

A Proposed Standard for Numerical Metadata*

Victor Eijkhout[†] and Erika Fuentes[‡]

Abstract

We propose a standard for generating and storing metadata describing numerical problems, in particular properties of matrices and linear systems. The standard comprises a storage and a generation component. The storage consists of an XML file format and an internal data format with various access routines; the generation standard describes a format for software that produces metadata. We give the abstract description of the XML storage format, APIs (Application Programmer Interfaces) for generating and storing metadata, and a core set of categories of data to be stored, and software to generate them. The standard defines an open-ended format, allowing for other parties to define additional metadata categories to be generated and stored within this framework.

1 General discussion

Matrix storage formats, both file formats and data structures, traditionally limit themselves to specifying only the minimally necessary description of the data: the matrix size, and the matrix elements themselves with a fairly explicit description of the nonzero structure for sparse matrices. However, we can associate with matrix data any number of *derived properties*, such as norms, spectral properties, or graph properties in the sparse case.

There is no standard way of generating and storing such data, making interoperability hard between software modules written by different authors. Such interoperability would be valuable in a number of contexts. For instance, linear algebra algorithms often need, or at least have a use for, difficult to compute matrix statistics, such as condition number estimates. Thus, the full algorithm consists of two disparate modules: one analyser that estimates the numerical quantity, and the algorithm proper which uses this quantity to fine-tune its workings. While any ad hoc fit can be made between such analysis-producing and analysis-consuming software, a more general solution would make componentization of such software possible.

We note that analysis modules need not limit themselves, as in the example just given, to calculation or estimation of numerical quantities from the problem data. We can also envision that an application annotate its data before passing it to the numerical routine. Thus, information like the nature of a differential equation or its discretisation can be preserved, and translated to useful numerical information by a different kind of analysis module.

There is also use for a more permanent storage format of numerical metadata. We will argue both points, the programmatic and the storage aspects of metadata, in detail below.

The existence of a metadata standard for numerical data – we limit ourselves here to matrix data, though extension of these ideas to other fields is natural (see for instance [4, ch. 7]) – makes the following software functionalities possible.

* 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 grant # 0203984.

[†] Texas Advanced Computing Center, The University of Texas at Austin

[‡] Innovative Computing Laboratory, University of Tennessee

- First of all, it allows numerical algorithms to request metadata, not easily derivable from more traditional inputs, that will assist in the computation process.
- Secondly, it allows numerical data processing program components to annotate data with information that normally gets lost in the interface to the numerics.
- Thirdly, it makes it possible to encode two sorts of expert knowledge: the mapping of application-oriented data to numerics, and the decision making process based on wider aspects of the numerical data.

In two other applications, furthermore, we need not only a standard format for generating and storing such data programmatically, but also for more permanent file storage. The first application is the development of the Intelligent Agent of a Self-Adapting Numerical Software system [7, 6]. Here, the exhaustive analyses of properties of a number of matrices are stored in a database for subsequent analysis, together with performance results, for instance from solving linear systems with these matrices. New matrices can then be matched up against this database for recommendations as to preferred solution methods.

The second application where a metadata file format can be beneficial is in matrix collections, such as Matrix Market [14] or the University of Florida sparse matrix collection [5]. A standard format makes it easier to automate the insertion and analysis of matrices, as well as enabling complicated database queries. On the retrieval side of the collection, it means that any dataset extracted from the collection comes with substantial standardized information.

Our proposal for a metadata standard comprises the following.

- First of all we propose a standard for storing metadata. The standard has a combined sequential/recursive structure that gives easy access to a number of commonly needed elements, as well as extendability that allows third parties to add metadata categories.
- We propose a number of categories of metadata, that are picked to cover most common applications in numerical matrix analysis.
- We propose an API for generating numerical metadata in this format; an example implementation is given in our AnaMod (Analysis Modules) library; for software availability see the conclusion to this article.
- We define an API (Application Programmer Interface) that allows access to the metadata data structures from inside application codes. The API allows both retrieval as well as insertion of data; an example implementation is given in our NMD (Numerical MetaData library).
- We propose an XML file format for permanent storage of the matrix metadata. We have written software that converts from this XML format to internal data structures.

While the XML file format is the most visible aspect of this proposed standard, not all applications need it. Some of the usage scenarios below will both generate and consume the metadata in internal form, that is without involving an external dump to XML.

Section 2 will give examples of the use of metadata in practice. We define the metadata format in the abstract as an XML format in section 3.1; that section will also describe the API to convert and access the contents of the XML file in the context of a program. Section 4 gives a proposed core set of metadata elements, and section 5 outlines some future directions for this software.

2 Use of metadata in practice

In this section we give a few motivating examples of the use of metadata. Our formalization of the metadata and access to it will follow in the next section.

2.1 Usage scenario 1: intelligent algorithm

As the simplest example of the use of metadata, we consider algorithms where some parameter could be tuned if certain properties of the input were known. For instance, for GMRES [16] there is a theorem where the norm reduction in one restart cycle is estimated in terms of the restart length, for indefinite matrices where only a small number of eigenvalues have negative real parts. Knowing the structure of the spectrum then makes it possible to choose the restart length intelligently.

At the moment, any software component that needs metadata on numerical input needs to compute this itself, or know how to call external software that can perform the computation. To disentangle these concerns, we can envision the specifications of the numerical component asking for a certain piece of metadata, which can then be computed on demand by the main program, or, slightly more ambitiously, by a programming framework that controls the integration of the components [9].

2.2 Usage scenario 2: multi-method decision making

More complicated than a single component needing some piece of metadata, we can envision a decision making process in some physics application that chooses between different variants of a numerical algorithm based on metadata. For instance, a factorisation routine for symmetric matrices can unambiguously choose a Cholesky factorisation if the matrix is positive definite. If it is known to be indefinite, the routine can perform an LU factorisation with (partial) pivoting from the outset.

In this case, the metadata is again computed on demand, but need not necessarily be passed on to the component that is ultimately chosen. We may require the standardised format in this case to include application-related metadata from the physics application. The decision making component will then not just compute numerical metadata, but will also perform some mapping from physics characteristics – the metadata as provided by the application – to numerical characteristics. For the above example, the translation from physics to mathematics is simple: a matrix is positive definite if the operator is coercive.

2.3 Usage scenario 3: self-adapting intelligent agent

Expanding on the, somewhat ad-hoc, decision making component in the previous scenario, we can envision a decision maker that evolves over time, saving matrix analyses and performance results, to make increasingly accurate algorithm predictions as more problems are handled by it [1, 7]. The metadata here plays two roles: first of all, it is used as the storage format for the analysis of earlier encountered matrices. Secondly, for any new matrices to be handled by the intelligent agent, the agent calls analysis modules that will fill in the metadata structure. Its contents can then be matched against the database of earlier matrices to arrive at an algorithmic recommendation.

In this scenario we need the metadata both internally as a data structure, and externally stored in the analysis database, for which we use the XML format. In the previous two scenarios an external format was only needed if the data is passed by file between application components.

2.4 Usage scenario 4: data repository

There exist several public (and probably several more private) matrix repositories. We name Matrix Market [14] and the University of Florida collection [5] as commonly known examples. Our metadata format, and in particular the XML file format, can both simplify the upkeep and increase the usefulness of such collections.

The metadata standard, coupled with the existence of standard-conforming analysis modules, makes it easier to add new matrices to the collection. Since analysis modules can be written by third parties, adding new

statistics to the collection also becomes easy. The XML standard coupled with an XSL style sheet also makes the display of matrices a cinch.

On the user side, it means that a matrix retrieved from such a collection will come with a file of metadata that can immediately be absorbed by the user's program. Right now, every matrix storage format has its own standard for storing such metadata, and usually there is no core set that can be depended on to be present.

3 Metadata definition

We propose to organize metadata in a two-level structure, where the top level items are called *categories* and the second level *components*. Organization in categories can be done along several principles.

- By topic, for instance having a category for metadata relating to the nonzero structure of the matrix, or to its spectrum.
- By generating software. Below we present the AnaMod package that computes a number of basic categories. However, it can be extended by interfacing it to external packages. Those are best incorporated as a separate category.
- Application-specific. Some metadata items may not be of general use, in which case they are best confined to their own category.

The organization in categories also allows for multiple ways of computing the same item.

The two-level setup of the metadata is reflected in the APIs of both the storage library NMD and the computation library AnaMod.

3.1 Metadata storage

In several of the usage scenarios above we have seen the need for permanent storage of the matrix metadata. We propose an XML [18] based file format here. The format is formalized in an XML schema [19], and we provide an example XSL style sheet [20] that will display the XML file on a web page.

The XML metadata on a matrix can be stored in at least two ways:

1. In the 3rd usage scenario (Intelligent Agent) we retain only the matrix analyses, discarding the matrices themselves. This means that the XML files can be stored by themselves, in a file system or in a database.
2. In the 4th usage scenario (Matrix Repository) the matrices are retained; in that case we can merge the matrix file and the analysis file if the storage format allows this. For instance, the Matrix Market format [15] has a provision for unlimited comments. The XML file could be inserted in this manner.

In this section we discuss the API to the data structures. This discussion is only meant to convey the essential ideas; a full users guide can be found in [10].

3.1.1 Usage

The API to our metadata standard consists of a small number main routines, in three categories, plus some lower level utility routines:

- Constructing the data structure, and adding categories and elements to it;
- Retrieving information from the data structure, and inserting new information into it;
- Reading an XML file into a data structure, writing this data structure to an XML file, and displaying the XML file as HTML.

The cleanest use of metadata is in the context of a component-based programming framework [9], as described above. The component that needs the metadata would declare this fact to the framework, which then creates the relevant categories in the metadata structure, and calls an appropriate analysis module which fills in the requested elements.

A workflow that involves a programming framework (section 2.1) would be as follows:

1. A main program would declare to the framework which categories and elements are needed;
2. the framework would find the analysis modules that compute these, and create space for the elements they compute;
3. the framework would then activate the analysis modules.

In the absence of such a framework, the creating and inserting can either be done by the application or by the analysis module. In the former case the analysis module merely needs to return numerical values, which the application will insert into the metadata structure after creating space for it. In the latter case, the analysis module will create and fill in the requested categories and elements. This requires the analysis module to have been written with knowledge of the metadata formalism. We note that both approaches suffer from an entanglement of concerns, which is neatly obviated by the use of a framework.

We envision a workflow in a more traditional program/library context as follows:

1. A main program knows what analysis modules are available;
2. the analysis modules declare what the name of their category and its elements are;
3. the program then creates space in the metadata structure for these elements – or the ones it needs – and calls the analysis module to fill them in.

3.1.2 Data structures for numerical metadata

In this section we explain the NMD (Numerical MetaData) library which is designed to facilitate the manipulation of stored numerical metadata. The main data object is of type `NMD_metadata`, which is a pointer to a structure. Hidden to the user there are `NMD_metadata_category` and `NMD_metadata_item` structures which hold category and module data respectively.

A metadata structure is constructed and deleted with

```
int NMDCreateObject(NMD_metadata*);  
int NMDDestroyObject(NMD_metadata);
```

after which categories and components are created with

```
int NMDCreateCategory(NMD_metadata, char *cat);  
int NMDCreateComponent(NMD_metadata, char*, char*, NMDDataType);
```

Both create calls take text labels, the component create call has an extra argument for the component data type, which is an enumerated type.

The library has various utility functions, such as tests for existence:

```
int NMDHasCategory(NMD_metadata, char *cat, int*);  
int NMDHasComponent(NMD_metadata obj, char *cat, char *cmp, int *f);
```

Once the metadata structure has been set up, values can be set and queried with

```
int NMDSetValue(NMD_metadata, char*, char*, void*);  
int NMDGetValue(NMD_metadata, char*, char*, NMDDataType*, void*, int*);
```

Note that the actual values are passed, both ways, as void pointers, and the query routine has an output flag to indicate success or failure.

3.1.3 Conversion routines

The routines in this section convert from a numerical metadata structure to an XML file and the other way around, and convert the XML file to HTML based on an XSL style sheet. We provide both an XML schema validating the XML files, and a sample XSL style sheet.

```
int nmdSerialize
(NMD_metadata *obj, char **buf, int *len);

int nmdDeserialize
(char *buf, int len, NMD_metadata **obj, char *schema_name);

int nmdXML2HTML
(char *xmlfile, char *xslfile, char *htmlfile);
```

3.2 Analysis modules for metadata generation

In this section we present our AnaMod (Analysis Modules) library. The library present a number of analysis routines for common (and some uncommon) matrix quantities, but it also present a uniform way of registering and using them. This makes it possible to incorporate other analysis software into the AnaMod framework.

Unlike the NMD library, which is a standalone product, AnaMod heavily relies on the Petsc library [2].

3.2.1 Uniform access

For a flexible architecture we have to go beyond simple computational routines like

```
ComputeMatrixTrace(<matrix>, &trace);
```

since we can not assume these routines to be known by name. Rather, a more dynamic approach results from having a general computational routine

```
ComputeQuantity(<matrix>, "simple", "trace", (void*)&trace );
```

and enquiry routines such as

```
SystemHasModule("simple", "trace", &flag );
```

Such routines are a simple but effective API to a system where modules can be added dynamically and transparently.

3.2.2 The AnaMod API

As discussed above, our model is one where computational routines are dynamically registered. This is done with

```
PetscErrorCode RegisterModule
(char*, char*, AnalysisDataType,
int (*f)(NumericalProblem, AnalysisItem*, PetscTruth*));
```

where the arguments are the category and component name, the datatype of the computed quantity, and the computational routine.

The uniform computational routine in AnaMod is then

```
PetscErrorCode ComputeQuantity
(NumericalProblem, char*, char*, AnalysisItem*, PetscTruth*);
```

Our semantics state that this routine is allowed to fail for any number of reasons (invalid data, overrun of compute time), which is reported in the trailing flag parameter.

Several utility routines exist, such as the test for a routine having been defined:

```
PetscErrorCode HasComputeModule(char *cat, char *cmp, PetscTruth *f);
```

3.2.3 Data format dependence

Since analysis modules need access to the matrix to be analysed, there will be dependence on the format used. Although the syntax of the compute routine specifies an abstract ‘numerical problem’ as input, any specific problem will have to be cast. In our case, the software is based on the Petsc library [2, 3], so the computational modules cast the problem to a Petsc Mat object.

4 Core set of categories and elements

Our proposed metadata format is quite abstract, allowing for the inclusion of any kind of data. However, there are many matrix statistics in general usage. We propose a number of categories of statistics, covering these common elements. Apart from the common sense notion that an initial delineation of such elements will prevent conflicting third-party definitions later on, we hope that providing a sufficient vocabulary for common applications will also enhance our chances for wide adoption of our proposal.

In this section we will outline our proposed basic categories of metadata.

4.1 Storage format

In genera we do not attach XML files or data structures to abstract mathematical operators, but to representations of these operators. This implies that a description of the storage format, independent of the mathematical properties of the stored object, is entirely appropriate. However, we are not aiming to give a full formalization of matrix storage formats, but rather information knowing which can make processing the file more efficient.

Here are some of the elements of the storage format metadata category:

Metadata category: Storage Format		
Element	Value type	Description
format	char*	name of storage format
elements	integer	number of stored elements
zeros_stored	logical	are any zero elements stored?
unique	logical	is every (i, j) location specified at most once?
symmetry	"upper", "lower"	is this a symmetric matrix with only half the elements stored?
sorted	"row", "column"	are elements sorted?

Comments.

- The number of nonzeros of a matrix is a precisely defined number. In the context of a matrix stored on file we record the number of *stored* elements, which can be larger than the number of nonzeros if some zeros are explicitly stored.

- The `symmetry` element is not concerned with numerical values or even structural symmetry; if the matrix is indeed symmetric, this element records whether it has indeed been stored as such.
- The `unique` element allows for the case of unassembled finite element matrices.

The Harwell-Boeing file format [12], used to define the Harwell-Boeing test collection [8], has a small set of such storage metadata information, encoded in the file name extension. Its three letters denote the number field, symmetry, and assembled / unassembled respectively. Of these, we move the number field information to the category of structural data; section 4.2.

The Matrix Market format [15] specifies some of the above elements on the first line of the file. It also contains a provision for further comment lines; however, no formal proposal for standardising these comments is made.

4.2 Structure data

In applications such as non-linear system solving, a sequence of matrices often occurs that vary in their numerics, but that will have some structural invariants, typically because they arise from the same discretisation of a physical domain. Here we define a category of structure information to capture these elements. Another way to characterise these elements is to say that they largely depend on the nonzero structure of the matrix.

Metadata category: Structural Statistics		
Element	Value type	Description
<code>m, n, nnz</code> <code>number_system</code>	integer "integer", "real", "complex", "pattern"	size, number of nonzero
<code>shape</code>	"dense", "banded", "triangular", "diagonals"	
<code>symmetry</code> <code>bandwidth,</code>	logical int[2]	structural symmetry left and right halfbandwidth
<code>diagonals</code>	int, int[]	number of diagonals and their locations
<code>diagonal</code>	"positive", "semi-pos.", "zero from"+int	zero and sign structure of diagonal
<code>block_size</code>	int int*	regular/irregular block structure
<code>single_elt_rows</code>	int kind, int[]	type and location

Some comments.

- Left and right halfbandwidths p_ℓ, p_r are defined by

$$j < i - p_\ell, j > i + p_r \Rightarrow a_{ij} = 0$$

- The zero structure and sign structure of the matrix diagonal are often determined by the application. For instance, both mixed finite elements and KKT optimisation lead to matrices with a zero $(2, 2)$ block, hence with a diagonal that is zero from a certain point onwards.
- In some applications, matrices will have rows containing only a single element, typically the diagonal, and often with value one. These rows correspond either to Dirichlet boundary conditions, or to nodes

in a fictitious domain; see section 5.3 for further discussion. The `kind` parameter takes values 1 for all ones on the diagonal; 2 for non-unit values on the diagonal; 3 for the possibility of off-diagonal single elements.

4.3 Simple data

Under simple data we rank numerical data that can be computed exactly in time proportional to the number of nonzeros.

Metadata category: Simple Statistics		
Element	Value type	Description
<code>norm_1,</code> <code>norm_inf, norm_F</code> <code>symmetry_type</code>	double "symmetric", "anti-s.", "complex-s.", "Hermitian", "anti-H."	norms
<code>s/a_norm_1,</code> <code>s/a_norm_F</code>	double	norms of symmetric and antisymmetric part

At this point we remark on an implementational detail. The norms of the symmetric and anti-symmetric part of a matrix ($(A + A^t)/2$ and $(A - A^t)/2$ respectively) are computed with almost identical code. Our software actually computes both simultaneously, and stores the quantities in the matrix object, possibly to be retrieved later at negligible cost. Petsc has support for this mechanism.

4.4 Spectral data

Of particular relevance to applications involving iterative methods are various measures of spectral matrix data. We identify the condition number, various measures of the spectrum (field of values), and the departure from normality. Other candidates for this category would be some description of pseudo-spectra [17] and polynomial numerical hulls [11].

Metadata category: Spectral Statistics		
Element	Value type	Description
<code>condition</code>	double	condition number estimate
<code>ellipse</code>	double[4]	centre and axes of ellipse enclosing the field of values
<code>hessenberg</code>	double[][]	Hessenberg matrix from a short Arnoldi run
<code>dep_norm</code>	double[2]	departure from normality (upper/lower bound)

Spectral information such as this is not easily computable. In the routines provided in the AnaMod library they are approximated by evaluating the Hessenberg matrix that arises from a short GMRES run.

The elements in this category are not mutually exclusive; for instance, the enclosing ellipse of the field of values can be computed from the Hessenberg matrix. This means that we run into the consistency problem alluded to in section 5.2. The reason for storing the full Hessenberg matrix is that from the way the spectrum estimates develop we can gain a higher confidence in the bound derived in the final step.

4.5 Application-derived data

The above categories were characterised by the fact that their elements can be directly derived from the matrix, and that often this derivation has some computational cost. By contrast, the generating application can supply information that is lost once the matrix is formed, and that can be given at essentially zero cost.

Knowing details of the application that generated the matrix can often be valuable. For instance, iterative methods are unlikely to be successful in solving linear systems that come from discretised ODEs; matrices from constrained optimization have a structure that can be exploited by certain algorithms; the decision to extract a symmetric part of the matrix makes less sense in the case of upwind differencing than for central differencing of the convection term; it is valuable to know what type of boundary conditions were applied and in which variables; et cetera.

This part of our core set of metadata categories we leave undefined for now; we hope to develop this later in collaboration with application scientists.

4.6 A custom category: variability

In this section we present an example of a custom category. The following elements could be characterised as simple, but since they are non-standard we give them a new category. They gave various heuristic measurements of how far the matrix is from a model problem.

Such measurements have successfully been used in our SALSA system [7, 6]. For instance, a large difference between variability in rows and columns is a good indicator for asymmetric (left or right) scaling of the matrix. Low values of the diagonal variance imply that the work of scaling the matrix may not be worth the effort and storage space.

Metadata category: Variability Measures		
Element	Value type	Description
Gershgorin	double[2]	lower and upper bound on eigenvalues
variability	double[2]	element variability in rows and columns
diagonal	double[2]	average value and variance of the diagonal
symmetry	double[2]	norms of symmetric and anti-symmetric part

For completeness we give the definition of the less familiar of the above elements:

- Gershgorin bounds:

$$\min_i |a_{ii}| - \sum_{j \neq i} |a_{ij}|, \quad \max_i |a_{ii}| + \sum_{j \neq i} |a_{ij}|$$

- Row and column variability:

$$\max_i \log_{10} \frac{\max_j |a_{ij}|}{\min_j |a_{ij}|}, \quad \max_j \log_{10} \frac{\max_i |a_{ij}|}{\min_i |a_{ij}|}$$

- Diagonal average and variance:

$$\alpha = \sum_i |a_{ii}|/N, \quad \sigma = \sqrt{1/N \sum (|a_{ii}| - \alpha)^2}$$

5 Future work

There are a few problems that still need addressing in a further refinement of this draft standard. We will outline these issues below; however, we start with a brief discussion of further custom categories that address other issues than mathematical properties of the data.

5.1 Custom categories beyond linear algebra

The metadata categories mentioned above, both the core categories and the example custom category, were solely concerned with the mathematical properties of matrix data. It also makes sense to use metadata to describe further problem properties.

application properties There are various items of information that are known to the application, that are lost in the interface to the numerical linear algebra software, and that are potentially useful, for example in the ‘intelligent agent’ scenario (section 2.3). Examples are: coordinates of the grid points, which could be used in geometrical domain decomposition, or the order of finite element approximation used.

platform properties Various implementation details of algorithms can be informed by parameters of the computational platform. For instance, Langou [13] used the ratio between the cost of inner products and matrix-vector products to tune parameters in a multiple-rhs GMRES method.

5.2 Relations between data elements

It is inevitable that some elements of metadata categories will not fully independent. As a simplest example, an element of one category may be identical to an element of another category. This can happen in the case where a category is added which comprises an already existing full set of statistics, for instance the statistics currently used on Matrix Market [14].

As another example, the core set of elements proposed below contains as part of the structural information both the left and right halfbandwidth of a matrix. From this we can derive the bandwidth by simple addition. However, while we want the user to be able to query the bandwidth, we do not want to store this quantity explicitly, since this might cause consistency problems. Hence we could define this element as

```
(computed (+ (component "structure" "left-halfbandwidth")
              (component "structure" "right-halfbandwidth") ) )
```

For a more complicated example of dependence of metadata elements, the spectral positive definiteness element is implied by the Gershgorin bounds (which are in the simple category of our proposed core set; section 4) being positive. Such a relation could be defined by

```
(implies (> (component "simple" "gershgorin-bound") 0)
         (t (component "spectral" "positive-definite") ) )
```

in the XML schema. Note that this example involves a relation between elements from different categories.

5.3 Linked XML files

In the course of numerical treatment, matrices undergo various transformations that may influence the metadata. For instance, prior to solving a system, the matrix may be permuted using some fill reducing ordering. This changes the structural properties while leaving spectral properties intact. As another example, in some codes, Dirichlet boundary nodes get written into the matrix, leading to rows with a single element 1 on the diagonal, thereby inducing a multiple eigenvalue 1. This may influence the condition number of the matrix. However, of interest to us is the ‘effective condition number’ found by removing these rows. These ‘single element rows’ also influence structural properties of the matrix.

For such reasons we need to establish a way of linking XML files together, where a linked file corresponds to a matrix derived from another, and which may inherit certain metadata elements, and differ in others.

5.4 Inexact data

Certain elements of the core set of categories we outline in section 4 can not be computed exactly, the condition number estimate being one. For such quantities the element can have an attribute giving either an uncertainty interval, some measure of confidence, or (perhaps in addition to the previous) a statement as to how the quantity was computed.

For an example of the latter, consider the departure from normality. This quantity is an important determinant of the behaviour of iterative methods. However, it can not efficiently be computed, and there are several existing bounds. In this case we may want an array of bounds, each one ‘signed’ by a different author or piece of software.

One solution would be to let quantities such as `double condition` be replaced by a list

```
struct {double condition; char *authority;} *condition;
```

This accomodates a number of differently derived estimates, each signed off on by a different algorithm.

The alternative to having a list of bounds would be to have different components, corresponding to the different estimates. Clearly, since these different components would all be specifying the same element, this would be a violation of our two-level setup, hence less desirable as a practical implementation.

5.5 Size of stored data

In the case where metadata is stored in a data structure to be used only inside a code, the size of stored elements is largely irrelevant, since their storage can be implemented through storing a pointer to the actual data. (Should we ever want to store objects such as a preconditioner, then this is in fact the only solution in practice.) However, for metadata that will be written to file, the size of stored data *is* a problem.

Here we will limit ourselves to identifying a succession of metadata classes of ever increasing size. Let us consider the estimation of a matrix spectrum, in particular the condition number, by performing a relatively short run, say $k = 100$ iterations, of GMRES.

- An actual estimate of the condition number is a single scalar; estimating the enclosing ellipse of the field of values takes four scalars.
- Instead of describing the enclosing ellipse, we could store the actual k values approximating the field of values.
- As we argue in section 4.4, it is more informative to store the $k \times k$ Hessenberg matrix, rather than its eigenvalues.
- Through this GMRES process, or other processes, we can compute a few eigenvectors corresponding to outer eigenvalues. Storing these takes $O(N)$ space.

Finally, if we are considering preconditioned processes, storing the preconditioner may be advantageous from a point of preventing recomputation. However, many preconditioners are given only in operator form, so although storing them may be possible, it is not as straightforward as storing the original matrix.

6 Conclusion

We have argued the need for a numerical metadata standard in a world where software components originating from different sources need to interact in a composite application. We have proposed both a standard format for metadata, as well as initial categories to fill in this format. However, our format is highly extendable to allow for custom metadata categories.

Our proposed standard is formalized in an XML file format and a programming API. We have written an XML schema to validate the files, as well as an XSL style sheet for display of the metadata as HTML. The

API has been realised in two libraries, NMD and AnaMod, that we have written. The full software package is available from <http://sourceforge.net/projects/salsa/>.

References

- [1] D.C. Arnold, S. Blackford, J. Dongarra, V. Eijkhout, and T. Xu. Seamless access to adaptive solver algorithms. In M. Bubak, J. Moscinski, and M. Noga, editors, *SGI Users' Conference*, pages 23–30. Academic Computer Center CYFRONET, October 2000.
- [2] Satish Balay, William D. Gropp, Lois Curfman McInnes, and Barry F. Smith. Efficient management of parallelism in object oriented numerical software libraries. In E. Arge, A. M. Bruaset, and H. P. Langtangen, editors, *Modern Software Tools in Scientific Computing*, pages 163–202. Birkhauser Press, 1997.
- [3] Satish Balay, William D. Gropp, Lois Curfman McInnes, and Barry F. Smith. PETSc home page. <http://www.mcs.anl.gov/petsc>, 1999.
- [4] A. R. Conn, N. I. M. Gould, and Ph. L. Toint. *LANCELOT: a Fortran package for Large-scale Nonlinear Optimization (Release A)*. Springer Series in Computational Mathematics. Springer Verlag, Heidelberg, Berlin, New York, 1992.
- [5] T. Davis. University of florida sparse matrix collection. <http://www.cise.ufl.edu/research/sparse/matrices>, <ftp://ftp.cise.ufl.edu/pub/faculty/davis/matrices>, described in NA Digest, vol. 92, no. 42, October 16, 1994, NA Digest, vol. 96, no. 28, July 23, 1996, and NA Digest, vol. 97, no. 23, June 7, 1997.
- [6] Jack Dongarra and Victor Eijkhout. Self-adapting numerical software and automatic tuning of heuristics. In *Proceedings of the International Conference on Computational Science, June 2–4, 2003, St. Petersburg (Russia) and Melbourne (Australia), Lecture Notes in Computer Science 2660*, pages 759–770. Springer Verlag, 2003.
- [7] Jack Dongarra and Victor Eijkhout. Self-adapting numerical software for next generation applications. *Int. J. High Perf. Comput. Appl.*, 17:125–131, 2003. also Lapack Working Note 157, ICL-UT-02-07.
- [8] I.S. Duff, R.G. Grimes, and J.G. Lewis. Sparse matrix test problems. *ACM Trans. Math. Software*, 15:1–14, 1989.
- [9] Thomas Eidson, Jack Dongarra, and Victor Eijkhout. Applying aspect-orient programming concepts to a component-based programming model. In *Proceedings of the 17th International Parallel and Distributed Processing Symposium (IPDPS) April 22–26, 2003, Nice, France*, 2003.
- [10] Erika Fuentes. Users' guide to the numerical metadata library. Technical report, ICL, University of Tennessee, 2003.
- [11] A. Greenbaum. Generalizations of the field of values useful in the study of polynomial functions of a matrix. *Lin. Alg. Appl.*, 347:233–249, 2002.
- [12] <http://math.nist.gov/MatrixMarket/formats.html#hb>.
- [13] J. Langou. *Iterative methods for solving linear systems with multiple right-hand sides*. Ph.D. dissertation, INSA Toulouse, June 2003. CERFACS TH/PA/03/24.
- [14] <http://math.nist.gov/MatrixMarket>.
- [15] <http://math.nist.gov/MatrixMarket/formats.html#mm>.
- [16] Yousef Saad and Martin H. Schultz. GMRes: a generalized minimal residual algorithm for solving nonsymmetric linear systems. *SIAM J. Sci. Stat. Comput.*, 7:856–869, 1986.
- [17] Lloyd N. Trefethen. Pseudospectra of linear operators. *SIAM Review*, 39:383–406, 1997.
- [18] <http://www.w3.org/TR/REC-xml>.
- [19] <http://www.w3.org/TR/xmlschema-formal/>.
- [20] <http://www.w3.org/TR/xsl/>.