Ch. 8 - Finite Difference Methods

Make the leap from:

- Ordinary Differential Equations (ODEs): 1 independent variable
- **Partial Differential Equations (PDEs)**: >1 independent variable
- $\S 8.1$: Start by applying finite difference methods similar to those introduced in Ch. 7
- $\S 8.2$: Consider iterative solution of discretized equations
- $\S\S 8.3 - 8.5$: Consider alternative approach based on Fourier transforms
- Stepping methods covered in Ch. 9

Before getting into the details, quickly review classification of PDEs

Start with PDEs common in engineering: General **Second-order Quasi-linear PDE**

$$A(x,y)\frac{\partial^2 u}{\partial x^2} + 2B(x,y)\frac{\partial^2 u}{\partial x \partial y} + C(x,y)\frac{\partial^2 u}{\partial y^2} = F(x,y,u,\frac{\partial u}{\partial x},\frac{\partial u}{\partial y})$$

Typical notations:

- Use $x, y, z, \text{and } t$ as independent variables
- Designate dependent variable as u, v, w, ψ, or ω
- Use subscripts (sometimes with a comma) as shorthand for partial derivatives
- Equation above abbreviates to:

$$A(x,y)\,u_{xx} + 2B(x,y)\,u_{xy} + C(x,y)\,u_{yy} = F(x,y,u,u_x,u_y)$$

- **Constant coefficients** $\implies$ PDE further simplifies to:

$$Au_{xx} + 2Bu_{xy} + Cu_{yy} = F(x,y,u,u_x,u_y)$$

Usual approach to classification (in addition to determining order)

- Focus on highest (2^{nd}) order derivatives
- Consider **characteristic solutions** of the form $u = f(kx + y)$.
  - Chain rule $\implies$ $u_x = kf', u_y = f'$
  - Plug into PDE:

$$Ak^2 f'' + 2Bkf'' + Cf'' = 0$$

  - Cancel f' leaves quadratic equation to solve for k:

$$Ak^2 + 2Bk + C = 0$$

  - Major distinction of interest involves whether this quadratic has real roots.

- When $Ak^2 + 2Bk + C = 0$ has real roots k_1, k_2:
- Values of $u(kx + y)$ are constant along **real characteristic curves (or lines or directions)** where $k_i x + y = 0$
- Information propagates along real characteristics
- "propagates" $\iff$ wave behavior
- Existence of real characteristics determined by the **discriminant** $B^2 - AC$

Classification summary for $Au_{xx} + 2Bu_{xy} + Cu_{yy} = F(x, y, u, u_x, u_y)$:

Type	Discriminant Value	Classic Example	Classic Equation	Normal Form Equation
Hyperbolic	$B^2 - AC > 0$	Wave	$u_{tt} = c^2 u_{xx}$	$u_{vw} = f_1$
Parabolic	$B^2 - AC = 0$	Heat	$u_t = c^2 u_{xx}$	$u_{ww} = f_2$
Elliptic	$B^2 - AC < 0$	Laplace	$u_{xx} + u_{yy} = 0$	$u_{vv} + u_{ww} = f_3$

Different classes of equations have different properties that require choosing appropriate numerical methods.

Section 8.1 - Finite Difference Discretization

This section opens with a non-trivial example, but let's begin with simpler "classics".

Start with **Laplace's equation**:

- Arises in a variety of applications: gravitation, electrostatics, fluid flow (just about anything with a potential)
- Elliptic
- Allows a straightforward iterative solution
- Model of diffusion $\Longrightarrow$ temperature, chemical concentrations etc. evolve toward local average to reduce gradients

Boundary Conditions (BCs)

Start simple with Cartesian coordinates and Dirichlet boundary conditions (BCs).

> Dirichlet BCs $\iff$ specify value of dependent variable on boundary
> Neumann BCs $\iff$ specify value of derivative on the boundary

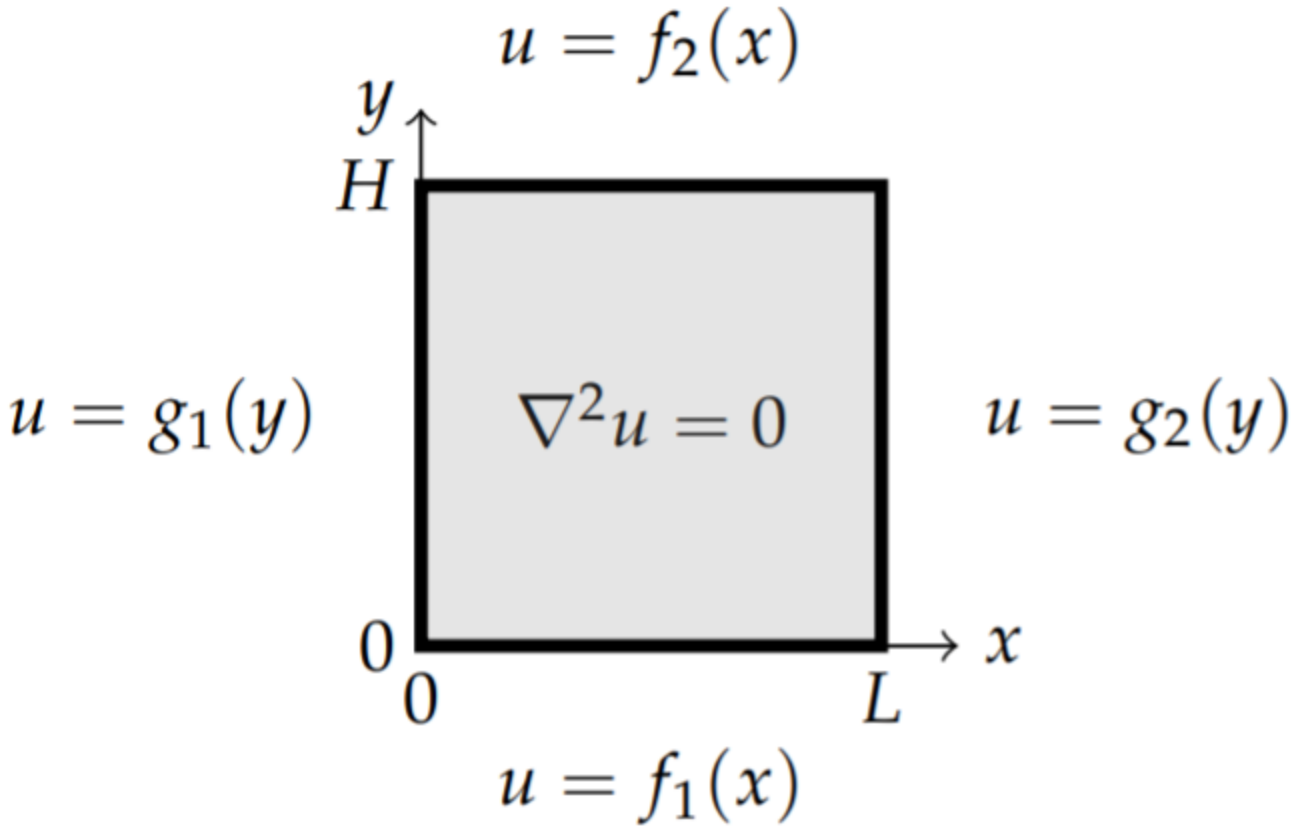
Compute a numerical solution for the following problem:

$$\nabla^2 u = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = 0 \text{ on } x \in [0, L_x];\ y \in [0, L_y]$$

with boundary conditions:

$$u(0, y) = u(L_x, y) = u(x, 0) = 0,\ u(x, L_y) = U$$

Sketch the domain with given information:



We have $L_x = L, L_y = H, f_1 = g_1 = g_2 = 0$ and $f_2 = U$

Quick review of analytic solution by Separation of Variables

Look for solution of $u_{xx} + u_{yy} = 0$ of the form $u(x, y) = X(x)Y(y)$.

Substitute into PDE $\rightarrow X''Y + XY'' = 0$

Move one term to other side and divide through by

$$XY \rightarrow \frac{XY''}{XY} = -\frac{X''Y}{XY} \rightarrow \frac{Y''}{Y} = -\frac{X''}{X} = const = \omega^2$$

Choose the constant to be positive to give $X'' + \omega^2 X = 0$ so the X solution is $X(x) = Acos(\omega x) + Bsin(\omega x)$

Now apply boundary conditions (BCs):

$$u(0, y) = 0 \implies X(0) = 0 \implies X(x) \sim sin(\omega x)$$
$$u(L_x, y) = 0 \implies X(L_x) = 0 \implies sin(\omega L_x) = 0 \implies L_x = n\pi$$

Sequence of solutions with $X_n(x) \sim sin(\omega_n x)$ where $\omega_n = \frac{n\pi}{L_x}$

The Y equation becomes $Y'' - \omega_n^2 Y = 0$ which has solutions

$$Y_n(y) = A_n exp(\omega_n y) + B_n exp(-\omega_n y)$$

or the more convenient linear combination

$$Y_n(y) = C_n cosh(\omega_n y) + D_n sinh(\omega_n y)$$

The boundary condition $u(x,0) = 0 \implies C_n = 0$ and linearity allows us to write the general solution as a linear combination of these solutions:

$$u(x,y) = \sum_n D_n sin(\omega_n x) sinh(\omega_n y)$$

Form of the analytic solution:

$$u(x,y) = \sum_n D_n sin(\omega_n x) sinh(\omega_n y)$$

Choose coefficients D_n to satisfy given distribution $u(x, L_y) = U(x)$ on "top edge" .

How? Expand the Fourier series expansion for a periodic extension of the remaining boundary condition $U(x, L_y) = U(x)$. For this example, $U(x) = const.$

So is the trivial Fourier series $U(x) = U$ relevant? No, because it has the wrong parity: need a Fourier sin series (*odd* periodic extension...)

Will this work for general domain shapes? NO! BCs must also be separable $\iff$ boundaries must lie on constant coordinate curves.

For general boundaries, numerical methods become essential.

Back to our main focus on numerical methods...

Start with basic discretization/iteration method:

- Discretize the domain to produce a regular grid
  - n_x be the number of grid points interior to the domain along the x direction
    $n_x + 2$ along the axis (including a boundary point at each end of the interval)
    $n_x + 1$ intervals along the axis, so the spacing between the nodes is
    $\Delta x = L_x/(n_x + 1)$
    Discrete coordinate values $x_i = (i+1)\Delta x,\ i \in [0, n_x - 1]$.
  - Maintained python indexing (`for i in range(nx)`) in the interior
    - Boundary corresponds to $x_{-1} = 0$ and $x_{n_x} = L_x$.
  - Likewise index y values with `for j in range(ny)`
    - $\Delta y = L_y/(n_y + 1)$
    - $y_j = (j+1)\Delta x,\ j \in [0, n_y - 1]$
    - Boundary corresponds to $y_{-1} = 0$ and $y_{n_y} = L_y$.

```
# Implement the discretization
Lx, Ly = 1,1
nx, ny = 4,4
x = np.linspace(0,Lx,nx)
y = np.linspace(0,Ly,ny) #linspace across full domain including boundaries
dx,dy = Lx/(nx-1), Ly/(ny-1) #spacing between nodes
x_in, y_in = x[1:-1], y[1:-1] #coords of interior points
mg = np.meshgrid(x,y) # useful for plotting
print(mg)
```

```
[array([[0. , 0.33, 0.67, 1. ],
[0. , 0.33, 0.67, 1. ],
[0. , 0.33, 0.67, 1. ],
[0. , 0.33, 0.67, 1. ]]),
array([[0. , 0. , 0. , 0. ],
[0.33, 0.33, 0.33, 0.33],
[0.67, 0.67, 0.67, 0.67],
[1. , 1. , 1. , 1. ]])]
```

Continuous domain has been replaced by a discrete grid of points, (x_i, y_j)

Solve for the values of the dependent variable on the grid (subscripts are indices):

$$u_{i,j} = u(x_i, y_j) = u\big(i\Delta x, j\Delta y\big)$$

Replace derivatives with central difference approx. to get equation for each grid point:

$$\frac{\partial^2 u}{\partial x^2} \rightarrow \frac{1}{\Delta x^2}(u_{i-1,j} - 2u_{i,j} + u_{i+1,j})$$
$$\frac{\partial^2 u}{\partial y^2} \rightarrow \frac{1}{\Delta y^2}(u_{i,j-1} - 2u_{i,j} + u_{i,j+1})$$

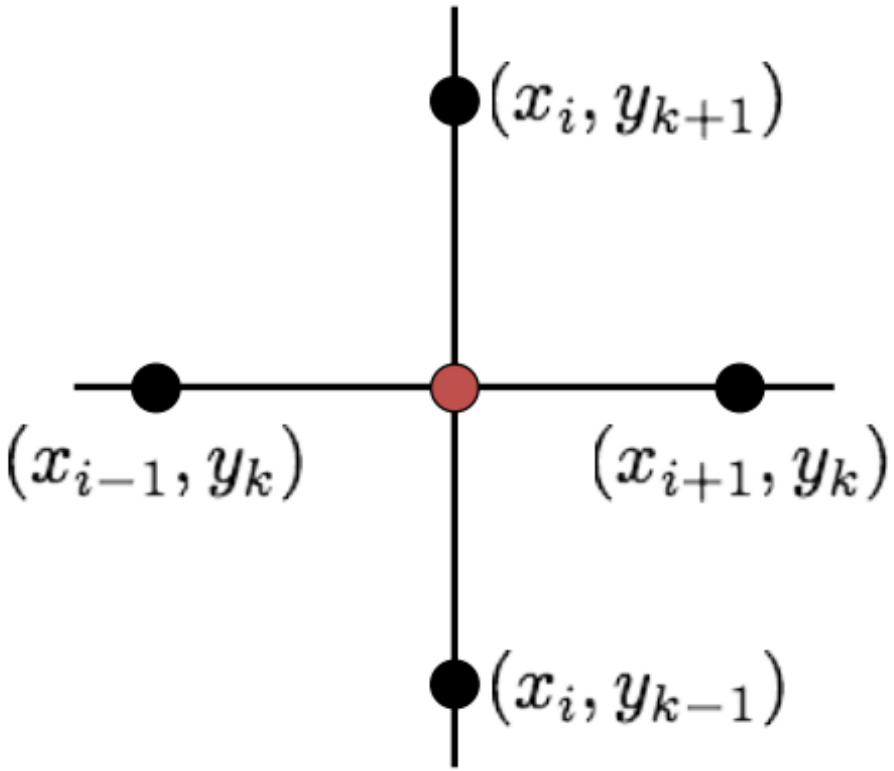
Equation for gridpoint `i,j`:

$$\nabla^2 u = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} \rightarrow u_{i-1,j} + u_{i+1,j} + u_{i,j-1} + u_{i,j+1} - 4u_{i,j} = 0$$

More readable as code: `u[i-1,j] + u[i+1,j] + u[i,j-1] + u[i,j+1] - 4*u[i,j]==0`

Classic **stencil computation**:

- Think of the solution as an array of values living on the grid
- To get equation at a grid point, overlay stencil of coefficients at the grid position and compute linear combination of stencil coefficients and grid values
- Equivalent to list correlation or list convolution



- Classic **5-point Laplace stencil**
  - Coefficient -4 on central point (red)
  - Coefficient +1 on neighbors (black)
  - "Tensor product stencil of radius 1"
    - Do same stencil with coefficients $1, -2, 1$ along each axis
    - "radius 1" $\iff$ stencil covers 1 nearest neighbor in each direction

Laplace equation has no source terms so the right-hand side is zero

Non-zero RHS $\iff$ Poisson equation; similar approach can be applied.

Iterative solver:

- Solve for $u_{i,j}$ in terms of the values at the neighboring gridpoints:

$$u_{i,j} = \frac{1}{4}(u_{i-1,j} + u_{i+1,j} + u_{i,j-1} + u_{i,j+1})$$

- Use this as the basis for an explicit iterative solver (k is the iteration number)
- Update the central value $u_{i,j}^{k+1}$ based on known neighbor values at iteration k:

$$u_{i,j}^{k+1} = \frac{1}{4}(u_{i-1,j}^{k} + u_{i+1,j}^{k} + u_{i,j-1}^{k} + u_{i,j+1}^{k})$$

- Repeatedly update each grid point with the average of the neighboring values until results converge(!?)

Implement the iterative Laplace solver to see how it works:

See Ch. 8 Finite Difference notebook for code and results.

Laplace Equation - Direct Solve

Summary of iterative solver implemented above:

- Use central difference operator to create a stencil
- Apply the stencil to update the value at individual grid points: solve one equation at a time...
- Repeatedly update values on the grid so that the 2D array of values converges to an approximate solution of the Laplace equation

Alternative approach: **Direct Solve**

- Collect the discretized equations for the full grid of nodes
- Construct one big system of equations to solve for the full grid of values
- Linear PDE $\implies$ linear system of algebraic equations
- Convenient to recast the problem into the traditional $Ax = b$ form so we can use standard linear solvers
  - How do you do that when the unknowns correspond to a 2-dimensional array instead of a 1-dimensional array?
  - Usual approach is to **flatten** the $n \times n$ 2D array into a $n^2 \times 1$ 1D array
  - Instead of stacking the rows to create a matrix, concatenate one row after another in 1 long vector (i.e., a 1D numpy array).

Concrete 4×4 example:

$$\boldsymbol{u}_{2D} = \begin{bmatrix} u[0,0] & u[0,1] & u[0,2] & u[0,3] \\ u[1,0] & u[1,1] & u[1,2] & u[1,3] \\ u[2,0] & u[2,1] & u[2,2] & u[2,3] \\ u[3,0] & u[3,1] & u[3,2] & u[3,3] \end{bmatrix}$$

Stencil:

$$\begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}$$

The 2D array flattens into the following 1D array: $\boldsymbol{u}_{1D} =$

$$\begin{bmatrix} u[0,0] \\ u[0,1] \\ u[0,2] \\ u[0,3] \\ u[1,0] \\ u[1,1] \\ u[1,2] \\ u[1,3] \\ u[2,0] \\ u[2,1] \\ u[2,2] \\ u[2,3] \\ u[3,0] \\ u[3,1] \\ u[3,2] \\ u[3,3] \end{bmatrix}$$

Flattened 1D array often presented as its transpose:

$$\boldsymbol{u}_{1D}^T = [u[0,0] \quad u[0,1] \quad u[0,2] \quad u[0,3] \quad u[1,0] \quad u[1,1] \quad u[1,2] \quad u[1,3] \quad u[2,0] \quad u[2,1] \quad u[2,2] \quad u[2,3] \quad u[3,0] \quad u[3,1] \quad u[3,2] \quad u[3,3]]$$

```
u = [u[0,0], u[0,1], u[0,2], u[0,3], u[1,0], u[1,1], u[1,2], u[1,3],
u[2,0], u[2,1], u[2,2], u[2,3], u[3,0], u[3,1], u[3,2], u[3,3]]
```

Applying the stencil at each node on the grid to get linear system $A\boldsymbol{u}_{1D} = \boldsymbol{b}$.

Look at a typical row and identify the coefficients associated with the node at the center of the stencil and its left, right, top, and bottom neighbors.

$$A\boldsymbol{u}_{2D} = \begin{bmatrix} -4 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & -4 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & -4 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & -4 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & -4 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & -4 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & -4 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 1 & -4 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & -4 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & -4 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & -4 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & -4 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & -4 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & -4 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & -4 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & -4 \end{bmatrix} \begin{bmatrix} u[0,0] \\ u[0,1] \\ u[0,2] \\ u[0,3] \\ u[1,0] \\ u[1,1] \\ u[1,2] \\ u[1,3] \\ u[2,0] \\ u[2,1] \\ u[2,2] \\ u[2,3] \\ u[3,0] \\ u[3,1] \\ u[3,2] \\ u[3,3] \end{bmatrix} = \begin{bmatrix} -U \\ -U \\ -U \\ -U \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

The non-zero entries in $\boldsymbol{b}$ arise in rows corresponding to grid points adjacent to non-homogeneous boundary values.

The array has a block matrix structure:

$$A = \begin{bmatrix} B & I & 0 & 0 \\ I & B & I & 0 \\ 0 & I & B & I \\ 0 & 0 & I & B \end{bmatrix}$$

where I is the 4×4 identity matrix, 0 is a 4×4 matrix of zeros and

$$B = \begin{bmatrix} -4 & 1 & 0 & 0 \\ 1 & -4 & 1 & 0 \\ 0 & 1 & -4 & 1 \\ 0 & 0 & 1 & -4 \end{bmatrix}$$

Note that your matrix will be different around the edges as you adjust for different boundary conditions.

For example, Eq. (8.1.14) in the text considers periodic boundary conditions resulting in having I instead of 0 in the lower left and upper right positions in the block matrix.

$$A = \begin{bmatrix} B & I & 0 & I \\ I & B & I & 0 \\ 0 & I & B & I \\ I & 0 & I & B \end{bmatrix}$$

Section 8.3 Fast Poisson Solvers: The Fourier Transform

Step up from Laplace's equation by adding source term to get the Poisson equation:

$$\nabla^2 u = \omega$$

Would our approach to the Laplace equation extend to handle the Poisson equation? What would need to be changed?

The basic answer is YES as long as:

- We properly include the source term as part of the right-hand side vector b
- We know we can solve the resulting linear system using a factorization method that involves $O(N^2)$ operations.

That approach is fine until we want to handle really large domains or really fine discretization and the $O(N^2)$ computation becomes prohibitively expensive (i.e. slow).

Consider alternative approach with a fundamentally distinct mathematical foundation.

Pretty much everything we have done so far has employed Taylor series together with discretization of the independent variable (typically $x, y,$ or t).

Taylor series methods "always" incur truncation error.

Alternative approach: **Apply the Fourier Transform (FT) and discretize both independent variable and the transform variable (frequency/wavenumber)**

Start with a quick refresher on the Fourier transform pair and key properties.

Basics of Fourier Transform

- The Fourier Transform is an integral transform involving the complex exponential.
- Integration over the domain of the independent variable transforms a function $f(x)$ into a function $\widehat{f(x)} = F(k)$ in the wavenumber (or frequency) domain.
- FT has an associated **Inverse Fourier Transform (IFT)** that integrates a function of the wavenumber to transform back to the original (spatial or temporal) domain.
- Together they form the Fourier transform pair:

$$F(k) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} e^{-ikx} f(x) dx \qquad = \widehat{f(x)}$$

$$f(x) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} e^{ikx} F(k) dk$$

- There are other versions (usually with factors of $\sqrt{2\pi}$ moved around), but we'll stick with this version that has optimal symmetry.
- Here integrals are defined over infinite domains,
  but $\exists$ related transform pairs on finite/discrete domains.
- Recall major operational property that makes the Fourier transform useful:

Fourier transform of the derivative

$$\begin{aligned}
\widehat{f'(x)} &= \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} e^{-ikx} f'(x) dx && \text{Basic definition} \\
&= \frac{1}{\sqrt{2\pi}} f(x) e^{-ikx} \Big|_{-\infty}^{\infty} + \frac{ik}{\sqrt{2\pi}} \int_{-\infty}^{\infty} e^{-ikx} f(x) dx && \text{Integration by parts} \\
&= ik\widehat{f(x)} && \text{Assume asymptotic boundedness}
\end{aligned}$$

Key operational property: differentiation simplifies to multiplication $\iff d/dx \to ik$

Example using Fourier Transform (FT) to solve differential equation (DE):

$y'' - \omega^2 y = -f(x); \quad x \in [-\infty, \infty]; \quad f(x) \to 0 \text{ as } x \to \pm\infty$

Apply FT to DE+BCs: $\widehat{y''} - \omega^2 \hat{y} = -\hat{f}$

Apply operational property ($d/dx \to ik$): $-k^2 \hat{y} - \omega^2 \hat{y} = -\hat{f}$

Obtain algebraic equation for transformed solution: $(k^2 + \omega^2)\hat{y} = \hat{f}$

Solve for transformed solution: $\hat{y} = \dfrac{\hat{f}}{k^2 + \omega^2}$

Invert transform to obtain solution of DE+BCs:

$$y(x) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} e^{ikx} \hat{y}\, dk = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} e^{ikx} \frac{\hat{f}}{k^2 + \omega^2}\, dk$$

(Evaluate by residue theory or IFT table)

- Notation warning: FT definition differs in transform domain variable: ω, f, or in this case k. Here independent variable is k and ω is a constant.
- Above we saw an example where we can apply the "theoretical" transform/inverse.
  - For common functions, look up transforms/inverses in tables or compute integrals using complex variable methods (residue theory)
- For more "realistic" problems on finite domain, discretize (hopefully, no surprise...):
  - Domain becomes discrete grid (both independent and transformed variables)
  - Forcing function $f(x)$ and solution $y(x)$ become arrays of values on the discrete independent variable grid
  - Transform variable becomes array of values on the wavenumber/freq. grid
  - FT $\rightarrow$ Finite Fourier Transform or Fast Fourier Transform (FFT)
  - Integral with exponential kernel $\rightarrow$ matrix multiplication with Vandermonde matrix (powers of fundamental complex exponential)
  - Inversion integral $\rightarrow$ multiplication by (known) matrix inverse

- Such simplification makes FT of interest, but FAST Fourier Transform (FFT) makes it practical and important for engineering computation
- Order of computation for multiplying $n \times n$ matrix by n-vector?
  - Naive matrix mult. $\sim O(n^2)$
  - FAST algorithm uses divide-and-conquer to achieve $\sim O(n\,log(n))$
  - Developed by Cooley and Tukey *ca.* 1965
  - One of the top 10 algorithms of 20^{th} century
  - Makes real-time signal processing a reality and changed the world...

Let's take a look at how the FFT works using standard python library code in `np.fft`

Canonical Fourier Transform example: Gaussian function $f(x) = exp(-\alpha x^2)$

Transform is also a Gaussian:

$$\widehat{f(x)} = \frac{1}{\sqrt{2\alpha}} exp(-\frac{k^2}{4\alpha}); \qquad \text{Eq.}(8.4.1)$$

As in the text, set $\alpha = 1$ to keep things simple

Choose the interval $x \in [-10, 10]$ subdivided into $N = 2^7 = 128$ intervals.

Code below replicates Fig. 8.4 using the `numpy.fft` versions of `fft` and `fftshift`.

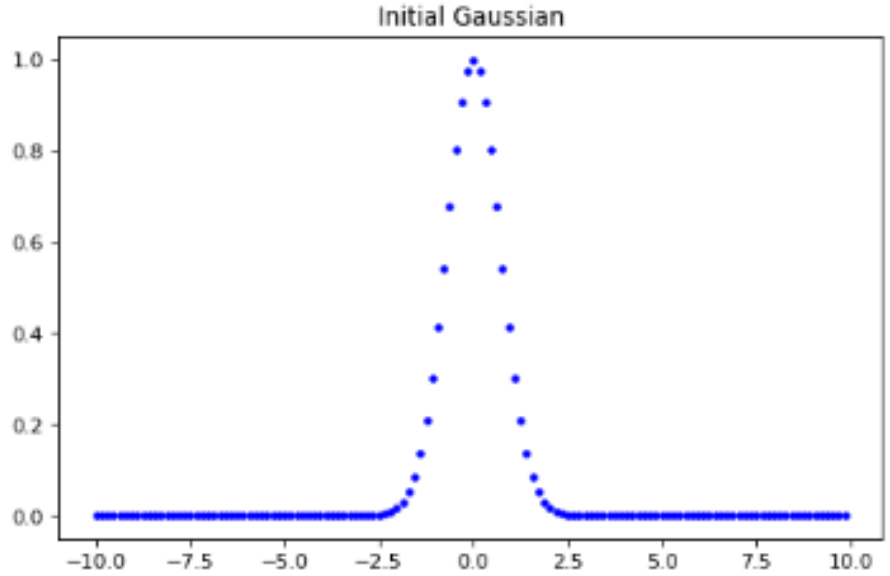
Note that **shifting** occurs, but we usually do not deal with this explicitly.

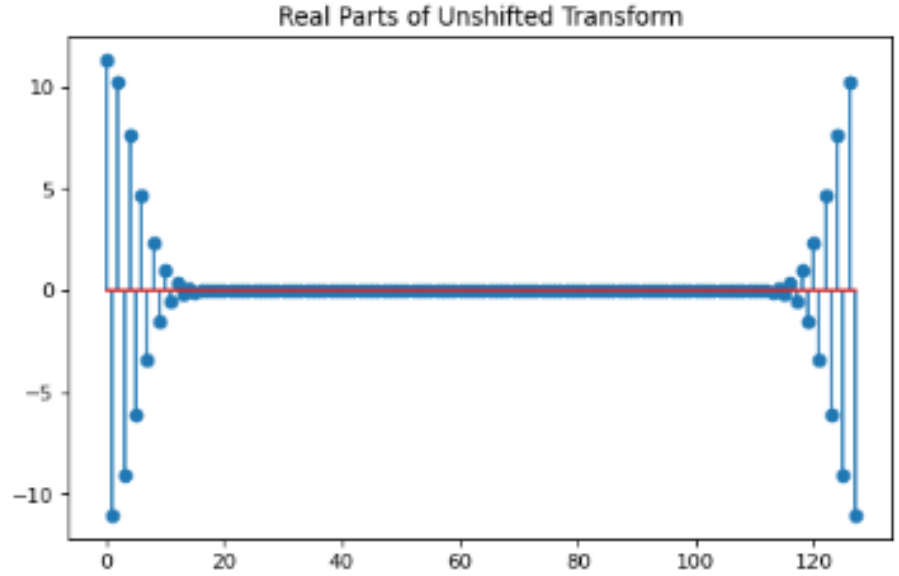
Use an inverse transform that properly inverts the shift; e.g. `np.fft.ifft`

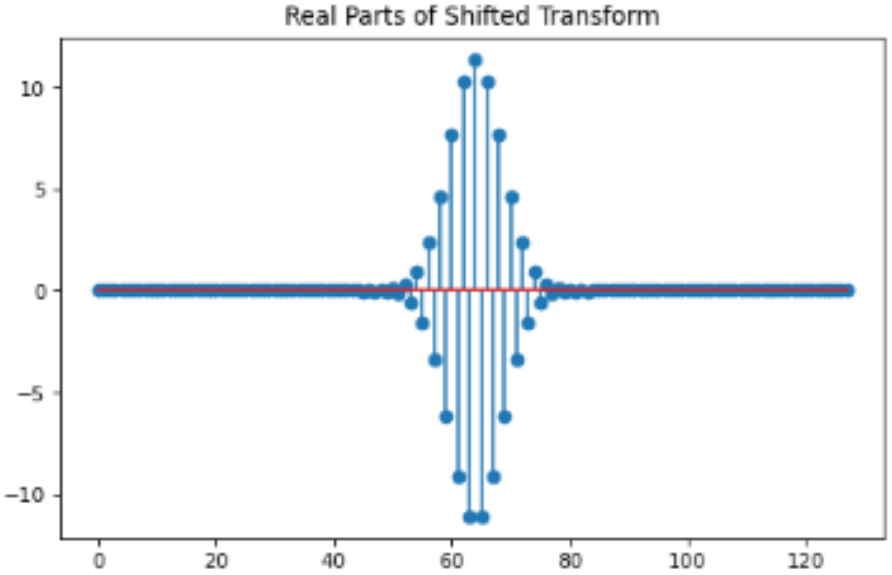
Library implementation of FFT (from `numpy.fft`)

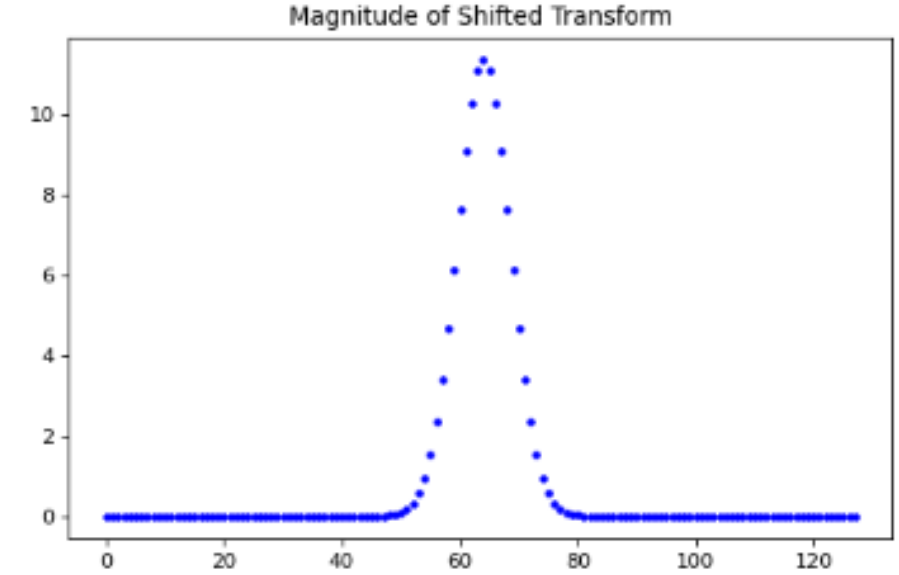
```
import numpy as np
import matplotlib.pyplot as plt

alpha = 1.
L = 20
n = 128

x2 = np.linspace(-L/2,L/2,n+1) #note that this L differs by factor of 2 from L as introduced on p. 188
x = x2[:n]

u = np.exp(-alpha*x*x)
ut = np.fft.fft(u)
utshift = np.fft.fftshift(ut)

fig, ((ax1, ax2), (ax3, ax4))= plt.subplots(nrows=2,ncols=2,figsize=(16,10),facecolor='white',dpi=80)

ax1.plot(x,u,'b.'); ax1.set(title="Initial Gaussian")
ax2.stem(np.real(ut)); ax2.set(title="Real Parts of Unshifted Transform")
ax3.stem(np.real(utshift)); ax3.set(title="Real Parts of Shifted Transform")
ax4.plot(np.abs(utshift),'b.'); ax4.set(title="Magnitude of Shifted Transform")
plt.show()
```

Example: Discrete Fourier Transform and Inverse of Gaussian Pulse





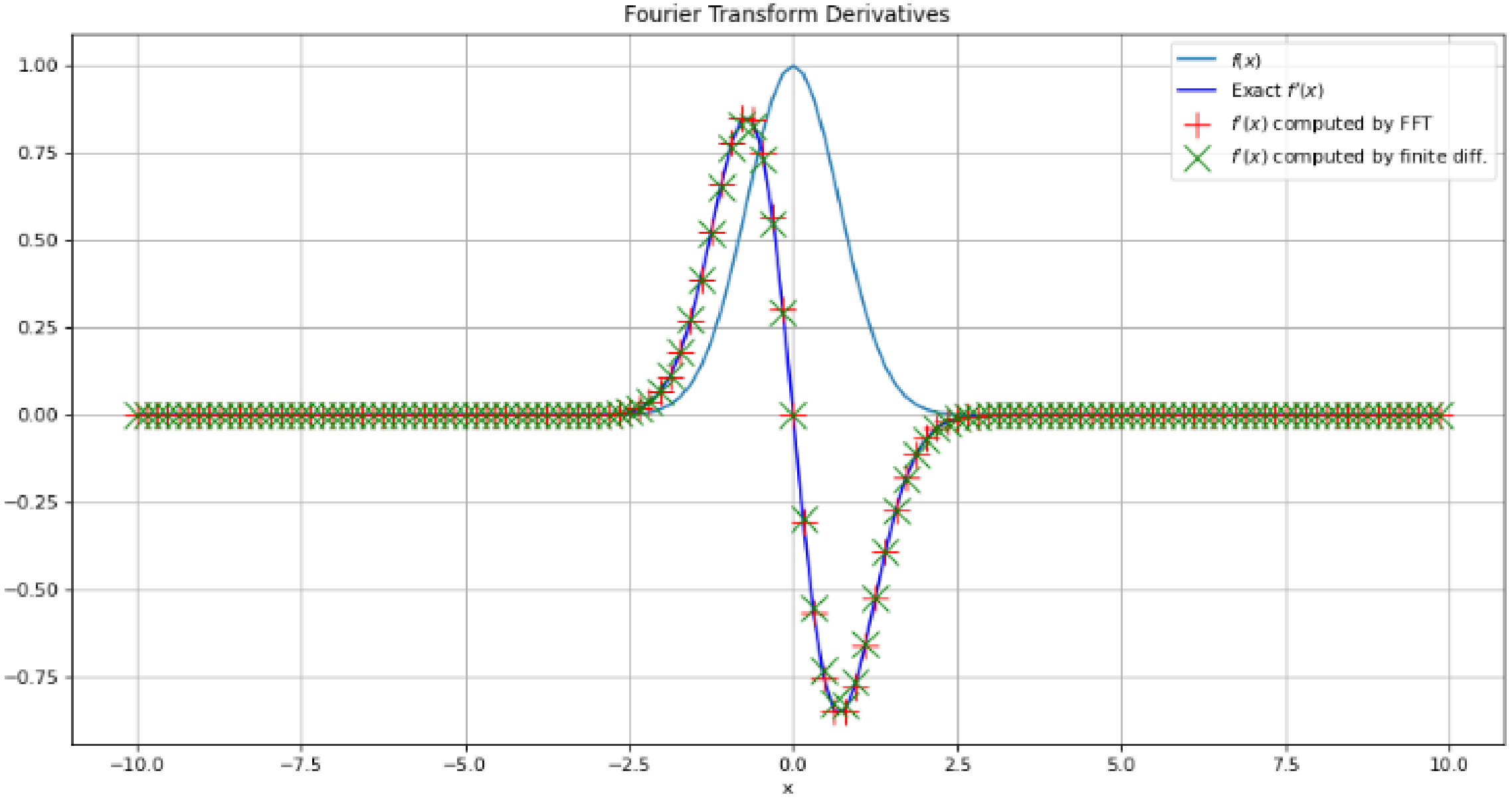




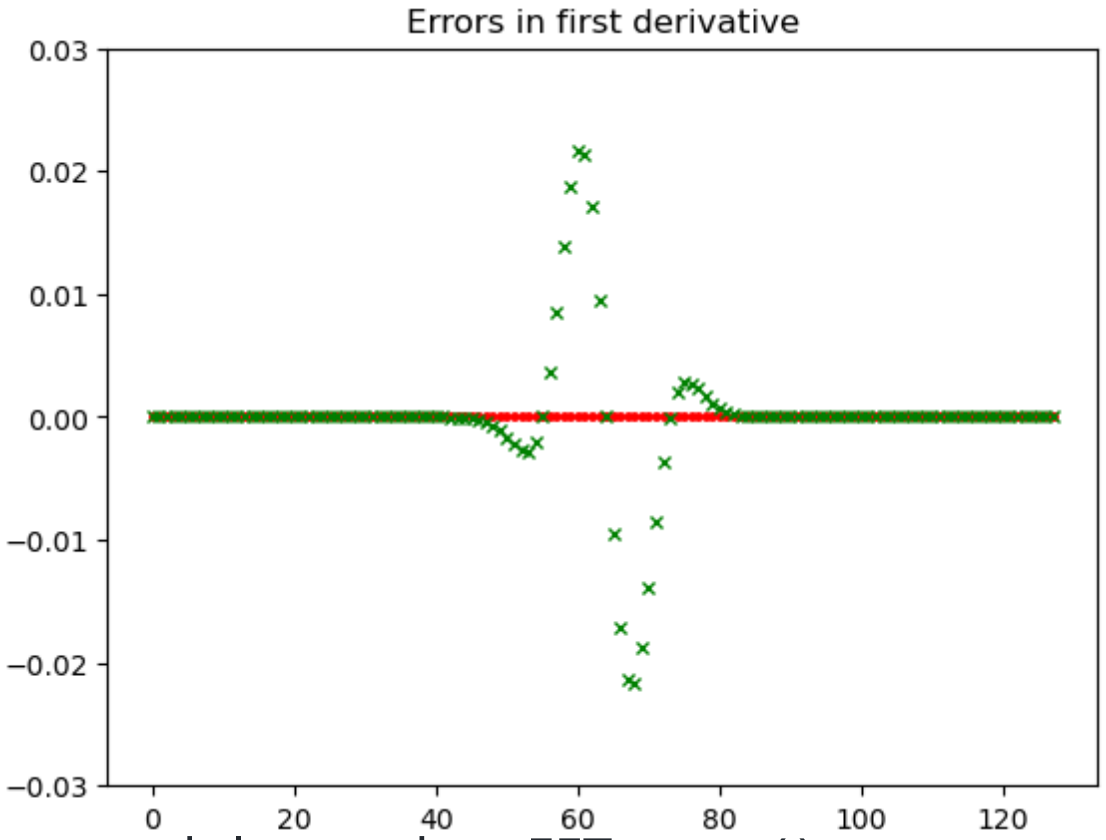
Compare derivatives computed by FFT and finite difference

```
# compute fft and discrete grid of wavenumber values
ut = np.fft.fft(u) # compute the FFT
k = (2*np.pi/L)*np.array([*range(int(n/2)),
                          *range(int(-n/2),0,1)]) # *range unpacks the range iterator

df = 1j*k #'1j' is python for sqrt(-1)
#'df*' is "FFTish" for 'd/dx'
ut1 = df*ut    #compute transform of 1st derivative

u1 = np.fft.ifft(ut1) #inverse FFT ut1 to get du/dx

# compute derivatives by finite diff for comparison
dudt = np.copy(u)
# d2udt2 = np.copy(u)
h = 20./(n-1)
for i in range(1, n-1):
    dudt[i] = (u[i+1] - u[i-1])/(2.*h)
```

Plot derivative estimates: FFT vs. finite difference



Both derivatives look good, but take a closer look...



Finite difference (x) error much larger than FFT error (.)

Finite difference: roundoff error AND truncation error

FFT: only roundoff error. No truncation error!