Numerical Methods

C. David Sherrill
School of Chemistry and Biochemistry
School of Computational Science and Engineering

Representation of Integers on Computers

0	**0**	**0**	**0**	**0**	**0**	**0**	**0**	0
0	**0**	**0**	**0**	**0**	**0**	**0**	**1**	$2^0 = 1$
0	**0**	**0**	**0**	**0**	**0**	**1**	**0**	$2^1 = 2$
0	**0**	**0**	**0**	**0**	**1**	**0**	**0**	$2^2 = 4$
1	**0**	**0**	**0**	**0**	**0**	**0**	**0**	$2^7 = 128$
1	**1**	**1**	**1**	**1**	**1**	**1**	**1**	$2^8 - 1 = 255$

Representation of Integers on Computers

- Most modern machines represent numbers with 32 or 64 bits at a time
- A 32-bit unsigned integer can hold numbers as large as $2^{32}-1$ = 4,294,496,295
- A signed 32-bit integer (default integer in C++) can hold numbers as large as $2^{31}-1$ = 2,147,483,647 (we lose one bit to track the sign) – this can be too small for some scientific computations
- In C++ we can use a 64-bit integer if we need to (type “long int”)
- Recent versions of Python assume all integers are 64-bit

Representation of Floating Point Numbers

- Floating point numbers are represented by
  $(-1)^s \times 1.m \times 2^e$
  where s is the sign, m is the mantissa, and e is the exponent
- 32 bit float: 1 sign bit, 8 exponent bits, 23+1 mantissa bits (23 explicit, one implied by format)
- Note: the 8 bits for the exponent can only handle exponents from -126 to +127 (-127 and +128 are reserved for special numbers)
- 64 bit float ("double precision"): 1 sign bit, 11 exponent bits, 52+1 mantissa bits
- Note: that's 23 or 52 *binary* digits, not decimal digits; the number of significant decimal digits is only about $\log_{10}(2^{24}) = 7$ (32-bit floats) or $\log_{10}(2^{53})=15.9$ (64-bit doubles)

Example: Loss of Precision in 32-bit Floats

```
#include <stdio.h>

int main(void)
{
  float a = 6.022E23;
  printf("a   = %27.2f\n", a);
  float b = 0.1;
  printf("b   = %27.2f\n", b);
  float c = a+b;
  printf("a+b = %27.2f\n", c);
}
```

```
a     = 602200013124147498450944.00
b     =                       0.10
a+b = 602200013124147498450944.00
```

7 good digits in base 10

Partially cleaning things up with 64-bit doubles

```
#include <stdio.h>

int main(void)
{
  double a = 6.022E23;
  printf("a   = %27.2f\n", a);
  double b = 0.1;
  printf("b   = %27.2f\n", b);
  double c = a+b;
  printf("a+b = %27.2f\n", c);
}
```

```
a     = 602200000000000027262976.00
b     =                       0.10
a+b = 602200000000000027262976.00
```

16 good digits in base 10

Numerical Stability

- We can get numerical errors from (a) roundoff error due to the finite number of bits used to represent the values, and (b) errors in the input values
- Some approaches are more numerically stable than others
- For scientific computing, we generally want to use *double precision* (64 bits) rather than *single precision* (32 bits) floating point values (although GPU's entice people to try single precision because it's substantially faster)
- Libraries like LAPACK can sometimes estimate errors in computed quantities (e.g., eigenvalues) and also provide subroutines that are meant to be more numerically robust

Numerical Stability

- Avoid adding very small quantities to very large quantities… numerical precision can be improved by accumulating results into a higher-precision variable (e.g., quadruple precision), or by adding numbers in order of increasing magnitude; also see methods like the Kahan summation algorithm (aka compensated summation)
- Avoid subtracting very similar values … this can cause a loss of precision (sometimes re-casting the mathematics of the problem can avoid this)
- Let x and y be normalized floating-point machine numbers such that $x > y > 0$. If $1 - (y/x) < 2^{-p}$ for some positive integer p, then at least p significant binary digits are lost in the subtraction $x - y$.

Numerical Linear Algebra

Computers are Good at Linear Algebra

$$\begin{bmatrix} 1 & 1 & 0 \\ 2 & 0 & 1 \\ 0 & 3 & 2 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \\ 2 \end{bmatrix} = \begin{bmatrix} 1 \\ 4 \\ 4 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 1 & 0 \\ 2 & 0 & 1 \\ 0 & 3 & 2 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \\ 2 \end{bmatrix} = 1 \begin{bmatrix} 1 \\ 2 \\ 0 \end{bmatrix} + 2 \begin{bmatrix} 0 \\ 1 \\ 2 \end{bmatrix} = \begin{bmatrix} 1 \\ 4 \\ 4 \end{bmatrix}$$

Vector computers used to literally operate on entire vectors of numbers at one time… now superscalar architectures are good at performing the same operation (say, multiplying two numbers and adding the result to an accumulated sum) over and over

From Matrix-Vector Products to Matrix Multiplication

$$\begin{bmatrix} 1 & 1 & 0 \\ 2 & 0 & 1 \\ 0 & 3 & 2 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \\ 2 \end{bmatrix} = \begin{bmatrix} 1 \\ 4 \\ 4 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 1 & 0 \\ 2 & 0 & 1 \\ 0 & 3 & 2 \end{bmatrix} \begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 0 \\ 2 & 0 & 0 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 \\ 4 & 0 & 2 \\ 4 & 3 & 0 \end{bmatrix}$$

Memory Access Patterns are Important for Efficiency

$$\begin{bmatrix} 1 & 1 & 0 \\ 2 & 0 & 1 \\ 0 & 3 & 2 \end{bmatrix} \begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 0 \\ 2 & 0 & 0 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 \\ 4 & 0 & 2 \\ 4 & 3 & 0 \end{bmatrix}$$

1	1	0	2	0	1	0	3	2
1	0	1	0	1	0	2	0	0

Basic Linear Algebra Subprograms (BLAS)

- Many problems in scientific computation involve common linear algebra operations like matrix-vector products, matrix multiplies, etc.
- These are used so often that the BLAS library providing these operations became very popular
- There are various BLAS libraries optimized for different architectures; they conform to the same interface (Fortran and C bindings available) so they can be interchanged
- The reference implementation (e.g., available from netlib.org, or 'yum install blas') is *not optimized* and is *not recommended* --- optimization can make a huge difference in performance
- Call BLAS and let others take care of the hard work of writing a super-efficient matrix multiply or other linear algebra operations!

BLAS Libraries

- Intel Math Kernel Library (MKL): Intel-Optimized BLAS library for Intel processors (including Xeon Phi)
- Automatically Tuned Linear Algebra Software (ATLAS): Tunes itself to be optimized for various architectures (checks for SIMD operations, tests different block sizes)
- OpenBLAS: Optimized BLAS based on the previous GotoBLAS, for x86, x86-64, MIPS, ARM
- ESSL: IBM’s version for PowerPC
- cuBLAS: Optimized for NVIDIA GPUs
- Many others

BLAS Alternatives

- Eigen: Generic C++ template interface for matrix/vector operations
- Elemental: Distributed-memory, can handle sparse matrices
- MAGMA: Heterogeneous and hybrid (CPU/GPU) architectures
- Many others

BLAS Levels

- Level 1 – Vector operations: copy a vector, scale a vector, swap 2 vectors, dot product of 2 vectors, 2-norm of a vector, etc.
  AXPY : $y \leftarrow ax + y$
- Level 2 – Matrix-vector product, outer product of 2 vectors
  GEMV: $y \leftarrow aAx + by$
  $Tx = y$ (T triangular)
- Level 3 – Matrix-matrix operations
  GEMM: $C \leftarrow aAB + bC$
- Versions for single-precision (S), double-precision (D), and complex numbers (C) … this letter is the first letter of each BLAS subroutine name…. e.g., DGEMM is a “double precision generalized matrix multiply”

BLAS Example: DAXPY

$y \leftarrow ax + y$

DAXPY(integer len, double a, double* x, int incx, double* y, int incy);

len = length of vector

a = constant

x = vector

incx = “increment” in x or “stride” of x … use every incx-th element (usually 1)

y = vector

incy = “increment” or “stride” in y … use every incy-th element

BLAS Example DGEMM
C ← alpha*op(A)*op(B) + beta*C

DGEMM(char transa, char transb, int m, int n, int k, double alpha, double *A, int lda, double *B, int ldb, double beta, double *C, int ldc);

If transa = ‘t’ or ‘T’, op(A) = transpose(A)
If transb = ‘t’ or ‘T’, op(B) = transpose(B)
… very handy if the matrix we need to multiply is the transpose of what we have stored!

m = rows of op(A)
n = cols of op(B)
k = “links” … cols of op(A) and rows of op(B)

lda = “leading dimension” of A… lda, ldb, ldc let us do the DGEMM operation on submatrices within larger matrices

C/C++ vs Fortran Confusion

Warning! The original BLAS documentation was written for Fortran. Fortran BLAS libraries can be called from C/C++ with the appropriate "name mangling", but one has to be very wary about Fortran vs C conventions for matrices!

- Fortran: "column major"… Go down each column, one at a time
- C/C++: "row major" … Go down each row, one at a time

Fortunately, there are now also native C interfaces to BLAS libraries (the documentation for which may or may not have been totally and correctly converted from Fortran…)

LAPACK: The **L**inear **A**lgebra **Pack**age

- Analogue of BLAS, but for eigenvalue problems, singular value decomposition, systems of linear equations, and linear least squares problems
- Matrix factorizations including LU, QR, Cholesky, and Schur decomposition
- Like most versions of BLAS, assumes dense matrices
- Utilizes BLAS routines for lower-level operations (so architecture-dependent optimization at the LAPACK layer may be less important)
- ScaLAPACK is a distributed-parallel version
- Actively developed, so new algorithmic advances in linear algebra are incorporated
- Included with MKL and ESSL libraries, and some of the routines are also included in ATLAS

LAPACK: The **L**inear **A**lgebra **Pack**age

- Supports real and complex numbers
- Supports both single- and double-precision floating point numbers
- Naming scheme – subroutines are named in the form XYYZZ(Z), where
  - X: S = Single precision, D = Double precision, C = single precision complex, Z = double precision complex
  - YY (matrix type): DI = diagonal, GB = general band, GE = general, HE = (complex) Hermitian, SP = symmetric with packed storage, SY = symmetric, OR = orthogonal real, PO = symmetric or Hermitian positive definite, …
- Example: DSYEV: obtain the eigenvalues (EV) and (optionally) the eigenvectors of a double-precision (D) symmetric (SY) matrix

DSYEV

- Subroutine dsyev (character JOBZ, character UPLO, integer N,
  double precision, dimension (LDA, *) A, integer LDA,
  double precision, dimension (*) W,
  double precision, dimension (*) WORK,
  integer LWORK, integer INFO)
- JOBZ = ‘N’ : Eigenvalues only; ‘V’ : eigenvalues and eigenvectors
- A = double-precision array of dimension (LDA, N). If UPLO = ‘U’, use the leading N-by-N upper-triangular part of A; if UPLO=‘L’, use the leading N-by-N lower triangular part of A. On exit, if JOBZ = ‘V’ and INFO=0, A contains the eigenvectors
- W = Array to store the eigenvalues
- WORK = Array for temporary storage during work
- LWORK = Size of work array provided (-1 to query optimal size)
- INFO = status flag, =0: successful exit; =-p, p-th argument had illegal value; >0 algorithm failed to converge

C Interface

- As for BLAS, LAPACK also now has a C interface (officially supported by Netlib)
- Prefix Fortran LAPACK subroutine names by LAPACKE_ to call, e.g., LAPACKE_dsyev
- Both BLAS and LAPACK can be compiled using 4-byte or 8-byte integers. If linking a FORTRAN library, need to know which was used in compiling it. From C interface, a datatype lapack_int is used, which can be defined either as an int (4 bytes) or a long int (8 bytes)
- This interface also supported by MKL
- The C interface is at two layers … the higher-layer automatically takes care of creating work arrays, etc.

Availability in Higher-Level Tools

Optimized BLAS and LAPACK libraries can be called through some popular, higher-level tools like MATLAB, NumPy, and R … these often hide some of the complexities and are easier to use, yet they have the speed under the hood

Eigenvalue Problems in Quantum Chemistry

- The most important equation in quantum chemistry, the Schrödinger Equation, is an eigenvalue equation

$$H\psi^i = E^i\psi^i$$

- Each eigenvalue represents the total energy of the system of nuclei and electrons (relative to infinitely separated nuclei and electrons)
- Each eigenvalue represents a “state” of the system (wavefunction)

Basis Representation

- We can turn the general quantum-mechanical problem into a linear algebra problem by introducing a basis
- For a general problem with N electrons, we introduce an appropriate "N-electron" basis, e.g., a basis of Slater determinants $\{ \Phi_k \}$ (and we restrict our choice by "excitation classes")
- We can then represent a wavefunction ψ^i as a column vector giving the components of that wavefunction in the N-electron basis,

$$c^i_k = \langle \Phi_k | \psi^i \rangle$$

- The Hamiltonian operator H is represented as a matrix, with matrix elements

$$H_{kl} = \langle \Phi_k | H | \Phi_l \rangle$$

- This leads to the matrix representation

$$\mathbf{H}\mathbf{c}^i = E^i \mathbf{c}^i$$

Self-Consistent Field Models

- We do similar things for Hartree—Fock and Density Functional Theory, except here our mean-field model means we only need a single Slater determinant
- We still have an eigenvalue problem, because now we need to solve for the orbitals that make up the Slater determinant
- We express each molecular orbital as a linear combination of atomic orbitals, and the expansion coefficients are obtained by diagonalizing the Fock matrix

$$\mathbf{F}\,\mathbf{c}^i = \varepsilon^i\,\mathbf{c}^i$$

$$F_{\mu\nu} = \langle \mu | F | \nu \rangle$$

$$c^i_\mu = \langle \mu | c^i \rangle,\ \phi^i = \Sigma\, c^i_\mu\, | \mu \rangle$$

Solving the Real Problem

- In quantum chemistry, we can need to diagonalize rather large matrices … the Fock matrix is a symmetric square *n* x *n* matrix where *n* is usually a few thousand or less
- The Hamiltonian matrix for the N-electron problem is also a symmetric square *n* x *n* matrix, but now *n* can be billions or more --- we take advantage of the fact that for this matrix, we usually only need the lowest few eigenvalues/eigenvectors and use Krylov subspace methods … but this still involves standard eigenvalue techniques within the space spanned by the trial vectors

A Brief Introduction to the NumPy Module

- Actively-developed, easily accessible, can link to optimized BLAS/LAPACK like MKL
- “NumPy is the fundamental package for scientific computing in Python” --- Official NumPy documentation
- Provides a multidimensional array object and routines to manipulate this object, including basic linear algebra
- Beginning to be used by a large number of other Python modules --- NumPy arrays are an important Python data type now

NumPy Arrays

- Have a fixed size (unlike Python lists) --- attempts to change the size will delete the array and re-create it at a different size
- All elements are of the same type (again unlike Python lists)

```
import numpy as np
a = np.array([1, 2], [3, 4]])
print(a)
```

```
[[1 2]
  [3 4]]
```

Complex Arrays

Complex numbers are also supported

```
import numpy as np
a = np.array([1, 2, 3], dtype = np.complex_)
print(a)
```

```
[ 1.+0.j, 2.+0.j, 3.+0.j]  # note use of j instead of i for sqrt(-1)
```

Supported data types

dtype can take on many possible values, including

- bool_
- intc: identical to C integer (normally 32 or 64 bits)
- intp: “pointer” integer, same as C ssize_t (signed size_t)
- int8, int16, int32, int64 (signed)
- uint8, uint16, uint32, uint64 (unsigned)
- float_ : float64
- float16, float32, float64
- complex64: 32-bit float for real part, 32-bit for imaginary part
- complex128: two 64-bit floats]
- complex_ : short for complex128

Shape

```
import numpy as np
a = np.array([[1,2,3],[4,5,6]])
print(a)
```

```
[[1 2 3]
 [4 5 6]]
```

```
print(a.shape)
```

```
(2,3)
```

```
import numpy as np
a = np.array([[1,2,3],[4,5,6]])
a.reshape(3,2)
print(a)
```

```
[[1 2]
 [3 4]
 [5 6]]
```

Transpose

```
import numpy as np
a = np.array([[1,2,3],[4,5,6]])
print(a)
```

```
[[1 2 3]
 [4 5 6]]
```

```
print(a.T)
```

```
[[ 1 4]
 [ 2 5]
 [ 3 6]]
```

Indexing

```
import numpy as np
a = np.array([[1,2,3],[4,5,6]])
print(a)
```

```
[[1 2 3]
 [4 5 6]]
```

```
print(a[1][0])   # a[1, 0] also works and may be more efficient
```

```
4
```

```
print(a[0][2])
```

```
3
```

Automatic Array Creation

```
import numpy as np
a = np.zeroes((2,2)) # zero matrix
print(a)
```

```
[[0. 0.]
 [ 0.  0.]]
```

```
b = np.eye(2) # identity matrix
print(b)
```

```
[[1. 0.]
 [0. 1.]]
```

```
a = np.random.random((2,2))
print(a)
```

```
[[0.9138  0.0381]
 [ 0.7381 0.5323]]
```

Slicing

```
import numpy as np
a = np.array([[1,2,3,4], [5,6,7,8], [9,10,11,12]])
b = a[:2, 1:3] # grab rows 0 and 1, and columns 1 and 2
print(b)
[[2 3]
  [6 7]]
```

A “slice” gives a different *view* of the original array; changing the slice changes the original array

```
b[0,0] = 99
print(a[0,1])
99
```

Array Math

```
a = np.array([[1, 2],
              [3, 4]])

b = np.array([[0, 1],
              [1, 0]])

print(a+b) # or print(np.add(a,b))
[[1 3]
 [4 4]]

Print(a*b) # or print(np.multiply(a,b))

[[0 2]

 [3 0]] # is this what you expected??
```

Inner Product of Two Vectors

```
a = np.array([1, 2, 3])

b = np.array([0, 1, 1])

print(a.dot(b)) # or print(np.dot(a,b))
5
```

Matrix Multiplication

```
a = np.array([[1, 2],
              [3, 4])

b = np.array([[0, 1],
              [1, 0])

print(a.dot(b)) # or print(np.dot(a,b))
[[2 1]
 [4 3]]
```

Eigenvalues and Eigenvectors with NumPy

```
From numpy import linalg as LA
a = np.array([1, -1],
                    [1, 1]])
w, v = LA.eig(a) # w has eigenvalues, v has eigenvectors as its cols
print w
array([ 1. + 1.j, 1. – 1.j])
print v
array([[ 0.70710+0.j          ,  0.70710+0.j ],
          [  0.00000-0.70710j,  0.00000+0.70710j]])
```