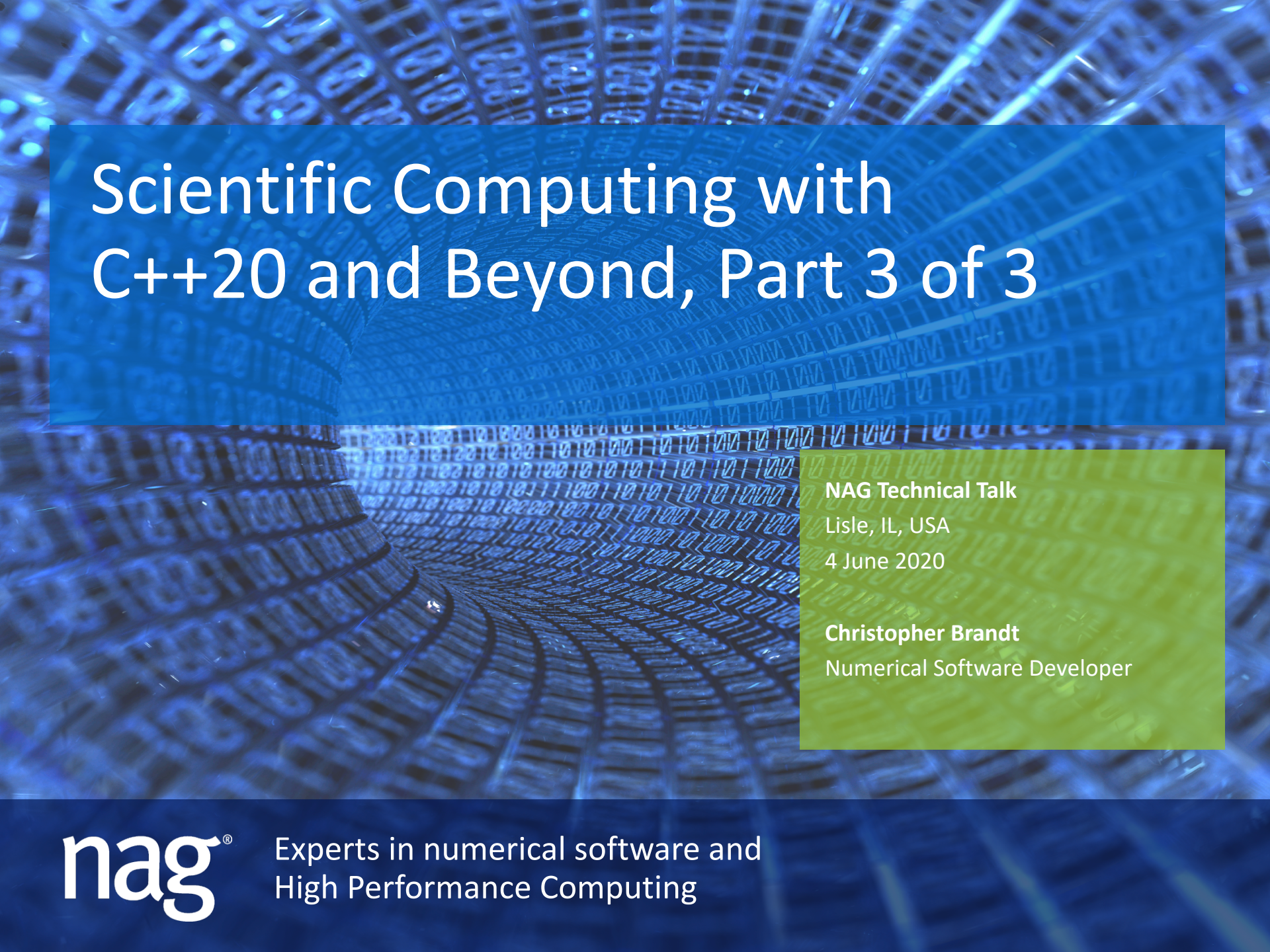


Scientific Computing with C++20 and Beyond, Part 3 of 3

NAG Technical Talk
Lisle, IL, USA
4 June 2020

Christopher Brandt
Numerical Software Developer

nag®

Experts in numerical software and
High Performance Computing

Order of Topics

- Part 1
  - Generic programming
  - concepts
  - Reference types
  - span
- Part 2
  - mdspan
  - mdarray
- Part 3
  - blas
  - linear_algebra

Background

- Purpose of standardization is to codify existing practice for general application
- Attempts to standardize linear algebra within C++ date back to (at least) 2006
- C++ community has strong feelings, both for and against, including linear algebra within the Standard

Historical Lessons

- Review of existing linear algebra libraries
  - C++ libraries
  - Fortran libraries
  - MATLAB
  - Python libraries
- Interviews and conversations with authors and implementors
  - Chris Luchini (POOMA)
  - Nasos Iliopoulos, John Michopoulos (Boost::uBlas)
  - Jack Dongarra (BLAS, LAPACK, etc.)
  - Cleve Moler (MATLAB)
  - Matthieu Brucher (scikit-learn)

Lessons Learned from C++ Linear Algebra Libraries

- Object-oriented numerics / POOMA (early 1990s)
  - Easier to use and more reusable
  - First application of expression templates to improve performance and enabling interfaces that look like mathematical notation
  - Naïve abstraction can hurt performance
- Blitz++, Eigen
  - Expression templates can improve performance
  - Engines can provide implementation polymorphism
- Trilinos
  - Generic iteration over multidimensional sequences

Lessons Learned from Fortran Libraries

- LINPACK, EISPACK, and LAPACK
  - Write algorithms generic on the matrix element type
- BLAS
  - Draw the line between libraries based on developer expertise
  - Enable numerical linear algebra experts to think about algorithms at a high level (e.g., blocking, recursion, etc.)
- High-Performance Fortran
  - Make expensive operations explicit

Lessons Learned from MATLAB

- Familiar to both students and practitioners
- Users appreciate concise syntax and a low barrier to entry
- Interfaces meant for interactive use may need a different design than interfaces in a non-interactive C++ Standard Library

Lessons Learned from Python Libraries

- Users find Numpy's Fortran/MATLAB-style slice notation and its integration with many provided numerical algorithms attractive
- Many potential users of a C++ linear algebra library would be familiar with Python and Numpy
- Python, as a general purpose programming language, enables easy Numpy integration with a wide range of user applications (e.g., databases, websites, etc.)

Lessons Learned from Other Standardization Efforts

- BatchedBLAS
  - Standardization effort began circa 2016, and continues with regular workshops and publications
  - The interface of batched linear algebra operations matters a lot for performance, but may constrain generality
- GraphBLAS
  - It is both possible and performant to express many graph algorithms using a small number of linear algebra primitives
  - Ability to permit both custom arithmetic types and custom operations

Conclusions

- Provide low-level facilities, from which a higher-level interface may be built
- Establish core set of functionality to serve as foundation for future functionality
- Compatible with custom element types
- Ability to configure execution model for target architecture (eventually)

A free function dense linear algebra interface based on the BLAS (C++23)*

`blas`

blas – Overview

- Create an idomatic C++ linear algebra library based on the reference BLAS
- Existing C or Fortran BLAS library is not required
- Begin with core BLAS dense linear algebra functionality, then deviate only as much as necessary
  - Generic on element type
  - Tag classes
  - Free functions for scalar coefficients
  - Mixed precision
  - Optional execution policy arguments

blas – xAXPY Specification and Functionality

```
! Computes:  y <- alpha*x + y

Subroutine SAXPY(n, alpha, x, incx, y, incy)
Subroutine DAXPY(n, alpha, x, incx, y, incy)
Subroutine CAXPY(n, alpha, x, incx, y, incy)
Subroutine ZAXPY(n, alpha, x, incx, y, incy)


! Where:

!   Integer, Intent(In)      :: n, incx, incy
!   S|D|C|G, Intent(In)      :: alpha, x(*)
!   S|D|C|G, Intent(InOut)   :: y(*)
```

blas – linalg_add Overload Set

```
// Computes:  z = x + y

template <class in_object_1_t,
          class in_object_2_t,
          class out_object_t>
void linalg_add(in_object_1_t  x,
                in_object_2_t  y,
                out_object_t   z);

template <class ExecutionPolicy,
          class in_object_1_t,
          class in_object_2_t,
          class out_object_t>
void linalg_add(ExecutionPolicy&&  exec,
                in_object_1_t      x,
                in_object_2_t      y,
                out_object_t       z);
```

blas – linalg_add Examples

```
// Input parameters
std::mdspan<float,  6> x = ...
std::mdspan<float,  6> y = ...
std::mdspan<float,  6> w = ...
std::mdspan<double, 6> z = ...

float alpha = ...
float beta  = ...



// y = alpha*x + y
linalg_add( scaled_view(alpha, x), y, y );

// w = alpha*x + y
linalg_add( scaled_view(alpha, x), y, w );

// z = alpha*x + beta*y
linalg_add( scaled_view(alpha, x), scaled_view(beta, y), z );
```

blas – Additional Facilities

- Free functions
  - scaled_view
  - conjugate_view
  - transpose_view
  - conjugate_transpose_view
- Used only within function call to modify the type (behavior) of the mdspan object
- Uses an expression template to return an “augmented” mdspan object yielding the desired functionality

blas – Tag Classes

- Triangle
  - lower_triangle
  - upper_triangle
- Diagonal
  - implicit_unit_diagonal
  - explicit_diagonal

blas – Execution Policy

- Standardization effort dates back to at least 2012
- Incredibly ambitious effort to provide generic facilities to access all types of computer hardware
- Simplified version exists with C++17 parallel algorithms (CPU only)
- Currently targeted for C++23, though C++26 may be more realistic

blas – Argument Aliasing and Zero Scalars

- The BLAS uses INTENT(INOUT) arguments to express “updates” to a vector or matrix
- The BLAS users the values of scalar multiplier argument (“alpha” or “beta”) to decide whether to treat vectors or matrices as write only
- Translating INTENT(INOUT) to a C++ idiom
  - For certain routines, provide both in-place and not-in-place overloads, where only not-in-place takes an optional ExecutionPolicy argument
  - Else, if the BLAS function conditionally updates, retain read-write
  - Else, if the BLAS function uses a scalar beta argument to decide write vs. read-write, provide an additional overload

blas – xGEMV Specification and Functionality

```
! Computes:  y <- alpha*A*x + beta*y
!       Or:  y <- alpha*transpose(A)*x + beta*y
!       Or:  y <- alpha*conjugate_transpose(A)*x + beta*y

Subroutine SGEMV(trans, m, n, alpha, a, lda, x, incx, beta, y, incy)
Subroutine DGEMV(trans, m, n, alpha, a, lda, x, incx, beta, y, incy)
Subroutine CGEMV(trans, m, n, alpha, a, lda, x, incx, beta, y, incy)
Subroutine ZGEMV(trans, m, n, alpha, a, lda, x, incx, beta, y, incy)


! Where:

!   Integer, Intent(In)      :: m, n, lda, incx, incy
!   S|D|C|G, Intent(In)      :: alpha, a(lda,*), x(*), beta
!   S|D|C|G, Intent(InOut)   :: y(*)
!   Character(1), Intent(In) :: trans
```

blas – matrix_vector_product Overwriting Overload

```
// Computes:  y = A*x

template <class in_vector_t,
          class in_matrix_t,
          class out_vector_t>
void matrix_vector_product(in_matrix_t   A,
                           in_vector_t   x,
                           out_vector_t  y);


template <class ExecutionPolicy
          class in_vector_t,
          class in_matrix_t,
          class out_vector_t>
void matrix_vector_product(ExecutionPolicy&&  exec,
                           in_matrix_t        A,
                           in_vector_t        x,
                           out_vector_t       y);
```

blas – matrix_vector_product Updating Overload

```
// Computes:  z = A*x + y

template <class in_vector_1_t,
          class in_matrix_t,
          class in_vector_2_t,
          class out_vector_t>
void matrix_vector_product(in_matrix_t    A,
                           in_vector_1_t  x,
                           in_vector_2_t  y
                           out_vector_t   z)

template <class ExecutionPolicy
          class in_vector_1_t,
          class in_matrix_t,
          class in_vector_2_t,
          class out_vector_t>
void matrix_vector_product(ExecutionPolicy&&  exec,
                           in_matrix_t        A,
                           in_vector_1_t      x,
                           in_vector_2_t      y
                           out_vector_t       z);
```

blas – matrix_vector_product Examples

```
// Input parameters
std::mdspan<double, 6, 6> A = ...
std::mdspan<double, 6>    x = ...
std::mdspan<double, 6>    y = ...
std::mdspan<double, 6>    w = ...
std::mdspan<double, 6>    z = ...

double alpha = ...
double beta  = ...


// y = alpha*A*x
matrix_vector_product( scaled_view(alpha, A), x, y );

// w = alpha*transpose(A)*x
matrix_vector_product( scaled_view(alpha, transpose_view(A)), x, w );

// y = alpha*A*x + beta*y
matrix_vector_product( scaled_view(alpha, A), x, scaled_view(beta, y), y );

// z = alpha*A*x + beta*y
matrix_vector_product( scaled_view(alpha, A), x, scaled_view(beta, y), z );
```

blas – Future Work

- Current roadmap
  - Arithmetic operators (P1385 linear_algebra proposal)
  - Overloads for basic_mdarray
  - Support for any container that implements the get_mdspan method
  - Batched linear algebra overloads
- Anticipated
  - LAPACK and related functionality
  - Banded matrix layouts
  - Tensors
  - Many others

blas – Implementation

- Reference Implementation
  - https://github.com/kokkos/stdBLAS
  - Build and install via CMake
  - Compatible with C++14 or higher (GCC 9.2, Clang 10.0)
  - Still evolving
    - R1 completed in June 2019
    - R2 completed in January 2020
    - R3 currently underway

blas – Reference

- C++ Standards Committee Papers
  - A Free Function Linear Algebra Interface Based on the BLAS (P1673R2)
  - Evolving a Standard C++ Linear Algebra Library from the BLAS (P1674R0)
  - Historical Lessons for C++ Linear Algebra Library Standardization (P1417R0)
  - A Unified Executors Proposal for C++ (P0443R13)
- Additional Reading
  - mdspan in C++: A Case Study in the Integration of Performance Portable Features Into International Language Standards (SC19)

Linear algebra support for the C++ Standard Library (C++23)*

`linear_algebra`

linear_algebra – Motivation

- Provide a set of types for representing the mathematical objects and operations relevant to linear algebra
- Maintain the functionality and expressiveness and mathematical notation
- Exhibit out-of-the-box performance in the neighborhood of LAPACK, Blaze, Eigen, etc.
- Provide facilities for customization that enable optimizations for a specific problem domain on a specific hardware

linear_algebra – Example

```
// Dynamically-allocated vector objects
std::math::dyn_vector<double> x{ 1.0, 2.0 };
std::math::dyn_vector<double> y{ 2.0, 2.0 };


// Fixed-size matrix objects
std::math::fs_matrix<double, 2, 2> A{ 1.0, 2.0, 3.0, 4.0 };
std::math::fs_matrix<double, 2, 2> B{ 1.0, 1.0, 1.0, 1.0 };


auto result = 1.5 * x * (A.t() + B) * 0.75 * y; // = 49.5
```

linear_algebra – Design Overview

- New `std::math` namespace
- MathObject Types
  - `std::math::vector<EngineType, OperationTraits>`
  - `std::math::matrix<EngineType, OperationTraits>`
- Engine Types
  - `std::math::fs_vector_engine`
  - `std::math::fs_matrix_engine`
  - `std::math::dr_vector_engine`
  - `std::math::dr_matrix_engine`
  - `std::math::vector_view_engine`
  - `std::math::matrix_view_engine`

linear_algebra – Design Overview

- Operation Traits
  - Element promotion traits
    - Element negation traits
    - Element addition traits
    - Element subtraction traits
    - Element multiplication traits
  - Engine promotion traits
    - Engine negation traits
    - Engine addition traits
    - Engine subtraction traits
    - Engine multiplication traits
  - Arithmetic operation traits
    - Negation traits
    - Addition traits
    - Subtraction traits
    - Multiplication traits

linear_algebra – Operations

- Negation
  - -vector
  - -matrix
- Addition
  - vector + vector
  - matrix + matrix
- Subtraction
  - vector - vector
  - matrix - matrix

linear_algebra – Operations

- Scalar-vector multiplication
  - vector * scalar
  - scalar * vector
- Scalar-matrix multiplication
  - matrix * scalar
  - scalar * matrix
- Vector-matrix multiplication
  - vector * matrix
  - matrix * vector

linear_algebra – Operations

- Vector * vector multiplication
  - operator* implements the inner product
  - Outer product implemented as the free function outer_product
- Matrix * matrix multiplication

linear_algebra – Construction

```
// Using type alias
std::math::fs_vector<double, 5>    v2;
std::math::dyn_vector<double>      v1(5);

std::math::fs_matrix<double, 5, 5> m1;
std::math::dyn_vector<double>      m2(5, 5);



// Low level interface
std::math::vector<std::math::fs_vector_engine<double, 5>,
                  std::math::matrix_operation_traits>  v3;

std::math::vector<std::math::dr_vector_engine<double, std::allocator<double>>,
                  std::math::matrix_operation_traits>  v4(5);

std::math::matrix<std::math::fs_matrix_engine<double, 5, 5>,
                  std::math::matrix_operation_traits>  m3;

std::math::matrix<std::math::dr_matrix_engine<double, std::allocator<double>>,
                  std::math::matrix_operation_traits>  m4(5, 5);
```

linear_algebra – Type Resolution

```
std::math::fs_vector<float,  5> v1;
std::math::fs_vector<double, 5> v2;

auto v = v1 + v2;

// v is type: vector<fs_vector_engine<double, 5>, matrix_operation_traits>



std::math::fs_matrix<double, 5, 5> m1;
std::math::dyn_matrix<double>      m2(5, 5);

auto m = m1 + m2;

// m is type: matrix<dr_matrix_engine<double, allocator<double>,
//                   matrix_operation_traits>
```

linear_algebra – Customization Points

- Non-Standard Library element types
  - E.g., fixed-point arithmetic
- Element promotion
  - E.g., adding two float elements to yield a double
- New engine types
- Arithmetic operations
  - E.g., multiple cores, distributed computations

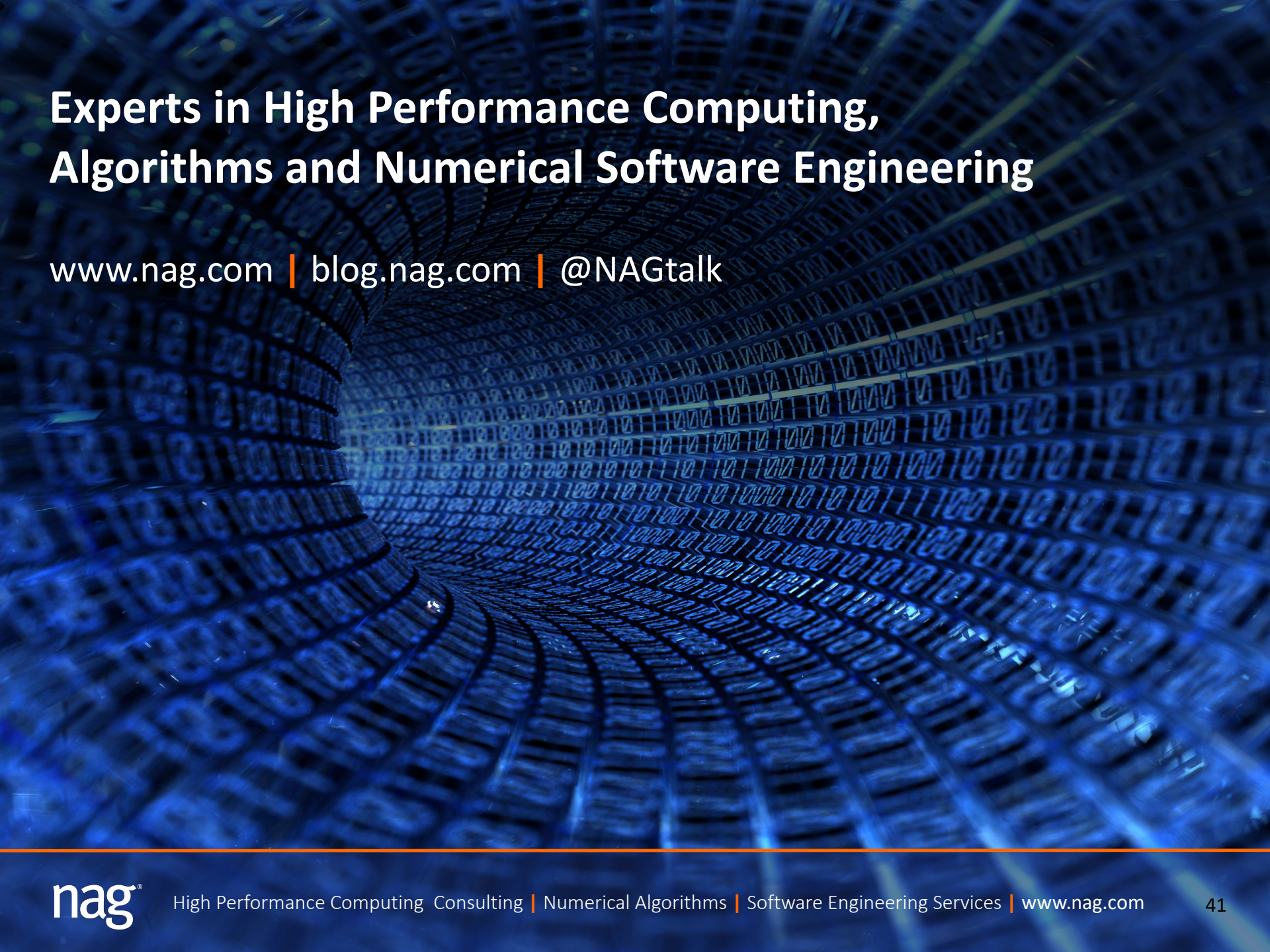
linear_algebra – Implementation

- Reference Implementation
  - https://github.com/BobSteagall/wg21
  - Build and install via CMake
  - Compatible with C++17 (GCC 9.2, Clang 10.0)
  - Still evolving
    - R5 completed in January 2020
    - R6 completed in March 2020
    - R7 currently underway

linear_algebra – Reference

- C++ Standards Committee Papers
  - What do we need from a linear algebra library (P1166R0)
  - A Proposal to Add Linear Algebra Support to the C++ Standard Library (P1385R6)
- Conference Talks
  - Standardising a Linear Algebra Library, Meeting Cpp 2019, Guy Davidson
  - Linear Algebra for the Standard C++ Library, C++Now 2019, Bob Steagall

Questions?


Experts in High Performance Computing,
Algorithms and Numerical Software Engineering
www.nag.com | blog.nag.com | @NAGtalk