# A Self-Adapting System for Linear Solver Selection

Victor Eijkhout (University of Texas)
joint work with
Erika Fuentes (University of Tennessee)
Naren Ramakrishnan and Pilsung Kang (Virginia Tech)
Sanjukta Bhowmick and David Keyes
(Columbia University, New York)
Yoav Freund (UCSD)

Support: NSF

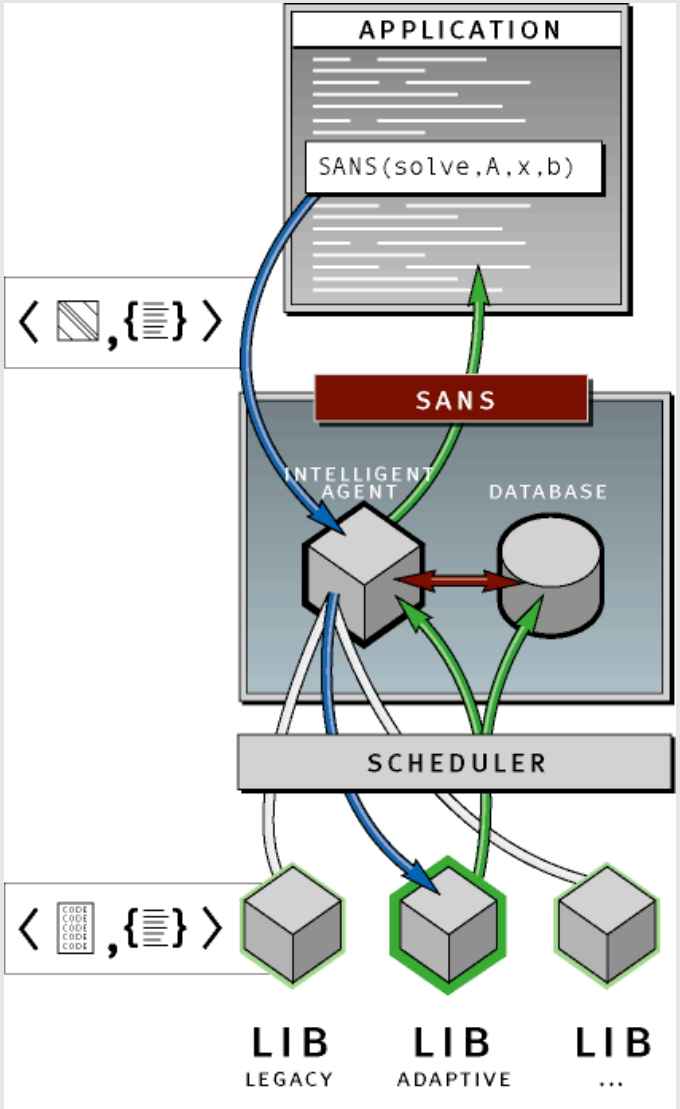
# Self-Adapting Numerical Software; problem statement

On several levels of a scientific problem-solving environment, decision are beyond the user's province.

- Kernel performance is not portable or persistent
- Scheduling of parallel jobs is extremely dynamic
- Solving scientific problems takes numerical knowledge outside the application field

# Proposed solution

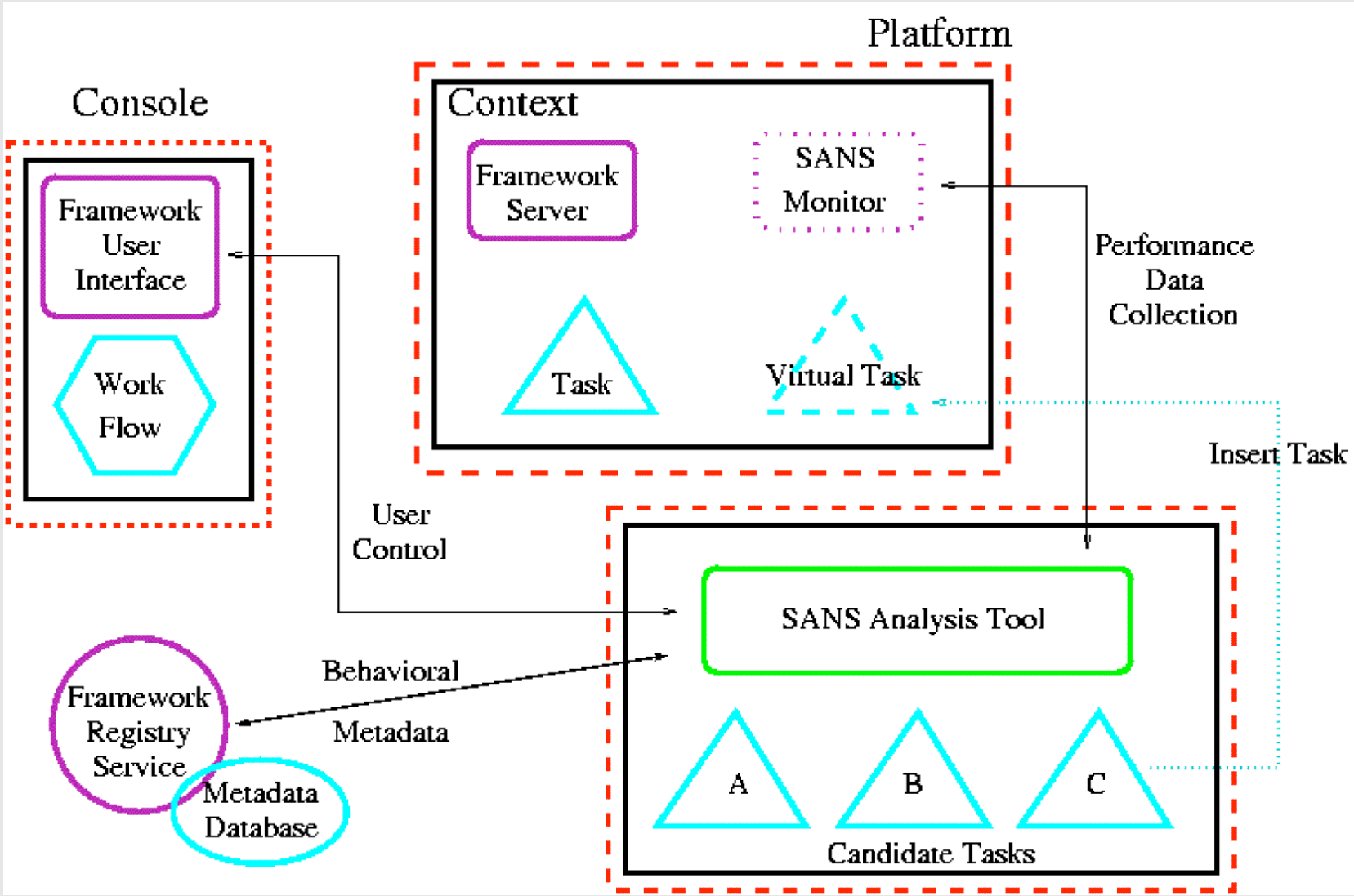
- Adaptive software: inspect user data and computational platform, find best algorithm
- Heuristic investigation: most questions can not be answered exactly; we settle for good enough
- Self-adapting software: remember results of production runs, learn over time




Three levels of adaptivity

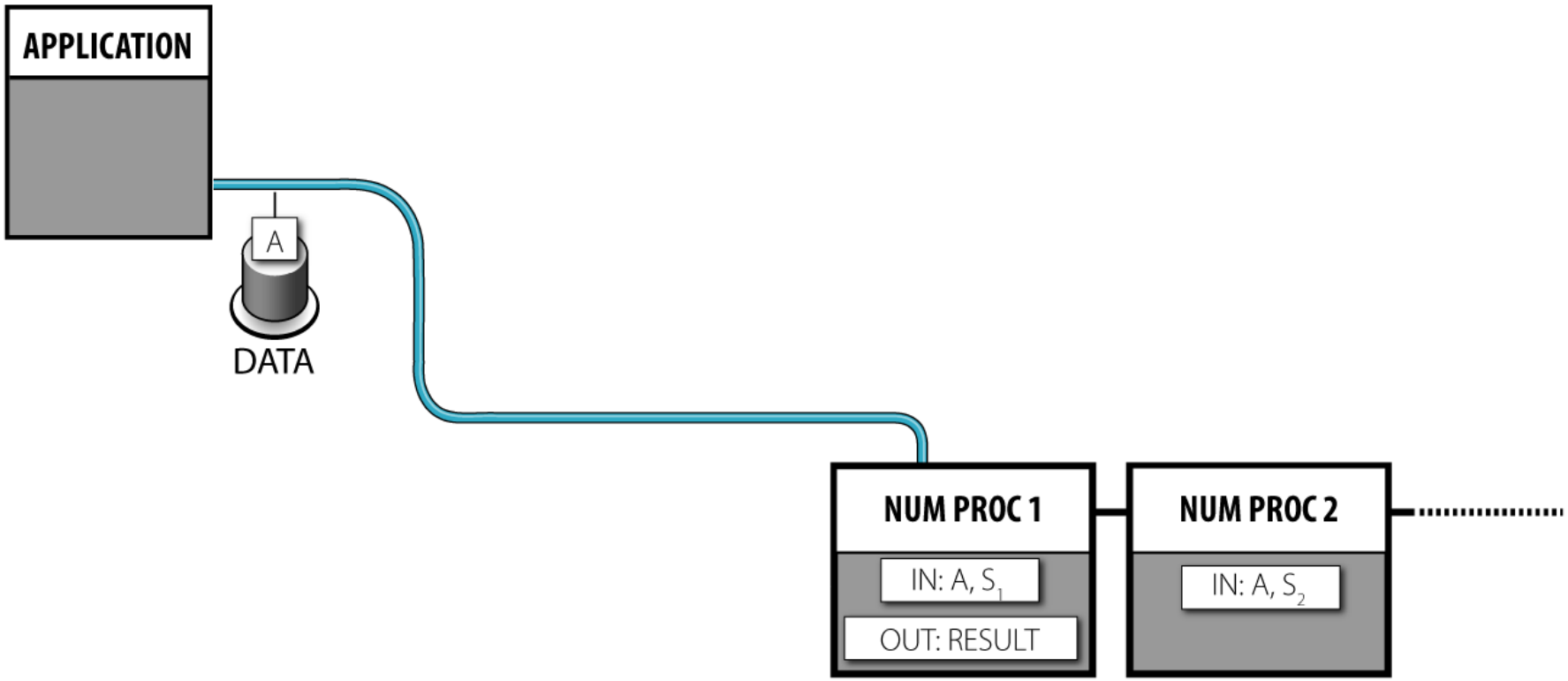
- Kernel level: Atlas and such; independent of user data, can be done in one-time installation
- Network level: Grads and such; modest interaction with user data, but no major influence
- Algorithm level: analysis dynamically, completely based on user data

# Component-based framework

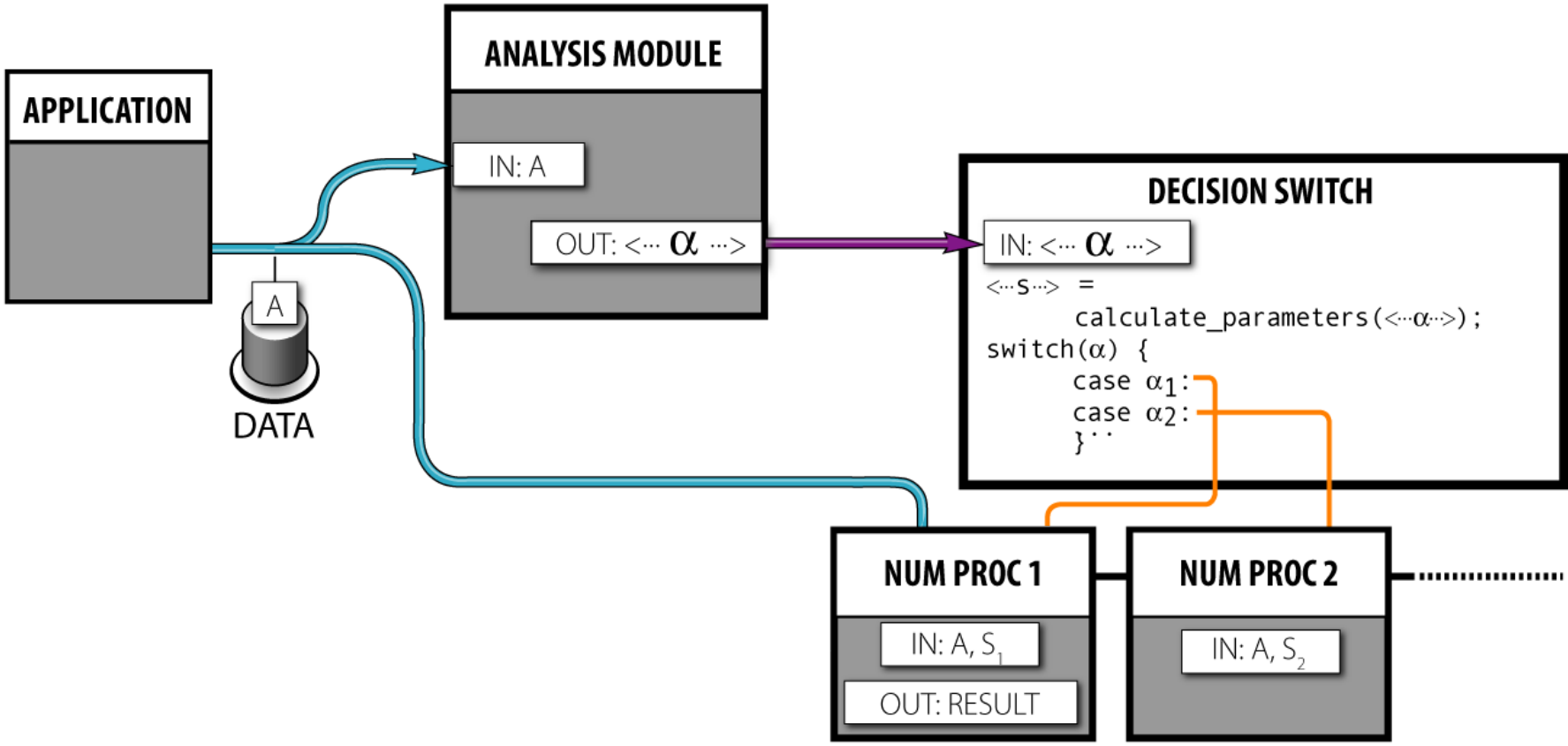


# Adaptive algorithm choice

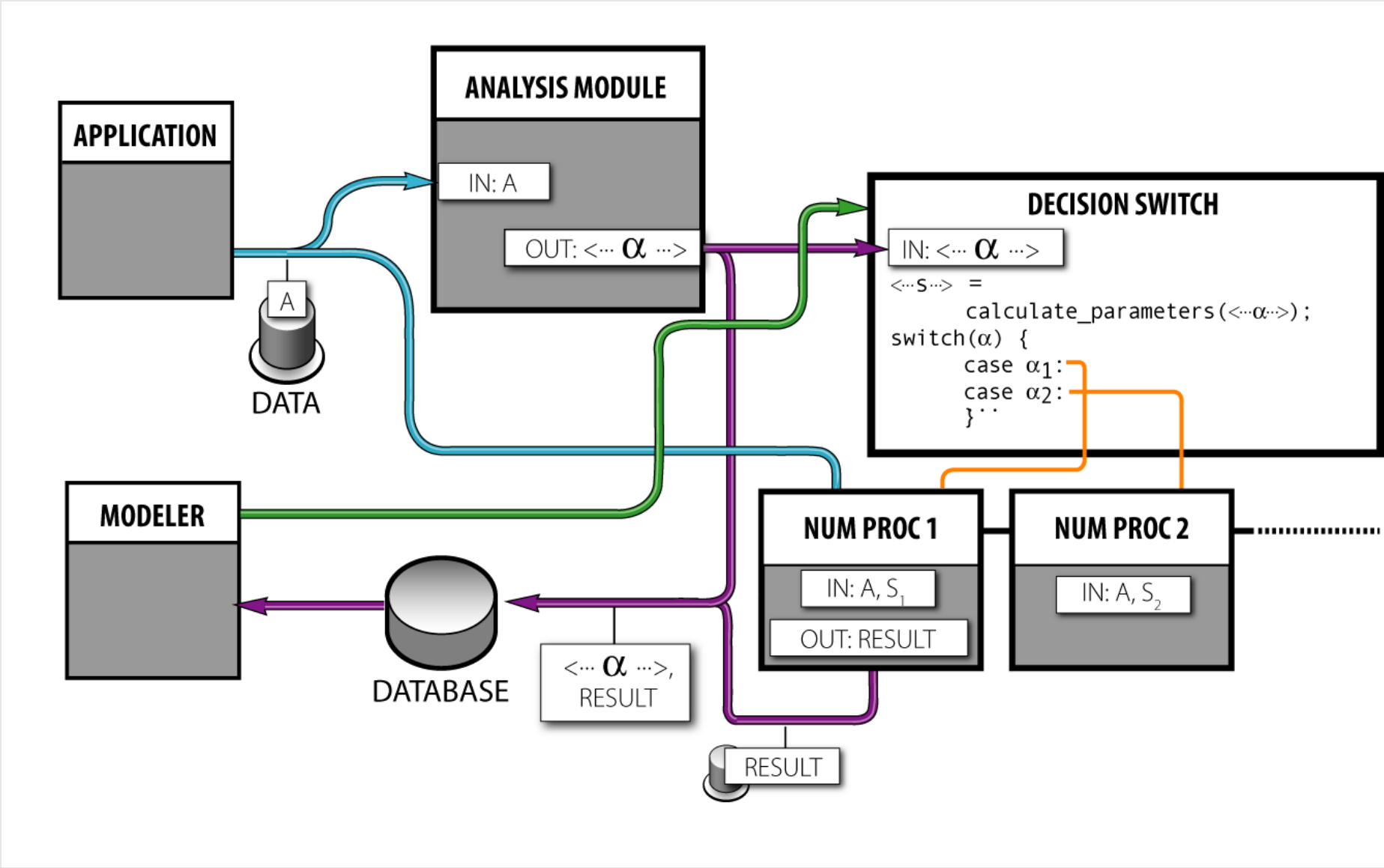
# Hand-coded decisions



# Use of analysis modules



# Self-adaptive system



# Abstract description of adaptive systems

# Numerical problems

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

Dependent on the application area
linear systems:

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

# Results

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

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

# Solution methods

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

```
Solver(Problem problem,Result *solution);
```

In the SALSA system, the solvers are the `KSP` objects of the Petsc library.

# Method selection

$T(A, M)$: time taken by method $M$ on problem $A$

Function $\Pi : \mathbb{A} \mapsto \mathbb{M}$

$$\Pi(A) = M \quad \equiv \quad \forall_{M' \in \mathbb{M}} : T(A, M) \leq T(A, M')$$

Often enough:

$$\Pi'(A) = M \quad \equiv \quad M \text{ is such that } T(A, M) < \infty$$

Defining $\Pi$ in terms of $\mathbb{A}$ impractical: need problem features

# Features, metadata

$\mathbb{F}$: *the space of features of the numerical problems*

Features (Numerical Metadata) is divided in categories and elements.

Example categories: structural, normlike, spectral

Often a single computational function for one category.

# Feature classes

| | |
|---|---|
| structure | Structural symmetry, bandwidth, sparsity, skyline |
| iprs | more structural properties |
| simple | various norms, including norm of symmetric/antisymmetric part |
| variance | diagonal average and variance |
| normality | various measure of departure from normality |
| spectrum | ellipse around FOV, condition, max/min singular values |
| $\Rightarrow$ | 75 features in all |

# The NMD library

# Metadata API

In SALSA: NMD library

```
NMDCreateObject(NMD_metadata *obj);
NMDCreateCategory(NMD_metadata,char *cat);
NMDCreateComponent
      (NMD_metadata,char *cat,char *cmp,NMDDataType t);
NMDSetValue(NMD_metadata,char *cat,char *cmp,void *val);
NMDGetValue(NMD_metadata,char *cat,char *cmp,
       NMDDataType *t,void *val,int *success);
```

# NMD capabilities

- Storage and retrieval
- Manipulation of whole categories
- Conversion to XML, display with XSL

# The AnaMod library

# Feature extraction

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

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

Modular design:

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

# Method selection, 2nd try

$\Pi : \mathbb{F} \mapsto \mathbb{M}$: *the function that picks the best solution method*

$\Pi(\bar{x}) = M$ if $\Phi(A) = \bar{x}$ and $M$ s.t. $\forall_{M' \in \mathbb{M}} : T(A, M) \leq T(A, M')$.

(what if $\Phi(A_1) = \Phi(A_2)$?)
Method picking:

$$\Pi \circ \Phi : \mathbb{A} \mapsto \mathbb{M}$$

*Intelligent Solver*:

$$Q = \lambda(A)\Pi(\Phi(A))(A)$$

Complication: methods are actually compound objects

# The SysPro library

# Preprocessors

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

Example: scale or permute a linear system

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

Example: iterative and direct methods

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

# Preprocessor classes

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

Each preprocessor belongs to a class:

$$\mathbb{P} = \mathbb{P}^1 \cup \cdots \mathbb{P}^n, \qquad p_i \in \mathbb{P}^i$$

For instance: left / right / symmetric scaling.

# Preprocessor implementation

$$\mathbb{P} = \{\mathbb{A} \mapsto \mathbb{A}\}$$

```
SystemPreprocessor(char *type,char *choice,
    Problem in,Problem *out);
```

Dynamic discovery:

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

# Method selection, 3rd try

$\Pi_i : \mathbb{F} \mapsto \mathbb{P}_i$: *the selector function for preprocessor i*

$\Pi_k : \mathbb{F} \mapsto \mathbb{K}$: *the method selector*

Intelligent preprocessor application:

$$Q_i = \lambda(A)\Pi_i(\Phi(A))(A)$$

$\Rightarrow Q_i \in \mathbb{P}$.

# The ModBaR library

# Method suitability

Suitability functions:

| |
|---|
| $\mathbb{S} = \{\mathbb{F} \to [0,1]\}$: the space of suitability measurements of feature vectors |

"Conjugate gradient method: only for symmetric positive definite" (semantic specification of library routines)

| |
|---|
| $\mathbb{B}: \mathbb{M} \to \mathbb{S}$: the method suitability function |

then

$$\Pi(f) = \arg\max_{m} \mathbb{B}(m)(f).$$

# Classifier construction

$\mathbb{D}\colon \mathbb{F} \times \mathbb{M} \to \mathbb{T}$: *the database of features and performance results of solved problems*

Mark the best method among those tested:

$$\mathbb{D}'\colon \mathbb{F}\times\mathbb{M} \to \{0,1\} \quad \Leftrightarrow \quad \mathbb{D}'(f,m) = 1 \;\equiv\; m = \underset{m\colon (m,f)\in\mathbb{D}}{\arg\min}\; \mathbb{D}(f,m)$$

Describe problems solved best by each method:

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

Classifier construction

$$\mathbb{C}\colon P(\mathbb{F}) \to \mathbb{S}$$

Then $\mathbb{B} = \mathbb{C} \circ \mathbb{B}'$.

# Implementation of classifier construction

Requires functional language; implementation in C possible

# Declare suitability function

```
FeatureSet symmetry;
NewFeatureSet(&symmetry);
AddToFeatureSet
  (symmetry,"simple","symmetry-snorm",PETSC_NULL);
AddToFeatureSet
  (symmetry,"simple","symmetry-anorm",PETSC_NULL);
PreprocessorSetSuitabilityFunction
  (cur,(void*)symmetry,&onlyforsymmetricproblem);
```

# Use suitability function

```
void onlyforsymmetricproblem
    (NumericalProblem p,FeatureSet s,SuitabilityValue *val)
{
NewFeatureValues(&values);
InstantiateFeatureSet(p,s,values);
GetFeatureValue(values,0,&sn,&f1);
GetFeatureValue(values,1,&an,&f2);
if (f1 && f2 && an.r>1.e-12*sn.r) {
  *val = ....   /* problem too unsymmetric */
}
}
```

# The Salsa System

# Capabilities of Salsa

- Extraction of features
- Storage of features
- Testbed for iterative linear system solvers (direct, nonlinear)
- Statistical analysis
- Automatic heuristic building and application
- Application monitor, dynamic testing, heuristic tuning
- . . . under development. . .

# Solver capabilities

- Using Petsc:
- Iterative solvers and preconditioners
- Pre-processing of the system: scaling, distribution, approximation, Schur complement (not yet implemented)

# Machine Learning / Statistical analysis

# Statistical approach

- Method choice can be approached as a classification problem
  - Class $\equiv$ Method
  - Classifier $\equiv$ Way of picking method
- Fuentes: Bayesian decision theory
- Ramakrishnan and Kang: Reinforcement Learning
- Bhowmick/Freund/Keyes: Boosting, Decision trees
- Zhang: Support Vector Machines (not in Salsa)

# Bayesian decision making

# Bayesian decision theory

Picking a method becomes a classification problem:

> *Given problem features $\bar{x}$, for which method $\omega_i$ is the suitability $P(\omega_i|\bar{x})$ maximized.*

# Bayesian decision theory'

- Take large set of test problems $\bar{x}$, and test them with all methods
- Assign them to the class of the fastest method
- $\Rightarrow$ Method $\equiv$ Class $\equiv$ set of problems on which this method is the fastest
- Construct the probability density functions (see below)
- Method choice: for a problem with features $\bar{x}$, take class $i$ that maximizes $P(\omega_i|\bar{x})$

# Bayesian decision theory

- Bayes' theorem:

$$P(A|B) = \frac{P(B|A)P(B)}{P(A)}$$

(from $P(A \wedge B) = P(A|B)P(B) = P(B|A)P(A)$)

- Used as:
  let $x$ be a feature vector, $\omega_i$ a class

$$P(\omega_i|\bar{x}) = \frac{P(\bar{x}|\omega_i)P(\bar{x})}{P(\omega_i)}$$

'what is the chance $\omega_i$ is the best method for problem $\bar{x}$'

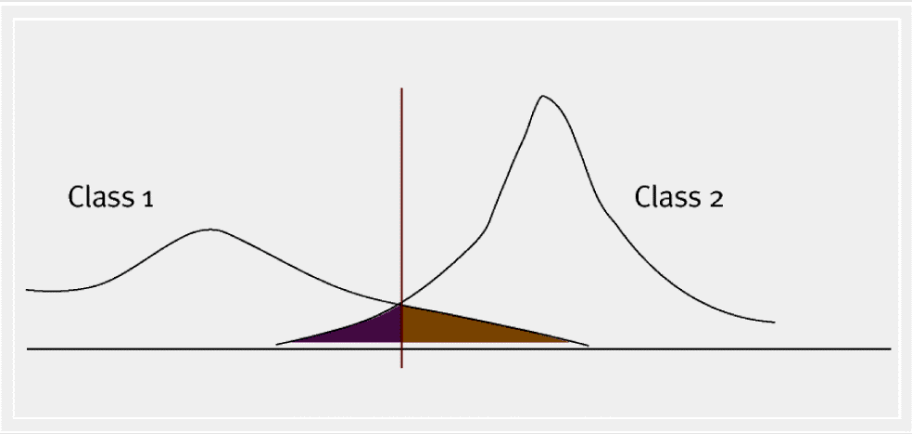
# Class-conditional probability

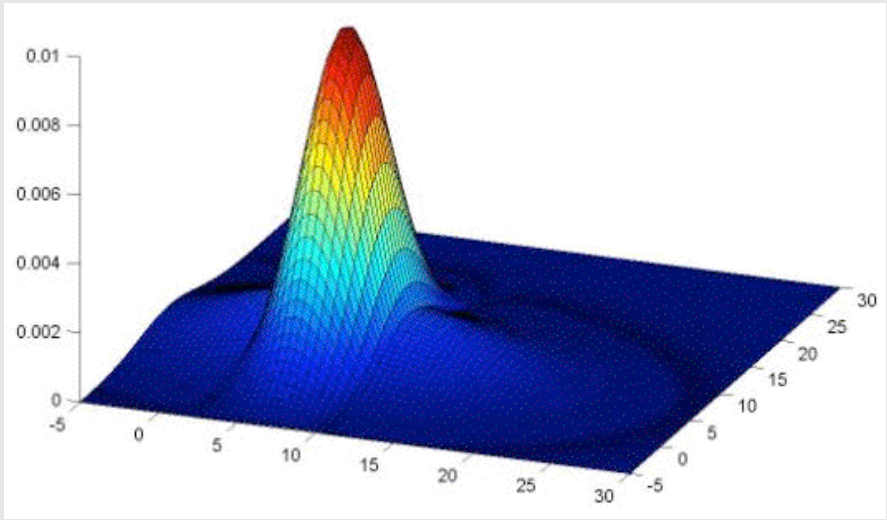
$$P(\omega_i|\bar{x}) = \frac{P(\bar{x}|\omega_i)P(\bar{x})}{P(\omega_i)}$$

- Most important quantity: 'class-conditional probability' $P(x|\omega_i)$
- Choice of 'probability density function'
- Assumption of Gaussian distribution:

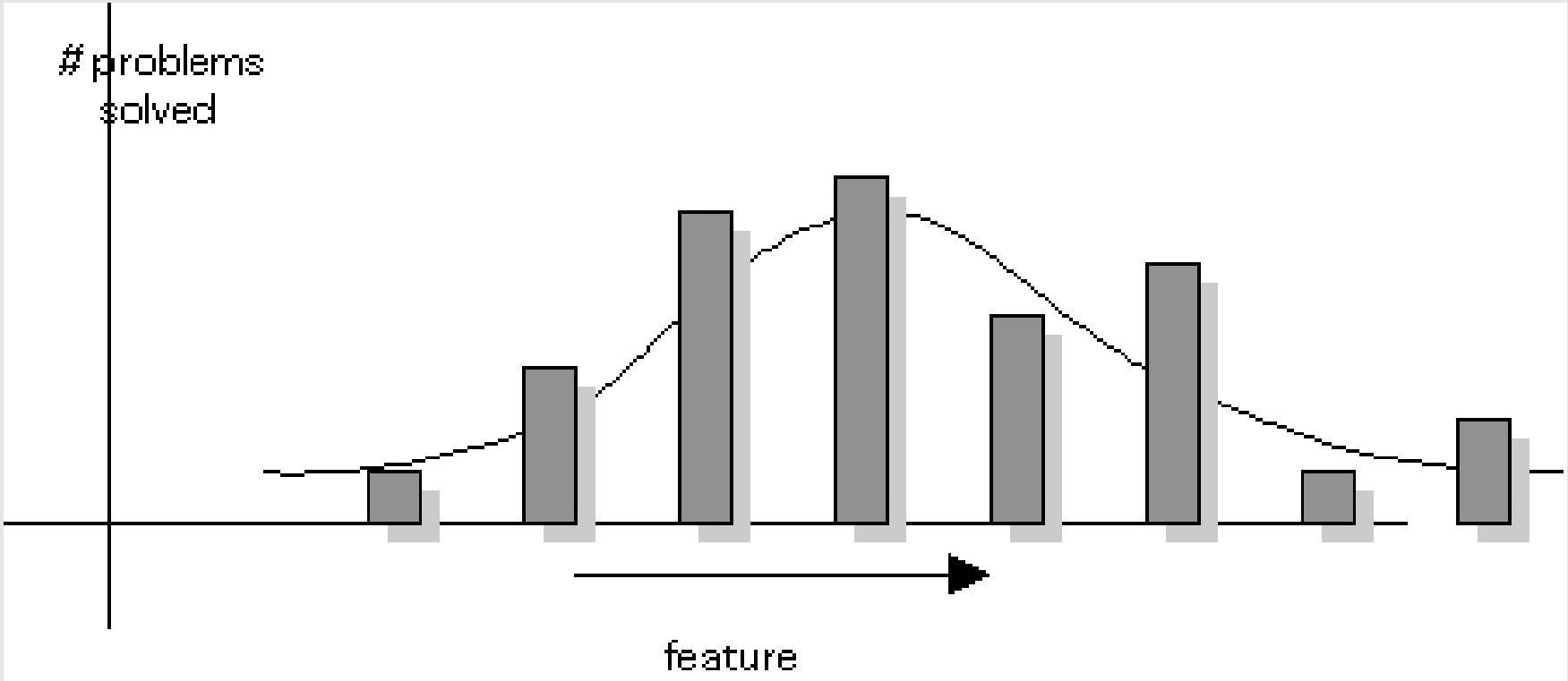
$$P(\bar{x}, \bar{\mu}, \Sigma) = \frac{1}{(2\pi)^{d/2}|\Sigma|^{1/2}} \exp\left\{-\frac{1}{2}(\bar{x}-\bar{\mu})^t\Sigma^{-1}(\bar{x}-\bar{\mu})\right\}.$$

where mean $\bar{\mu}$ and variance $\Sigma$ derived from data


Class 1
Class 2

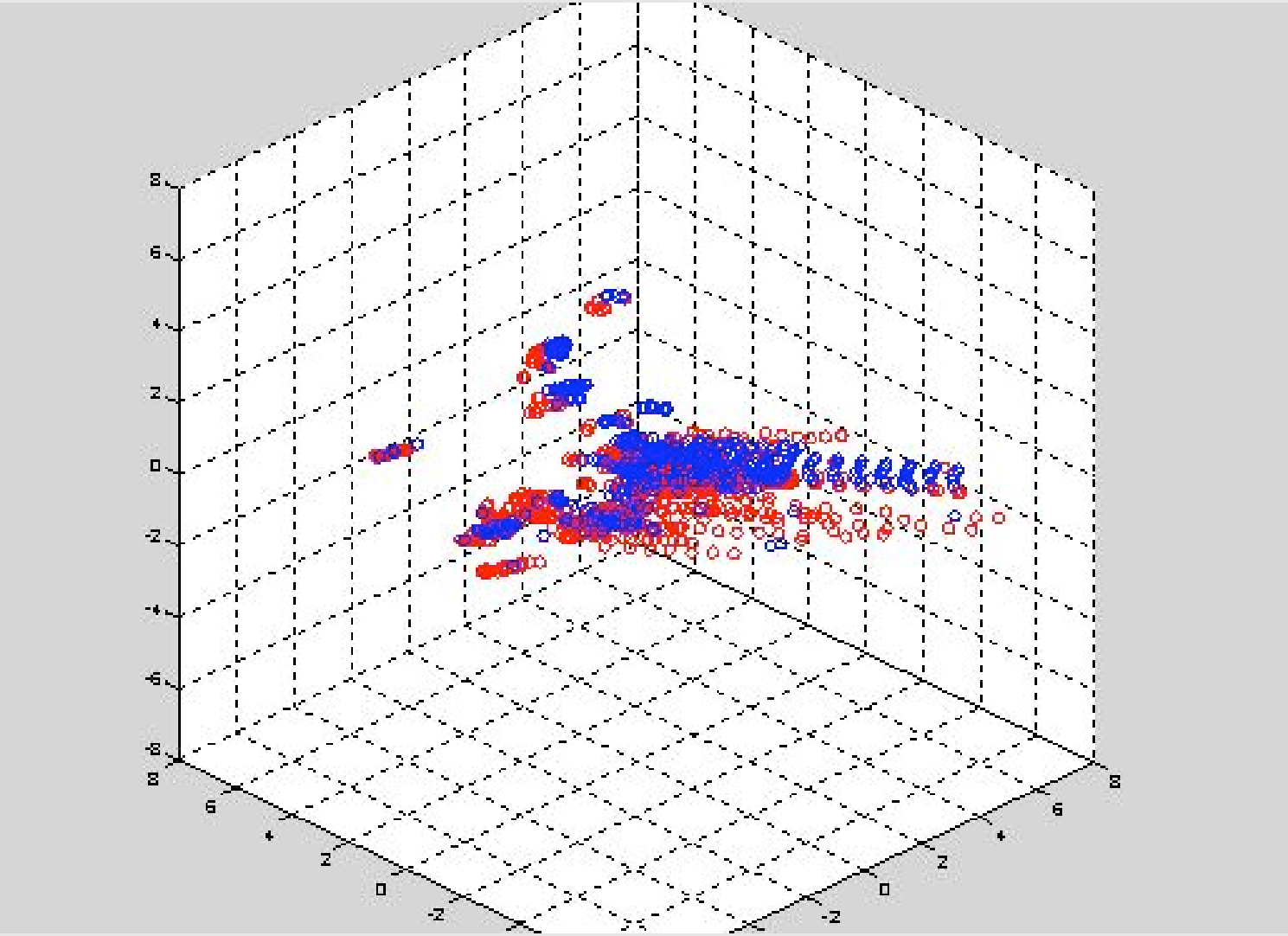

0.01
0.008
0.006
0.004
0.002
0
-5
0
5
10
15
20
25
30

# Gaussian assumption

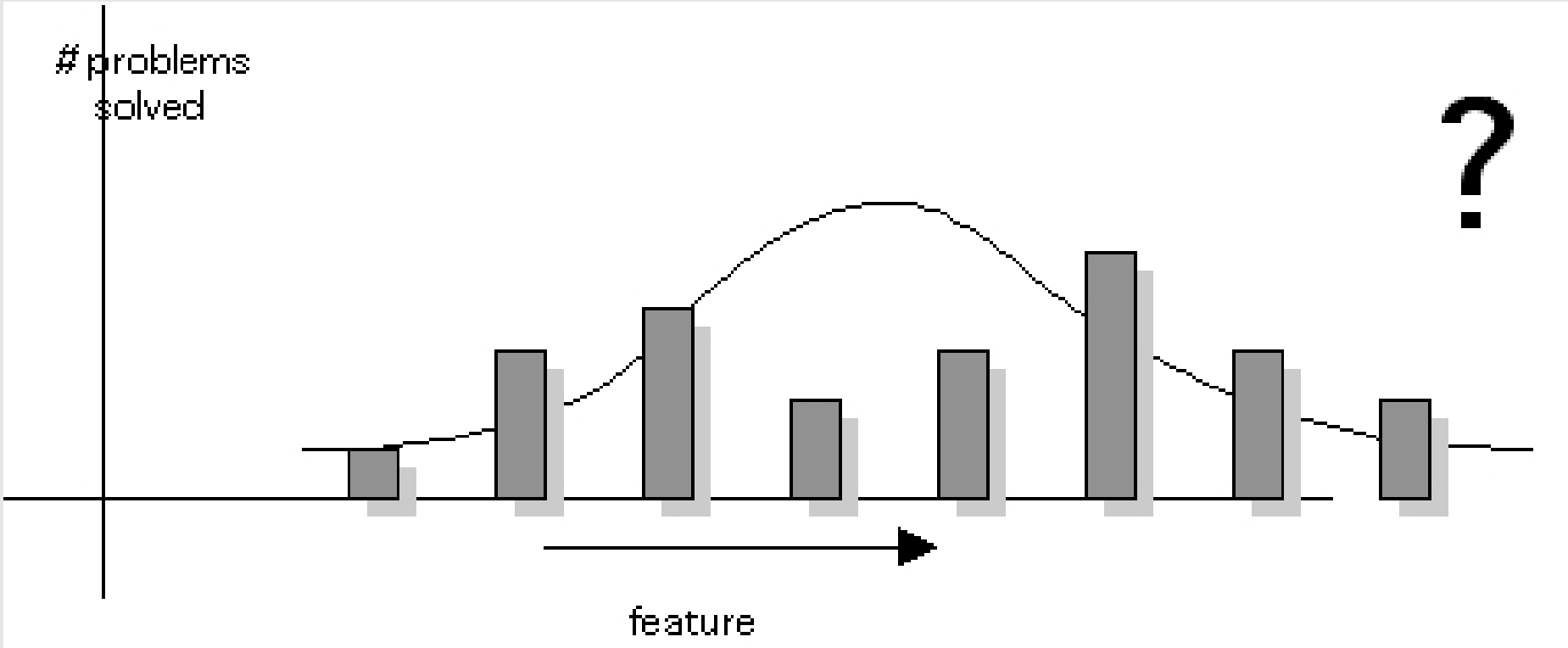


# Discriminant analysis

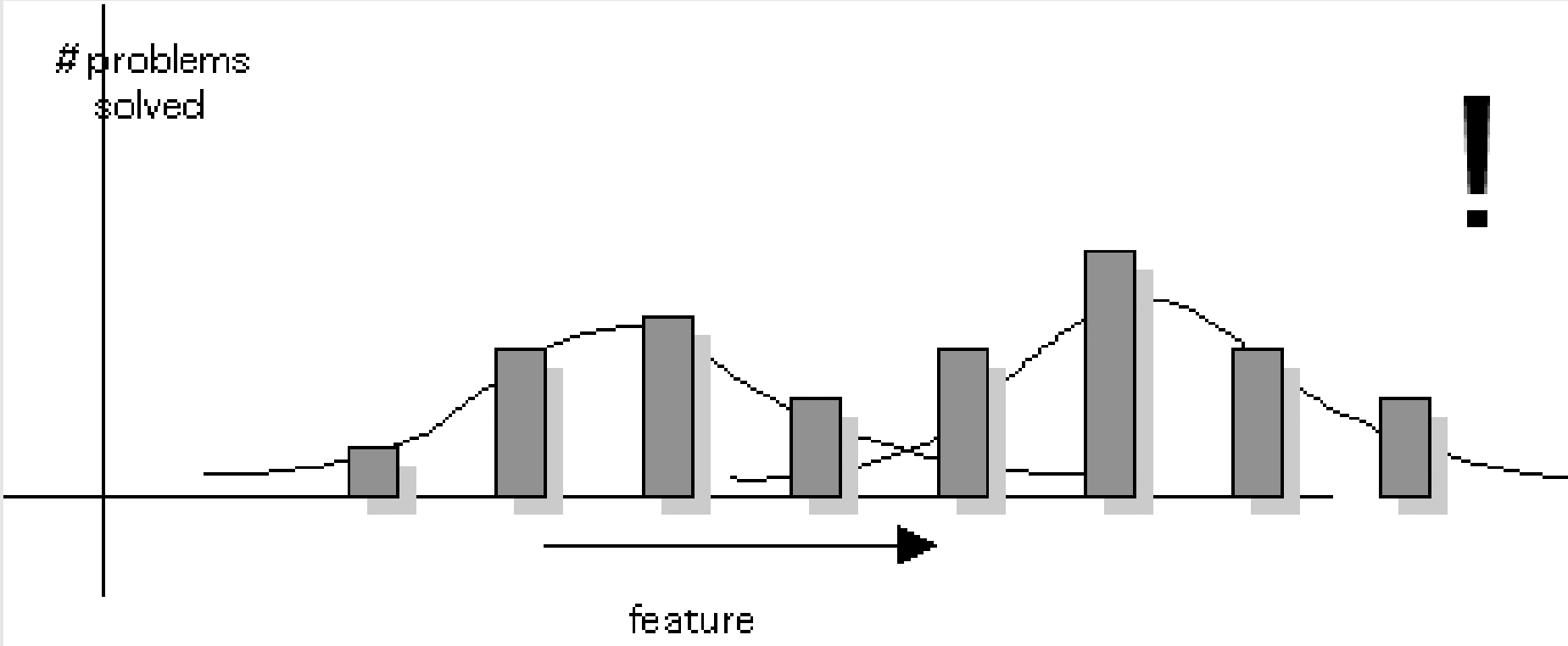
The assumption of a Gaussian distribution may not be justified:



# Kernel mixtures



# Kernel mixtures



# Test of Bayesian classification

# Outline of the test

- Apply principal component analysis (PCA) to features
- Use principal components to classify iterative methods:
  bcgs, gmres, tfqmr, direct
  maximum over all other options

# Relevant features

2 trace/norminf

7 symm_anorm/symm_snorm

10 row_variability

25 ellipse_ay/ax

28 kappa

29 positive_fraction

31 sigma_min / norm1

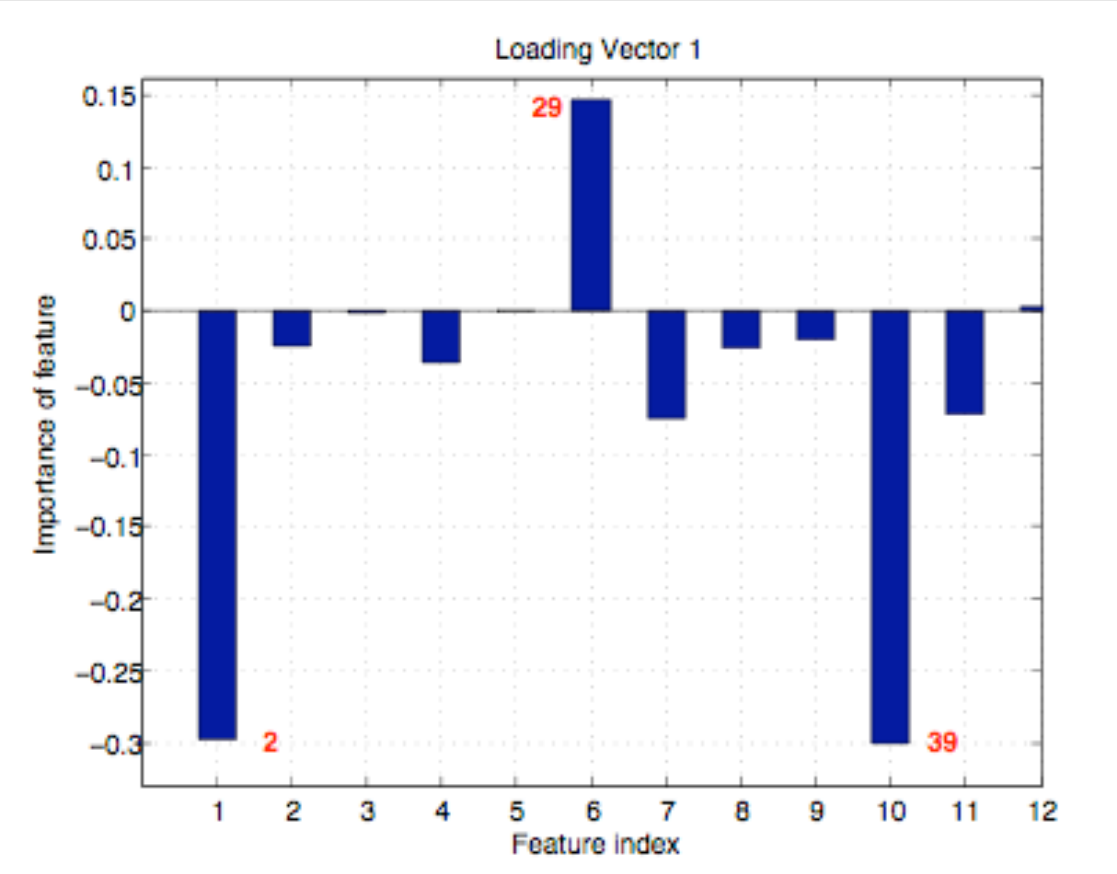
33 lambda_max_mag_im/norm1

34 lambda_min_mag_re/ lambda_max_mag_re

39 commutator_normF/ normF$^2$

40 ruhe75_bound

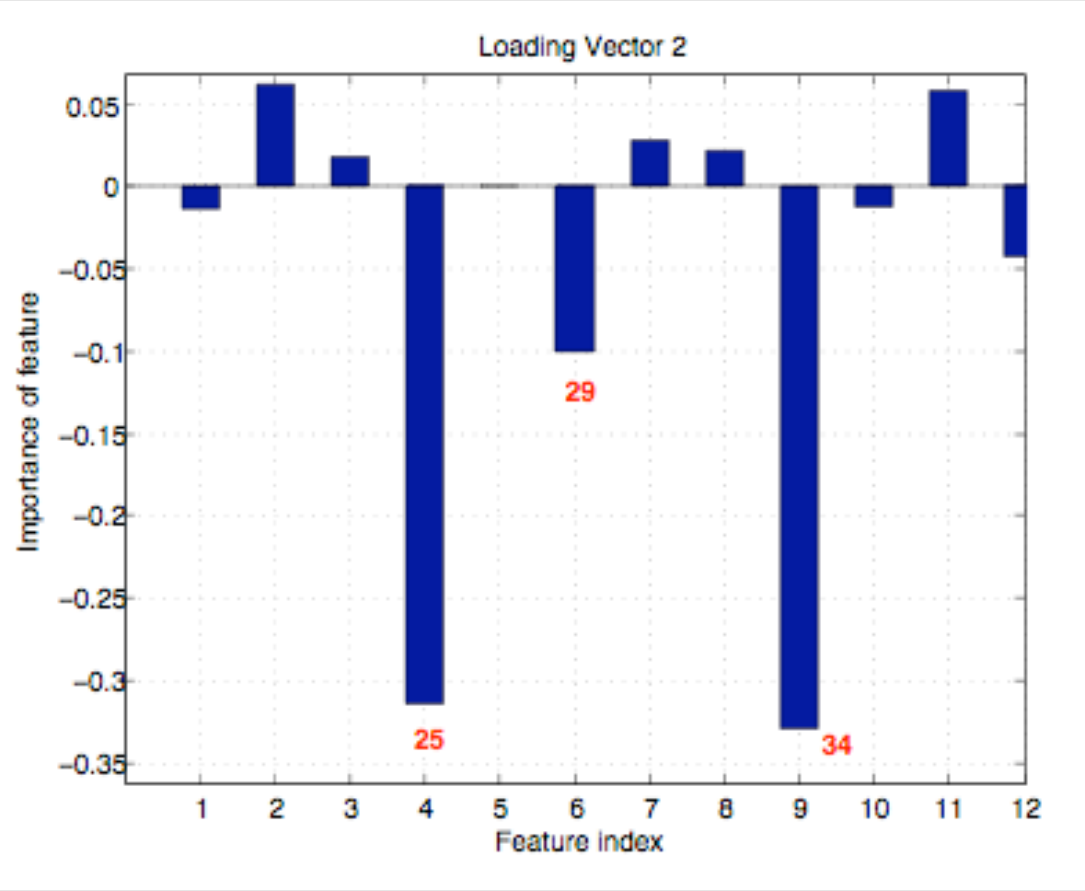
41 lee95_bound




2: trace/norminf,

29: positive fraction of spectrum,

39: commutator norm/ Frobenius norm squared




25: spectrum aspect ratio

29: positive fraction

34: $\mathrm{Re}(\lambda_{\mathsf{min}})/\mathrm{Re}(\lambda_{\mathsf{max}})$

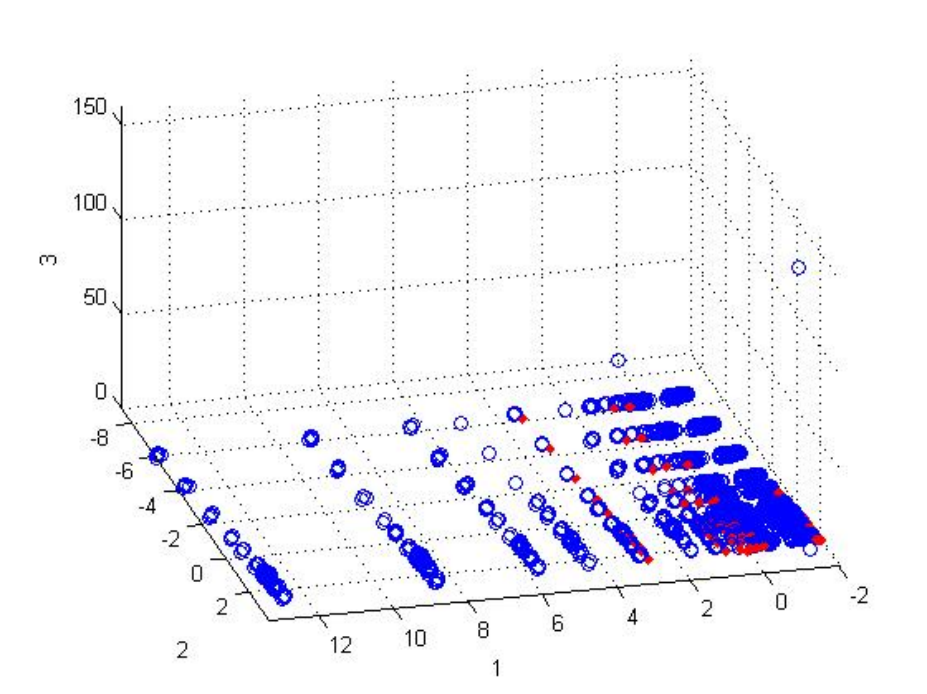
# Overall reliability

single gaussian rate: [0.456790 0.081571 0.830189 1.000000],
sum: 2.368550

kernel mixture rate: [0.901235 0.734139 0.443396 1.000000],
sum= 3.078770

# Speed vs reliability

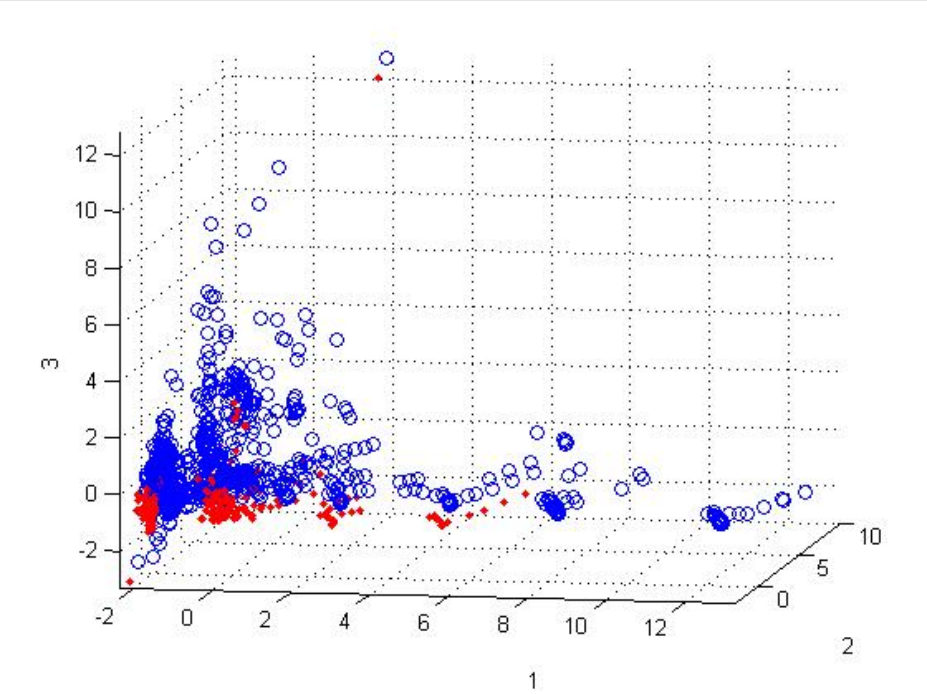
Accuracy figures measure true positives: not always the best measure.
Converge: 93.92%; No-Converge: 56.17%



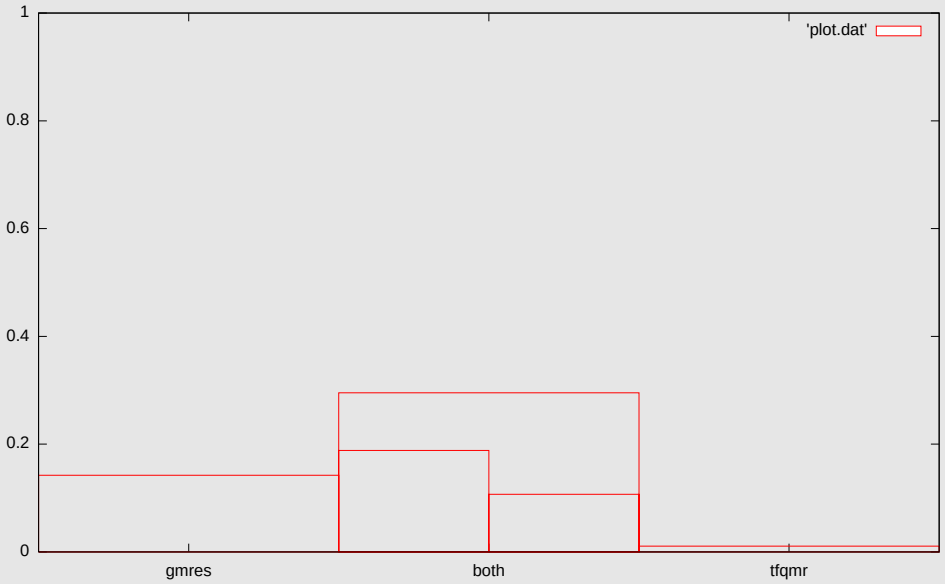

Serious danger of picking non-converging method.
Solution: classify on complement of convergence region, instead of non-convergence

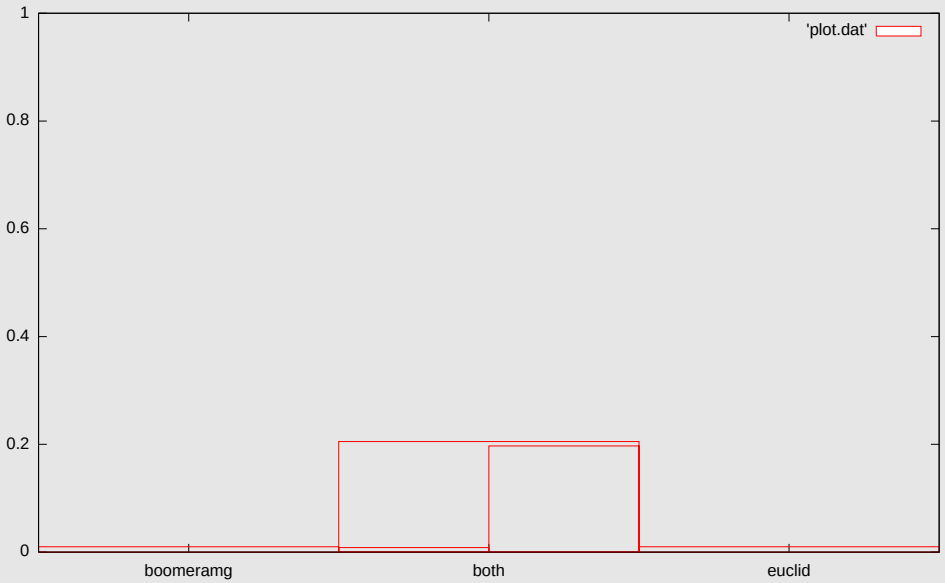
# Order of decisions

Classification of preconditioners easier than of iterators:



# Relative speed and reliability



# Reinforcement learning

# Supervised vs Trial/Error learning

- Bayesian learning is supervised: someone knows the right answer
- Reinforcement learning: by trial and error 'method A is better than B under conditions C'
- Advantages:
  - potentially accurate results with far less work, also easier to update dynamically
  - no database needed: learn through trial-and-error
  - multi-step analysis: possible to pin-point source of error (iterator, preconditioner, etc)

# Reinforcement learning

- Related to Dynamic Programming
- Problem formulated as stochastic shortest path problem (states, actions, instantaneous and final reward)
- Policy is determined through experience: running simulator

# RL for linear solvers

- Stages: pick load distribution, scaling, preconditioner, iterative solver
- Reward: negative of setup and solve time
- State space is divided in overlapping 'tiles'

# RL math

.

- Bellman optimality

$$R(i) = \max_{a \in A(i)} \left[ r(i,a) + \sum_{j \in S} p(i,a,j) R(j) \right]$$

- Q-factor for state-action pair $(i, a)$:

$$Q(i,a) = \sum_{j \in S} p(i,a,j)[r(i,a,j) + R(j)]$$

  then $R(i) = \max_{a \in A(i)} Q(i,a)$

- Observation: $Q(i,a)$ is the expectation of a random variable

# Q-learning

- Now use Robbins-Monro algorithm:

$$E[X] = \lim_{n\to\infty} (1/n) \sum_{i}^{n} s_i = \lim_{n\to\infty} X^n$$

then

$$X^{n+1} = (1 - \alpha^{n+1})X^n + \alpha^{n+1} s_{n+1}, \qquad \alpha^n = 1/n$$

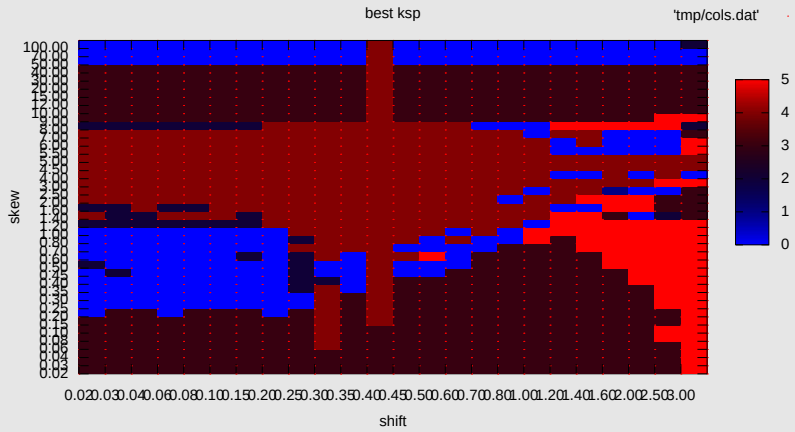
- Q-learning:

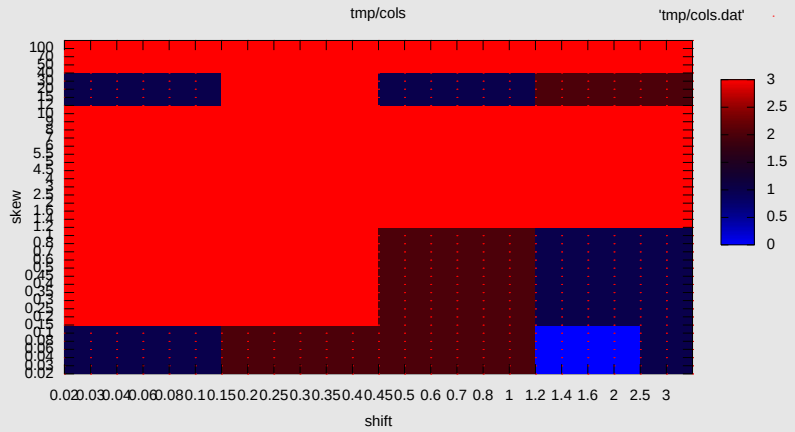
$$Q^{n+1}(i,a) \leftarrow (1-\alpha^{n+1})Q^n(i,a) + \alpha^{n+1} \left[ r(i,a,j) + \max_{b \in A(j)} Q^n(j,b) \right]$$

- Note: no probabilities! This can be run through a simulator.

# Test of Reinforcement Learning

## Best vs Recommended:




- Picks the right method in less than 60% of the cases
- however, in 95% is within 5% of optimal time.
- Possible spread can over 50×

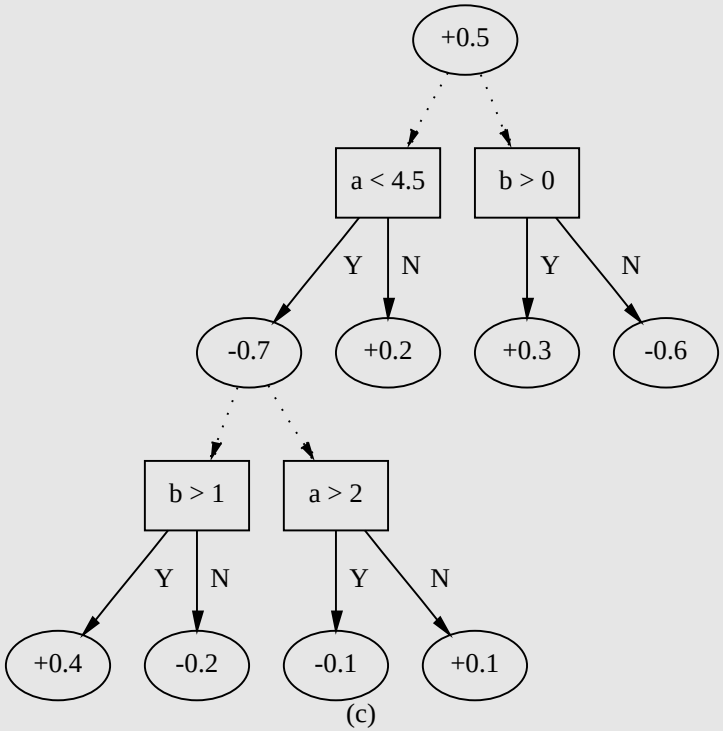
# Boosting

# Abstract formulation of machine learning

Binary classification rule maps instance space $\mathbb{A} \times \mathbb{M} \mapsto \{0, 1\}$
(equivalent to earlier $\mathbb{A} \mapsto \mathbb{M}$)
Without definition: weak PAC learning, strong PAC learning,
Boosting turns weak learning algorithm into strong.

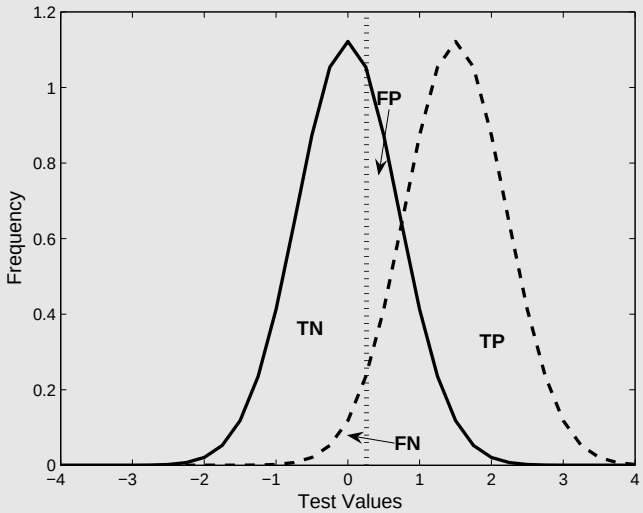
# Alternating decision trees

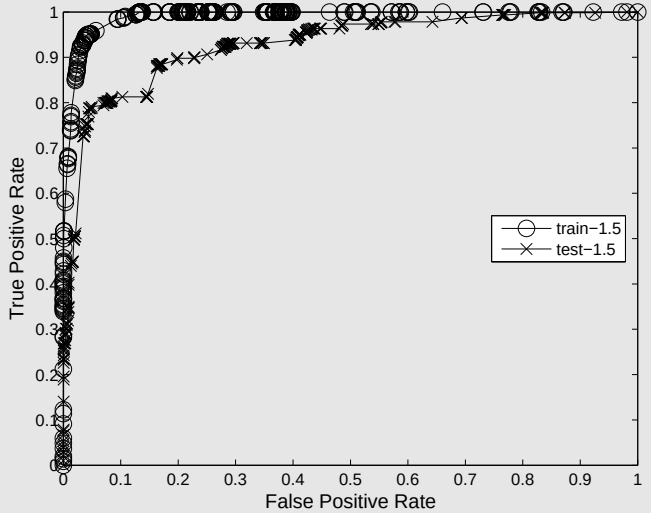
## Use as weak learner (MLJava package)



# Binary classification

Decision: does the method work faster than some reference method
Measurement by ROC curves:



# Conclusions and future work

- Software system: underway to a modular setup
- Bayesian: high reliability;
  to explore: combinations of steps
- RL: promising;
  to explore: more irregular test data, more adaptive algorithm
- Boosting: also promising;
  to explore: classification of reliability; prediction of components seperately.
- In general: non-model test data, more method choices, put all the pieces together.
- Integration in Petsc...
- Release of all software (`http://icl.cs.utk.edu/salsa`)