The Numerical Algorithms Group

Combining mathematics and technology for enhanced performance

Numerical Algorithms for Posterity

Brian Ford, Director of NAG Limited and

Mike Dewar, Senior Technical Consultant

Over 30 years of mathematical excellence

NAG Numerical Libraries

- §Re-usable, application-neutral code
- §Each piece of algorithmic code supported by:
  - §examples
  - §test material
  - §user-level documentation
  - §programmer-level documentation
- §Different language implementations of same algorithm
  - §Ada, Algol-68, C, Fortran-77, Fortran-90, Pascal, …

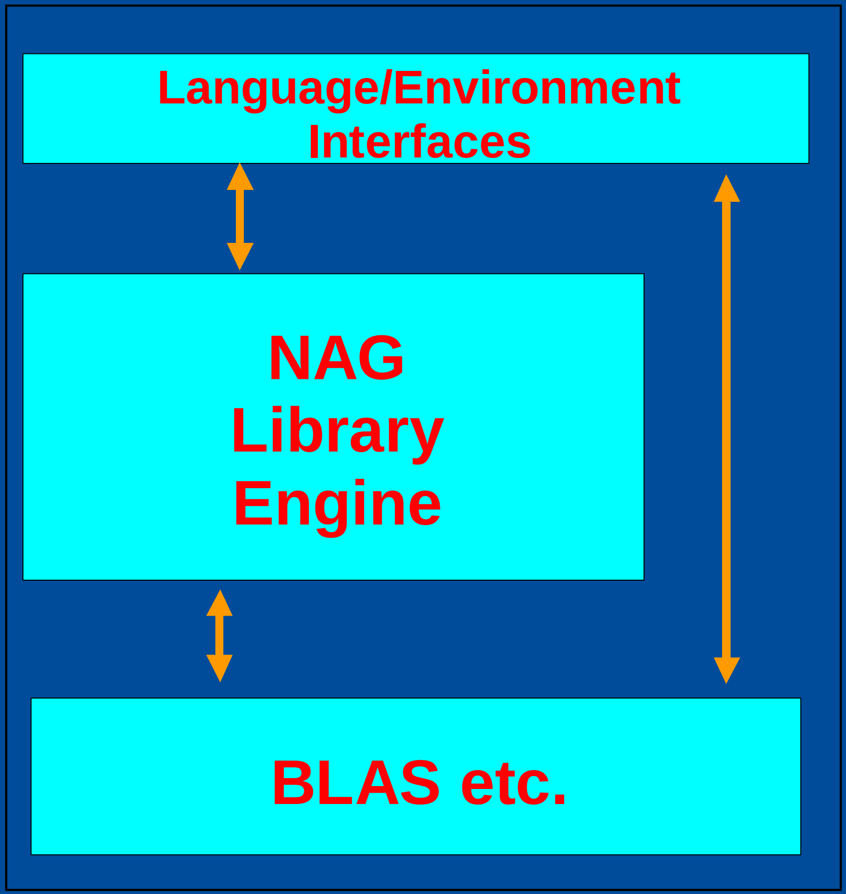
Advantages

- Interfaces exploit strengths of host language
  - Fortran-77: arrays, no dynamic memory allocation
  - C: arrays, structs, dynamic memory allocation
  - Fortran-90: arrays, derived types, generic interfaces, dynamic memory allocation
  - …
- Documentation and examples oriented towards host language

Disadvantages

- Developers waste time re-inventing wheels
- Support effort spread across multiple versions
  - testing, bug-fixing, documentation etc.
- Not all languages can make efficient use of performance-enhancing technology such as BLAS, vendor maths libraries etc.
  - e.g. BLAS 2-D arrays in column order so C programmes must transpose before calling them
- Not all languages optimise well for numerical computation

The New NAG Library Architecture



Software:

- Language-independent algorithmic core
- Interfaces customised to host environment
- Ability to make efficient use of BLAS etc.

Documentation:

- Customised to host environment

QA Material:

- Separate testing of different components

The NAG Library Engine

- Base material written in extended Fortran-77
  - Simple language which is relatively easy to
    - interface with other languages
    - transform into other languages
    - process with software tools
  - Highly optimisable, very efficient for numerical computation
  - Extensions allow for dynamic memory allocation
    - mechanism works with all modern Fortran compilers
  - Substantial body of existing, tested material which can be adopted (e.g. NAG Fortran Library Mark 20)

The NAG Library Engine contd.

- Engine code can be automatically translated to other languages as necessary
  - to remove Fortran run-time dependencies
  - in principle, to provide platform independent code (Java, C#)
- Clear software guidelines for authors
  - Enforced by tools and by peer review
  - Emphasis on clarity and efficiency of code
    - e.g. dynamic memory allocation only used in engine when it is more efficient
- Routine interfaces designed to simplify interfacing to other languages
- *While the engine is derived from the old NAG Library, it is fundamentally different!*

Customising the Engine for a Language or Environment

- For each routine we need to create
  - an interface or wrapper
  - end-user documentation
  - examples
- As far as possible, this is an automatic process
- Wrappers are non-trivial. In general they:
  - transform between interface data-structures and engine data-structures
  - ensure that all arrays are in engine-order (column-major to allow for efficient use of BLAS)
  - allocate and de-allocate memory for workspace (and sometimes output parameters)
  - check constraints on all parameters they use
  - construct error messages to be returned to the user

Simple Example: Engine, Fortran-77 & C

```
SUBROUTINE C06PFFN(IEMODE,DIRECT,NDIM,L,ND,N,X,WORK,LWORK,ERRBUF,IFAIL)
     INTEGER              IEMODE, IFAIL, L, LWORK, N, NDIM
     CHARACTER            DIRECT
     CHARACTER*200        ERRBUF
     COMPLEX *16          WORK(LWORK), X(N)
     INTEGER              ND(NDIM)

SUBROUTINE C06PFF(DIRECT,NDIM,L,ND,N,X,WORK,LWORK,IFAIL)
     INTEGER             IFAIL, L, LWORK, N, NDIM
     CHARACTER           DIRECT
     COMPLEX *16         WORK(LWORK), X(N)
     INTEGER             ND(NDIM)

void c06pfc(Nag_TransformDirection direct, Integer ndim, Integer l,
            const Integer nd[], Integer n, Complex x[], NagError *fail)
```

Engine Interface

- IEMODE
  - determines whether the routine is being asked to check constraints and calculate workspace required or to perform the algorithm
- ERRBUF
  - is used to return diagnostic information (type of error, point where algorithm failed etc.)
- Helper Routines
  - are used to mediate between user-supplied call-back functions and the engine

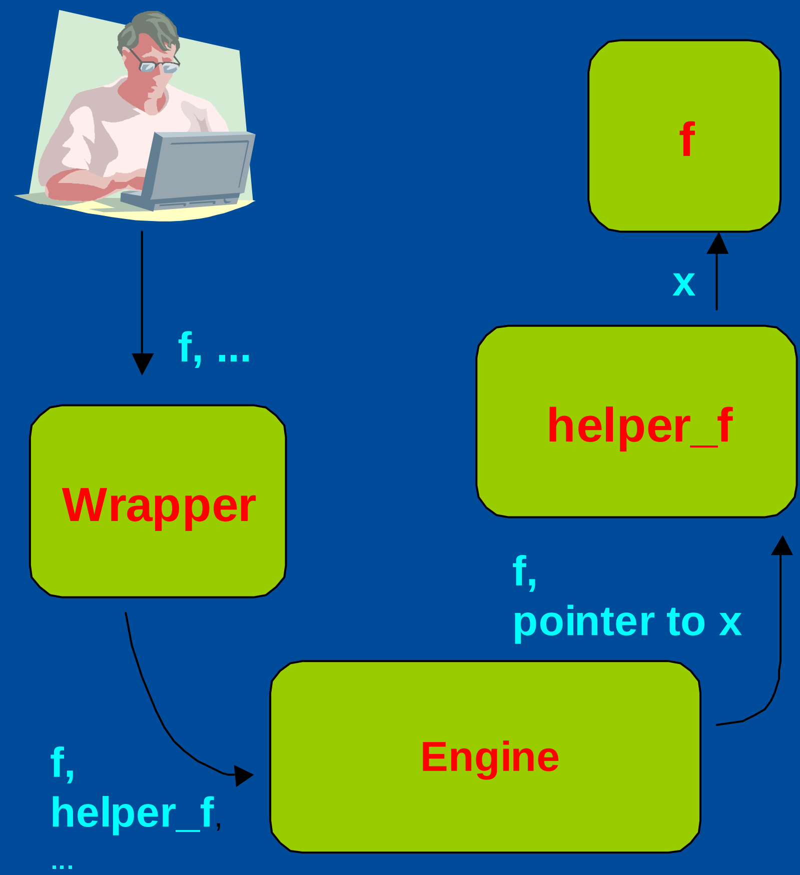
Call-backs

- Algorithms often ask the user to supply part of the problem data as a subroutine
- In our architecture this will naturally follow the semantics of the host environment and cannot necessarily be called directly from the engine, which uses Fortran-77 semantics
- A helper function is provided in the wrapper to mediate between the two
- For example, C naturally uses call-by-value and Fortran uses call-by-reference

Example of call-back and helper

```
double f(double x) {
  return(x^2);
}

double helper_f(
  double(*f)(double),
  double *x) {
 /* De-reference pointer */
  return f(*x);
}
```



NAG Routine Specifications

- Source from which wrappers and documentation (and eventually examples) are generated
- XML documents which include
  - user documentation
  - abstract specifications of each routine parameter
  - information about array dimensions, constraints, parameter dependencies …
  - information about error conditions including type of error, text to be displayed when error occurs etc.
- All text is environment-neutral
- Parameters which are created in the wrappers are distinguished from engine parameters
  - relationships between interface parameters and engine parameters can be specified

Parameter Descriptions

§ Every parameter may have the following information associated with it:

- § name (generic, language specific)
- § concrete engine type (integer, real, …)
- § abstract type (structured matrix, enumeration …)
- § intent (in, out, …)
- § purpose (data, algorithm, workspace, dimension …)
- § relationship to other parameters (*leading dimension of M*)
- § optional or required
- § constraints/recommended values
- § generic documentation
- § environment-specific documentation

Documentation

- Generic documentation can be processed to produce environment-specific version.
- Example: Fortran

X(M) – ***real*** array *Input*

On entry: X(i) must be set to the value of x_i, for i=1,2,…,m. The X(i) need not be ordered.

Constraint: $X_{MIN} \leq X(i) \leq X_{MAX}$, and the X(i) must be distinct.

- Example: C

x[**m**] – const double *Input*

On entry: **x**[i-1] must be set to the value of x_i, for i=1,2,…,m. The **x**[i] need not be ordered.

Constraint: $X_{MIN} \leq \mathbf{x}[\mathbf{i}] \leq X_{MAX}$, and the **x**[**i**] must be distinct.

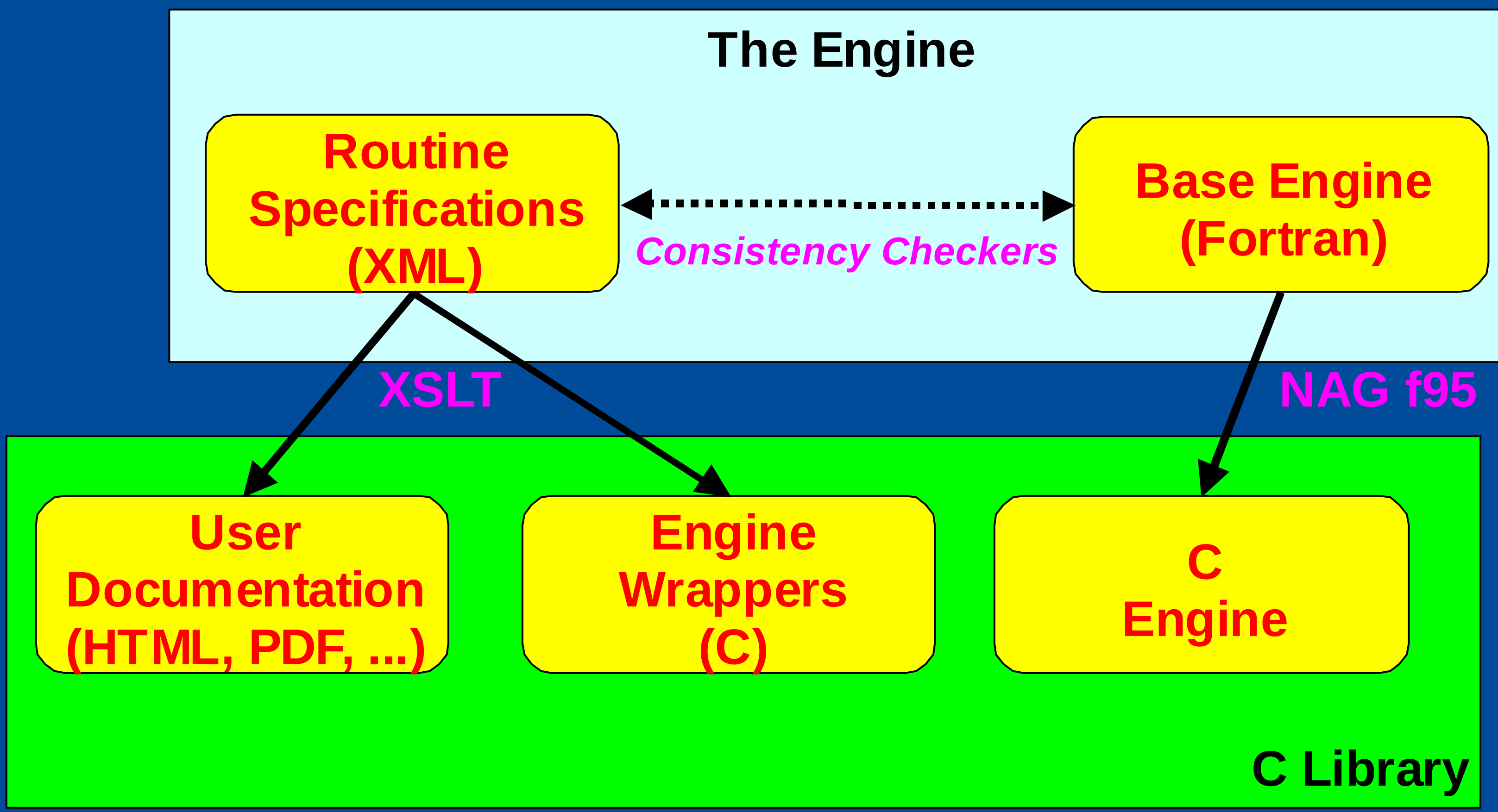
Example 1: NAG C Library Mark 7

- The first product to be based on the engine
- Over 400 fully-documented new routines introduced
- Interface customisations include
  - Use of existing NAG C structs and enumerated types
  - Introduction of new structs and enumerated types where necessary
  - Arrays may be provided in either row- or column-major order
  - Elimination of workspace arrays
  - Elimination of flags which determine whether an optional parameter is being used (in C they are set to NULL)
  - Error messages using C terminology
  - …

Production Process

- Engine delivered to users as C code
  - generated using NAG's f95 compiler technology
  - no dependencies on third-party compiler run-times
- Wrappers and documentation generated using XSLT
  - declarative information stored in specification files so can be re-used in other products
  - transformation to C material encoded as style sheets
- Test and QA material currently written by hand, although some tools were written to semi-automate the process

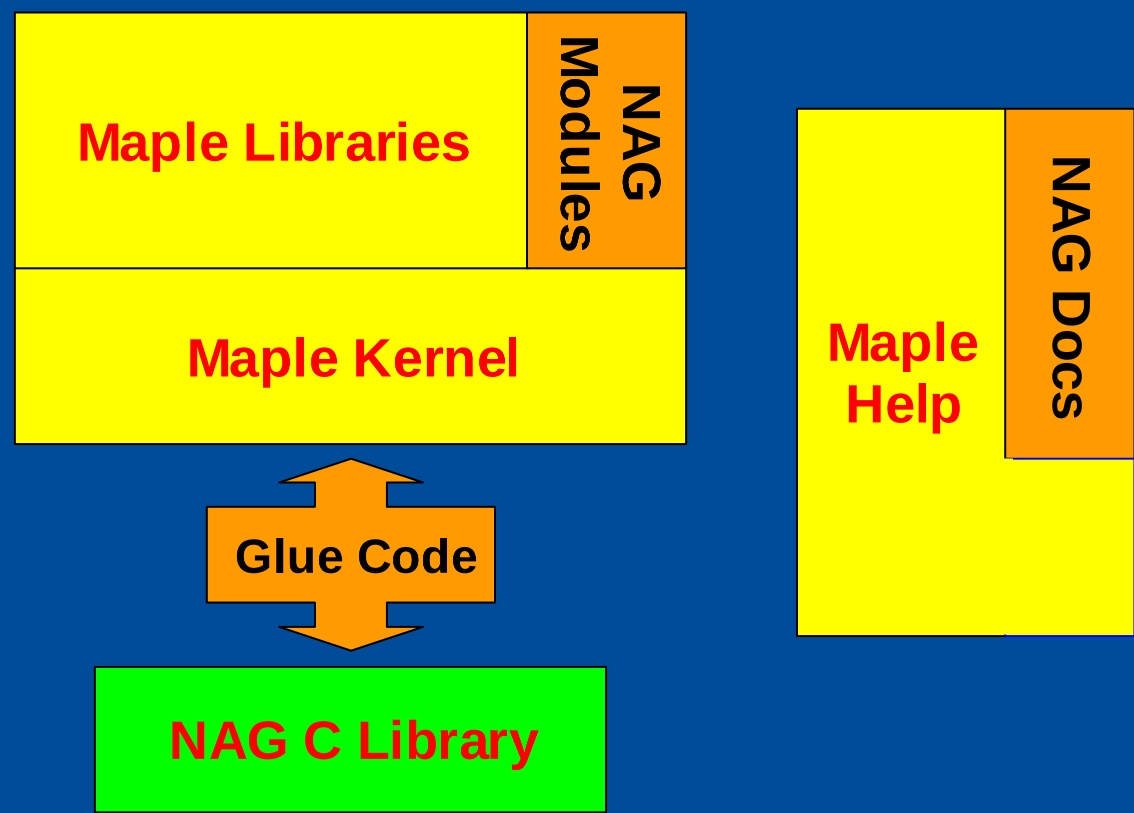
Creating The NAG C Library



Example 2: The NAG-Maple Connector Product

- Overview:
  - The Connector Product is a bridge between two existing products
  - Allow access to all NAG C routines from within Maple
  - Support prototyping
    - develop program calling NAG inside Maple
    - generate stand-alone C code for use outside Maple
  - Document all routines inside the Maple environment using Maple conventions and terminology

Connector Product Architecture



Conclusions

- New architecture is extremely flexible
  - major productivity gains for full-scale products such as C Library Mark 7
  - ability to produce lightweight interfaces such as Visual Basic very rapidly
  - opportunity to embed NAG products in other environments (such as Maple)
- Continued focus on correctness and efficiency
- Simplified, more effective maintenance
- A better service for our users