Image Cover Sheet

CLASSIFICATION	SYSTEM NUMBER
UNCLASSIFIED	498409

TITLE

OBJECT ORIENTED PROGRAMMING OR REUSABLE APPLICATIONS: A MODELLING APPROACH

System Number:

Patron Number:

Requester:

Notes:

DSIS Use only:

Deliver to:

DEPARTMENT OF NATIONAL DEFENCE CANADA

OPERATIONAL RESEARCH AND ANALYSIS

DIRECTORATE OF AIR OPERATIONAL RESEARCH

RESEARCH NOTE

DAOR-RN-9603

OBJECT ORIENTED PROGRAMMING FOR REUSABLE APPLICATIONS: A MODELLING APPROACH

by

Dr. Gregory W Frank

APRIL 1996

OTTAWA, CANADA

National Defence Défense nationale

OPERATIONAL RESEARCH AND ANALYSIS

CATEGORIES OF PUBLICATION

ORA Reports are the most authoritative and most carefully considered publications of the ORA scientific community. They normally embody the results of major research activities or are significant works of lasting value or provide a comprehensive view on major defence research initiatives. ORA Reports are approved personally by DGOR, and are subject to peer review.

ORA Project Reports record the analysis and results of studies conducted for specific sponsors. This Category is the main vehicle to report completed research to the sponsors and may also describe a significant milestone in ongoing work. They are approved by DGOR or DORA and are subject to peer review. They are released initially to sponsors and may, with sponsor approval, be released to other agencies having an interest in the material.

Directorate Research Notes are issued by directorates. They are intended to outline, develop or document proposals, ideas, analysis or models which do not warrant more formal publication. They may record development work done in support of sponsored projects which could be applied elsewhere in the future. As such they help serve as the corporate scientific memory of the directorates.

ORA Journal Reprints provide readily available copies of articles published with ORA approval, by ORA researchers in learned journals, open technical publications, proceedings, etc.

ORA Contractor Reports document research done under contract for ORA agencies by industrial concerns, universities, consultants, other government departments or agencies, etc. The scientific content is the responsibility of the originator but has been reviewed by the scientific authority for the contract and approved for release by DGOR or DORA.

DEPARTMENT OF NATIONAL DEFENCE

CANADA

OPERATIONAL RESEARCH AND ANALYSIS

DIRECTORATE OF AIR OPERATIONAL RESEARCH

RESEARCH NOTE

DAOR-RN-9603

OBJECT ORIENTED PROGRAMMING FOR REUSABLE APPLICATIONS: A MODELLING APPROACH

by

Dr. Gregory W. Frank

Approved By: Roger L. Roy

Roger L. Roy, Director Air Operational Research

OTTAWA, CANADA

APRIL 1996

ABSTRACT

This Note discusses a potential paradigm for computer-based modelling and simulation activities within the Directorate of Air Operational Research. Adherence to Object Oriented Programming principles is advocated for modelling or simulation projects which are likely to be reused in follow-on or adjunct studies. Object Oriented Programming tenets of encapsulation, inheritance and polymorphism can increase analyst productivity by simplifying maintenance and modification tasks.

RÉSUMÉ

On discute d'un paradigme dans les activités de modélisation et de simulation informatisées de la Direction recherche opérationnelle (Air). L'adhérence aux principes de la programmation orientée par objets est préconisée pour les projets de simulation et de modélisation qui ont un potentiel d'être ré-utilisés dans d'autres analyses. Les principes polymorphique, d'enchaînement, et d'encapsulation de la programmation orientée par objets peuvent augmenter la productivité d'un analyste grâce à une simplification des taches lors des modifications et de l'entretien des programmes.

TABLE OF CONTENTS

OBJECT ORIENTED PROGRAMMING FOR REUSABLE APPLICATIONS: A MODELLING APPROACH

I. INTRODUCTION

BACKGROUND

1. The Directorate of Air Operational Research (DAOR) conducts research and analysis studies for elements within the Canadian Forces (CF) and Department of National Defence (DND). One of the most important research tools used in the directorate is computer simulation and modelling. This activity uses computer-based representations of real-world systems to estimate those system's characteristics. The system being modelled may be too complex, too costly, or too critical to permit study in any other form. The *modelling* process consists of constructing a computer-based description of the system and its behaviour. *Simulation* uses the computer to evaluate the model numerically and produce estimates of the system's actual characteristics (Reference [1]).

2. DAOR has a number of simulations which have been developed in-house or collected from other sources over the past few years. As the number of simulations has increased, so has their complexity, as more sophisticated algorithms and programming techniques have been developed. These advances are not always synonymous with better analysis: too often the result is more complex and more arcane computer code which the analyst does not understand and therefore does not trust.

3. The simulations and models used in DAOR generally fall into one of two broad categories, denoted by:

a. *Class A*: analysis tools developed for a single use in response to a specific issue, and;

b. *Class B*: simulation tools developed for multiple uses in response to general issues.

Both classes of tools arise naturally in the process of satisfying clients' taskings. Tools developed in response to a specific issue tend to be single use. These computer models are often rapidly developed, used for the specific study and not used again. In contrast, tools developed for general issues tend to be more complex, take longer to develop and test, and be re-used.

4. A problem arises when the same development process is used for both classes. Ideally, the development of a single-use program will differ greatly from that of a multiple-use application. Single use programs tend to be written quickly, without a great deal of consideration given to structure or modifications, and little or no documentation. Multiple use programs may take longer to write, have some thought given to structure and future modifications, and sufficient documentation to aid future developers.

5. The premise of this note is that computer-based tools which are carefully developed can be re-used. By identifying potential Class B applications early in the development cycle, a product may be developed which can be easily modified for future taskings. By advocating the careful development and maintenance of selected tools in a corporate database, model development times can be reduced and analyst efficiency boosted.

6. The concept of Object-Oriented Programming (OOP) has been developed over the past decade. It is a programming paradigm that exploits increases in computer power and processing speed to produce code that is easier to develop, use, and maintain. This Research Note will discuss the usefulness of an object-oriented approach to modelling of Class B applications within DAOR. Its advantages and disadvantages are outlined, and illustrated by a practical example.

AIM

7. This Research Note identifies the central issues concerning OOP as a modelling approach for re-usable applications within DAOR.

SCOPE

8. This note is intended as a summary of issues of interest in Operational Research as it is conducted on a day-to-day basis. It is not intended as a text or introduction to Object-Oriented Analysis, Design or Programming. Examples are taken from recent DAOR studies under DGOR Activities 23213 (Analysis of Space-Related and Space Systems) and Activity 23210 (System Requirement for Tactical Air Operations).

II. MODELLING PARADIGMS

INTRODUCTION

9. This section describes and contrasts OOP with traditional model development. Traditional model development follows the procedural software development method used extensively during the 1980s. The limitations of procedural modelling are described, and the alternative modelling paradigm offered by OOP presented. The section closes with an inventory of the strengths and weaknesses encountered with each method.

PROCEDURAL MODEL DEVELOPMENT

10. Traditionally, the activity of computer programming has been regarded as *procedural*. That is, a program is viewed as a sequence of step-by-step instructions. An application executes each statement in the literal order of its commands. The procedural approach was developed with the first electronic computers. Successive generations of computer hardware and software led to its evolution through a number of distinct phases, until the procedural approach reached its zenith in the 1980s.

11. In the 1960s, advances in computer hardware and software technology allowed large and complex programs to be developed for the first time. Until that time, programs tended to be developed in an ad hoc and unstructured manner, while programming languages had few provisions for streamlining the execution of complex tasks. By the 1970s, the discipline of software engineering had developed in response to this lack of global planning and organization. The software engineering approach emphasized planning and control of the development process. The notion of structured programming thus emerged: order is maintained by rigidly organizing data and code in separate channels. Data conforms to data structures, while program code is written as a set of modular functions and procedures.

12. Software engineering principles assumed as a premise that each application was unique, and thus should be developed from scratch. By the late 1980s, the complexity of programs had

again reached the stage that software development was becoming an increasingly costly activity. Modern applications, with their reliance on Graphical User Interfaces (GUIs) and constantly increasing complexity, pushed traditional procedural programming and software engineering to their limits.

OBJECT ORIENTED MODEL DEVELOPMENT

13. One potential resolution to this problem emerged from the academic environment in the late 1980s in the form of Object Oriented Programming (OOP). Since then, OOP has emerged as perhaps the most influential software engineering methodology of the 1990s. OOP is founded upon three architectural fundamentals:

a. *encapsulation*: the fusion of data and code into a single entity;

b. *inheritance*: the ability of descendant objects to inherit ancestors' functionality; and

c. *polymorphism*: the ability of a common method name to exhibit object-specific behaviour.

Encapsulation

14. OOP maintains the structured approach advocated by software engineering principles, carried a stage further. Procedural programming dictates the parallel development of data structures and the functions which act upon them. OOP extends this notion by allowing data and functions to be fused into a single programmatic entity, an *object*, which not only carries its own data but can operate upon it as well. This notion of *encapsulation* is the first tenet of OOP, and permits the development of extremely well-controlled and modular code. Many OOP languages, such as C++, allow an object's data to be protected by being unavailable to the non-object parts of a program. This allows extremely rigid control over accessibility to data, and greatly reduces the opportunities for important data to be changed accidentally.[1]

---

[1] Data encapsulation refers to the manner in which the program treats data and code as a unified object. The user still maintains control over the data, which can be supplied from external files or keyboard input. Thus classified data can be kept separate from computer programs, as in the case of a procedural model.

Inheritance

15. The second tenet of OOP is the idea of *inheritance*. In the same way that a procedural language allows specialized data structures to be defined and built up, OOP supports the inheritance of descendant objects from simpler ancestors. The advantage of OOP is that the ancestor's methods as well as its data are inherited. That is, a descendant object immediately inherits its ancestor's behaviours and functionality, as well as all of its data. This greatly simplifies the task of maintaining and reusing code.

16. OOP-designed code can be easily maintained: modifying the behaviour of a single ancestor object automatically modifies the behaviour of all descendants. In contrast, modifying procedural code is often a lengthy and careful exercise in bookkeeping, as each entry in a catalogue of data structures and functions is modified and checked. Objects can be reused: once a satisfactory class of objects has been developed to perform a set of basic tasks, they can easily be adapted or modified for other tasks. Each application need no longer start from scratch: data and functionality which is common across several applications need be developed only once.

Polymorphism

17. The third and final tenet of OOP is the concept of *polymorphism*.[2] While inheritance allows descendant objects to inherit their ancestor's functionality, mechanisms exist to over-ride ancestor functions in favour of specialized behaviours. Polymorphic objects can implement descendant's behaviour in unique ways.

Example: Object Oriented Graphics Hierarchy

18. To illustrate OOP principles, consider a graphics program which displays images on the computer screen. The ancestor object of all images is the object *Point*. A *Point* is a single pixel which can be lit or not. Encapsulation combines *Point's* data fields, consisting of (x,y) screen co-ordinates, with its methods, consisting of *Show* and *Hide* functions to switch that pixel on and off. Inheritance allows descendant objects to be defined which inherit *Point's* functionality. For example, *Line* and *Triangle* objects may be defined as descendants of *Point:* a *Line* is a set of two connected *Points* which may be lit or not. Polymorphism allows inherited methods to exhibit

---

[2] From the Greek, meaning 'many bodied' or 'many formed'.

specialized behaviours. All objects share *Show* and *Hide* functions which perform the same actions. However, each object shows its behaviour in a different way.

PROCEDURAL vs. OOP SOFTWARE LIFE CYCLE

19. Figure 1 shows the major steps in the procedural software development life cycle. The exact nature of the steps themselves will not be discussed here; full descriptions of each phase can be found in standard texts on software engineering (for references, see Reference [3]). In the procedural life cycle, steps are executed sequentially. This process makes it difficult to revisit steps once they have been completed. Modifying such a project is difficult, as there is no clear mechanism for inserting alterations without either restarting the project or neatly incorporating revisions into an existing application.

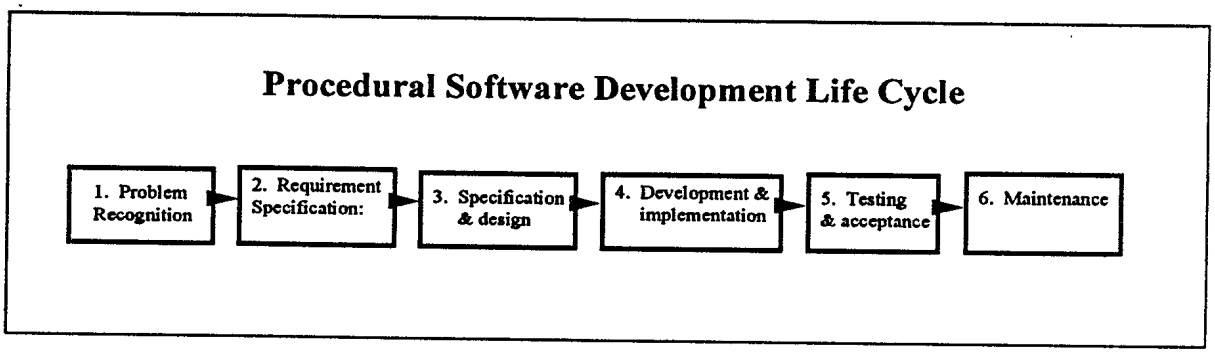

Figure 1. The procedural software development life cycle.

20. Figure 2 shows the corresponding development life cycle for object oriented software applications. Each step is analogous to its procedural counterpart, but with an explicit emphasis on the object oriented nature of each process. Because the tenets of OOP incorporate modularity and reuse, the designer is given much more flexibility in application development. Revisions can easily be made. As a project develops through each phase, any earlier phase can be revisited to incorporate modifications. By adhering to OOP principles, modifications are not disruptive and are less likely to have unforeseen side effects. Figure 2 shows one possible life cycle: an

application develops from lower to higher numbered phases. At any step a previous phase can be revisited.

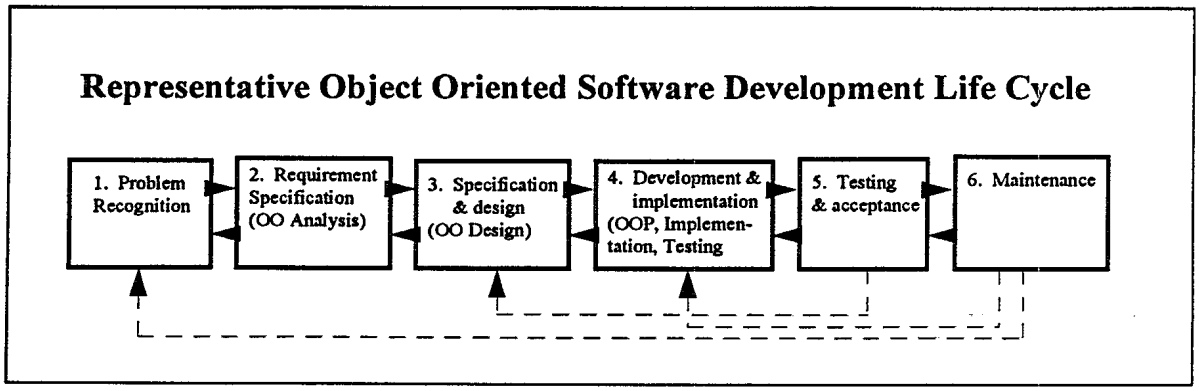

Figure 2. The object-oriented software development life cycle.

COSTS AND BENEFITS OF PROCEDURAL AND OOP MODELLING

21. This section identifies the main costs and benefits of each modelling paradigm. Figure 3 is a schematic of the effort required for each phase of the software development life cycle. Effort as a function of time is indicated for both procedural and OOP approaches.

22. *Problem recognition*. In neither paradigm is problem recognition a high-effort activity. While care must be taken in developing a problem statement, operations research applications are usually developed to address a specific problem which has already been clearly stated.

23. *Requirement specification*. Divergence usually begins at this stage of development. The procedural approach allows the developer to begin modelling with only a coarse idea of requirements. Some effort is necessary to develop software specifications which should solve the problem of phase one. In contrast, OOP development requires a significant effort at this stage as an object architecture is established, and high-level object functions and interactions are described.

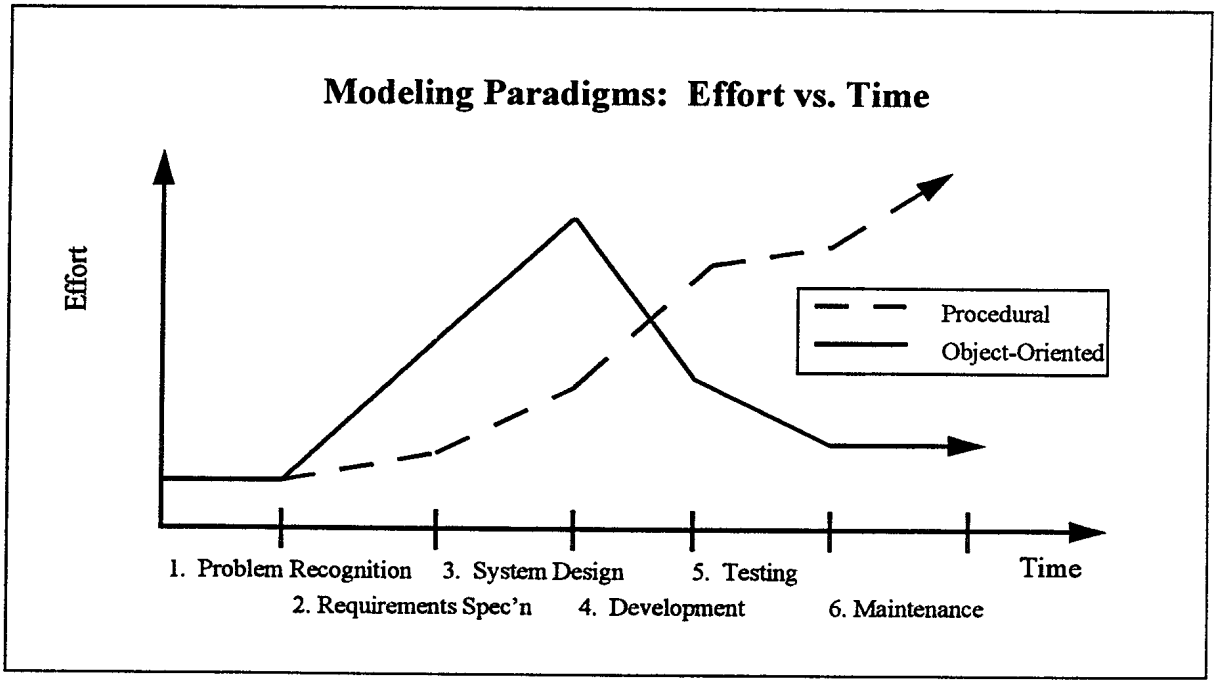

Figure 3. Effort vs. Time for Procedural and Object-Oriented Application Development

24. *System design.* Effort increases during the procedural system design process. This phase involves the translation of system requirements into basic data structures and program control functions. OOP development requires a substantial effort at this stage as objects are identified, data attributes and object methods are found, and interactions between objects examined.

25. *Development and implementation.* Previous to this stage, OOP development has involved more effort than procedural development. This trend often reverses at this step. Under procedural development, code must be carefully constructed, tested, and incorporated. The inevitable modifications to data structures and functions cause effort levels to increase greatly, owing to the difficulty of revisiting the design phase without compromising the remainder of the application. In contrast, OOP development often requires less effort than at previous steps. The effort spent in

developing object architecture, structure and interactions is often rewarded by faster and easier development.

26. *Testing and acceptance*. Testing inevitably reveals flaws from previous phases of development. Procedural models suffer at this stage from the difficulty of revisiting design and development phases. OOP models may again experience a reduction in effort: modularity and data privacy mean that modifications can be made without large-scale disruption. Effort expended during the design phase often means that objects have been carefully examined and tested before reaching this phase, and are less likely to require extensive modification.

27. *System maintenance*. Software engineering texts indicate that system maintenance often consumes more person hours than the rest of the development process. The rigidly sequential nature of procedural design makes maintenance and modification difficult if not impossible at this phase. The modularity and flexibility of OOP allows easier system maintenance, as objects can be surgically removed and modified without disrupting the remainder of the application. The ability to revisit earlier stages of the design phase makes system maintenance of a well-designed OOP application a moderately low-effort activity.

A MODELLING STRATEGY FOR DAOR

28. The above cost/benefit survey suggests a modelling strategy for DAOR which takes advantage of the strengths of both procedural and object oriented model development. Figure 3 argued that the development cycle of the generic procedural model is characterized by constantly increasing effort as a function of time. Later stages of the software development cycle typically require more effort than earlier stages. This suggests procedural methods be used for single-use applications, or Class A models. These can be rapidly developed, and may be sufficiently simple that even the high-effort tasks of development and testing need not be onerous. Further, single-use models have little or no need for maintenance.

29. Figure 3 also argued that object oriented model development is often characterized by high effort levels in the early stages. The effort required to develop a skeletal object hierarchy can be rewarded by faster progress in later stages of the software development cycle. OOP is therefore a feasible alternative for multiple-use applications, or Class B models. These models are often

complex, require a significant time to develop and test, and require some measure of maintenance when re-used. The initial effort to develop a coherent OOP approach leads to faster overall development, simpler re-use, and overall improved analyst efficiency.

30. OOP has emerged as the dominant software engineering methodology of the decade. Firms which have adopted it have often found a substantial increase in production. In comparison with procedural development, an carefully constructed OOP approach can more than double programmer productivity. Veteran OOP programmers can produce more accurate, better-written and more maintainable applications than their procedural counterparts (Reference [3]).

31. DAOR's reliance on modelling and simulation as tools to address sponsors' questions is likely to increase. The present budgetary and personnel realities lead to the requirement to adapt a more efficient and flexible model development mechanism, which emphasizes re-use when appropriate. OOP provides one such mechanism.

32. Annex A of this report demonstrates the OOP approach by introducing a modelling element common to many OR applications. A numerical integration algorithm is designed and implemented in C++ using the principles discussed. By designing with a view to re-use, the numerical integration object can easily be modified to accommodate specific studies. Two examples are taken from recent DAOR projects: a ballistic missile model which solves Newton's equations of motion for a point mass in orbit about a uniform spherical body; and an air-to-air missile model which solves the equations of motion for a mass-burning missile moving through a model atmosphere.

IV. CONCLUSION

33. This Research Note has discussed the potential of an Object Oriented Programming approach to reusable application development within the Directorate of Air Operational Research. Object Oriented Programming, or OOP, has emerged from the academic community to become perhaps the pivotal software engineering paradigm of the decade. The tenets of traditional procedural program design are carried a step further by combining data and behaviours within a single programmatic entity.

34. OOP offers the advantage of low maintenance and rapid modification of existing packages, making it an attractive strategy for reusable applications. The primary disadvantage of the paradigm is in the longer project start-up time and the time required to develop familiarity with OOP concepts. The effect of this disadvantage can be reduced by using OOP only in high-value applications likely to benefit from OOP by reuse.

35. The present personnel and budgetary realities stress the need for careful planning at the early stages of project development. OOP is one possible mechanism for allowing Operational Research analysts to "work smarter - not just harder".

V. REFERENCES

1. Law, A.M., W.D. Keller, *Simulation Modelling and Analysis,* Second Edition, McGraw-Hill (New York 1991).
2. Press, W.H., B.P. Flannery, S.A. Teukolsky, W.T. Vetterling, *Numerical Recipes, The Art of Scientific Computing*, Cambridge University Press (New York 1986).
3. Perry, G., *Teach Yourself Object-Oriented Programming with Turbo C++ in 21 Days,* SAMS Publishing (Indianapolis Indiana 1993).
4. Frank, G.W., TMDPA*K: A Theatre Missile Defence Simulation Package (Technical Documentation and User's Guide)* Directorate of Air Operational Research Note 94/1.
5. Frank, G.W., *Analysis of AIM-7M versus AA-10C (Long-Range ALAMO) in Beyond Visual Range Engagement (U),* Operational Research and Analysis Project Report PR 9509 (December 1995) (SECRET).
6. Vinh, N.X., A. Busemann, R.D. Culp, *Hypersonic and Planetary Entry Flight Mechanics,* University of Michigan Press (Ann Arbor 1980).

ANNEX A: AN OBJECT-ORIENTED NUMERICAL INTEGRATION ALGORITHM

A-1. This annex describes a mathematical algorithm cast using OOP design principles. The algorithm chosen is a numerical integration procedure for sets of Ordinary Differential Equations (ODEs). ODEs provide mathematical descriptions of changes in system states with time, and are the mathematical laws governing the temporal evolution of many systems of interest. The numerical procedure used is a fourth-order Runge Kutta, with built in error control and self-adapting step size.

RUNGE-KUTTA ALGORITHM

A-2. Consider a set of n first-order ODEs:

$$\frac{d}{dt}x_i(t) = f_i(t,x_1,\ldots,x_n), i = 1,\ldots,n. \qquad \text{(A.1)}$$

Assume the functions f_i on the right hand side are known and satisfy certain minimal criteria of smoothness and continuity. Assume also the system's initial state at $t=0$ is known. The solution of the ODE is the set of dependent variables $\{x_i\}$ parameterized as a function of the independent variable t. For example, the governing equations may be Newton's equations of gravitation governing a satellite in orbit about the Earth. The independent variable is time, the dependent variables are position and velocity. Initial conditions may be the position and velocity at orbital insertion. The solution is either an analytical or numerical form for position and velocity as a function of time.

A-3. Numerical solutions of equations (A.1) estimate analytic solutions by approximating the continuous differentiation process by a sequence of discrete steps. Assume that the solution of (A.1) has been found at $t=t_0$ as $x=x_0$. The Runge-Kutta numerical integration algorithm applied

over a step of size h will produce a numerical solution at $t=t_0+h$. The numerical solution is approximated by the following sequence (Reference [2]):

$$\begin{aligned} k_1 &= hf(t_0, x_0), \\ k_2 &= hf(t_0 + \frac{h}{2}, x_0 + \frac{k_1}{2}), \\ k_3 &= hf(t_0 + \frac{h}{2}, x_0 + \frac{k_2}{2}), \\ k_4 &= hf(t_0 + h, x_0 + k_3), \end{aligned} \tag{A.2}$$

$$x_1 = x_0 + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4) + O(h^5).$$

To avoid a proliferation of subscripts, a one-dimensional case (n=1) is shown. It can be shown that, under sufficiently general conditions, the numerical solution equals the analytic solution to within an error on the order of h^5.

A-4. The algorithm described here uses an adjustable step size to control the truncation error. This is accomplished by comparing the approximate solutions produced by step sizes $2h$ and h, denoted by x_1 and x_2 respectively. If the analytical solution is $x(t+2h)$, then

$$\begin{aligned} x(t+2h) &= x_1 + (2h)^5 C + O(h^6), \\ x(t+2h) &= x_2 + 2h^5 C + O(h^6), \end{aligned} \tag{A.3}$$

where C is a constant over the interval, to order h. This provides an estimate of truncation error via

$$\Delta = x_2 - x_1. \tag{A.4}$$

Equations (A.3, A.4) can be used to extend the accuracy of the numerical integration to fifth order by removing the constant term on the right hand side of equations (A.3):

$$x(t+2h) = x_2 + \frac{\Delta}{15} + O(h^6). \tag{A.5}$$

Equation (A.5) is a computationally efficient way of extending the algorithm's accuracy.

A-5. Equations (A.3,A.4) indicates how the step size should be adjusted to maintain the estimated truncation error at a set level. Since Δ scales as h^5, if a step h_0 produces an error Δ_1, then the step size h_1 that would have produced an error Δ_1 is estimated as

$$h_1 = h_0 \left| \frac{\Delta_1}{\Delta_0} \right|^{1/5}. \tag{A.6}$$

NUMERICAL INTEGRATION CODE

A-6. The following section documents C++ computer code developed to demonstrate OOP principles.[2] The following objects are defined:

a. Dependent variable object (Dependent_variables);

b. Single integration step at constant step size (Runge_kutta);

A-7. The following constants and header files are used:

```
const int nmax = 10;
const double tiny = 1.0e-10;

#include <iostream.h>
#include <iomanip.h>
#include <math.h>
#include <fstream.h>
```

For the purposes of this demonstration, dynamical systems with a maximum of `nmax = 10` degrees of freedom can be modelled. The constant `tiny = 10^-10` is a machine zero: floating point numbers differing by less than this amount will be considered as equal.

[2] This section is not a C++ tutorial; Reference [3] is a comprehensive introduction to this versatile programming language.

Class Dependent_variables

A-8. A class Dependent_variables is introduced to encapsulate the set of dependent variables. A basic set of methods define constructor, destructor, field value assignment and retrieval, and overloaded arithmetic operators. A collection of Set and Get methods allow access to data members while enforcing the data privacy required by encapsulation principles. An overloaded output function permits customized reports to the screen or data files. Table A-1 summarizes the object's members and methods. The coding of the Dependent_variables class appears below.

TABLE A-1

DEPENDENT_VARIABLES MEMBER/METHOD TABLE

Name	DEPENDENT_VARIABLES		
File	ODEINT.CPP		
Members	Field	Type	Comment
	n	Integer	No. dependent variables
	x	Double[nmax]	Array of dep. variables' values
Methods	Function	Returns	Comment
	Dependent_variables	Constructor	Constructor: two argument sets provided: (1) void: returns 0-dimensional array of 0 values; (2) int I: returns i-dimensional array of 0-values.
	~Dependent_variables	Destructor	Standard destructor.
	operator=	Dependent_variables	Componentwise assignment.
	operator+	Dependent_variables	Componentwise addition.
	operator-	Dependent_variables	Componentwise subtraction.
	setn	Void	Sets value n.
	getn	Integer	Returns value n.
	set	Void	Input integer i, double d; sets x[i] = d.
	get	Double	Input integer I, returns x[i].
	operator*	Dependent_variables	Friend function: scalar multiplication.
	operator<<	friend ostream	Overloaded output operator.
Description: Object is array of NMAX double initialized to 0.0. User provided n<=NMAX components, overloaded (vector) addition, subtraction, scalar multiplication.			

```
//  ******************************************************
//  *                                                    *
//  *       Class DEPENDENT_VARIABLES                    *
//  *                                                    *
//  ******************************************************
//
//  Simple object is an array of NMAX double.  Holds
//  dependent variables during integration.
//
//  Fields:
//                n   integer      Number of dependent variables
//                x   double    Array holding values of variables
//
//  Methods:
//                Constructor, destructor, set, get, overloaded operators.


class Dependent_variables
{
  private:
    int n;
    double x[nmax];
  public:
    Dependent_variables(void);
    Dependent_variables(const int & i);
    ~Dependent_variables(void);
    Dependent_variables & operator=(const Dependent_variables & D);
    Dependent_variables operator+(const Dependent_variables & v);
    Dependent_variables operator-(const Dependent_variables & v);
    void setn(const int & i);
    void set(const int & i,const double & d);
    double get(const int & i);
    int  getn(void);
    void report(void);
    friend Dependent_variables operator*(const float & f,
                        Dependent_variables & v);
    friend ostream & operator<<(ostream & out, Dependent_variables D);
};



Dependent_variables::Dependent_variables(void)
{
  n = 0;
  for (int i=0;i<nmax;i++) {x[i] = 0.0;}
}
//        Default constructor assigns zero values, dimension n=0.



Dependent_variables::Dependent_variables(const int & i)
{
  n = i;
  for (int j=0;j<nmax;j++) {x[j] = 0.0;}
}
//        Constructor assigns zero values, dimension n=i.
```

```
Dependent_variables::~Dependent_variables(void)
{
  // No code required.
}


void Dependent_variables::setn(const int & i)
{
  n = i;
}
// Set number of dependent variables to i.


int Dependent_variables::getn(void)
{
  return n;
}
// Return number of dependent variables.


void Dependent_variables::set(const int & i, const double & d)
{
  x[i] = d;
  return;
}
// Set ith component equal to d.


double Dependent_variables::get(const int & i)
{
  return x[i];
}
// Get ith component x[i].


void Dependent_variables::report(void)
{
  cout << "\n**********************************\n";
  cout << "*    DEPENDENT_VARIABLES REPORT:    \n";
  cout << "* n = " << n << "\n";
  cout << "* i, x[i]:  \n";
  for (int i=0;i<nmax;i++)
  {
    cout << "* " << i << "   " << x[i] << "\n";
  }
  cout << "**********************************\n\n";
}
```

```
Dependent_variables & Dependent_variables::operator=(const Dependent_variables &
D)
{
  if (this == &D) {return *this;}
  n = D.n;
  for (int i=0;i<nmax;i++) {x[i] = D.x[i];}

  return *this;
}
// Overloaded assignment.



Dependent_variables Dependent_variables::operator+(const Dependent_variables & v)
{
        Dependent_variables tempv(n);
        float tempf;

        for (int i=0; i<v.n; i++)
        {
          tempf = x[i] + v.x[i];
          tempv.x[i] = tempf;
        }
        return(tempv);
}


Dependent_variables Dependent_variables::operator-(const Dependent_variables & v)
{

        Dependent_variables tempv(n);
        float tempf;

        for (int i=0; i<v.n; i++)
        {
          tempf = x[i] - v.x[i];
          tempv.x[i] = tempf;
        }
        return(tempv);
}



Dependent_variables operator*(const float & f, Dependent_variables & v)
{
  Dependent_variables tempv(v.n);
  for (int i=0;i<v.n;i++)
  {
    tempv.x[i] = f*v.x[i];
  }
  return(tempv);
}
// Vector multiplication: friend function is used for multiplication
// by scalar from left.
```

```
ostream  & operator<<(ostream & out, Dependent_variables D)
{
  for (int i=0;i<D.n;i++)
  {
     out << setw(16) << setiosflags(ios::right) << D.x[i];
  }
  return out;
}
```

Class Runge_kutta

A-8. The class Runge_kutta performs a numerical integration at constant step size over a single step. The object integrates equations (A.1) using the discretization method of (A.2). C++ code appears below. Note how the right hand side of the governing equations (A.1) is provided as a virtual method: the default shown is for constant acceleration: ancestor objects can over-ride this method to provide specialized behaviour. Note also how functionality is provided: the class is shown how to integrate itself in the function RK4.

TABLE A-2

RUNGE_KUTTA MEMBER/METHOD TABLE

Name	**RUNGE_KUTTA**		
File	**ODEINT.CPP**		
Members	**Field**	**Type**	**Comment**
	n	Integer	Number dependent variables (n<=nmax)
	x	Double	Independent variable.
	y	Dependent_variables	Dependent variable array at x.
	dydx	Dependent_variables	Array of derivatives of y wrt x.
	h	Double	Solution interval.
	yout	Dependent_variables	Dependent variables at (x+h).
Methods	**Function**	**Returns**	**Comment**
	Runge_kutta	Constructor	Two constructors: (1) no arguments, returns all members set of zero; (2) argument list N,X,Y,H.
	derivs	void	Evaluates dydx for given values of (x,y)
	RK4	void	Advance solution across incremental interval.
	report	void	Detailed report of object values to screen.
	~Runge_kutta	Destructor	Standard destructor.
	operator<<	friend ostream	Overloaded output.
Description: Object integrates governing equations across interval (x,x+h) in a single application of fourth-order Runge-Kutta algorithm. derivs method contains RHS of governing equations (default = constant acceleration); overload this method to customize descendant objects.			

```
//  ********************************************************
//  *                                                      *
//  *              Class RUNGE_KUTTA                       *
//  *                                                      *
//  ********************************************************


class Runge_kutta
{
  protected:
    int n;                      // Number of dependent variables, n < nmax.
    double x;                // Independent variable.
    Dependent_variables y;   // Array of dependent variables.
    Dependent_variables dydx;// Array of derivatives of y evaluated at x.
    double h;                   // Solution interval.
    Dependent_variables yout; // Values of y(x+h).
  public:
    Runge_kutta(void);
    Runge_kutta(const int & N, const double & X,
            const Dependent_variables Y, const double & H);
    virtual void derivs(void);
    void RK4(void);       // Advance solution across incremental interval.
    void report(void);
    ~Runge_kutta(void);
    friend ostream & operator<<(ostream & out, Runge_kutta D);
};



Runge_kutta::Runge_kutta(void)
{
  n = 2;
  x = 0.0;
  h = 0.1;

  // Note:  y,dydx,yout are initialized to zero via default constructors.
}



Runge_kutta::Runge_kutta(const int & N, const double & X, const Dependent_variables
Y,
                  const double & H)
                  : n(N),x(X),y(Y),h(H),yout(N)
{
  // Code required only to initialize DYDX from DERIVS data type.
  // Initialize DYDX:
  dydx.setn(n);
  derivs();
}
```

```
void Runge_kutta::derivs(void)
{
    // Returns RHS of governing equations
    dydx.set(0,y.get(1));
    dydx.set(1,1.0);

//    dydx.x[0] = y.x[1];
//    dydx.x[1] = 1.0;
}



void Runge_kutta::report(void)
{
  cout << "\n";
  cout << "*************************************\n";
  cout << "*    RUNGE_KUTTA Report:            \n";
  cout << "*\n";
  cout << "*   n: "   << n    << "\n";
  cout << "*   x: "   << x    << "\n";
  cout << "*   y: "   << y    << "\n";
  cout << "*   y':"   << dydx << "\n";
  cout << "*   h: "   << h    << "\n";
  cout << "*   yout:" << yout << "\n";
  cout << "*************************************\n\n";
}



void Runge_kutta::RK4(void)
{
  // Introduce fractional intervals to reduce runtime:
  double h2 = h/2.0;
  double h6 = h/6.0;

  // Temporary storage.
  Dependent_variables ytemp(n),dydxtemp(n),dydxtemp2(n);
  Dependent_variables yorig(n),dydxorig(n);
  double xorig;
  double xtemp;

  yorig = y;
  dydxorig = dydx;
  xorig = x;

  // First step:  Linear extrapolation from LH endpoint to midpoint.
  xtemp = xorig + h2;
  ytemp = yorig + h2*dydx;

  // Second step:  Refined extrapolation to midpoint.
  x = xtemp;
  y = ytemp;
  derivs();
  ytemp = yorig + h2*dydx;
  dydxtemp = dydx;
```

```
    // Third trial step, halfway across interval.
    x = xtemp;
    y = ytemp;
    derivs();
    ytemp = yorig + h*dydx;
    dydxtemp2 = dydx;

    // Temporary storage.
    dydxtemp2 = dydxtemp + dydxtemp2;

    // Fourth trial step across interval.
    x = x+h;
    y = ytemp;
    derivs();
    dydxtemp = dydx;

    yout = yorig + h6*(dydxorig + dydxtemp + 2.0*dydxtemp2);

    x = xorig;
    y = yorig;
    dydx = dydxorig;
}


Runge_kutta::~Runge_kutta(void)
{
  // No code required.
}

ostream & operator<<(ostream & out, Runge_kutta R)
{
  out << "Number of dependent variables: " << R.n << "\n";
  out << "Value of independent variable: " << R.x << "\n";
  out << "Step size: " << R.h << "\n";

  out << "Y: " << R.y;
  out << "DYDX: " << R.dydx;
  out << "YOUT: " << R.yout;

  return out;
}
```

Class Runge_kutta_dumb

A-9. Class Runge_kutta_dumb is a descendant of class Runge_kutta. This class incorporates the basic integration algorithm into a simple Runge Kutta driver. The set of governing equations are integrated across a single time step without regard to integration accuracy or stepsize control. This class is primarily a development step while constructing a more sophisticated algorithm, and is unlikely to be used in practice.

TABLE A-3

RUNGE_KUTTA_DUMB MEMBER/METHOD TABLE

Name	RUNGE_KUTTA_DUMB		
File	ODEINT.CPP		
Members	**Field**	**Type**	**Comment**
	x_initial	Double	Initial independent variable value.
	x_final	Double	Final independent variable value.
	steps	Integer	Number of uniform steps over interval.
	y_initial	Dependent_variables	Array of initial dependent variable values.
Methods	**Function**	**Returns**	**Comment**
	Runge_kutta_dumb	Constructor	Multiple constructors: (1) void arguments, returns zero values; (2) arguments (XI,XF,S,YI,N).
	pushforward	void	Performs STEP RK4 procedures over interval.
	~Runge_kutta_dumb	Destructor	Standard destructor.
Description: Object integrates governing equations across interval (x,x+h) in multiple applications of fourth-order Runge-Kutta algorithm. Direct descendant of Runge_kutta object.			

```
// ************************************************
// *                                              *
// *        Class RUNGE_KUTTA_DUMB                *
// *                                              *
// ************************************************
//
// Fourth-order Runge-Kutta with constant stepsize.  Direct adaptation
// of and heir of Class Runge_kutta.  Source: Numerical Recipes pp 550-554.



class Runge_kutta_dumb: Runge_kutta
{
  protected:
    double x_initial;        // Initial value of independent variable.
    double x_final;          // Final value of independent variable.
    int steps;              // Number of steps over interval.
    Dependent_variables y_initial;// Initial value of dependent variables.
  public:
    Runge_kutta_dumb(void);
    Runge_kutta_dumb(const double & XI,const double & XF,const int & S,
                     const Dependent_variables & YI, const int & N);
    void pushforward(void);
    ~Runge_kutta_dumb(void);
    friend ostream & operator<<(ostream & out, Runge_kutta_dumb R);
};



Runge_kutta_dumb::Runge_kutta_dumb(void): Runge_kutta(),x_initial(0.0),
                                          x_final(0.0),steps(0),y_initial()
{
  // No code required.
}



Runge_kutta_dumb::Runge_kutta_dumb(const double & XI,const double & XF,const int
& S,
                     const Dependent_variables & YI,const int & N)
                     : x_initial(XI), x_final(XF), steps(S), y_initial(YI),
                     Runge_kutta(N,XI,YI,(XF-XI)/S)
{
  // No code required.
}
```

```
void Runge_kutta_dumb::pushforward(void)
{
  x = x_initial;
  y = y_initial;
  for (int i=0;i<steps;i++)
  {
     derivs();
     RK4();
     x = x+h; y = yout;
  }
}
// Dumb RK4 driver using fixed step size.  Integrates across an interval
// from x_initial to x_final in steps equal intervals.  Output: YOUT
// contains values y(x_final).


Runge_kutta_dumb::~Runge_kutta_dumb(void)
{
  // No code required.
}


ostream & operator<<(ostream & out, Runge_kutta_dumb D)
{
  out << "Initial value: " << D.x_initial << "\n";
  out << "Final value  : " << D.x_final << "\n";
  out << "Steps:         " << D.steps   << "\n";

  out << "Number of dependent variables: " << D.n << "\n";
  out << "Step size: " << D.h << "\n";

  out << "Y initial: " << D.y_initial;
  out << "DYDX: " << D.dydx;
  out << "YOUT: " << D.yout;

  return out;
}
```

Class Runge_kutta_smart

A-10. Class Runge_kutta_smart is also a descendant of class Runge_kutta. This class incorporates the basic integration algorithm into a more sophisticated Runge Kutta driver. The set of governing equations are integrated across a single time step while monitoring step size and estimated integration accuracy. Truncation error from the fourth-order routine is estimated, giving fifth-order accuracy.

TABLE A-4

RUNGE_KUTTA_SMART MEMBER/METHOD TABLE

Name	RUNGE_KUTTA_SMART		
File	ODEINT.CPP		
Members	**Field**	**Type**	**Comment**
	h_actual	double	Actual stepsize used.
	h_next	double	Suggested stepsize for next iteration.
	accuracy	double	Required integration accuracy.
	y_error	Dependent_variables	Error scaling array.
	y_best	Dependent_variables	Best solution so far.
	y_scale	Dependent_variables	Scaling array for intermediate error estimates.
	ok	Integer	Quality check: 1 = pass; 0 = failure.
Methods	**Function**	**Returns**	**Comment**
	Runge_kutta_smart	Constructor	Multiple constructors: (1) void: returns zero arguments; (2) arguments (N,X,Y,H,A).
	test	void	Test current candidate solution.
	pushforward	void	Integrate initial conditions across interval.
	report	void	Detailed member report to screen.
	~Runge_kutta_smart	Destructor	Standard destructor.
	operator<<	friend ostream	Overloaded output operator.
Description: Object integrates governing equations across interval (x,x+h) using fourth-order Runge-Kutta algorithm with adaptive stepsize. Direct descendant of Runge_kutta object.			

```
// ********************************************************
// *                                                      *
// *        Class RUNGE_KUTTA_SMART                       *
// *                                                      *
// ********************************************************
//
// Fifth-order Runge-Kutta with adaptive stepsize. Direct adaptation
// of and heir of Class Runge_kutta. Source: Numerical Recipes pp 550-
// 554.


class Runge_kutta_smart : public Runge_kutta
{
  protected:
    double h_actual;      // Actual stepsize used.
    double h_next;                // Suggested stepsize for next iteration.
    double accuracy;      // Required accuracy for stepsize.
    Dependent_variables y_error;// Error scaling vector.
    Dependent_variables y_best;// Best solution to date.
    Dependent_variables y_scale;// Scaling constants.
    int ok;                       // Set to '1' if step passes quality control.
  public:
    Runge_kutta_smart(void);
```

```
    Runge_kutta_smart::Runge_kutta_smart(const int & N, const double & X,
                    const Dependent_variables Y,const double & H,
                    const double & A);
    void test(void);
    void pushforward(void);
    void report(void);
    ~Runge_kutta_smart(void);
    friend ostream & operator<<(ostream & out, Runge_kutta_smart D);
};


Runge_kutta_smart::Runge_kutta_smart(void)
                  :Runge_kutta(),h_actual(0.0),h_next(0.0),accuracy(0.0),
                   ok(0),y_error(),y_best(),y_scale()
{
  // No code required.
}


Runge_kutta_smart::Runge_kutta_smart(const int & N, const double & X,
                   const Dependent_variables Y,
                   const double & H,
                   const double & A)
                  :Runge_kutta(N,X,Y,H),
                  h_actual(H),
                  h_next(H),
                   accuracy(A),ok(0),y_error(N),y_best(N),y_scale(N)
{
  Dependent_variables t1(N);
  for (int i=0; i<N; i++) {t1.set(i,1.0);}
  y_scale = t1;
}


void Runge_kutta_smart::report(void)
{
  cout << "\n*************************************\n";
  cout << "*  RUNGE_KUTTA_SMART REPORT \n";
  cout << "*  h_actual: " << h_actual << "\n";
  cout << "*  h_next:   " << h_next   << "\n";
  cout << "*  accuracy: " << accuracy << "\n";
  cout << "*  y_error:  " << y_error  << "\n";
  cout << "*  y_best:   " << y_best   << "\n";
  cout << "*  y_scale:  " << y_scale  << "\n";
  cout << "*  ok:       " << ok       << "\n";
  cout << "*************************************\n\n";
}
```

```
void Runge_kutta_smart::test(void)
{
  // Phase 1:  Step across interval in two half-steps.

  double x_original = x;
  Dependent_variables y_original = y;

  h = h_next/2.0; derivs();
  RK4(); x = x+h; y = yout; derivs();
  RK4(); x = x+h; y = yout;

  // Save resulting dependent variable values.
  Dependent_variables y1 = y;

  // Phase 2:  Step across interval in one step.
  x = x_original;
  y = y_original;
  h = h_next;derivs();
  RK4(); x = x+h; y = yout;

  // Phase 3:  Evaluate accuracy:
  double error_max = -1.0e10;
  for (int i=0;i<n;i++)
  {
        y_error.set(i,abs(yout.get(i) - y1.get(i)));
        if (y_error.get(i) > error_max) {error_max = y_error.get(i);}
        if (error_max < abs(y_error.get(i)/y_scale.get(i)))
        {error_max = abs(y_error.get(i)/y_scale.get(i));}
  }

  error_max = error_max/accuracy;

  // Phase 4:  If state of sin exists, rescale.
  if (error_max > 1.0)
  {
    double safety = 0.90, shrink = -0.25;
    h_next = safety * h_actual * pow(error_max,shrink);
  }
  else
  {
    ok = 1;
    y_best = y;
    h_next = h_actual;
  }

  // Tidy things up.
  y = y_original;
  x = x_original;
}
```

```
void Runge_kutta_smart::pushforward(void)
{

  // Test interval for appropriate step size.
  // Adjust if necessary.

  int loopcond = 0,loopcount = 0;
  while (loopcond == 0)
  {
    test();
    if (ok ==1 || loopcount > 0) {loopcond = 1;}
    else {loopcount = loopcount + 1;}
  }

  // Fifth-order truncation error.
  x = x + h_actual;
  y = y_best + (0.06666666667)*y_error;
}

Runge_kutta_smart::~Runge_kutta_smart(void)
{
  // No code required.
}



ostream & operator<<(ostream & out, Runge_kutta_smart D)
{
  out << "Class RUNGE_KUTTA_SMART: \n";
  out << "Step size used: " << D.h_actual << "\n";
  out << "Next step size: " << D.h_next << "\n";
  out << "Accuracy:       " << D.accuracy   << "\n";
  out << "Error in y:     " << D.y_error << "\n";
  out << "Best solution:  " << D.y_best << "\n";
  out << "----------------------------------------\n";
  out << "Y: " << D.y;
  out << "DYDX: " << D.dydx;
  out << "YOUT: " << D.yout;
  out << "----------------------------------------\n";

  return out;
}
```

Class Runge_kutta_driver

A-11. Class Runge_kutta_driver is a descendant of class Runge_kutta_smart. The adaptive integration algorithm is incorporated into a driver which steps across a larger interval in a set of smaller, quality controlled, steps. This class is the base class for user applications requiring numerical integration of governing sets of ODEs.

TABLE A-5

RUNGE_KUTTA_DRIVER MEMBER/METHOD TABLE

Name	RUNGE_KUTTA_DRIVER		
File	ODEINT.CPP		
Members	Field	Type	Comment
	x_in	Double	Initial independent variable value.
	x_out	Double	Final independent variable value.
	h_min	Double	Minimum allowable step size.
	step	Double	Interval width.
	y_in	Dependent_variables	Initial dependent variable values.
	y_out	Dependent_variables	Final dependent variable values.
	good_step	Integer	No. accepted integration steps.
	bad_step	Integer	No. Rejected integration steps.
Methods	Function	Returns	Comment
	Runge_kutta_driver	Constructor	Multiple constructors: (1) void, returns zero members; (2) arguments (N,X,Y,H,A,XIN,HMIN); (3) arguments (N,X,Y,S).
	drive	void	Integrate across STEP.
	update	void	Prepare member fields for next integration.
	report	void	Detailed member value output to screen
	operator<<	friend ostream	Overloaded output operator.
Description: Object integrates governing equations across interval (x,x+step) using fourth-order Runge-Kutta algorithm with adaptive stepsize. Direct descendant of Runge_kutta_smart object.			

```
//  ********************************************************
//  *                                                      *
//  *        Class RUNGE_KUTTA_DRIVER                      *
//  *                                                      *
//  ********************************************************
//
//  Fifth-order Runge-Kutta driver with adaptive stepsize.
//  Direct adaptation of and heir of Class Runge_kutta_smart.
//  Source: Numerical Recipes pp 550-554.


class Runge_kutta_driver : public Runge_kutta_smart
{
  protected:
    double x_in;              // Initial independent variable value.
    double x_out;             // Final independent variable value.
    double h_min;             // Minimum acceptable integration increment.
    double step;              // Step size = x_out - x_in.
    Dependent_variables y_in;  // Initial dependent variable values.
    Dependent_variables y_out; // Final dependent variable values.
    int good_step;            // Number of good integration increments.
    int bad_step;             // Number of bad integration increments.
  public:
    Runge_kutta_driver(void);
    Runge_kutta_driver(const int & N, const double & X,
                       const Dependent_variables Y,const double & H,
                       const double & A,const double & XIN, const double & HMIN,
                       const Dependent_variables & YIN, const double & S);
    Runge_kutta_driver(const int & N, const double & X,
                       const Dependent_variables Y,const double & S);
    ~Runge_kutta_driver(void);
   void drive(void);
   void update(void);
   void report(void);
   friend ostream & operator<<(ostream & out, Runge_kutta_driver D);
};


Runge_kutta_driver::Runge_kutta_driver(void)
                    : Runge_kutta_smart(),
                     x_in(0.0),x_out(0.0),h_min(0.0),
                     y_in(),y_out(),good_step(0),bad_step(0),step(0.0)
{
  // No code required.
}
```

```
Runge_kutta_driver::Runge_kutta_driver(const int & N, const double & X,
                    const Dependent_variables Y,const double & H,
                    const double & A,
                    const double & XIN, const double & HMIN,
                    const Dependent_variables & YIN, const double & S)
                   : Runge_kutta_smart(N,X,Y,H,A),
                   x_in(XIN),h_min(HMIN),y_in(YIN),step(S),y_out(N)
{
  x_out = XIN + S;
  y_out = YIN;
  good_step = 0;
  bad_step  = 0;
}


Runge_kutta_driver::Runge_kutta_driver(const int & N, const double & X,
                    const Dependent_variables Y,const double & S)
                   : Runge_kutta_smart(N,X,Y,S/10.0,1.0e-10),
                   x_in(X),h_min(1.0e-6),y_in(Y),step(S),y_out(N)
{
  x_out = X + S;
  y_out = Y;
  good_step = 0;
  bad_step  = 0;
}



Runge_kutta_driver::~Runge_kutta_driver(void)
{
  //  No code required.
}



void Runge_kutta_driver::drive(void)
{
  // Assign value to Runge Kutta vector:
  x = x_in; y = y_in; h = h_actual;
  good_step = 0; bad_step = 0;

  int loopcond = 0, loopnum = 0;
  while (loopcond == 0)
  {
    // Scaling to monitor accuracy.
    derivs();
    for (int i=0;i<n;i++)
    {
         y_scale.set(i, abs(y.get(i)) + abs(h*dydx.get(i)) + tiny);
    }

    // If step overshoots endpoint, reduce stepsize.
    if ((x+h-x_out)*(x+h-x_in) >= -tiny) {h = x_out-x;}
```

```
      // Propagate solution forward through h.
      pushforward();

      // Assess success.
      if (h_actual == h) {good_step = good_step+1;}
      else               {bad_step  = bad_step + 1;}

      // Done yet?
      if ((x-x_out)*(x-x_in) >= -tiny)
      {
            y_out = y_best;
            loopcond = 1;
      }

      if (fabs(h_next) <= h_min)
      {
            cout << "Stepsize below minimum (next, min):";
            cout << h_next << "   " << h_min << "\n";
      }
      h = h_next;

      loopnum = loopnum+1;
      // if (loopnum > 100) {loopcond = 1;}
    }
    if (h>step) {h=step;}
}



void Runge_kutta_driver::update(void)
{
  x_in  = x_out;
  y_in  = y_out;
  x_out = x_out + step;
}



ostream & operator<<(ostream & out, Runge_kutta_driver D)
{
  out << D.x_in << "   " << D.y_in << "\n";

  return out;
}
```

```
void Runge_kutta_driver::report(void)
{
  cout << "\n*********************************\n";
  cout << "*   RUNGE_KUTTA_DRIVER: Report  \n";
  cout << "*   x:       " << x       << "\n";
  cout << "*   y:       " << y       << "\n";
  cout << "*   x_in:     " << x_in     << "\n";
  cout << "*   x_out:    " << x_out    << "\n";
  cout << "*   h:       " << h       << "\n";
  cout << "*   h_min:    " << h_min    << "\n";
  cout << "*   y_in:     " << y_in     << "\n";
  cout << "*   y_out:    " << y_out    << "\n";
  cout << "*   good_step: " << good_step << "\n";
  cout << "*   bad_step:  " << bad_step  << "\n";
  cout << "*********************************\n";
}
```

DEMONSTRATION

A-12. This section applies the OOP-oriented numerical integration algorithm to two examples taken from recent DAOR projects. First: a 1993 study developed a Ballistic Missile Defence package to examine the central elements of missile defence (Reference [4]). Missiles and interceptors were modelled using satellite motion drivers from previous studies. Second: in 1995, studies were conducted examining the ability of a CF-18 fighter aircraft carrying various air-to-air missiles against hostile aircraft (Reference [5]). Air-to-air missiles were modelled using procedural numerical integration codes incorporated into an existing simulation environment.

Ballistic Missile Class

A-13. The following code fragment shows how a ballistic missile object may be inherited from the base Runge Kutta driver object. The equations of motion governing an impulsively accelerated point mass in motion about a uniform sphere are presented. This C++ implementation exploits the polymorphic behaviour possible with the virtual derivs() method to override the default governing equations inherited from the Runge_kutta class. The main() program constructs a Missile class instance and propagates it through 200 seconds of flight. Output appears at Table A-6 and Figure A-1.

A-14. The equations governing the motion of an impulsively accelerated missile moving in the gravitational field of a uniform, spherical, non-rotating Earth in vacuum are:

$$\ddot{\vec{r}} = -\frac{\mu}{r^2}\hat{r}, \tag{1}$$

where overdots denote differentiation with respect to time, r is the distance from the missile to the Earth centre, arrows denote vectors in an Earth-centred Cartesian co-ordinate system, and hats denote unit vectors. In component form the two-dimensional version of these equations are

$$\begin{aligned} \ddot{x} &= -\frac{\mu}{r^3}x, \\ \ddot{y} &= -\frac{\mu}{r^3}y, \\ r &= \sqrt{x^2 + y^2}. \end{aligned} \tag{2}$$

These are written as a set of four first-order ODEs in dependent variables (y_0,y_1,y_2,y_3) as:

$$\begin{aligned} \dot{y}_0 &= y_1, \\ \dot{y}_1 &= -\frac{\mu}{r^3}y_0, \\ \dot{y}_2 &= y_3, \\ \dot{y}_3 &= -\frac{\mu}{r^3}y_2, \end{aligned} \tag{3}$$

where $y_0 = x$, $y_2 = y$.

A-15. The Missile object class is inherited from the Runge_kutta_driver class as indicated in the code segment above. The entire code modification required for the missile class is shown, all other behaviours are inherited from the ancestor class. The code for the Missile class is roughly one page long, of which default constructors and destructors account for fully half. The only substantive modification is to the derivs() function which returns the values of the governing equations.

A-16. Output is directed to a sequential file object. The example used is from a DAOR publication (Reference [4]) showing the trajectory of a 1,000-km maximum range Theatre Ballistic

Missile used in a depressed mode over a 556.6 km range. The missile has a burnout speed of 3012.9 m/sec, and is launched at an elevation of 15.7 degrees and reaches a maximum altitude of 39.3 km above the Earth's surface over its 193.5 second flight time.

A-17. An excerpt of the missile data file appears at Table A-6. The entire data file appears graphically at Figure A-1. Output data is given with respect to Cartesian co-ordinates (Range/Elevation) as measured at the launch site. Algorithms exist which will convert this data to altitude above a point on the Earth's surface (Reference [4]). Thus processed, this raw trajectory forms the more familiar parabolic curve. Owing to the Earth's curvature, the TBM impacts the Earth's surface a distance 24.25 km below the horizon. Intercept takes place at 193.5 seconds, as indicated at Reference [5].

TABLE A-6

DEMONSTRATION 1: BALLISTIC MISSILE DATA TABLE EXTRACT

Time (sec)	Alt (km)	Speed (z, m/sec)	Range (km)	Speed (x, m/sec)
0	0.0000	813.9671	0.00000	2900.833
1	0.8090	804.1701	2.90083	2900.831
2	1.6085	794.3755	5.80166	2900.824
3	2.3980	784.5834	8.70248	2900.813
4	3.1775	774.7938	11.60328	2900.797
5	3.9470	765.0065	14.50407	2900.777
6	4.7075	755.2216	17.40484	2900.753
7	5.4580	745.4391	20.30557	2900.724
8	6.1985	735.6589	23.20628	2900.691
9	6.9290	725.8810	26.10696	2900.653
10	7.6500	716.1053	29.00759	2900.611

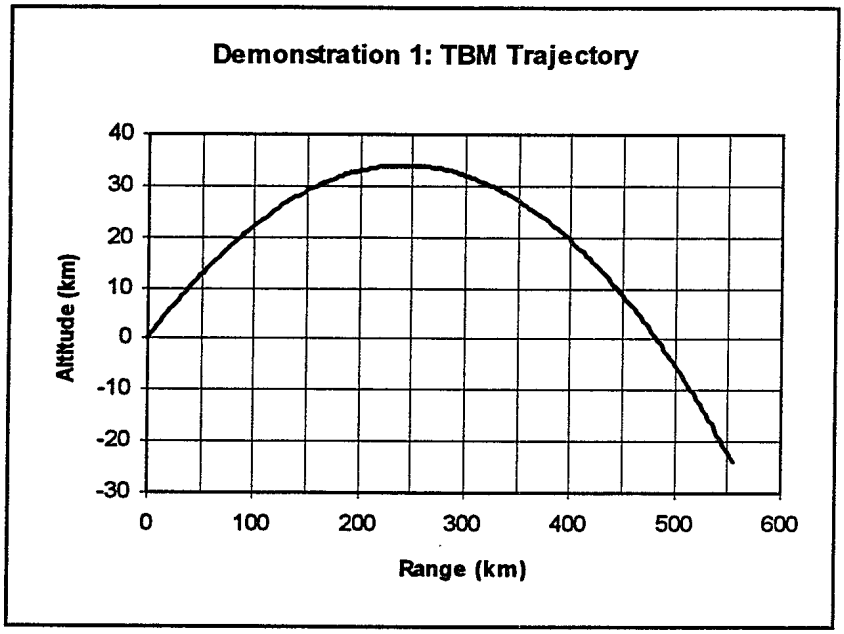

Figure A-1. Demonstration Theatre Ballistic Missile Trajectory

TABLE A-7

MISSILE MEMBER/METHOD TABLE

Name	**MISSILE**		
File	**ODEINT.CPP**		
Members	**Field**	**Type**	**Comment**
	(none)		
Methods	**Function**	**Returns**	**Comment**
	Missile	Constructor	Demonstration constructor: arguments (N,X,Y,H).
	~Missile	Destructor	Standard Destructor.
	derivs	virtual void	Virtual method overloads Runge_kutta.derivs with missile equations of motion.
	operator<<	friend ostream	Overloaded output operator.
Description: Demonstration ballistic missile object. Inherited from Runge_kutta_driver, implements motion in idealized Newtonian potential. Note use of overloaded DERIVS method exploiting polymorphic behaviour.			

```
class Missile : public Runge_kutta_driver
{
  public:
    Missile(const int & N, const double & X, const Dependent_variables Y,
            const double & H);
    ~Missile(void);
    virtual void derivs(void);
    friend ostream & operator<<(ostream & out, Missile M);
};


Missile::Missile(const int & N, const double & X,
            const Dependent_variables Y, const double & H)
                : Runge_kutta_driver(N,X,Y,H)
{
  // No code required.
}


Missile::~Missile(void)
{
  // No code required.
}


void Missile::derivs(void)
{
 double r = y.get(0)*y.get(0) + y.get(2)*y.get(2);
 r = pow(r,1.5);
 if (r <= tiny) {r=tiny;}
 r = 1.0/r;

 double mu = 3.986012e14;  // Bate Meuller White pg. 429.

 dydx.set(0,y.get(1));
 dydx.set(1,-mu*r*y.get(0));
 dydx.set(2,y.get(3));
 dydx.set(3,-mu*r*y.get(2));
}


ostream & operator<<(ostream & out, Missile M)
{
  out << M.x << M.y << "\n";
  return out;
}


void main(void)
{
  int n = 4;
  double x = 0.0, h = 1.0, a = 1.0e-6;

  double theta = 15.674*3.1415926353/180.0; // Depressed elevation angle
  double speed = 3012.868;                  // Burnout speed
```

```
    Dependent_variables y(4);
          y.set(0,6378145.0);                    // On Earth's surface (m)...
          y.set(1,speed*sin(theta));
          y.set(2,0.0);                          // ... from the equator (x=R, y=0)
          y.set(3,speed*cos(theta));

    ofstream outfile("missile.out");

    Missile m(n,x,y,h);


    for (int i=0;i<200;i++)                      // 200 seconds should be enough
    {
      outfile << m;
      m.drive(); m.update();
    }
}
```

Air-to-Air Missile Motion

A-18. High fidelity models of Air-to-Air missiles are significantly more complex than the simple impulsive missile model discussed above. The model described here is a simplification of a simulation kernel used in a recent DAOR investigation (Reference [5]). A nine degree of freedom model is used. Translational and rotational planar position and velocity are modelled, together with mass loss through propellant burn for a total of (2x2) + (2x2) + 1 = 9 degrees of freedom.

A-19. The equations of motion are taken from a widely available source (Reference [6]) To simplify implementation, the vehicle angle of attack is set equal to zero and suppressed as a control variable. To further simplify the equations of motion, angular degrees of freedom will be suppressed. Motion of the extended rigid body in the plane is described by two translational and one rotational degree of freedom, hence by three coupled second order ordinary differential equations (Reference [7]):

$$\begin{aligned} m\ddot{p} &= T\cos\varepsilon - D - W\sin(\eta), \\ m\ddot{q} &= T\sin\varepsilon - L + W\cos(\eta), \\ B\ddot{\eta} &= h_p L - h_q T\sin\varepsilon - M_q - K\dot{\eta}. \end{aligned} \tag{4}$$

Table A-8 describes each variable used above.

TABLE A-8
MISSILE EQUATIONS OF MOTION DATA DIRECTORY

Variable	Unit	Comment
m	kg	Missile mass
p	m	Axis aligned with velocity vector
T	N/sec	Thrust
ε	rad	Thrust offset angle
D	N	Drag force
W	N	Massile gravitational mass
η	rad	Elevation above horizon
q	m	Axis normal to p, directed upwards
L	N	Lift force
B	kg/m^2	Transverse moment of inertia
h_p	m	Centre of mass/pressure centre distance
h_q	m	Centre of mass/lift centre distance
M_q	kg/m^2	Aerodynamic pitching moment
K	kg/m^2	Mass flow moment

A-20. These equations are accompanied by two additional first order ordinary differential equations relating instantaneous vehicle location to the launch location:

$$\dot{x} = v\cos\eta, \qquad (5)$$
$$\dot{y} = v\sin\eta.$$

The equations of motion are solved by first reducing to a set of first order ordinary differential equations.

A-21. Lift and drag forces have the form

$$L = \frac{1}{2}\rho S C_L v^2, \qquad (6)$$
$$D = \frac{1}{2}\rho S C_D v^2.$$

where ρ is the atmospheric mass density, C_L, C_D lift and drag coefficients, and S the reference cross-sectional area. Lift and drag forces are proportional to the local atmospheric mass density, assumed to vary exponentially as

$$\rho(z) = \rho_0 e^{-z/H}, \tag{7}$$

where z is the vehicle altitude above the Earth's surface, and ρ_0 and H are empirally derived constants (Reference 1):

$\rho_0 = 1.225\ kg/m^3$; *and*
$H = 7.220\ km.$

Denote the overall mass flow rate by

$$c = \frac{dm}{dt}. \tag{8}$$

A-22. Introduce the variables

$$\begin{aligned} x_1 &= p, \\ x_2 &= \dot{p}, \\ x_3 &= q, \\ x_4 &= \dot{q}, \\ x_5 &= \eta, \\ x_6 &= \dot{\eta}, \\ x_7 &= x, \\ x_8 &= z, \\ x_9 &= m, \end{aligned} \tag{9}$$

This yields nine first order ordinary differential equations in nine unknowns:

$$
\begin{aligned}
\dot{x}_1 &= x_2, \\
\dot{x}_2 &= \frac{T}{x_9}\cos\varepsilon - \frac{D}{x_9} - g\sin(x_5), \\
\dot{x}_3 &= x_4, \\
\dot{x}_4 &= \frac{T}{x_9}\sin\varepsilon - \frac{L}{x_9} + g\cos(x_5), \\
\dot{x}_5 &= x_6, \\
\dot{x}_6 &= \frac{L}{B}h_p - \frac{T}{B}h_q\sin\varepsilon - \frac{M_q}{B} - \frac{K}{B}x_6, \\
\dot{x}_7 &= x_2\cos x_5, \\
\dot{x}_8 &= x_2\sin x_5, \\
\dot{x}_9 &= c.
\end{aligned}
\tag{10}
$$

These equations are solved using the Runge Kutta algorithm below.

A-23. Table A-9 shows an extract from the data table produced by the main program below. Figure A-2 shows tangential speed as a function of time for the first 30 seconds of missile flight. The missile is launched under conditions similar to those used as the baseline scenario in Reference [5]: at an altitude of 10 km and speed of 300 m/sec (roughly Mach 1). The figure shows good agreement with that of Reference [5], but it should be noted that exact agreement is not expected. The simple AIM-7 model used here assumes a constant thrust and constant drag. Further, missile body lift is not modelled nor are control surfaces in place to maintain a constant altitude. These modifications can easily be made to this base missile class.

TABLE A-9

DEMONSTRATION 2: AIR-TO-AIR MISSILE DATA TABLE EXTRACT

Time (sec)	Tangential Position (m)	Tangential Speed (m/sec)	Normal Position (m)	Normal Speed (m/sec)	Elevation (rad)	Angular velocity (rad/sec)	Range (m)	Altitude (m)	Mass (kg)
0	0.0000	300.0000	0.0000	0.0000	0.17452	0	0.00000	10000.000	231.3300
1	324.1147	348.2497	-4.8305	-9.6610	0.17452	0	56.27779	9680.808	226.9797
2	696.5060	396.5173	-19.3220	-19.322	0.17452	0	120.9381	9314.073	222.6294
3	1117.0750	444.5598	-43.4744	-28.9829	0.17452	0	193.9639	8899.893	218.2792
4	1585.4500	492.0732	-77.2878	-38.6439	0.17452	0	275.2904	8438.633	213.9289
5	2100.9200	538.6805	-120.7620	-48.3050	0.17452	0	364.7943	7930.992	209.5786
6	2662.3560	583.9211	-173.8980	-57.9661	0.17452	0	462.2794	7378.081	205.2283
7	3268.1220	627.238	-236.6950	-67.6272	0.17452	0	567.4621	6781.519	200.8780
8	3915.9720	667.9739	-309.1520	-77.2883	0.17452	0	679.9522	6143.508	196.5278
9	4602.9560	705.3698	-391.2710	-86.9494	0.17452	0	799.2368	5466.958	192.1775
10	5325.3160	738.5808	-483.0510	-96.6105	0.17452	0	924.6646	4755.571	187.8272

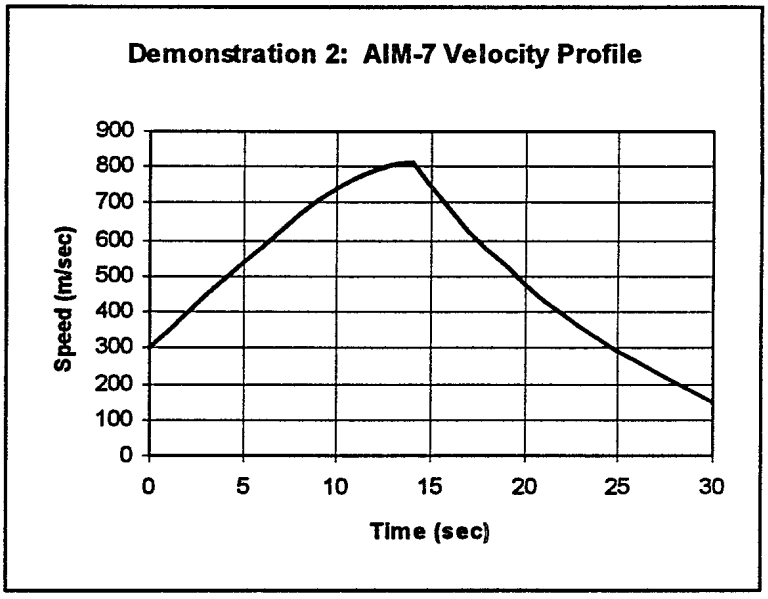

Figure A-2. Demonstration AIM-7 Air-to-Air Missile Velocity Profile

TABLE A-10

RUNGE_KUTTA_SMART MEMBER/METHOD TABLE

Name	MISSILE		
File	ODEINT.CPP		
Members	**Field**	**Type**	**Comment**
	thrust	Double	Instantaneous thrust (N)
	lift	Double	Lift coefficient.
	drag	Double	Drag coefficient.
	press_cent	Double	Pressure centre (m).
	lift_cent	Double	Lift centre (m).
	trans_mom	Double	Transverse momentum moment (kg m^2).
	rot_mom	Double	Rotational momentum moment (kg m^2).
	area	Double	Reference cross section (m^2).
	mass_flow	Double	Mass flow rate (kg/sec).
Methods	**Function**	**Returns**	**Comment**
	Missile	Constructor	Arguments (N,X,Y,H).
	derivs	virtual void	Overloaded derivative method.
	operator<<	friend ostream	Overloaded output operator.
Description: Object models AIM-7 type missile flyout with mass burn and exponentially dense atmosphere. Note use of virtual DERIVS function to overload Runge_kutta.derivs in computation of equations of motion.			

```
class Missile : public Runge_kutta_driver
{
 private:
   double thrust;
   double lift;
   double drag;
   double press_cent;
   double lift_cent;
   double trans_mom;
   double rot_mom;
   double area;
   double mass_flow;
 public:
   Missile(const int & N, const double & X, const Dependent_variables Y,
           const double & H);
   ~Missile(void);
   virtual void derivs(void);
   friend ostream & operator<<(ostream & out, Missile M);
};
```

```
Missile::Missile(const int & N,const double & X, const Dependent_variables Y,
                 const double & H): Runge_kutta_driver(N,X,Y,H)
{
  thrust = 12000.0;      // Mean thrust level (N/sec)
  lift = 0.0;            // Assume zero lift configuration
  drag = 0.087;          // Drag at Mach 2.0.
  press_cent = 1.0;      // Estimated pressure centre/CM value
  lift_cent = 1.0;       // Estimated lift centre/CM value
  trans_mom = 0.0;       // Transverse momentum moment (not used)
  rot_mom = 0.0;         // Rotational momentum moment (not used)
  area = 0.385;          // Reference area (m^2)
  mass_flow = -4.35; // Mean mass flow rate (kg/sec)
}


Missile::~Missile(void)
{
  // No code required.
}


void Missile::derivs(void)
{
 double speed2 = pow(y.get(1),2) + pow(y.get(3),2);

 if (x > 14.0) {thrust = 0.0; mass_flow = 0.0;}

 double density = 1.225*exp(-y.get(7)/7220.0);
// Exponential atmospheric density model.

 dydx.set(0,y.get(1));
 dydx.set(1, thrust/y.get(8) - area*drag*density/2.0*speed2/y.get(8) - 9.81*sin(y.get(4)));
 dydx.set(2,y.get(3));
 dydx.set(3, -9.81*cos(y.get(4)));
 dydx.set(4,y.get(5));
 dydx.set(5, 00.0);
 dydx.set(6,y.get(1)*sin(y.get(4)));
 dydx.set(7,-y.get(1)*cos(y.get(4)));
 dydx.set(8,mass_flow);
}


ostream & operator<<(ostream & out, Missile M)
{
  out << M.x << M.y << "\n";
  return out;
}


void main(void)
{
  int n = 9;
  double x = 0.0, h = 1.00;

  Dependent_variables y(n);
        y.set(0,0.0);       //Tangential position
```

```
            y.set(1,300.0);     //Tangential velocity
            y.set(2,0.0);       //Normal position
            y.set(3,0.0);       //Normal velocity
            y.set(4,0.17452);  //Elevation angle wrt horizon (10 deg)
            y.set(5,0.0);       //Angular velocity
            y.set(6,0.0);       //Horizontal position wrt launch
            y.set(7,10000.0); //Vertical position wrt launch
            y.set(8,231.33);   //Missile mass

        ofstream outfile("AIM7.out");
        Missile m(n,x,y,h);

        for (int i=0;i<90;i++)
        {
          cout << i << "\n";
          outfile << m;
          m.drive();
          m.update();
        }
}
```

DOCUMENT CONTROL DATA

(Security classification of Title, body of abstract, and indexing annotation must be entered when the overall document is classified)

1. ORIGINATOR (the name and address of the organization preparing the document. Organizations for whom the document was prepared, e.g. Establishment sponsoring a contractor's report, or tasking agency, are entered in section 8.)

OPERATIONAL RESEARCH AND ANALYSIS
National Defence Headquarters
Ottawa, Ontario K1A 0K2

2 SECURITY CLASSIFICATION (overall security classification of the document including special warning terms if applicable)

UNCLASSIFIED

3. TITLE (the complete document title as indicated on the title page. Its classification should be indicated by the appropriate abbreviation (S, C, or U) in parenthesis after the title)

Object Oriented Programming for Resuable Applications: A Modelling Approach (U)

4. AUTHORS (Last name, first name, middle initial)

Frank, Gregory W.

5. DATE OF PUBLICATION (month and year of publication of document)

April 1996

6a. NO. OF PAGES (total containing information. Include Annexes, Appendices, etc.)

50

6b. NO OF REFS (total cited in the document)

6

7. DESCRIPTIVE NOTES (the category of the document, e.g. technical report, technical note or memorandum. If appropriate, enter the type of report, e.g. interim, progress, summary, annual, or final. Give the inclusive dates when a specific report period is covered.

DAOR RESEARCH NOTE

8. SPONSORING ACTIVITY (the name of the department project office sponsoring the research and development. Include the address.)

OPERATIONAL RESEARCH AND ANALYSIS

9a. PROJECT OR GRANT NO. (if appropriate, the applicable research and development project or grant number under which the document was written. Please specify whether project or grant)

9b. CONTRACT NO. (if appropriate, the applicable number under which the document was written

10a. ORIGINATOR'S DOCUMENT NUMBER (the official document number by which the document is identified by the originating activity. This number must be unique to this document)

DAOR-RN-9603

10b. OTHER DOCUMENT NOS. (Any other numbers which may be assigned this document either by the originator or by the sponsor)

11. DOCUMENT AVAILABILITY (any limitations on further dissemination of the document, other than those imposed by security classification)

(X) Unlimited distribution
( ) Distribution limited to defence departments and defence contractors; further distribution only as approved
( ) Distribution limited to defence departments and Canadian defence contractors; further distribution only as approved
( ) Distribution limited to government departments and agencies; further distribution only as approved
( ) Distribution limited to defence departments; further distribution only as approved
( ) Other (Please specify)

12. DOCUMENT ANNOUNCEMENT (any limitation to the bibliographic announcement of this document. This will normally correspond to the Document Availability (11). However, where further distribution (beyond the audience specified in 11) is possible, a wider announcement audience may be selected)

13. ABSTRACT (a brief and factual summary of the document. It may also appear elsewhere in the body of the document itself. It is highly desirable that the abstract of classified documents be unclassified. Each paragraph of the abstract shall begin with an indication of the security classification of the information in the paragraph (unless the document itself is unclassified) represented as (S), (C), or (U). It is not necessary to include here abstracts in both official languages unless the text is bilingual.

This note discusses a potential paradigm for computer-based modelling and simulation activities within the Directorate of Air Operational Research. Adherence to Object Oriented Programming principles is advocated for modelling or simulation projects which are likely to be reused in follow-on or adjunct studies. Object Oriented Programming tenets of encapsulation, inheritance and polymorphism can increase analyst productivity by simplifying maintenance and modification tasks.

14. KEYWORDS, DESCRIPTORS or IDENTIFIERS (technically meaningful terms or short phrases that characterize a document and could be helpful in cataloguing the document. They should be selected so that no security classification is required. Identifiers, such as equipment model designation, trade name, military project code name, geographic location may also be included. If possible keywords should be selected from a published thesaurus e.g. Thesaurus of Engineering and Scientific Terms (TEST) and that thesaurus identified. If it is not possible to select indexing terms which are unclassified, the classification of each should be indicated as with the title.)

OBJECT ORIENTED PROGRAMMING (OOP)
MODELLING
SIMULATION
SOFTWARE
COSTS AND BENEFITS
RUNGE-KUTTA ALGORITHM

Canada

498409