1996

Knowledge Discovery in Computational Science: A Case Study in Algorithm Selection

N. Ramakrishnan

John R. Rice
Purdue University, jrr@cs.purdue.edu

Report Number:
96-081

KNOWLEDGE DISCOVER IN COMPUTATIONAL SCIENCE: A CASE STUDY IN ALGORITM SELECTION

N. Ramakrishnan
J. R. Rice

Department of Computer Sciences
Purdue University
West Lafayette, IN 47907

CSD-TR 96-081
December 1996
(Revised February 1997)

Knowledge Discovery in Computational Science: A Case Study in Algorithm Selection

N. Ramakrishnan and J.R. Rice
Department of Computer Sciences
Purdue University, West Lafayette, IN 47906

Keywords: Computational Science, Problem Solving Environments, Knowledge Discovery, Data Mining, Algorithm Selection, Feature Determination

Abstract

Computational Science is an interdisciplinary field that addresses all aspects of science and engineering that utilize computing as their main tool. One important research issue in computational science is the selection of appropriate algorithms for solving particular problems. In other words, given a problem and some performance objectives (speed, accuracy, cost etc.), one is to select the best (or nearly best) algorithm. In this paper we formulate these issues, illustrate the general methodology and present new techniques for the domain of numerical quadrature. Our technique uses a combination of inductive logic programming, symbolic and numeric identification of problem features and a repository of algorithm performance data to map problems plus performance objectives to appropriate algorithms. We discuss how this selection problem is closely related to knowledge discovery and data mining in particular problem domains.

1. INTRODUCTION

Computational Science & Engineering – CS&E – is a burgeoning field of research that encompasses all fields of science and engineering that use computing as their main tool. In other words, computing is specifically used to advance a scientific discipline as contrasted to merely providing support for performing traditional tasks in science. It is popularly regarded as the third way to do science, complementing theoretical and experimental work. Work in this area involves the entire computational process: details of computer architecture, system software and the design of efficient algorithms. CS&E can be likened to be a pyramid with a square base, with the driving science and engineering application at the tip. The four nodes at the base are:

—Algorithms: Numeric, Symbolic and Geometric

—Computer Architecture

—System Software

—Performance Evaluation and Analysis

Most research in CS&E involves designing computational models of real–life scientific phenomena and simulating such complex models with advanced computing methods (Henceforth, scientific is understood to encompass engineering also). This

work requires application scientists to have a thorough understanding of computer architectures, underlying software and numerical algorithms which many, perhaps most, of them do not have. They do not (and should not be forced to) invest the time and energy to learn such elaborate details of the computational process. Moreover, much of this information involves the 'play' between algorithms, applications and architectures and is not readily available to the application scientist. Scientists need automated systems to assist them, ones that conduct knowledge discovery and data mining in the computational settings of scientific domains. Examples of areas where such systems can be applied are algorithm selection, mathematical reasoning, systematic methods of scientific inference, inferring scientific laws from experimental data, discrete model–building for routine tasks in science, inferring patterned behavior and causal relations, fitting models, uncovering invariants (as in data–driven discovery), qualitative analysis, objective function clustering, etc. A more elaborate discussion of these tasks of scientific discovery (albeit in the context of theoretical and experimental science) is provided in [9]. In this paper, we specifically concentrate on automatic algorithm selection techniques — efficient schemes that guide the user from a high–level specification of a scientific problem to a solution satisfying his performance requirements.

Efficient algorithm selection systems also aid in the creation of Problem Solving Environments (PSEs) [4] which are high–level environments for CS&E. A PSE is a computer system that provides all the computational facilities necessary to solve a target class of problems. Moreover, PSEs use the vernacular of the target class of problems, so scientists can utilize them without specialized knowledge of the underlying computer hardware or software infrastructure. The important features of a PSE are advanced solution methods, automatic or semiautomatic selection of solution methods (the subject of this paper) and a natural user interface that provides the user (in this case, a scientist) with a high–level abstraction of the underlying computational facilities. We believe that the need for automatic algorithm selection systems will become increasingly more critical due to several reasons:

—PSEs are becoming more ubiquitous and widely accepted in scientific communities

—The advent of "net–centric" computing has provided the impetus for developing network based scientific software servers, in particular WWW based "Problem Solving Services"; this has led to a rapid increase in the number of online algorithms/methods that are made available to the application scientist

—Such systems also aid (indirectly) in the performance evaluation of scientific software

—If successful, PSEs serve as high level front–ends to software repositories of numerical algorithms; they can also complement the services of software 'indexing' systems like GAMS – the Guide to Available Mathematical Software, maintained at the National Institute of Standards and Technology.

In this paper, we report on the system GAUSS that performs automatic algorithm selection within a traditional, well–understood scientific domain – numerical integration. The task is to decide on an efficient algorithm to numerically integrate a given mathematical function, given constraints on accuracy and time. GAUSS

also allows the numerical scientist to conduct automated knowledge discovery in its domain. Our current implementation of GAUSS banks on a three–pronged architecture to aid in the algorithm selection process.

2. KNOWLEDGE DISCOVERY IN SCIENCE AND CS&E

We begin with a discussion of the history of similar systems in science. The 70's witnessed the early utilization of AI systems in scientific domains with popular expert systems such as PROSPECTOR and DENDRAL. Data driven scientific discovery was pursued by systems like BACON. These programs looked for relationships among variables in scientific data and BACON is credited with rediscovering Kepler's laws of planetary motion and some important empirical relationships in physical chemistry. Lenat's AM program (for Automated Mathematician) conducts discovery in traditional mathematics and has been shown to 'uncover interesting concepts' such as that of prime numbers, perfect numbers and Goldbach's conjecture. Systems have been developed for performing analytical differentiation, discovering logic proofs, validating theorems in geometry and unraveling molecular structures. Further applications abound in biology, astronomy, physics and graph theory. A comprehensive summary of early computer science research on scientific discovery is provided in [10]. Recently, applications have demonstrated considerable expertise in catalytic chemistry, particle physics and cell biology. Qualitative analysis of dynamical systems has been undertaken in [1] — the approach taken is to automate most routine tasks in numerical modeling and to exploit techniques like imagistic reasoning and computer vision to make programs 'see' what to compute and how to compute it. A recent AAAI Symposium on "Intelligent Scientific Computation" [6] highlights the contributions made by AI to Scientific Computing (and vice versa).

With more scientific data getting organized into huge databases and repositories, the fields of KDD (Knowledge Discovery in Databases) and data mining have contributed to many discoveries in science. While KDD is a frontier for both AI and database technology and acts as a link between these two diverse fields, data mining can be viewed as an attempt to find unknown (and possibly interesting) patterns and relations in large databases. In this sense, data mining can be viewed to form an abstraction of the database being 'mined'. These methodologies have found widespread application in real–life scientific applications such as the automatic analysis of deviations in health care data, cataloging sky surveys and predicting equity returns. While several techniques have been proposed to perform data mining, the suitability (or lack thereof) of a particular technique depends on the application at hand. The role and representation of domain knowledge in discovery, modeling subjectivity and the actual process of scientific discovery are important research issues in the application of KDD and data mining in scientific domains.

Several arguments have been put forth to explain the success of such systems in science. Popular beliefs attribute them to the speed/efficiency/memory of computers and the ability of computers to conduct exhaustive searches. A common underlying theme of most of these systems is that they can be construed as attempting to conduct "heuristic search in matrix spaces". The success of the early expert systems is credited to the fact that they encode domain specific knowledge and make such expertise more broadly available than in the past. The AM pro-

gram uses several heuristics to determine whether a certain concept is 'interesting' enough to be investigated. This is achieved by a small amount of background domain knowledge and a judicious choice of heuristics that guide the search into 'interesting territories'. At the other end of the spectrum, Valdes-Perez [9] argues that successful scientific discovery is more likely to be facilitated by a change in the task representation of the problem space. In particular, representational schemas motivated by a certain field of science were found to offer valuable insights into a completely different domain. A case in point is the opinion [1] that problem solvers employing visual or diagrammatic representations are more efficient than those relying on purely linguistic or symbolic paradigms.

While most of these systems have contributed importantly to the scientific understanding about their respective domains, their modes of operation are far removed from the environment where a computational scientist conducts research. For instance, the computer lies at the heart of his activities and if such scientific discovery systems are to become useful in this redefined context, they will have to inherit and represent the many additional aspects of computation that are so integral to conducting science. For example, while a traditional discovery program in mathematics need concentrate on only the algorithmic aspects of a problem strategy and other abstract concepts, its 'computational' counterpart has to cater also to the implementation aspects. This brings in issues normally associated with the performance evaluation of systems.

Some important research issues specific to conducting discovery in CS&E settings are:

—The representation of performance data, automating performance evaluation and analysis; Is there a way to provide a taxonomy for such data so that learning in one domain can be used in a bigger 'enclosing' domain?

—The automatic identification of features of numerical entities like functions, graphs, regions, boundaries, equations, operators, etc. Is it possible or economic to conduct automatic feature identification — particular emphasis needs to be laid on the computational framework within which such identification is made.

—What are effective ways to represent knowledge about numerical and computational domains?

While the possibilities for scientific discovery in computational domains are potentially enormous, this paper deals specifically with a very important requirement that is universal in many computational settings – the need for efficient algorithm selection.

3. THE ALGORITHM SELECTION PROBLEM

The algorithm selection problem has its origins in an early paper by Rice [8]. Given a problem in scientific computation and performance criteria constraints on its solution (such as accuracy, time, cost, etc.), it is required to decide on a good (enough) algorithm to achieve the desired objectives. Even for 'routine' tasks in scientific computing, this can get quite complicated. An abstract model for this problem is also proposed in [8]. The salient features of this model are reproduced in Fig. 1, where p is the problem given and w are the performance criteria. The problem p is 'represented' by the feature(s) f in the feature space. The task is

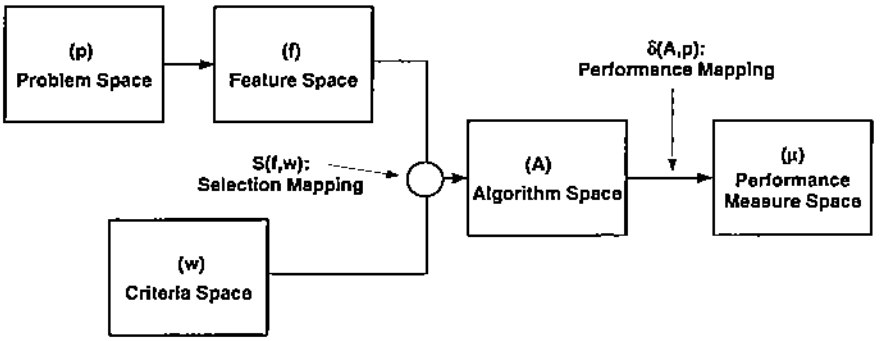


Fig. 1. The algorithm selection problem

to 'construct' a selection mapping S that provides a good algorithm A to solve p (where "good" is measured by w). We therefore need a 'means' to determine a 'good enough' algorithm A subject to the constraint that the $\mu = \delta(A, p)$ value (the performance of algorithm A on p) is "optimized" (satisfies the constraints w to the 'best extent'). The "best" selection is then the mapping that is better (in the sense of producing a better performance indicator in the performance measure space) than other possible mappings. Though this formulation caters to a wide variety of problems, methodologies have been developed with specific reference to the performance evaluation of numerical software.

This problem has been pursued in several diverse fields such as:

—Selecting solvers for partial differential equations

—Selecting solvers for ordinary differential equations

—Selecting algorithms for classifying data sets

The difficulty is further compounded by the huge dimensionality of the problem (and feature) and algorithm spaces, lack of understanding of how the problem characteristics affect algorithm performance, and the inherent uncertainty in interpreting and assessing the performance measures of a particular algorithm for a particular problem. In the context of numerical computing, several strategies have been proposed to circumvent these difficulties. A simple method is to use an exemplar based approach as in case–based reasoning — we pick a problem from a database that most closely resembles the one at hand, and use the performance data for this problem to predict the algorithm strategy for the desired problem. Clearly, this simplistic scheme degrades as the number of test problems (the problem solving "experience") increases; comparing a given problem with every one in the database could be time–consuming. Another way is to create 'problem classes' and have a two–step procedure – the first step classifies the problem into one/more of several classes and having thus reduced the complexity of the problem, the second step maps a problem class to appropriate algorithms. The efficacy of this method is dependent on how well the problem classes reflect the pattern inherent in the domain and the mechanisms used to perform classification. If the domain does not exhibit any kind of regularities, one will have to introduce an unreasonable number of classes to represent the domain. Our studies have shown that specialized schemes

tailored for classification in scientific domains [5] fare better than general-purpose algorithms. The automatic inference of such classes is another research issue that our group is working on. This is also known as clustering or unsupervised learning in the statistical learning parlance. Prior work on the algorithm selection problem has not concentrated on knowledge discovery; rather, the focus was on mapping problems to algorithms without (necessarily) making any inferences about the domain.

We feel that a satisfactory solution to the algorithm selection problem, in addition to solving the problem *per se*, also allows the computational scientist to arrive at qualitative conclusions about the domain in the form of broad, general heuristics that encode application–specific knowledge.

4. THE GAUSS SYSTEM

Now, we introduce GAUSS — a system for the automatic selection of quadrature (integration) routines in numerical computation. Given a problem in numerical integration and constraints on time and accuracy, GAUSS selects an efficient algorithm to solve it. It uses a performance database of various algorithms and test problems, an automatic feature identification module, and a knowledge methodology that represents information about the numerical domain in terms of logic formulas. **Note:** It is to be emphasized that GAUSS does not *evaluate* integrals (which is a popular bed for demonstrating concepts in AI) but only recommends algorithms to evaluate them.

4.1 The Problem of Numerical Quadrature

The problem of quadrature is important in the context of determining the areas under curves, volumes bounded by surfaces, etc. In other words, the value of the definite integral of a function defines the area bounded by the function curve between the limits of integration; $\int_a^b f(x)dx$ denotes the area under the curve $f(x)$ between the limits $x = a$ and $x = b$.

Numerical quadrature/integration [3] is an approximate technique for evaluating definite integrals wherein the integral is approximated by (in most cases) a linear combination of the values of the integrand:

$$\int_a^b f(x)dx \approx w_1 f(x_1) + w_2 f(x_2) + \cdots + w_n f(x_n),$$

$$-\infty \leq a \leq b \leq +\infty$$

In the above formulation, $x_1, x_2, \cdots, x_n$ are points (abscissae/nodes) in the interval of integration $[a, b]$, and the numbers $w_1, w_2, \cdots, w_n$ are 'weights' that accompany these points. For example, Fig. 2 shows how the area under the curve $f(x)$ is approximated by a series of rectangles, each bounded by the envelope of the curve — this is a special case of the approximation shown above. In more complicated approximations, the rectangles are replaced by trapezoids or other shapes that represent the area better. Various formulations of this general method give rise to specialized rules for numerical integration. For example, the rules of the Gaussian type are designed so that the quadrature rule is 'exact' for polynomials of certain degrees – this entails using non–uniformly spaced nodes, in general. Other

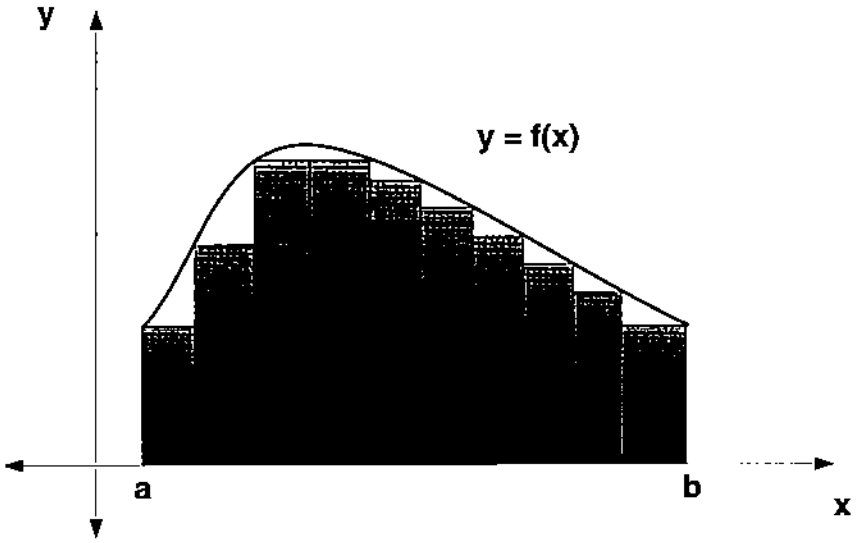


Fig. 2: Numerical integration of the function $f(x)$. It can be seen that the integrand $f(x)$ is evaluated 11 times in the region $[a, b]$ to yield the approximate integral value, the shaded area.

recastings of the basic method give rise to the popular trapezoidal and Simpson's rules of numerical integration.

The algorithm selection problem for this domain can be formally stated as follows:

Select an algorithm to evaluate $I = \int_a^b f(x)dx$
so that relative error $\epsilon_r < \theta$ and N_i is minimized

where θ is an user-specified error requirement and N_i is the number of 'function evaluations', i.e., the number of times $f(x)$ is evaluated to yield the desired accuracy (This value is 11 in Fig. 2). We choose ϵ_r and N_i as performance criteria because:

—For most software implementations of integration routines, an absolute error ϵ_a and a relative error ϵ_r are input. For the integral $I = \int_a^b f(x)dx$, these routines compute $\{R_n, E_n\}$ where the first term in the tuple is the estimate of the integral using n values of $f(x)$ while E_n is the relevant error estimate for R_n. Most of the automatic quadrature routines terminate when the error condition

$$| R_n - I | \leq E_n \leq max(\epsilon_a, \epsilon_r \mid R_n \mid)$$

is satisfied. In most of the literature (on performance evaluation of numerical integration software) and in a majority of the implementations, the routines are made to impose a strictly relative accuracy by setting ϵ_a to zero. Thus, we have chosen ϵ_r to be the main accuracy criterion.

—The time required (excluding the time to evaluate $f(x_i)$) by a numerical quadrature rule seems to vary quite widely from one implementation to another, even for the same generic technique (method) with the same number of nodes. Moreover, most efficient quadrature routines are of the 'adaptive' nature so that the weights

(w_i) and nodes (x_i) are chosen dynamically during the computation. Finally, in most applications the computing time is dominated by the time to do function evaluations. Thus, a more uniform metric seems to be the number of function evaluations (N_f) required to determine an integral. From the experiments conducted, these seem to be fairly constant over the wide range of implementations available.

Note: Only one–dimensional integrals are considered in this study. Further, methods for principal value integrals, interval analysis techniques, parallel methods of numerical integration, Monte Carlo methods and number theoretic methods are excluded. The integrand is also assumed to be either (i) a function available in symbolic form and for which it is possible to write a FORTRAN/C subroutine/function or (ii) it is input as a finite number of data values $\{x_i, y_i\}$. The latter case is particularly important because it precludes any attempt to symbolically determine the features of functions. There are numerous practical situations that produce data in this form – the results of sampling experimentally observed data, for example.

This problem is further complicated by the multitude of adaptive algorithms applicable to certain problems. An interesting study by Rice [7] concludes that there are between 1 and 10 million adaptive quadrature algorithms that are potentially interesting and significantly different from one another!! This staggering number arises from the possible ways of permuting the choice of rules, processor components, error bounds, data structures, etc.

The architecture of GAUSS consists of three main modules:

(1) **PES:** The Performance Evaluation System – The purpose of the PES is to conduct performance evaluation of quadrature algorithms for given test problems and provide the results in the form of logic formulas for use in the Knowledge Base (KB). This represents the mapping δ in Fig. 1.

(2) **AFD:** The Automatic Feature Determination module uses symbolic, numeric and imagistic techniques to ascertain 'interesting' features of functions and the domains of integration. The AFD models the mapping from the problem space to the feature space in Fig. 1.

(3) **KB:** The Knowledge Base deals effectively with learning the selection mapping S in Fig. 1, codes the information provided by the PES and the AFD and applies ILP (Inductive Logic Programming) to construct predicate logic formulas that faithfully represent all the 'positive' examples in the knowledge base.

4.2 Performance Evaluation of Algorithms

The performance evaluation of numerical algorithms is facilitated by software libraries like NAG, IMSL and QUADPACK. We have made extensive use of GAMS — a cross–index and virtual repository of numerical and statistical software components — to identify appropriate routines for quadrature. The GAMS category H2a is for the evaluation of one dimensional integrals. We have utilized 124 routines from various sites in this study. A variety of test problems are included in these implementations and several integrals with 'interesting' properties have been considered in this study. Most of these test problems are parameterized; this gives rise to a huge number of test problems.

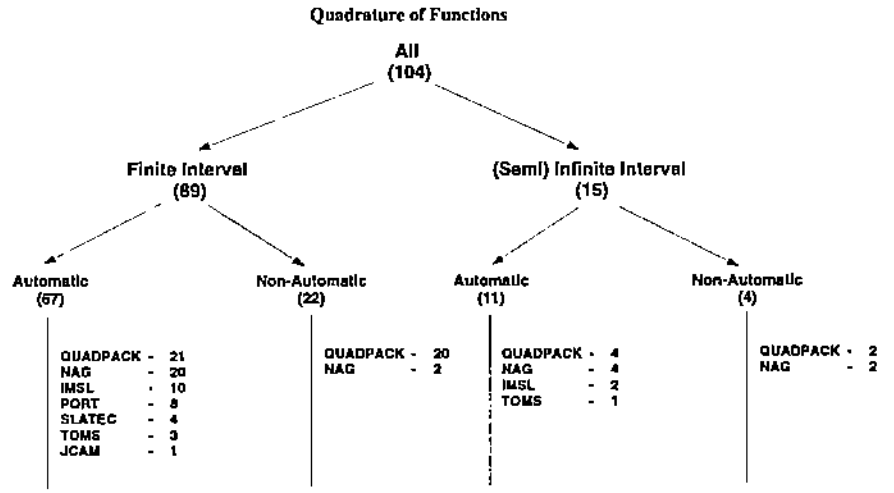


Fig. 3: The taxonomy of the Type 1 routines in GAUSS. The numbers in parantheses denote the number of routines below each node in the tree. QUADPACK, NAG, etc. denote the libraries from which the modules are obtained.

(Some of the routines do not apply to all integrals, e.g., a certain Gaussian routine might require that the integrand be expressed in the form $w(x)f(x)$ with $w(x)$ being some special weight function, such as a sinusoidal function.) For each routine and each applicable integrand, experiments were conducted with varying requirements on the relative error accuracy ϵ_r. The number of accuracy levels was 10 and the strictest error requirement used was 10^{-8}.

4.2.1 *The Quadrature Routines.* Our collection of integration routines consists of two main flavors – (i) Type 1: 104 routines that integrate functions defined in a symbolic form, and (ii) Type 2: 20 routines that integrate from points sampled in the domain of integration. A more detailed description of the Type 1 routines is given in Fig. 3.

The libraries from which the routines are obtained are QUADPACK, NAG, IMSL, PORT, SLATEC, JCAM and the collected algorithms of the ACM (TOMS). The automatic quadrature routines are those in which the user specifies the required accuracy and the algorithm attempts to achieve it. The non–automatic ones, on the other hand, use a preset number of nodes and so cannot guarantee user accuracy constraints. Most of the Type 1 routines are variations on a single family of algorithms. For example, DQAG and QAG are both automatic adaptive integrators and handle many non-smooth integrands using Gauss–Kronrod formulas. The former produces double precision results while QAG's outputs are in single precision. Also, both of these routines dynamically select among other routines considered in this study, such as QK15, QK21, QK31, QK41, etc., which are themselves nonautomatic Kronrod routines with a varying number of nodes. In other words, most of the automatic routines are in fact 'polyalgorithms' based on nonautomatic routines. One important objective of this study is to determine when an automatic algorithm is preferable to a non–automatic routine. While most of the routines were declared as general purpose modules, some of them require a special

Lyness's Integral

$$I(\lambda) = \int_1^2 \frac{0.1}{(1-\lambda)^2+0.01}dx$$

Piessens' integrals

$$\int_0^1 x^\alpha log(\frac{1}{x})dx = \frac{1}{1+\alpha^2}$$

$$\int_0^\infty \frac{x^{\alpha-1}}{(1+10x)^2}dx = \frac{(1-\alpha)\pi}{10^\alpha sin(\pi\alpha)}$$

$$\int_0^\infty x^2 e^{-2^{-\alpha}x}dx = 2^{3\alpha+1}$$

$$\int_0^1 \mid x-1/3 \mid^\alpha dx = \frac{(2/3)^{(\alpha+1)}+(1/3)^{(\alpha+1)}}{\alpha+1}$$

A problem with one peak is

$$\int_1^2 \frac{10^\alpha}{(x-\lambda)^2+10^{2\alpha}}dx$$

while one with three such peaks is

$$\int_1^2 \sum_{i=1}^3 \frac{10^\alpha}{(x-\lambda_i)^2+10^{2\alpha}}dx$$

Fig. 4. Examples of Test Integrands

integrand such as weight functions, oscillatory or singular integrands. For example, QAWO is an automatic adaptive integrator for integrands with oscillatory sine or cosine functions and QAWS is one for functions with explicit algebraic and/or logarithmic endpoint singularities. There are other routines that use transformations, Newton–Cotes quadrature, Clenshaw–Curtis quadrature, monotone stable formulas, Patterson's quadrature formulas and differential equation solvers. It can be seen from Fig. 3 that the number of automatic algorithms far outnumber the non–automatic routines. Also, there are more algorithms available for integration over finite intervals than for infinite intervals.

Of the 20 Type 2 routines, 14 are automatic routines and 6 are non–automatic routines. Most of these routines evaluate the integral by approximating the data points provided by some representation such as piecewise polynomials, overlapping parabolas, cubic splines, Hermite functions and B–splines. The quality of the answer is therefore dependent on the efficiency of the approximation technique.

4.2.2 *Test Problems.* We have used a wide variety of test integrands, most of them with special properties. The total number of test integrands used in this study was 286. The integrals were selected so that they exhibit interesting or common features such as smoothness (or its absence), singularities, discontinuities, peaks, and oscillation. Some of the functions were selected so that they satisfy the special considerations on which some algorithms are designed. For example, routine QDAWO requires that its argument contain a sine or a cosine. Most of the functions are parameterized which generates families of integrands with similar features and characteristics – this aids in the generalization of the system. The number of experiments thus rises to a huge number (286 functions times 124 routines times 10 levels of accuracy = 354,640). However, as mentioned before, some routines are not applicable to quite a few integrands and results for a whole family of integrands can be quickly and automatically determined by scripting programs in GAUSS. Examples of test integrands are given in Fig. 4.

4.2.3 *Sample Experiment.* A particularly interesting example that is simple and illustrates the methodology is the effect of the QUADPACK routine QNG applied

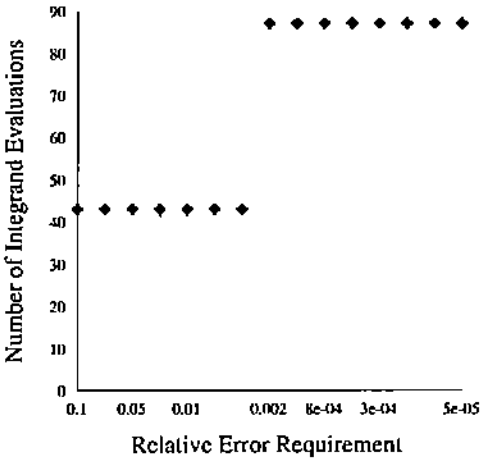


Fig. 5. Performance of QNG for the sample integral evaluation

to the integral

$$\int_0^1 x^{1/2} log(x) dx = \frac{4}{9}.$$

When the relative error requirement is reduced from 0.1 down to $3E-05$, the number of function evaluations varies as shown in Fig. 5.

It is to be noted that QNG is a simple non–adaptive automatic quadrature routine based on a sequence of rules with increasing degrees of algebraic precision. As shown in Fig. 5, an error requirement of 0.1 results in the evaluation of the integrand at 43 points. Incidentally, this also satisfies all error requirements till 0.005. For a more stringent requirement of 0.002, we get 87 integrand evaluations which actually produces a relative error of $0.22E-04$. This explains the pattern of the graph in Fig. 5. Also, when we impose an error requirement of $1E-05$, we get the following output from the PES:

```
integral approximation = -0.44444460E+00
Estimate of absolute error = 0.22E-04
Number of function evaluations =    87
error code = 1
```

This error code (which was 0 for the previous error requirements), indicates that the maximum number of function evaluations has been achieved. For QNG, this means that the maximum number of 'steps' have been executed and that the integral is probably too difficult to be computed by this routine. For an adaptive routine, this means that the limit on the number of interval subdivisions has been achieved, which has been a priori set within the integrator.

We provide all of the above information as predicate logic statements to GAUSS. There is also a 'threshold predicate' which indicates the 'breakdown' point for a routine with a certain problem. In this manner, we take each integrand and impose varying requirements on the relative accuracy to determine the number of nodes

needed to obtain the desired accuracy. The abrupt change in the above graph at a certain point is due to the non–adaptive nature of the integration routine. For adaptive routines, smoother transitions are observed.

For example, the above information is coded as:

```
nfe(P1,QNG,0.1,43).
nfe(P1,QNG,0.05,43).
....
nfe(P1,QNG,0.002,87).
breakdown(P1,QNG,1E-05).
errorcode(P1,QNG,1E-05,1).
...
```

(Note: The nfe predicate gives the number of function evaluations.) This information does not directly determine which routine is better for problem P1. To determine this, we introduce high–level rules which give rise to consequent predicates such as:

```
bestmethod(P1,QNG,0.1).
....
```

Various other error codes are possible from the routines – they could mean occurrence of excessive roundoff error, too small an interval to be subdivided (by an adaptive routine), difficulties encountered in integrand behavior, divergent (or slowly convergent) results, or a limiting number of cycles obtained. Many of these error codes are input to GAUSS as valuable information that it might (possibly) use to determine whether a particular routine is appropriate for a certain problem.

Special care also has to be taken if the integrand is not defined at one or more points in the integration interval. If the routine is not one that is particularly appropriate for singular integrals, then the function value (at the points where it is undefined) has to be set equal to the limit value of the function. Whenever this limit does not exist or is infinite, zero is substituted. Extensive literature on test problems helps to identify such problematic integrands.

4.3 Automatic Feature Determination

The AFD module of GAUSS determines features of integrands and the domains of integration. We are not attempting to conduct "feature discovery" here – rather, we are more interested in automating the determination of 'known', predefined features of functions and domains. Some of the more useful features that are specially relevant to this problem domain (and which affect the applicability of algorithms) are — whether the integrand can be expressed as $w(x)f(x)$ with several desirable features (such as $w(x)$ being one of several weight functions and f smooth on $[a, b]$), whether the integrand is smooth in $[a, b]$, whether we know the location of the singularities of f, whether we know the location (if any) of singularities of f', whether f has end–point singularities, whether f exhibits an oscillatory behavior of non–specific type, and whether the range of integration $[a, b]$ is finite, semi–infinite or infinite. The AFD module utilizes three main mechanisms to determine such features of functions - symbolic analysis, 'numeric' techniques and user input.

4.3.1 *Symbolic Techniques.* These techniques can be applied for symbolic functions as used by a Type 1 algorithm. In other words, we have a closed mathematical expression for the function to be integrated and it is possible to write an algorithm to calculate this function. Symbolic techniques help determine the first few characteristics described previously by taking advantage of the rule–based programming interface to packages like *Mathematica*. In this approach, calculations are executed by specifying collections of 'transformation' rules. When a given input pattern matches a pattern in a certain rule, then that expression is transformed by the rule. Such rules in *Mathematica* can be applied just once to a given expression or can be applied repeatedly until the expression no longer changes. This approach allows us to construct pattern matches for determining many important features. For example, the set of discontinuities of various functions can be expressed as set functions of discontinuities of their constituent functions all the way down to the 'basis' functions. Zeros of polynomials and other expressions required by these rules are determined from the root finders present in *Mathematica*. One root finder, for example, attempts to find the roots of the given expression using Newton's method and its variants. Newton's method, however requires that the function be analytic or at least contain analytic regions around each root. It also requires that the gradient of the function be calculated. If the gradient of the function is not 'well–behaved' at some points, then Newton's method does not work. We can then resort to other techniques such as the secant method or Brent's algorithm. These do not require that *Mathematica* calculate gradients as does Newton's method. Also, we are interested in determining the behavior of functions only within the domain of integration. Therefore, one stopping criteria for these root finders is that they terminate the iterations if the approximations go beyond the range of interest.

However, there are various occasions where the above rules fail to work, mainly because the root finders in *Mathematica* exit with an error code. Whenever a rule fails, the features are determined by other means and input to the system. Currently the rules symbolically determine the following features of functions (with limitations):

(1) Whether the integrand can be expressed as $w(x)f(x)$ with several desirable features such as – $w(x)$ being one of several weight functions and f is smooth on $[a, b]$ – This is currently implemented by attempting factorization of the integrand. Example choices for $w(x)$ are $sin(ax)$, $cos(ax)$, $(x-a)^{\alpha}(b-x)^{\beta}$ and $1/(x-c)$.

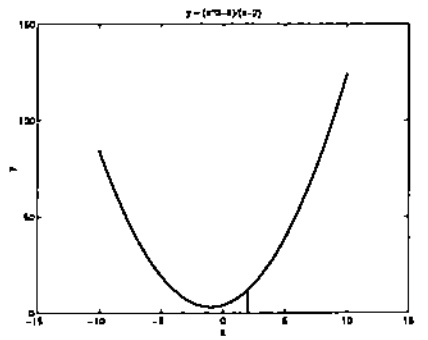
(2) The location of discontinuities (if any) of f and f' in $[a, b]$ – GAUSS can distinguish between two kinds of discontinuities – essential and removable. Intuitively, the first kind are those that make the curve look discontinuous in appearance. In the second case, the curves, while mathematically undefined at some points, are actually smooth and differentiable (the derivative exists) at these points.

(3) The location of the singularities of f

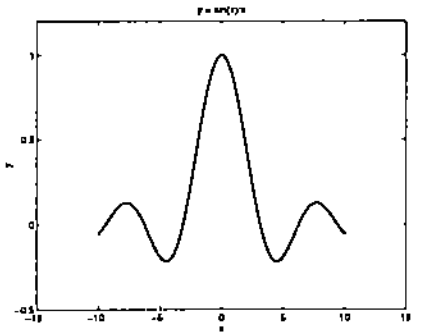
(4) Are there singularities of the derivatives of f?

(5) The location of the singularities of f'

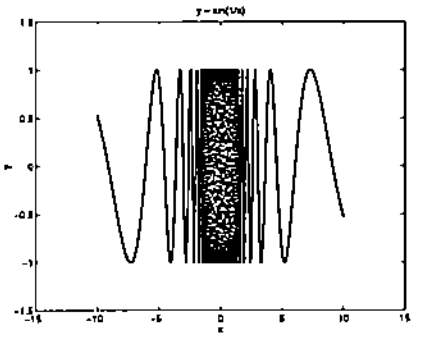
(6) Does f have end–point singularities? – This is determined by checking the derivative of f at the limits of the domain of integration.



```
In[1] := AFD[(x^3-8)/(x-2)]
Out[1]=
The function has a disc.  at 2(x).
Lt (x^2 + 2 x + 4)  = 12
x -> 2
Since the limit exists,
this is a removable discontinuity.
At other points, this function can be
expressed as (x^2 + 2 x + 4).
```



```
In[2] := AFD[Sin[x]/x]
Out[2]=
The function has a disc.  at 0(x).
Lt      Sin[x]
        ------    = 1
x -> 0    x
Since the limit exists,
this is a removable discontinuity.
```



```
In[3]:= AFD[Sin[(1/x)]]
Out[3]=
The function has a disc.  at 0(x).
Lt            1
       Sin  --- =
x -> 0       x
does not exist.
Since the limit does not exist,
this is a non-removable discontinuity.
```

Fig. 6: Symbolic analysis of three functions in the AFD module of GAUSS. The In statements refer to the input to *Mathematica* and the Out statements are the responses. The notation $0(x)$ refers to the fact that x is evaluated at 0.

(7) Whether the range of integration is finite, semi-infinite or infinite.

A transcript of a session with the symbolic analyzer in the AFD module is given in Fig. 6. The evaluation of the function $sin(\frac{1}{x})$ in the last example of Fig. 6 provides interesting insights into the behavior of the AFD module. As can be seen, the function is not defined for $x = 0$. If any neighborhood of $x = 0$ is chosen, then $y = sin(\frac{1}{x})$ takes all values between -1 and $+1$ there. Another way of saying this is that the x-axis serves as an asymptote of the function $sin(\frac{1}{x})$. In this case, the AFD module has correctly determined that this is not a discontinuity that can be 'removed'. This information when provided to the KB, causes it to select those quadrature algorithms that are specially suited for such functions.

While providing much useful information, it should be noted, however, that the symbolic analysis module is not applicable if the integration problems are of Type 2. Also, if $f(x)$ is a 'black-box' executable routine (meaning that it cannot be inspected), then we are not able to do symbolic analysis on the function. Moreover, symbolic analysis proves to be very costly if the functions being inspected are more complicated than those we have here.

4.3.2 *'Numeric' Techniques.* These methods are used to determine features of functions without using their symbolic representation. These techniques are able to ascertain if the functions exhibit characteristics such as oscillatory behavior, exponential patterns, etc., without analyzing them symbolically. Rather, they take as input, a graphical representation of the function and classify them into one or more of several categories. We utilize a primitive kind of method, drawing from earlier research on handwritten character recognition. The approach is as follows:

(1) Draw the integrand in the domain of interest and map it onto a rectangular grid of 1000 by 1000 pixels.

(2) 'Flood' a 1000×1000 binary array based on the bit positions of the appropriate elements in the graphic. If a bit is lighted, enter a 1 in the corresponding position; else, enter a 0.

(3) With this binary array, compute the 'invariant moments' $\mu_1, \mu_2, \mu_3, \mu_4, \mu_5, \mu_6$. Moment invariants are dimensionless quantities and serve to represent the image and they are invariant under translation, scaling and rotation operations. In this study, we are mostly interested in the first two transformations (since 'rotating' a singular function by 90° causes it to cease being so!!). Thus, only those moment invariants that are 'fixed' under translation and scaling are considered.

(4) Having thus represented a function $f(x)$ by a few features (the moments), we train a neural network to classify functions into one of several categories.

The net effect of this is that we have some features that can be used to determine if a new function behaves in a manner similar to that of any previously seen function. Oscillatory behavior, singularities and asymptotic behavior of functions over infinite domains can be easily detected by this technique. The total number of categories defined is 20. Examples of categories are logarithmic functions, trigonometric functions, exponential functions, miscellaneous transcendental functions, functions with $x^{n/m}$ and $(a+bx)^m$, functions with (x^2-a^2) and x^m, functions with (a^2-x^2) and x^m and other algebraic functions. Some of the above categories include functions that may or may not have singularities (depending on whether the mathematical pattern occurs in the numerator or the denominator). Two categories relate to singularities in the derivative of the function.

A high classification accuracy of 89.40% is achieved with the given test problem set. We utilize a conventional three layer feed forward neural network and train it by an algorithm that our group has devised. This is a neuro–fuzzy classification algorithm for neural networks and has been shown to produce results comparable to benchmark methods such as RProp, C4.5, etc. [5]. Moreover, our algorithm handles mutually non–exclusive categories that are very common in scientific domains, i.e., one 'pattern' can belong to more than one category at the same time. The

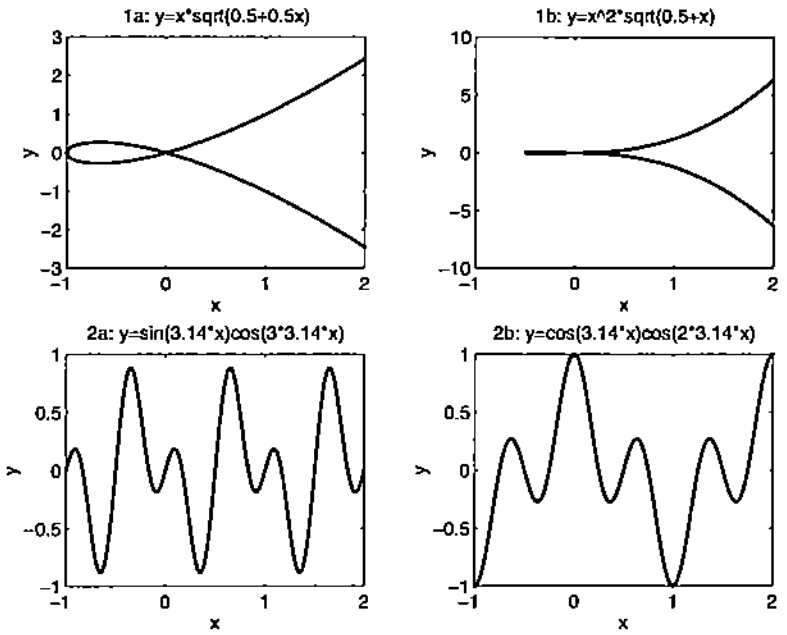


Fig. 7. Plots of functions that are 'similar-looking'

total number of functions 'plotted' was 1000. To test the feasibility of the current approach, this set of 1000 was initially split into two parts - 650 functions and 350 functions. The first set was used as the training set and the second part was used for testing. This gave rise to a generalization accuracy of 82% and a recall accuracy of 86.15%, bringing the overall accuracy to 84.70%. Having thus realized that the method yields sufficient accuracy, the system was 're-trained' with the whole set of 1000 functions, which gave rise to the accuracy of 86.87%. (Most of the errors were caused by the neural network getting confused between logarithmic and exponential functions).

Example: Certain functions whose moment invariant features were deemed to be quite similar are shown in Fig. 7.

Function	$log \mid \mu_1 \mid$	$log \mid \mu_2 \mid$	$log \mid \mu_3 \mid$	$log \mid \mu_4 \mid$	$log \mid \mu_5 \mid$	$log \mid \mu_6 \mid$
1a	-6.746	-14.593	-26.353	-24.804	-50.846	-32.106
1b	-6.456	-14.219	-26.854	-24.855	-48.954	-32.667

The above table depicts the moment invariant features of the two functions whose graphs are shown in the top row of Fig. 7. As can be seen, these values are indeed quite similar. Moreover, this approach can be used to 'evolve' clusters or families of functions. That is, instead of specifying to the system what kind of function an image represents, we can make it figure this out for itself and let it create a new cluster if it deems that the function currently presented is one not encountered before.

4.3.3 *User Input.* In our design of the GAUSS system, we introduced a scheme wherein the user could have a say in the automatic feature identification process. The reason for this is two–fold: (i) the various submodules of the AFD module could lead to uncovering 'conflicting' features and (ii) the symbolic component sometimes 'exits' with an error code indicating that the function could not be analyzed. In either case, we require that the user intervene and supply the desired features or provide alternative mechanisms to obtain the same. We are also working on using confidence factors to guide the AFD module so that user involvement can be minimized. A slightly different approach to automatic feature identification is described in the section on future work. All features determined by these three means are encoded as logic rules in GAUSS.

4.4 The "Expert" Methodology

The approach in GAUSS is a highly knowledge intensive one. Its paradigm of learning is inductive logic programming [2], a classical technique that has recently seen a resurgence of interest, mainly due to reports of successful developments and applications. Background knowledge is represented as a set of predicate definitions and positive and negative examples. Given this, an ILP system attempts to construct a predicate logic formula so that all the positive examples can be logically derived from the background knowledge and no negative example can be logically derived. The advantages of such a system lie in the generality of representation of background knowledge. One major application of ILP is automatic program synthesis from examples — in other words, to induce logic programs from sample input and output combination pairs. While a major drawback of ILP systems is their inability to handle numerical data and quantitative attribute information, our current methodology makes up for these drawbacks by delegating such duties to the PES and the AFD modules. The ILP system used by GAUSS is Golem, an empirical ILP system, i.e., a batch non–interactive system that learns the definition of a single predicate from a large collection of examples. It has been observed that empirical ILP systems are more suited to practical applications. Golem uses the basic principle of relative least general generalization which is a form of cautious generalization. The predicate that Golem attempts to learn in GAUSS is the `method` function which determines the best method for a given `Problem`.

4.4.1 *Observations.* The entire gamut of performance data gathered, and symbolic features extracted (or input by the 'user') are coded as predicate logic rules. Sample rules extracted by GAUSS are:

```
method(Problem, dqagk6) :-
        smooth(Problem),
        accuracy(Problem, -8),
        image(Problem, type3),
        nosing(Problem),
        noderivsing(Problem),
        range(Problem, finite).

method(Problem, dqagk1) :-
        smooth(Problem),
```

```
        accuracy(Problem, -8),
        image(Problem, type2),
        sing(Problem),
        derivsing(Problem),
        range(Problem, finite).

method(Problem, qags) :-
        accuracy(Problem, Acc),
        image(Problem, type2),
        sing(Problem),
        endptsing(Problem),
        noderivsing(Problem),
        range(Problem, finite).

method(Problem, qags) :-
        accuracy(Problem, Acc),
        image(Problem, type6),
        sing(Problem),
        endptsing(Problem),
        noderivsing(Problem),
        range(Problem, finite).
```

The first rule indicates that the method DQAGK6 is most suitable to problem `Problem` if `Problem` has 'behavior of TYPE3', the function is smooth, the range of integration is finite, there are no singularities of the function or its derivative and the accuracy level required is 10^{-8}. TYPE3 represents oscillatory behavior of a certain type and DQAGK6 is a simple globally adaptive QUADPACK routine (with the parameter KEY set to 6).

The last two rules are of particular interest – GAUSS has determined that the routine QAGS is good for any integrand with any error criterion if the function has end point singularities. True enough, TYPE2 and TYPE6 are types in imagistics that correspond to functions that have singularities and have logarithms and exponentials in them. Such rules have been found to be 'faithful' to the data gathered in this study.

η_{best}	$\eta_{2ndbest}$	$\eta_{reasonable}$	$\eta_{hopeless}$
87%	7%	3%	3%

Based on these rules, algorithm selection was conducted with the entire set of problems considered. The table above shows the accuracy figures obtained. The first entry indicates that GAUSS decided on the best known algorithm for 87% of the cases, selected the second best algorithm 7% of the time, and so on. It is seen that GAUSS selects a 'good' algorithm most of the time.

4.4.2 *Knowledge Discovery.* On examining the rules mined by GAUSS, we observe several heuristics about the domain of numerical integration and associated

algorithms. First, the rules inferred by GAUSS are quite similar to the decision tree models and ready reference 'lookup tables' formulated by domain experts. Second, most rules discovered correspond to popular opinion and general knowledge about numerical integration routines. It was found, for example that the adaptive algorithms use fewer function evaluations to achieve high–accuracy results than their non–adaptive counterparts; conversely, they use more evaluations to meet low–accuracy constraints. A high accuracy adaptive algorithm has been found to be more suitable for an oscillating integrand. This could possibly be due to the fact that in an oscillating function, subdivisions are spread over the entire domain of integration and hence a smaller number of subdivisions are required to achieve a fairly high degree of accuracy. Conversely, integrands with singularities or peaks are more amenable to low and medium accuracy adaptive routines. There are many more such observations, and we have reproduced only the most interesting here. Finally, GAUSS has helped identify 'redundant' algorithms, i.e., algorithms which perform almost exactly the same for the test functions considered in this work. For example, on examining the output generated from Golem, it was found that the rules selecting the algorithms DPCHIA, DCSITG and DCSPQU contained the same antecedents. On further examination, one sees that the performance data for these algorithms is nearly the same. All of these are Type 2 routines – DPCHIA evaluates the given integral using piecewise cubic Hermite functions, DCSITG evaluates the integral of a cubic spline and DCSPQU also uses spline interpolation. Thus, these three routines are mathematically very similar and incidentally, they yield the best overall performance for problems of Type 2.

5. FUTURE WORK AND CONCLUSIONS

Systems like GAUSS find application in several CS&E problems. In addition to selecting a suitable algorithm to solve a given problem, GAUSS also conducts scientific discovery in the domain of numerical integration and has come up with several broad heuristics that encode the domain knowledge. These results seem to indicate favorably toward the use of ILP modules like Golem. The difference between our earlier approaches [5] and this is that the newer method is highly knowledge intensive and, more importantly, uses relational descriptions of objects to influence the decision making process. In our earlier approaches, the background knowledge about the domain could be expressed in only a limited form. We also feel that data mining was largely a success in this domain primarily because data was specifically collected for the purpose of mining for knowledge and not as a byproduct of other tasks, as is often the case. It is our belief that systems like GAUSS will form a major component of future PSEs for computational science.

5.1 Future Research Directions

—The 'imagistic component' can be enhanced and true 'discovery' of function classes (clustering) can be attempted. In the current schema, some more function classes may be introduced into the numeric component of the AFD module such as inverse trigonometric functions, hyperbolic and inverse hyperbolic functions, and various other combinations of the basic functions.

—The current algorithms are assumed to be 'monolithic' ones that are applied to the entire domain of integration. However, there exist occasions when a domain

needs to be split up and different methods used to evaluate the integrals at different sub–intervals. Such 'composite' rules can be included in this approach.

—In research on similar lines, Abelson et al., [1], have shown how their systems for intelligent scientific computing automatically construct numerical procedures through liberal use of high–order procedural abstractions. This methodology would be appropriate for our domain. A potential system could 'chain' together necessary routines to determine features of functions. An agent based implementation is a possible means of achieving this task.

—Automatic feature discovery (as opposed to mere identification) is another active area of research that can be pursued in this domain. It might be the case that some interesting (and important) features of functions do not have a 'neat' representation in the domain space but are nevertheless critical to understanding why some algorithms perform better on some problems than others.

REFERENCES

[1] H.A. Abelson, M. Eisenberg, M. Halfant, J. Katzenelson, E. Sacks, G.J. Sussman, J. Wisdom, and K. Yip. Intelligence in Scientific Computing. *Communications of the ACM*, 32(5):546–562, May, 1989.

[2] I. Bratko and S. Muggleton. Applications of Inductive Logic Programming. *Communications of the ACM*, 38(11):65–70, November, 1995.

[3] P. J. Davis and P. Rabinowitz. *Methods of Numerical Integration*. Academic Press, Orlando, Florida, 1984. Second Edition.

[4] E. Gallopoulos, E. Houstis, and J.R. Rice. Computer as Thinker/Doer: Problem-Solving Environments for Computational Science. *IEEE Computational Science and Engineering*, vol.1(2):pp.11–23, 1994.

[5] A. Joshi, S. Weerawarana, N. Ramakrishnan, E.N. Houstis, and J.R. Rice. Neuro–Fuzzy Support for PSEs: A Step Toward the Automated Solution of PDEs. *IEEE Computational Science and Engineering*, vol.3(1):pp.44–56, 1996.

[6] E. Kant, R. Keller, and S. Steinberg. *Working Notes of the AAAI Fall Symposium on Intelligent Scientific Computation*. American Association for Artificial Intelligence, 1992. Cambridge, Massachusetts.

[7] J.R. Rice. A Metalgorithm for Adaptive Quadrature. *Journal of the ACM*, vol.22(1):pp.61–82, 1973.

[8] J.R. Rice. The Algorithm Selection Problem. *Advances in Computers*, 15:pp.65–118, 1976.

[9] R.E. Valdes-Perez. Generic Tasks of Scientific Discovery. *Working Notes of AAAI Spring Symposium on Systematic Methods of Scientific Discovery*, 1995.

[10] R.E. Valdes-Perez. Computer Science Research on Scientific Discovery. *Knowledge Engineering Review*, vol.11(1):pp.57–66, 1996.