Applied Mathematics 225

Unit 2: Advanced numerical linear algebra

Lecturer: Chris H. Rycroft

Prologue: linking to objects and libraries in C++

As C++ programs grow in length, it becomes less desirable to compile a single monolithic `.cc` file each time.

We aim to structure our program around C++ classes, self-contained functions, *etc.* that we want to re-use without recompiling each time. C++ provides us with a mechanism for doing this.

When compiling a C++ program, the final stage is linking, where the compiler searches for precompiled code to link into the current executable

Example three-file project

These files are contained in the `am225_examples/2a_linking` directory:

- `file_output.cc` – a file that contains the definition of a function called `gnuplot_output` for outputting a 2D array into a standard binary format that can be read by Gnuplot[1]
- `file_output.hh` – contains the declaration of the `gnuplot_output` function, but not its definition
- `gp_test.cc` – an executable program that creates a test 2D array of data and calls the `gnuplot_output` function to save it to a file

[1] Within Gnuplot, type "`help binary matrix nonuniform`" for documentation.

Structure of gp_test.cc

```
#include <cmath>

#include "file_output.hh"

int main() {

    // Code ...

    gnuplot_output("test_out.gnu",fld,m,n,ax,bx,ay,by);

    // Code ...
}
```

The program includes file_output.hh, so the compiler knows about the existence of the gnuplot_output function.

Direct compilation – a linking error

Compiling `gp_test.cc` gives a linking error:

```
macmini:unit2/gp_example% g++ -Wall -o gp_test gp_test.cc
Undefined symbols for architecture x86_64:
  "gnuplot_output(char const*, double*, int, int, double,
      double, double, double)", referenced from:
      _main in gp_test-4c26fb.o
ld: symbol(s) not found for architecture x86_64
clang: error: linker command failed with exit code 1 (use -v
     to see invocation)
```

Compiler does not ever see the definition of `file_output.hh`

Compilation

First compile `file_output.cc` using the `-c` flag:

```
g++ -Wall -c file_output.cc
```

This creates an object file called `file_output.o`. It has the assembled machine code of the `gnuplot_output` function. Then compile the main program:

```
g++ -Wall -o gp_test gp_test.cc file_output.o
```

Successful compilation: compiler links the precompiled `gnuplot_output` into the `gp_test` executable.

Build system

A project could consist of multiple object files and multiple executables.

Object files need to be built prior to linking to executables. Some object files might depend on others.

To simplify compilation we need a build system. We demonstrate the use of GNU Make[2] although there are many others.

A file called "`Makefile`" contains the dependencies between programs. Typing "`make`" on the command line recompiles only the files whose date stamps indicate they are out of date.

[2] https://www.gnu.org/software/make/

A Makefile for the example

```
# Specify compiler and flags
cxx=g++
cflags=-Wall

# What executables should be built
all: gp_test

# Rule to build object file
file_output.o: file_output.cc file_output.hh
	$(cxx) $(cflags) -c file_output.cc

# Rule to build executable
gp_test: gp_test.cc file_output.o
	$(cxx) $(cflags)  -o gp_test gp_test.cc file_output.o
```

From objects to libraries

Many object files can be joined together into a library file with a "`.a`" suffix,[3] using the `ar` command line utility.

For our example:

```
ar rs libfoutput.a file_output.o
```

Here, only on object file is passed as arguments, but usually more would be provided.

Compilation can alternatively link to this file:

```
g++ -Wall -o gp_test gp_test.cc libfoutput.a
```

[3] The "A" stands for archive.

Linking to libraries

We can link to system or third-party libraries in the exact same way. On Mac/Linux systems, many system header files are contained in `/usr/include`, and the precompiled libraries are in `/usr/lib`.[4]

Three relevant compiler flags:

- `-I<dir>` – tell the compiler to look in `<dir>` to resolve any `#include` commands (`e.g.` for header files)
- `-L<dir>` – tell the compiler to look in `<dir>` to look for libraries
- `-l<library>` – link the library `lib<library>.a` to the compiled program, searching in the directories provided by `-L` (plus default system directories)

Alternative compilation command:

```
g++ -Wall -o -L. gp_test gp_test.cc -lfoutput
```

[4] The general principles are the same on Windows.

Numerical Linear Algebra

Problems involving Numerical Linear Algebra are ubiquitous in scientific computing

- Many scientific problems can be expressed as solving linear systems of equations
- When we discretize ODEs and PDEs, we frequently need to solve systems of equations for the discretized function values
- Data analysis requires solving overdetermined linear least squares problems
- Eigenvalue problems occur in many scenarios (*e.g.* resonance, graph analysis)
- Many nonlinear problems are most effectively solved by approximating them with a sequence of linear problems

Topics covered in AM205[5]

- The LU factorization
- The QR decomposition
- Singular Value Decomposition
- Eigenvalue algorithms (power method, Rayleigh quotient, Krylov methods)
- Multigrid method

[5] See AM205 units 2 and 5.

Topics we will cover

We will examine the design of two widely-used and powerful libraries, BLAS and LAPACK, for numerical linear algebra.

We will look at Krylov methods for the solution of linear algebra problems, and the associated issue of preconditioning.

We will look at the Fast Fourier Transform, which can be viewed as a solution method for many matrix problems of interest.

We will look at domain decomposition for parallelizing numerical linear algebra routines.

Book

We will make use of the following textbook, and follow its notation:

- James W. Demmel, *Applied Numerical Linear Algebra*, SIAM, 1997.

Numerical linear algebra is an active area of research, with much interest in efficient parallel methods for supercomputing applications.

One issue is fault tolerance. When running on 10^5 CPUs, probability of failure on one of CPU is high. Aim for algorithms that can deal with this failure.

Some history

Since linear algebra often reduces to model problems (*i.e.* solve $Ax = b$, find the eigenvalues of A) it is well-suited to being solved by libraries.

An early library was LINPACK,[6] used on supercomputers in the 1970s and 1980s. The LINPACK benchmark is still used to test the speed of supercomputers in the TOP500[7] list.

LINPACK was designed before modern memory hierarchies became important for optimal performance. Is has largely been superseded by BLAS (Basic Linear Algebra Subroutines) and LAPACK (Linear Algebra PACKage), which take memory hierarchies into account.

[6] http://www.netlib.org/linpack/

[7] https://www.top500.org

Example

Define t_{arith} as the time to do a floating point operation and t_{mem} as the time to move memory between hierarchy levels. Assume $t_{\text{arith}} \ll t_{\text{mem}}$ on modern systems.

Consider adding two $n \times n$ matrices together. We can't do any better than the following procedure on each matrix entry:

- Read the two numbers from memory ($2t_{\text{mem}}$)
- Add the two numbers (t_{arith})
- Store the result in memory (t_{mem})

Total time is $n^2(3t_{\text{mem}} + t_{\text{arith}})$.

A measure of memory efficiency

Suppose an algorithm requires m memory references and f floating point operations. Then the total running time is

$$ft_{\text{arith}} + mt_{\text{mem}} = ft_{\text{arith}}\left(1 + \frac{m}{f}\frac{t_{\text{mem}}}{t_{\text{arith}}}\right) = ft_{\text{arith}}\left(1 + \frac{1}{q}\frac{t_{\text{mem}}}{t_{\text{arith}}}\right),$$

where q is a measure of memory efficiency. Higher q is better.

Previous matrix addition example had $q = 1/3$.

BLAS (Basic Linear Algebra Subroutines)

To obtain optimal performance, we want to minimize memory access. This will depend on hardware.

BLAS provides functions to perform basic linear algebra operations (*e.g.* dot product, matrix–matrix multiply) that are tuned to the hardware.

Chip vendors provide these. Examples are Intel MKL (Math Kernel Library)[8] and AMD BLIS library.[9]

There is also ATLAS (Automatically Tuned Linear Algebra Software),[10] an open-source library that self-tunes during compilation.

---

[8] https://software.intel.com/en-us/mkl

[9] https://developer.amd.com/amd-cpu-libraries/blas-library/

[10] http://math-atlas.sourceforge.net

Matrix multiplication, $C = C + AB$ (unblocked)

```
1:  for i = 1 : n do
2:      Read row i of A into fast memory
3:      for j = 1 : n do
4:          Read C_ij into fast memory
5:          Read column j of B into fast memory
6:          for k = 1 : n do
7:              C_ij = C_ij + A_ik B_kj
8:          end for
9:          Write C_ij into slow memory
10:     end for
11: end for
```

Operation count

Memory references:

- n^2 operations to read in A once
- n^3 operations to read in B n times
- $2n^2$ operations to read/write C once

Floating point operations

- n^3 multiplications
- n^3 additions

Hence memory efficiency is

$$q = \frac{f}{m} = \frac{2n^3}{n^3 + 3n^2} \approx 2.$$

Matrix multiplication, $C = C + AB$ (blocked)

Divide C into and $N \times N$ block matrix, with blocks C^{ij} of size $(n/N) \times (n/N)$.

```
1:  for i = 1 : N do
2:      for j = 1 : N do
3:          Read C^ij into fast memory
4:          for k = 1 : n do
5:              Read A^ik into fast memory
6:              Read B^kj into fast memory
7:              C^ij = C^ij + A^ik B^kj
8:          end for
9:          Write C^ij into slow memory
10:     end for
11: end for
```

Operation count

Memory references:

- Nn^2 operations to read in A N times
- Nn^2 operations to read in B N times
- $2n^2$ operations to read/write C once

Total is $(2N + 2)n^2$ memory operations. If fast memory (cache) size is M then we require $M \geq 3(n/N)^2$. For optimal performance $N \approx n\sqrt{3/M}$.

$2n^3$ floating point operations as before.

Hence memory efficiency is

$$q = \frac{f}{m} = \frac{2n^3}{2Nn^2} = \frac{n}{N} \approx \sqrt{M/3}.$$

Levels of improvement

- Level 1 BLAS: $q < 1$, *e.g.* matrix addition
- Level 2 BLAS: $q \approx 2$, *e.g.* matrix–vector multiply
- Level 3 BLAS: $q \gg 2$, *e.g.* matrix–matrix multiply

Many other standard algorithms (*e.g.* Gaussian elimination) can be reorganized to achieve Level 3 BLAS.

BLAS levels are a common benchmark for evaluating the memory efficiency of algorithms in research papers.

BLAS matrix–matrix multiply example

Computer demo: timing comparison using BLAS routine.

Aside: a *fast* matrix multiply

The discrete Fourier transform takes complex numbers $x_0, x_1, \ldots, x_{n-1}$ and computes

$$y_k = \sum_{j=0}^{n} x_j e^{-2\pi ijk/n}.$$

Originally thought to require $O(n^2)$ floating point operations. The fast Fourier transform reduces this to $O(n \log n)$ floating point operations.

Unresolved question: is there a fast matrix multiply?

There are only $O(n^2)$ elements in two $n \times n$ matrices, yet our standard algorithm requires $O(n^3)$ floating point operations—is that the best we can do?

Aside: a *fast* matrix multiply

Strassen's algorithm (1969) is a recursive approach to replace multiplying $n \times n$ matrices by seven multiplications of $(n/2) \times (n/2)$ matrices. Complexity is therefore $O(n^{\log_2 7}) = O(n^{2.807})$. Can be used in practical calculations.[11]

Improved by Coppersmith and Winograd in 1990 to $O(n^{2.375477})$. Current best is $O(n^{2.3728639})$ by François Le Gall. Algorithmic prefactors are generally too large to be practical.

Generally thought that an $O(n^2)$ algorithm, perhaps with additional logarithmic factors, is possible.

[11] This will feature on a homework problem.

LAPACK

LAPACK[12] uses BLAS to efficiently perform many linear algebra operations:

- Solving linear systems
- Solving linear least-squares problems
- LU, QR, Cholesky decompositions
- Eigenvalue computations

It has specialized algorithms for banded matrices, symmetric matrices, orthogonal matrices, Hessenberg matrices, *etc.*

[12] http://www.netlib.org/lapack/

Radial basis functions

An example problem in dense numerical linear algebra

Suppose that a function has been sampled at irregular points $\vec{x}_k$ for $k = 1, \ldots, n$. Let f_k be the corresponding function values.

We aim to construct a smooth function approximation that matches the given data points.

Introduce a radial function $\phi(r)$. Define function as

$$f(\vec{x}) = \sum_{k=1}^{N} w_k \phi(||\vec{x} - \vec{x}_k||_2)$$

where w_k are weights.

Radial basis functions

We want the function to match the data, so that $f(\vec{x}_k) = f_k$. This gives a linear system of equations for the w_k:

$$\begin{pmatrix} \phi_{11} & \phi_{12} & \cdots & \phi_{1n} \\ \phi_{21} & \phi_{22} & \cdots & \phi_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ \phi_{n1} & \phi_{n2} & \cdots & \phi_{nn} \end{pmatrix} \begin{pmatrix} w_1 \\ w_2 \\ \vdots \\ w_n \end{pmatrix} = \begin{pmatrix} f_1 \\ f_2 \\ \vdots \\ f_n \end{pmatrix}$$

where $\phi_{ij} = \phi(||\vec{x}_i - \vec{x}_j||_2)$.

Choices for the radial function

Let ϵ be an inverse length scale. Some common choices of radial function are

- Gaussian, $\phi(r) = e^{-(\epsilon r)^2}$
- Multiquadric, $\phi(r) = \sqrt{1 + (\epsilon r)^2}$
- Inverse multiquadric, $\phi(r) = 1/\sqrt{1 + (\epsilon r)^2}$
- Polyharmonic spline,

$$\phi(r) = \begin{cases} r^k & \text{for } k = 1, 3, 5, \ldots, \\ r^k \log r & \text{for } k = 2, 4, 6, \ldots \end{cases}$$

Many possibilities, but it is desirable to obtain a symmetric positive definite matrix. Equivalent to requiring that the Fourier transform is positive everywhere.

The Gaussian and multiquadric functions are positive definite.[13]

[13] R. Schaback, A practical guide to radial basis functions, 2007.

Using LAPACK to solve dense linear systems

Computer demo: using LAPACK to compute radial basis function interpolations

Krylov methods revisited[14]

Given a matrix A and vector b, a Krylov sequence is the set of vectors

$$\{b, Ab, A^2b, A^3b, \ldots\}$$

The corresponding Krylov subspaces are the spaces spanned by successive groups of these vectors

$$\mathcal{K}_m(A, b) \equiv \text{span}\{b, Ab, A^2b, \ldots, A^{m-1}b\}$$

An important advantage: Krylov methods do not deal directly with A, but rather with matrix–vector products involving A

This is particularly helpful when A is large and sparse, since matrix–vector multiplications are relatively cheap

[14] This is a quick review of material from AM205 unit 5.

Arnoldi Iteration

We define a matrix as being in Hessenberg form in the following way:

- A is called upper-Hessenberg if $a_{ij} = 0$ for all $i > j + 1$
- A is called lower-Hessenberg if $a_{ij} = 0$ for all $j > i + 1$

The Arnoldi iteration is a Krylov subspace iterative method that reduces A to upper-Hessenberg form

Arnoldi Iteration

For $A \in \mathbb{C}^{n \times n}$, we want to compute $A = QHQ^*$, where H is upper Hessenberg and Q is unitary (*i.e.* $QQ^* = \mathrm{I}$)

However, we suppose that n is huge! Hence we do not try to compute the full factorization

Instead, let us consider just the first $m \ll n$ columns of the factorization $AQ = QH$

Therefore, on the left-hand side, we only need the matrix $Q_m \in \mathbb{C}^{n \times m}$:

$$Q_m = \left[\begin{array}{c|c|c|c} & & & \\ q_1 & q_2 & \cdots & q_m \\ & & & \end{array}\right]$$

Arnoldi Iteration

On the right-hand side, we only need the first m columns of H

More specifically, due to upper-Hessenberg structure, we only need $\widetilde{H}_m$, which is the $(m+1) \times m$ upper-left section of H:

$$\widetilde{H}_m = \begin{bmatrix} h_{11} & & \cdots & h_{1m} \\ h_{21} & h_{22} & & \\ & \ddots & \ddots & \vdots \\ & & h_{m,m-1} & h_{mm} \\ & & & h_{m+1,m} \end{bmatrix}$$

$\widetilde{H}_m$ only interacts with the first $m+1$ columns of Q, hence we have

$$AQ_m = Q_{m+1}\widetilde{H}_m$$

Arnoldi Iteration

$$\begin{bmatrix} & A & \end{bmatrix} \begin{bmatrix} q_1 & \cdots & q_m \end{bmatrix} = \begin{bmatrix} q_1 & \cdots & q_{m+1} \end{bmatrix} \begin{bmatrix} h_{11} & \cdots & h_{1m} \\ h_{21} & \cdots & h_{2m} \\ & \ddots & \vdots \\ & & h_{m+1,m} \end{bmatrix}$$

The m^{th} column can be written as

$$Aq_m = h_{1m}q_1 + \cdots + h_{mm}q_m + h_{m+1,m}q_{m+1}$$

Or, equivalently

$$q_{m+1} = (Aq_m - h_{1m}q_1 - \cdots - h_{mm}q_m)/h_{m+1,m}$$

Arnoldi iteration is just the Gram–Schmidt method that constructs the h_{ij} and the (orthonormal) vectors q_j, $j = 1, 2, \ldots$

Arnoldi Iteration

1: choose b arbitrarily, then $q_1 = b/\|b\|_2$
2: **for** $m = 1, 2, 3, \ldots$ **do**
3: $\quad v = Aq_m$
4: $\quad$ **for** $j = 1, 2, \ldots, m$ **do**
5: $\quad\quad h_{jm} = q_j^* v$
6: $\quad\quad v = v - h_{jm} q_j$
7: $\quad$ **end for**
8: $\quad h_{m+1,m} = \|v\|_2$
9: $\quad q_{m+1} = v/h_{m+1,m}$
10: **end for**

This is akin to the modified Gram–Schmidt method because the updated vector v is used in line 5 (vs. the "raw vector" Aq_m)

Also, we only need to evaluate Aq_m and perform some vector operations in each iteration

Lanczos Iteration

Lanczos iteration is the Arnoldi iteration in the special case that A is hermitian

However, we obtain some significant computational savings in this special case

Let us suppose for simplicity that A is symmetric with real entries, and hence has real eigenvalues

Then $H_m = Q_m^T A Q_m$ is also symmetric, and hence must be tridiagonal

Lanczos Iteration

Since H_m is now tridiagonal, we shall write it as

$$T_m = \begin{bmatrix} \alpha_1 & \beta_1 & & & \\ \beta_1 & \alpha_2 & \beta_2 & & \\ & \beta_2 & \alpha_3 & \ddots & \\ & & \ddots & \ddots & \beta_{m-1} \\ & & & \beta_{m-1} & \alpha_m \end{bmatrix}$$

The consequence of tridiagonality: Lanczos iteration is much cheaper than Arnoldi iteration!

Lanczos Iteration

Which leads to the Lanczos iteration

1: $\beta_0 = 0$, $q_0 = 0$
2: choose b arbitrarily, then $q_1 = b/\|b\|_2$
3: **for** $m = 1, 2, 3, \ldots$ **do**
4: $\quad v = Aq_m$
5: $\quad \alpha_m = q_m^T v$
6: $\quad v = v - \beta_{m-1} q_{m-1} - \alpha_m q_m$
7: $\quad \beta_m = \|v\|_2$
8: $\quad q_{m+1} = v/\beta_m$
9: **end for**

Solving linear systems with Krylov methods

We aim to use Krylov methods to solve linear systems $Ax = b$

Only place to look is in the Krylov subspace. Try a solution $x_k \in \mathcal{K}_k$. Suppose true solution is $x = A^{-1}b$ and residual is $r_k = b - Ax_k$. Could aim for

- Minimizing $\|x_k - x\|_2$. There is not enough information in the Krylov subspace to do this.
- Minimizing $\|r_k\|_2$. This leads to algorithms such as MINRES for symmetric A and GMRES for nonsymmetric A.
- For symmetric A, define the norm $\|x - x_k\|_A$. Minimizing this results in the conjugate gradient method.

Conjugate Gradient Method

The CG algorithm is given by

1: $x_0 = 0,\ r_0 = b,\ p_1 = b$
2: **for** $k = 1, 2, 3, \ldots$ **do**
3: $\quad z = Ap_k$
4: $\quad \nu_k = (r_{k-1}^T r_{k-1})/(p_k^T z)$
5: $\quad x_k = x_{k-1} + \nu_k p_k$
6: $\quad r_k = r_{k-1} - \nu_k z$
7: $\quad \mu_k = (r_k^T r_k)/(r_{k-1}^T r_{k-1})$
8: $\quad p_{k+1} = r_k + \mu_k p_k$
9: **end for**

See AM205 unit 5 for a full discussion of this algorithm. At every stage x_k minimizes $\|x_k - x\|_A$ within $\mathcal{K}_k(A, b)$.

Basic conjugate gradient example

Consider the one-dimensional Poission equation for $u(x)$,

$$\frac{\partial^2 u}{\partial x^2} = f$$

on the interval $[0, 1]$, with Dirichlet conditions $u(0) = u(1) = 0$.

Discretize as $u_j = u(jh), f_j = f(jh)$ where $h = 1/n-1$. Hence $u_0 = u_{n-1} = 0$ and

$$\frac{u_{j+1} - 2u_j + u_{j-1}}{h^2} = f_j$$

for $j = 1, \ldots, n-2$.

Basic conjugate gradient example

```
scafell:unit2/lec7+8% ./basic_cg_test
# Iter 0, residual 3
# Iter 1, residual 5.61249
# Iter 2, residual 4.1833
# Iter 3, residual 2.73861
# Iter 4, residual 1.22474
# Iter 5, residual 0
```

Residuals decrease, although it is typical to see non-monotoic behavior.

After five iterations, the solution x is contained within the Krylov subspace, and the residual decreases to zero.

Compactly supported radial basis functions

We return to the radial basis function example. Since the conjugate gradient method is best-suited sparse matrices, we use radial functions of compact support. Define

$$(1-r)_+^k = \begin{cases} (1-r)^k & \text{for } 0 \leq r < 1, \\ 0 & \text{for } r \geq 1 \end{cases}$$

Wendland's functions are compact, k-differentiable, and positive definite.[15]

$\phi(r)$	k
$(1-r)_+^2$	0
$(1-r)_+^4(4r-1)$	2
$(1-r)_+^6(35r^2+18r+3)$	4
$(1-r)_+^8(32r^3+25r^2+8r+1)$	6

[15] The set given here are positive definite up to three dimensions.

Convergence

Convergence of the conjugate gradient method is better when the matrix A has a small condition number

A way to improve convergence is to use preconditioning. We find a matrix M that is an approximation to A, and solve $M^{-1}Ax = M^{-1}b$. We want

- M is symmetric and positive definite
- $M^{-1}A$ is well conditioned and has few extreme eigenvalues
- $Mx = b$ is easy to solve

Preconditioned Conjugate Gradient Method

The preconditioned CG algorithm is given by

1: $x_0 = 0,\ r_0 = b,\ p_1 = M^{-1}b,\ y_0 = M^{-1}r_0$
2: **for** $k = 1, 2, 3, \ldots$ **do**
3: $\quad z = Ap_k$
4: $\quad \nu_k = (y_{k-1}^T r_{k-1})/(p_k^T z)$
5: $\quad x_k = x_{k-1} + \nu_k p_k$
6: $\quad r_k = r_{k-1} - \nu_k z$
7: $\quad y_k = M^{-1} r_k$
8: $\quad \mu_k = (y_k^T r_k)/(y_{k-1}^T r_{k-1})$
9: $\quad p_{k+1} = y_k + \mu_k p_k$
10: **end for**

Examples of preconditioning

- Diagonal (Jacobi) preconditioning: define $M = \text{diag}(a_{11}, a_{22}, \ldots, a_{nn})$. Straightforward to invert.
- Block Jacobi preconditioning: Write the matrix in block form as

$$A = \begin{pmatrix} A_{11} & A_{12} & \cdots & A_{1k} \\ \vdots & \vdots & \ddots & \vdots \\ A_{k1} & A_{k2} & \cdots & A_{kk} \end{pmatrix}$$

Define

$$M = \begin{pmatrix} A_{11} & & & \\ & \ddots & & \\ & & & A_{kk} \end{pmatrix}$$

Performing M^{-1} requires inverting each block—much faster than solving the original matrix

Examples of preconditioning

- Incomplete LU/Cholesky factorization: a full LU or Cholesky factorization of a sparse matrix results in fill-in of the zero entries. Adjust algorithm to obtain approximate result with minimum fill-in.
- Multigrid: the multigrid algorithm is an iterative procedure for solving matrix problems, by applying successive V-cycles. Let M^{-1} be the matrix applying one V-cycle—good approximation to the inverse of A.

Radial basis function timing example

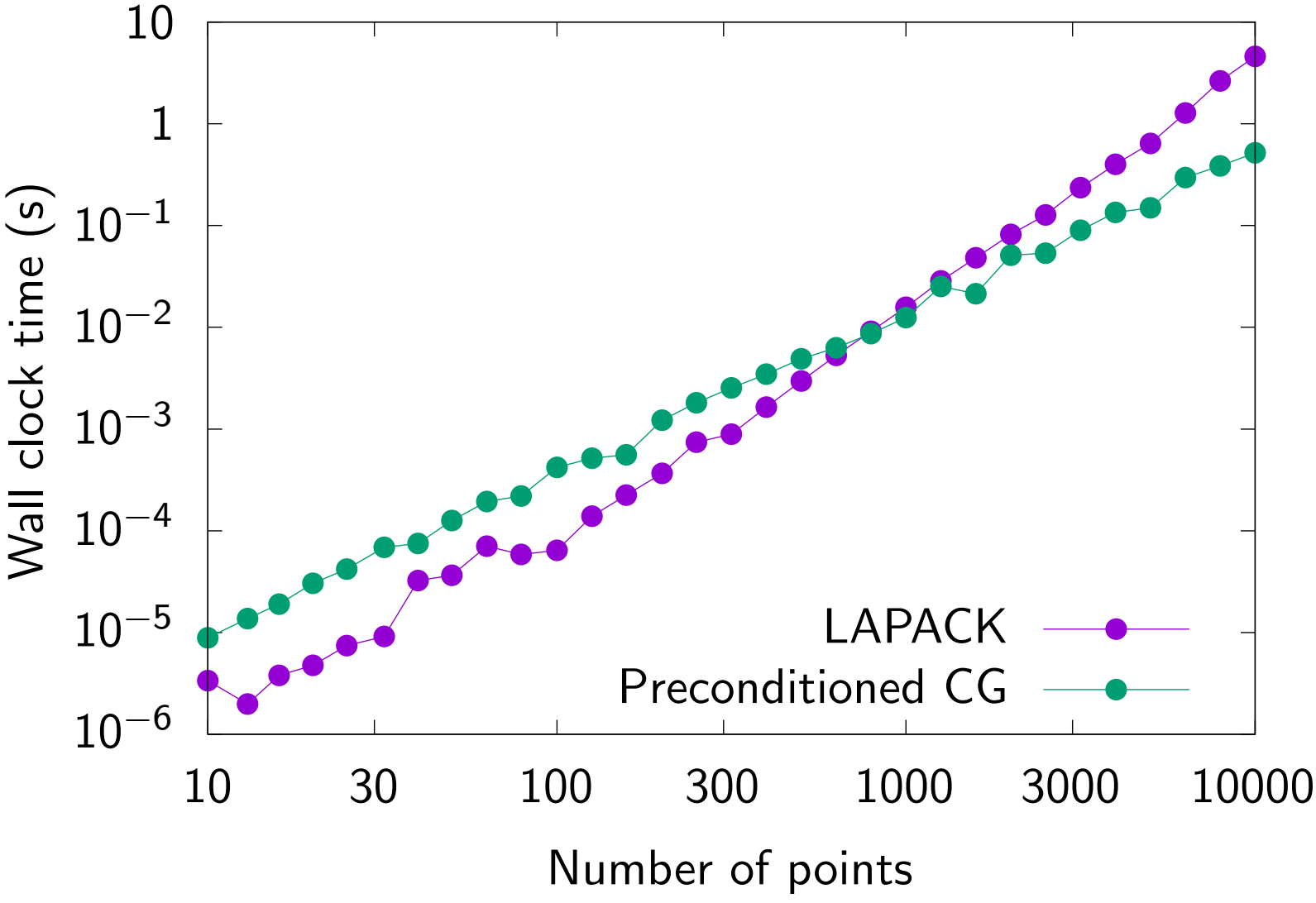
Tested RBF example using points from $n = 10$ to $n = 10^4$.

Use compact Wendland functions with a radius of $5/\sqrt{n}$. Gives approximately 15 non-zero entries per row of matrix.

Two solution algorithms:

- LAPACK – dense linear algebra
- Preconditioned CG – use block Jacobi preconditioner with blocks of size $\sqrt{n}$.

Radial basis function timing example



Radial basis function timing example

For small systems with $n < 800$, dense linear algebra is faster.

For large systems with $n \geq 800$, the $O(n^3)$ scaling of LAPACK makes it inefficient.

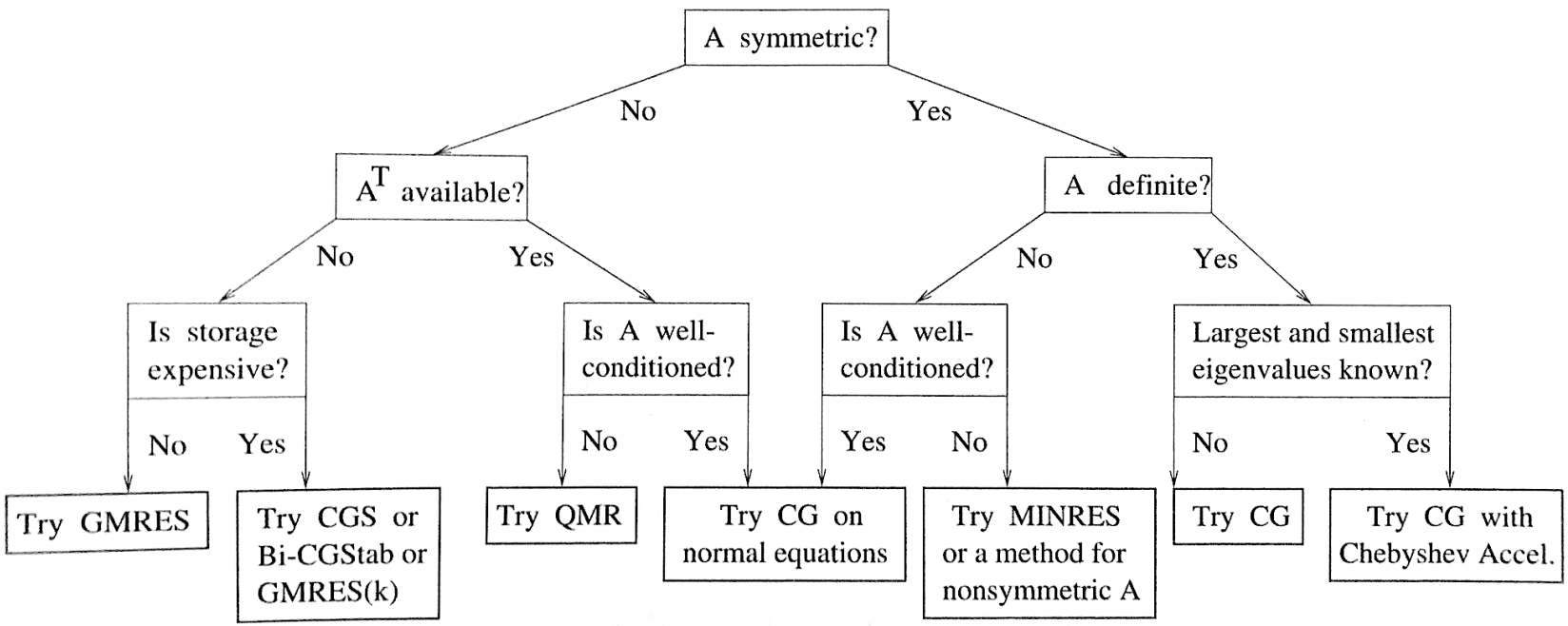
Preconditioned CG has $O(n^{2.37})$ scaling in this example, and therefore becomes the best choice for large numbers of points.

This timing comparison is heavily dependent on the matrix structure and sparsity. LAPACK does better for denser matrices.

Other Krylov methods

The conjugate gradient method only applies to symmetric positive definite linear systems.

There are many related algorithms for solving different types of linear systems. The following flow chart from the textbook[16] illustrates some of the different possibilities.



[16] J. A. Demmel, *Applied Numerical Linear Algebra*, SIAM 1997.

GMRES: Generalized Minimum RESidual method

Consider a general matrix A that may not be symmetric. Short recurrence no longer holds so we must use the Arnoldi algorithm to obtain

$$H_k = Q_k^T A Q_k$$

where Q_k is orthogonal and H_k is upper Hessenberg.

Choose $x_k = Q_k y_k \in \mathcal{K}_k(A, b)$ to minimize the residual $\|r_k\|_2$.

GMRES: Generalized Minimum RESidual method

Manipulating the residual gives

$$
\begin{aligned}
\|r_k\|_2 &= \|b - Ax_k\|_2 \\
&= \|b - AQ_k y_k\|_2 \\
&= \|b - (QHQ^T)Q_k y_k\|_2 \\
&= \|Q^T b - HQ^T Q_k y_k\|_2 \\
&= \left\| e_1 \|b\|_2 - \begin{pmatrix} H_k & H_{uk} \\ H_{ku} & H_u \end{pmatrix} \begin{pmatrix} y_k \\ 0 \end{pmatrix} \right\|_2 \\
&= \left\| e_1 \|b\|_2 - \begin{pmatrix} H_k \\ H_{ku} \end{pmatrix} y_k \right\|_2
\end{aligned}
$$

Here the u subscript refers to the remaining parts of the full matrix H that are not in H_k. e_1 is the first unit vector.

The final line is a linear least-squares problem for y_k, which can be solved using the QR algorithm.

GMRES: solving the least-squares problem

Normally, performing a QR factorization would require $O(k^3)$ iterations.

But here, we require the QR factorization of the $(k+1) \times k$ Hessenberg matrix. We can perform the QR factorization by performing k Givens rotations to rotate out the terms below the diagonal.

GMRES requires $O(kn)$ memory to store the vectors Q_k. A variant to minimize the growth of computation and storage is to stop after k steps, and restart by solving $Ad = r_k = b - Ax_k$, after which the solution is given by $d + x_k$.

This is called GMRES(k). It is still more expensive than conjugate gradient.

The Fast Fourier Transform

Consider a one-dimensional Poisson problem

$$-\frac{d^2 v}{dx^2} = f(x)$$

for a function $v(x)$ on $[0, 1]$ with boundary conditions $v(0) = v(1) = 0$.

Discretize with $N + 2$ evenly spaced points with grid spacing $h = 1/(N + 1)$, so that $x_i = hi$.

Second-order centered finite difference gives

$$-v_{i-1} + 2v_i - v_{i+1} = h^2 f_i$$

for $i = 1, \ldots N$.

Matrix formulation

Writing all equations in a linear system yields

$$T_N \begin{pmatrix} v_1 \\ \vdots \\ v_N \end{pmatrix} = \begin{pmatrix} 2 & -1 & & 0 \\ -1 & \ddots & \ddots & \\ & \ddots & \ddots & -1 \\ 0 & & -1 & 2 \end{pmatrix} \begin{pmatrix} v_1 \\ \vdots \\ v_N \end{pmatrix} = h^2 \begin{pmatrix} f_1 \\ \vdots \\ f_N \end{pmatrix}.$$

The eigenvectors of T_N are

$$z_j(k) = \sqrt{\frac{2}{N+1}} \sin \frac{jk\pi}{N+1}$$

with corresponding eigenvalues

$$\lambda_j = 2\left(1 - \cos \frac{\pi j}{N+1}\right).$$

Poisson's equation in two dimensions

Now consider the two dimensional Poisson problem

$$-\frac{\partial^2 v}{\partial x^2} - \frac{\partial^2 v}{\partial y^2} = f(x, y)$$

on the unit square $[0, 1]^2$ with $v = 0$ on the boundary.
Discretize using a $(N + 2) \times (N + 2)$ grid with $x_j = jh$ and $y_k = kh$ with $h = 1/(N + 1)$. Write

$$v_{jk} = v(jh, kh), \qquad f_{jk} = f(jh, kh).$$

Equations in the linear system are

$$4v_{jk} - v_{j-1,k} - v_{j+1,k} - v_{j,k-1} - v_{j,k+1} = h^2 f_{jk}.$$

Matrix formulation

Rewrite unknowns v_{jk} as occupying an $N \times N$ matrix V. Then

$$2v_{jk} - v_{j-1,k} - v_{j+1,k} = (T_N V)_{jk},$$
$$2v_{jk} - v_{j,k-1} - v_{j,k+1} = (VT_N)_{jk}.$$

Hence the problem can be written as

$$T_N V + VT_N = h^2 F$$

where F is an $N \times N$ with entries f_{jk}.

Eigenvectors and eigenvalues for 2D problem

Let $V = z_j z_k^{\mathsf{T}}$. Then

$$\begin{aligned} T_N V + V T_N &= (T_N z_j) z_k^{\mathsf{T}} + z_j (z_k^{\mathsf{T}} T_N) \\ &= (\lambda_j z_j) z_k^{\mathsf{T}} + z_j (z_k^{\mathsf{T}} \lambda_k) \\ &= (\lambda_j + \lambda_k) z_j z_k^{\mathsf{T}} \\ &= (\lambda_j + \lambda_k) V \end{aligned}$$

and hence $z_j z_k^{\mathsf{T}}$ is an eigenevector of the 2D problem with eigenvalue $\lambda_j + \lambda_k$. We obtain a full set of N^2 eigenvectors for the problem.

Solving the equation via an eigendecomposition

Let $T_N = Z\Lambda Z^{\mathsf{T}}$ be the eigendecomposition of T_N. Note that $Z^{\mathsf{T}}Z = I$ since Z is orthogonal. Then

$$Z\Lambda Z^{\mathsf{T}}V + V(Z\Lambda Z^{\mathsf{T}}) = h^2F$$

and

$$Z^{\mathsf{T}}Z\Lambda Z^{\mathsf{T}}VZ + Z^{\mathsf{T}}V(Z\Lambda Z^{\mathsf{T}})Z = h^2Z^{\mathsf{T}}FZ,$$

which becomes

$$\Lambda V' + V'\Lambda = h^2F'$$

where $V' = Z^{\mathsf{T}}VZ$ and $F' = Z^{\mathsf{T}}FZ$.

Solving the equation via an eigendecomposition

Hence

$$\lambda_j v'_{jk} + v'_{jk} \lambda_k = h^2 f'_{jk}$$

and so

$$v'_{jk} = \frac{h^2 f'_{jk}}{\lambda_j + \lambda_k}.$$

Three steps to obtain a solution:

1. Compute $F' = Z^{\mathsf{T}} F Z$ ($O(N^3)$ operations[17])
2. Find $v'_{jk} = h^2 f'_{jk} / (\lambda_j + \lambda_k)$ ($O(N^2)$ operations)
3. Compute $V = Z V' Z^{\mathsf{T}}$ ($O(N^3)$ operations)

However, we will soon see that the Fast Fourier Transform allows steps 1 and 3 to be performed in $O(N^2 \log N)$ operations, turning this into a practical algorithm.

[17] Assuming a conventional matrix–matrix multiplication routine.

Alternative viewpoint: the Kronecker product

Write $\mathrm{vec}(V)$ to be the operator converting the $N \times N$ matrix into an N^2-vector of unknowns. Write

$$T_{N\times N} = I \otimes T_N + T_N \otimes I = (Z \otimes Z)(I \otimes \Lambda + \Lambda \otimes I)(Z \otimes Z)^{\mathsf{T}}.$$

Then

$$\begin{aligned}
\mathrm{vec}(V) &= (T_{N\times N})^{-1}\,\mathrm{vec}(h^2 F) \\
&= \Big((Z \otimes Z)(I \otimes \Lambda + \Lambda \otimes I)(Z \otimes Z)^{\mathsf{T}}\Big)^{-1}\,\mathrm{vec}(h^2 F) \\
&= (Z \otimes Z)(I \otimes \Lambda + \Lambda \otimes I)^{-1}(Z^{\mathsf{T}} \otimes Z^{\mathsf{T}})\,\mathrm{vec}(h^2 F).
\end{aligned}$$

While this is less notationally elegant, it makes it clear that the solution procedure could be extended to arbitrary dimensions.

i.e. in 3D, we would consider $(Z \otimes Z \otimes Z)$, applying the matrix Z to field values in each coordinate direction.

The Discrete Fourier Transform

For notational convenience, we now switch to numbering rows and columns starting from zero.

Definition: The discrete Fourier transform (DFT) of a vector $x \in \mathbb{C}^N$ is $y = \Phi x$ where Φ is an $N \times N$ matrix with terms $\phi_{jk} = \omega^{jk}$ and $\omega = e^{-2\pi i/N}$ is the Nth root of unity. The inverse discrete Fourier transform (IDFT) is $x = \Phi^{-1} y$.

$\Phi/\sqrt{N}$ is a symmetric unitary matrix, $\Phi^{-1} = \Phi^*/N = \bar{\Phi}/N$.

Connection to 2D Poisson problem

The procedure to solve the 2D Poisson problem required multiplication by Z, where[18]

$$z_{jk} = \sqrt{\frac{2}{N+1}} \sin \frac{\pi(j+1)(k+1)}{N+1}.$$

Consider the $(2N+2) \times (2N+2)$ DFT matrix whose (j,k) entry is

$$\exp\left(\frac{-2\pi ijk}{2N+2}\right) = \exp\left(\frac{-\pi ijk}{N+1}\right) = \cos\frac{\pi jk}{N+1} - i\sin\frac{\pi jk}{N+1}.$$

The $N \times N$ matrix Z is proportional to the imaginary part of Φ for $1 \le j \le N, 1 \le k \le N$.

Hence if we can multiply efficiently by Φ, then we can multiply efficiently by Z. In fact, Z is the discrete sine transform (DST), a variant of the DFT for real data.

[18] Note this is slightly different due to the shift in matrix indexing.

Connection to discrete convolution

Let $a(x) = \sum_{k=0}^{N-1} a_k x^k$ and $b(x) = \sum_{k=0}^{N-1} b_k x^k$ be polynomials. Let

$$c(x) = a(x)b(x) = \sum_{k=0}^{2N-1} c_k x^k$$

be the product of the two. The coefficients of $c(x)$ are given by $c_k = \sum_{j=0}^{k} a_j b_{k-j}$.

Theorem: Let $a = (a_0, \ldots, a_{N-1}, 0, \ldots, 0)^{\mathsf{T}}$ and $b = (b_0, \ldots, b_{N-1}, 0, \ldots, 0)^{\mathsf{T}}$ be $2N$-vectors containing the polynomial coefficients. Let $c = (c_0, c_1, \ldots, c_{2N-1})^{\mathsf{T}}$. Then

$$(\Phi c)_k = (\Phi a)_k (\Phi b)_k.$$

Connection to discrete convolution

To prove this theorem, define $a' = \Phi a$. Then

$$a'_k = \sum_{j=0}^{2N-1} a_j \omega^{jk} = a(\omega^k).$$

If b' and c' are defined similarly, then

$$a'_k b'_k = a(\omega^k) b(\omega^k) = c(\omega^k) = c'_k.$$

Since this is true for all k, it follows that $(\Phi c)_k = (\Phi a)_k (\Phi b)_k$.

The Fast Fourier Transform

Finding the discrete Fourier transform is equivalent to evaluating the polynomial $a(x) = \sum_{k=0}^{N-1} a_k x^k$ at $x = \omega^j$ for $0 \leq j \leq N-1$. Assuming $N = 2^m$, write

$$\begin{aligned} a(x) &= a_0 + a_1 x + \ldots a_{N-1} x^{N-1} \\ &= (a_0 + a_2 x^2 + a_4 x^4 + \ldots) + x(a_1 + a_3 x^2 + a_5 x^5 + \ldots) \\ &= a_{\text{even}}(x^2) + x a_{\text{odd}}(x^2). \end{aligned}$$

Hence we need to evaluate two polynomials of degree $N/2 - 1$ at $(\omega^j)^2$ for $0 \leq j \leq N-1$.

Key observation: $\omega^{2j} = \omega^{2(j+N/2)}$, and hence we only need to evaluate the two polynomials at $N/2$ points. Reduces work by a factor of two.

Fast Fourier Transform

Applying this procedure recursively leads to the following algorithm

1: **function** FFT(a, N)
2: **if** $N = 1$ **then**
3: **return** a
4: **else**
5: $a'_{\text{even}} = \text{FFT}(a_{\text{even}}, N/2)$
6: $a'_{\text{odd}} = \text{FFT}(a_{\text{odd}}, N/2)$
7: $\omega = e^{-2\pi i/N}$
8: $w = (\omega^0, \ldots, \omega^{N/2-1})$
9: **return** $a' = (a'_{\text{even}} + w * a'_{\text{odd}}, a'_{\text{even}} - w * a'_{\text{odd}})$
10: **end if**

Here the $*$ operator refers to componentwise multiplication.

Complexity of the FFT

We assume that the values of ω are already precomputed. Number of operations satisfies $C(N) = 2C(N/2) + 3N/2$. Hence

$$\begin{aligned} C(N) &= 2C\left(\frac{N}{2}\right) + \frac{3N}{2} = 4C\left(\frac{N}{4}\right) + 2\frac{3N}{2} \\ &= 8C\left(\frac{N}{8}\right) + 3\frac{3N}{2} = \dots \\ &= \frac{3N\log_2 N}{2}. \end{aligned}$$

This is much better than the original $O(N^2)$ complexity from performing direct sums to evaluate the DFT.

Two-dimensional Poisson problem requires $O(N)$ FFTs, and thus overall time scales like $O(N^2 \log N)$.

Fast Fourier transform libraries

Chip vendors such as Intel provide tuned libraries for the FFT. The Intel MKL contains FFT routines.

FFTW[19] (`www.fftw.org`) is a widely-used open source library.

FFTW follows some similar design principles to BLAS, organizing computation in a cache-friendly manner, and exploiting vectorized instructions where possible.

FFTW provides very good performance across a wide range of platforms. While best perfomance is achieved for grid sizes N that are powers of two, FFTW achieves $O(N \log N)$ performance for any grid size. Multithreaded and parallel routines available.

[19] Stands for "The Fastest Fourier Transform in the West". Developed by Matteo Frigo and Steven Johnson at MIT.

Frequency analysis of music samples

Consider the following four music samples of length 0.1 s:

- Vocal sample from *You Couldn't Be Cuter* by Sylvia McNair. This is a jazz standard, but McNair is primarily an opera singer.
- Vocal sample from *Marry Me* by St. Vincent.
- Vocal sample from *It's Alright, Ma (I'm Only Bleeding)* by Bob Dylan.
- Drum sample from *Guess They Never Told You* by The American Symphony of Soul. (Drums played by AM225 TF Dan Fortunato.)

Samples were prepared using Audacity, which exports the sound signal as single-precision floats. Contains stereo information (left and right channels).

FFTW example for sound sample analysis #1

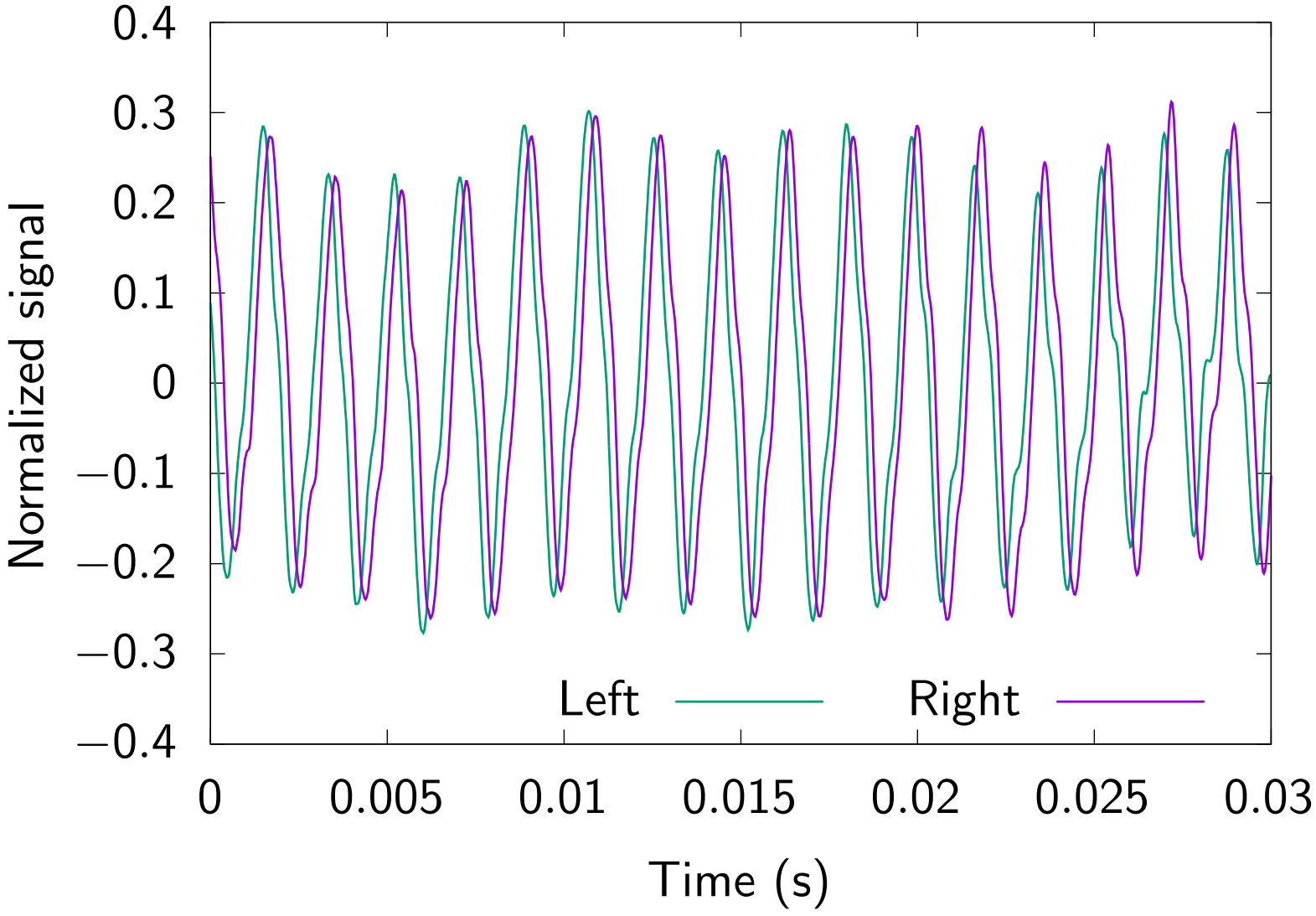
```
#include <cstdio>
#include <cmath>
#include <fftw3.h>
#include "omp.h"

const int n=4096;

int main() {

    // Read in the binary data stereo sample in single precision
    float e[n];
    FILE *fp=fopen("asos.raw","rb");
    fread(e,sizeof(float),2*n,fp);
    fclose(fp);

    // Allocate memory for FFTW input data, and convert sound
    // sample to double precision, just getting left channel
    double *f=fftw_alloc_real(n),re,im;
    for(int i=0;i<n;i++) f[i]=e[2*i];

    ...
```

FFTW example for sound sample analysis #2

```
    ...

    // Allocate memory for complex FFTW output data
    int fftn=n/2+1;
    fftw_complex *c=fftw_alloc_complex(n);

    // Make FFTW plan, and perform the transform
    fftw_plan plan_dft(fftw_plan_dft_r2c_1d(n,f,c,FFTW_ESTIMATE));
    fftw_execute(plan_dft);

    // Output magnitudes of each term
    for(int i=0;i<fftn;i++) {
        re=c[i][0];
        im=c[i][1];
        printf("%g %g\n",44000./n*i,sqrt(re*re+im*im));
    }

    // Free dynamically allocated memory
    fftw_destroy_plan(plan_dft);
    fftw_free(c);
    fftw_free(f);
}
```

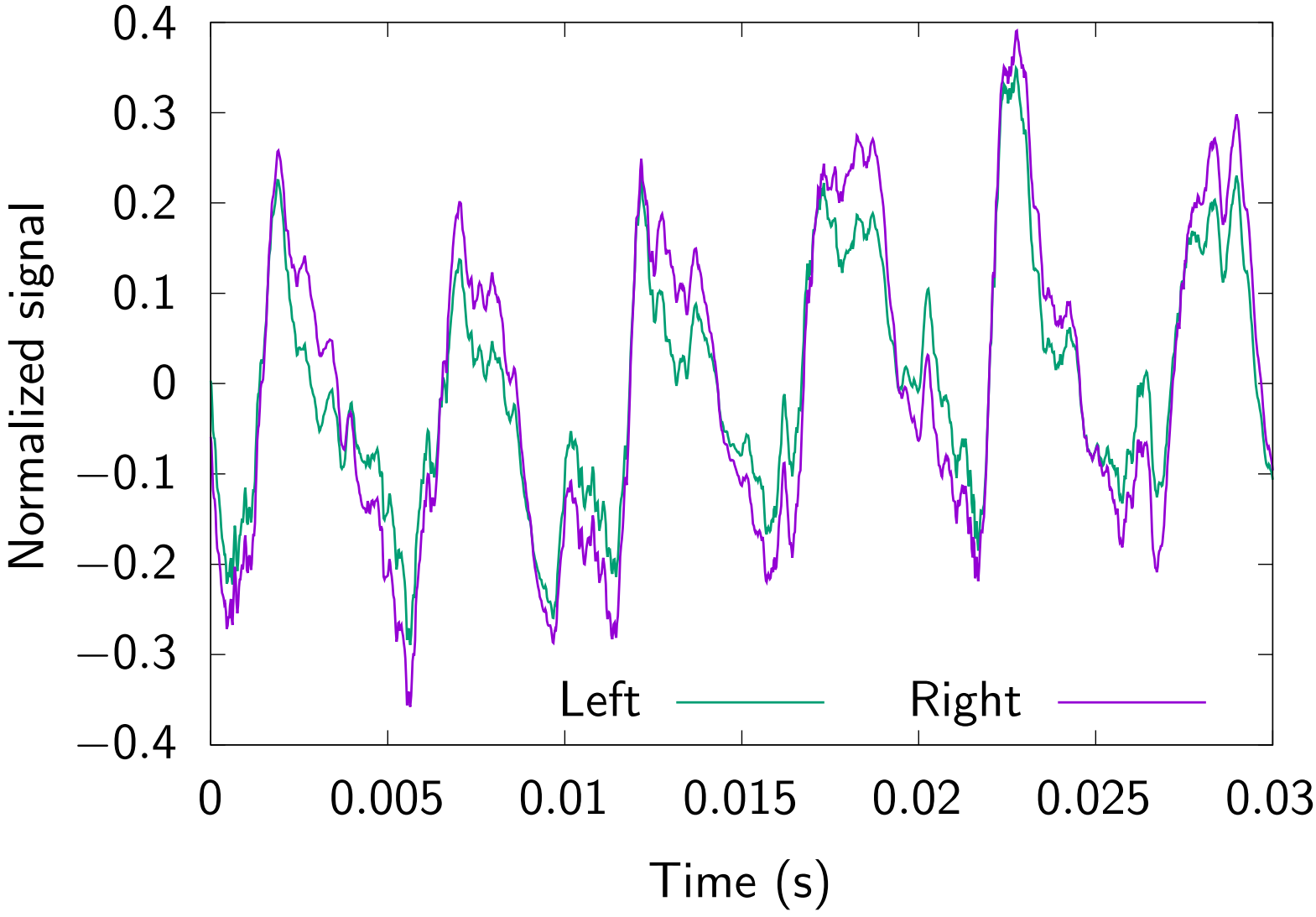
Sylvia McNair waveform

(Showing both stereo channels)



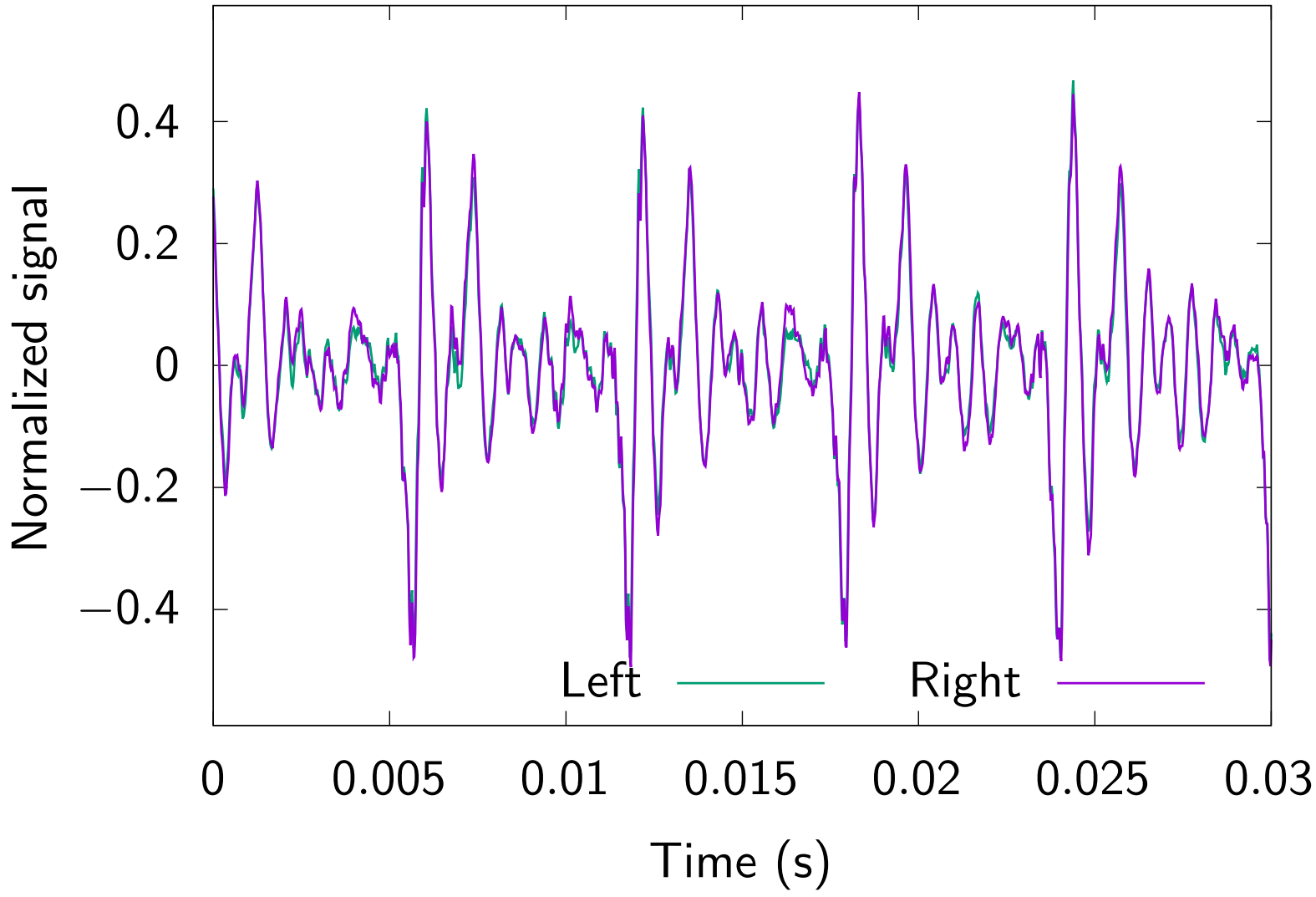
St. Vincent waveform

(Showing both stereo channels)



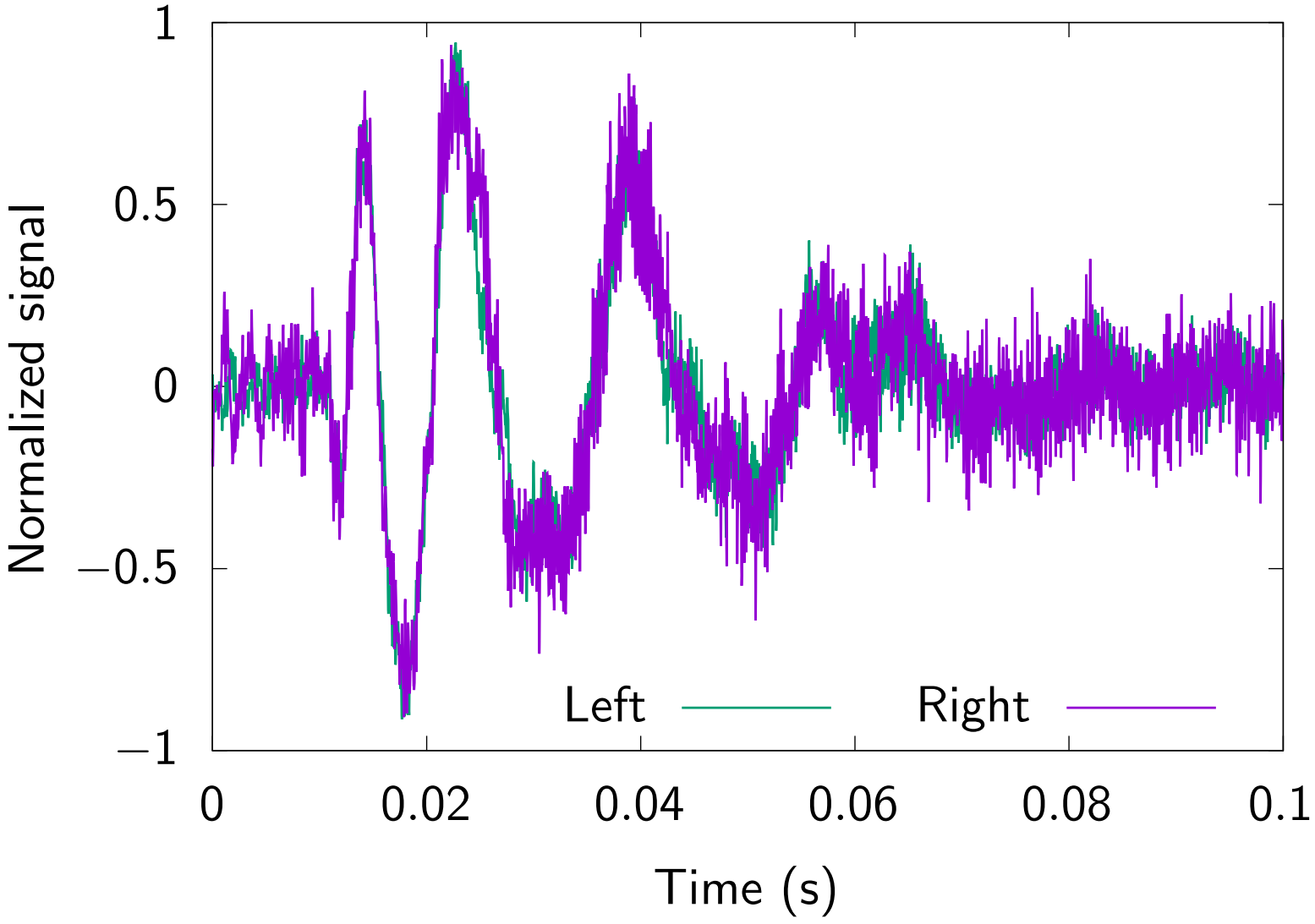
Bob Dylan waveform

(Showing both stereo channels)

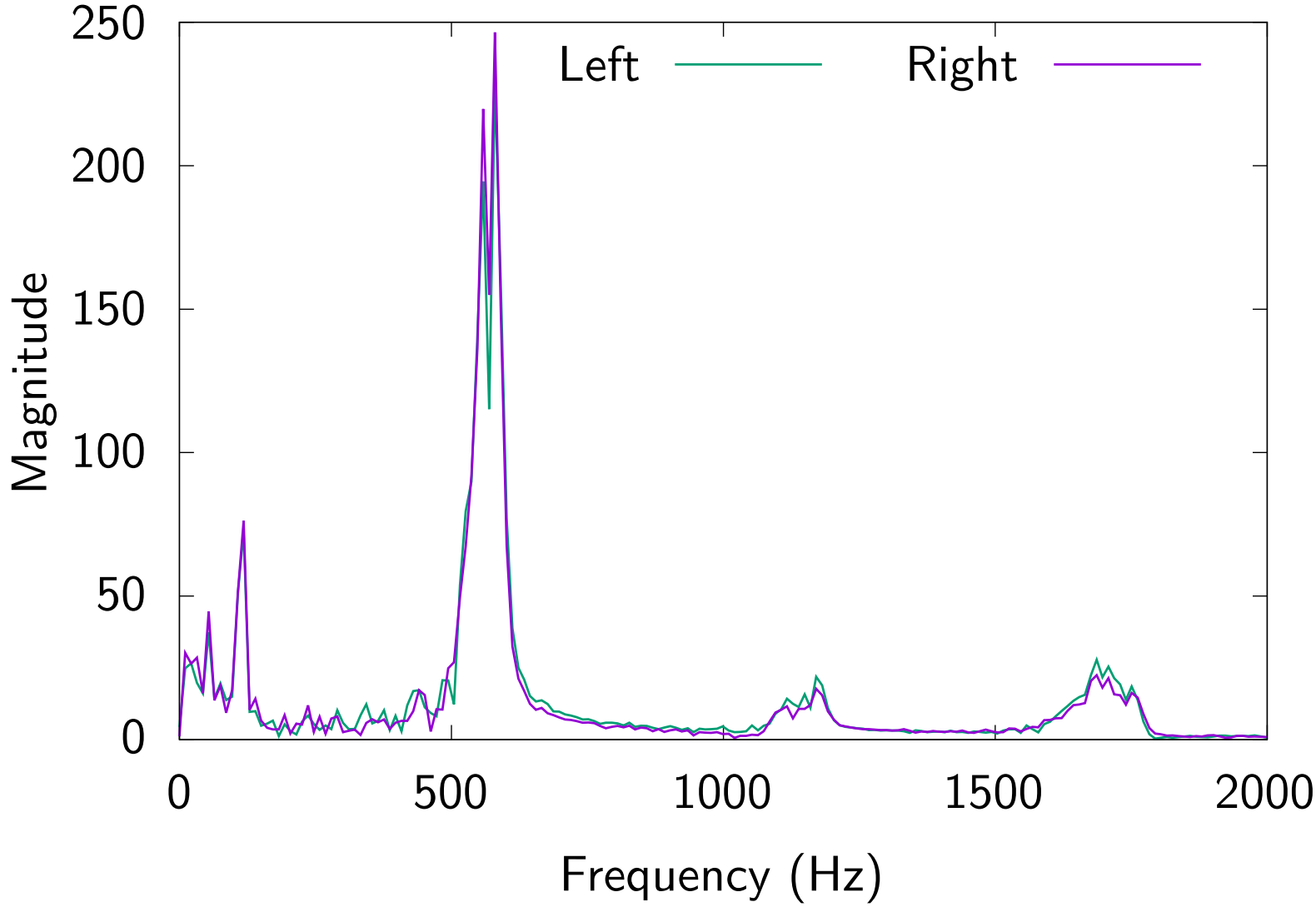


American Symphony of Soul drum waveform

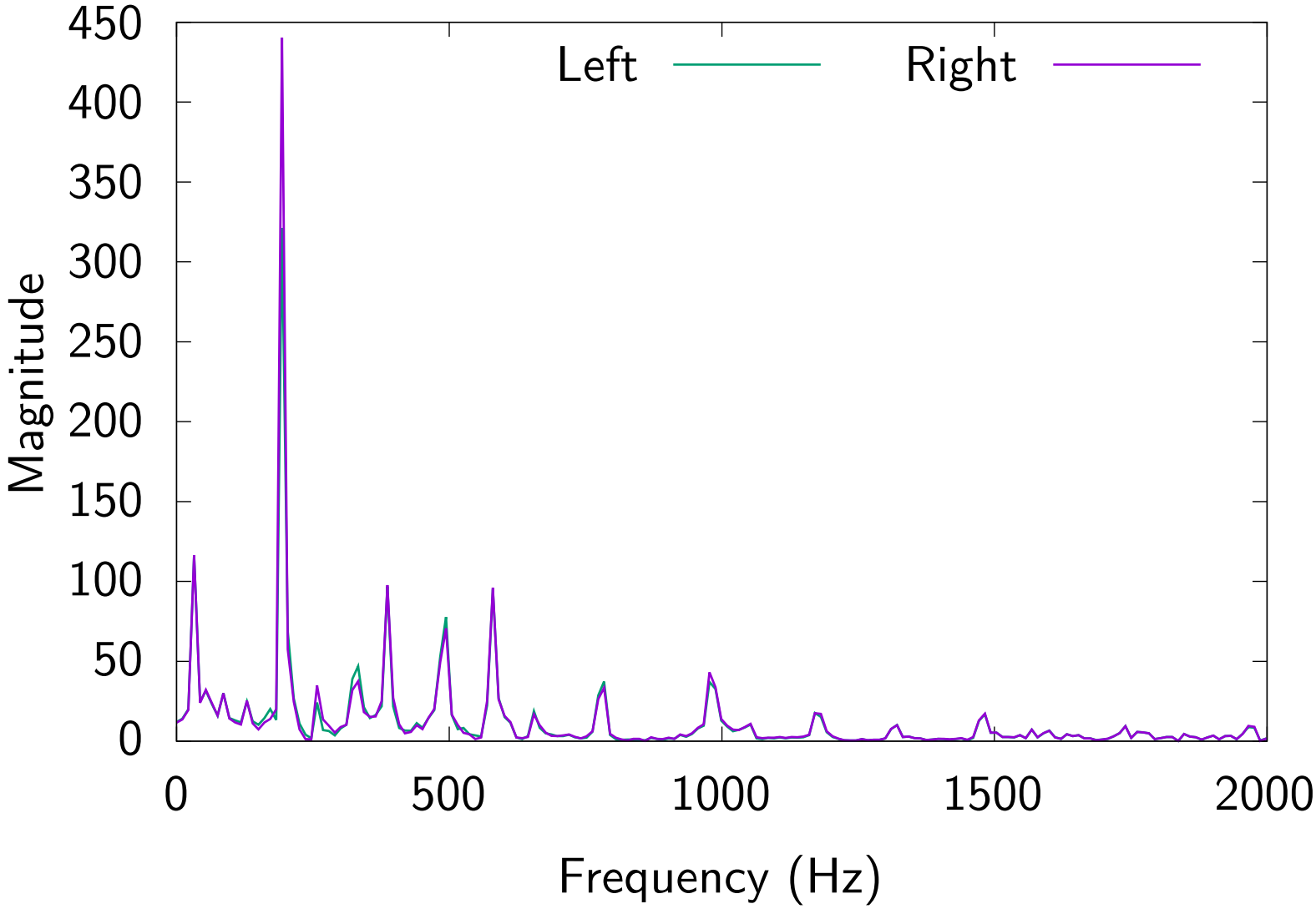
(Showing both stereo channels)



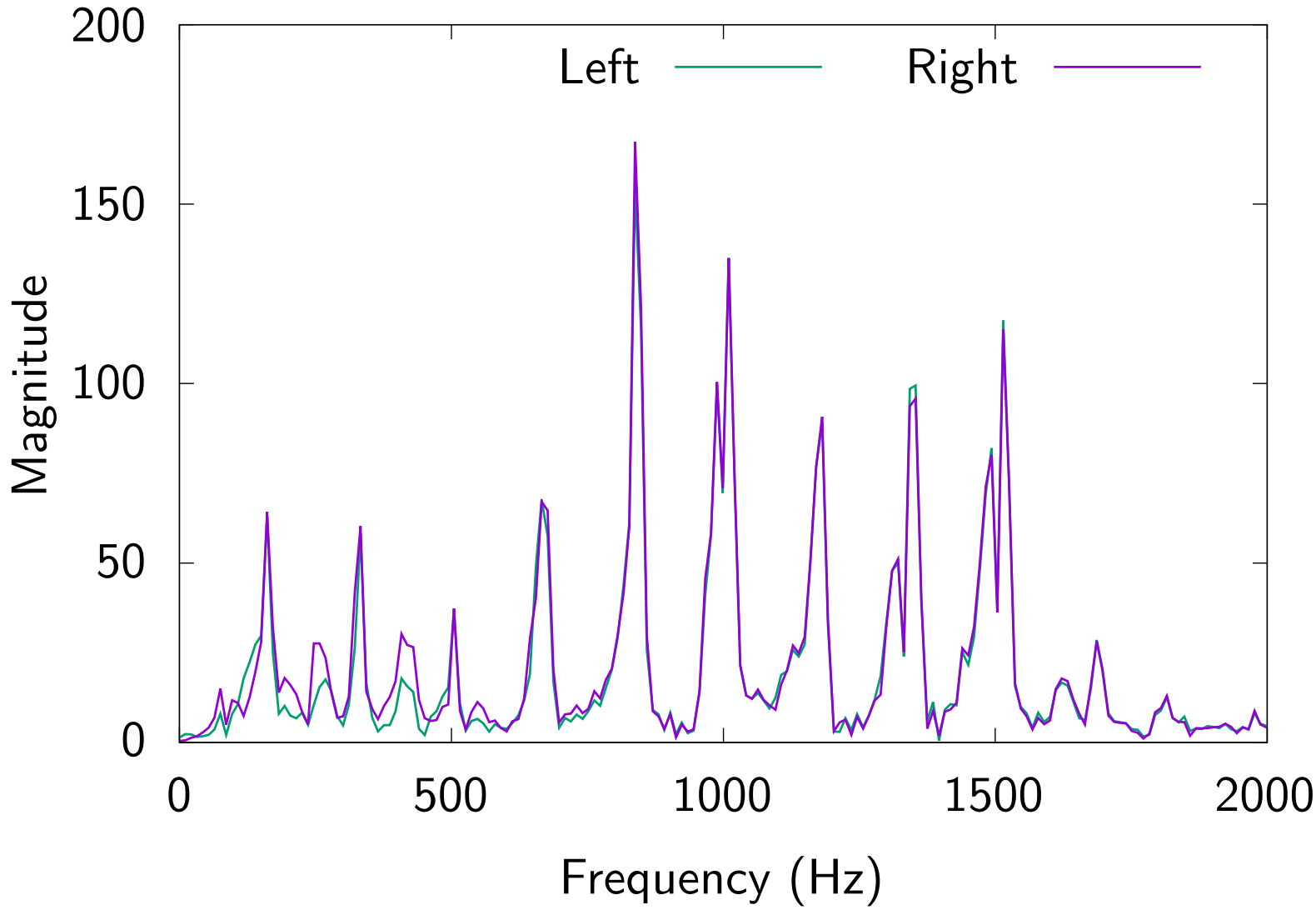
Sylvia McNair frequency spectrum



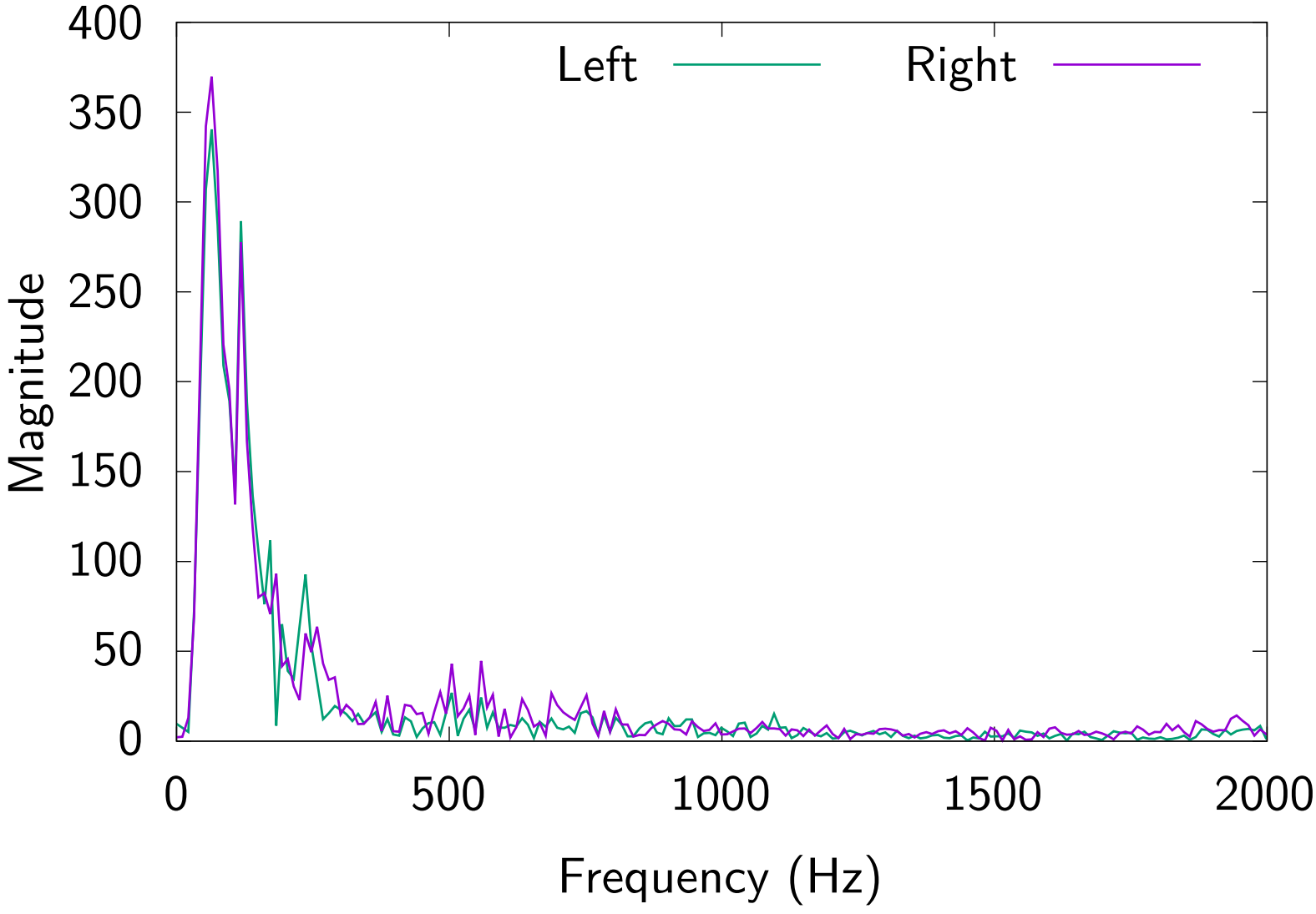
St. Vincent frequency spectrum



Bob Dylan frequency spectrum

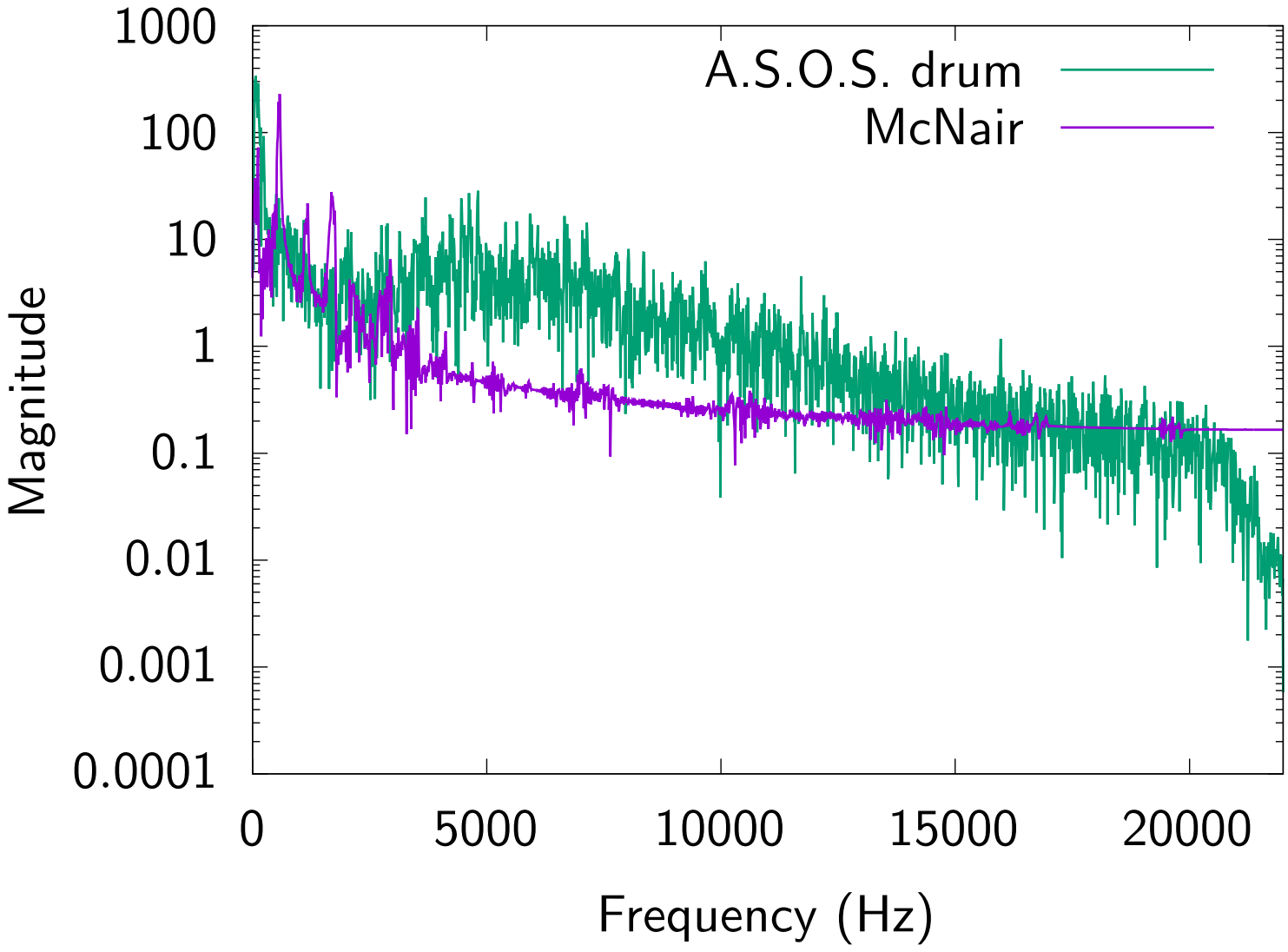


American Symphony of Soul drum frequency spectrum



Semi-log comparison of Sylvia McNair and A.S.O.S.

(Showing that the drum sample has high-frequency components)



FFTW performance tuning and execution

FFTW comes with its own memory allocation routines, which ensure that memory is allocated on 16-byte boundaries, which can be beneficial for some vectorized machine instructions:

```
const int n=1024;
double *src=fftw_alloc_real(n);
fftw_complex *dest=fftw_alloc_complex(n);
```

Before executing an FFT, an `fftw_plan` must be created to tell FFTW the source and destination, plus the size and type of FFT:

```
fftw_plan plan_dft(fftw_plan_dft_r2c_1d(n,src,dest,FFTW_ESTIMATE));
```

The final option of `FFTW_ESTIMATE` tells FFTW to use heuristics to plan the FFT for good performance. Alternatively `FFTW_MEASURE` can be used, which performs some trial FFTs to test for best performance. The `FFTW_PATIENT` option enables even more tests.

FFTW performance tuning and execution

Once the plan is set up, the FFT is performed using:

```
fftw_execute(fftw_plan);
```

The same plan can be used on different arrays (*e.g.* `src2` and `dest2`), so long as the memory alignment is the same:

```
fftw_execute(fftw_plan,src2,dest2);
```

At the end of the program, the arrays and plans must be explicitly freed:

```
fftw_destroy_plan(plan_dft);
fftw_free(dest);
fftw_free(src);
```

Return to the 2D Poisson problem

In the last lecture we introduced a model 2D Poisson problem

$$-\frac{\partial^2 v}{\partial x^2} - \frac{\partial^2 v}{\partial y^2} = f(x, y)$$

on the unit square $[0, 1]^2$ with $v = 0$ on the boundary. Discretized using a $(N + 2) \times (N + 2)$ grid with $x_j = jh$ and $y_k = kh$ with $h = 1/(N + 1)$.

Problem was rewritten as

$$T_N V + V T_N = h^2 F$$

where T_N is a triangular matrix, V is a matrix containing the solution, and F is a matrix containing the source term.

FFT solution method

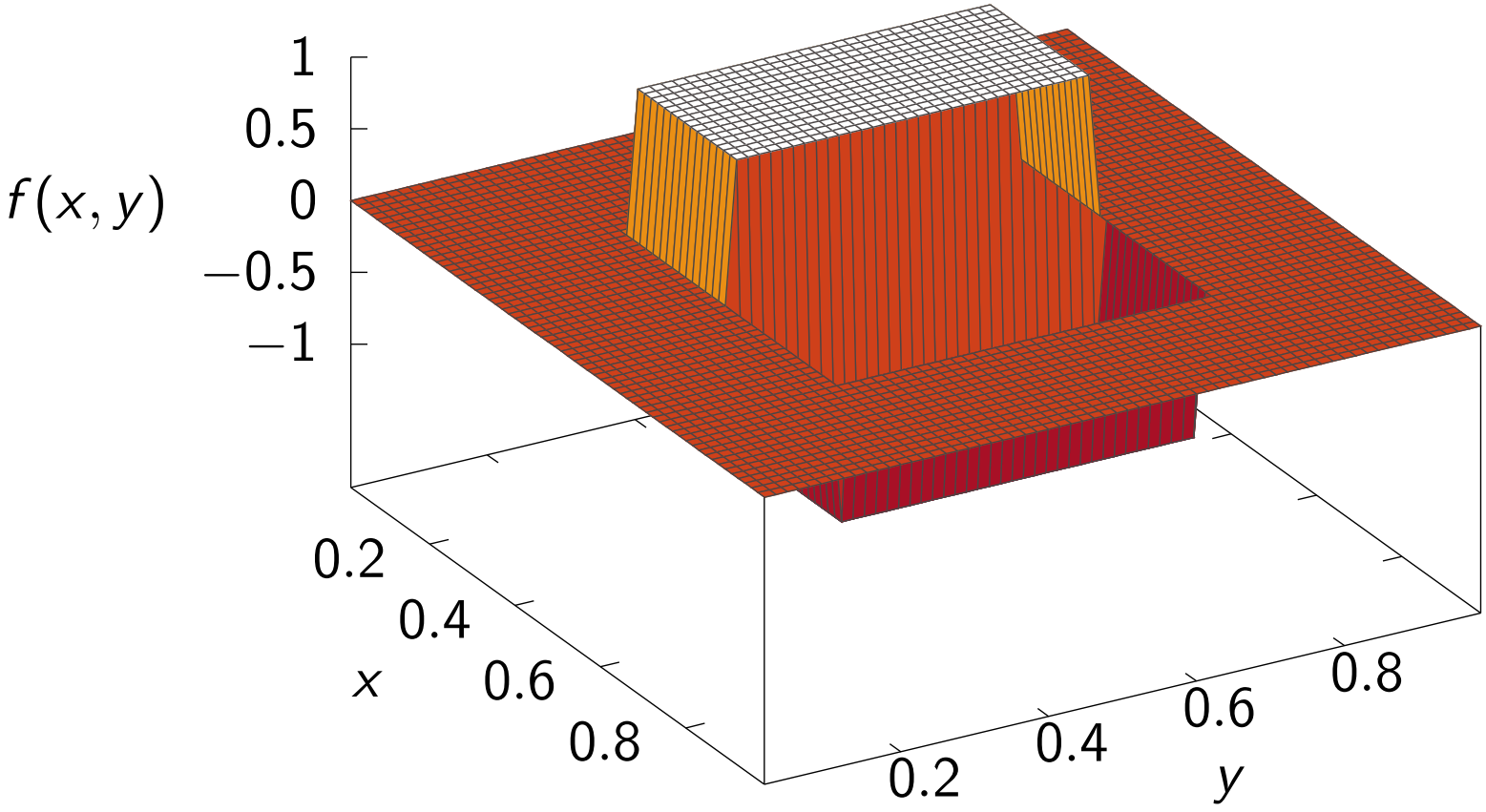
Let $T_N = Z\Lambda Z^{\mathsf{T}}$ be the eigendecomposition of T_N. Then a solution method is

1. Compute $F' = Z^{\mathsf{T}} F Z$
2. Find $v'_{jk} = h^2 f'_{jk}/(\lambda_j + \lambda_k)$
3. Compute $V = Z V' Z^{\mathsf{T}}$

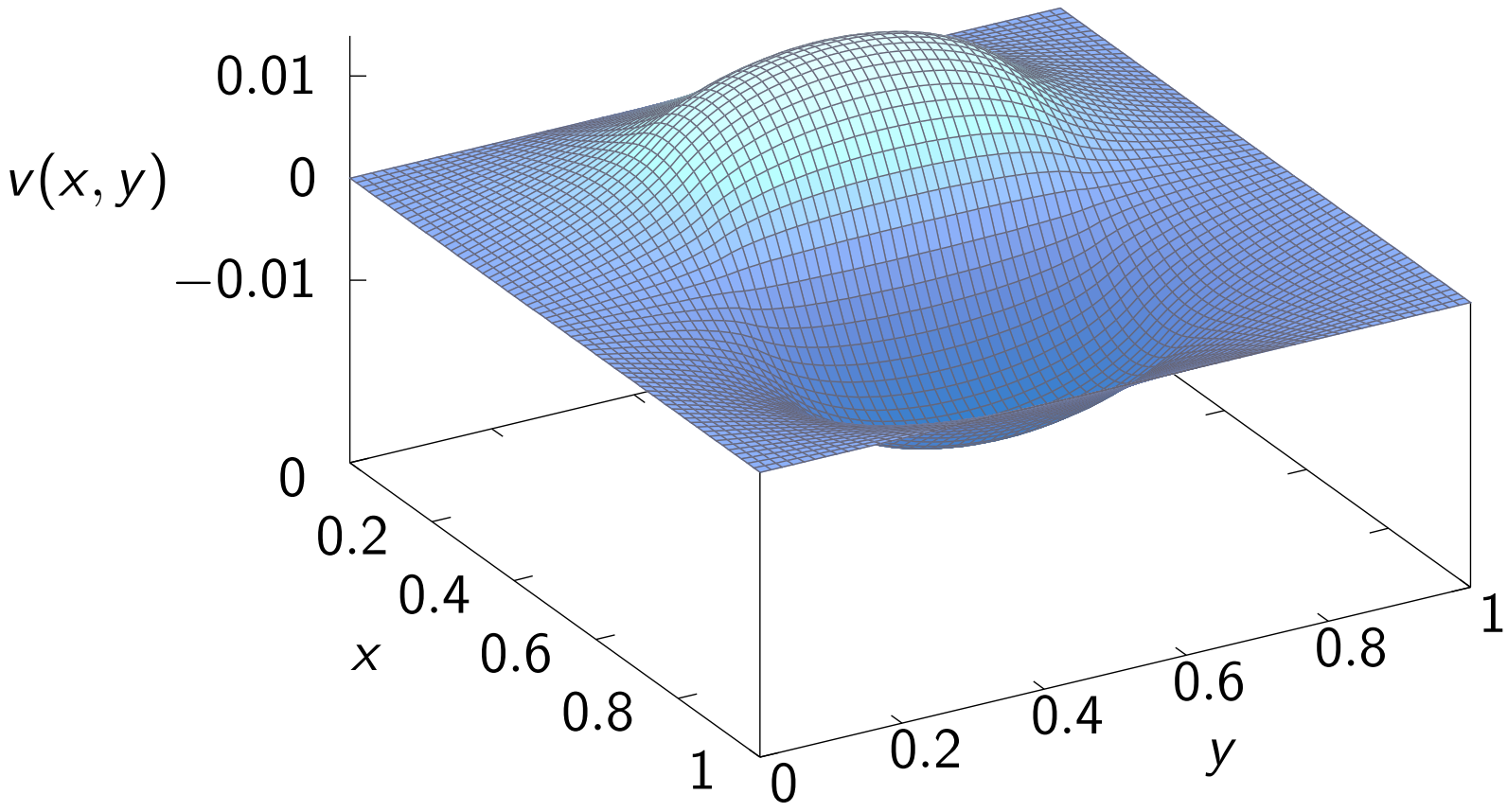
Multiplication be Z is equivalent to the one-dimensional discrete sine transform, and thus can be solved efficiently with FFTW.

Computer demo: solution to the 2D Poisson equation using FFTW.

FFT source term



FFT solution



Testing convergence

The solution method to Poisson problem is based on a second-order finite-difference stencil. We would like to test the actual convergence properties of the solution.

For a general equation and source term, it is difficult to write down an analytical solution.

An approach for cases like this is to use the method of manufactured solutions, by writing down the solution v, and finding the source term that will give it.

Method of manufactured solutions

Propose $v(x, y) = e^x x(1-x)y(1-y)$. This matches the given boundary conditions.

Then

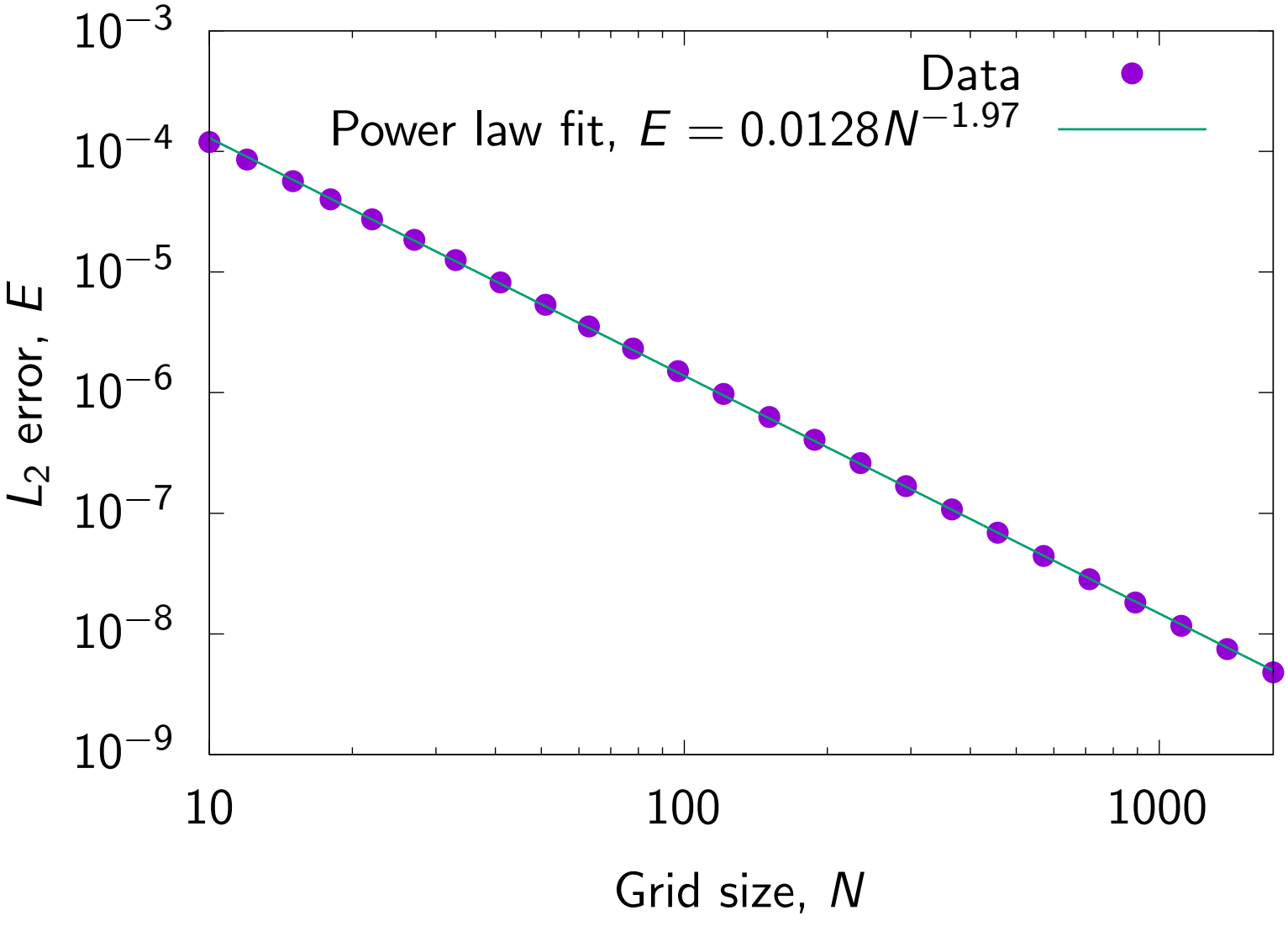
$$\frac{\partial^2 v}{\partial x^2} = e^x x(3+x)(y-1)y,$$
$$\frac{\partial^2 v}{\partial y^2} = 2e^x x(x-1).$$

Hence if

$$f(x, y) = -e^x x(-2 - 3y + 3y^2 + x(2 - y + y^2))$$

then the equation $-\nabla^2 v = f$ is satisfied. The program `pfft_conv.cc` runs a convergence analysis, comparing the numerical solution to this analytical one.

Poisson code convergence



Comments on convergence

$O(h^2)$ convergence is achieved—this is expected since the solution is based on a second-order stencil.

While the choice of $f(x, y)$ is arbitrary, it is helpful to choose something that does not align with simple functions, or known eigenfunctions of the system, to provide a better indication of the typical behavior.

If the solution aligned with an eigenfunction, then the convergence properties may be atypical.

Spectral methods

The fast Fourier transform is also useful in the context of spectral methods, a class of numerical methods for very accurately solving problems that feature smooth solutions.

We will not discuss spectral methods in detail, but we will give a few examples.

We aim to approximate a solution $u(x)$ on some domain by a finite sum $v(x) = \sum_{k=0}^{N} a_k \phi_k(x)$ for some set of functions ϕ_k.

Spectral methods

A spectral method is characterized[20] by the following three characteristics:

1. The approximations $\sum_{k=0}^{N} a_k \phi_k(x)$ should converge rapidly for smooth functions.
2. Given coefficients a_k it should be easy to determine b_k such that
$$\frac{d}{dx}\left(\sum_{k=0}^{N} a_k \phi_k(x)\right) = \sum_{k=0}^{N} b_k \phi_k(x)$$
3. It should be fast to convert between coefficients a_k $(k = 0, \ldots, N)$ and the values for the sum $v(x_j)$ at a set of nodes x_j $(j = 0, \ldots, N)$

[20] B. Fornberg, *A Practical Guide to Pseudopsectral Methods*, Cambridge University Press, 1998.

Spectral methods

Consider the periodic interval $[0, 2\pi)$. Then the complex exponentials $\phi_k(x) = e^{ikx}$ satisfy all three properties.

For a smooth function, the Fourier expansion $v(x) = \sum_{k=0}^{N} a_k e^{ikx}$ converges exponentially.

The derivative $\partial_x v$ has coefficients $b_k = ika_k$.

The fast Fourier transform allows us to convert between node values $v(x_j)$ and coefficients a_k efficiently in $O(N \log N)$ time.

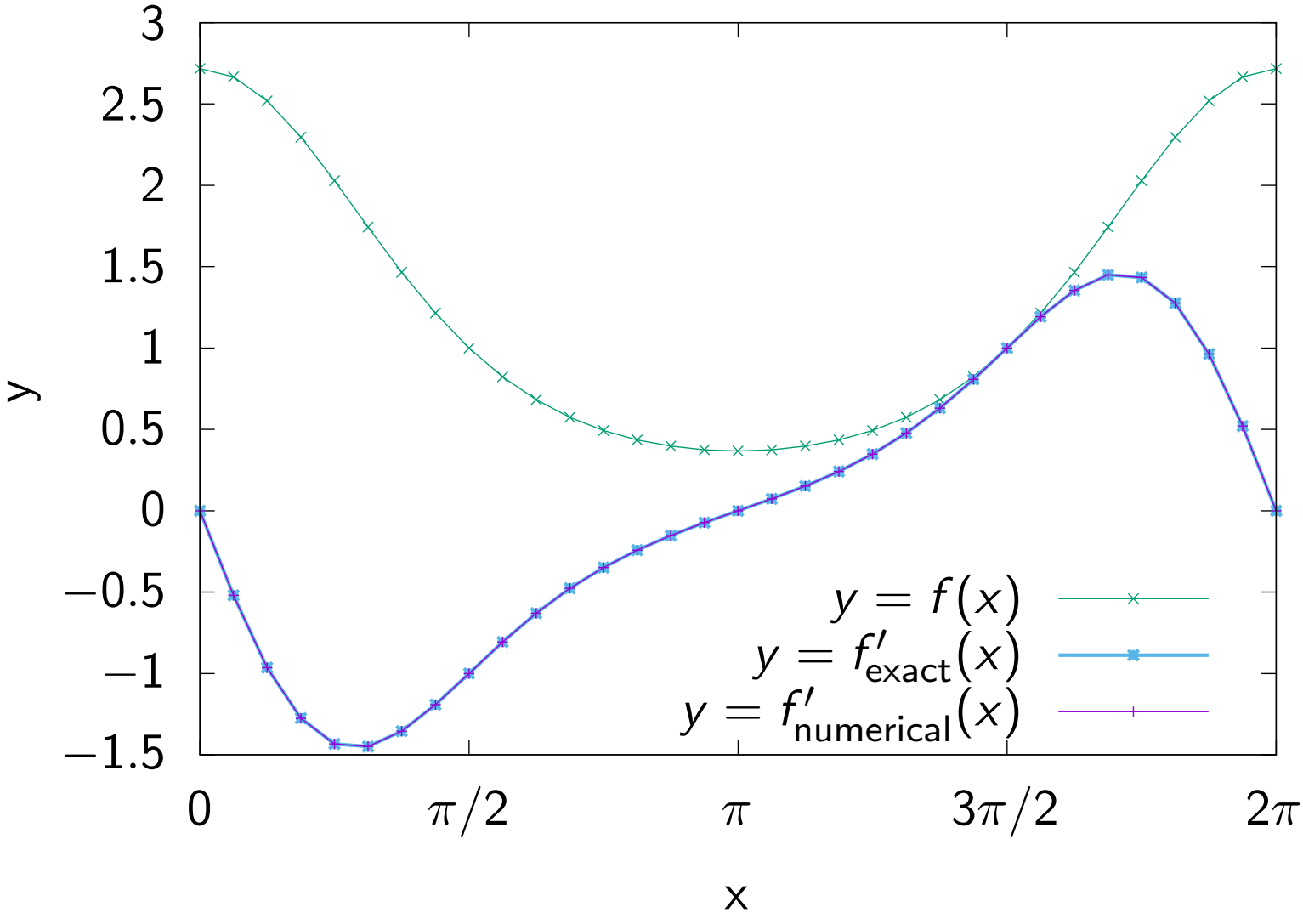
Spectral derivative

Computer demo: calculating the spectral derivative of $f(x) = \exp(\cos x)$

Only 32 points are required to achieve machine epsilon in double precision!

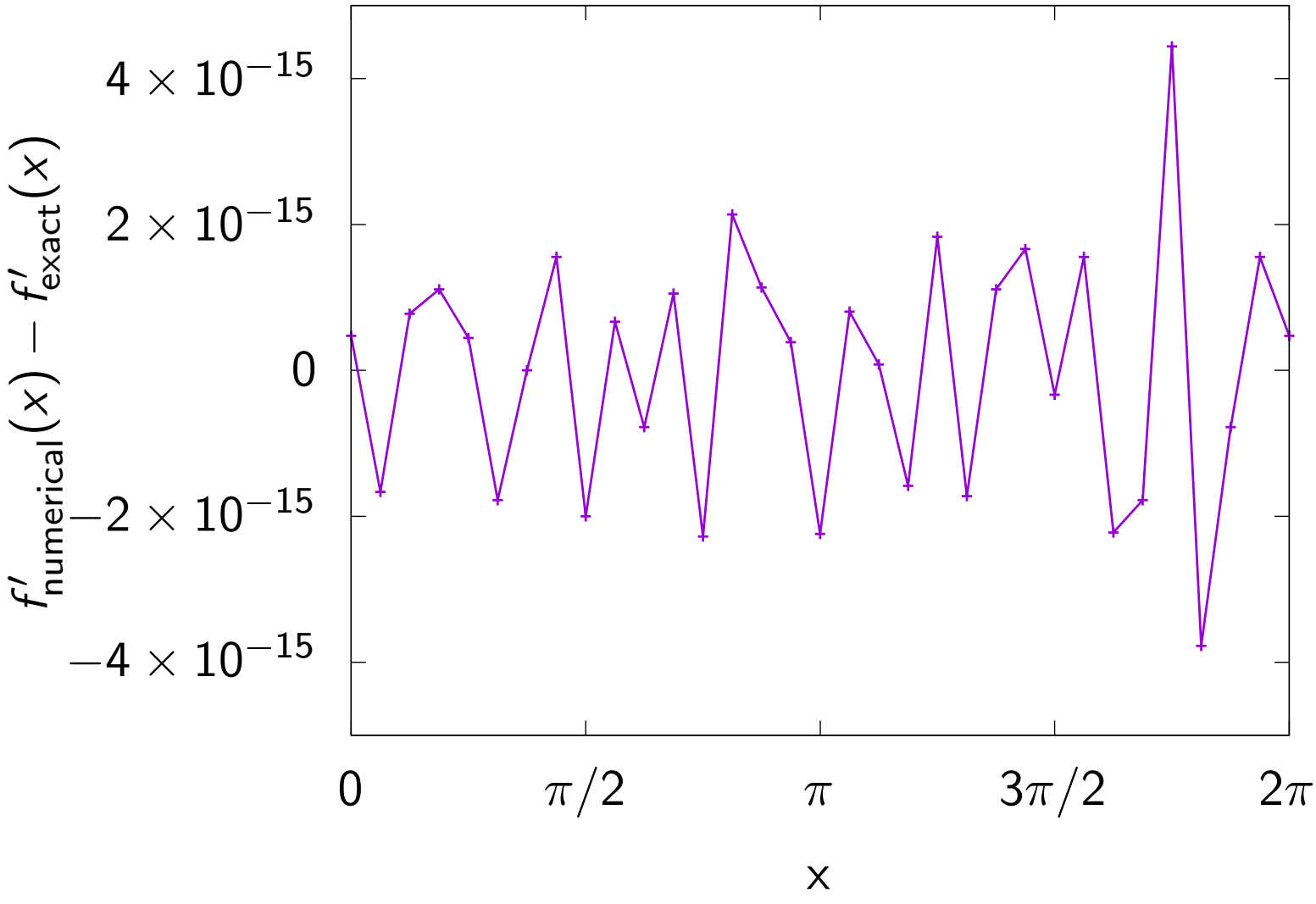
This is far better than typical finite-difference stencils, although it requires smooth periodic functions. If the functions lose regularity, the exponential convergence is lost.

Spectral derivative



Spectral derivative error

Confirming that the errors are on the order of machine epsilon



Solving PDEs with spectral methods

Spectral methods an attractive choice for problems on periodic intervals where smooth solutions are expected. Many nonlinear wave equations have this form.

An example is the Kortweg–de Vries (KdV) equation to model waves on shallow water surfaces. For a function $u(t, x)$ the KdV equation is

$$u_t + uu_x + a^2 u_{xxx} = 0,$$

where a is a constant.[21]

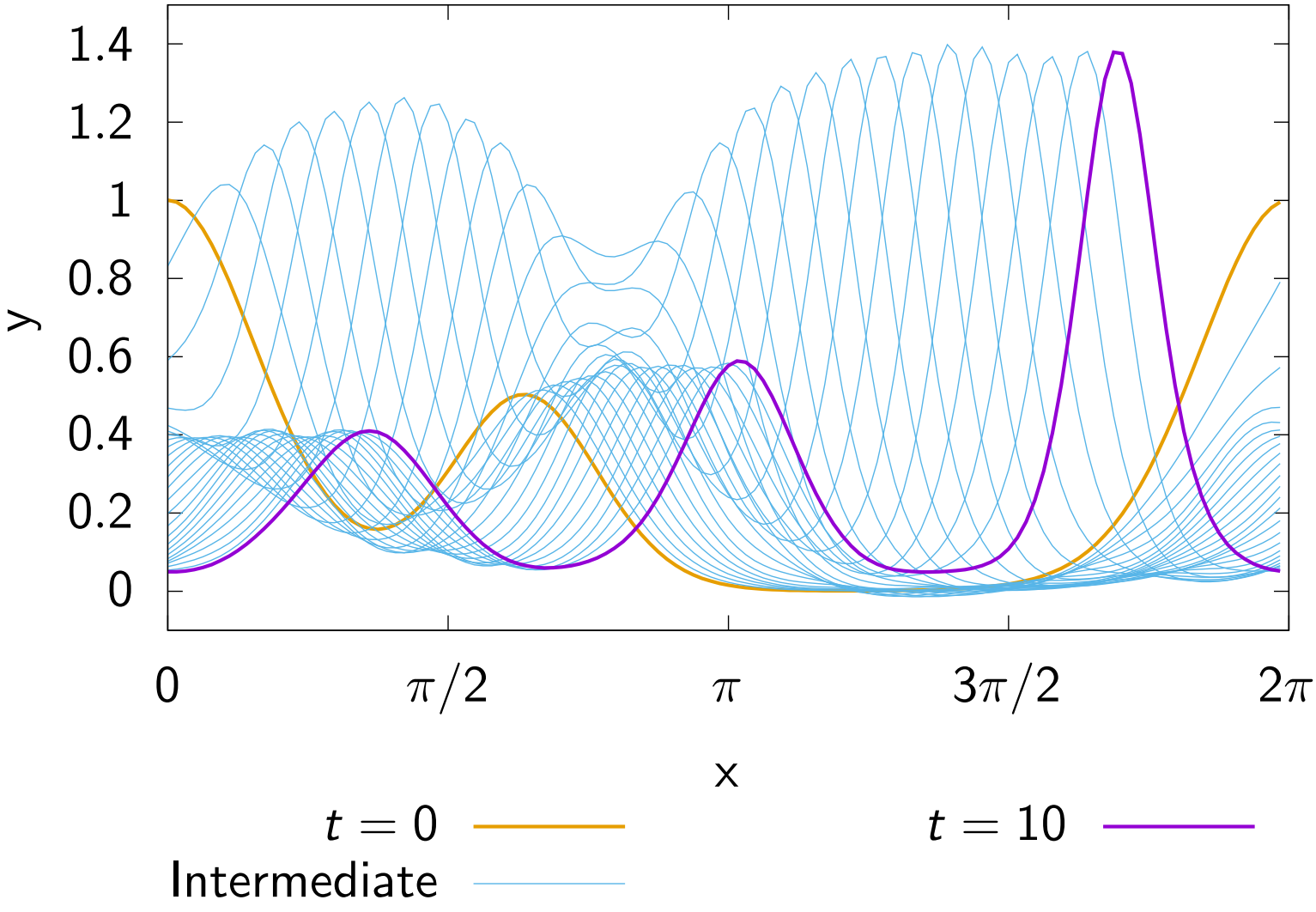
[21] The prefactors in front of the terms are not important, since they can be changed by rescaling t, u, or x.

KdV equation

Computer demo: The program `kdv_test.cc` solves the KdV equation.

It uses fourth-order Runge–Kutta method for timestepping, and spectral methods to evaluate the spatial derivatives. This gives highly accurate solutions.

KdV example solution



Domain decomposition

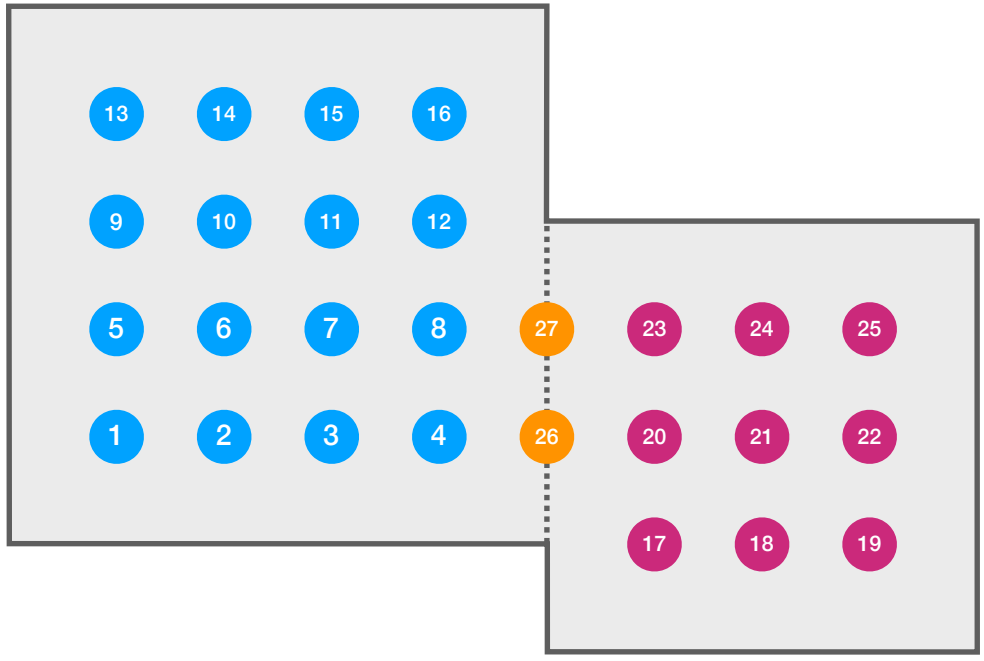
There are a wide variety of different numerical approaches for numerical linear algebra, each with its own strengths. For example:

- BLAS/LAPACK: optimized for dense linear algebra, and based upon direct solution algorithms such as LU, QR, Cholesky, *etc.*
- Krylov methods: well-suited to arbitrary sparse matrix algebra.
- Fast Fourier transform: efficient for high-accuracy problems on structured grids.
- Multigrid: very efficient for sparse linear systems arising in physical PDE problems.

In practice, we often encounter problems that are composed of parts that are suited to different methods, or are too large to fit on a single processor. Domain decomposition allows us to split up a large linear system into components. Additionally, the components may be computed in parallel.

Case I: Non-overlapping grids

Consider solving the Poisson equation with zero Dirichlet boundary coefficients on the domain shown below.



The domain can be decomposed into (1) a square grid with $N = 4$, (2) a square grid with $N = 3$, and (3) two connecting gridpoints.

We know how to solve the Poisson equation on the two square domains using the FFT.

Case I: Non-overlapping grids

Let the solution vector be $v = (v_1, v_2, v_3) \in \mathbb{R}^{27}$ be broken up into the three sets of gridpoints. Let $f \in \mathbb{R}^{27}$ be the corresponding source term.

Write the matrix equation as $Av = h^2 f$ where h is the grid spacing. With the second-order finite difference stencils, A has the form

$$A = \left(\begin{array}{c|c|c} A_{11} & 0 & A_{13} \\ \hline 0 & A_{22} & A_{23} \\ \hline A_{13}^{\mathsf{T}} & A_{23}^{\mathsf{T}} & A_{33} \end{array}\right).$$

Key observation: By construction the A_{12} term is absent and there is no direct coupling between domain 1 and domain 2.

Schur complement

Performing a block LDU decomposition yields

$$A = \begin{pmatrix} I & 0 & 0 \\ 0 & I & 0 \\ A_{13}^{\mathsf{T}} A_{11}^{-1} & A_{23}^{\mathsf{T}} A_{22}^{-1} & I \end{pmatrix} \begin{pmatrix} I & 0 & 0 \\ 0 & I & 0 \\ 0 & 0 & S \end{pmatrix} \begin{pmatrix} A_{11} & 0 & A_{13} \\ 0 & A_{22} & A_{23} \\ 0 & 0 & I \end{pmatrix}$$

where

$$S = A_{33} - A_{13}^{\mathsf{T}} A_{11}^{-1} A_{13} - A_{23}^{\mathsf{T}} A_{22}^{-1} A_{23}$$

is defined as the Schur complement. The inverse of A is

$$A^{-1} = \begin{pmatrix} A_{11}^{-1} & 0 & -A_{11}^{-1} A_{13} \\ 0 & A_{22}^{-1} & -A_{22}^{-1} A_{23} \\ 0 & 0 & I \end{pmatrix} \begin{pmatrix} I & 0 & 0 \\ 0 & I & 0 \\ 0 & 0 & S^{-1} \end{pmatrix} \begin{pmatrix} I & 0 & 0 \\ 0 & I & 0 \\ -A_{13}^{\mathsf{T}} A_{11}^{-1} & -A_{23}^{\mathsf{T}} A_{22}^{-1} & I \end{pmatrix}.$$

Schur complement

In this form, solving the system $Av = h^2 f$ is broken down into simpler components:

- Performing A_{11}^{-1} and A_{22}^{-1}: we can use our previous solvers to compute these operations.
- Multiplying by A_{13} and A_{23}: this is simple to do, especially since these matrices are sparse.
- Performing the inverse S^{-1}: this is the most challenging part. However, S is smaller than the original matrix since it only involves the connecting gridpoints.

Options for inverting the Schur complement

Option 1: Compute S exactly. This can be done by performing the matrix products Se_k for unit vectors e_k in domain 3. Each matrix product requires one solve of A_{11}^{-1} and A_{22}^{-1}. Since S is an SPD matrix, it can be solved efficiently via Cholesky factorization.

Option 2: Use a Krylov subspace method such as conjugate gradient. Since we can efficiently multiply by S, this is attractive. In addition, S generally turns out to be better conditioned than the original matrix.

Generalization to more subdomains

Consider $k > 2$ non-overlapping subdomains, with the boundary gridpoints indexed as $k+1$. The matrix has the form

$$A = \left(\begin{array}{ccc|c} A_{1,1} & & 0 & A_{1,k+1} \\ & \ddots & & \vdots \\ 0 & & A_{k,k} & A_{k,k+1} \\ \hline A_{1,k+1}^{\mathsf{T}} & \cdots & A_{k,k+1}^{\mathsf{T}} & A_{k+1,k+1} \end{array}\right)$$

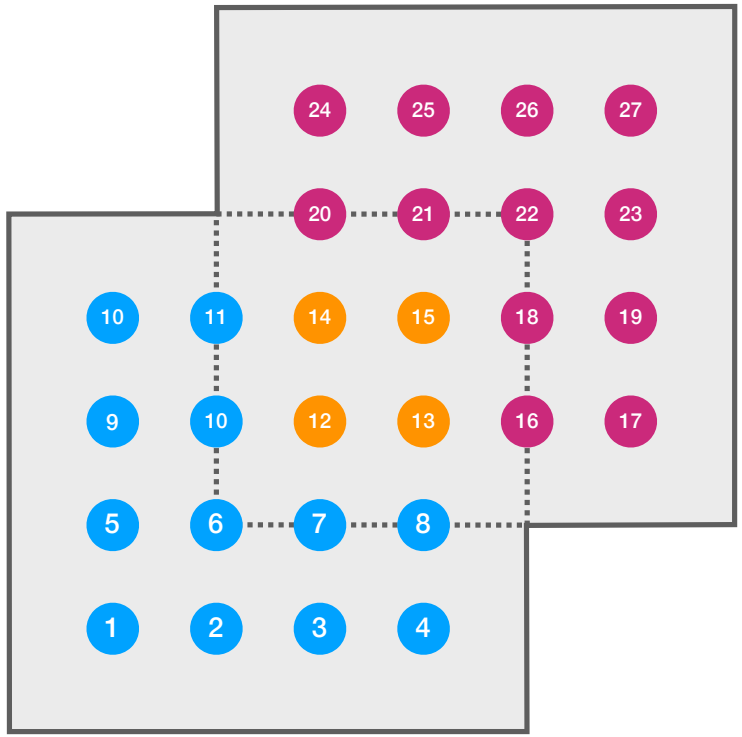
and the Schur complement is

$$S = A_{k+1,k+1} - \sum_{j=1}^{k} A_{j,k+1}^{\mathsf{T}} A_{j,j}^{-1} A_{j,k+1}.$$

Note that many steps (*e.g.* the inversions $A_{j,j}^{-1}$) can be computed in parallel.

Case II: Overlapping domains

Consider solving the Poisson equation with zero Dirichlet boundary conditions on the domain shown below.



This domain consists of two overlapping squares with $N = 4$. It can be broken down into (B) points in the first square but not the second, (C) points common to both squares, and (D) points in the second square but not the first.

Case II: Overlapping domains

The matrix has the form

$$A = \begin{pmatrix} A_{B,B} & A_{B,C} & 0 \\ A_{C,B} & A_{C,C} & A_{C,D} \\ 0 & A_{D,C} & A_{D,D} \end{pmatrix}.$$

The representation can be amalgamated in two ways:

$$A = \begin{pmatrix} A_{BC,BC} & A_{BC,D} \\ A_{D,BC} & A_{D,D} \end{pmatrix} = \begin{pmatrix} A_{B,B} & A_{B,CD} \\ A_{CD,B} & A_{CD,CD} \end{pmatrix}.$$

Here the BC and CD suffixes correspond to the combined gridpoints in those domains. Similarly the solution vector can be decomposed as

$$v = \begin{pmatrix} v_{BC} \\ v_D \end{pmatrix} = \begin{pmatrix} v_B \\ v_{CD} \end{pmatrix}.$$

Iterative Schwarz methods

For overlapping domains, we introduce two different iterative approaches. Define $b = h^2 f$ to the be source term and consider solving $Av = b$.

Given a solution v_i, the additive Schwarz method proceeds as follows to obtain a better answer v_{i+1}:

1. Calculate $r = b - Av_i$.
2. Calculate $w = A^{-1}_{BC,BC} r_{BC}$.
3. Calculate $x = A^{-1}_{CD,CD} r_{CD}$.
4. Define the new solution[22] as

$$v_{i+1} = v_i + \begin{pmatrix} w_B \\ (w_C + x_C)/2 \\ x_D \end{pmatrix}.$$

[22] Note that this is a bit different to the *Applied Numerical Linear Algebra* textbook. In the C domain, Demmel takes a sum instead of an average. I have found an average leads to better convergence.

Comments on the additive Schwarz method

Note that steps 2 and 3 can be calculated efficiently using the fast solvers on the square grids. Steps 2 and 3 can also be done in parallel.

In step 3 of the additive Schwarz method, we use the original residual r from step 1, even though we have new information from doing the solve on the BC in step 2. This suggests a modified approach.

The multiplicative Schwarz method

The multiplicative Schwarz method recomputes the residual before doing the solve on CD.[23] This leads to the following iteration:

1. $r_{BC} = (b - Av_i)_{BC}$
2. $v_{i+\frac{1}{2}} = v_i + A^{-1}_{BC,BC} r_{BC}$
3. $r_{CD} = (b - Av_{i+\frac{1}{2}})_{CD}$
4. $v_{i+1} = v_{i+\frac{1}{2}} + A^{-1}_{CD,CD} r_{CD}$

This typically gives better performance than the additive Schwarz method. However, the ability to parallelize in a straightforward manner is lost, since the second solve $A^{-1}_{CD,CD}$ incorporates information from the first.

[23] This is similar to the logic behind moving from the Jacobi method to the Gauss–Seidel method. See the AM205 notes.