xbe` template (partial).

```
if (G.flags[G.i_one_time_parms]) {
  lls = X.rprm[nr_lls];
  lm  = X.rprm[nr_lm ];
  llr = X.rprm[nr_llr];

  ls = lls + lm;
  lr = llr + lm;
  le = (ls*lr/lm) - lm;
  l1 = lr/(lm*le);
  l2 = 1.0 + (lls/lm);
  l3 = lls/lm;

  poles = X.iprm[ni_poles];
  p = (double)(poles);
  x1 = 0.75*p*lm;
  x2 = 0.5*p;

  X.rprm[nr_ls] = ls;
  ...
}
```

Figure 11. One-time parameter section of `indmc1.xbe`.

```
if (G.flags[G.i_explicit]) {
  rs  = X.rprm[nr_rs ];
  lls = X.rprm[nr_lls];
  ...
  l1  = X.rprm[nr_l1 ];
  ...

  vqs = X.val_vr[nvr_vqs];
  vds = X.val_vr[nvr_vds];
  wrm = X.val_vr[nvr_wrm];
  tl  = X.val_vr[nvr_tl ];

  psids = X.val_aux[na_psids];
  psidr = X.val_aux[na_psidr];
  psiqs = X.val_aux[na_psiqs];
  psiqr = X.val_aux[na_psiqr];

  ids = l1*psids - psidr/le;
  iqs = l1*psiqs - psiqr/le;

  idr = psids/lm - l2*ids;
  iqr = psiqs/lm - l2*iqs;

  tem0 = x1*(iqs*idr-ids*iqr);
  wr   = x2*wrm;

  X.f[nf_1] = vds-rs*ids;
  X.f[nf_2] = vqs-rs*iqs;
  X.f[nf_3] = -wr*psiqr-rr*idr;
  X.f[nf_4] =  wr*psidr-rr*iqr;
  X.f[nf_5] = (tem0-tl)/j;
}
```

Figure 12. Function evaluation in `indmc1.xbe` for explicit methods.

```
if (G.flags[G.i_implicit]) {
  // parameter/variable assignment not shown
  ids = (l1*psids) - (psidr/le);
  iqs = (l1*psiqs) - (psiqr/le);
  idr = (psids/lm) - (l2*ids);
  iqr = (psiqs/lm) - (l2*iqs);
  tem0 = x1*(iqs*idr-ids*iqr);
  wr = x2*wrm;

  if (G.flags[G.i_function]) {
    X.g[ng_1] = vds-rs*ids;
    X.g[ng_2] = vqs-rs*iqs;
    X.g[ng_3] = (-wr)*psiqr-rr*idr;
    X.g[ng_4] = ( wr)*psidr-rr*iqr;
    X.g[ng_5] = (tem0-tl)/j;
  }
  if (G.flags[G.i_jacobian]) {
    ids_psids = l1;
    ids_psidr = -1.0/le;
    ...
    J.dgdvr[ng_1][nvr_vds] = 1.0;
    J.dgdaux[ng_1][na_psids] = -rs*ids_psids;
    J.dgdaux[ng_1][na_psidr] = -rs*ids_psidr;
    ...
  }
}
```

Figure 13. Function evaluation in `indmc1.xbe` for implicit methods.

$$i_{ds} = \frac{L_r}{L_m L_e}\,\psi_{ds} - \frac{1}{L_e}\,\psi_{dr} \tag{11}$$

$$i_{qs} = \frac{L_r}{L_m L_e}\,\psi_{qs} - \frac{1}{L_e}\,\psi_{qr} \tag{12}$$

$$i_{dr} = \frac{1}{L_m}\,\psi_{ds} - \left(\frac{L_{ls}}{L_m} + 1\right) i_{ds} \tag{13}$$

$$i_{qr} = \frac{1}{L_m}\,\psi_{qs} - \left(\frac{L_{ls}}{L_m} + 1\right) i_{qs} \tag{14}$$

$$T_{em} = \frac{3}{4}\,P L_m\,(i_{qs} i_{dr} - i_{ds} i_{qr}) \tag{15}$$

with $L_e = \frac{L_s L_r}{L_m} - L_m, L_s = L_{ls} + L_m$, and $L_r = L_{lr} + L_m$.

Figures 10–13 show the various sections of the `indmc1` template. The input variables are v_{qs}, v_{ds}, T_L, and the output variable is ω_{rm}. In addition, it has internal (auxiliary) variables $\psi_{ds}, \psi_{dr}, \psi_{qs}, \psi_{qr}$ which are involved in the model equations. The statements `f_1`, `f_2`, etc. in figure 10 are used to inform the simulator which derivatives are involved in the given equation. The statements `g_1`, `g_2`, etc. are used to indicate which variables are involved in the right-hand side of the corresponding equation.

In the induction machine equations (equations (6)–(15)), there are some 'one-time' calculations, e.g., calculation of L_e, which are not required to be performed in every time step. GSEIM provides a flag for this purpose, as seen in

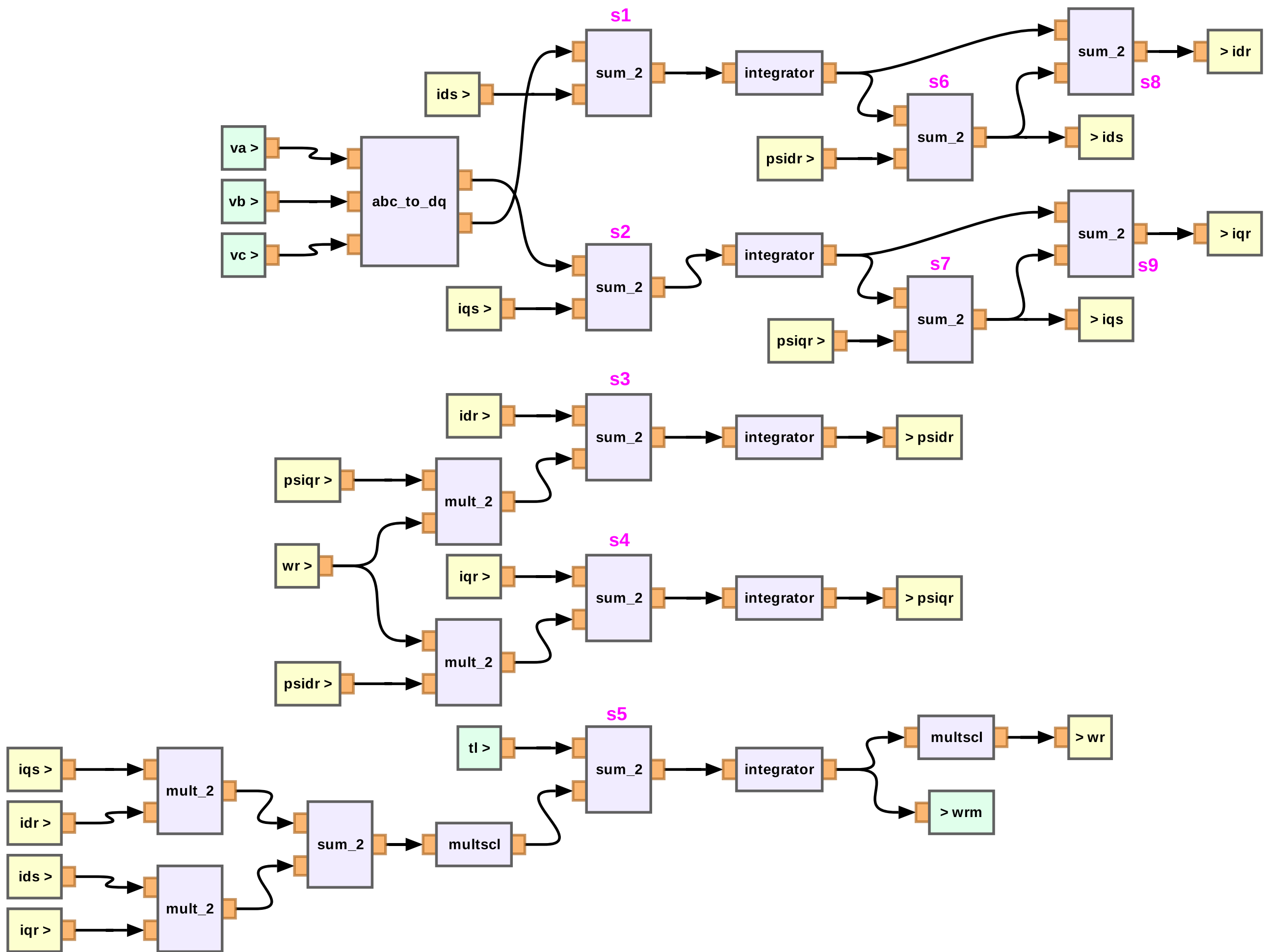

Figure 14. Subcircuit for the induction machine model given by equations (6)–(15).

```
def s_indmc_parm(dict1):
    j = float(dict1['j'])
    llr = float(dict1['llr'])
    ...
    ls = lls + lm
    lr = llr + lm
    le = (ls*lr/lm) - lm
    l1 = lr/(lm*le)
    l2 = 1.0 + (lls/lm)
    l3 = lls/lm
    x1 = 0.75*poles*lm
    x2 = 0.5*poles
    ...
    s6_k1 = l1
    s6_k2 = -1.0/le
    ...
    dict1['s6_k1'] = '%14.7e' % (s6_k1)
    dict1['s6_k2'] = '%14.7e' % (s6_k2)
```

Figure 15. Parameter computation for the induction machine subcircuit.

figure 11. When this flag is set by the main program, the template computes L_e and other one-time parameters, and saves them in the `X.rprm` vector. Subsequently, these parameters need not be computed again.

The function assignment sections of `indmc1.xbe` are shown separately in figures 12 and 13 for explicit and implicit methods, respectively. In the explicit case (figure 12), the function f_1 (i.e., `f[nf_1]`) is computed as per the right-hand side of equation (6), and so on. In the implicit case (figure 13), the function g_1 is similarly computed. However, in this case, the derivatives of g_1 with respect to each of the variables involved in this equation also need to be computed.

As seen from the above example, writing a new template, particularly the Jacobian assignment part, requires some systematic effort. For this reason, some simulation packages allow the use of hardware description languages such as Openmodelica [13] and Verilog-AMS [14], which require from the user only functions in symbolic form. The derivatives are then computed internally by the simulator.

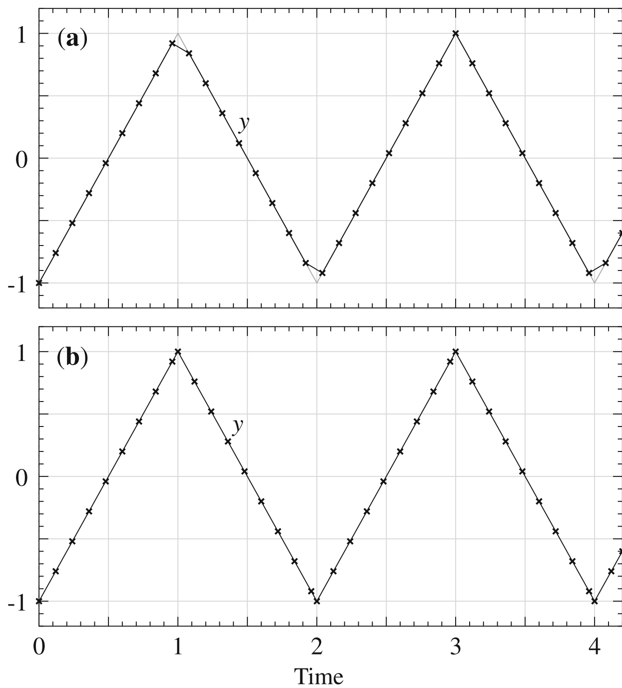

Figure 16. Triangle source waveform: (**a**) with a constant time step and (**b**) with time step adjusted for tracking abrupt changes. Crosses show the simulator time points.

While the 'low-level' approach used in GSEIM demands more effort from the user in developing new element templates, it does offer more flexibility—the user is not constrained by the limitations of a high-level language. Furthermore, library development is a one-time activity; once an element template is developed and tested, no further coding is required. We believe therefore that our low-level approach has significant practical relevance.

5. Subcircuits

In many situations, the system of interest is hierarchical in nature, and building it in a modular fashion is easier or more convenient than assembling all the basic blocks at one level. Like simulation packages such as SPICE [15], Simulink [1], Dymola [2], GSEIM also allows hierarchical system building. In this section, we consider the induction machine model of Section 4 and describe how it can be implemented as a 'subcircuit' (a hierarchical block) rather than writing an element template. For this purpose, we rewrite equations (6)–(15) such that each of them can be implemented using basic blocks such as adder, multiplier, integrator, etc.

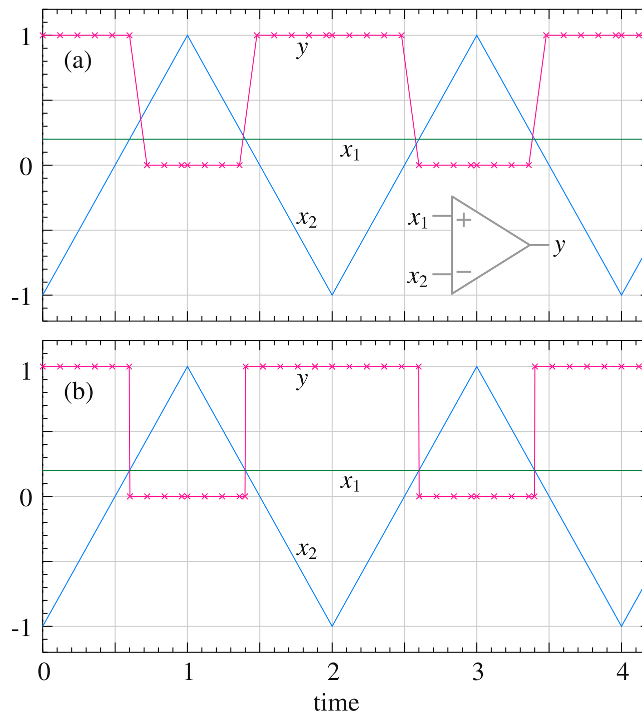

Figure 17. Comparator input and output waveforms: (**a**) with a constant time step and (**b**) with additional time points obtained by extrapolation. Crosses show the simulator time points.

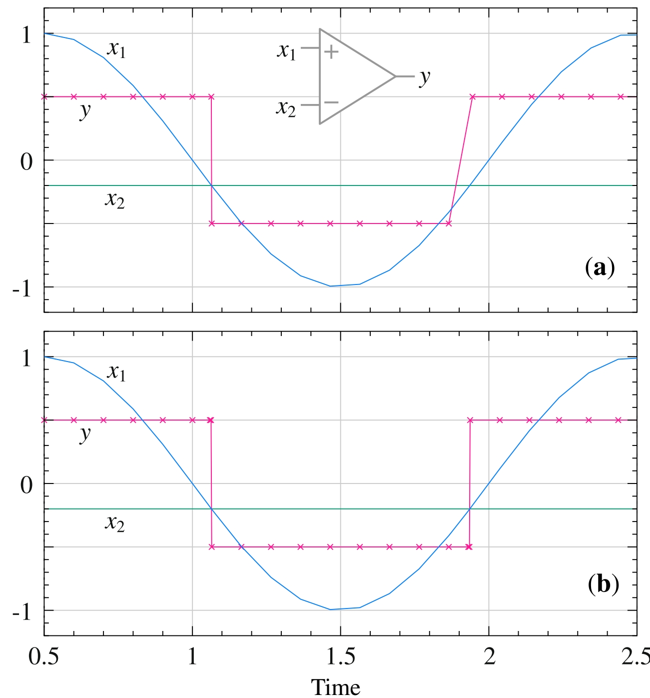

Figure 18. Comparator input and output waveforms: (**a**) with linear extrapolation and (**b**) with quadratic extrapolation.

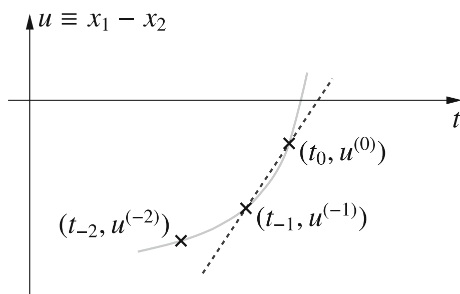

Figure 19. Example of failure of the linear extrapolation technique for treating crossover events.

As an example, equation (8) can be rewritten as

$$\psi_{dr} = \int \left(-\frac{P}{2}\,\omega_{rm}\psi_{qr} - r_r i_{dr} \right) dt \qquad (16)$$

which can be implemented using a multiplier (to multiply ω_{rm} and ψ_{qr}), and the `sum_2` and `integrator` elements described in Section 4. Treating equations (6)–(10) in this manner, we obtain the subcircuit that is shown in figure 14.

The following additional points about the implementation may be noted:

(a) 'Virtual' sources and sinks (shown in light yellow colour) are used in order to make wiring less cumbersome. For example, note the virtual sink marked '>idr' and the virtual source marked 'idr>', the two corresponding to the same node.

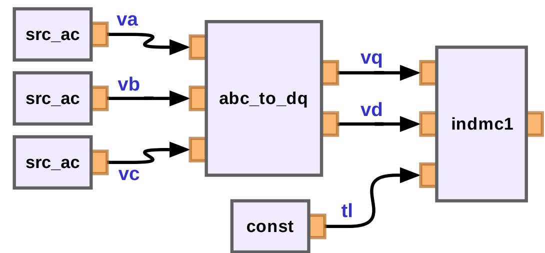

Figure 20. GSEIM schematic diagram for simulation of free acceleration of induction motor using `indmc1.xbe`.

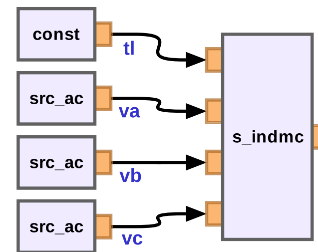

Figure 21. GSEIM schematic diagram for simulation of free acceleration of induction motor using induction machine subcircuit.

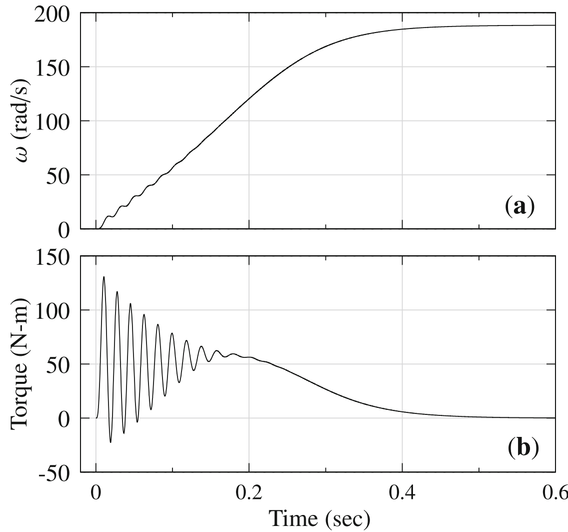

Figure 22. Simulation results for free acceleration of induction motor: (**a**) speed and (**b**) torque.

(b) Input and output pads (shown in light green colour) are used to indicate the input and output ports the subcircuit symbol will have when it is invoked (from a higher level). For the induction machine subcircuit, `va`, `vb`, `vc`, `tl` are the input ports, and `wrm` is the output port.

(c) The subcircuit has the following parameters (not shown in figure 14): `j`, `llr`, `lls`, `lm`, `poles`, `rr`, `rs`, which correspond to $J, L_{lr}, L_{ls}, L_m, P, r_r, r_s$, respectively, in equations (6)–(10). In implementing the equations, we need to compute quantities that depend on these parameters. For example, consider equation (11) for i_{ds}, implemented using the `sum_2` element marked as `s6` in the figure. This element gives $i_{ds} = k_1\psi_{ds} + k_2\psi_{dr}$, which requires $k_1 = \frac{L_r}{L_m L_e}$ and $k_2 = -\frac{1}{L_e}$ to be assigned. For all such assignments, the user is expected to supply a python function that is specific to the concerned subcircuit. For the induction machine subcircuit (`s_indmc`), the python block is shown in figure 15. The calculations for k_1 and k_2 of `s6` are shown specifically in the figure. Similarly, several other quantities are computed and stored in the python dictionary received by the `s_indmc_parm` function as an argument. With this mechanism, the user has significant flexibility in implementing element equations.

(d) The user can define 'output parameters' for a subcircuit and use those at higher levels, as described in [10]. The output parameters can be mapped to nodes within the subcircuit or to the output parameters of the blocks involved in the subcircuit. These features provide a mechanism for viewing various quantities of interest at different levels when the system is simulated.

6. Handling abrupt changes

In many systems of practical interest, some of the variables are expected to vary abruptly. If the abrupt transitions are missed out by the simulator, it would affect the appearance of the plots of those variables, and more importantly, the accuracy of the simulation results in some cases. As an example, consider a triangle source with period $T = 2$ s. If a constant time step $\Delta t = 0.12$ s is used, some of the peak or valley points are missed out by the simulator, as shown in figure 16(a). This situation can be improved by tracking the time points where the source output is going to reach a peak or a valley. For this purpose, the triangle source template in GSEIM takes the current time point from the simulator and returns the time of the next 'break' (peak or valley). Using this information, GSEIM decides whether the normal time step or a reduced time step should be used

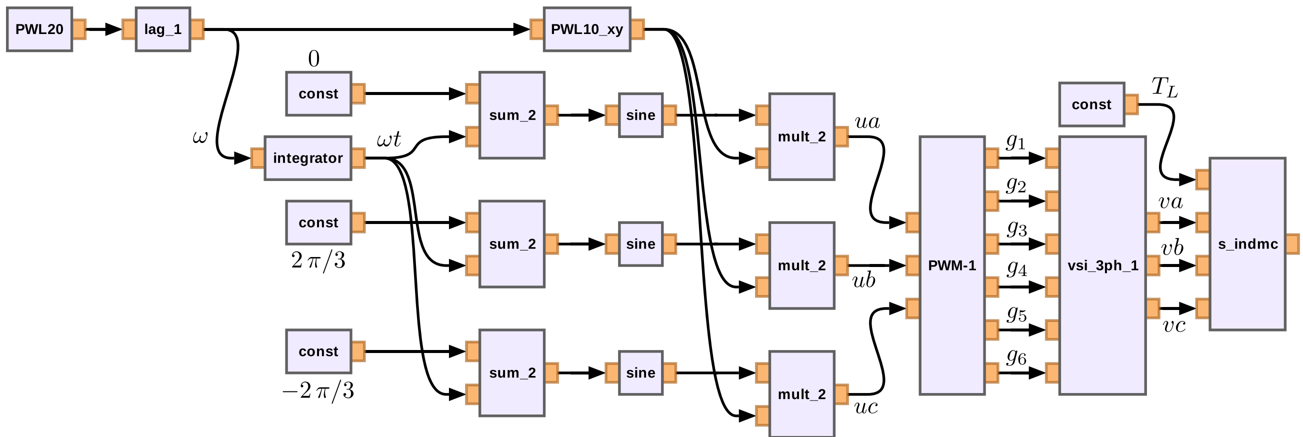

Figure 23. GSEIM schematic diagram for *V*/*f* control of an induction motor.

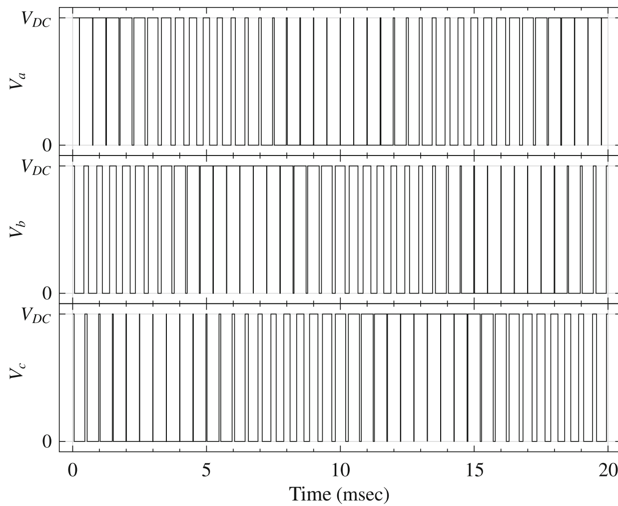

Figure 24. Voltages V_a, V_b, V_c in the steady state for the system of figure 23.

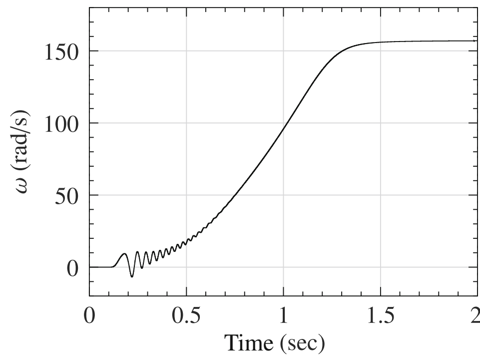

Figure 25. Speed versus time for the system of figure 23.

next time. Figure 16(b) shows the triangle source waveform obtained with this approach.

Next, consider a comparator with inputs x_1, x_2 and output y. Figure 17(a) shows the x_1, x_2, and y waveforms when a constant time step is used. Abrupt changes in $x_2(t)$ are tracked by the simulator (as discussed above); however, $y(t)$ is not resolved correctly by the simulator, e.g., see the transition at $t = 2.6$ s. One way to improve the y waveform is to uniformly reduce the time step, but this would make the simulation slower, and from an accuracy perspective, small uniform time steps may not even be required.

For resolving the abrupt transition in $y(t)$ on a shorter time scale but without making the 'normal' time step (denoted by $\Delta t_{\mathrm{normal}}$) small, GSEIM uses the following scheme. The comparator template stores the previous time point t_{-1} and the corresponding solutions $x_1^{(-1)}, x_2^{(-1)}$. Knowing these and the current time point and solutions $(t_0, x_1^{(0)}, x_2^{(0)})$, it uses linear extrapolation to compute the time t' at which $x_1 - x_2$ would cross zero. If t' is within $\Delta t_{\mathrm{normal}}$ of t_0, GSEIM places one time point just before and one time point just after t'. Figure 17(b) shows the waveforms obtained with this method. The transition at $t = 2.6$ s is now seen to be resolved properly.

When the linear extrapolation technique is used for the comparator inputs shown in figure 18(a), the zero crossing at $t \approx 1.93$ s is not treated correctly because the linear extrapolation overestimates t' in this case, as illustrated in figure 19. This problem can be addressed as follows. The comparator stores, in addition to t_0, t_{-1} (and the corresponding solutions), one more past time point t_{-2} and the corresponding solutions $x_1^{(-2)}, x_2^{(-2)}$. It uses this information to fit a quadratic which passes through $(t_0, u^{(0)}), (t_{-1}, u^{(-1)}), (t_{-2}, u^{(-2)})$ (where $u \equiv x_1 - x_2$), and computes t' at which it goes through zero. Figure 18(b) shows the waveforms obtained with the quadratic extrapolation scheme. It can be seen that the transition at $t \approx 1.93$ s is now treated correctly.

7. Simulation examples

We now look at simulation examples which demonstrate the capabilities of GSEIM. We consider two examples, both involving the induction machine model described by equations (6)–(10). Details regarding setting up the schematic, running the simulation, viewing the plots are described in [10] and are not reproduced here. Also, a detailed description of the system being simulated, values of the machine parameters, etc. are not included. The simulation times mentioned in the following are for a desktop computer (Ubuntu-19) with 3.2 GHz clock and 8 GB RAM.

7.1 *Free acceleration of induction motor*

In this example, we consider free acceleration of an induction motor with load torque $T_L = 0$. Figure 20 shows the GSEIM schematic diagram for the system when the induction motor template discussed in Section 4 is used. In this case, the conversion of V_a, V_b, V_c to V_d, V_q is required, and it is performed by the `abc_to_dq` element. Figure 21 shows the GSEIM schematic diagram for the same problem when the induction machine subcircuit of Section 5 is used. In this case, the *abc*-to-*dq* conversion is incorporated within the subcircuit. Figures 22(a) and (b) show the simulation results for speed and torque, respectively. The capability of GSEIM to produce plots of interest without cluttering the schematic diagram with scopes may be noted.

7.2 *V/f control of induction motor*

The GSEIM schematic diagram for *V/f* control of an induction motor [9] is shown in figure 23. The induction machine block shown in the figure corresponds to the

subcircuit of figure 14; however, we can also use the element template `indmc1.xbe` (see Section 4) directly. The `pwl20` element, which allows piecewise linear waveforms, is used to generate the frequency command which is smoothened using the `lag_1` element, satisfying

$$\frac{dy}{dt} = \frac{1}{T_r}(-y + x) \quad (17)$$

The *V/f* conversion is provided by `pwl10_xy`.

Simulation of this system is computationally more demanding than the free acceleration example because (a) its size is larger and (b) accurate resolution of pulse width modulation (PWM) voltages requires smaller time steps. For resolving the PWM waveforms correctly, the linear extrapolation technique described in Section 6 has been used. Figure 24 shows the PWM voltages in the steady state. It is seen that the transitions between low and high levels are treated properly. Figure 25 shows the motor speed versus time.

It is interesting to compare the simulation times when two different approaches are used: (a) induction machine as a subcircuit (described in Section 5) and (b) induction machine as an element template (described in Section 4). To enable a fair comparison, the same numerical method (RKF45) was used in both cases, and the algorithmic parameters were kept the same. The simulation times were found to be 2.33 and 1.55 s, respectively, for (a) and (b). This clearly brings out the advantage of implementing equations at a low level (the template level) rather than as a subcircuit. On the other hand, the subcircuit implementation is often easier, and therefore it may be preferred when simulation time is not large enough to be of concern.

8. Conclusions and future work

In summary, we have presented a new general-purpose ODE solver called GSEIM which allows the use of explicit as well as implicit methods. The organisation of the program has been described. A useful feature of GSEIM is that it allows the user to write new templates to incorporate element equations. In addition, it also allows the use of subcircuits (hierarchical blocks). These two facilities are illustrated with the help of examples. The importance of correct handling of abrupt changes is pointed out, and the techniques used by GSEIM to handle abrupt changes are explained. Finally, two representative simulation examples are presented. The computational advantage of incorporating element equations in a basic template rather than a subcircuit is brought out.

The GSEIM package, along with a users' manual, is available under the GNU general public license [10]. From the examples presented in this paper, it can be seen that GSEIM has features comparable to the commonly used Simulink package. The fact that it is an open-source package is expected to make GSEIM an attractive alternative, especially for educational purposes. It is envisaged that users will add to the element library, make up useful simulation examples, and make them available to other users through the GSEIM repository.

Further developments in GSEIM are expected to address the following issues:

(a) Additional features in the schematic capture GUI such as rectangular wires, 'bus' facility (i.e., grouping of several connections into a single wire), use of images in the element symbols, provision for adding text to the canvas, etc.

(b) Allowing electrical type elements, which satisfy some form of Kirchhoff's current and voltage laws, in addition to the 'flow-graph' type elements currently available. The option of using explicit or implicit methods provided by GSEIM is expected to be particularly useful in this development. Addition of electrical elements will significantly enhance the applications handled by GSEIM.

(c) Allowing different time stepping schemes for flow-graph elements and electrical elements.

References

[1] *Simulink – Simulation and Model-Based Design*. https://in.mathworks.com/products/simulink.html
[2] *DYMOLA Systems Engineering*. https://www.3ds.com/products-services/catia/products/dymola/
[3] *Xcos: Dynamic Systems Modeler*. https://www.scilab.org/software/xcos
[4] McCalla W J 1987 *Fundamentals of Computer-Aided Circuit Simulation.* Kluwer Academic Publishers, Boston
[5] Shampine L F 1994 *Numerical Solution of Ordinary Differential Equations.* Chapman and Hall, New York
[6] Gerald C F and Whitley P O 1999 *Applied Numerical Analysis.* Pearson Education India, New Delhi
[7] Chapra S C and Canale R P 2000 *Numerical Methods for Engineers.* Tata McGraw-Hill, New Delhi
[8] Burden R L and Faires J D 2001 *Numerical Analysis.* Thomson, Singapore
[9] Patil M B, Ramanarayanan V and Ranganathan V T 2009 *Simulation of Power Electronic Circuits.* Narosa, New Delhi
[10] *GSEIM*. https://github.com/gseim/gseim
[11] Bank R E, Coughran W M, Fichtner W, Grosse E H, Rose D J and Smith R K 1985 *IEEE Trans. Comput.-Aided Des. Intell. Circuits Syst.* **4** 436
[12] *GNU Radio Manual and C++ API Reference*. http://www.gnuradio.org
[13] *OpenModelica*. https://openmodelica.org/
[14] *Verilog-AMS Language Reference Manual.* https://www.accellera.org/downloads/standards/v-ams
[15] Nenzi P 2014 Ngspice circuit simulator release 26