# C++ implementations of numerical methods for solving differential-algebraic equations: Design and optimization considerations

**Article** *in* ACM Transactions on Mathematical Software · December 1999

DOI: 10.1145/332242.334001 · Source: DBLP

Christopher E. Kees
Louisiana State University
**86** PUBLICATIONS **1,847** CITATIONS

# C++ Implementations of Numerical Methods for Solving Differential-Algebraic Equations: Design and Optimization Considerations

CHRISTOPHER E. KEES and CASS T. MILLER
The University of North Carolina, Chapel Hill

Object-oriented programming can produce improved implementations of complex numerical methods, but it can also introduce a performance penalty. Since computational simulation often requires intricate and highly efficient codes, the performance penalty of high-level techniques must always be weighed against the improvements they enable. These issues are addressed in a general object-oriented (OO) toolkit for the numerical solution of differential-algebraic equations (DAEs). The toolkit can be configured in several different ways to solve DAE initial-value problems with an adaptive multistep method. It contains a wrapped version of the Fortran 77 code DASPK and a translation of this code to C++. Two C++ constructs for assembling the tools are provided, as are two implementations of an important DAE test problem. Multiple configurations of the toolkit for DAE test problems are compared in order to assess the performance penalties of C++. The mathematical methods and implementation techniques are discussed in detail in order to provide heuristics for efficient OO scientific programming and to demonstrate the effectiveness of OO techniques in managing complexity and producing better code. The codes were tested on a variety of problems using publicly available Fortran 77 and C++ compilers. Extensive efficiency comparisons are presented in order to isolate computationally inefficient OO techniques. Techniques that caused difficulty in implementation and maintenance are also highlighted. The comparisons demonstrate that the majority of C++'s built-in support for OO programming has a negligible effect on performance, when used at sufficiently high levels, and provides flexible and extensible software for numerical methods.

Categories and Subject Descriptors: D.3.2 [**Programming Languages**]: Language Classifications—*C++*; *Fortran 77*

General Terms: Algorithms, Design, Experimentation, Languages, Performance

Additional Key Words and Phrases: Differential-algebraic equations

This work was supported in part by U.S. Army Waterways Experiment Station Contract DACA39-95-K-0098, Army Research Office Grant DAAL03-92-G-0111, and National Institute of Environmental Health Sciences Grant 5 P42 ES05948. Computing activity was partially supported by allocations from the North Carolina Supercomputing Center.
Authors' address: Department of Environmental Sciences and Engineering, The University of North Carolina, Chapel Hill, 104 Rosenau Hall, Chapel Hill, NC 27599–7400.

## 1. INTRODUCTION

The scientific computing community's need for implementations of complex numerical algorithms has awakened interest in the potential benefits of object-oriented programming (OOP) compared to traditional approaches. These benefits include (1) more flexible and extensible implementations of numerical algorithms, (2) simplified user interfaces, and (3) better readability from the developer's perspective [Barton and Nackman 1994; Donescu and Laursen 1996; Dubois-Pelerin et al. 1992]. Implementing numerical algorithms in a purely procedural style can obscure the high-level conceptual structure of both the algorithm and the code. The end result can be a code that is difficult to understand and modify. In the OOP paradigm the major concepts (abstractions) in the numerical algorithm are implemented as well-defined, self-contained modules. These modules interact with one another through simple interfaces, which hide low-level implementation details from high-level conceptual views. The loosely coupled, modular structure of the code is easier to maintain and modify and more clearly represents the abstract form of numerical algorithms.

There are trade-offs in using OOP techniques for numerical programming. For numerical simulation, the computational overhead of the language constructs supporting OOP can outweigh the usefulness of the paradigm [Dubois-Pelerin et al. 1992]. C++ and Fortran 90 both have some built-in support for OOP although both are extensions of procedural languages. C++ is a suitable OOP language for numerical programming because programmers can work with low-level machine details to ensure efficiency (when necessary) and high-level OOP abstractions to gain the power of the high-level OOP paradigm [Barton and Nackman 1994]. Since C++ is in widespread use in industry, software and tutorial support abounds for beginning programmers. The availability of compilers in industry does not, however, ensure the efficiency of C++ for numerical programming or the availability of standard numerical libraries in C++.

The run-time efficiency of implementations in C++ relative to analogous implementations in Fortran 77 and C is an open question for numerical programmers and is the main motivation for the comparisons presented in this work. Large, complicated numerical codes for solving partial differential equations (PDEs) are often targeted at problems that can tax the resources of available computers, so their run-time efficiency is very important. Although many PDE codes have been developed in C++, their performance is usually assumed to be worse than that of comparable Fortran 77 codes, and, with some notable exceptions [Arge et al. 1997], the literature describing them rarely provides detailed quantitative comparisons with Fortran implementations. One factor affecting performance may be the lack of fully featured optimizing compilers supporting standard C++, likely due to the recent standardization of C++, and the immaturity of methods for optimizing OO code. The lack of stable optimizing C++ compilers implementing the recent C++ language standard and the often significant computational differences in C++ and Fortran 77 implementations of similar complex numerical algorithms have impeded evaluation of the performance penalty of OOP in C++.

In this work we present the DAE Toolkit (DAE-TK), an OO code for general DAE solution methods. We have implemented an adaptive multistep method as one possible tool configuration. The method is the same one originally implemented in the Fortran 77 code DASPK, and we have implemented the algorithm so that the C++ version is mathematically and numerically equivalent to the DASPK implementation, while still using the OOP paradigm to construct the code. Having a translation of the original Fortran 77 algorithm, we are able to assess the performance of C++ for large-scale numerical programs. In presenting this work our objectives are (1) to illustrate the advantages to numerical programming resulting from OOP techniques, (2) to highlight OOP design considerations that are useful in complex numerical applications, and (3) to evaluate the performance penalty of particular language constructs and coding styles in C++.

These objectives could be met with several approaches. We chose the solution of DAEs as our approach because (1) modern DAE solvers have sufficient complexity to show a clear contrast between procedural and OO approaches, (2) solving DAEs can require sufficient computational effort to provide a meaningful evaluation of alternative implementations, (3) DAEs must be solved in many scientific applications, and (4) sophisticated, well-tuned solvers for DAEs based upon relatively mature methods have been coded using popular procedural languages, providing a good basis for comparison with OO methods.

## 2. BACKGROUND

The importance of solving DAE initial-value problems (IVPs) numerically has been recognized for over 20 years [Brenan et al. 1996]. Interest in DAEs arose because many mathematical models in science and engineering occur naturally as systems of differential equations with algebraic constraints; DAEs have arisen as constrained variational problems in mechanics and as network models in electrical engineering [Gear et al. 1985; Newcomb 1981]. Often these problems were converted to systems of ordinary differential equations (ODEs) and solved numerically, although such numerical solutions often failed to satisfy the algebraic constraints [Brenan et al. 1996]. Another factor that has produced interest in DAEs is stiffness in systems of ODEs. Systems that appear naturally as DAEs are often stiffer when they are rewritten as ODEs. The method of lines (MOL) solution of PDEs is one case where this commonly occurs [Brenan et al. 1996; Kelley et al. 1998; Tocci et al. 1997]. Another problem that is often encountered in the MOL solution of PDEs is that the sparse structure of a DAE is destroyed when it is converted to an explicit system of ODEs [Tocci et al. 1997]. This loss of sparse structure can lead to extreme inefficiency for large systems.

The development of efficient numerical methods for solving DAEs is an active area of research, although a variety of efficient methods already exists [Brenan et al. 1996]. These solution approaches can be lumped into three categories: single-step, multistep, and extrapolation methods. While the efficiency of a method depends on the characteristics of the problem being solved, backward difference formula (BDF) methods—a class of multistep methods—were developed early on [Gear 1971] and remain among the most popular. Even within this limited class of method, developing an efficient solver requires many decisions that affect performance: which specific set of BDFs to use; which error estimate to use; how to control the order of the approximation and the step-size; and which nonlinear and linear algebraic equation solution methods (including matrix storage considerations) to incorporate in the DAE solution method.

The variety of design considerations has led to the development of several general codes using a BDF solution approach. Of these codes, the DASSL family is the most popular for solving DAEs [Brown et al. 1994; Maly 1996; Petzold 1983]. DASSL implements the fixed leading coefficient (FLC) BDF in a variable-step-size variable-order scheme. The scheme in DASSL adjusts order and step-size to maximize efficiency while maintaining a prescribed local truncation error of the computed solution. DASPK is essentially backward compatible with DASSL, but it has additional linear algebra capabilities. In particular, it can use a matrix-free Jacobian with the generalized minimal residual (GMRES) method, making it appropriate for solving extremely large systems. In addition, there are other codes based on DASSL for more specific applications [Maly 1996]. Modifications to these standard solvers, in some cases, yield more efficient solvers than the general-purpose codes typically used [Kelley and Miller 1998; Tocci et al. 1997]. Such modifications, however, require a detailed understanding of both the theory and construction of DASPK and can be time-consuming to implement and evaluate.

### 2.1 Object-Oriented Scientific Computing

Modern DAE solvers can be large and complicated, leading to difficulties in code maintenance and extension. These difficulties can be especially prevalent in codes written in procedural styles [Barton and Nackman 1994]. In order to create less complex codes some scientists and engineers have begun to use OOP languages instead of procedural languages such as Fortran 77 and C.

The three fundamental concepts of OOP are objects, class hierarchies, and polymorphism. An *object* is an entity in the programming language composed of data and functions that operate on the data. A *class* is an abstraction that represents all objects with the same configuration of data and functions. A class is called a *child* class if it has a superset of the data and functions of another class. Likewise a class is a *parent* class if it has a subset of the data and functions of another class. A child class is said to *inherit* such a subset of data and functions from a parent class; and a set of classes related by inheritance is called a class *hierarchy*. *Polymorphism* is the binding of multiple implementations of an abstract method to a single method name or interface. In order to support polymorphism the code or language must provide some mechanism for choosing which implementation is bound to the interface at a specific location in the code. This binding can occur at compile time or at run-time. See Barton and Nackman [1994] and Stroustrup [1997] for a more complete discussion of OOP and the C++ syntax supporting OOP.

OOP has been used to implement complex software systems, such as finite-element method (FEM) solvers [Donescu and Laursen 1996; Dubois-Pelerin and Pegon 1997; Eyheramendy and Zimmermann 1996; Zimmermann and Eyheramendy 1996; Zimmermann et al. 1992], grid generation and triangulation algorithms [Karamete et al. 1997], and adaptive numerical solvers [Besson and Foerch 1997; Lewis et al. 1997; Liu et al. 1996]. These works clearly demonstrate the conceptual appeal of OO approaches for managing complexity, but run-time efficiency is also important in computationally intensive scientific computing applications. How to use the features of OO methods to produce run-time-efficient applications warrants further investigation. The appropriate OO language in which to construct complex, computationally demanding scientific computing applications is the first decision affecting run-time efficiency; two leading candidates are Fortran 90 and C++.

Fortran 90 is a superset of Fortran 77 that adds support for classes and objects with the `MODULE`, `DATA`, and `INTERFACE` constructs. It supports limited forms of polymorphism through operator overloading, but lacks explicit support for inheritance and run-time polymorphism. Fortran 90 can be used to support a larger set of OO constructs than are implemented in standard Fortran 90 [Decyk et al. 1997]. Comparisons between C++ and Fortran 90 showed that there was little difference in performance among C++, Fortran 90, and Fortran 77 for the Livermore kernels [Cary et al. 1996]. These kernels contain only low-level operations, however, so this result cannot be extrapolated to full-scale usage of OOP.

C++ is an OOP language that is a superset of the C language [Stroustrup 1997]. Many groups have developed complex C++ applications: the linear algebra packages LAPACK++ [Dongarra et al. 1993], SparseLib++ [Dongarra et al. 1994], IML++ [Dongarra et al. 1996]; the PDE packages KASKADE [Beck et al. 1995], DIFFPACK [Bruaset and Langtangen 1997], and Overture;[1] the adaptive mesh refinement package SAMRAI,[2] and the computational science toolkits POOMA and PETE, to name but a few.[3] In spite of the large number of scientific computing packages being developed in C++ there are few published comparisons of efficiency for closely corresponding implementations in Fortran 77 or Fortran 90.

A performance penalty of less than 20% was suggested in Bruaset and Langtangen [1997] for simple linear algebra calculations, but the penalty

[1]Brown, D. L., Quinlan, D. J., and Henshaw, W. See http://www.llnl.gov/CASC/Overture/.
[2]Kohn, S., Garaizar, X., Hornung, R., and Smith, S. See http://www.llnl.gov/CASC/SAMRAI/.
[3]For POOMA see Karmesin, S. et al., http://www.acl.lanl.gov/Pooma/. And for PETE see Crotinger et al., http://www.acl.lanl.gov/pete/.

for using high-level constructs, such as virtual functions, was not studied in detail. A more complete analysis of performance penalties is given in Arge et al. [1997], where several large-scale PDE codes in C++ and Fortran 77 are compared along with a range of smaller test problems. The C++ codes had normalized run-times as low as 0.8 and as high as 2.5 when using Fortran 77 codes for the same test problems as benchmarks. The implementation details, however, tended to be much more general for the C++ codes, and, therefore, the run-time efficiency comparisons tended to favor the more specialized Fortran 77 implementations. The authors suggest that the C++ compilers used for the comparisons were still relatively immature and that future compilers might optimize C++ better. Efficiency comparisons were presented in Besson and Foerch [1997] which showed that C++ had normalized run-times between 0.71 and 1.8 with respect to Fortran 77 when comparing the same FEM method implemented in a significantly more general way in the C++ code. Robison [1996] provided details of how some low-level OO constructs in C++ can be optimized at least as well as C or Fortran 77 by modern C++ compilers but noted that the performance penalty of higher-level OO constructs was still an open question.

## 3. NUMERICAL METHODS

Numerical approaches for the solution of DAEs can be complex. This complexity results from the predominant use of multistep, variable-order, variable-step-size, error-controlled approaches. In this work, we use the most common approach for solving DAEs: a BDF approach. The complete details of this approach are covered elsewhere [Brenan et al. 1996], but we will summarize it so that the nature of the method's complexity and the details of the OO implementation can be appreciated.

### 3.1 Scope

We consider DAEs of the general nonlinear, fully implicit vector form

$$F(t, y, y') = 0, \tag{1}$$

where $t \in \mathcal{R}$, $y$, $y' \in \mathcal{R}^n$, and $F : \mathcal{R}^{2n+1} \rightarrow \mathcal{R}^n$. The function $F$ is often called the *residual* function. Equations in the system described by (1) are some combination of ODEs and algebraic equations. Solutions of (1), in addition to obeying the ODEs, must take their value on some $r$-manifold in $\mathcal{R}^n$ that is determined partly by the algebraic equations [Brenan et al. 1996]. An IVP consists of a DAE and a set of initial values $t_0$, $y_0$, $y'_0$. For a given IVP, the methods presented in this work generate a numerical solution approximation to the solution at a finite number of times on an interval $[t_0, T]$.

The DAEs considered are assumed solvable in the usual sense and of index 0 or 1 [Brenan et al. 1996]. An index 0 DAE system is equivalent to a system of ODEs. An index 1 DAE system is a system of equations that must be differentiated once in order to derive a solvable system of ODEs [Brenan et al. 1996; Gear 1986; Leimkuhler 1988]. While higher-index systems arise in important applications [Brenan et al. 1996], the solution of such systems is an active area of research and considered beyond the scope of this work.

## 3.2 Solution Algorithm

A common approach for solving DAEs based upon the BDF approach follows the simplified algorithm for a time step:

—predict the solution of the DAE by extrapolating from previously computed points in the solution history with a Lagrange polynomial of suitable order;

—form a corrector equation for the unknown solution vector $y_{n+1}$ using a Lagrange polynomial of suitable order that terminates at the unknown solution point;

—solve the nonlinear corrector equation to an appropriate error tolerance if possible; otherwise reduce step-size and/ or order and retry the step;

—update the solution history;

—estimate the allowable step-size and order of the solution method for the next step; and

—output solution information if desired, performing any necessary interpolation.

Within this general approach, choices exist regarding (1) the form of the BDF used, (2) the range of orders for the method, (3) the strategy for controlling solution order and step-size, (4) the nonlinear algebraic equation solution method used, (5) the nonlinear iteration matrix update procedure, (6) the linear algebraic system solution methods, and (7) the range of matrix structures and coefficient storage methods implemented. Clearly, these decisions can significantly affect the performance of the solver and the details of any code that implements these methods. Using traditional procedural approaches, changes in these choices can be difficult and time-consuming to implement, even for the developers of the code.

## 3.3 BDF Approach

Generally, in BDF methods the derivative of a Lagrange polynomial is substituted for the derivative appearing in the DAE to produce a nonlinear system. The solution of the nonlinear algebraic system approximates the solution of the DAE. The Lagrange polynomial passes through the unknown solution at the current time and interpolates a finite number of previously approximated solution points (the solution history).

To develop these ideas further, we consider briefly the fixed leading coefficient (FLC) BDF predictor/corrector formulation implemented in this work. Given $k + 1$ previous solutions for $y$, a Lagrange predictor polynomial is constructed through the solution such that

$$\omega^{p}_{k,\,n+1}(t_{n-i}) = y_{n-i}, \quad i = 0, 1, \ldots, k \tag{2}$$

where $\omega$ is a Lagrange polynomial expression; the superscript $p$ indicates that it is a predictor expression; $k$ denotes the order, which ranges between 1 and 5; and $n + 1$ is a time-step index that refers to the time for which a solution is sought. This polynomial yields an explicit approximation for the solution and its derivative at $t_{n+1}$, given by

$$y^{p}_{n+1} = \omega^{p}_{n+1}(t_{n+1}) \tag{3}$$

$$y'^{p}_{n+1} = \omega'^{p}_{n+1}(t_{n+1}). \tag{4}$$

We then construct a $k$th degree Lagrange corrector polynomial and implicitly define its value at $t_{n+1}$ by requiring that it satisfy the DAE at time $t_{n+1}$. Instead of requiring that the polynomial interpolate the solution at the last $k$ steps as one does with a variable coefficient BDF, we define the corrector polynomial such that it interpolates the predictor polynomial at $k$ equally spaced steps before $t_{n+1}$. Hence, the corrector polynomial satisfies the conditions:

$$\omega^{c}_{n+1}(t_{n+1} - ih_{n+1}) = \omega^{p}_{n+1}(t_{n+1} - ih_{n+1}) \quad i = 1,2, \ldots, k \tag{5}$$

$$F(t_{n+1}, \omega^{c}_{n+1}(t_{n+1}), \omega'^{c}_{n+1}(t_{n+1})) = 0 \tag{6}$$

where $\omega'^{c}_{n+1}(t_{n+1}) = y'^{c}_{n+1}$ is the FLC BDF for the derivative at $t_{n+1}$.

It is straightforward to show that [Jackson and Sacks-Davis 1980]

$$\omega'^{c}_{n+1}(t_{n+1}) = y'^{p}_{n+1} + \alpha(y^{c}_{n+1} - y^{p}_{n+1}) \tag{7}$$

where

$$\alpha = \frac{\alpha_0}{h_{n+1}} = \frac{1}{h_{n+1}}\sum_{j=1}^{k}\frac{1}{j}, \tag{8}$$

and $h_{n+1}$ is the temporal step-size taken to solve for $y_{n+1}$. The key feature of this approach is that $\alpha$ remains fixed unless the order or step-size changes.

## 3.4 Error Control

The FLC BDFs can be used to derive the error estimate [Jackson and Sacks-Davis 1980]

$$\tau_n = C_n h_n^{k+1} y^{k+1}(t_n) + O(h_{n-k}^{k+2}) \tag{9}$$

where it is assumed that $y$ is $C^{k+1}$-continuous and

$$C_n = \frac{\xi_1 \cdots \xi_k}{(k+1)!} \frac{(1 - \alpha_{n,0} + \alpha_0)}{\alpha_0} \tag{10}$$

$$\xi_i = \frac{(t_n - t_{n-i})}{h_n} \tag{11}$$

$$\alpha_{n,0} = -1 + \frac{1}{\xi_2} + \cdots + \frac{1}{\xi_k} \tag{12}$$

$$\alpha_0 = -1 + \frac{1}{2} + \cdots + \frac{1}{k}. \tag{13}$$

To save computational effort, $\tau_n$ is approximated by

$$\tau \approx K_n \| y_n^c - y^p \| \tag{14}$$

where $K_n$ is an expression involving the BDF coefficients, which can be found in Gear [1973], Jackson and Sacks-Davis [1980], and Brenan et al. [1996]. This approximation is asymptotically correct when the last $k$ steps were taken at constant step-size, as well as under slightly more general conditions [Brenan et al. 1996; Gear 1973; Jackson and Sacks-Davis 1980].

## 3.5 Solution of Algebraic Systems

To approximate the solution at the current time, we solve a system of algebraic equations for the unknown $y_{n+1}^c$, which, in terms of the predictor-corrector scheme, is

$$F[t_{n+1}, y_{n+1}^c, y'^p_{n+1} + \alpha(y_{n+1}^c - y_{n+1}^p)] = 0 \tag{15}$$

or in simplified notation as

$$F(t, y, \alpha y + \beta) = 0 \tag{16}$$

where all variables are evaluated at $t_{n+1}$, where $\alpha$ is a constant that depends upon the step-size and order of the Lagrange polynomial, and where $\beta = y'^p_{n+1} - \alpha y^p_{n+1}$.

When the original system of DAEs is nonlinear either in $y$ or in $y'$, then (16) is a nonlinear system of algebraic equations. This nonlinear system of equations is solved using a modified Newton iteration method, which is sometimes referred to as the chord method [Kelley 1995]. Instead of updating the Jacobian of the nonlinear system at every iteration, as in a standard Newton method, the modified Newton method applied to (16) reuses Jacobians for multiple iterations and over multiple time steps. When solution order or time step have changed enough so that the modified Newton iteration converges slowly (or not at all), the Jacobian is updated for the current system. We use the predicted value of the solution as the initial iterate, and, because this value is normally a good estimate of the solution, the nonlinear iteration typically converges in less than three iterations.

The chord iteration method is $q$-linearly convergent, so there exists a $\rho \in (0,1)$ such that

$$\|e_{(i+1)}\| \leq \rho \|e_{(i)}\| \tag{17}$$

where $e_{(i)} = y_{(i)} - y^*$, $i$ is an iteration index, and $y^*$ is the true solution. This allows the iteration to be terminated whenever

$$\frac{\rho}{1-\rho}\|y_{(i+1)} - y_{(i)}\| \leq \epsilon \tag{18}$$

where $\epsilon$ is the tolerance for the nonlinear iteration.

The norm in (18) is given by

$$\|y\| = \sqrt{(\sum(y^j/w^j)^2)}, \tag{19}$$

where

$$w^j = y_n^j{*}rtol^j + atol^j, \tag{20}$$

$j$ is the vector index, $y$ is an arbitrary vector, $y_n$ is the solution at the last step, and $rtol$ and $atol$ are relative and absolute tolerance vectors [Brenan et al. 1996]. In the implementations the relative and absolute tolerances are defined by the user and enter into the error expressions only through the norm defined in (19).

Success of the chord iteration approach is based on the sometimes false assumption that the Jacobian varies slowly with respect to time and is primarily affected by step-size and order changes. When the Jacobian varies quickly with time, the DAE solution scheme can be made more efficient by adding additional conditions to determine when Jacobians should be recomputed [Kelley et al. 1998; Tocci et al. 1997].

The nonlinear algebraic equations solution approach in turn requires the solution of linear systems of equations. This is accomplished using lower-upper decomposition for sparse banded systems, and preconditioned GMRES for large, sparse, nonsymmetric systems [Brown et al. 1994].

## 4. IMPLEMENTATIONS

### 4.1 High-Level Design

In the following sections, we will describe how we used C++'s built-in support for objects, class hierarchies, and polymorphism to design and implement DAE-TK. We will use the types `real` and `Vec` in declarations and definitions to represent the mathematical abstractions of real numbers and $n$-dimensional real vectors. The `real` type is simply defined in a compilation unit as either double or float, and `Vec` is a simple numerical vector type similar to that found in the Template Numerical Toolkit (http://math.nist.gov/tnt.html).

4.1.1 *Problem Definition.* We see from the definition of a DAE IVP in Section 3 that a numerical method must be supplied with a residual function ($F$ in the notation of Section 3) that defines the DAE and the set of initial values $t_0$, $y_0$, and $y'_0$. We can use this information as the basis for the DAE IVP abstraction. This abstraction is represented in the code as the virtual base class `DaeDefinition` whose interface consists of the following pure virtual functions:

```
virtual bool residual(const real& t,const Vec& y,const
                    Vec& yp, Vec& F)=0;
virtual const real& getT0()=0;
virtual const Vec& getY0()=0;
virtual const Vec& getY0prime()=0;
```

where `y` = $y$, `yp` = $y'$, and `F` = $F(t, y, y')$ in the notation of Section 3; `residual` returns the Boolean value true if the function `residual` encountered an error in the input values or false otherwise (this will be the convention for all Boolean return values); and each of the functions beginning with `get` return the values that they describe. In order to solve a problem with DAE-TK a user must supply a child class that implements these functions in order to determine the specific DAE IVP that it represents.

4.1.2 *Integration Methods.* We form a class hierarchy of DAE solver abstractions which are derived from the abstract base class `Integrator`. The function that provides the characteristic behavior of the numerical method abstraction in the public interface of `Integrator` is defined as

```
virtual bool calculateSolution(const real& tout,Vec& y, Vec&
yp)=0;
```

where `tout` is the time where the solution is desired, where `y` on return is the desired solution at `tout`, and where `yp` is the derivative of the solution at `tout`. Any class derived from `Integrator` must redefine `calculateSolution` so that it applies a numerical method to a DAE IVP and obtains the solution at `tout`. The numerical method may require that many steps be taken in order to find the solution; it may require only one step; or it may require only an interpolation from existing solutions.

In order to provide finer-grained control over the numerical method, `Integrator` also contains the function

```
virtual bool step(const real& tOut,real& tStep,Vec& yAtTStep,
                    Vec& ypAtTStep)=0;
```

which takes a single step in the direction of `tOut` and returns the solution at the time of this step, `tStep`. This function allows a user of an `Integrator` class to implement specific error checking or adaption after each step in specific problem-dependent situations.

```
class BackwardEuler : public Integrator
{
 public:

   //constructor
   BackwardEuler(DaeDefinition& problem):theDae(&problem) {}
   virtual bool step(const real& tOut,real& tStep,
                    Vec& solutionAtTStep, Vec& solutionPrime)
   {
     //implement the backward Euler method for a step
     ...
     //test whether the computed values satisfy the DAE
     theDae->residual(tStep,solutionAtTStep,solutionPrime,F);
     if (F > tolerance)
      return solverFailed = true;
     else
      return solverFailed = false;
   }
   ...
 private:
   DaeDefinition* theDae;
   Vec F;
   bool solverFailed;
   ...
}
```

Fig. 1.

4.1.3 *Integrator-DaeDefinition Interfaces*. Before we consider how Integrators assemble a numerical method out of lower-level tools, we first consider how the DaeDefinition and Integrator hierarchies might be connected using polymorphism. A general DAE solver can benefit from a polymorphic relationship between the interface defined abstractly in Dae-Defintion and the many problem-specific implementations. This can be accomplished in two different ways.

Suppose we wish to implement the backward Euler method for DAEs in our Integrator hierarchy. Consider the sketch of an implementation, shown in Figure 1. Now consider the use of such a class on a specific user-defined child class of DaeDefinition (Figure 2). In this style of implementation a BackwardEuler object obtains the information it needs about child classes in the DaeDefinition hierarchy through a pointer to the base class DaeDefinition. The virtual function mechanism ensures that the correct implementation of the functions in the interface are called (i.e., the child class implementation in SimpleDaeProblem and not the base class). The compiler will generate a virtual function table so that when step calls theDae->residual(...), the actual type of the pointer theDae is identified (at run-time), and the corresponding residual() of that type is called. This type of polymorphism is also called "dynamic binding" or "run-time polymorphism" because the correct implementation of the virtual function is bound to the function call at run-time.

Now consider an alternate implementation (Figure 3). In this implementation BackwardEuler is a class template parameterized on class T, which is any class that has the same functions in its interface as DaeDefinition.

```
//define the DAE IVP
class SimpleDaeIvp : public DaeDefinition
{
public:
  //define the residual function
  residual(const real& t, const Vec& y, const Vec& yp, Vec& F)
  {
   F(0)=yp(0)-(4.0*y(0))+y(1)-y(2)+((t+2)*(t+2));
   F(1)=yp(1)+y(0)-3.0*y(1)+2.0*y(2)-(2.0*t*t)-t-15;
   F(2)=yp(2)-y(0)+(2.0*y(1))-(3.0*y(2))+(3.0*t*t)-t+10;
   return false; //function defined for all real y and yp
  }
 //define the initial values
 getY0(){...}
 ...
}

int main()
{
  SimpleDaeIvp daeProblem;  //instantiate the DAE IVP problem
  BackwardEuler daeSolver(daeProblem); //instantiate the DAE solver
  ...
  //take a step using the backward Euler Method
  daeSolver.step(...);
  ...
```

Fig. 2.

```
template<class T>
class BackwardEuler : public Integrator
{
 public:
   BackwardEuler(T& problem):theDae(&problem) {}
   virtual bool step(const real& tOut,real& tStep,
                     Vec& solutionAtTStep, Vec& solutionPrime)
   {
     //Implement Backward Euler
     ...
     theDae->residual(...);
     ...
   }
   ...
 private:
   T* theDae;
   ...
}
```

and its use:

```
int main()
{
  SimpleDaeProblem daeProblem;  //instantiate the DAE IVP problem
  //instantiate the DAE solver for this problem
  BackwardEuler<SimpleDaeIvp> daeSolver(daeProblem);
  ...
}
```

Fig. 3.

When this `Integrator` is used with a specific problem definition class (i.e., a child class of `DaeDefinition`), the compiler will generate a class definition for that specific problem definition class. The compiler will require that the problem definition be defined correctly and that the generated class will contain a pointer to the actual type, `T`, of the object pointed to by `theDae`. In this way there is no dynamic binding of the function calls, but rather, they are bound directly at compile-time. This kind of polymorphism is called "static binding," compile-time polymorphism," or "parametric polymorphism" [Stroustrup 1997].

We implemented both styles of polymorphism in our toolkit in order to investigate the effects of dynamic binding on performance. For each `Integrator` in our toolkit there is one version that links to child classes of `DaeDefinition` using run-time polymorphism and another version using compile-time polymorphism instead. The member function `residual` is the lowest-level virtual function call in the codes we developed, so we can evaluate the effects of run-time polymorphism on performance by comparing the run-time efficiency of these `Integrators`.

4.1.4 *Support Hierarchies*. The numerical methods that we have discussed must solve linear and nonlinear systems of algebraic equations in order to approximate the solution. Candidates for classes in an OOP implementation are then linear and nonlinear solvers. Other abstractions arise from the mathematics or out of convenience to the implementation, such as Jacobians, vector-valued functions, norms, and data modules.

The default OOP structure for our code organizes these groups of abstractions into class hierarchies that use run-time polymorphism as described for the `DaeDefinition-Integrator` interface in order to link to each other. Each base class representing the interface of a certain abstraction has a hierarchy of derived classes that implement the interface in different ways. This hierarchical structure allows the user and developers to create a numerical method for solving DAEs by assembling the method out of four main modules: an `Integrator`, a `NonlinearSolver`, a `LinearSolver`, and a `Jacobian`. The necessary information about the DAE IVP is obtained through a pointer to a `DaeDefinition` object or a type parameter in the case of the templatized `Integrators`.

The current class hierarchies in DAE-TK are given in Figure 4 which uses the Unified Modeling Language (UML) to represent OOP concepts graphically [Quatrani 1998]. The boxes represent classes, and the arrows connecting them represent inheritance relationships. The arrow points toward the parent class in each relationship. The main functions in the interface of each base class of a hierarchy are given below the name of the base class (with the exception of the `DataCollector` hierarchy, which contains too many functions in the interface to list).

The `Integrator` hierarchy contains two implementations of the FLC BDF method: `FLCBDF` is our C++ implementation, and `DASPK` is a "wrapper" for the Fortran 77 routine `ddaspk.f`. `FLCBDFT` and `DASPKT` use compile-time polymorphism as described above to link to problem definition

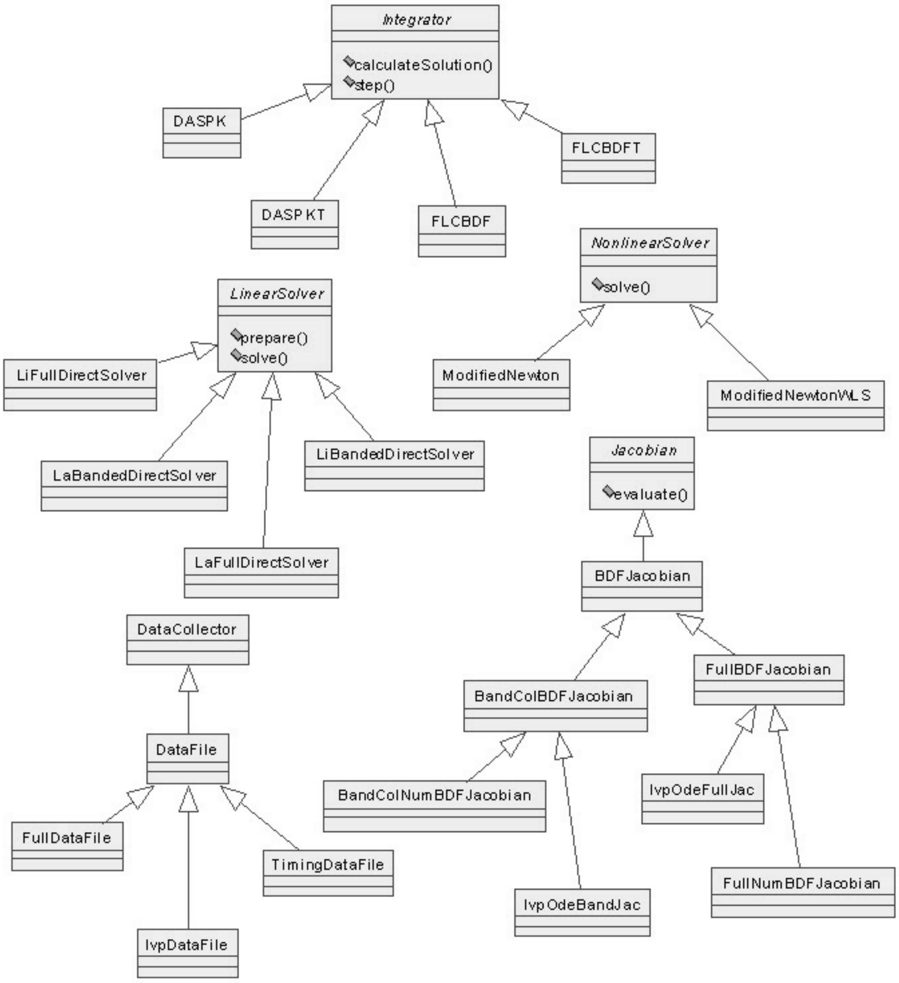

Fig. 4. DAE-TK class hierarchies.

classes, but are identical in all other respects to FLCBDF and DASPK respectively.

The NonlinearSolver class hierarchy contains two child classes: one for the modified Newton method and one for a modified Newton method containing a line search algorithm. The LinearSolver class hierarchy contains several solvers that are wrappers for Fortran 77 routines in LINPACK and LAPACK [Dongarra et al. 1984; 1993].

The Jacobian class hierarchy contains a number of classes that represent different structural forms of the Jacobian (e.g., full or banded) and encapsulate the various methods of computing Jacobians. The Jacobians can be computed numerically from the residual function using functionality in `FullNumBDFJacobian` or `BandColNumBDFJacobian`. The user can derive a class in order to define the Jacobian using analytical derivatives of the residual function.

The `DataCollector` class hierarchy is used to collect solution data and information on the run-time characteristics of the code. It contains several functions in its interface that are omitted for the sake of brevity. These functions can be used by all classes in DAE-TK in order to provide data throughout the solution process.

We arrived at this design through several iterations of implementation and testing of DAE-TK. Our initial design consisted of a set of abstractions that seemed to match the high-level conceptual pieces of the algorithm. During early iterations it became apparent that abstractions were poorly chosen, as interfaces became large and as extraneous information was passed through the interfaces at inopportune times. For instance, we added a `Jacobian` class because without it information about the particular BDF method and problem definition being used were necessarily passed through the `NonlinearSolver` interface. The implementations of the `NonlinearSolver` child classes were overly complicated and lacked generality, because the otherwise general nonlinear solution methods had to process information about the integration method. We eventually settled on the current group of abstractions, because the module interfaces are simple while still allowing complicated numerical methods to be assembled together to form a solution method for DAEs. Many alternatives to this design exist.

As an example of how modules are pieced together to form a working implementation of a numerical method for a particular problem, consider the segment of code shown in Figure 5. For a templatized `Integrator` the only change to this code would be in the `Integrator` declaration lines:

```
FLCBDFT integrator(dae,linearSolver,nonlinearSolver,
             Jacobian,norm,data);
```

The structure created by this sample problem is illustrated in Figure 6, again using UML. The boxes represent objects, and the lines represent interfaces through which information is exchanged between objects. The information is exchanged in these cases by accessing functions in the public interface of the base classes in DAE-TK through pointers to these base classes.

DAE-TK is flexible because it allows many different configurations of numerical methods to be assembled by interchanging classes from the various hierarchies. This feature is especially useful with `LinearSolvers` and `DataCollectors`, since most methods for solving linear equations are only useful for limited groups of linear systems, and the organization of output data can be as varied as the problems to be solved. The code can easily be extended because the clean and simple interfaces of the various class hierarchies specify what behavior should be provided by new classes and allow developers to concentrate on the implementation details of the module—not how these modules should be introduced into the larger

```
 //Define the residual function globally for a simple DAE

 class SimpleDaeIvp : public DaeDefinition
 {
   public:
     //define constructor
     SimpleDaeIvp():y0(3),y0prime(3)
     {
      //set initial conditions
      t0;
      y0(0)=3.0; y0(1)=-7.0; y0(2)=-2.0;
      y0prime(0)=13.0; y0prime(1)=-5.0; y0prime(2)=1.0;
     }
     bool residual(const double& t,const Vec& y,const Vec& yp, Vec& F)
     {
      F(0)=yp(0)-(4.0*y(0))+y(1)-y(2)+((t+2)*(t+2));
      F(1)=yp(1)+y(0)-3.0*y(1)+2.0*y(2)-(2.0*t*t)-t-15;
      F(2)=yp(2)-y(0)+(2.0*y(1))-(3.0*y(2))+(3.0*t*t)-t+10;
     return false; //function defined for all real y and yp
    }
    getT0(){return t0;}
    getY0(){return y0;}
    getY0Prime(){return y0Prime;}
  private:
    real t0;
    Vec y0,y0Prime;
}

int main()
{
  //Initialize data
  ParamDatabase pd("driverInput.txt"); //read in parameters from a file

  int neq=3;  //set number of equations to 3

  double atol(pd("atol")),rtol(pd("rtol")),  //set tolerances

  //allocate vectors for solution
    Vec y(neq),yp(neq);

    //Instantiate problem definition object
    SimpleDaeIvp dae();

    //Instantiate FLCBDF solver and supporting objects for this problem
    FullDataFile data(dae.getT0());
    WeightedRMSNorm norm(neq);
    norm.setTolerances(atol,rtol);
    FullNumBDFJacobian Jacobian(neq);
    LaFullDirectSolver linearSolver(Jacobian);
    ModifiedNewton nonlinearSolver(linearSolver, norm,data, neq);
    FLCBDF integrator(dae,linearSolver,nonlinearSolver,Jacobian,norm,data);
    integrator.setTolerances(atol,rtol);

    // Solve problem on user defined discretization
    for (int i=1;i<=pd("numberOfSteps");i++)
      {
       integrator.calculateSolution(double(i)*pd("stepSize"),y,yp);
      }
    return 0;
 }
```

Fig. 5.

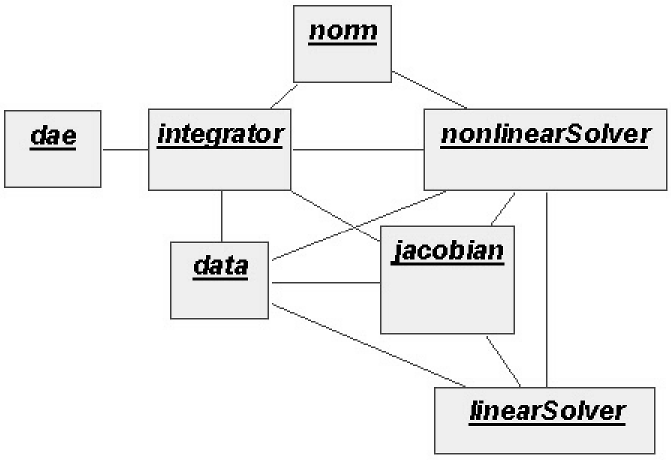

Fig. 6. Example configuration.

context of the numerical method. We will see in the Section 4.2 how OOP also makes the low-level implementations more understandable by organizing the implementation details into levels that correspond to conceptual levels of the numerical algorithm.

4.1.5 *Lessons Learned*. We tried many different implementations before arriving at this structure. At one stage we transformed all the major instances of run-time polymorphism into compile-time polymorphism through templates. Compile times and code bloat due to template instantiation were so severe that this method was not feasible. Our current version makes only limited use of templates for this purpose in the `FLCBDFT` and `DASPKT` integration modules. In addition the many nested template declarations that the fully templated version demanded were tedious and error prone, although errors were caught at compile time. We found that such an implementation was not more run-time efficient than the virtual-function-based design, but we do think templates have a place in numerical codes. In fact, templates are ideal for implementing polymorphism where the overhead of virtual functions is unacceptable as well as in their traditional use for implementing container classes. We have also experimented with an interface class to automate the assembly of numerical methods out of objects in the class hierarchy; future versions of DAE-TK will likely contain this useful feature. The current version allows rapid prototyping of research codes and problem-specific optimizations to be carried out while generating a minimum of single-use code. Some burden is still placed on the user to assemble a compatible set of tools, as no interface class is included. An interesting aspect of our design is that our class hierarchies are based on abstractions for methods rather than abstractions for the objects to which these methods are applied. For instance, we have a `LinearSolver` class hierarchy rather than a `LinearSystem` class hierar-

```
//formulate system of corrector equations
correctorEquations.setParameters(alpha,tnPlusH,*ynprime);

//evaluate corrector equations at current predictor
correctorEquations(*yn,residualVector);

//check Jacobian in storage and update if necessary
analyzeJacobian();
/* solve the nonlinear system of corrector equations.
   yCminusyP is the difference in the solution and initial guess;
   *yn contains the predictor value on entry and corrector value;
   on exit; and, residualVector is that value of F on entry. */
didNotConverge=nonlinearSolver->solve(yCminusyP,*yn,residualVector,
                                       correctorEquations);
```

Fig. 7.

chy. The emphasis on algorithm abstractions is simply a reflection (1) that the important mathematical complexity in building solution methods for DAEs is in the choice of algorithms and (2) that freedom from the multitude of representations of mathematical equations is important.

## 4.2 Low-Level Details

Many important and useful details lie hidden in the implementations of these class hierarchies. We now focus on the use of DAE-TK to implement FLC BDF methods and how this differs from the implementation in DASPK. We demonstrate how OOP manages complexity for this level of the implementation.

The numerical method outlined in Section 3 proceeds by converting the DAE to a nonlinear system, then converting the nonlinear system to a linear system and finally solving the linear system. Class `FLCBDF` uses classes in the `NonlinearSolver`, `LinearSolver`, and `Jacobian` class hierarchies to execute these conceptual steps. For example the nonlinear system of equations (`correctorEquations`) is formulated and solved with the lines shown in Figure 7. The pointer `nonlinearSolver` can point to any child class in the `NonlinearSolver` hierarchy. In a similar manner the class `ModifiedNewton` solves the linear system it generates with the following line:

```
linearSolver->solve(fAtX,p);
```

Again, the pointer `linearSolver` can point to any class in the `LinearSolver` hierarchy. The corresponding calls in `ddaspk.f` are shown in Figures 8 and 9. Figure 8 shows the call for the solution of the nonlinear system, while Figure 9 shows the call for the solution of the linear system. The OOP approach drastically simplifies the interfaces to the solve functions (`DNSD`, `DGESL`, `DGBSL` in Fortran 77) and makes it unnecessary to decide through control statements whether the problem produces a banded linear system or a full linear system. Information such as matrix structure, tolerances, and vector lengths are removed from the section where a

```
      CALL DNSD(X,Y,YPRIME,NEQ,RES,PDUM,WT,RPAR,IPAR,DUMSVR,
     *          DELTA,E,WM,IWM,CJ,DUMS,DUMR,DUME,EPCON,S,TEMP1,
     *          TOLNEW,MULDEL,MAXIT,IRES,IDUM,IERNEW)
C


      LIPVT = IWM(LLCIWP)
      MTYPE=IWM(LMTYPE)
      GO TO(100,100,300,400,400),MTYPE
```

Fig. 8.

```
C     Dense matrix.
C
100   CALL DGESL(WM,NEQ,NEQ,IWM(LIPVT),DELTA,0)
      RETURN
C
C     Dummy section for MTYPE=3.
C
300   CONTINUE
      RETURN
C
C     Banded matrix.
C
400   MEBAND=2*IWM(LML)+IWM(LMU)+1
      CALL DGBSL(WM,MEBAND,NEQ,IWM(LML),
     *  IWM(LMU),IWM(LIPVT),DELTA,0)
```

Fig. 9.

```
      SELECT CASE (LIN_PAR%MTYPE)
C
C     Dense matrix.
C
      CASE (1,2)
      CALL DGESL(WM,NEQ,NEQ,LIN_PAR%IPVT,DELTA,0)
C
C     Dummy section for MTYPE=3.
C
      CASE (3)
C
C     Banded matrix.
C
      CASE (4,5)
      CALL DGBSL(WM,LIN_PAR%MEBAND,NEQ,LIN_PAR%LML,
     *  LIN_PAR%LMU,LIN_PAR%IPVT),DELTA,0)
     RETURN
```

Fig. 10.

generic nonlinear or linear system is to be solved and are moved to sections of the code where the problem is described. If the object-oriented features of Fortran 90 were used fully to reimplement DASPK the interface could also be simplified. For example, if modules, user-defined types, and assumed shape and allocatable arrays were used the nonlinear solver interface could be simplified to

```
SUBROUTINE DNSD(X,Y,YPRIME,E,NON_LIN_PAR)
```

and the interface to the linear solvers could become as shown in Figure 10. We used a CASE statement to mimic the run-time polymorphism for linear solvers, but one might also create structures using pointers to mimic run-time polymorphism more elegantly [Decyk et al. 1997].

The code from `ddaspk.f` still demonstrates that good Fortran 77 code can be modular and implement polymorphism by controlling program flow. By encapsulating well-written Fortran 77 libraries in C++ *wrappers*, we can take advantage of this modularity to incorporate efficient legacy code into the software. The procedure for calling Fortran 77 code from C++ and calling C++ from Fortran 77 is described in Barton and Nackman [1994]. The methods are unfortunately platform dependent because some data types in C++ and Fortran 77 do not always have the same machine representations, and function names are not always represented in the same way on different compilers. On most UNIX platforms, however, the differences are usually minor, and a header file for interlanguage communication is sometimes provided. In our experience the cost in programmer time for mixed-language programming has been insignificant compared to the other details of porting and optimizing the code for new architectures.

In the `Integrator` hierarchy, DASPK makes use of the Fortran 77 routine `ddaspk.f`. In the same way, we use the LAPACK and LINPACK linear algebra routines in the `LinearSolver` hierarchy as well as the BLAS subroutines throughout the implementation for vector operations.

Vector and matrix algebra is implemented in two very different ways throughout the code. First, the code was designed using overloaded operators for vector and matrix operations such as addition and scalar- and matrix/vector-multiplication. We used "expression templates"[4] generated by the Portable Expression Template Engine (PETE) package to implement the overloaded operators efficiently. Since compilers which have not fully implemented the ANSI standard cannot always compile expression templates, we also rewrote expressions using overload BLAS routines which link to vendor-tuned BLAS libraries. The decision to compile with BLAS or overloaded operators is made with a simple preprocessor definition. We find that the overloaded operators are slightly faster for our architecture and our test problems, but the difference in efficiency is minor.

4.2.1 *Lessons Learned.* We found the following guidelines helpful for writing low-level code in C++: (1) avoid copying and constructing large objects such as vectors unnecessarily; (2) use inline functions to break apart complicated tasks while maintaining efficiency; (3) for function input arguments use call-by-const-reference for large objects and call-by-value for small objects; and (4) use efficient math-intrinsic functions (e.g., pow and sqrt). Guidelines (1)–(3) are suggestions that are simply considered good coding practice for C++. Tracing the calls to constructors and destructors is helpful for finding unnecessary copying and constructing of large objects

[4]See Veldhuizen, T., Blitz++. http://monet.uwaterloo.ca/blitz/.

during the optimization of a code. While profiling our code we discovered that large differences in efficiency were often caused by different implementations of intrinsic math functions. Detecting and rectifying these inefficiencies required a combination of profiling, examining the executables' symbol tables, and researching the compiler documentation for our architecture.

## 5. RESULTS AND DISCUSSION

We used a variety of DAE and ODE IVPs in evaluating the performance and accuracy of the toolkit. For this work we settled on two classes of problems: a 1996 test set of IVPs compiled at the Centrum voor Wiskunde en Informatica by van der Houwen et al. and the MOL solution of a PDE derived from a Richards' equation model for fluid flow in unsaturated porous media [Kelley et al. 1998].

### 5.1 IVP Test Set

The test problems from the 1996 IVP test set were the chemical Akzo Nobel, HIRES, Pollution, Ring Modulator, and EMEP problems. We verified that our code solved these to a variety of test tolerances and that it had similar run-time characteristics (function evaluations, Jacobian updates, and step and order selection histories) to DASPK by examining data collected by the `FullDataFileclass` in the `DataCollector` hierarchy. Since run-times were on the order of zero to 10 seconds, performance comparisons were not meaningful for these problems.

### 5.2 Richards' Equation

Our main test problem for performance comparisons was a stiff system of ODEs that we obtained by applying the MOL to the PDE:

$$[c(\psi) + S_s S_a(\psi)]\frac{\partial \psi}{\partial t} = \frac{\partial}{\partial z}\left[K_z(\psi)\frac{\partial \psi}{\partial z} + 1\right] \tag{21}$$

where

$$\theta(\psi) = \epsilon S_a \tag{22}$$

$$S_a(\psi) = (\theta_s - \theta_r)S_e(\psi) + \theta_r \tag{23}$$

$$c(\psi) = \frac{d\theta}{d\psi} \tag{24}$$

$$K_z(\psi) = K_s[S_e(\psi)]^{1/2}[1 - 1 - \{S_e(\psi)^{1/m}\}^m]^2 \tag{25}$$

$$S_e(\psi) = [1 + (-\alpha\psi)^n]^m \tag{26}$$

Table I. Compiler Options

| | |
|---|---|
| aCC | -Dhpux +DA2.0 +DS2.0 +Oall +Odataprefetch -DNDEBUG |
| KCC | -O4 -Bstatic +K3 --abstract_pointer --abstract_float --restrict -D_BUILTIN_MATH -DNDEBUG --backend +Oall --backend +Odataprefetch |
| f77 | -Dhpux +DA2.0 +DS2.0 +Oall +Odataprefetch +U77 |

and where $S_s$, $\alpha$, $\theta_s$, $\theta_r$, $m$, $n$, $\epsilon$, and $K_s$ are constants. This equation arises in a one-dimensional model of groundwater flow through the unsaturated zone and is referred to as a form of Richards' equation. If we apply a centered finite-difference approximation in space we obtain the system of ODEs:

$$A(\psi)\frac{\partial \psi_i}{\partial t} = \frac{K_{i+1/2}(\psi_{i+1} - \psi_i) - K_{i-1/2}(\psi_i - \psi_{i-1})}{\Delta z^2} + \frac{K_{i+1/2} - K_{i-1/2}}{\Delta z} \tag{27}$$

where

$$A(\psi) = c(\psi) + S_s S_a(\psi) \tag{28}$$

and $i = 1, \ldots, N$. We used discretizations of $N = 801$ and $N = 6401$ nodes and integration intervals of $[0,1e-2]$, $[0,3e-1]$, and $[0,1e-7]$ for comparison purposes. This discretization and formulation are the same as those used previously on this problem with a MOL solution implemented using DASPK [Kelley et al. 1998; Tocci et al. 1997].

We used a C++ and a Fortran 77 implementation of this DAE IVP for testing purposes. That is, we wrote two distinct child classes of `DaeDefinition` for this problem; one implements the definition of the DAE IVP entirely in C++, and the other calls a Fortran 77 routine for the `residual` function evaluation. The Fortran 77 `residual` routine can also be called directly by DASPK in order to provide a pure Fortran 77 baseline for comparisons. Since DAE-TK contains four `Integrator` implementations, we have a total of eight codes solving the same problem with DAE-TK and an additional code which uses only Fortran 77. We chose not to compare to a recent Fortran 90 version of DASPK because the Fortran 90 code was not developed for serial architectures and is, therefore, not as efficient as the original Fortran 77 code for our platform [Maier and Petzold 1993].

## 5.3 Testing Conditions

The test platform was a Hewlett-Packard 9000/780 workstation running HP-UX 10.20. We compiled all codes using both Hewlett-Packard's advanced C++ compiler (aCC) and Kuck and Associates' C++ compiler (KCC) with the most aggressive optimization options turned on (Table I). Both these compilers are publicly available.

Table II. Run Times

| Discretization → | 801,t = 1e-2 | | 801,t = 3e-1 | | 6401,t = 1e-7 | |
|---|---|---|---|---|---|---|
| Configuration ↓ | Total | Norm | Total | Norm | Total | Norm |
| DA/T/C++ | 59.16 | 0.92 | 608.97 | 0.96 | 32.33 | 0.87 |
| DA/T/F77 | 57.79 | 0.90 | 575.41 | 0.90 | 37.08 | 1.00 |
| DA/V/C++ | 59.96 | 0.93 | 615.38 | 0.97 | 32.54 | 0.88 |
| DA/V/F77 | 57.36 | 0.89 | 572.63 | 0.90 | 36.69 | 0.99 |
| FL/T/C++ | 70.92 | 1.10 | 745.98 | 1.17 | 41.62 | 1.12 |
| FL/T/F77 | 68.86 | 1.07 | 685.76 | 1.08 | 45.97 | 1.24 |
| FL/V/C++ | 71.60 | 1.11 | 734.16 | 1.15 | 41.76 | 1.12 |
| FL/V/F77 | 69.27 | 1.08 | 690.72 | 1.08 | 45.64 | 1.23 |
| F77 | 64.22 | 1.00 | 637.51 | 1.00 | 37.13 | 1.00 |

### 5.4 Test Results

Table II shows the total run-times in seconds and the normalized run-times for the Richards' equation example. We give only the results from the KCC-generated executables, since they were slightly more efficient than the aCC-generated executables and allowed the use of expression templates for operator overloading. The run-time of DASPK solving the Fortran 77 problem definition without using any of the DAE-TK code (the last row of the table) is taken as the benchmark. The stripped executable size for the pure Fortran 77 code is 176 kilobytes, while the DAE-TK executables were between 200 and 400 kilobytes. Each label in the left-hand column of Table II indicates the configuration of DAE-TK: the `Integrator` type (DA for DASPK or DASPKT, FL for FLCBDF or FLCBDFT), polymorphism implementation of the `Integrator-DaeDefinition` interface (T for compile time polymorphism using templates, V for run-time polymorphism using virtual functions), and problem definition language (C++ or F77).

5.4.1 *Low-Level Code Optimization*. The C++ and Fortran 77 implementations of the `residual` function in the `DaeDefinition` classes cause differences in run-time of only a few percent. It appears that the low-level code in the `DaeDefinition` is optimized well by both the Fortran 77 and C++ compilers and that significant gains are not made by using either language exclusively for low-level code although the Fortran 77 generally appears to be faster. However, for the largest problem the C++ low-level implementation was about 12% faster than the Fortran 77 implementation. The gains in efficiency are likely due to a compiler-specific optimization. As mentioned above, differences in intrinsic math functions are often the cause of differences in efficiency of low-level code.

5.4.2 *Implementing Polymorphism*. The use of virtual functions to support run-time polymorphism appears to have negligible run-time penalty when used at high levels. The lowest-level virtual function call in the */V/* configurations is the `residual` function containing the inner loop of the method, which contains two power function evaluations, a square root and several floating-point multiplications, divisions, and additions. It is called anywhere from 1 to 10 times per time-step of the BDF method. The */T/* configuration of the toolkit, in which the `residual` function is not virtual, does not show significant improvement in performance over the */V/* configuration, nor does the pure Fortran 77 implementation of the example outperform the DA/V/F77 configuration for this example. The latter uses DASPK and the Fortran 77 problem definition, but because the `Integrator` and problem definition are linked through C++ wrappers, the `residual` function is still virtual. This proves conclusively that the run-time penalty is negligible for the C++ supporting structure of DAE-TK.

5.4.3 *Integrator Implementations.* The `FLCBDF Integrator` is generally 5–20% slower than the pure Fortran 77 implementation for the aCC compiler configurations. Profiling shows that the inefficiency in the C++ version is not due to any single function or section of code. We showed in Section 5.4.1 that the OOP techniques have a negligible performance penalty at high levels but that the use of C++ might have a slight penalty at low levels. Thus, we suspect that the performance penalty arises from accumulation of inefficiencies at low levels caused by the C++ compiler. The compilers we used in this work have, in general, optimized better than older compilers, and we suspect this trend will continue as C++ compilers mature under standardization.

## 6. CONCLUSIONS

The structure of DAE-TK demonstrates how OOP can make implementations of complex methods easier to understand and modify than traditional procedural approaches. Large hierarchies of linear algebra and data collection classes verify that the code is easily extensible and more flexible than procedural codes. Highly optimized legacy libraries like LAPACK, LINPACK, and BLAS are easily exploited to extend the performance and the range of the code's linear algebra capabilities.

The compilers we tested were able to optimize C++ well enough that our translated C++ version of a FLC BDF method was, on average, no more than 20% slower than the original Fortran 77 version of the method. The C++ `residual` function appears to be only slightly less efficient than the Fortran 77 residual function in some cases.

The performance penalty of using run-time polymorphism was found to be negligible as compared to compile-time polymorphism or code without polymorphism, as long as its use remained outside the inner loops. Compile-time polymorphism using a template approach at a high level provided only slight performance benefits and adversely affected readability, code size, and compilation time. At low levels the compile-time polymorphism in the expression templates technique for matrix/vector algebra was slightly more efficient than vendor-tuned BLAS for these problems.

## CODE AVAILABILITY

DAE-TK and drivers for the test problems used in this work can be obtained from ftp://pavo.sph.unc.edu/chris kees/dae-tk.

ACKNOWLEDGMENTS

We thank the students, staff, and collaborators of the *Center for Multiphase Research* at the University of North Carolina for help with this work, especially Phillip B. Calvin, Mathew W. Farthing, and Joseph F. Kanney for experimenting with early versions of DAE-TK in their work and participating in discussions on its design.

## REFERENCES

ANDERSON, E., BAI, Z., BISCHOF, C., DEMMEL, J., DONGARRA, J., DU CROZ, J., GREENBAUM, A., HAMMARLING, S., MCKENNEY, A., OSTROUCHOV, S., AND SORENSEN, D. 1994. *LAPACK User's Guide Release 2.0*. SIAM, Philadelphia, PA. http://www.netlib.org/lapack/lug/lapack lug.html

ARGE, E. ET AL. 1997. On the numerical efficiency of C++ in scientific computing. In *Numerical Methods and Software Tools in Industrial Mathematics* Birkhäuser Boston Inc., Cambridge, MA, 91–118.

BARTON, J. J. AND NACKMAN, L. R. 1994. *Scientific and Engineering C++*. Addison-Wesley, Reading, MA.

BECK, R., ERDMANN, B., AND ROITZSCH, R. 1995. KASKADE 3.0: An object-oriented adaptive finite element model. Tech. Rep. TR 95-4. Konrad-Zuse-Zentrum fur Informationstechnik, Berlin, Germany.

BESSON, J. AND FOERCH, R. 1997. Large scale object-oriented finite element code design. *Comput. Methods Appl. Mech. Eng. 142*, 1-2, 165–187.

BRENAN, K., CAMPBELL, S., AND PETZOLD, L. 1996. *Numerical Solution of Initial-Value Problems in Differential-Algebraic Equations*. SIAM, Philadelphia, PA.

BROWN, P. N., HINDMARSH, A. C., AND PETZOLD, L. R. 1994. Using Krylov methods in the solution of large-scale differential-algebraic systems. *SIAM J. Sci. Comput. 15*, 6 (Nov. 1994), 1467–1488. Also available as a technical report from Lawrence Livermore National Laboratory

BRUASET, A. M. AND LANGTANGEN, H. P. 1997. Object-oriented design of preconditioned iterative methods in diffpack. *ACM Trans. Math. Softw. 23*, 1, 50–80.

CARY, J. R., SHASHARINA, S. G., CUMMINGS, J. C., REYNDERS, J. V., AND HINKER, P. J. 1997. Comparison of C++ and Fortran 90 for object-oriented scientific programming. *Comput. Phys. Commun. 105*, 1, 20–36.

DECYK, V. K., NORTON, C. D., AND SZYMANSKI, B. K. 1997. Expressing object-oriented concepts in Fortran 90. *SIGPLAN Fortran Forum 16*, 1, 13–18.

DONESCU, P. AND LAURSEN, T. A. 1996. A generalized object-oriented approach to solving ordinary and partial differential equations using finite elements. *Finite Elem. Anal. Des. 22*, 1, 93–107.

DONGARRA, J., BUNCH, J., MOLER, C., AND STEWART, P. 1984. Linpack. (Software). http://www.netlib.org/linpack

DONGARRA, J. J., LUMSDAINE, A., NIU, X., POZO, R., AND REMINGTON, K. 1994. A sparse matrix library in C++ for high performance architectures. In *Proceedings of the 2nd Annual Object-Oriented Numerics Conference* (OON-SKI '94, Sun River, OR, Apr.), 214–218.

DONGARRA, J., LUMSDAINE, A., POZO, R., AND REMINGTON, K. 1996. Iml++ version 1.2 iterative methods library reference guide.

DONGARRA, J. J., POZO, R., AND WALKER, D. W. 1993. LAPACK++: A design overview of object-oriented extensions for high performance linear algebra. In *Proceedings of the Conference on Supercomputing* (Supercomputing '93, Portland, OR, Nov. 15–19), B. Borchers and D. Crawford, Eds. IEEE Computer Society Press, Los Alamitos, CA, 162–171.

DUBOIS-PELERIN, Y. AND PEGON, P. 1997. Improving modularity in object-oriented finite element programming. *Commun. Numer. Methods Eng. 13*, 3, 193–198.

DUBOIS-PÈLERIN, Y., ZIMMERMANN, T., AND BOMME, P. 1992. Object-oriented finite element in programming: II. A prototype program in Smalltalk. *Comput. Methods Appl. Mech. Eng. 98*, 3 (Aug. 1992), 361–397.

EYHERAMENDY, D. AND ZIMMERMANN, T. 1996. Object-oriented finite elements: II. A symbolic environment for automatic programming. *Comput. Methods Appl. Mech. Eng. 132*, 3-4, 277–304.

GEAR, C. 1971. The simultaneous numerical solution of differential-algebraic equations. *IEEE Trans. Circuit Theory CT-18*, 89–95.

GEAR, C. W. 1973. Asymptotic estimation of errors and derivatives for the numerical solution of ordinary differential equations. Tech. Rep. 598. Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, IL.

GEAR, C. 1986. Differential-algebraic equation index transformations. University of Illinois at Urbana-Champaign, Champaign, IL.

GEAR, C., LEIMKUHLER, B., AND GUPTA, G. 1985. Automatic integration of Euler-Lagrange equations with constraints. *J. Comput. Appl. Math. 12/13* (May), 77–90.

JACKSON, K. R. AND SACKS-DAVIS, R. 1980. An alternative implementation of variable step-size multistep formulas for stiff ODEs. *ACM Trans. Math. Softw. 6*, 3 (Sept.), 295–318.

KARAMETE, B., TOKDEMIR, T., AND GER, M. 1997. Unstructured grid generation and a simple triangulation algorithm for arbitrary 2-D geometries using object-oriented programming. *Int. J. Num. Methods Eng. 40*, 2, 251–268.

KELLEY, C. 1995. *Iterative Methods for Linear and Nonlinear Equations*. SIAM, Philadelphia, PA.

KELLEY, C. T., MILLER, C. T., AND TOCCI, M. D. 1998. Termination of Newton/Chord iterations and the methods of lines. *SIAM J. Sci. Comput. 19*, 1, 280–290.

LEIMKUHLER, B. J. 1988. Approximation methods for the consistent initialization of differential-algebraic equations. Ph.D. Dissertation. University of Illinois at Urbana-Champaign, Champaign, IL.

LEWIS, R., MASTERS, I., AND CROSS, J. 1997. Automatic timestep selection for the super-time-stepping acceleration on unstructured grids using object-oriented programming. *Commun. Numer. Methods Eng. 13*, 3, 249–260.

LIU, J.-L., LIN, I.-J., SHIH, M.-Z., CHEN, R.-C., AND HSIEH, M.-C. 1996. Object-oriented programming of adaptive finite element and finite volume methods. *Appl. Numer. Math. 21*, 4, 439–467.

MAIER, R. S. AND PETZOLD, L. R. 1993. User's guide to DASPKMP and DASPKF90. Army High Performance Computing Research Center, Minneapolis, MN.

MALY, T. AND PETZOLD, L. R. 1996. Numerical methods and software for sensitivity analysis of differential-algebraic systems. *Appl. Numer. Math. 20*, 1-2, 57–79.

NEWCOMB, R. W. 1981. The semistate description of nonlinear and time-variable circuits. *IEEE Trans. Circ. Syst. 28*, 1, 203–216.

PETZOLD, L. 1983. *A Description of DASSL: A Differential/Algebraic System Solver*. North-Holland Publishing Co., Amsterdam, The Netherlands.

QUATRANI, T. 1998. *Visual Modeling with Rational Rose and UML*. Addison Wesley object technology series. Addison-Wesley Longman Publ. Co., Inc., Reading, MA.

ROBISON, A. D. 1996. C++ gets faster for scientific computing. *Comput. Physics 10*, 5, 458–462.

STROUSTRUP, B. 1997. *The C++ Programming Language*. 3rd ed. Addison-Wesley, Reading, MA.

TOCCI, M., KELLEY, C., AND MILLER, C. 1997. Accurate and economical solution of the pressure-head form of Richards' equation by the method of lines. *Adv. Water Resour. 20*, 1, 1–14.

ZIMMERMAN, T. AND EYHERAMENDY, D. 1996. Object-oriented finite elements I. Principles of symbolic derivations and automatic programming. *Comput. Methods Appl. Mech. Eng. 132*, 3-4, 259–276.

ZIMMERMANN, T., DUBOIS-PÈLERIN, Y., AND BOMME, P. 1992. Object-oriented finite element programming: I. Governing principles. *Comput. Methods Appl. Mech. Eng. 98*, 2 (July 1992), 291–303.

Received: July 1998; revised: June 1999 and September 1999; accepted: September 1999