Machine Learning for Multi-stage Selection of Numerical Methods[*]

Victor Eijkhout
Texas Advanced Computing Center, The University of Texas at Austin
USA

Erika Fuentes
Innovative Computing Laboratory, University of Tennessee
USA

Abstract
In various areas of numerical analysis, there are several possible algorithms for solving a problem. In such cases, each method potentially solves the problem, but the runtimes can widely differ, and breakdown is possible. Also, there is typically no governing theory for finding the best method, or the theory is in essence uncomputable. Thus, the choice of the optimal method is in practice determined by experimentation and 'numerical folklore'. However, a more systematic approach is needed, for instance since such choices may need to be made in a dynamic context such as a time-evolving system.
Thus we formulate this as a classification problem: assign each numerical problem to a class corresponding to the best method for solving that problem.
What makes this an interesting problem for Machine Learning, is the large number of classes, and their relationships. A method is a combination of (at least) a preconditioner and an iterative scheme, making the total number of methods the product of these individual cardinalities. Since this can be a very large number, we want to exploit this structure of the set of classes, and find a way to classify the components of a method separately.
We have developed various techniques for such multi-stage recommendations, using automatic recognition of super-clases. These techniques are shown to pay off very well in our application area of iterative linear system solvers.
We present the basic concepts of our recommendation strategy, and give an overview of the software libraries that make up the Salsa (Self-Adapting Large-scale Solver Architecture) project.

1. Introduction

In various areas of numerical analysis, there are several possible algorithms for solving a problem. Examples are the various direct and iterative solvers for sparse linear systems, or routines for eigenvalue computation or numerical optimization. Typically, there is no governing theory for finding the best method, or the theory is too expensive to compute. For instance, in iterative linear system solving, there are many preconditioners and iterative schemes; the convergence behaviour is determined by

[*]This work was funded in part by the Los Alamos Computer Science Institute through the subcontract # R71700J-29200099 from Rice University, and by the National Science Foundation under grants 0203984 and 0406403.

the decomposition of the right-hand side in terms of the eigenvectors of the preconditioned operator. However, computing this spectrum is expensive, in fact more expensive than solving the problem to begin with, and the dependency of the iterative scheme on this decomposition is not known in explicit form.
Thus, the choice of the optimal method is in practice determined by experimentation and 'numerical folklore'. However, a more systematic approach is needed, for instance since such choices may need to be made in a dynamic context such as a time-evolving system. Certain recent efforts have tried to tackle this situation by the application of various automatic learning techniques Arnold et al. (2000); Bhowmick et al. (2006); Dongarra et al. (2006); Houstis et al. (2000); Ramakrishnan & Ribbens (2000); Wilson et al. (2000). Such methods extract the problem features, typically relying on a database of prior knowledge, in order to recommend a suitable method. However, we may characterize this earlier research as either being too general and lacking in specific implementations, or being too limited to one area, lacking the conceptual necessary for generalization to other areas.
In this paper we present a theoretical framework for such recommender systems, as well as libraries that we have developed that embody these concepts. Our framework is more general than what has been studied before, in that we consider methods to be structured objects, consisting of a composition of algorithms. Correspondingly, we describe strategies for recommending methods that acknowledge this structure, rather than considering methods to be elements out of a flat set of elements.
In addition to the abstract mathematical description, we propose software interfaces implementing the various entities in the framework. Some of this software has been realized in our Salsa (Self-Adapting large-scale Solver Architecture) project; see Salsa Project (n.d.a;n).
We start by giving a brief introduction to our field in section 2, after which we formalize the problem in sections 3 and 4. Section 5 has a general discussion of intelligent method selection, while section 6 details strategies for recommending composite method objects. Practical experiments are reported in section 7.

2. Application: linear system solving

Many problems in engineering and in sciences such as physics, astronomy, climate modeling, reduce to a few basic numerical problems:

- Linear system solving: given a square, nonsingular, matrix A and a vector f, find a vector x such that $Ax = f$. Linear system solving also appears as a subproblem in nonlinear systems and in time-dependent problems.
- Eigenvalue calculations: given a matrix A, find the pairs $\langle u, \lambda \rangle$ such that $Au = \lambda u$. Some eigenvalue calculations again lead to linear systems.
- Numerical optimization: find the vector x that maximizes a function $f(x)$, often linear: $f(x) = a^t x$, under certain constraints on x: $g(x) = 0$.

For each of these numerical problems, more than one solution algorithm exists, often without a sufficient theory that allows for easy selection of the fastest or most accurate algorithm. This can be witnessed in the NEOS server Czyzyk et al. (1996; 1998), where a user can choose from various optimization algorithm, or the PETSc numerical toolkit Balay et al. (1999); Gropp & Smith (n.d.) which allows the user to combine various predefined solver building blocks. The aim is then to find an algorithm that will reliably solve the problem, and do so in the minimum amount of time.

> Since several algorithm candidates exist, without an analytic way of deciding on the best algorithm in any given instance, we are looking at the classification problem of using non-numerical techniques for recommending the best algorithm for each particular problem instance.

In this chapter we will focus on linear system solving. The textbook algorithm, Gaussian elimination, is reliable, but for large problems it may be too slow or require too much memory. Instead, often one uses *iterative solution methods*, which are based on a principle of successive approximation. Rather than computing a solution x directly, we compute a sequence $n \mapsto x_n$ that we hope converges to the solution. (Typically, we have residuals r_n that converge to zero, $r_n \downarrow 0$, but this behaviour may not be monotone, so no decision method can be based on its direct observation.)

2.1 Feature selection

As noted above, there is no simple, efficiently computable, way of deciding on the best algorithm for solving a linear system. Theoretically it is known that the behaviour of iterative solvers depends on the expansion of the right hand side in the eigenvector basis, but computing this expansion is far more expensive than solving the linear system to begin with. Also, there are a few negative results Greenbaum & Strakos (1996) that stand in the way of easy solutions.
However, using a statistical approach, we have no lack of features. For instance, we typically are concerned with sparse matrices, that is, matrices for which only a few elements per row are nonzero. Thus, we can introduce features that describe the sparsity structure of the matrix. From a theoretical point of view, the sparsity structure is not relevant, but in practice, it can correlate to meaningful features. For instance, high order finite element methods lead to larger numbers of nonzeros per row, and are typically associated with large condition numbers, which are an indication of slow convergence. Thus, indirectly, we can correlate these structural features with solver performance.

2.2 Multilevel decisions

As we remarked above, in essence we are faced with a classification problem: given a linear system and its features, what is the best method for solving it. However, the set of classes, the numerical methods, has structure to it that makes this a challenging problem.
It is a truism among numerical mathematicians that an iterative method as such is virtually worthless; it needs to be coupled to a 'preconditioner': a transformation B of the linear system that tames some of the adverse numerical properties. In its simplest form, applying a preconditioner is equivalent to solving the equivalent system

$$\tilde{A}x = \tilde{f}, \qquad \text{where } \tilde{A} = BA \text{ and } \tilde{f} = Bf$$

or

$$\tilde{A}\tilde{x} = f, \qquad \text{where } \tilde{A} = AB \text{ and } \tilde{x} = B^{-1}x \text{ so } x = B\tilde{x}.$$

Since there are many choices for the preconditioner, we now have a set of classes of cardinality the number of preconditioners times the number of iterative schemes.
Additionally, there are other transformations such as permutations P^tAP that can be applied for considerations such as load balancing in parallel calculations. In all, we are faced with a large number of classes, but in a set that is the cartesian product of sets of individual decisions that together make up the definition of a numerical method. Clearly, given this structure, treating the set of classes as just a flat collection will be suboptimal. In this chapter we consider the problem of coming up with better strategies, and evaluating their effectiveness.

2.3 Software aspects

The research reported in this paper has led to the development of a set of software libraries, together called the SALSA project, for Self-Adapting Large-scale Solver Architecture. The components here are

- Feature extraction. We have written a custom library, AnaMod, for Analysis Modules for this.

- Storage of features and runtime results. During runtime we use our NMD, for Numerical Meta-Data, library. For permanent storage we are using a standard MySQL database.
- Learning. We use various Matlab toolboxes here, though ultimately we want to move to an open source infrastructure.
- Process control. The overall testbed that controls feature extraction, decision making, and execution of the chosen algorithms, is a custom library named SysPro, for System preProcessors.

Since the overall goal is to optimize solution time, the feature extraction and process control software need to be fast. Though we will not elaborate on this point, this means that we occasionally have to trade accuracy for speed.

3. Formalization

In this section we formalize the notion of numerical problems and numerical methods for solving problems. Our formalization will also immediately be reflected in the design of the libraries that make up the Salsa project.

3.1 Numerical Problems and Solution Methods

The central problem of this paper, selecting the optimal algorith for a numerical problem, implies that we have a specific problem context in mind. That can be linear system solving, eigenvalue computations, numerical quadrature, et cetera. For each of these areas, there is a definition of a numerical problem: in the case of linear systems, a problem will be a pair $\langle A, b\rangle$ of matrix and righthand side; for eigenvalue problems it denotes a matrix plus optionally an interval for the eigenvalues, et cetera.
It would be pointless to capture this diversity of problems in a rigorous definition, so we will use a leave the concept of 'numerical problem' largely undefined. We denote the space of numerical problems by

$$\boxed{\mathbb{A}\text{: the set of numerical problems in a class}}$$

where the exact definition of this will depend on the problem area.
The implementation of numerical problems is similarly domain-specific. As an example, we give the following possible definition in the area of linear system solving: [1]:

```
struct Problem_ {
    LinearOperator A;
    Vector RHS,KnownSolution,InitialGuess;
    DesiredAccuracy constraint;
};
typedef struct Problem_* Problem;
```

Corresponding to the problem space, there is a result space

$$\boxed{\mathbb{R} = \mathbb{S} \times \mathbb{T}\text{: results, the space of solutions plus performance measurements}} \tag{1}$$

containing computed solutions and performance measurements of the computing process.
The following is an illustration of results and performance measurements for the case of linear system solving.

[1] After this example we will tacitly omit the `typedef` line and only give the structure definition.

```
struct PerformanceMeasurement_ {
    int success;
    double Tsetup,Tsolve;
    double *ConvergenceHistory;
    double BackwardError,ForwardError;
}
struct Result_ {
    Vector ComputedSolution;
    PerformanceMeasurement performance;
}
```

Some component of the performance measurement, typically a timing measurement, will be used to judge optimality of solution methods.
Having defined problems and results, we define the space of methods as the mappings from problems to results.

$\boxed{\mathbb{M} = \{\mathbb{A} \mapsto \mathbb{R}\}}$: the space of methods (potentially) solving the class of numerical problems

We allow for methods to fail to compute the solution, for instance by the divergence of an iterative method, or exceeding a set runtime limit. We formalize this by introducing a timing function $T(A,M)$, denoting the time that it takes method M to solve the problem A, where in case of failure we set $T(A,M) = \infty$.
We will defer further discussion of the method space $\mathbb{M}$ for section 4.

3.2 Features and features extraction

As in natural in any Machine Learning context, we introduce the concept of problem features, which will be the basis for any recommendation strategy.

$\mathbb{F}$: the space of feature vectors of the numerical problems

and

$\Phi : \mathbb{A} \mapsto \mathbb{F}$: a function that extracts features of numerical problems

where $\Phi(A) = \bar{x} \in \mathbb{F}$ is the feature vector describing the numerical problem.
Feature space can be complicated in structure: elements of $\bar{x}$ can be real numbers (for instance matrix elements), positive real (norms), integer (quantities related to matrix size and sparsity pattern), or elements of a finite set of choices. They can even be array-valued in any of these types.
The NMD (Numerical MetaData) library Eijkhout & Fuentes (2007) provides an API for feature storage, including various utility functions.
We posit the existence of a parametrized function that can compute one feature at a time:

```
ComputeQuantity(Problem problem,
  char *feature,ReturnValue *result,TruthValue *success);
```

Here, the `ReturnValue` type is a union of all returnable types, and the `success` parameter indicates whether the quantity was actually computed. Computation can fail for any number of reasons: if sequential software is called in parallel, if properties such as matrix bandwidth are asked of an implicitly given operator, et cetera. Failure is not considered catastrophic, and the calling code should allow for this eventuality.
For a general and modular setup, we do not hardwire the existence of any module. Instead, we use a function

```
DeclareModule(char *feature,ReturnType type,
    void(*module)(Problem,char*,ReturnValue*,TruthValue*));
```

to add individual feature computations to a runtime system.
The AnaMod (Analysis Modules) library Eijkhout & Fuentes (2007) implements the above ideas, though in a slightly different form.

4. Numerical methods

In a simple-minded view, method space can be considered as a finite, unordered, collection of methods $\{M_1,\ldots,M_k\}$, and in some applications this may even be the most appropriate view. However, in the context of linear system solving a method is a more structured entity: each method consists at least of the choice of a preconditioner and the choice of an iterative scheme (QMR, GMRES, et cetera), both of which are independent of each other. Other possible components of a method are scaling of the system, and permutations for improved load balancing. Thus we arrive at a picture of a number of preprocessing steps that transform the original problem into another one with the same solution – or with a different solution that can easily be transformed into that of the original problem – followed by a solver algorithm that takes the transformed problem and yields its solution.
This section will formalize this further structure of the method space.

4.1 Formal definition

Above, we had defined $\mathbb{M}$ as the set of mappings $\mathbb{A} \mapsto \mathbb{R}$. We now split that as the preprocessors

$$\boxed{\mathbb{P} = \{\mathbb{A} \mapsto \mathbb{A}\}\text{: the set of all mappings from problems into problems}}$$

and the solvers[2]

$$\boxed{\mathbb{K} = \{\mathbb{A} \mapsto \mathbb{R}\}\text{: the set of all solvers}}$$

To illustrate the fact that the preprocessing stages are really mappings $\mathbb{A} \mapsto \mathbb{A}$, consider a right scaling D of a linear system, which maps the problem/solution tuple $\langle A,b,\bar{x}\rangle$ to $\langle AD,b,D^{-1}x\rangle$.
To model the fact that we have different kinds of preprocessors, we posit the existence of subsets

$$\mathbb{P}_i \subset \mathbb{P},$$

and we will assume that the identity mapping is contained in each. For instance, one $\mathbb{P}_i$ could be the set of scalings of a linear system:

$$\mathbb{P}_4 = \{'\text{none}','\text{left}','\text{right}','\text{symmetric}'\}$$

Other possibilities are permutations of the linear system, or approximations of the coeffient matrix prior to forming the preconditioner.
Applying one preprocessor of each kind then gives us the definition of a method:

$$m \in \mathbb{M}\colon\ m = k \circ p_n \circ \cdots \circ p_1, \quad k \in \mathbb{K}, p_i \in \mathbb{P}_i \tag{2}$$

We leave open the possibility that certain preprocessors can be applied in any sequence (for instance scaling and permuting a system commute), while for others different orderings are allowed but not equivalent. Some preprocessors may need to be executed in a fixed location; for instance, the computation of a preconditioner will usually come last in the sequence of preprocessors.
Typically, a preprocessed problem has a different solution from the original problem, so each preprocessor has a backtransformation operation, to be applied to the preprocessed solution.

[2] In the context of linear system solving, these will be Krylov methods, hence the choice of the letter ‘K’.

4.2 Implementation

The set of system preprocessors, like that of the analysis modules above, has a two level structure. First, there is the preprocessor type; for instance 'scaling'. Then there is the specific choice within the type; for instance ''left scaling'. Additionally, but not discussed here, there can be parameters associated with either the type or the specific choice; for instance, we can scale by a block diagonal, with the parameter indicating the size of the diagonal blocks.
We implement the sequence of preprocessors by a recursive routine:

```
PreprocessedSolving
    (char *method,Problem problem,Result *solution)
{
  ApplyPreprocessor(problem,&preprocessed_problem);
  if ( /* more preprocessors */ )
    PreprocessedSolving(next_method,
      preprocessed_problem,&preprocessed_solution);
  else
    Solve(final_method,
      preprocessed_problem,&preprocessed_solution);
  UnApplyPreprocessor(preprocessed_solution,solution);
}
```

The actual implementation is more complicated, but this pseudo-code conveys the essence.
We again adopt a modular approach where preprocessors are dynamically declared:

```
DeclarePreprocessor(char *type,char *choice,
    void(*preprocessor)(Problem,Problem*));
```

The SysPro (System Preprocessor) library provides a number of preprocessors, including the forward and backward transformation of the systems. It also includes a framework for looping over the various choices of a preprocessor type, for instance for an exhaustive test.

5. Method selection

Our method selection problem can be formalized as of constructing a function

$\boxed{\Pi : \mathbb{A} \mapsto \mathbb{M}\text{: the problem classification function}}$

that maps a given problem to the optimal method. Including feature extraction, we can also define

$\boxed{\Pi : \mathbb{F} \mapsto \mathbb{M}\text{: the classification function in terms of features}}$

We start with a brief discussion of precisely what is meant by 'optimal'. After that, we will refine the definition of Π to reflect the preprocessor/solver structure, and we will address the actual construction of Π.

5.1 Different classification criteria

The simplest (non-constructive) definition of the method selection function Π is:

$$\Pi(A) = M \qquad \equiv \qquad \forall_{M'\in\mathbb{M}} \colon T(A,M) \leq T(A,M') \tag{3}$$

Several variant definitions are possible. Often, we are already satisfied if we can construct a function that picks a working method. For that criterium, we define Π non-uniquely as

$$\Pi'(A) = M \qquad \text{where } M \text{ is any method such that} \qquad T(A,M) < \infty \tag{4}$$

Also, we usually do not insist on the absolutely fastest method: we can relax equation (3) to

$$\Pi(A) = M \qquad \equiv \qquad \forall_{M' \in \mathbb{M}} \colon T(A,M) \leq (1-\epsilon)T(A,M') \tag{5}$$

which, for sufficient values of ϵ, also makes the definition non-unique. In both of the previous cases we do not bother to define Π as a multi-valued function, but implicitly interpret $\Pi(A) = M$ to mean 'M is one possible method satisfying the selection criterion'.
Formally, we define two classification types:

classification for reliability This is the problem equation (4) of finding any method that will solve the problem, that is, that will not break down, stall, or diverge.

classification for performance This is the problem equation (3) of finding the fastest method for a problem, possibly within a certain margin.

In a logical sense, the performance classification problem also solves the reliability problem. In practice, however, classifiers are not infallible, so there is a danger that the performance classifier will mispredict, not just by recommending a method that is slower than optimal, but also by possibly recommending a diverging method. Therefore, in practice a combination of these classifiers may be preferable.
We will now continue with discussing the practical construction of (the various guises of) the selection function Π.

5.2 Examples

Let us consider some adaptive systems, and the shape that $\mathbb{F}$, $\mathbb{M}$, and Π take in them.

5.2.1 Atlas

Atlas Whaley et al. (2001) is a system that determines the optimal implementation of Blas kernels such as matrix-matrix multiplication. One could say that the implementation chosen by Atlas is independent of the inputs[3] and only depends on the platform, which we will consider a constant in this discussion. Essentially, this means that $\mathbb{F}$ is an empty space. The number of dimensions of $\mathbb{M}$ is fairly low, consisting of algorithm parameters such unrolling, blocking, and software pipelining parameters.
In this case, Π is a constant function defined by

$$\Pi(f) \equiv \min_{M \in \mathbb{M}} T(A,M)$$

where A is a representative problem. This minimum value can be found by a sequence of line searches, as done in Atlas, or using other minimization techniques such a modified simplex method Yi et al. (2004).

5.2.2 Scalapack/LFC

The distributed dense linear algebra library Scalapack Choi et al. (1992) gives in its manual a formula for execution time as a function of problem size N, the number of processors N_p, and the block size N_b. This is an example of a two-dimensional feature space (N, N_p), and a one-dimensional method space: the choice of N_b. All dimensions range through positive integer values. The function involves architectural parameters (speed, latency, bandwidth) that can be fitted to observations.
Unfortunately, this story is too simple. The LFC software Roche & Dongarra (2002) takes into account the fact that certain values of N_p are disadvantageous, since they can only give grids with bad aspect ratios. A prime number value of N_p is a clear example, as this gives a degenerate grid. In such a case it is often better to ignore one of the available processors and use the remaining ones in a better shaped

[3] There are some very minor caveats for special cases, such as small or 'skinny' matrices.

grid. This means that our method space becomes two-dimensional with the addition of the actually used number of processors. This has a complicated dependence on the number of available processors, and this dependence can very well only be determined by exhaustive trials of all possibilities.

5.2.3 Collective communication

Collective operations in MPI Otto et al. (1995) can be optimized by various means. In work by Vadhyar *et al.* Vadhiyar et al. (2000), the problem is characterized by the message size and the number of processors, which makes $\mathbb{F}$ have dimension 2. The degrees of freedom in $\mathbb{M}$ are the segment size in which messages will be subdivided, and the 'virtual topology' algorithm to be used.
Assume for now that the method components can be set independently. The segment size is then computed as $s = \Pi_s(m,p)$. If we have an *a priori* form for this function, for instance $\Pi_s(m,p) = \alpha + \beta m + \gamma p$, we can determine the parameters by a least squares fit to some observations.
Suppose the virtual topology depends only on the message size m. Since for the virtual topology there is only a finite number of choices, we only need to find the crossover points, which can be done by bisection. If the topology depends on both m and p, we need to find the areas in (m,p) space, which can again be done by some form of bisection.

5.3 Database

In our application we need to be explicit about the database on which the classification is based. That issue is explored in this section.

5.3.1 General construction

The function Π is constructed from a database of performance results that results from solving a set of problems $\mathbf{A} \subset \mathbb{A}$ by each of a collection of methods $\mathbf{M} \subset \mathbb{M}$, each combination yielding a result $r \in \mathbb{R}$ (equation (1)). Thus we store features of the problem, an identifier of the method used, and the resulting performance measurement:

$$\boxed{\mathcal{D} \subset \mathbb{D} = \{\mathbb{F} \times \mathbb{M} \rightarrow \mathbb{T}\}\text{: the database of features and performance results of solved problems}}$$

We posit a mechanism (which differs per classifcation strategy) that constructs Π from a database $\mathcal{D}$. Where needed we will express the dependency of Π on the database explicitly as $\Pi_{\mathcal{D}}$. In the case of Bayesian classification the construction of Π from $\mathcal{D}$ takes a particularly elegant form, which we will discuss next.

5.3.2 Bayesian classification; method suitability

In methods like Bayesian classification, we take an approach to constructing Π where we characterize each method individually, and let Π be the function that picks the most suitable one.
Starting with the database $\mathcal{D}$ as defined above, We note for each problem – and thus for each feature vector – which method was the most successful:

$$\mathcal{D}'\colon \mathbb{F} \times \mathbb{M} \rightarrow \{0,1\} \quad \text{defined by} \quad \mathcal{D}'(f,m) = 1 \equiv m = \operatorname*{argmin}_m \mathcal{D}(f,m)$$

This allows us to draw up indicator functions for each method[4]:

$$\mathbb{B}'\colon \mathbb{M} \rightarrow \text{pow}(\mathbb{F}) \quad \text{defined by} \quad f \in \mathbb{B}'(m) \Leftrightarrow \mathbb{D}'(f,m) = 1 \tag{6}$$

[4] There is actually no objection to having $\mathcal{D}(f,m)$ return 1 for more than one method m; this allows us to equivocate methods that are within a few percent of each other's performance. Formally we do this by extending and redefining the $\arg\min$ function.

These functions are generalized (multi-dimensional) histograms: for each method they plot the feature (vector) values of sample problems for which this method is was found to be optimal. However, since these functions are constructed from a set of experiments we have that, most likely, $\cup_{m\in\mathbb{M}}\mathbb{B}'(m) \subsetneq \mathbb{F}$. Therefore, it is not possible to define

$$\Pi(f) = m \quad \equiv \quad f \in \mathbb{B}'(m),$$

since for many values of f, the feature vector may not be in *any* $\mathbb{B}'(m)$. Conversely, it could also be in $\mathbb{B}'(m)$ for several values of m, so the definition is not well posed.
Instead, we use a more general mechanism. First we define suitability functions:[5]

$$\boxed{\mathbb{S} = \{\mathbb{F} \to [0,1]\}\text{: the space of suitability measurements of feature vectors}} \tag{7}$$

This is to be interpreted as follows. For each numerical method there will be one function $\sigma \in \mathbb{S}$, and $\sigma(f) = 0$ means that the method is entirely unsuitable for problems with feature vector f, while $\sigma(f) = 1$ means that the method is eminently suitable.
We formally associate suitability functions with numerical methods:

$$\boxed{\mathbb{B}\colon \mathbb{M} \to \mathbb{S}\text{: the method suitability function}} \tag{8}$$

and use the function $\mathbb{B}$ to define the selection function:

$$\Pi(f) = \arg\max_m \mathbb{B}(m)(f). \tag{9}$$

Since elements of $\mathbb{S}$ are defined on the whole space $\mathbb{F}$, this is a well-posed definition.
The remaining question is how to construct the suitability functions. For this we need constructor functions

$$\boxed{\mathbb{C}\colon P(\mathbb{F}) \to \mathbb{S}\text{: classifier constructor functions}} \tag{10}$$

The mechanism of these can be any of a number of standard statistical techniques, such as fitting a Gaussian distribution through the points in the subset $\cup_{f\in F} f$ where $F \in P(\mathbb{F})$.
Clearly, now $\mathbb{B} = \mathbb{C} \circ \mathbb{B}'$, and with the function $\mathbb{B}$ defined we can construct the selection function Π as in equation (9).

5.4 Implementation of Bayesian classification

Strictly speaking, the above is a sufficient description of the construction and use of the Π functions. However, as defined above, they use the entire feature vector of a problem, and in practice a limited set of features may suffice for any given decision. This is clear in such cases as when we want to impose "This method only works for symmetric problems", where clearly only a single feature is needed. Computing a full feature vector in this case would be wasteful. Therefore, we introduce the notation $\mathbb{F}_I$ where $I \subset \{1,\ldots,k\}$. This corresponds to the subspace of those vectors in $\mathbb{F}$ that are null in the dimensions not in I.
We will also assume that for each method $m \in \mathbb{M}$ there is a feature set I_m that suffices to evaluate the suitability function $\mathbb{B}(m)$. Often, such a feature set will be common for all methods in a set $\mathbb{P}_i$ of preprocessors. For instance, graph algorithms for fill-in reduction only need structural information on the matrix.
Here is an example of the API (as used in the Salsa system) that defines and uses feature sets. First we define a subset of features:

[5] Please ignore the fact that the symbol $\mathbb{S}$ already had a meaning, higher up in this story.

```
FeatureSet symmetry;
NewFeatureSet(&symmetry);
AddToFeatureSet(symmetry,
    "simple","norm-of-symm-part",&sidx);
AddToFeatureSet(symmetry,
    "simple","norm-of-asymm-part",&aidx);
```

After problem features have been computed, a suitability function for a specific method can then obtain the feature values and use them:

```
FeatureSet symmetry; // created above
FeatureValues values;
NewFeatureValues(&values);
InstantiateFeatureSet(problem,symmetry,values);
GetFeatureValue(values,sidx,&sn,&f1);
GetFeatureValue(values,aidx,&an,&f2);
if (f1 && f2 && an.r>1.e-12*sn.r)
   printf("problem too unsymmetric\n");
```

6. Classification of composite methods

In section 4 we showed how, in our application area of linear system solving, the space of methods has a structure where a method is a composite of a sequence of preprocessing steps and a concluding solver; equation equation (2). Accordingly, our recommendation function Π will be a composite:

$$\Pi = \langle \Pi_k, \{\Pi_i\}_{i \in \mathbb{P}} \rangle.$$

In the previous section, we posited a general mechanism (which we described in detail for methods like Bayesian classification) of deriving $\Pi_{\mathcal{D}}$ from a database $\mathcal{D} \subset \{\mathbb{F} \times \mathbb{M} \to \mathbb{T}\}$. In this section we will consider ways of defining databases $\mathcal{D}_{\mathbb{K}}, \mathcal{D}_{\mathbb{P}}$ and attendant functions $\Pi_{\mathbb{K}}, \Pi_{\mathbb{P}}$, and of combining these into an overall recommendation function.
For simplicity of exposition, we restrict our composites to a combination of one preprocessor (in practice, the preconditioner), and one solver (the iterative method); that is $\mathbb{M} = \mathbb{P} \times \mathbb{K}$.
At first we consider the performance problem, where we recommend a method that will minimize solution time (refer to section 5.1 for a definition of the two types of classification). Then, in section 6.4 we consider the reliability problem of recommending a method that will converge, no matter the solution time.

6.1 Combined recommendation

In this strategy, we ignore the fact that a method is a product of constituents, and we simply enumerate the elements of $\mathbb{M}$. Our function Π is then based on the database

$$\mathcal{D} = \{\langle f, \langle p,k \rangle, t \rangle \mid \exists_{a \in \mathcal{A}} \colon t = T(p,k,a)\}$$

and the recommendation function is a straightforward mapping $\Pi^{\text{combined}}(f) = \langle p,k \rangle$.
For Bayesian classification we get for each $\langle p,k \rangle \in \mathbb{M}$ the class

$$C_{p,k} = \{A \in \mathcal{A} \colon T(p,k,A) \text{ is minimal}\}$$

and corresponding function $\sigma_{p,k}$. We can then define

$$\Pi^{\text{combined}}(f) = \arg\max_{p,k} \sigma_{p,k}(f)$$

The main disadvantage to this approach is that, with a large number of methods to choose from, some of the classes can be rather small, leading to insufficient data for an accurate classification.
In an alternative derivation of this approach, we consider the $C_{p,k}$ to be classes of preprocessors, but conditional upon the choice of a solver. We then recommend p and k, not as a pair but sequential: we first find the k for which the best p can be found:

$$\Pi^{\text{conditional}} = \begin{cases} \text{let } k := \arg\max_k \max_p \sigma_{p,k}(f) \\ \text{return } \langle \arg\max_p \sigma_{p,k}(f), k \rangle \end{cases}$$

However, this is equivalent to the above combined approach.

6.2 Orthogonal recommendation

In this strategy we construct separate functions for recommending elements of $\mathbb{P}$ and $\mathbb{K}$, and we put together their results.
We define two derived databases that associate a solution time with a feature vector and a preprocessor or solver separately, even though strictly speaking both are needed to solve a problem and thereby produce a solution time. For solvers:

$$\mathcal{D}_{\mathbb{K}} = \left\{ \langle f, k, t \rangle \mid k \in \mathbb{K}, \exists_{a \in \mathcal{A}} \colon f = \phi(a), t = \min_{p \in \mathbb{P}} T(p, k, a) \right\}$$

and for preprocessors:

$$\mathcal{D}_{\mathbb{P}} = \left\{ \langle f, p, t \rangle \mid p \in \mathbb{P}, \exists_{a \in \mathcal{A}} \colon f = \phi(a), t = \min_{k \in \mathbb{K}} T(p, k, a) \right\}.$$

From these, we derive the functions $\Pi_{\mathbb{K}}, \Pi_{\mathbb{P}}$ and we define

$$\Pi^{\text{orthogonal}}(f) = \langle \Pi_{\mathbb{P}}(f), \Pi_{\mathbb{K}}(f) \rangle$$

In Bayesian classification, the classes here are

$$C_k = \{A \colon \min_p T(p, k, A) \text{ is minimal over all } k\}$$

and

$$C_p = \{A \colon \min_k T(p, k, A) \text{ is minimal over all } p\},$$

giving functions σ_p, σ_k. (Instead of classifying by minimum over the other method component, we could also use the average value.) The recommendation function is then

$$\Pi^{\text{orthogonal}}(f) = \langle \arg\max_p \sigma_p(f), \arg\max_k \sigma_k(f) \rangle$$

6.3 Sequential recommendation

In the sequential strategy, we first recommend an element of $\mathbb{P}$, use that to transform the system, and recommend an element of $\mathbb{K}$ based on the transformed features.
Formally, we derive $\Pi_{\mathbb{P}}$ as above from the derived database

$$\mathcal{D}_{\mathbb{P}} = \left\{ \langle f, p, t \rangle \mid p \in \mathbb{P}, \exists_{a \in \mathcal{A}} \colon f = \phi(a), t = \min_{k \in \mathbb{K}} T(p,k,a) \right\}.$$

but $\Pi_{\mathbb{K}}$ comes from the database of all preprocessed problems:

$$\begin{array}{l} \mathcal{D}_{\mathbb{K}} = \cup_{p \in \mathbb{P}} \mathcal{D}_{\mathbb{K},p}, \\ \mathcal{D}_{\mathbb{K},p} = \{ \langle f, k, t \rangle \mid k \in \mathbb{K}, \exists_{a \in \mathcal{A}} \colon f = \phi(p(a)), t = T(p,k,a) \} \end{array}$$

which gives us a single function $\Pi_{\mathbb{P}}$ and individual functions $\Pi_{\mathbb{K},p}$. This gives us

$$\Pi^{\text{sequential}}(f) = \langle \text{let } p{:=}\Pi_{\mathbb{P}}(f), k{:=}\Pi_{\mathbb{K}}(p(f)) \text{ or } \Pi_{\mathbb{K},p}(p(f)) \rangle$$

For Bayesian classification, we define the classes C_p as above:

$$C_p = \{ A \colon \min_k T(p,k,A) \text{ is minimal over all } p \},$$

but we have to express that C_k contains preconditioned features:

$$C_k = \{ A \in \bigcup_p p(\mathcal{A}) \colon T(p,k,A) \text{ is minimal, where } p \text{ is such that } A \in p(\mathcal{A}) \}$$

Now we can define

$$\Pi^{\text{sequential}}(f) = \langle \text{let } p{:=}\arg\max_p \sigma_p(f), k{:=}\arg\max_k \sigma_k(p(f)) \rangle$$

This approach to classification is potentially the most accurate, since both the preconditioner and iterator recommendation are made based on the features of the actual problem they apply to. This also means that this approach is the most expensive; both the combined and the orthogonal approach require only the features of the original problem. In practice, with a larger number of preprocessors, one can combine these approaches. For instance, if a preprocessor such as scaling can be classified based on some easy to compute features, it can be tackled sequentially, while the preconditioner and iterator are then recommended with the combined approach based on a full feature computation of the scaled problem.

6.4 The reliability problem

In the reliability problem we classify problems by whether a method converges on them or not. The above approaches can not be used directly in this case, for several reasons.

- The above approaches are based on assigning each problem to a single classes based on minimum solution time. In the reliability problem each problem would be assigned to multiple classes, since typically more than one method would converge on the problem. The resulting overlapping classes would lead to a low quality of recommendation.
- The sequential and orthogonal approaches would run into the additional problem that, given a preconditioner, there is usually at least one iterative method that gives a converging combination. Separate recommendation of the preconditioner is therefore impossible.

Instead, we take a slightly different approach. For each method m we define a function $\Pi^{(m)} : \mathbb{F} \mapsto \{0,1\}$ which states whether the method will converge given a feature vector of a problem. We can then define

$$\Pi(f) = \text{a random element of } \{m \colon \Pi^{(m)}(f) = 1\}$$

For Bayesian classification, we can adopt the following strategy. For each $M \in \mathbb{M}$, define the set of problems on which it converges:

$$C_M = \{A \colon T(M,A) < \infty\}$$

and let $\bar{C}_M$ be its complement:

$$\bar{C}_M = \{A \colon T(M,A) = \infty\}.$$

Now we construct functions $\sigma_M, \bar{\sigma}_M$ based on both these sets. This gives a recommendation function:

$$\Pi(f) = \{M \colon \sigma_M(f) > \bar{\sigma}_M(f)\}$$

This function is multi-valued, so we can either pick an arbitrary element from $\Pi(f)$, or the element for which the excess $\sigma_M(f) - \bar{\sigma}_M(f)$ is maximized.
The above strategies give only a fairly weak recommendation from the point of optimizing solve time. Rather than using reliability classification on its own, we can use it as a preliminary step before the performance classification.

7. Experiments

In this section we will report on the use of the techniques developed above, applied to the problem of recommending a preconditioner and iterative method for solving a linear system. The discussion on the experimental setup and results will be brief; results with much greater detail can be found in Fuentes (2007).
We start by introducing some further concepts that facilitate the numerical tests.

7.1 Experimental setup

We use a combination of released software from the Salsa project Salsa Project (n.d.a;n) and custom scripts. For feature computation we use AnaMod Eijkhout & Fuentes (2007); The Salsa Project (n.d.); storage and analysis of features and timings is done with MySQL and custom scripting in Matlab and its statistical toolbox.
The AnaMod package can compute 45 features, in various categories, such as structural features, norm-like features, measures of the spectrum and of the departure from normality. The latter two are obviously approximated rather than computed exactly.
The Salsa testbed gives us access to the iterative methods and preconditioners of the Petsc package. including the preconditioners of externally interfaced packages such as Hypre Falgout et al. (2006); Lawrence Livermore Lab, CASC group (n.d.).

7.2 Practical issues

The ideas developed in the previous sections are sufficient in principle for setting up a practical application of machine learning to numerical method selection. However, in practice we need some auxiliary mechanisms to deal with various ramifications of the fact that our set of test problems is not of infinite size. Thus, we need

- A way of dealing with features that can be invariant or (close to) dependent in the test problem collection.

- A way of dealing with methods that can be very close in their behaviour.
- An evaluation of the accuracy of the classifiers we develop.

7.2.1 Feature analysis

There are various transformations we apply to problem features before using them in various learning methods.

Scaling Certain transformations on a test problem can affect the problem features, without affecting the behaviour of methods, or being of relevance for the method choice. For instance, scaling a linear system by a scalar factor does not influence the convergence behaviour of iterative solvers. Also, features can differ in magnitude by order of magnitude. For this reason, we normalize features, for instance scaling them by the largest diagonal element. We also mean-center features for classification methods that require this.

Elimination Depending on the collection of test problems, a feature may be invariant, or dependent on other features. We apply Principal Component Analysis Jackson (2003) to the set of features, and use that to weed out irrelevant features.

7.2.2 Hierarchical classification

It is quite conceivable that certain algorithms are very close in behaviour. It then makes sense to group these methods together and first construct a classifier that can recommend first such a group, and subsequently a member of the group. This has the advantage that the classifiers are build from a larger number of observations, giving a higher reliability.
The algorithm classes are built by computing the independence of methods. For two algorithms x and y, the 'independence of method x from method y' is defined as

$$I_y(x) = \frac{\#\text{cases where } x \text{ works and } y \text{ not}}{\#\text{cases where } x \text{ works}}$$

The quantity $I_y(x) \in [0,1]$ describes how much x succeeds on different problems from y. Note that $I_y(x) \neq I_x(y)$ in general; if x works for every problem where y works (but not the other way around), then $I_y(x) = 0$, and $I_x(y) > 0$.

7.2.3 Evaluation

In order to evaluate a classifier, we use the concept of accuracy. The accuracy α of a classifier is defined as

$$\alpha = \frac{\#\text{problems correctly classified}}{\text{total } \#\text{problems}}$$

A further level of information can be obtained looking at the details of misclassification: a 'confusion matrix' is defined as $A = (\alpha_{ij})$ where α_{ij} is the ratio of problems belonging in class i, classified in class j, to those belonging in class i. With this, α_{ii} is the accuracy of classifier i, so, for an ideal classifier, A is a diagonal matrix with a diagonal $\equiv 1$; imperfect classifiers have more weight off the diagonal.
A further measure of experimental results is the confidence interval z, which indicates an interval in which the resulting accuracy for a random trial will be away for the presented average accuracy α by $\pm z$ Douglas & Montgomery (1999) of the time. We use z to delimit the confidence interval since we have used the *Z-test* Douglas & Montgomery (1999), commonly used in statistics. The confidence interval is a measure of how 'stable' the resulting accuracy is for an experiment.

7.3 Numerical test

We tested a number of iterative methods and preconditioners on a body of test matrices, collected from Matrix Market and a few test applications. The iterative methods and preconditioners are from the PETSc library.

As described above, we introduced superclasses, as follows:

- **B**={ *bcgs, bcgsl, bicg* }, where *bcgs* is BiCGstab van der Vorst (1992), and *bcgsl* is BiCGstab(ℓ) Sleijpen et al. (n.d.) with $\ell \geq 2$.
- **G**={ *gmres, fgmres* } where *fgmres* is the 'flexible' variant of GMRES Saad (1993).
- **T**={ *tfqmr* }
- **C**={ *cgne* }, conjugate gradients on the noral equations.

for iterative methods and

- **A** = { *asm, rasm, bjacobi* }, where *asm* is the Additive Schwarz method, and *rasm* is its restricted variant Cai & Sarkis (1999); *bjacobi* is block-jacobi with a local ILU solve.
- **BP** = { *boomeramg, parasails, pilut* }; these are three preconditioners from the *hypre* package Falgout et al. (2006); Lawrence Livermore Lab, CASC group (n.d.)
- **I** = { *ilu, silu* }, where *silu* is an ILU preconditioner with shift Manteuffel (1980).

for preconditioners.

(a) Iterative Methods

ksp	$\alpha \pm z$
bcgsl	0.59±0.02
bcgs	0.71±0.03
bicg	0.68±0.06
fgmres	0.80±0.02
gmres	0.59±0.04
lgmres	0.81±0.03
tfqmr	0.61±0.05

(b) Preconditioners

pc	$\alpha \pm z$
asm	0.72±0.05
bjacobi	0.11±0.11
boomeramg	0.71±0.06
ilu	0.66±0.02
parasails	0.46±0.12
pilut	0.80±0.06
rasm	0.70±0.04
silu	0.83±0.02

Table 1. Accuracy of classification using one class per available method

In table 1 we report the accuracy (as defined above) for a classification of all individual methods, while table 2 gives the result using superclasses. Clearly, classification using superclasses is superior. All classifiers were based on decision trees Breiman et al. (1983); Dunham (2002).

Finally, in figures 1, 2 we give confusion matrices for two different classification strategies for the preconditioner / iterative method combination. The orthogonal approach gives superior results, as evinced by the lesser weight off the diagonal. For this approach, there are fewer classes to build classifiers for, so the modeling is more accurate. As a quantative measure of the confusion matrices, we report in table 3 the average and standard deviation of the fraction of correctly classified matrices.

(a) Iterative methods

Super	Class	α	Compound
B		0.95	
	bcgs	0.93	0.87
	bcgsl	0.92	0.87
	bicg	0.89	0.84
G		0.98	
	fgmres	0.96	0.94
	gmres	0.91	0.89
	lgmres	0.94	0.93
T		0.91	
	tfqmr	–	0.91

(b) Preconditioners

Super	Class	α	Compound
A		0.95	
	asm	0.98	0.93
	bjacobi	0.67	0.64
	rasm	0.82	0.78
BP		0.99	
	boomeramg	0.80	0.80
	parasails	0.78	0.77
	pilut	0.97	0.96
I		0.94	
	ilu	0.82	0.75
	silu	0.97	0.91

Table 2. Hierarchical classification results

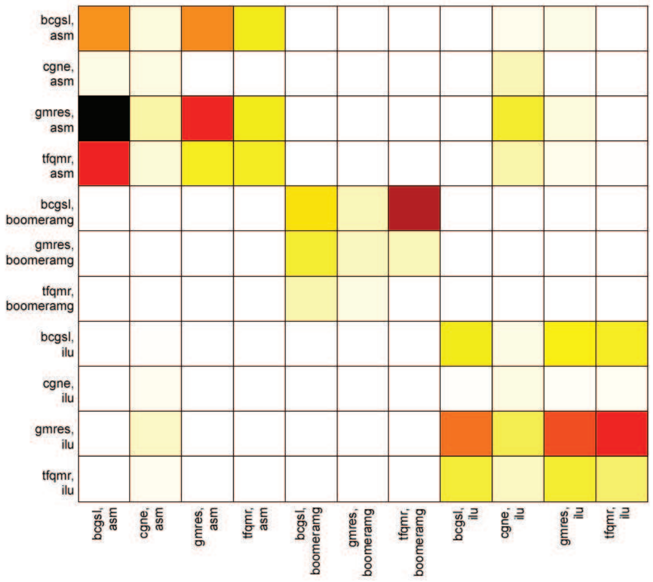

Fig. 1. Confusion matrix for combined approach for classifying (pc, ksp).

7.4 Reliability and Performance

Above (section 5.1) we defined the Performance recommendation problem of finding the fastest method, and the Reliability recommendation problem of finding one that works at all. Since orthogonal recommendation is not possible for the reliability problem (section 6.4), we use combined recommendation there. In our tests, this strategy turns out to recommend a subset of the actually converging methods, so it is indeed a valuable preprocessing step.

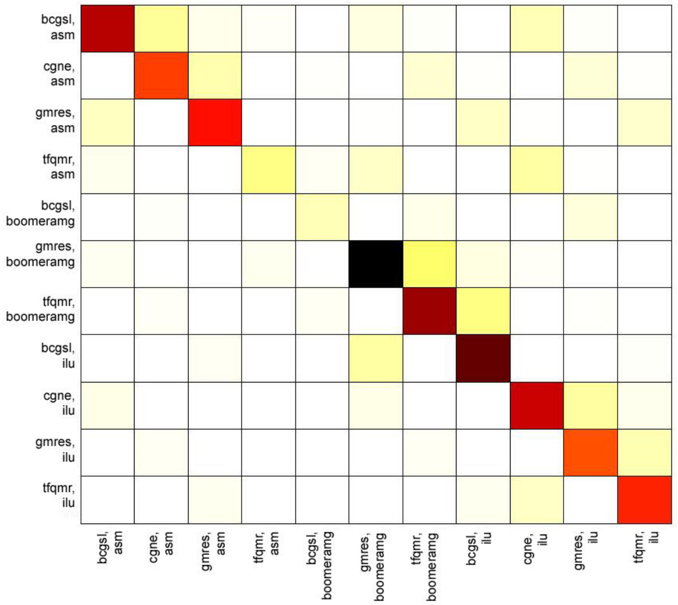

Fig. 2. Confusion matrix for orthogonal approach for classifying (pc, ksp).

strategy	average	std.dev.
combined	.3	.15
orthogonal	.78	.04

Table 3. Average and standard deviation of the correct classification rate

8. Conclusion

We have defined the relevant concepts for the use of machine learning for algorithm selection in various areas of numerical analysis, in particular iterative linear system solving. An innovative aspect of our approach is the multi-leveled approach to the set of objects (the algorithms) to be classified. An important example of levels is the distinction between the iterative process and the preconditioner in iterative linear system solvers. We have defined various strategies for classifying subsequent levels. A numerical test testifies to the feasibility of using machine learning to begin with, as well as the necessity for our multi-leveled approach.

9. References

Arnold, D., Blackford, S., Dongarra, J., Eijkhout, V. & Xu, T. (2000). Seamless access to adaptive solver algorithms, *in* M. Bubak, J. Moscinski & M. Noga (eds), *SGI Users' Conference*, Academic Computer Center CYFRONET, pp. 23–30.

Balay, S., Gropp, W. D., McInnes, L. C. & Smith, B. F. (1999). PETSc home page. http://www.mcs.anl.gov/petsc.

Bhowmick, S., Eijkhout, V., Freund, Y., Fuentes, E. & Keyes, D. (2006). Application of machine learning to the selection of sparse linear solvers, *Int. J. High Perf. Comput. Appl.* . submitted.

Breiman, L., Friedman, J., Olshen, R. & Stone, C. (1983). *CART: Classification and Regression Trees*.

Cai, X.-C. & Sarkis, M. (1999). A restricted additive Schwarz preconditioner for general sparse linear systems, *SIAM J. Sci. Comput.* **21**: 792–797.

Choi, Y., Dongarra, J. J., Pozo, R. & Walker, D. W. (1992). Scalapack: a scalable linear algebra library for distributed memory concurrent computers, *Proceedings of the fourth symposium on the frontiers of massively parallel computation (Frontiers '92), McLean, Virginia, Oct 19–21, 1992*, pp. 120–127.

Czyzyk, J., Mesnier, M. & More, J. (1996). The NetworkEnabled optimization system (neos) server, *Technical Report MCS-P615-0996*, Argonne National Laboratory, Argonne, IL.

Czyzyk, J., Mesnier, M. & Moré, J. (1998). The NEOS server, *IEEE J. Comp. Sci. Engineering* **5**: 68–75.

Dongarra, J., Bosilca, G., Chen, Z., Eijkhout, V., Fagg, G. E., Fuentes, E., Langou, J., Luszczek, P., Pjesivac-Grbovic, J., Seymour, K., You, H. & Vadiyar, S. S. (2006). Self adapting numerical software (SANS) effort, *IBM J. of R.& D.* **50**: 223–238. `http://www.research.ibm.com/journal/rd50-23.html`, also UT-CS-05-554 University of Tennessee, Computer Science Department.

Douglas, C. & Montgomery, G. (1999). *Applied Statistics and Probability for Engineers*.

Dunham, M. (2002). *Data Mining: Introductory and Advanced Topics*, Prentice Hall PTR Upper Saddle River, NJ, USA.

Eijkhout, V. & Fuentes, E. (2007). A standard and software for numerical metadata, *Technical Report TR-07-01*, Texas Advanced Computing Center, The University of Texas at Austin. to appear in ACM TOMS.

Falgout, R., Jones, J. & Yang, U. (2006). The design and implementation of hypre, a library of parallel high performance preconditioners, *Numerical Solution of Partial Differential Equations on Parallel Computers, A.M. Bruaset and A. Tveito, eds.*, Vol. 51, Springer-Verlag, pp. 267–294. UCRL-JRNL-205459.

Fuentes, E. (2007). *Statistical and Machine Learning Techniques Applied to Algorithm Selection for Solving Sparse Linear Systems*, PhD thesis, University of Tennessee, Knoxville TN, USA.

Greenbaum, A. & Strakos, Z. (1996). Any nonincreasing convergence curve is possible for GMRES, *SIMAT* **17**: 465–469.

Gropp, W. D. & Smith, B. F. (n.d.). Scalable, extensible, and portable numerical libraries, *Proceedings of the Scalable Parallel Libraries Conference, IEEE 1994*, pp. 87–93.

Houstis, E., Verykios, V., Catlin, A., Ramakrishnan, N. & Rice, J. (2000). PYTHIA II: A knowledge/database system for testing and recommending scientific software.

Jackson, J. (2003). *A User's Guide to Principal Components*, Wiley-IEEE.

Lawrence Livermore Lab, CASC group (n.d.). Scalable Linear Solvers. `http://www.llnl.gov/CASC/linear_solvers/`.

Manteuffel, T. (1980). An incomplete factorization technique for positive definite linear systems, *Math. Comp.* **34**: 473–497.

Otto, S., Dongarra, J., Hess-Lederman, S., Snir, M. & Walker, D. (1995). *Message Passing Interface: The Complete Reference*, The MIT Press.

Ramakrishnan, N. & Ribbens, C. J. (2000). Mining and visualizing recommendation spaces for elliptic PDEs with continuous attributes, *ACM Trans. Math. Software* **26**: 254–273.

Roche, K. J. & Dongarra, J. J. (2002). Deploying parallel numerical library routines to cluster computing in a self adapting fashion. Submitted.

Saad, Y. (1993). A flexible inner-outer preconditioned GMRES algorithm, *SIAM J. Sci. Stat. Comput.* **14**: 461–469.

Salsa Project (n.d.a). SALSA: Self-Adapting Large-scale Solver Architecture. `http://icl.cs.utk.edu/salsa/`.

Salsa Project (n.d.b). SourceForge: Self-Adapting Large-scale Solver Architecture. `http://sourceforge.net/projects/salsa/`.

Sleijpen, G., der Vorst, H. V. & Fokkema, D. (n.d.). Bicgstab(ℓ) and other hybrid Bi-cg methods. submitted.

The Salsa Project (n.d.). AnaMod software page. `http://icl.cs.utk.edu/salsa/software`.

Vadhiyar, S. S., Fagg, G. E. & Dongarra, J. J. (2000). Automatically tuned collective communications, *Proceedings of the 2000 ACM/IEEE conference on Supercomputing (CDROM)*, IEEE Computer Society, p. 3.

van der Vorst, H. (1992). Bi-CGSTAB: a fast and smoothly converging variant of Bi-CG for the solution of nonsymmetric linear systems, *SIAM J. Sci. Stat. Comput.* **13**: 631–644.

Whaley, R. C., Petitet, A. & Dongarra, J. J. (2001). Automated empirical optimization of software and the ATLAS project, *Parallel Computing* **27**(1–2): 3–35. Also available as University of Tennessee LAPACK Working Note #147, UT-CS-00-448, 2000 (`www.netlib.org/lapack/lawns/lawn147.ps`).

Wilson, D. C., Leake, D. B. & Bramley, R. (2000). Case-based recommender components for scientific problem-solving environments, *Proceedings of the Sixteenth International Association for Mathematics and Computers in Simulation World Congress*.

Yi, Q., Kennedy, K., You, H., Seymour, K. & Dongarra, J. (2004). Automatic Blocking of QR and LU Factorizations for Locality, *2nd ACM SIGPLAN Workshop on Memory System Performance (MSP 2004)*.