Lecture 4

James Camacho

Covering:

- Finite element methods,
- Various bases,
- Fourier/DCT transforms,
- Fourier analysis (faster solvers),
- Fourier analysis (stability).

Finite Element Methods

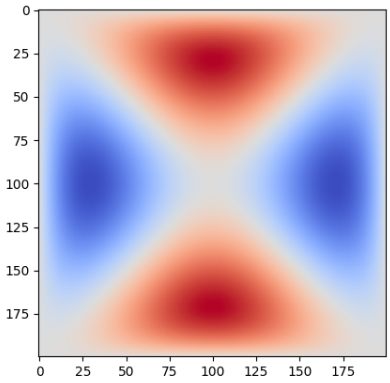

Figure 1: Finite element example with the Chebyshev polynomials.

In finite element methods, you approximate a function f as a sum of trial functions,

$$f(x) \approx \sum_{j=1}^{n} c_j \Phi_j,$$

then test the approximation using test functions $\{\Psi_i\}_{i=1}^{m}$. The test is whether

$$\int_0^1 \sum_{j=1}^{n} c_j \Phi_j(x) \Psi_{\text{test}}(x) \mathrm{d}x = \int_0^1 f(x) \Psi_{\text{test}}(x) \mathrm{d}x$$

for every test function. To save room, mathematicians use the notation $\langle u, \Psi \rangle$ to refer to the inner product

$$\int_0^1 f(x) \Psi(x) \mathrm{d}x.$$

Sometimes it's useful to insert a nonnegative weight w, so

$$\langle f, \Psi \rangle_w = \int_0^1 f(x) \Psi(x) w(x) \mathrm{d}x.$$

Writing the test equations in matrix form gives

$$\begin{bmatrix} \langle \Psi_1, \Phi_1 \rangle & \langle \Psi_1, \Phi_2 \rangle & \cdots & \langle \Psi_1, \Phi_n \rangle \\ \langle \Psi_2, \Phi_1 \rangle & \langle \Psi_2, \Phi_2 \rangle & \cdots & \langle \Psi_2, \Phi_n \rangle \\ \vdots & \vdots & \ddots & \vdots \\ \langle \Psi_m, \Phi_1 \rangle & \langle \Psi_m, \Phi_2 \rangle & \cdots & \langle \Psi_m, \Phi_n \rangle \end{bmatrix} \begin{bmatrix} c_1 \\ c_2 \\ \vdots \\ c_n \end{bmatrix} = \begin{bmatrix} \langle \Psi_1, f \rangle \\ \langle \Psi_2, f \rangle \\ \vdots \\ \langle \Psi_m, f \rangle \end{bmatrix},$$

or in bra-ket notation,

$$\langle \Psi | \langle \Phi | \mathbf{c} \rangle | w \rangle = \langle \Psi | f | w \rangle.$$

To solve the Poisson equation ($\nabla^2 u = f$), we can approximate u as a sum of basis functions, and solve

$$\langle \Psi | \langle \nabla^2 \Phi | \mathbf{c} \rangle | w \rangle = \langle \Psi | f | w \rangle.$$

Then $u \approx \langle \Phi | \mathbf{c} \rangle$.

Usually the test and trial bases are the same. Also, we can solve the system faster if the matrix has a small bandwidth; in particular orthogonal functions ($\langle \Phi_i, \Phi_j \rangle = 0$ for $i \neq j$) will give a diagonal system. In addition, it's good if they are easy to integrate and differentiate. Some common bases include hat functions, Legendre polynomials, and the Chebyshev polynomials.

For large systems, you divide the structure into a mesh and choose a basis for each cell (usually the same basis but shifted over). Outside its cell each function is defined to be zero so you only need to take inner products between functions within a cell. To keep it smooth across cells, you create boundary conditions, evaluating the function and its derivatives at the boundary. The Bernstein bases are particularly good for this, as although they are not orthogonal most of them vanish on the boundary (source).

Common Bases

Hat Functions

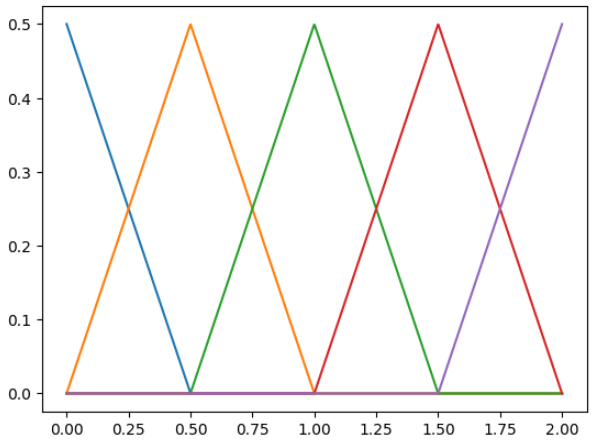

Figure 2: Hat Basis

The hat functions have a peak at the center of their cell. An equivalent basis would put a line with positive slope and a line with negative slope in each cell, so the hat functions are also called the "piecewise linear basis". We can extend them to higher dimensions using a tensor product,

$$\Phi^{2D} = \Phi^{1D} \otimes \Phi^{1D} \iff \Phi^{2D}_{ij} = \Phi^{1D}_i \Phi^{1D}_j.$$

Here I've shown the top half of some two dimensional hat functions (the pyramids should continue down after intersection):

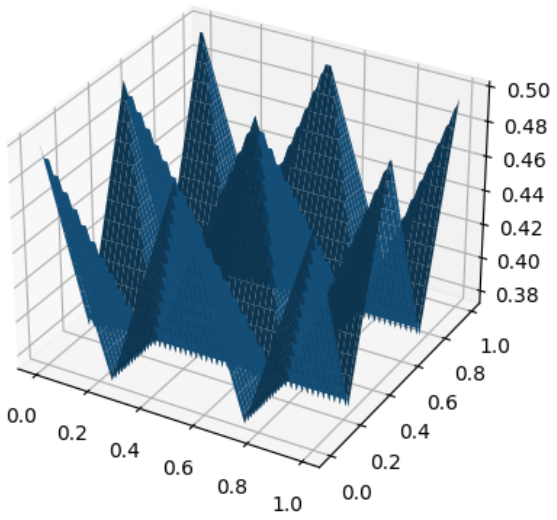

Figure 3: Two Dimensional Hat Basis

There is no weight for these bases, i.e. $w \equiv 1$. For solving the Poisson equation, the second derivative is zero, so

$$\langle \Psi | \langle \nabla^2 \Phi | \mathbf{c} \rangle | 1 \rangle = \langle \Psi | f | 1 \rangle$$

looks like it'll become

$$0 = \langle \Psi | f | 1 \rangle,$$

which doesn't really work. Luckily, the product rule comes to the rescue. We have

$$\nabla \cdot (\Psi \nabla \Phi) = \nabla \Psi \cdot \nabla \Phi + \Psi \nabla^2 \Phi \implies \int_0^1 \Psi \nabla^2 \Phi \mathrm{d}x = \Psi \nabla \Phi \Big|_0^1 - \int_0^1 \nabla \Psi \cdot \nabla \Phi \mathrm{d}x.$$

As

$$\Psi(0) = \Psi(1) = 0,$$

this reduces to

$$\langle \Psi, \nabla^2 \Phi \rangle = -\langle \nabla \Psi, \nabla \Phi \rangle.$$

This trick is often called integration by parts. We need to solve

$$-\langle \nabla \Psi | \langle \nabla \Phi | \mathbf{c} \rangle | w \rangle = \langle \Psi | f | 1 \rangle.$$

In one dimension we find

$$-\langle \Psi_i', \Phi_j' \rangle = \begin{cases} -1 & i = j \\ 0.5 & |i - j| = 1 \\ 0 & \text{otherwise.} \end{cases}$$

The finite element matrix ends up being the exact same as the finite difference one, times a constant. This happens in general for all dimensions, e.g. the two dimensional hat functions will give rise to the nine-point stencil finite difference matrix. The only difference from the finite difference equation is the right hand side, where we convolve f with the hat function rather than taking a single point.

Legendre Polynomials

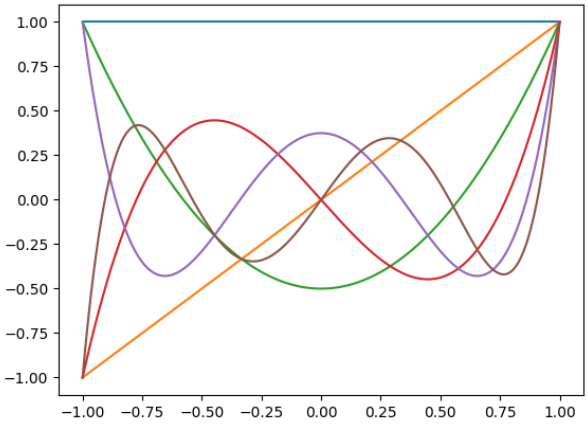

Figure 4: First Six Legendre Polynomials

Again, no weight for this basis, although the bounds on the integral go from minus one to one:

$$\langle P_i, P_j \rangle = \int_{-1}^{1} P_i(x) P_j(x) \mathrm{d}x.$$

The Legendre polynomials are orthogonal, and can be computed via Bonnet's recursion formula:

$$P_0 = 1; \quad P_1 = x; \quad P_{n+1} = \frac{(2n+1)xP_n - nP_{n-1}}{n+1}.$$

Due to orthogonality, the approximation formula

$$\langle \Psi | \langle \Phi | \mathbf{c} \rangle | w \rangle = \langle \Psi | f | w \rangle,$$

reduces to

$$c_i = \frac{\langle P_i, f \rangle}{\langle P_i, P_i \rangle}.$$

Solving the Poisson equation would give a much denser matrix.

Chebyshev Polynomials

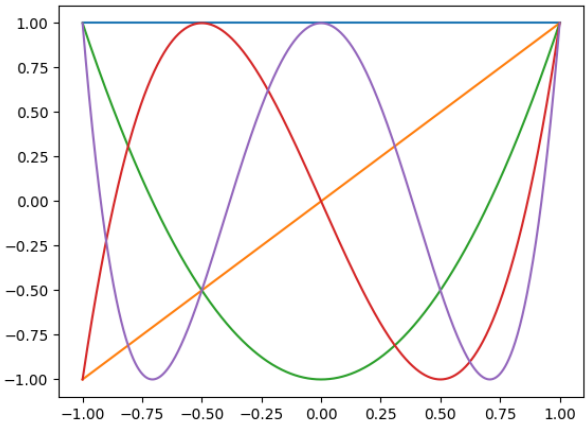

Figure 5: First Five Chebyshev Polynomials

The Chebyshev polynomials are defined by

$$T_n(\cos(x)) = \cos(nx),$$

and satisfy the recurrence

$$T_0 = 1; \quad T_1 = x; \quad T_{n+1} = 2xT_n - T_{n-1}.$$

Under the inner product

$$\langle T_i, T_j \rangle = \int_{-1}^{1} \frac{T_i(x)T_j(x)}{\sqrt{1-x^2}} \mathrm{d}x$$

(i.e. $w(x) = 1/\sqrt{1-x^2}$) we get

$$\langle T_i, T_j \rangle = \begin{cases} \pi & i = j = 0 \\ \pi/2 & i = j \neq 0 \\ 0 & \text{otherwise,} \end{cases}$$

so the approximation formula would reduce to

$$c_0 = \frac{1}{\pi}\langle f, 1 \rangle; \quad c_i = \frac{2}{\pi}\langle f, T_i \rangle.$$

Integrating and multiplying together Chebyshev polynomials is also pretty quick:

$$\int T_n \mathrm{d}x = \frac{1}{2}\left(\frac{T_{n+1}}{n+1} - \frac{T_{n-1}}{n-1}\right),$$

$$T_a T_b = \frac{1}{2}(T_{a+b} + T_{a-b}).$$

Unfortunately, taking derivatives is not so nice, though it can be sped up with the discrete cosine transform (e.g. see chebfun).

Fourier Transforms

Euler's Formula

Every positive number has two square roots, $+\sqrt{x}$ and $-\sqrt{x}$. When Cardano discovered the cubic formula, he found expressions involving the root of a negative number. He tried inventing a number $i = \sqrt{-1}$, plugged it in, and got out the correct roots.

Two hundred years later, de Moivre noticed every complex number could be written in *polar form*,

$$a + bi = r(\cos\theta + i\sin\theta) = f(r, \theta).$$

Multiplying two complex numbers resulted in multiplying their magnitudes and adding their angles, $f(r_1, \theta_1) \cdot f(r_2, \theta_2) = f(r_1 r_2, \theta_1 + \theta_2)$. This implies you could write it as

$$f(r, \theta) = r \cdot \text{base}^{\theta}$$

for some base. Taking a derivative with respect to θ gives

$$f'(r, \theta) = f(r, \theta) \ln \text{base},$$

and a picture reveals this is perpendicular to f.

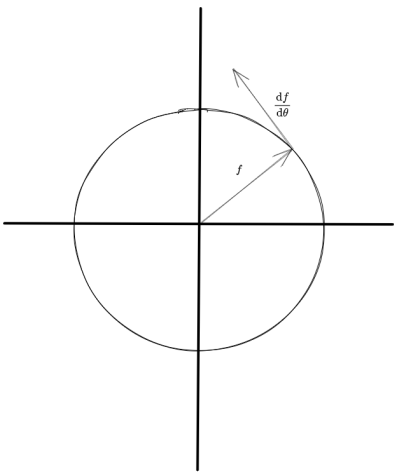

Figure 6: f describes a circle in the complex plane.

Therefore, $\ln \text{base} = f(1, 90°) = i$. Plugging this back in, we get Euler's formula,

$$\cos\theta + i\sin\theta = e^{i\theta}.$$

In particular,

$$\cos\theta = \Re(e^{i\theta}) = \frac{e^{i\theta} + e^{-i\theta}}{2} \quad \text{and} \quad \sin\theta = \Im(e^{i\theta}) = \frac{e^{i\theta} - e^{-i\theta}}{2i}.$$

Most of the properties of the Chebyshev polynomials can be derived using these formulas.

Fourier Transforms

The fourier transform takes an arbitrary function $f(t)$ and produces another function in the "frequency domain". It's defined as

$$F(y) = \int_{-\infty}^{\infty} f(x)e^{-2\pi i \cdot xy}\mathrm{d}x,$$

and is sometimes written $f \leftrightarrow F$. The inverse transform is the complex conjugate, replacing every i with $-i$. The cosine transform is the real part, and the sine transform is the imaginary part.

The discrete versions use a summation rather than an integral:

$$F_y = \frac{1}{\sqrt{N}} \sum_{x=0}^{N-1} e^{-\frac{2\pi i}{N}xy} f_x,$$

sending N points $f_0, f_1, \ldots, f_{N-1}$ in the time domain to N points $F_0, F_1, \ldots, F_{N-1}$ in the frequency domain. Written out as a matrix multiplication, this is

$$\begin{bmatrix} F_0 \\ F_1 \\ F_2 \\ F_3 \\ \vdots \\ F_{N-1} \end{bmatrix} = \frac{1}{\sqrt{N}} \begin{bmatrix} 1 & 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega & \omega^2 & \omega^3 & \cdots & \omega^{N-1} \\ 1 & \omega^2 & \omega^4 & \omega^6 & \cdots & \omega^{2(N-1)} \\ 1 & \omega^3 & \omega^6 & \omega^9 & \cdots & \omega^{3(N-1)} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega^{N-1} & \omega^{2(N-1)} & \omega^{3(N-1)} & \cdots & \omega^{(N-1)(N-1)} \end{bmatrix} \begin{bmatrix} f_0 \\ f_1 \\ f_2 \\ f_3 \\ \vdots \\ f_{N-1} \end{bmatrix},$$

where $\omega = e^{-\frac{2\pi i}{N}}$ is a root of unity (i.e. $\omega^N = 1$). Typically such a matrix-vector multiplication takes $O(N^2)$ time to compute, however we have a lot of repeating values in this matrix. The Cooley-Tukey algorithm takes advantge of this to compute it in only $O(N \log N)$ time.

Let D be the discrete fourier transform "operator" (i.e. matrix). Split up every other element of f into two separate components, f_{even} and f_{odd}. You may need to pad it with an extra zero to make them the same length. Now look at

$$\begin{aligned} F_y &= D(f_{\text{even}}) + (\omega^y D)(f_{\text{odd}}), \\ F_{y+\frac{N}{2}} &= D(f_{\text{even}}) + (\omega^{y+\frac{N}{2}} D)(f_{\text{odd}}) = D(f_{\text{even}}) - (\omega^y D)(f_{\text{odd}}). \end{aligned}$$

You can do two smaller fourier transforms ($\frac{N}{2} \times \frac{N}{2}$ matrices) and add/subtract them to get a larger transform! Recursing gives the Cooley-Tukey algorithm. One use is multiplying two polynomials together: you can evaluate each polynomial at the roots of unity, multiply the values together, then apply the inverse fourier transform to get the coefficients of their product.

Fourier Analysis

Finite Difference Speedup

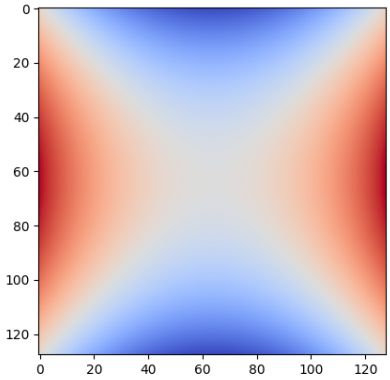

Figure 7: Solving the Poisson equation with a fourier transform.

Similar to the Taylor series and finite element methods, we can write continuous and periodic functions as a Fourier series:

$$u(x) = \sum_{n=-\infty}^{\infty} U_n e^{2\pi i n \cdot x}$$

for some coefficients U_n. Note, this is the same as a finite element approximation using the functions $e^{\pm 2\pi i n \cdot x}$, or equivalently, $\cos(2\pi i n \cdot x), \sin(2\pi i n \cdot x)$ due to Euler's formula. We can use the inner product

$$\langle e^{2\pi i n \cdot x}, e^{2\pi i m \cdot x} \rangle = \int_0^1 \overline{e^{2\pi i n \cdot x}} \cdot e^{2\pi i m \cdot x} \mathrm{d}x = \begin{cases} 1 & m = n, \\ 0 & \text{otherwise.} \end{cases}$$

Nearly everything will cancel out, so we can find an explicit formula for U_n:

$$U_n = \int_0^1 u(x) e^{-2\pi i n \cdot x} \mathrm{d}x.$$

The discrete case and two-dimensional case is similar. We can use this to quickly solve the discrete Poisson equation. For example, if

$$u(x, y) = \sum_{n=0}^{N-1} \sum_{m=0}^{M-1} U_{n,m} e^{2\pi i (\frac{m}{M} x + \frac{n}{N} y)},$$

then the five-point stencil finite difference method is

$$\begin{aligned} f = \nabla^2 u &\approx MN\left[-4u_{0,0} + u_{1,0} + u_{-1,0} + u_{0,1} + u_{0,-1}\right] \\ &= MN \sum_{n=0}^{N-1}\sum_{m=0}^{M-1}\left[e^{\frac{2i\pi}{M}x} + e^{-\frac{2i\pi}{M}x} + e^{\frac{2i\pi}{N}y} + e^{-\frac{2i\pi}{N}y} - 4\right] U_{n,m} e^{2\pi i(\frac{m}{M}x + \frac{n}{N}y)} \\ &= MN \sum_{n=0}^{N-1}\sum_{m=0}^{M-1}\left[2\cos\left(\frac{\pi x}{M}\right) + 2\cos\left(\frac{\pi y}{N}\right) - 4\right] U_{n,m} e^{2\pi i(\frac{m}{M}x + \frac{n}{N}y)} \end{aligned}$$

Taking a 2D discrete fourier transform of f and dividing the coefficients by

$$MN\left[2\cos\left(\frac{2\pi x}{M}\right) + 2\cos\left(\frac{2\pi y}{N}\right) - 4\right] = 4MN\left[\sin^2\left(\frac{\pi x}{M}\right) + \sin^2\left(\frac{\pi y}{N}\right)\right]$$

will give us the coefficients U. Then we can take the inverse transform to recover u. The periodicity of the Fourier series means it will solve it with wrapped boundary conditions. The overall running time goes from $O(N^3)$ to $O(N^2 \log N)$.

Stability Analysis

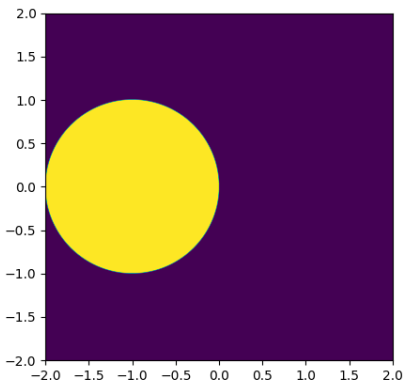

Figure 8: The stability region for the forward euler method.

The Stone-Weierstrass theorem, says any real-valued, continuous function can be written as an infinite sum of polynomials, in particular the Chebyshev polynomials. Making the change of variables $x \to \cos t$ makes the function periodic, but keeps it continuous. It would then become a sum of cosines, i.e. $\sum_\omega e^{i\omega t} + e^{-i\omega t}$. In general, any function that is continuous and periodic has a convergent Fourier series; note it doesn't have to be real-valued.

We can use the Fourier series to find the stability regions of Euler-like methods. If every function $y(t) = e^{\xi t}$ stays stable (note: $\xi = i\omega$), then so will any other continuous function. For example, with the trapezoidal method we have

$$\begin{aligned} y_{t+1} &= y_t + \frac{h}{2}[y'(t) + y'(t+1)] \\ &= y_t + \frac{h}{2}[\xi y_t + \xi y_{t+1}] \\ &\iff \\ y_{t+1} &= \frac{2 + h\xi}{2 - h\xi} y_t. \end{aligned}$$

To prevent y from blowing up we need

$$\left|\frac{2 + h\xi}{2 - h\xi}\right| \leq 1.$$

This ends up being equivalent to $\Re(h\xi) < 0$, so any step size h is fine as long as the function doesn't increase exponentially.

We can do a similar stability analysis on finite difference methods. Suppose we use Euler's method and the five-point stencil to solve the heat equation,

$$u^{t+1} = u^t + hMN[-4u^t_{0,0} + u^t_{1,0} + u^t_{-1,0} + u^t_{0,1} + u^t_{0,-1}].$$

Look at one coefficient of the Fourier series of u. From above, we know that

$$\nabla^2 u \leftrightarrow 4MN\left[\sin^2\left(\frac{\pi x}{M}\right) + \sin^2\left(\frac{\pi y}{N}\right)\right] U_{x,y}.$$

To keep the coefficients from increasing exponentially, we need

$$\left|1 + 4hMN\left[\sin^2\left(\frac{\pi x}{M}\right) + \sin^2\left(\frac{\pi y}{N}\right)\right]\right| \leq 1,$$

or equivalently,

$$h \leq \frac{1}{2MN\left[\sin^2\left(\frac{\pi x}{M}\right) + \sin^2\left(\frac{\pi y}{N}\right)\right]}.$$

In particular, you can choose $x = \frac{M}{2}, y = \frac{N}{2}$ and

$$\sin^2\left(\frac{\pi x}{M}\right) + \sin^2\left(\frac{\pi y}{N}\right) = 2,$$

so the method will always be stable iff

$$h \leq \frac{1}{4MN}.$$

If we choose this optimal time step, we have to perform Euler's method $O(MN)$ times to go from $t = 0$ to $t = 1$. Each time step involves $O(MN)$ operations, so the overall running time would be $O(M^2N^2)$. Usually $M = N$, so this would take $O(N^4)$ time, which is not very fast, especially for an $O(N^{-2})$ error term.

Homework Problems

1. **Coding:** Write a finite element method to approximate an arbitrary f as a sum of Chebyshev polynomials:

$$f(x) \approx \sum_{n=0}^{N} c_n T_n(x).$$

   Using the finite element method involves taking integrals, which you can use Chebyshv-Gauss quadrature for. This will make the whole running time $O(N^2)$. Can you find a faster algorithm that runs in only $O(N \log N)$ time?
2. **Math + Coding:** Code up a finite element method to solve the wave equation,

$$\frac{\partial^2 u}{\partial t^2} = \nu^2 \frac{\partial^2 u}{\partial x^2}$$

   where ν is the speed of the wave. Use the trial and test bases

$$\Psi = \Phi = \{\sin n(x - \nu t), \cos n(x - \nu t)\}_{n=0}^{N}.$$

3. **Math:** Perform a stability analysis on the Crank-Nicolson method applied to the discrete Poisson equation. It is essentially the trapezoidal method in time with the five-point stencil in space.
4. **Math:** In cryptography, the RSA algorithm involves modular arithmetic modulo N, where N is the product of two large primes. Being able to efficiently factor N would break the algorithm, but alas there no classical algorithms that can do so in polynomial time (in the number of bits of N). However, quantum computers can perform a Fourier transform in $O(n^2)$ time, where n is the number of quantum bits, much faster than the classical $O(n2^n)$ running time. Suggest an algorithm that uses the quantum Fourier transform to factor N in polynomial time. *Hint:* Euler's theorem may prove useful.