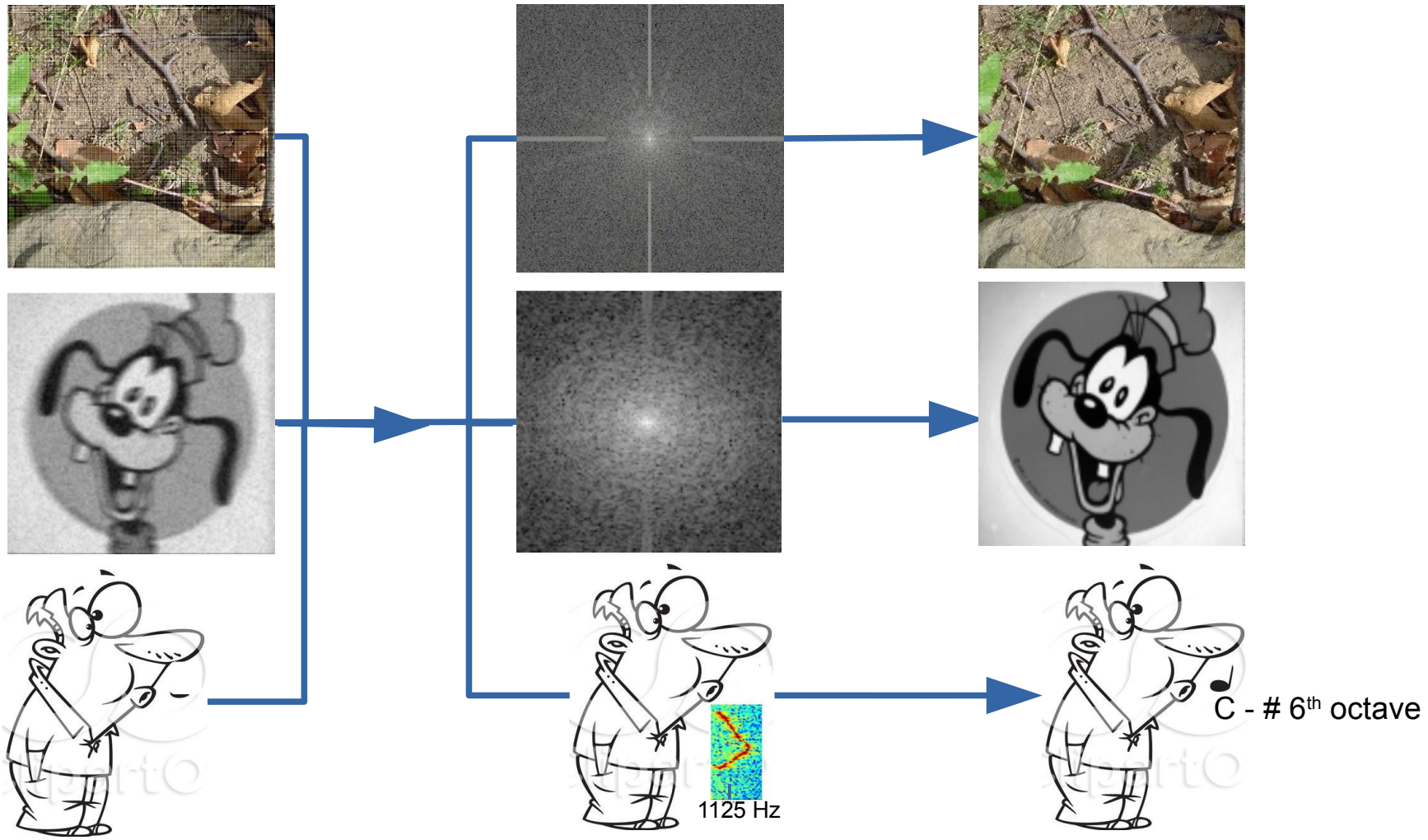
Special Topics in Computational Physics

Special Topics in Computational Physics:

- Fast Fourier Transforms (FFT)
- QuTiP: Quantum Toolbox in Python
- Monte Carlo Simulations and Metropolis Algorithm
- Partial Differential Equations: Laplace and Wave equations

FFT: The Fast Fourier Transform

FFT: The Fast Fourier Transform

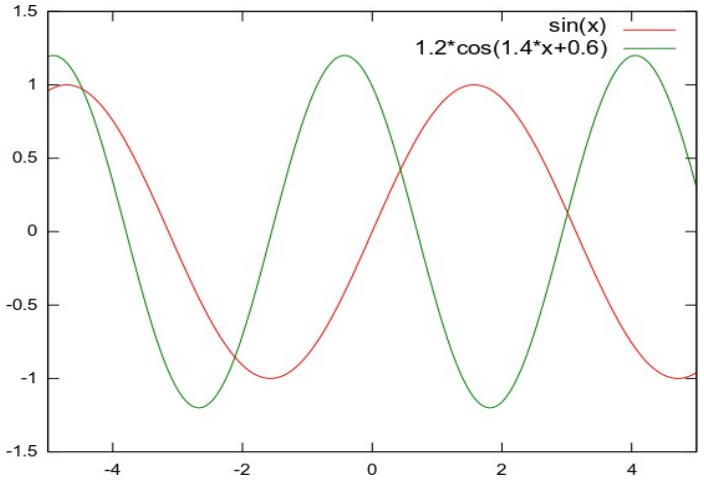


1. Introduction

Intuitive explanation of Fourier Theory

Any signal (Spatially and/or temporally varying data like images, sounds etc.) can be expressed as a sum of series of sinusoids

1 – Dimensional Example (we'll mostly stick to 1D for the rest of this course)

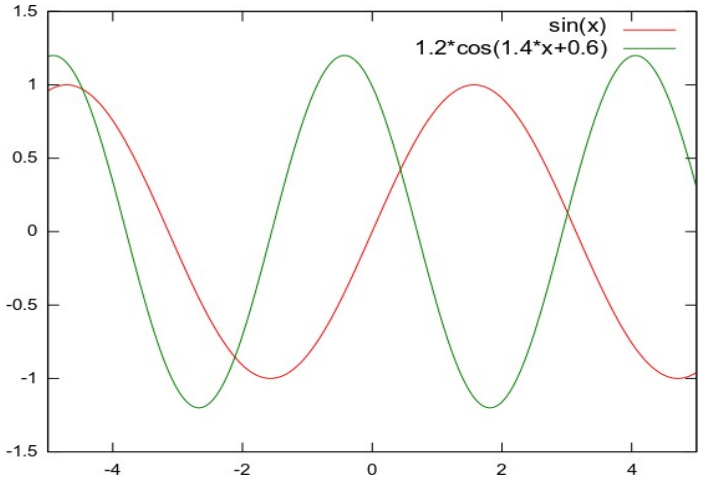


1. Introduction

Intuitive explanation of Fourier Theory

Any signal (Spatially and/or temporally varying data like images, sounds etc.) can be expressed as a sum of series of sinusoids

1 – Dimensional Example (we'll mostly stick to 1D for the rest of this course)



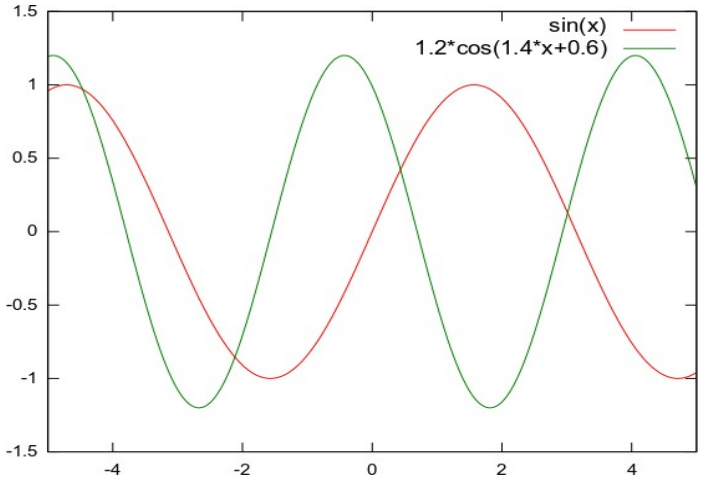
Add the components

1. Introduction

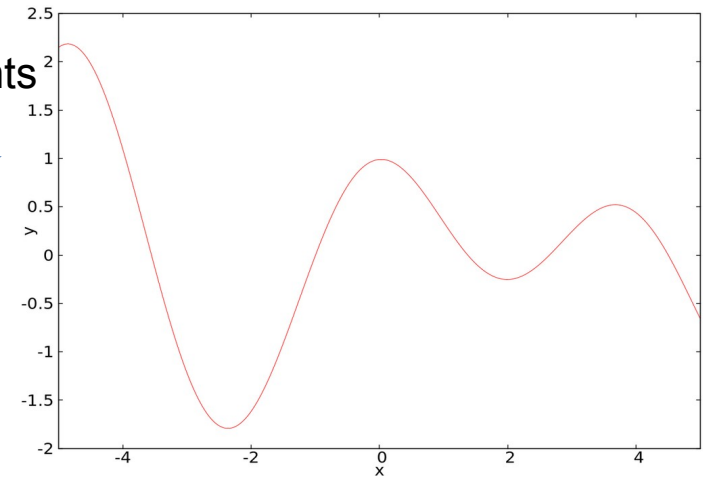
Intuitive explanation of Fourier Theory

Any signal (Spatially and/or temporally varying data like images, sounds etc.) can be expressed as a sum of series of sinusoids

1 – Dimensional Example (we'll mostly stick to 1D for the rest of this course)



Add the components

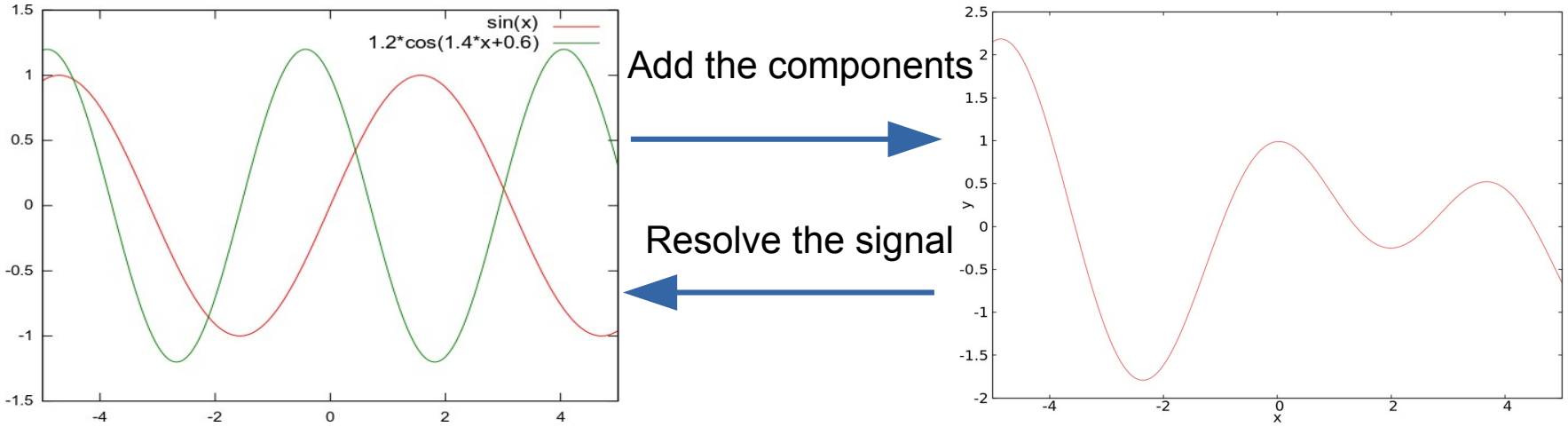


1. Introduction

Intuitive explanation of Fourier Theory

Any signal (Spatially and/or temporally varying data like images, sounds etc.) can be expressed as a sum of series of sinusoids

1 – Dimensional Example (we'll mostly stick to 1D for the rest of this course)

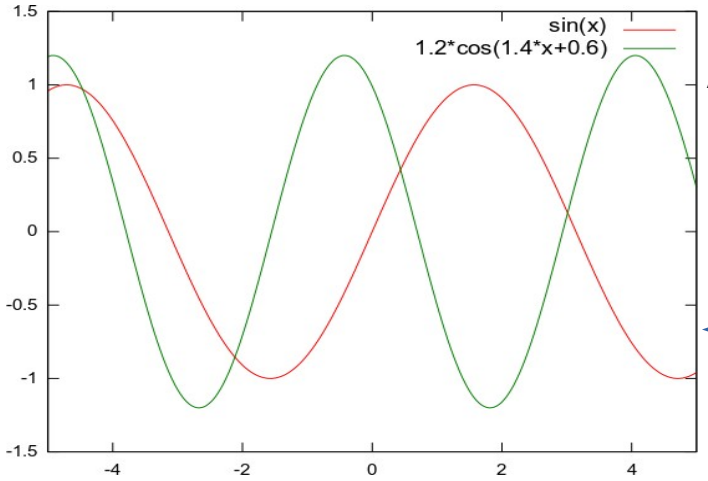


1. Introduction

Intuitive explanation of Fourier Theory

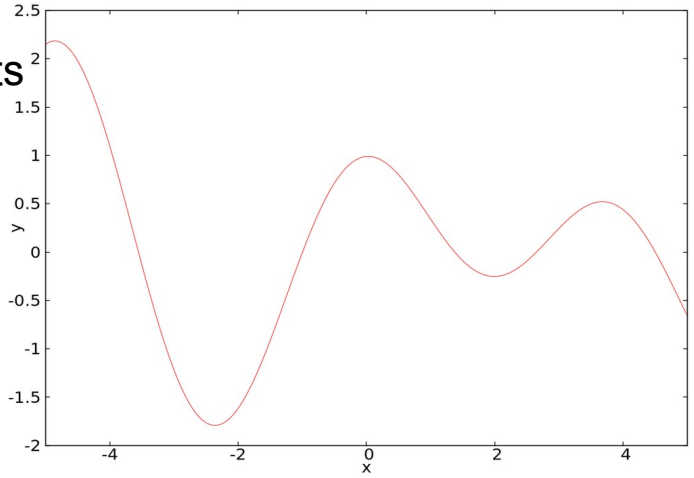
Any signal (Spatially and/or temporally varying data like images, sounds etc.) can be expressed as a sum of series of sinusoids

1 – Dimensional Example (we'll mostly stick to 1D for the rest of this course)



Add the components

Resolve the signal



$$\sin x = \frac{1}{2i}e^{ix} - \frac{1}{2i}e^{-ix} = \frac{1}{2}e^{-i\pi/2}e^{ix} + \frac{1}{2}e^{i\pi/2}e^{-ix}$$

$$1.2\ \cos(1.4\,x + 0.6) = \left\{\frac{1.2}{2}\ e^{i(0.6)}\ e^{i(1.4\,x)}\right\} + \left\{\frac{1.2}{2}\ e^{-i(0.6)}\ e^{-i(1.4\,x)}\right\}$$

Fourier Transform:

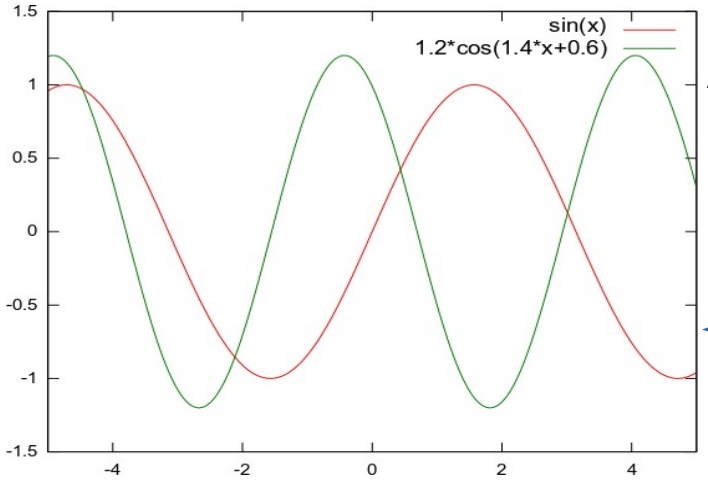
Plot of Amplitudes (Phases) as functions of resolved frequencies

1. Introduction

Intuitive explanation of Fourier Theory

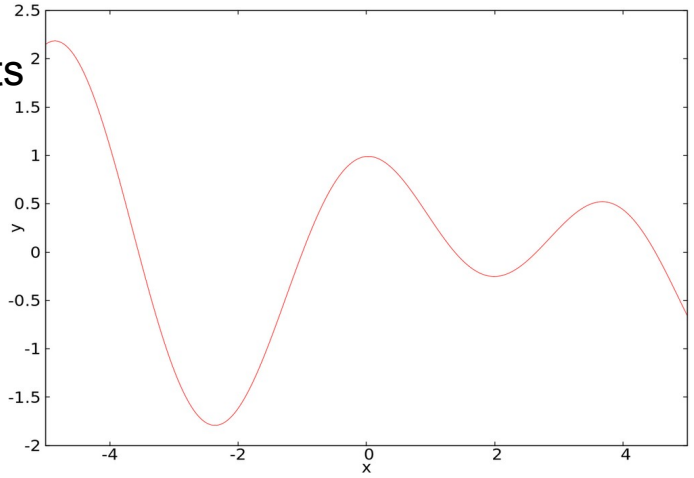
Any signal (Spatially and/or temporally varying data like images, sounds etc.) can be expressed as a sum of series of sinusoids

1 – Dimensional Example (we'll mostly stick to 1D for the rest of this course)



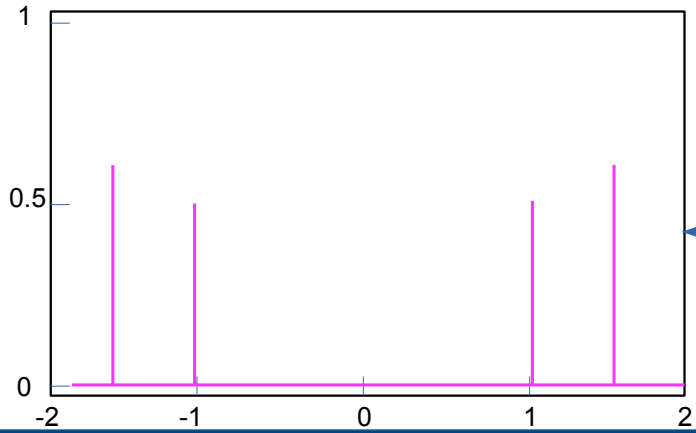
Add the components

Resolve the signal



$$\sin x = \frac{1}{2i}e^{ix} - \frac{1}{2i}e^{-ix} = \frac{1}{2}e^{-i\pi/2}e^{ix} + \frac{1}{2}e^{i\pi/2}e^{-ix}$$

$$1.2\ \cos(1.4\,x + 0.6) = \left\{\frac{1.2}{2}\ e^{i(0.6)}\ e^{i(1.4\,x)}\right\} + \left\{\frac{1.2}{2}\ e^{-i(0.6)}\ e^{-i(1.4\,x)}\right\}$$



Fourier Transform:

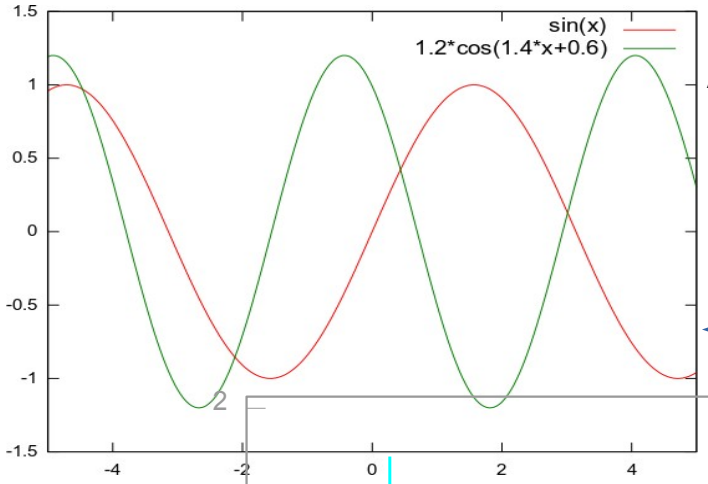
Plot of Amplitudes (Phases) as functions of resolved frequencies

1. Introduction

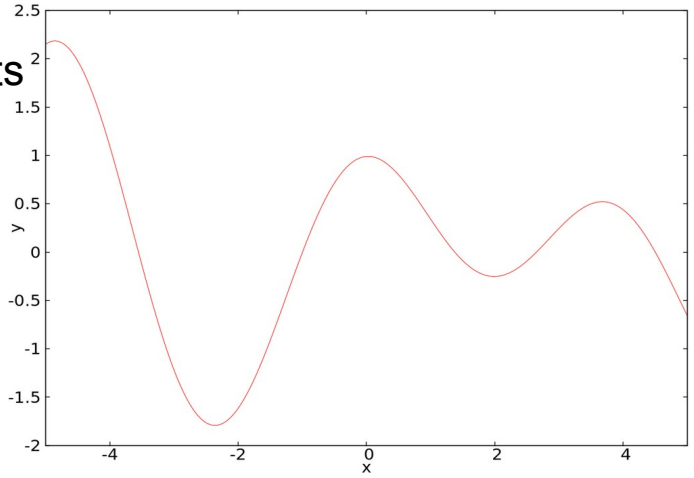
Intuitive explanation of Fourier Theory

Any signal (Spatially and/or temporally varying data like images, sounds etc.) can be expressed as a sum of series of sinusoids

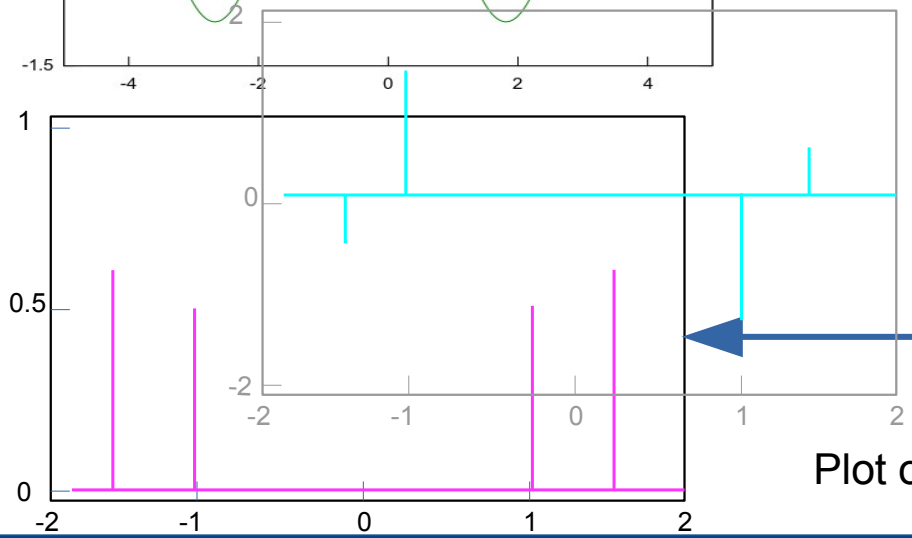
1 – Dimensional Example (we'll mostly stick to 1D for the rest of this course)



Add the components



Resolve the signal



$$\sin x = \frac{1}{2i}e^{ix} - \frac{1}{2i}e^{-ix} = \frac{1}{2}e^{-i\pi/2}e^{ix} + \frac{1}{2}e^{i\pi/2}e^{-ix}$$

$$1.2\ \cos(1.4\,x + 0.6) = \left\{\frac{1.2}{2}\ e^{i(0.6)}\ e^{i(1.4\,x)}\right\} + \left\{\frac{1.2}{2}\ e^{-i(0.6)}\ e^{-i(1.4\,x)}\right\}$$

Fourier Transform:

Plot of Amplitudes (Phases) as functions of resolved frequencies

1. Introduction

Intuitive explanation of Fourier Theory

Any signal (Spatially and/or temporally varying data like images, sounds etc.) can be expressed as a sum of series of sinusoids

2 – Dimensional Examples (we'll mostly stick to 1D for the rest of this course)

1. Introduction

Intuitive explanation of Fourier Theory

Any signal (Spatially and/or temporally varying data like images, sounds etc.) can be expressed as a sum of series of sinusoids

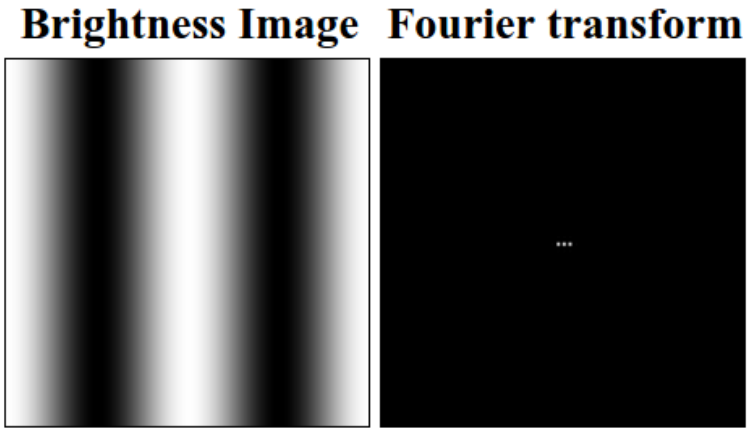
2 – Dimensional Examples (we'll mostly stick to 1D for the rest of this course)

1. Introduction

Intuitive explanation of Fourier Theory

Any signal (Spatially and/or temporally varying data like images, sounds etc.) can be expressed as a sum of series of sinusoids

2 – Dimensional Examples (we'll mostly stick to 1D for the rest of this course)

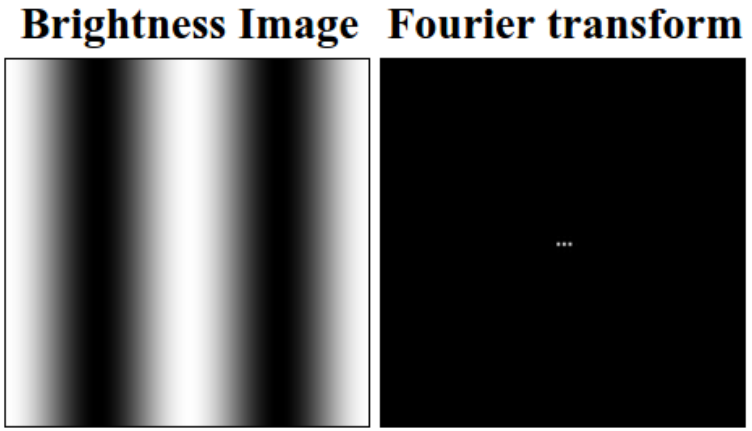


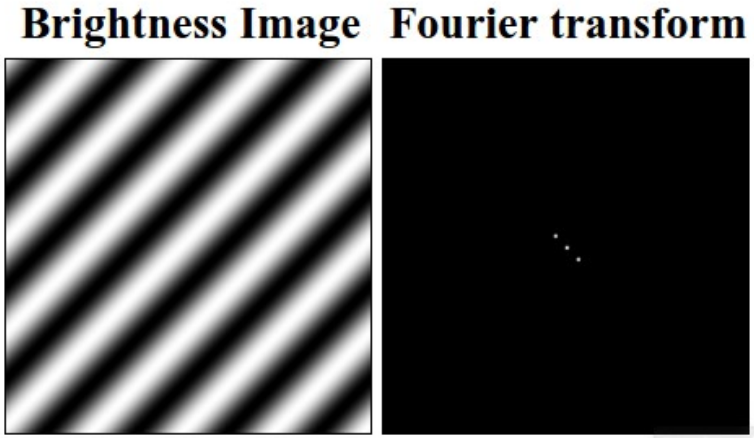
1. Introduction

Intuitive explanation of Fourier Theory

Any signal (Spatially and/or temporally varying data like images, sounds etc.) can be expressed as a sum of series of sinusoids

2 – Dimensional Examples (we'll mostly stick to 1D for the rest of this course)



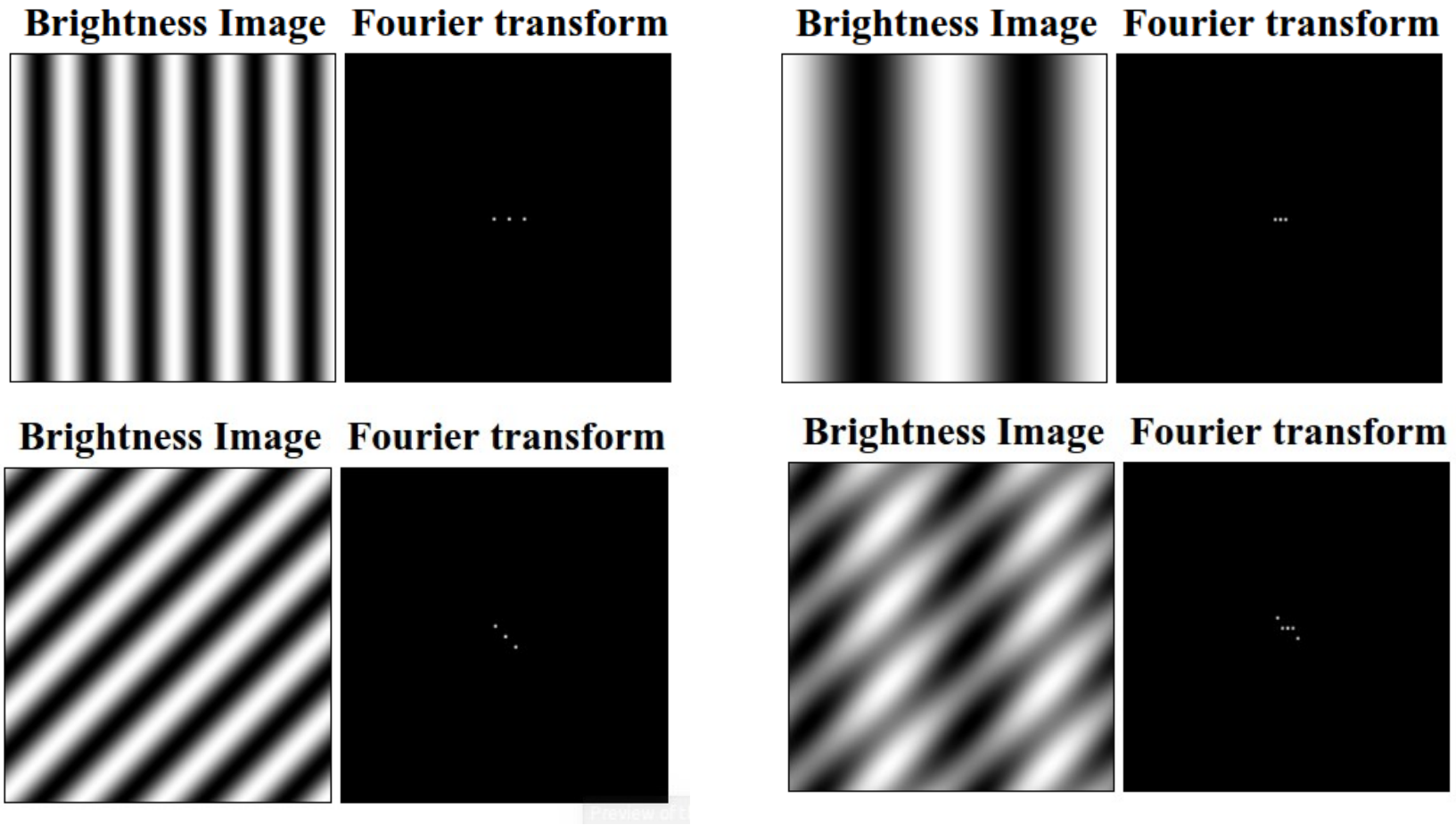


1. Introduction

Intuitive explanation of Fourier Theory

Any signal (Spatially and/or temporally varying data like images, sounds etc.) can be expressed as a sum of series of sinusoids

2 – Dimensional Examples (we'll mostly stick to 1D for the rest of this course)



1. Introduction

Intuitive explanation of Fourier Theory

Any signal (Spatially and/or temporally varying data like images, sounds etc.) can be expressed as a sum of series of sinusoids

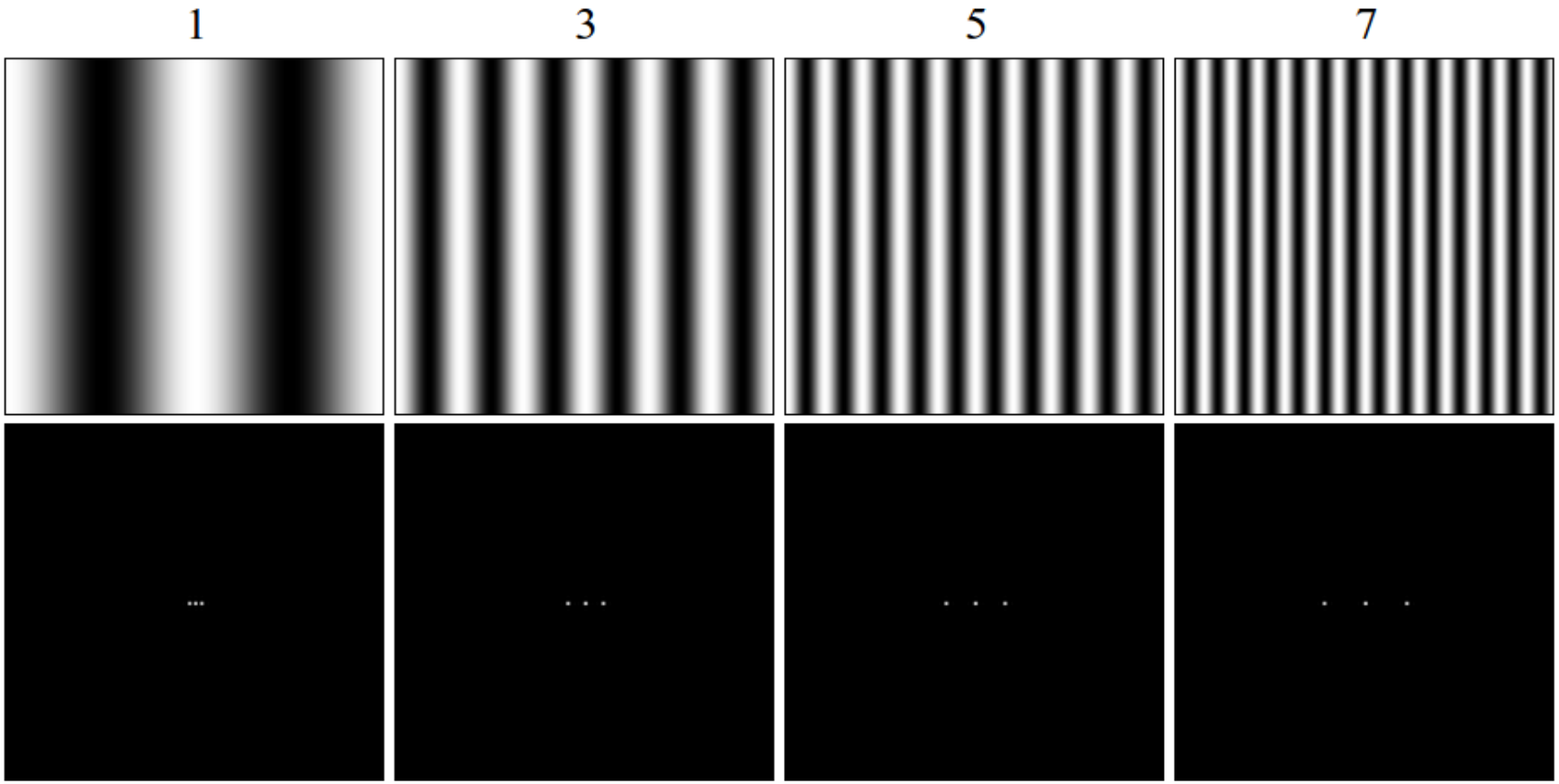
Higher Harmonics – If signal is “finite” (in extent) or has sharp discontinuities

1. Introduction

Intuitive explanation of Fourier Theory

Any signal (Spatially and/or temporally varying data like images, sounds etc.) can be expressed as a sum of series of sinusoids

Higher Harmonics – If signal is “finite” (in extent) or has sharp discontinuities

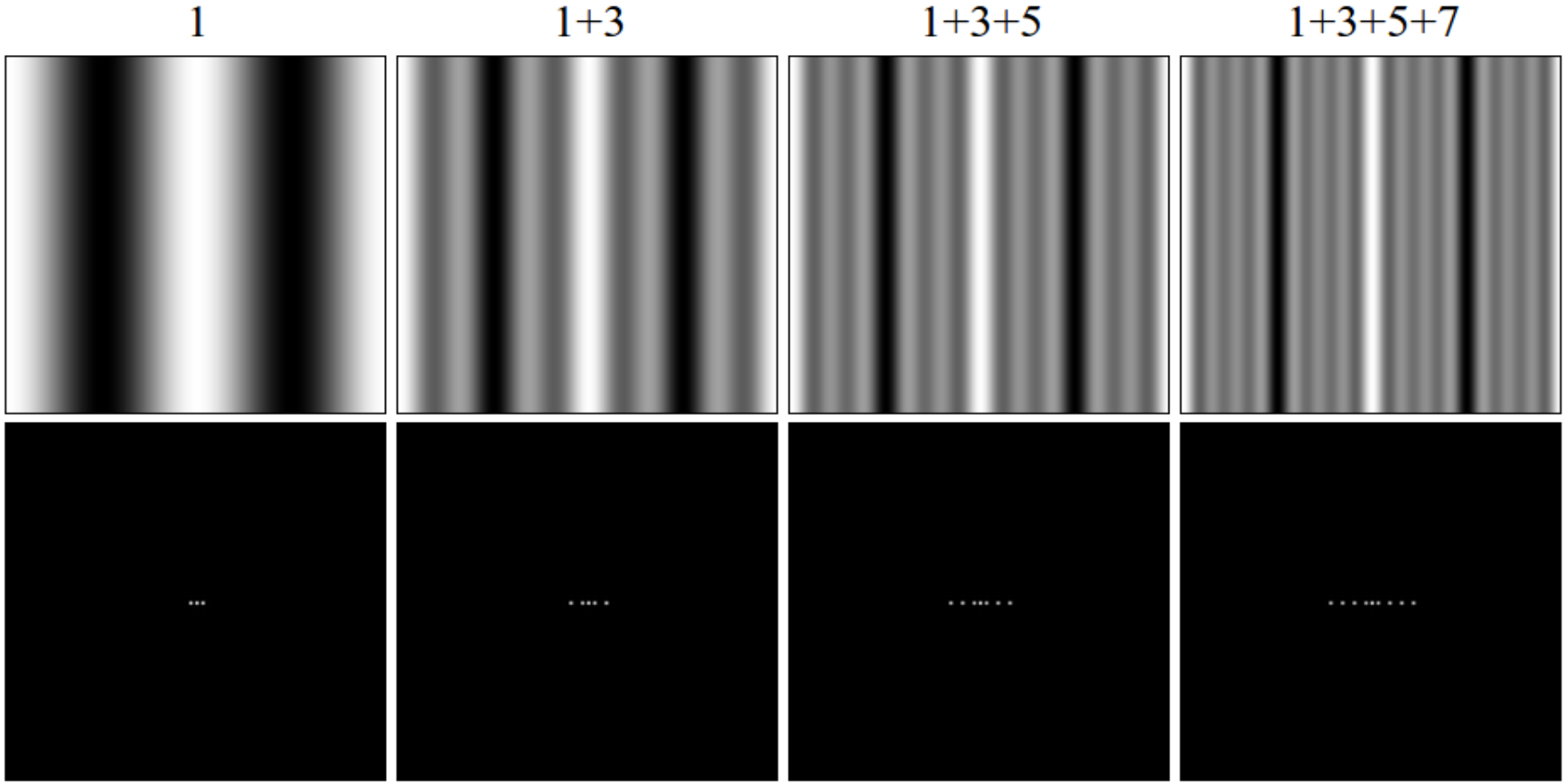


1. Introduction

Intuitive explanation of Fourier Theory

Any signal (Spatially and/or temporally varying data like images, sounds etc.) can be expressed as a sum of series of sinusoids

Higher Harmonics – If signal is “finite” (in extent) or has sharp discontinuities



1. Introduction

Intuitive explanation of Fourier Theory

Any signal (Spatially and/or temporally varying data like images, sounds etc.) can be expressed as a sum of series of sinusoids

Higher Harmonics – If signal is “finite” (in extent) or has sharp discontinuities

2. Theoretical Section:

- Formal Definition of Fourier Transform (FT):

  Suppose $f(x)$ is absolutely integrable in $\left(-\infty, \infty\right)$, then the FT is

$$\bar{f}(\omega) = \int_{-\infty}^{\infty} \mathrm{d}t \; f(t) \; e^{-i\omega t}$$

- Moreover, if the function is square integrable, then the inverse FT (IFT) is

$$f(t) = \frac{1}{2\pi} \int_{-\infty}^{\infty} \mathrm{d}\omega \; \bar{f}(\omega) \; e^{i\omega t}$$

2. Theoretical Section:

- Important Properties of Fourier Transform (FT):
  - 1. Derivative to Coefficient:

$$\overline{\frac{\mathrm{d}f}{\mathrm{d}t}} = i\omega\bar{f}$$

  - 2. Translation Property:

$$\overline{f(t-T)} = e^{-i\omega T}\bar{f}$$

  - 3. Convolution Property:

$$\overline{f_1 \otimes f_2} = \bar{f}_1 \ \bar{f}_2$$

where $\{f_1 \otimes f_2\}(x) \equiv \int_{-\infty}^{\infty} \mathrm{d}t \ f_1(x-t) f_2(t)$

  - 4. Parseval's theorem:

$$\int_{-\infty}^{\infty} \mathrm{d}t \ |f(t)|^2 = \frac{1}{2\pi} \int_{-\infty}^{\infty} \mathrm{d}\omega \ |\bar{f}(\omega)|^2$$

3. The Discreet Fourier Transform (DFT)

$$\bar{f}(\omega) = \int_{-\infty}^{\infty} \mathrm{d}t \; f(t) \; e^{-i\omega t}$$

- Given the function, sample it at discrete points and estimate the FT

  The value of the function is recorded at regular time intervals and assembled into a vector

$$|f\rangle = (f_1, f_2, f_3 \; \cdots \; f_N) \;\; , \;\; f_n = f(n\Delta)$$

$\Delta^{-1} =$ Sampling Rate

DFT

$$|\bar{f}\rangle = (\bar{f}_1, \bar{f}_2, \bar{f}_3 \; \cdots \; \bar{f}_N) \;\; , \;\; \bar{f}_n = ?$$

- Given a discreet vector of data, estimate the FT

  The data given might not correspond to a function sampled at regular intervals

  So regularize it (estimate the function locally by interpolation and sample it as above)

3. The Discreet Fourier Transform (DFT)

$$\bar{f}(k) = \int_{-\infty}^{\infty} \mathrm{d}t \; f(t)\, e^{-2\pi i k t} \;,\; k \in (-\infty, \infty)$$

$$|f\rangle = (f_1, f_2, f_3 \; \cdots \; f_N) \;,\; f_n = f(n\Delta)$$

DFT

$$\bar{f}_m = \sum_{n=-N/2+1}^{N/2-1} \Delta \; f_n \; e^{-2\pi i n \Delta k_m} \;,\; k_m = \frac{m}{N\Delta}$$

Choose

$$|\bar{f}\rangle = \left(\bar{f}_{-N/2} \cdots \;, \bar{f}_{-1}, \bar{f}_0, \bar{f}_1 \; \cdots \; \bar{f}_{N/2}\right) \quad \left[-\frac{N}{2}, +\frac{N}{2}\right]$$

3. The Discreet Fourier Transform (DFT)

$$\bar{f}(k) = \int_{-\infty}^{\infty} \mathrm{d}t \; f(t)\, e^{-2\pi i k t} \;,\; k \in (-\infty, \infty)$$

$$|f\rangle = (f_1, f_2, f_3 \; \cdots \; f_N) \;,\; f_n = f(n\Delta)$$

DFT

$$\bar{f}_m = \sum_{n=-N/2+1}^{N/2-1} \Delta \; f_n \; e^{-2\pi i m n/N}$$

$$|\bar{f}\rangle = \left(\bar{f}_{-N/2} \cdots \;, \bar{f}_{-1}, \bar{f}_0, \bar{f}_1 \; \cdots \; \bar{f}_{N/2}\right) \quad \left[-\frac{N}{2}, +\frac{N}{2}\right]$$

3. The Discreet Fourier Transform (DFT)

$$\bar{f}(k) = \int_{-\infty}^{\infty} \mathrm{d}t \; f(t)\, e^{-2\pi i k t} \;,\; k \in (-\infty, \infty)$$

$$|f\rangle = (f_1, f_2, f_3 \; \cdots \; f_N) \;,\; f_n = f(n\Delta)$$

DFT

$$\bar{f}_m = \sum_{n=-N/2+1}^{N/2-1} \Delta f_n \, e^{-2\pi i m n/N}$$

$$|\bar{f}\rangle = \left(\bar{f}_{-N/2} \cdots \;, \bar{f}_{-1}, \bar{f}_0, \bar{f}_1 \; \cdots \; \bar{f}_{N/2}\right) \quad \left[-\frac{N}{2}, +\frac{N}{2}\right]$$

3. The Discreet Fourier Transform (DFT)

$$\bar{f}_m = \sum_{n=0}^{N-1} f_n \, e^{-2\pi i m n/N}$$

Periodic in m with period N *i.e* $\bar{f}_{-j} = \bar{f}_{N-j}\ ,\ j = 1, 2, \ldots$

$$|\bar{f}\rangle = \left(\bar{f}_{-N/2} \cdots\ , \bar{f}_{-1}, \bar{f}_0, \bar{f}_1\ \cdots\ \bar{f}_{N/2}\right) \quad \left[-\frac{N}{2}, +\frac{N}{2}\right]$$

3. The Discreet Fourier Transform (DFT)

$$\bar{f}_m = \sum_{n=0}^{N-1} f_n \, e^{-2\pi i m n/N}$$

Periodic in m with period N *i.e* $\bar{f}_{-j} = \bar{f}_{N-j}\ ,\ j = 1, 2, \ldots$

$$|\bar{f}\rangle = \left(\bar{f}_0, f_2\ \ldots\ ,\ \ldots\ ,\ \ldots\ ;\ldots\ \bar{f}_{N-1}\right)\quad [1, N)$$

3. The Discreet Fourier Transform (DFT)

$$\bar{f}_m = \sum_{n=0}^{N-1} f_n \, e^{-2\pi i m n/N}$$

Periodic in m with period N *i.e* $\bar{f}_{-j} = \bar{f}_{N-j}\ ,\ j = 1, 2, \ldots$

$$|\bar{f}\rangle = \left(\bar{f}_0, f_2\ \ldots\ ,\ \ldots\ ,\ \ldots\ ;\ldots\ \bar{f}_{N-1}\right) \quad [1, N)$$

- Important Properties of DFT:
  - 1. Convolution Property

$$\left\{\overline{f^1 \otimes f^2}\right\}_l = \bar{f}_l^1\ \bar{f}_l^2$$

where $\left\{f^1 \otimes f^2\right\}_l \equiv \sum_{j=1}^{N} f_{l-j}^1 f_j^2$ extended by periodic summation

  - 2. Parseval's theorem:

$$N \sum_i |f_i|^2 = \sum_j |\bar{f}_j|^2$$

3. The Discreet Fourier Transform (DFT)

$$\bar{f}_m = \sum_{n=0}^{N-1} f_n \, e^{-2\pi i m n/N}$$

Inverse DFT

$$f_m = \frac{1}{N} \sum_{n=0}^{N-1} \bar{f}_n \, e^{2\pi i m n/N}$$

$$|f\rangle = (f_1, f_2, f_3 \cdots f_N) \;,\; f_n = f(n\Delta)$$

DFT

$$|\bar{f}\rangle = (\bar{f}_1, \bar{f}_2, \bar{f}_3 \cdots \bar{f}_N) \;,$$

Δ^{-1} = Sampling Rate
Δ = Sampling Interval

3. The Discreet Fourier Transform (DFT)

$$\bar{f}_m = \sum_{n=0}^{N-1} f_n \, e^{-2\pi i m n/N}$$

Inverse DFT

$$f_m = \frac{1}{N} \sum_{n=0}^{N-1} \bar{f}_n \, e^{2\pi i m n/N}$$

$$|f\rangle = (f_1, f_2, f_3 \cdots f_N) \;,\; f_n = f(n\Delta)$$

DFT

$$|\bar{f}\rangle = (\bar{f}_1, \bar{f}_2, \bar{f}_3 \cdots \bar{f}_N) \;,$$

Δ^{-1} = Sampling Rate
Δ = Sampling Interval

There is a critical frequency associated with the sampling rate

$$f_c = \frac{1}{2\Delta}$$ Nyquist frequency (Nyquist rate)

3. The Discreet Fourier Transform (DFT)

$$\bar{f}_m = \sum_{n=0}^{N-1} f_n \, e^{-2\pi i m n/N}$$

$$|f\rangle = (f_1, f_2, f_3 \ldots f_N) \;,\; f_n = f(n\Delta)$$

Δ^{-1} = Sampling Rate
Δ = Sampling Interval

DFT

$$|\bar{f}\rangle = (\bar{f}_1, \bar{f}_2, \bar{f}_3 \ldots \bar{f}_N) \;,$$

There is a critical frequency associated with the sampling rate

$f_c = \frac{1}{2\Delta}$ Nyquist frequency (Nyquist rate)

$$f(t) = \sin(2\pi f_c t)$$

Discretize and build the vector

$$f_n = \sin\left(2\pi f_c \times n\Delta\right)$$

3. The Discreet Fourier Transform (DFT)

$$\bar{f}_m = \sum_{n=0}^{N-1} f_n \, e^{-2\pi i m n/N}$$

$$|f\rangle = (f_1, f_2, f_3 \ldots f_N) \;,\; f_n = f(n\Delta)$$

Δ^{-1} = Sampling Rate
Δ = Sampling Interval

DFT

$$|\bar{f}\rangle = (\bar{f}_1, \bar{f}_2, \bar{f}_3 \ldots \bar{f}_N) \;,$$

There is a critical frequency associated with the sampling rate

$f_c = \frac{1}{2\Delta}$ Nyquist frequency (Nyquist rate)

$$f(t) = \sin(2\pi f_c t)$$

Discretize and build the vector

$$f_n = \sin\left(2\pi f_c \times n\Delta\right) = \sin n\pi = 0$$

3. The Discreet Fourier Transform (DFT)

$$\bar{f}_m = \sum_{n=0}^{N-1} f_n \, e^{-2\pi i m n / N}$$

$$|f\rangle = (f_1, f_2, f_3 \cdots f_N) \;,\; f_n = f(n\Delta)$$

Δ^{-1} = Sampling Rate
Δ = Sampling Interval

DFT

$$|\bar{f}\rangle = (\bar{f}_1, \bar{f}_2, \bar{f}_3 \cdots \bar{f}_N) \;,$$

There is a critical frequency associated with the sampling rate

$f_c = \frac{1}{2\Delta}$ Nyquist frequency (Nyquist rate)

- Any frequency in your data that is at Nyquist rate does not get detected !!
- Worse, any frequency that is more than the Nyquist rate **gives you a false frequency,** a phenomenon called ***aliasing***

3. The Discreet Fourier Transform (DFT)

$$\bar{f}_m = \sum_{n=0}^{N-1} f_n \, e^{-2\pi i m n / N}$$

There is a critical frequency associated with the sampling rate

Δ^{-1} = Sampling Rate
Δ = Sampling Interval

$$f_c = \frac{1}{2\Delta}$$ Nyquist frequency (Nyquist rate)

- Any frequency in your data that is at Nyquist rate does not get detected !!
- Worse, any frequency that is more than the Nyquist rate **gives you a false frequency,** a phenomenon called ***aliasing***

$e^{2\pi i f_1 t}$ $e^{2\pi i f_2 t}$ The total signal is being sampled at intervals Δ

3. The Discreet Fourier Transform (DFT)

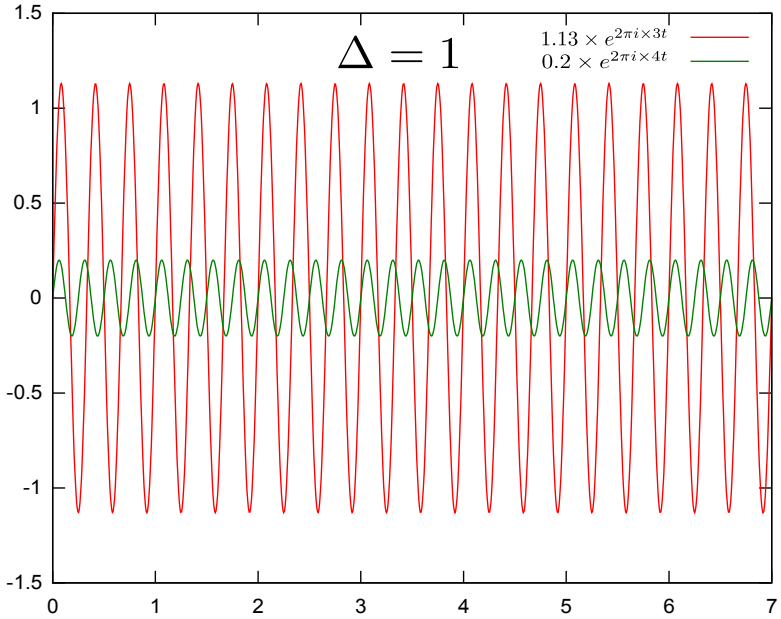
$$\bar{f}_m = \sum_{n=0}^{N-1} f_n \, e^{-2\pi i m n / N}$$

There is a critical frequency associated with the sampling rate

Δ^{-1} = Sampling Rate
Δ = Sampling Interval

$$f_c = \frac{1}{2\Delta}$$ Nyquist frequency (Nyquist rate)

- Any frequency in your data that is at Nyquist rate does not get detected !!
- Worse, any frequency that is more than the Nyquist rate **gives you a false frequency,** a phenomenon called ***aliasing***



The total signal is being sampled at intervals Δ

3. The Discreet Fourier Transform (DFT)

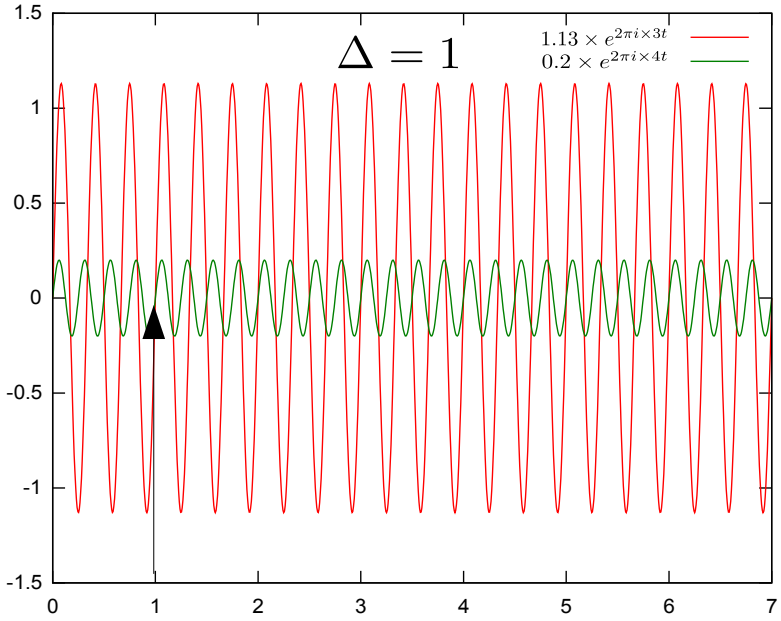
$$\bar{f}_m = \sum_{n=0}^{N-1} f_n \, e^{-2\pi i m n / N}$$

There is a critical frequency associated with the sampling rate

Δ^{-1} = Sampling Rate
Δ = Sampling Interval

$$f_c = \frac{1}{2\Delta}$$ Nyquist frequency (Nyquist rate)

- Any frequency in your data that is at Nyquist rate does not get detected !!
- Worse, any frequency that is more than the Nyquist rate **gives you a false frequency,** a phenomenon called ***aliasing***



The total signal is being sampled at intervals Δ

3. The Discreet Fourier Transform (DFT)

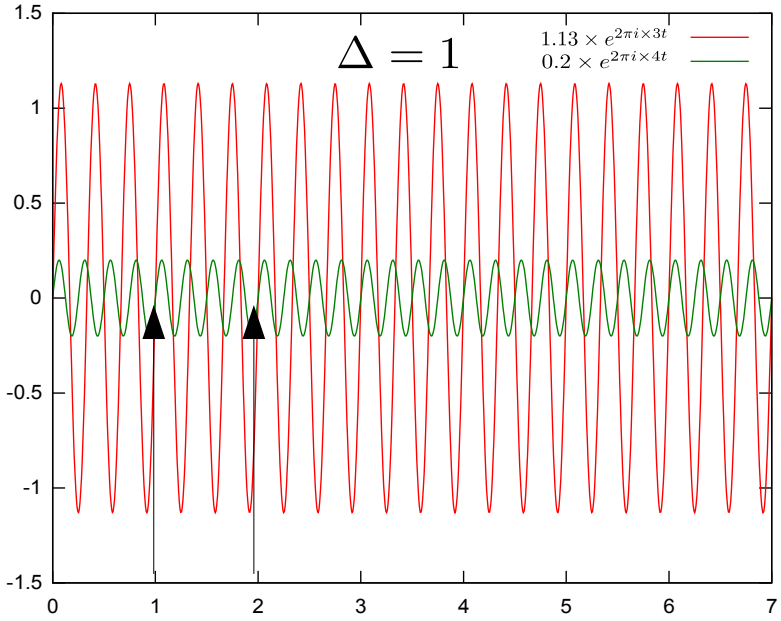
$$\bar{f}_m = \sum_{n=0}^{N-1} f_n \, e^{-2\pi i m n / N}$$

There is a critical frequency associated with the sampling rate

Δ^{-1} = Sampling Rate
Δ = Sampling Interval

$$f_c = \frac{1}{2\Delta}$$ Nyquist frequency (Nyquist rate)

- Any frequency in your data that is at Nyquist rate does not get detected !!
- Worse, any frequency that is more than the Nyquist rate **gives you a false frequency,** a phenomenon called ***aliasing***



The total signal is being sampled at intervals Δ

3. The Discreet Fourier Transform (DFT)

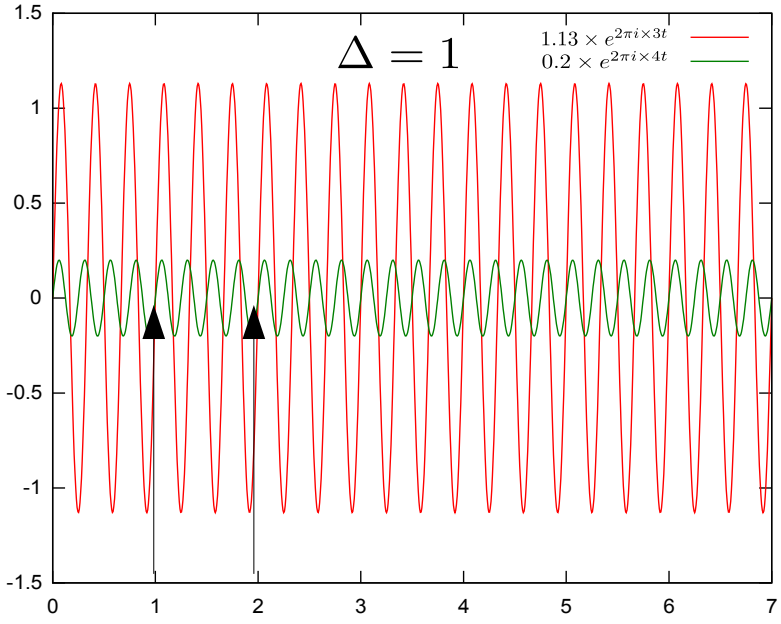
$$\bar{f}_m = \sum_{n=0}^{N-1} f_n \, e^{-2\pi i m n/N}$$

There is a critical frequency associated with the sampling rate

Δ^{-1} = Sampling Rate
Δ = Sampling Interval

$$f_c = \frac{1}{2\Delta}$$ Nyquist frequency (Nyquist rate)

- Any frequency in your data that is at Nyquist rate does not get detected !!
- Worse, any frequency that is more than the Nyquist rate **gives you a false frequency,** a phenomenon called ***aliasing***



The total signal is being sampled at intervals Δ

If the two frequencies differ by a multiple of the sampling rate, Then they give the same samples!

Sampling rate is just length of the interval $(-f_c, f_c)$

Any frequency outside this interval gets aliased into it

Hence the choice of k to lie between -N/2 and N/2

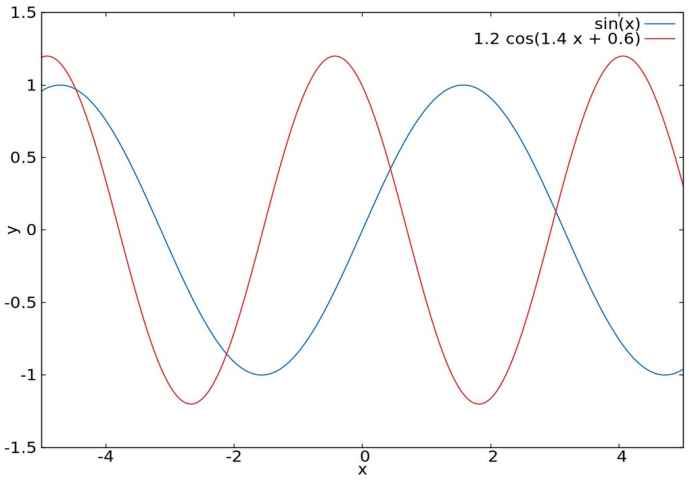
How to handle aliasing?

- Know the natural bandwidth of the signal
  If you already know something about what frequencies are there
  Make sure that your Nyquist rate is faster than the largest expected frequency

3. The Discreet Fourier Transform (DFT)

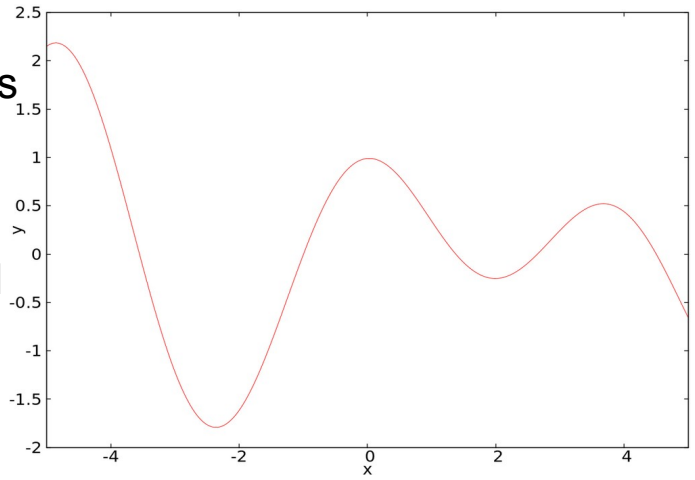
How to handle aliasing?

- Know the natural bandwidth of the signal
  If you already know something about what frequencies are there
  Make sure that your Nyquist rate is faster than the largest expected frequency



Add the components

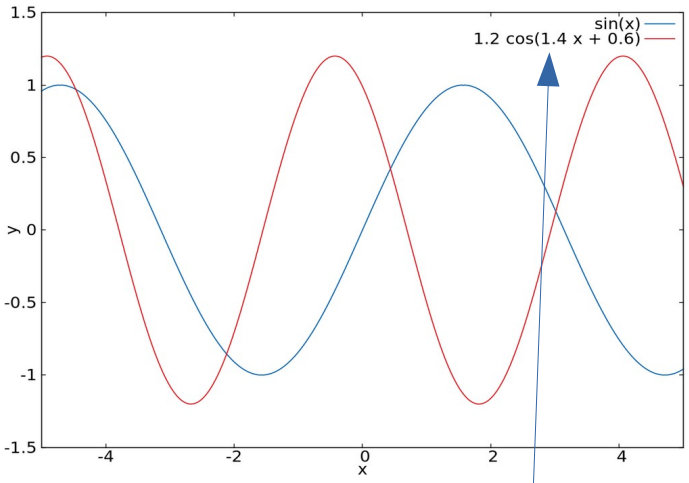
Resolve the signal



3. The Discreet Fourier Transform (DFT)

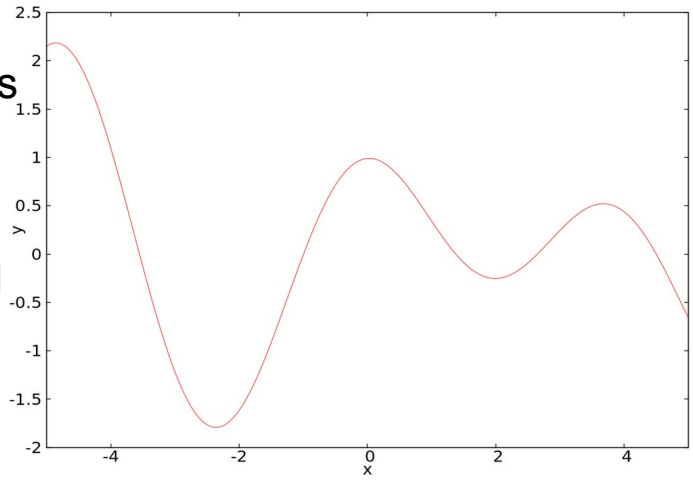
How to handle aliasing?

- Know the natural bandwidth of the signal
  If you already know something about what frequencies are there
  Make sure that your Nyquist rate is faster than the largest expected frequency



Add the components

Resolve the signal



$$\frac{1}{2\Delta} = f_c \gg \frac{1.4}{2\pi}$$

$$\Delta \ll \frac{\pi}{1.4} \approx 2.2$$

How to handle aliasing?

- Know the natural bandwidth of the signal
  If you already know something about what frequencies are there
  Make sure that your Nyquist rate is faster than the largest expected frequency

- Choose a Nyquist rate, and sample the data accordingly
  If the FFT is too large at the Nyquist edges, then you're aliasing high frequencies
  Increase the Nyquist rate and re-sample
  Keep doing this until the FFT at the Nyquist edges is small enough to ignore.

3. The Discreet Fourier Transform (DFT)

How to handle aliasing?

- Know the natural bandwidth of the signal
  If you already know something about what frequencies are there
  Make sure that your Nyquist rate is faster than the largest expected frequency

- Choose a Nyquist rate, and sample the data accordingly
  If the FFT is too large at the Nyquist edges, then you're aliasing high frequencies
  Increase the Nyquist rate and re-sample
  Keep doing this until the FFT at the Nyquist edges is small enough to ignore.

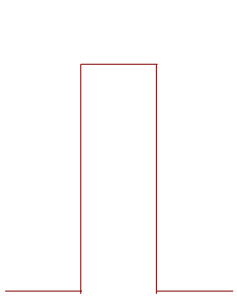
Infinitely extended signals with (quasi) periodic behavior are not expected to have too many frequencies (here, only 2)

3. The Discreet Fourier Transform (DFT)

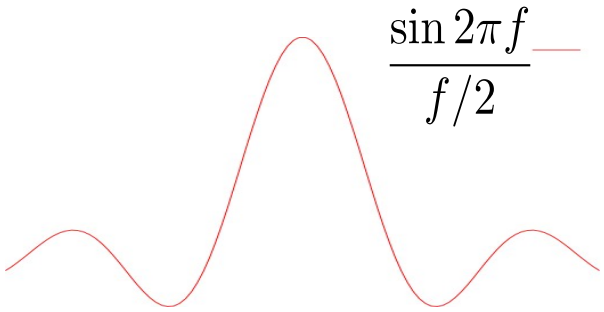
How to handle aliasing?

- Know the natural bandwidth of the signal
  If you already know something about what frequencies are there
  Make sure that your Nyquist rate is faster than the largest expected frequency

- Choose a Nyquist rate, and sample the data accordingly
  If the FFT is too large at the Nyquist edges, then you're aliasing high frequencies
  Increase the Nyquist rate and re-sample
  Keep doing this until the FFT at the Nyquist edges is small enough to ignore.



Brightness Image



Fourier Transform

Infinitely extended signals with (quasi) periodic behavior are not expected to have too many frequencies (here, only 2)

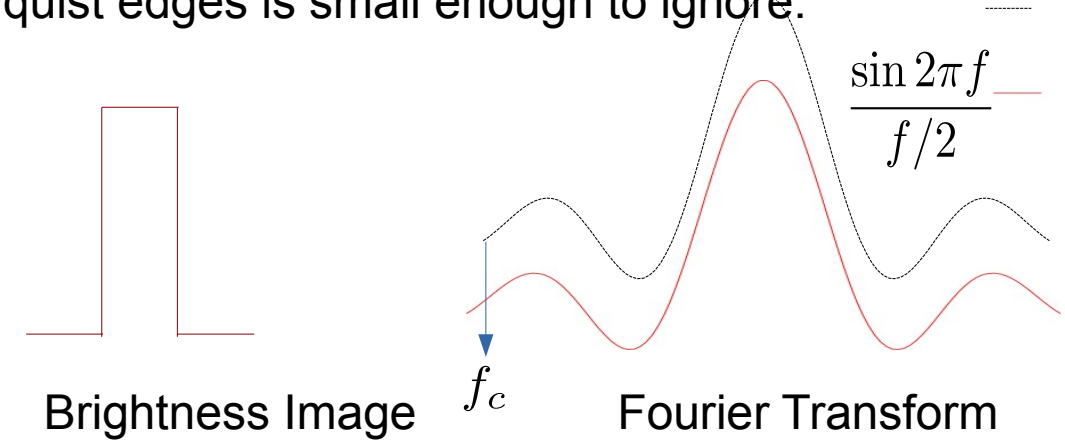
Finite signals, will have many many frequencies (here, a full continuum) - larger frequencies have smaller amplitudes

3. The Discreet Fourier Transform (DFT)

How to handle aliasing?

- Know the natural bandwidth of the signal
  If you already know something about what frequencies are there
  Make sure that your Nyquist rate is faster than the largest expected frequency

- Choose a Nyquist rate, and sample the data accordingly
  If the FFT is too large at the Nyquist edges, then you're aliasing high frequencies
  Increase the Nyquist rate and re-sample
  Keep doing this until the FFT at the Nyquist edges is small enough to ignore.



Infinitely extended signals with (quasi) periodic behavior are not expected to have too many frequencies (here, only 2)

Finite signals, will have many many frequencies (here, a full continuum) - larger frequencies have smaller amplitudes

Choose a sample rate. Sample. Take FFT and re - sample until aliasing falls off

3. The Discreet Fourier Transform (DFT)

$$\bar{f}_m = \sum_{n=0}^{N-1} f_n \, e^{-2\pi i m n/N}$$

$$\begin{aligned}
\bar{f}_0 &= e^{-2\pi i\, 0\times 0/N} f_0 + e^{-2\pi i\, 0\times 1/N} f_1 + e^{-2\pi i\, 0\times 2/N} f_2 + \ldots e^{-2\pi i\, 0\times (N-1)/N} f_{N-1} \\
\bar{f}_1 &= e^{-2\pi i\, 1\times 0/N} f_0 + e^{-2\pi i\, 1\times 1/N} f_1 + e^{-2\pi i\, 1\times 2/N} f_2 + \ldots e^{-2\pi i\, 1\times (N-1)/N} f_{N-1} \\
\bar{f}_2 &= e^{-2\pi i\, 2\times 0/N} f_0 + e^{-2\pi i\, 2\times 1/N} f_1 + e^{-2\pi i\, 2\times 2/N} f_2 + \ldots e^{-2\pi i\, 2\times (N-1)/N} f_{N-1} \\
\vdots &\quad \vdots \qquad\qquad\qquad\qquad \ddots \qquad\qquad\qquad\qquad \vdots
\end{aligned}$$

3. The Discreet Fourier Transform (DFT)

$$\bar{f}_m = \sum_{n=0}^{N-1} f_n \, e^{-2\pi i m n/N}$$

$$\begin{pmatrix} \bar{f}_0 \\ \bar{f}_1 \\ \vdots \\ \bar{f}_{N-1} \end{pmatrix} = \begin{pmatrix} e^{-2\pi i \, 0\times 0/N} & e^{-2\pi i \, 0\times 1/N} & \dots & e^{-2\pi i \, 0\times (N-1)/N} \\ e^{-2\pi i \, 1\times 0/N} & e^{-2\pi i \, 1\times 1/N} & \dots & e^{-2\pi i \, 1\times (N-1)/N} \\ \vdots & \vdots & \ddots & \vdots \\ e^{-2\pi i \, (N-1)\times 0/N} & e^{-2\pi i \, (N-1)\times 1/N} & \dots & e^{-2\pi i \, (N-1)\times (N-1)/N} \end{pmatrix} \begin{pmatrix} f_0 \\ f_1 \\ \vdots \\ f_{N-1} \end{pmatrix}$$

$$U$$

$$|\bar{f}\rangle = U \cdot |f\rangle \qquad U_{mn} = \exp\left(-2\pi i m n/N\right)$$

3. The Discreet Fourier Transform (DFT)

$$\bar{f}_m = \sum_{n=0}^{N-1} f_n \, e^{-2\pi i m n/N}$$

$$\begin{pmatrix} \bar{f}_0 \\ \bar{f}_1 \\ \vdots \\ \bar{f}_{N-1} \end{pmatrix} = \begin{pmatrix} e^{-2\pi i \, 0\times 0/N} & e^{-2\pi i \, 0\times 1/N} & \dots & e^{-2\pi i \, 0\times (N-1)/N} \\ e^{-2\pi i \, 1\times 0/N} & e^{-2\pi i \, 1\times 1/N} & \dots & e^{-2\pi i \, 1\times (N-1)/N} \\ \vdots & \vdots & \ddots & \vdots \\ e^{-2\pi i \, (N-1)\times 0/N} & e^{-2\pi i \, (N-1)\times 1/N} & \dots & e^{-2\pi i \, (N-1)\times (N-1)/N} \end{pmatrix} \begin{pmatrix} f_0 \\ f_1 \\ \vdots \\ f_{N-1} \end{pmatrix}$$

$$U$$

$$|\bar{f}\rangle = U \cdot |f\rangle \qquad U_{mn} = \exp\left(-2\pi i m n/N\right)$$

- Thus, DFT is basically just a Linear Algebra problem!

3. The Discreet Fourier Transform (DFT)

$$\bar{f}_m = \sum_{n=0}^{N-1} f_n \, e^{-2\pi i m n/N}$$

$$\begin{pmatrix} \bar{f}_0 \\ \bar{f}_1 \\ \vdots \\ \bar{f}_{N-1} \end{pmatrix} = \begin{pmatrix} e^{-2\pi i \, 0\times 0/N} & e^{-2\pi i \, 0\times 1/N} & \dots & e^{-2\pi i \, 0\times (N-1)/N} \\ e^{-2\pi i \, 1\times 0/N} & e^{-2\pi i \, 1\times 1/N} & \dots & e^{-2\pi i \, 1\times (N-1)/N} \\ \vdots & \vdots & \ddots & \vdots \\ e^{-2\pi i \, (N-1)\times 0/N} & e^{-2\pi i \, (N-1)\times 1/N} & \dots & e^{-2\pi i \, (N-1)\times (N-1)/N} \end{pmatrix} \begin{pmatrix} f_0 \\ f_1 \\ \vdots \\ f_{N-1} \end{pmatrix}$$

$$U$$

$$|\bar{f}\rangle = U \cdot |f\rangle \qquad U_{mn} = \exp\left(-2\pi i m n/N\right)$$

- Thus, DFT is basically just a Linear Algebra problem!
- The numpy and scipy modules already have routines for matrices.

3. The Discreet Fourier Transform (DFT)

$$\bar{f}_m = \sum_{n=0}^{N-1} f_n \, e^{-2\pi i m n/N}$$

$$\begin{pmatrix} \bar{f}_0 \\ \bar{f}_1 \\ \vdots \\ \bar{f}_{N-1} \end{pmatrix} = \begin{pmatrix} e^{-2\pi i \, 0\times 0/N} & e^{-2\pi i \, 0\times 1/N} & \dots & e^{-2\pi i \, 0\times (N-1)/N} \\ e^{-2\pi i \, 1\times 0/N} & e^{-2\pi i \, 1\times 1/N} & \dots & e^{-2\pi i \, 1\times (N-1)/N} \\ \vdots & \vdots & \ddots & \vdots \\ e^{-2\pi i \, (N-1)\times 0/N} & e^{-2\pi i \, (N-1)\times 1/N} & \dots & e^{-2\pi i \, (N-1)\times (N-1)/N} \end{pmatrix} \begin{pmatrix} f_0 \\ f_1 \\ \vdots \\ f_{N-1} \end{pmatrix}$$

$$U$$

$$|\bar{f}\rangle = U \cdot |f\rangle \qquad U_{mn} = \exp\left(-2\pi i m n/N\right)$$

- Thus, DFT is basically just a Linear Algebra problem!
- The numpy and scipy modules already have routines for matrices.
- So, are we done? Not quite.

4. The Fast Fourier Transform (FFT)

$$\bar{f}(\omega) = \int_{-\infty}^{\infty} \mathrm{d}t \; f(t) \; e^{-i\omega t}$$

$$|f\rangle = (f_1, f_2, f_3 \; \cdots \; f_N) \;\; , \;\; f_n = f(n\Delta)$$

$\Delta^{-1} =$ Sampling Rate

DFT

$$|\bar{f}\rangle = (\bar{f}_1, \bar{f}_2, \bar{f}_3 \; \cdots \; \bar{f}_N) \;\; , \;\; \bar{f}_n = \sum_{n=1}^{N} f_n \; e^{-2\pi i m n/N} \;\; ???$$

$$|\bar{f}\rangle = U \cdot |f\rangle$$

Is there a faster way than to do this brute force matrix-vector operation?

4. The Fast Fourier Transform (FFT)

$$\bar{f}(\omega) = \int_{-\infty}^{\infty} \mathrm{d}t \; f(t) \; e^{-i\omega t}$$

$$|f\rangle = (f_1, f_2, f_3 \; \cdots \; f_N) \;\; , \;\; f_n = f(n\Delta)$$

$\Delta^{-1} =$ Sampling Rate

FFT

$$|\bar{f}\rangle = (\bar{f}_1, \bar{f}_2, \bar{f}_3 \; \cdots \; \bar{f}_N) \;\; , \;\; \bar{f}_n = \sum_{n=1}^{N} f_n \; e^{-2\pi i m n/N} \;\; ???$$

$$|\bar{f}\rangle = U \cdot |f\rangle$$

Is there a faster way than to do this brute force matrix-vector operation?

Yes, there is! It's called the FFT algorithm.

4. The Fast Fourier Transform (FFT)

$$\bar{f}_m = \sum_{n=0}^{N-1} f_n \, e^{-2\pi i m n/N}$$

$$\begin{pmatrix} \bar{f}_0 \\ \bar{f}_1 \\ \vdots \end{pmatrix} = \begin{pmatrix} e^{-2\pi i \, 0\times 0/N} & e^{-2\pi i \, 0\times 1/N} & \cdots & e^{-2\pi i \, 0\times N-1/N} \\ e^{-2\pi i \, 1\times 0/N} & e^{-2\pi i \, 1\times 1/N} & \cdots & e^{-2\pi i \, 1\times N-1/N} \\ e^{-2\pi i \, 2\times 0/N} & e^{-2\pi i \, 2\times 1/N} & \cdots & e^{-2\pi i \, 2\times N-1/N} \\ \vdots & \vdots & \ddots & \vdots \end{pmatrix} \cdot \begin{pmatrix} f_0 \\ f_1 \\ \vdots \end{pmatrix}$$

As a special case, consider N=4

$$\begin{pmatrix} \bar{f}_0 \\ \bar{f}_1 \\ \bar{f}_2 \\ \bar{f}_3 \end{pmatrix} = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & -i & -1 & i \\ 1 & -1 & 1 & -1 \\ 1 & i & -1 & -i \end{pmatrix} \cdot \begin{pmatrix} f_0 \\ f_1 \\ f_2 \\ f_3 \end{pmatrix} =$$

4. The Fast Fourier Transform (FFT)

$$\bar{f}_m = \sum_{n=0}^{N-1} f_n \, e^{-2\pi i m n/N}$$

$$\begin{pmatrix} \bar{f}_0 \\ \bar{f}_1 \\ \vdots \end{pmatrix} = \begin{pmatrix} e^{-2\pi i \, 0\times 0/N} & e^{-2\pi i \, 0\times 1/N} & \dots & e^{-2\pi i \, 0\times N-1/N} \\ e^{-2\pi i \, 1\times 0/N} & e^{-2\pi i \, 1\times 1/N} & \dots & e^{-2\pi i \, 1\times N-1/N} \\ e^{-2\pi i \, 2\times 0/N} & e^{-2\pi i \, 2\times 1/N} & \dots & e^{-2\pi i \, 2\times N-1/N} \\ \vdots & \vdots & \ddots & \vdots \end{pmatrix} \cdot \begin{pmatrix} f_0 \\ f_1 \\ \vdots \end{pmatrix}$$

As a special case, consider N=4

$$\begin{pmatrix} \bar{f}_0 \\ \bar{f}_1 \\ \bar{f}_2 \\ \bar{f}_3 \end{pmatrix} = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & -i & -1 & i \\ 1 & -1 & 1 & -1 \\ 1 & i & -1 & -i \end{pmatrix} \cdot \begin{pmatrix} f_0 \\ f_1 \\ f_2 \\ f_3 \end{pmatrix} = \begin{pmatrix} f_0 + f_1 + f_2 + f_3 \\ f_0 - if_1 - f_2 + if_3 \\ f_0 - f_1 + f_2 - f_3 \\ f_0 + if_1 - f_2 - if_3 \end{pmatrix}$$

4. The Fast Fourier Transform (FFT)

$$\bar{f}_m = \sum_{n=0}^{N-1} f_n \, e^{-2\pi i m n/N}$$

$$\begin{pmatrix} \bar{f}_0 \\ \bar{f}_1 \\ \vdots \end{pmatrix} = \begin{pmatrix} e^{-2\pi i \, 0\times 0/N} & e^{-2\pi i \, 0\times 1/N} & \cdots & e^{-2\pi i \, 0\times N-1/N} \\ e^{-2\pi i \, 1\times 0/N} & e^{-2\pi i \, 1\times 1/N} & \cdots & e^{-2\pi i \, 1\times N-1/N} \\ e^{-2\pi i \, 2\times 0/N} & e^{-2\pi i \, 2\times 1/N} & \cdots & e^{-2\pi i \, 2\times N-1/N} \\ \vdots & \vdots & \ddots & \vdots \end{pmatrix} \cdot \begin{pmatrix} f_0 \\ f_1 \\ \vdots \end{pmatrix}$$

As a special case, consider N=4

16 multiplications and 12 additions!

$$\begin{pmatrix} \bar{f}_0 \\ \bar{f}_1 \\ \bar{f}_2 \\ \bar{f}_3 \end{pmatrix} = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & -i & -1 & i \\ 1 & -1 & 1 & -1 \\ 1 & i & -1 & -i \end{pmatrix} \cdot \begin{pmatrix} f_0 \\ f_1 \\ f_2 \\ f_3 \end{pmatrix} = \begin{pmatrix} f_0 + f_1 + f_2 + f_3 \\ f_0 - if_1 - f_2 + if_3 \\ f_0 - f_1 + f_2 - f_3 \\ f_0 + if_1 - f_2 - if_3 \end{pmatrix}$$

4. The Fast Fourier Transform (FFT)

$$\bar{f}_m = \sum_{n=0}^{N-1} f_n \, e^{-2\pi i m n/N}$$

$$\begin{pmatrix} \bar{f}_0 \\ \bar{f}_1 \\ \vdots \end{pmatrix} = \begin{pmatrix} e^{-2\pi i \, 0\times 0/N} & e^{-2\pi i \, 0\times 1/N} & \dots & e^{-2\pi i \, 0\times N-1/N} \\ e^{-2\pi i \, 1\times 0/N} & e^{-2\pi i \, 1\times 1/N} & \dots & e^{-2\pi i \, 1\times N-1/N} \\ e^{-2\pi i \, 2\times 0/N} & e^{-2\pi i \, 2\times 1/N} & \dots & e^{-2\pi i \, 2\times N-1/N} \\ \vdots & \vdots & \ddots & \vdots \end{pmatrix} \cdot \begin{pmatrix} f_0 \\ f_1 \\ \vdots \end{pmatrix}$$

As a special case, consider N=4

$$\begin{pmatrix} \bar{f}_0 \\ \bar{f}_1 \\ \bar{f}_2 \\ \bar{f}_3 \end{pmatrix} = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & -i & -1 & i \\ 1 & -1 & 1 & -1 \\ 1 & i & -1 & -i \end{pmatrix} \cdot \begin{pmatrix} f_0 \\ f_1 \\ f_2 \\ f_3 \end{pmatrix} = \begin{pmatrix} (f_0 + f_2) + (f_1 + f_3) \\ (f_0 - f_2) - i(f_1 - f_3) \\ (f_0 + f_2) - (f_1 + f_3) \\ (f_0 - f_2) + i(f_1 - f_3) \end{pmatrix}$$

4. The Fast Fourier Transform (FFT)

$$\bar{f}_m = \sum_{n=0}^{N-1} f_n \, e^{-2\pi i m n/N}$$

$$\begin{pmatrix} \bar{f}_0 \\ \bar{f}_1 \\ \vdots \end{pmatrix} = \begin{pmatrix} e^{-2\pi i \, 0\times 0/N} & e^{-2\pi i \, 0\times 1/N} & \cdots & e^{-2\pi i \, 0\times N-1/N} \\ e^{-2\pi i \, 1\times 0/N} & e^{-2\pi i \, 1\times 1/N} & \cdots & e^{-2\pi i \, 1\times N-1/N} \\ e^{-2\pi i \, 2\times 0/N} & e^{-2\pi i \, 2\times 1/N} & \cdots & e^{-2\pi i \, 2\times N-1/N} \\ \vdots & \vdots & \ddots & \vdots \end{pmatrix} \cdot \begin{pmatrix} f_0 \\ f_1 \\ \vdots \end{pmatrix}$$

As a special case, consider N=4

2 multiplications and 8 additions!
Much faster this way!!!

$$\begin{pmatrix} \bar{f}_0 \\ \bar{f}_1 \\ \bar{f}_2 \\ \bar{f}_3 \end{pmatrix} = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & -i & -1 & i \\ 1 & -1 & 1 & -1 \\ 1 & i & -1 & -i \end{pmatrix} \cdot \begin{pmatrix} f_0 \\ f_1 \\ f_2 \\ f_3 \end{pmatrix} = \begin{pmatrix} (f_0 + f_2) + (f_1 + f_3) \\ (f_0 - f_2) - i(f_1 - f_3) \\ (f_0 + f_2) - (f_1 + f_3) \\ (f_0 - f_2) + i(f_1 - f_3) \end{pmatrix}$$

4. The Fast Fourier Transform (FFT)

$$\bar{f}_m = \sum_{n=0}^{N-1} f_n \, e^{-2\pi i m n/N}$$

As a special case, consider N=4

2 multiplications and 8 additions!
Much faster this way!!!

$$\begin{pmatrix} \bar{f}_0 \\ \bar{f}_1 \\ \bar{f}_2 \\ \bar{f}_3 \end{pmatrix} = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & -i & -1 & i \\ 1 & -1 & 1 & -1 \\ 1 & i & -1 & -i \end{pmatrix} \cdot \begin{pmatrix} f_0 \\ f_1 \\ f_2 \\ f_3 \end{pmatrix} = \begin{pmatrix} (f_0 + f_2) + (f_1 + f_3) \\ (f_0 - f_2) - i(f_1 - f_3) \\ (f_0 + f_2) - (f_1 + f_3) \\ (f_0 - f_2) + i(f_1 - f_3) \end{pmatrix}$$

This observation may reduce the computational effort from $\mathcal{O}(N^2)$ into $\mathcal{O}(N \log_2 N)$

Much, much faster! Because $\lim_{N\to\infty} \frac{\log_2 N}{N} = 0$

4. The Fast Fourier Transform (FFT)

$$\bar{f}_m = \sum_{n=0}^{N-1} f_n \, e^{-2\pi i m n/N}$$

$$\begin{pmatrix} \bar{f}_0 \\ \bar{f}_1 \\ \vdots \end{pmatrix} = \begin{pmatrix} e^{-2\pi i \, 0\times 0/N} & e^{-2\pi i \, 0\times 1/N} & \dots & e^{-2\pi i \, 0\times N-1/N} \\ e^{-2\pi i \, 1\times 0/N} & e^{-2\pi i \, 1\times 1/N} & \dots & e^{-2\pi i \, 1\times N-1/N} \\ e^{-2\pi i \, 2\times 0/N} & e^{-2\pi i \, 2\times 1/N} & \dots & e^{-2\pi i \, 2\times N-1/N} \\ \vdots & \vdots & \ddots & \vdots \end{pmatrix} \cdot \begin{pmatrix} f_0 \\ f_1 \\ \vdots \end{pmatrix}$$

- The U matrix above consists only of roots of 1 *i.e.* 1, -1 & $\pm i$
- Thus, all we need to do is group those terms in f that multiply with $1, -1, i, \& -i$ respectively
- Add (or subtract) them as need be, then multiply them with $i, \& -i$ and add/subtract the results

4. The Fast Fourier Transform (FFT)

$$\bar{f}_m = \sum_{n=0}^{N-1} f_n \, e^{-2\pi i m n/N}$$

- The U matrix above consists only of 1s and roots of -1 i.e. $\pm i$
- Thus, all we need to do is group those terms in f that multiply with $1, -1, i, \& -i$ respectively

$$\bar{f}_m = \sum_{n=0}^{N-1} f_n \, e^{-2\pi i m n/N}$$

$$\sum_{j=0}^{N/2-1} f_{2j} \, e^{-2\pi i m \, (2j)/N} + \sum_{j=0}^{N/2-1} f_{2j+1} \, e^{-2\pi i m \, (2j+1)/N}$$

1. Separate the odd and even indices in the sum

4. The Fast Fourier Transform (FFT)

$$\bar{f}_m = \sum_{n=0}^{N-1} f_n \, e^{-2\pi i m n/N}$$

- The U matrix above consists only of 1s and roots of -1 i.e. $\pm i$
- Thus, all we need to do is group those terms in f that multiply with $1, -1, i, \& -i$ respectively

$$\bar{f}_m = \sum_{n=0}^{N-1} f_n \, e^{-2\pi i m n/N}$$

$$\sum_{j=0}^{N/2-1} f_{2j} \, e^{-2\pi i m \, (j)/(N/2)} + \sum_{j=0}^{N/2-1} f_{2j+1} \, e^{-2\pi i m \, (j)/(N/2)} \left(e^{-2\pi i/N}\right)^n$$

2. Rearrange terms in the exponent so N changes to N/2 and a root of -1 comes out

4. The Fast Fourier Transform (FFT)

$$\bar{f}_m = \sum_{n=0}^{N-1} f_n \, e^{-2\pi i m n/N}$$

- The U matrix above consists only of 1s and roots of -1 i.e. $\pm i$
- Thus, all we need to do is group those terms in f that multiply with $1, -1, i, \& -i$ respectively

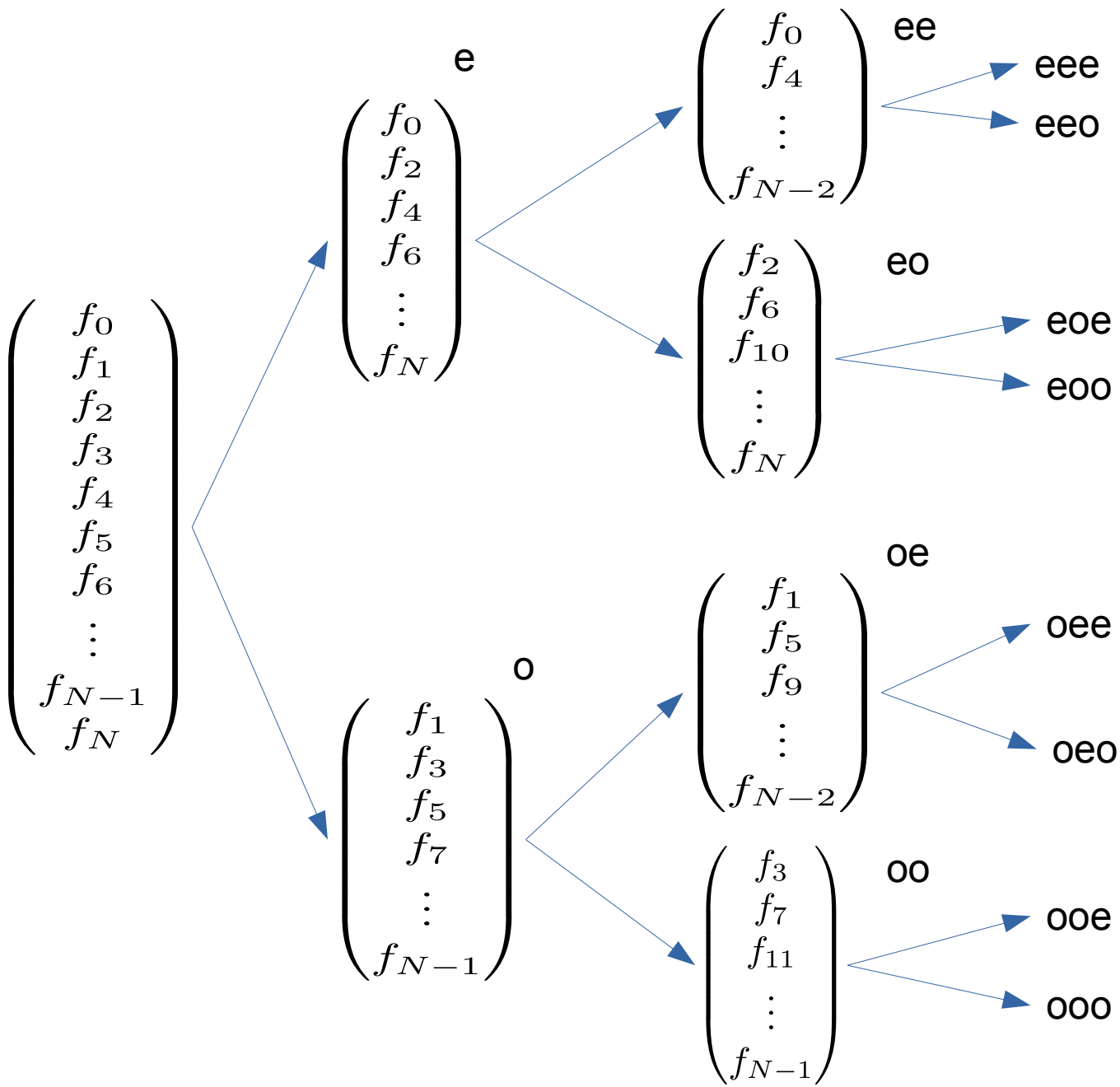
$$\bar{f}_m = \sum_{n=0}^{N-1} f_n \, e^{-2\pi i m n/N}$$

Two smaller DFT's:
Danielson Lanczos lemma (1942!)
Based on earlier works by Runge (1903)

$$\sum_{j=0}^{N/2-1} f_{2j} \, e^{-2\pi i m\,(j)/(N/2)} + \left(e^{-2\pi i/N}\right)^n \sum_{j=0}^{N/2-1} f_{2j+1} \, e^{-2\pi i m\,(j)/(N/2)}$$

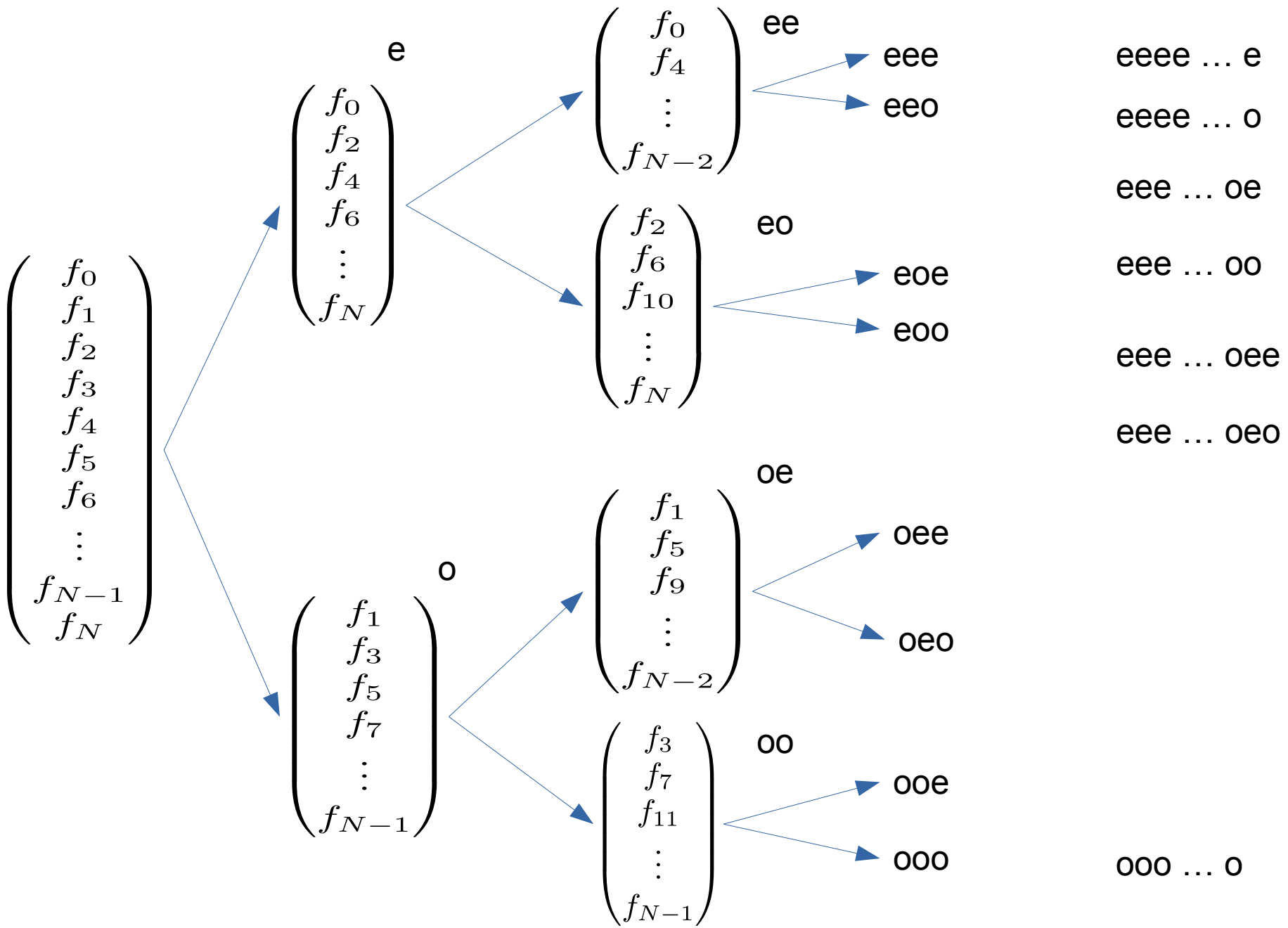
3. Split the FFT to a combination of two smaller FFT's

4. The Fast Fourier Transform (FFT)



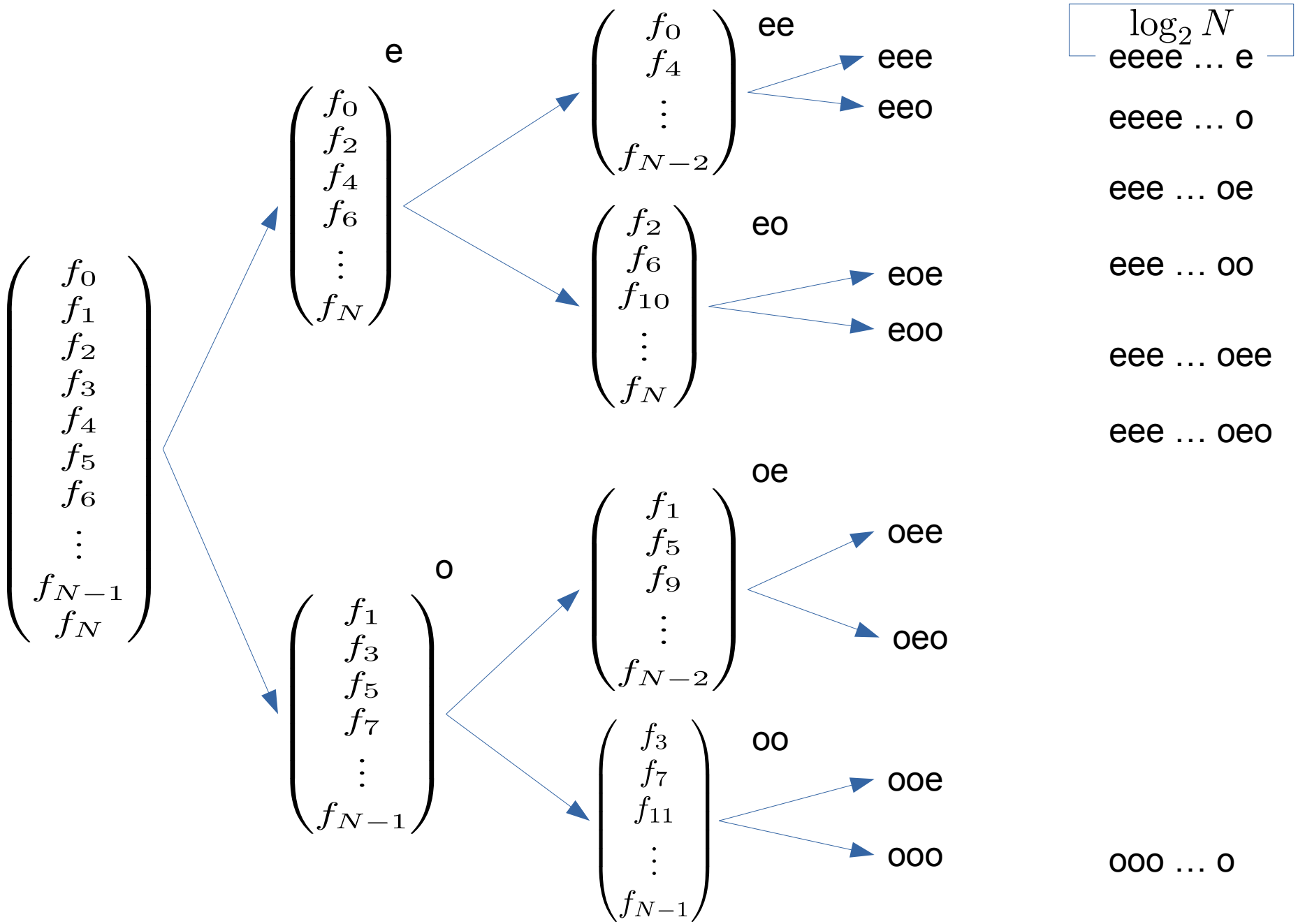
4. Repeat recursively until you wind up with only vectors of length 1by rearranging in bit reversed order of index

4. The Fast Fourier Transform (FFT)



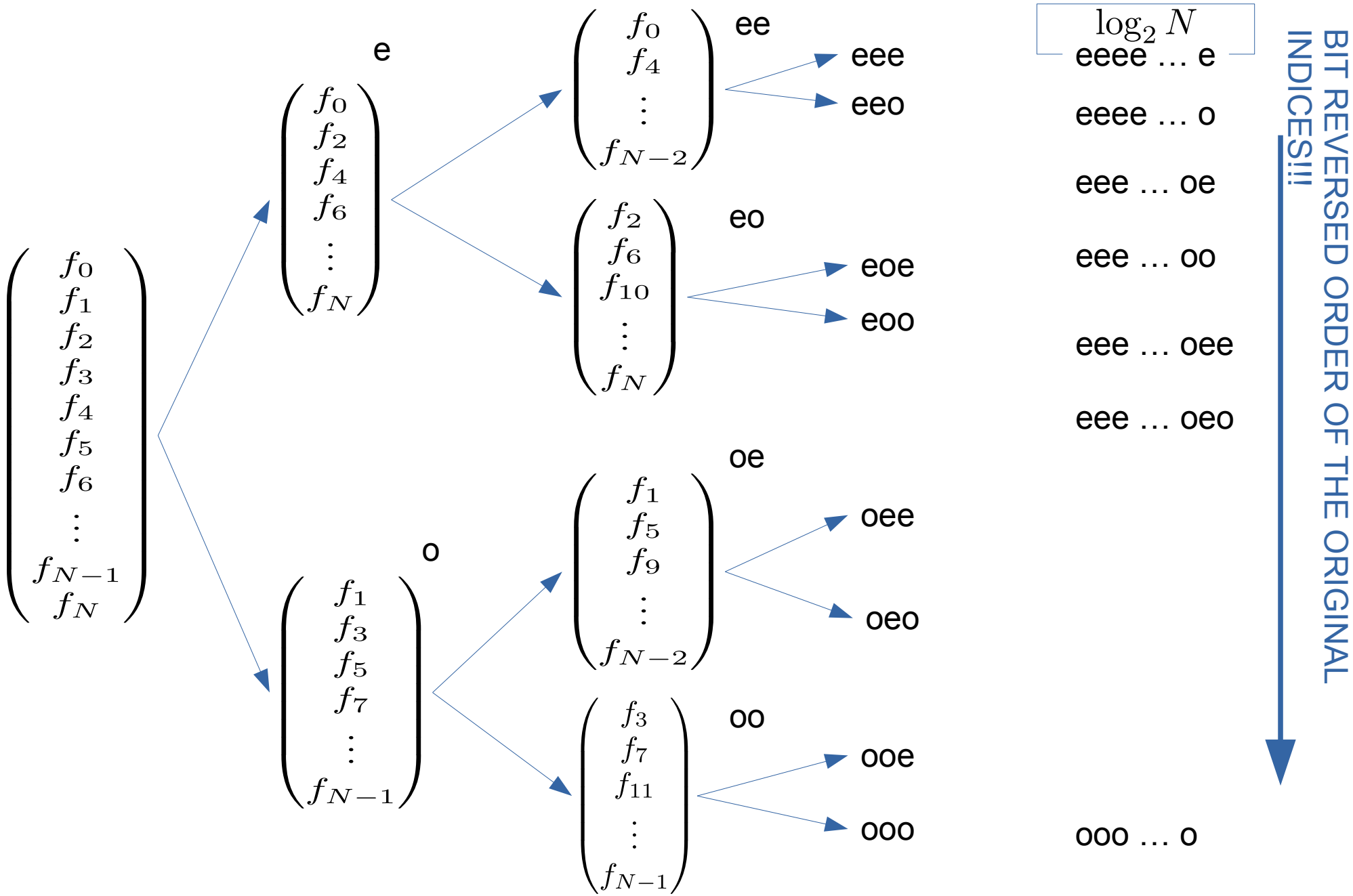
4. Repeat recursively until you wind up with only vectors of length 1by rearranging in bit reversed order of index

4. The Fast Fourier Transform (FFT)



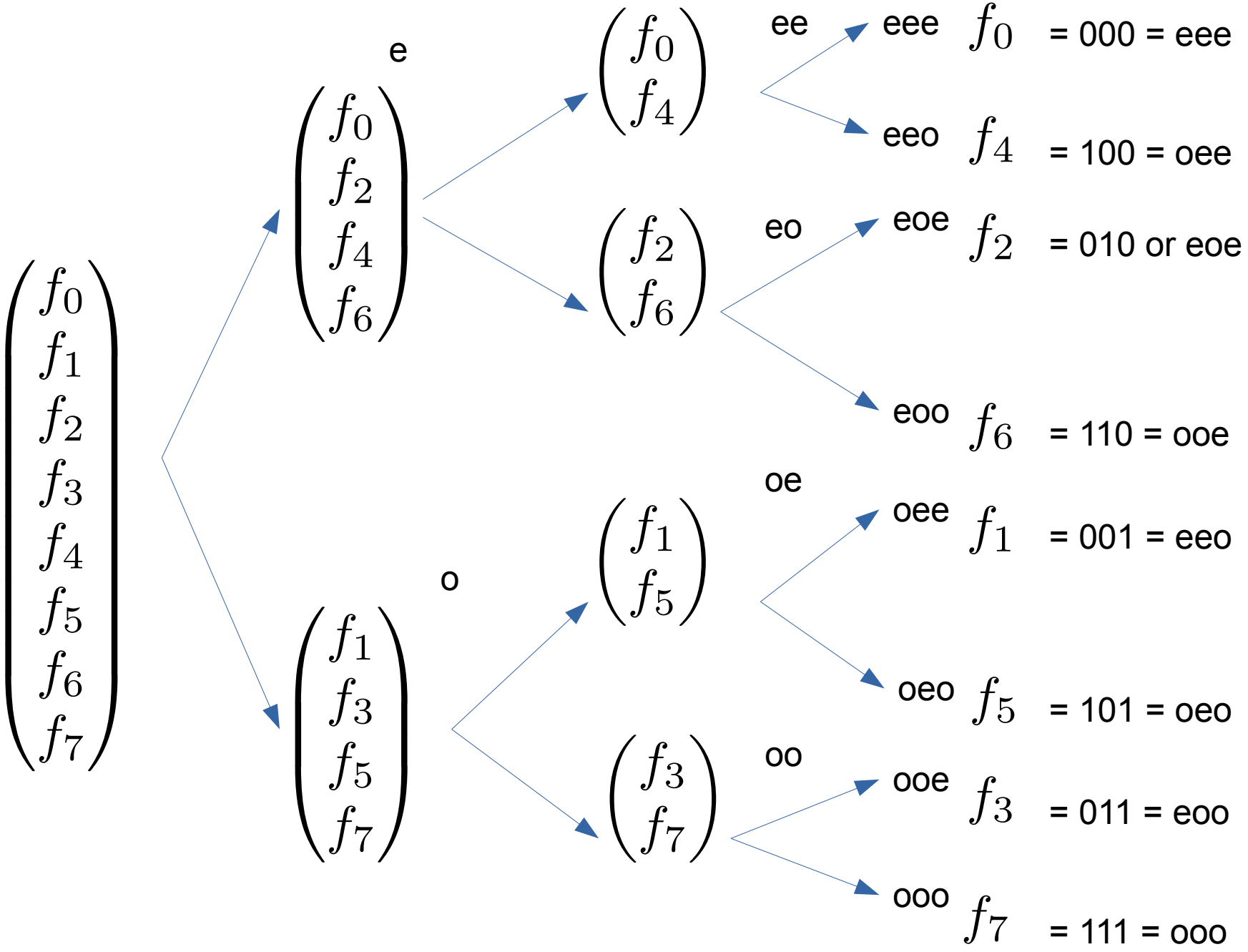
4. Repeat recursively until you wind up with only vectors of length 1by rearranging in bit reversed order of index

4. The Fast Fourier Transform (FFT)



4. Repeat recursively until you wind up with only vectors of length 1by rearranging in bit reversed order of index

4. The Fast Fourier Transform (FFT)



4. The Fast Fourier Transform (FFT)

$$\bar{f}_m = \sum_{n=0}^{N-1} f_n \, e^{-2\pi i m n/N}$$

Danielson-lanczos lemma (recursive)

$$\sum_{j=0}^{N/2-1} f_{2j} \, e^{-2\pi i m\,(j)/(N/2)} + \left(e^{-2\pi i/N}\right)^n \sum_{j=0}^{N/2-1} f_{2j+1} \, e^{-2\pi i m\,(j)/(N/2)}$$

What have we done to obtain FFT analytically?

1. Separate the odd and even indices in the sum
2. Rearrange terms in the exponent so N changes to N/2 and a root of -1 comes out
3. Split the FFT to a combo of two smaller FFT's (Danielson – Lanczos lemma)
4. Repeat recursively until you wind up with only vectors of length 1 by rearranging in bit reversed order of index

How should FFT be done numerically ?

1. Rearrange the vector in bit reversed order of index
2. Apply the Danielson - Lanzcos lemma to get the previous combination
3. Repeat this $\log_2 N$ times until full FFT vector is obtained. Only $N \log_2 N$ multiplications!

4. The Fast Fourier Transform (FFT)

$$\bar{f}_m = \sum_{n=0}^{N-1} f_n \, e^{-2\pi i m n / N}$$

Danielson-lanczos lemma (recursive)

$$\sum_{j=0}^{N/2-1} f_{2j} \, e^{-2\pi i m \, (j)/(N/2)} + \left(e^{-2\pi i/N}\right)^n \sum_{j=0}^{N/2-1} f_{2j+1} \, e^{-2\pi i m \, (j)/(N/2)}$$

FFT Algorithm in pseudocode:

The Cooley-Tukey algorithm (1965) uses the bit reversal idea
Gauss had already come up with this in the 19th century (1866), but C&T made it widely known

```
algorithm iterative-fft is
    input: Array a of n complex values where n is a power of 2.
    output: Array A the DFT of a.

    bit-reverse-copy(a, A)
    n ← a.length
    for s = 1 to log(n) do
        m ← 2s
        ωm ← exp(-2πi/m)
        for k = 0 to n-1 by m do
            ω ← 1
            for j = 0 to m/2 - 1 do
                t ← ω A[k + j + m/2]
                u ← A[k + j]
                A[k + j] ← u + t
                A[k + j + m/2] ← u - t
                ω ← ω ωm

    return A

algorithm bit-reverse-copy(a,A) is
    input: Array a of n complex values where n is a power of 2.
    output: Array A of size n.

    n ← a.length
    for k = 0 to n - 1 do
        A[rev(k)] := a[k]
```

5. Exercises

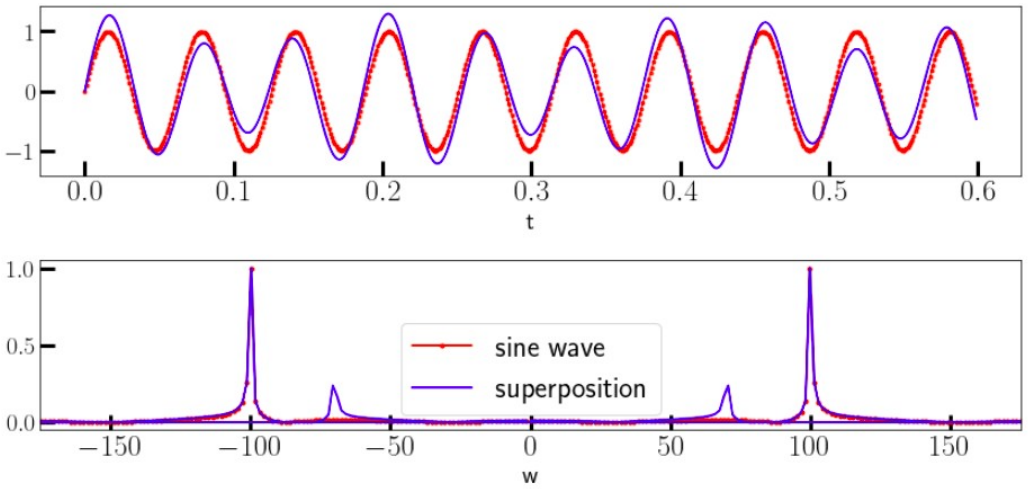
1a. FFT in Python:

$$\bar{f}_m = \sum_{n=0}^{N-1} f_n \, e^{-2\pi i m n / N}$$

- **Choose a sampling rate (say, 1000)**
- **Evaluate the corresponding sample interval**
- **In Python, build a 'time vector' of desired interval size in steps of the sample interval**
- **In Python, generate data of signal as a function of time where the signal is:**
  I. A sine wave of frequency 100
  II. A superposition of sine waves of different amplitudes (say, 1 and 0.3) and different frequencies (say 100 and 70 respectively)
- **For both sets of data, evaluate the FFT**
- **Pad the signal data to a size of the nearest power of 2 to the original size:**
  The 'numpy.fft.fft' routine has a kwarg of 'n=padded_size' for this
  To get the padded size, use 'padded_size = 2^nextpow2(original_size)' .
  The exponent of the nearest power of 2 to 'n' can be obtained as the ceiling of log(n) with base 2
- **Generate the frequencies where the FFTs were evaluated.**
  Use the 'numpy.fft.fftfreq' routine to do this automatically
- **Plot the absolute value of the FFTs as functions of the frequencies.**
  What can you infer from the FFT data?

5. Exercises

```
import numpy as np
import matplotlib.pyplot as plt
plt.rcParams['figure.figsize'] = (12,6)
plt.rcParams['font.size'] = 20

sample_rate = 1000

Delta = 1/sample_rate
sample_size = 4000
t = np.arange(sample_size) * Delta
padded_size = 2**np.ceil(np.log2(sample_size)).astype(int)

s1 = np.sin(100*t)
s2 = s1 + 0.3 * np.sin(70*t)

sp1 = np.fft.fft(s1, n=padded_size)
sp2 = np.fft.fft(s2, n=padded_size)

freq = np.fft.fftfreq(padded_size, d=t[1]-t[0])
f, (ax1, ax2) = plt.subplots(2,1)

ax1.plot(t[0:600], s1[0:600], "r.-", label='sine wave', alpha=0.6)
ax1.plot(t[0:600], s2[0:600], "b", label='superposition')
ax1.set_xlabel('t')

ax2.plot(2*np.pi*freq, np.abs(sp1)/np.amax(np.abs(sp1)), "r.-", label='sine wave')
ax2.plot(2*np.pi*freq, np.abs(sp2)/np.amax(np.abs(sp2)), "b", label='superposition')
ax2.legend()
ax2.set_xlabel('w')
ax2.set_xlim(-175,175)
plt.show()
```



5. Exercises

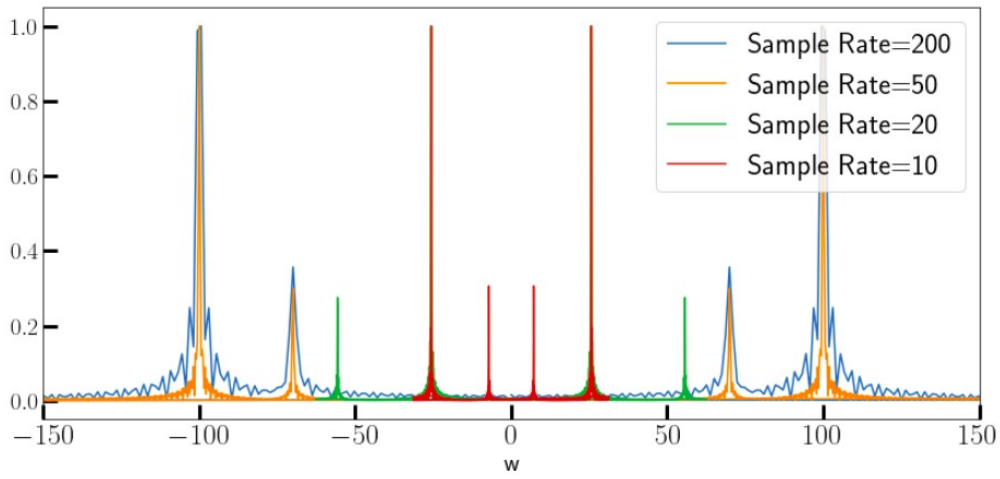
1b. FFT and Aliasing:

$$\bar{f}_m = \sum_{n=0}^{N-1} f_n \, e^{-2\pi i m n / N}$$

- **Take the previous signal and plot the FFT for a range of sample rates**
- **Can you see the effects of aliasing if the sample rate is too low? How low?**

```
import numpy as np
import matplotlib.pyplot as plt
plt.rcParams['figure.figsize'] = (12,6)
plt.rcParams['font.size'] = 20

sample_rates = [200, 50, 20, 10]
sample_size = 600

for s in sample_rates:
    Delta = 1/s
    t = np.arange(sample_size) * Delta
    padded_size = 2**np.ceil(np.log2(sample_size)).astype(int)
    signal = np.sin(100*t) + 0.3 * np.sin(70*t)
    ft = np.fft.fft(signal, n=padded_size)
    freq = np.fft.fftfreq(padded_size, d=t[1]-t[0])
    plt.plot(2*np.pi*freq, np.abs(ft)/np.amax(np.abs(ft)),\
            label=f'Sample Rate = {s}')

plt.xlabel('t')
plt.legend()
plt.xlabel('w')
plt.xlim(-150,150)
plt.show()
```

$$x(t) = \sin(100t) + 0.3\sin(70t)$$



5. Exercises

2. Using FFT to filter noisy signals:

- **Goto the GitHub repository for this course and download the following files:**

  i. ‘fftdata/times.npy’,
  ii. ‘fftdata/signal.npy’,
  iii. ‘fftdata/noisy_signal.npy’

- **Load these files into numpy arrays**

- **Plot the data from the signal files as a function of the data in the times file**

- **Now, do FFTs of the signal data and plot them as functions of frequency**
  For now, assume that the time data is regularly spaced, otherwise you’ll have to regularize it by interpolation

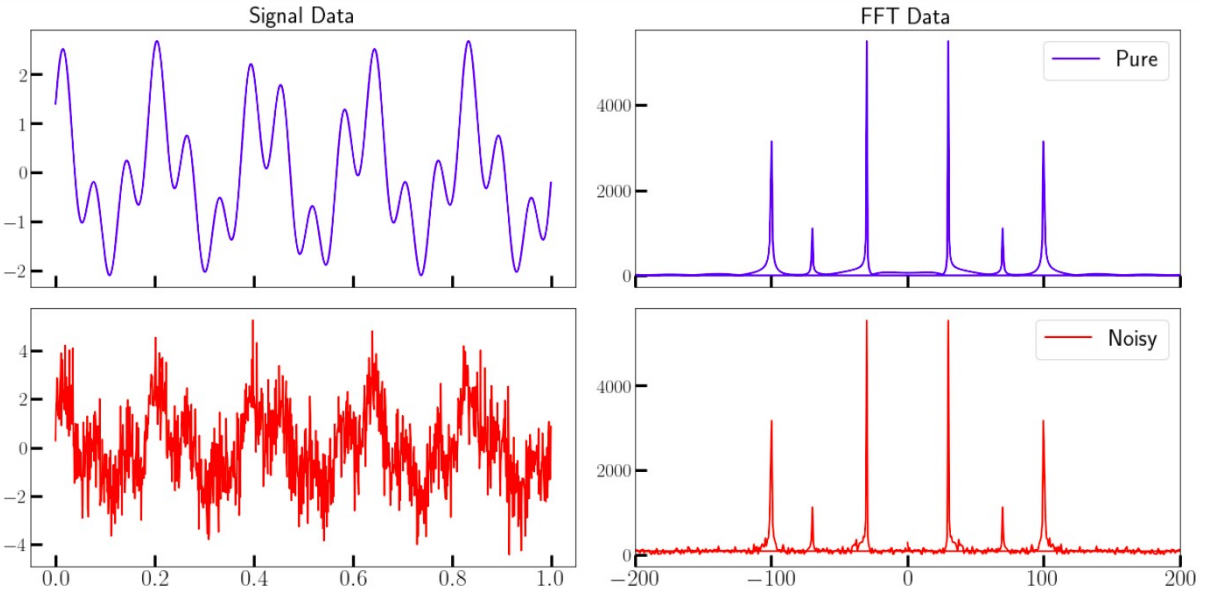
- **Is the FFT data what you expected?**

- **Clean up the FFT data of the noisy signal by removing the ‘noisy frequencies’ .**
  Visually estimate the noise threshold in the amplitudes and set all data below that to 0.

- **Finally, take an inverse FFT of the masked data and see if you succeeded in cleaning it up**

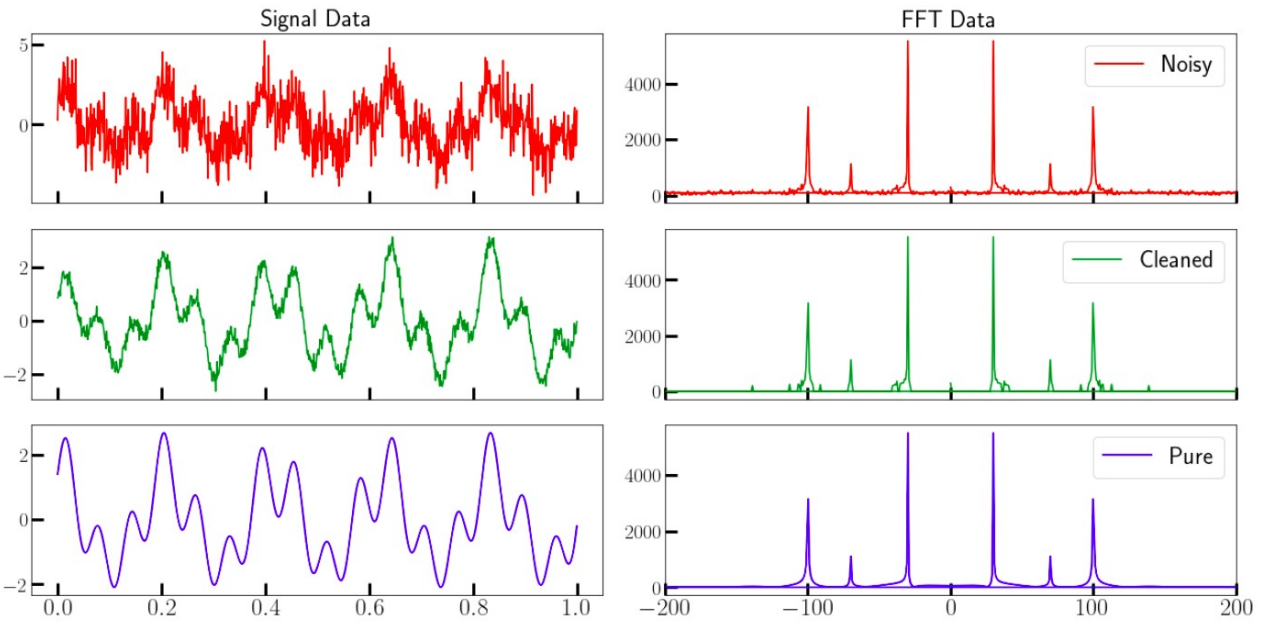
5. Exercises

```
import numpy as np
import matplotlib.pyplot as plt
plt.rcParams['figure.figsize'] = (18,9)
plt.rcParams['font.size'] = 20

# Load the datasets from files
times = np.load('fftdata/times.npy')
signal = np.load('fftdata/signal.npy')
noisy_signal = np.load('fftdata/noisy_signal.npy')

# Create a 2X2 grid of axes
f, axs = plt.subplots(2,2, sharex='col')

#Plot the raw datasets on the first column axes
axs[0,0].plot(times[0:1000], signal[0:1000], color="blue")
axs[0,0].set_title("Signal Data")
axs[1,0].plot(times[0:1000], noisy_signal[0:1000],color="red")

# Do the FFT of the datasets
sample_size = times.shape[-1]
padded_size = 2**np.ceil(np.log2(sample_size)).astype(int)
ft_sig = np.fft.fft(signal, n=padded_size)
ft_noisy = np.fft.fft(noisy_signal, n=padded_size)
freq = np.fft.fftfreq(padded_size, d=t[1]-t[0])

#Plot the fft datasets on the corresponding rows of the last column
axs[0,1].plot(2*np.pi*freq, np.abs(ft_sig), color='blue', label='Pure')
axs[0,1].set_title("FFT Data")
axs[1,1].plot(2*np.pi*freq, np.abs(ft_noisy), color='red', label='Noisy')
for ax in axs[:,1]:
    ax.set_xlim(-200,200)
    ax.legend()

plt.legend()
plt.show()
```



5. Exercises

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

threshold = 200
ft_abs      = np.abs(ft_noisy)
indices     = ft_abs > threshold   # filter out those value under 300
ft_clean    = indices * ft_noisy # noise frequency will be set to 0
cleaned_signal = np.fft.ifft(ft_clean) # Do inverse FFT

# Create a 2X2 grid of axes
f, axs = plt.subplots(3,2, sharex='col')

#Plot the raw datasets on the first column axes
axs[0,0].plot(times[0:1000], noisy_signal[0:1000], color='red')
axs[0,0].set_title("Signal Data")
axs[1,0].plot(times[0:1000], cleaned_signal[0:1000].real, color='green')
axs[2,0].plot(times[0:1000], signal[0:1000].real, color='blue')

#Plot the fft datasets on the corresponding rows of the last column
axs[0,1].plot(2*np.pi*freq, np.abs(ft_noisy), color='red', label='Noisy')
axs[0,1].set_title("FFT Data")
axs[1,1].plot(2*np.pi*freq, np.abs(ft_clean), color='green',
label='Cleaned')
axs[2,1].plot(2*np.pi*freq, np.abs(ft_sig), color='blue', label='Pure')
for ax in axs[:,1]:
    ax.set_xlim(-200,200)
    ax.legend()

plt.show()
```



Homework: Use a similar technique to separate the two frequencies in the cleaned data and plot them Independently.

5. Exercises

3. Compare FFT speed with DFT speed on random arrays of varying sizes.

- **Write (or find) a Python routine that:**

  i. Takes a numpy array **f** as its argument.
  ii. Evaluates the DFT vector **F** using the formula given on the right.
  iii. Evaluates the execution time.
  iv. Returns both of the above.

$$F = U \cdot f$$

$$U_{mn} = e^{-2\pi imn/N}$$

- **Generate random column vectors of sizes 16, 32, 64, 128, 256, 512, 1024, 2048, and 4096.**

- **Run the above subroutine and plot the execution time as a function of data size. Interpret the curve.**

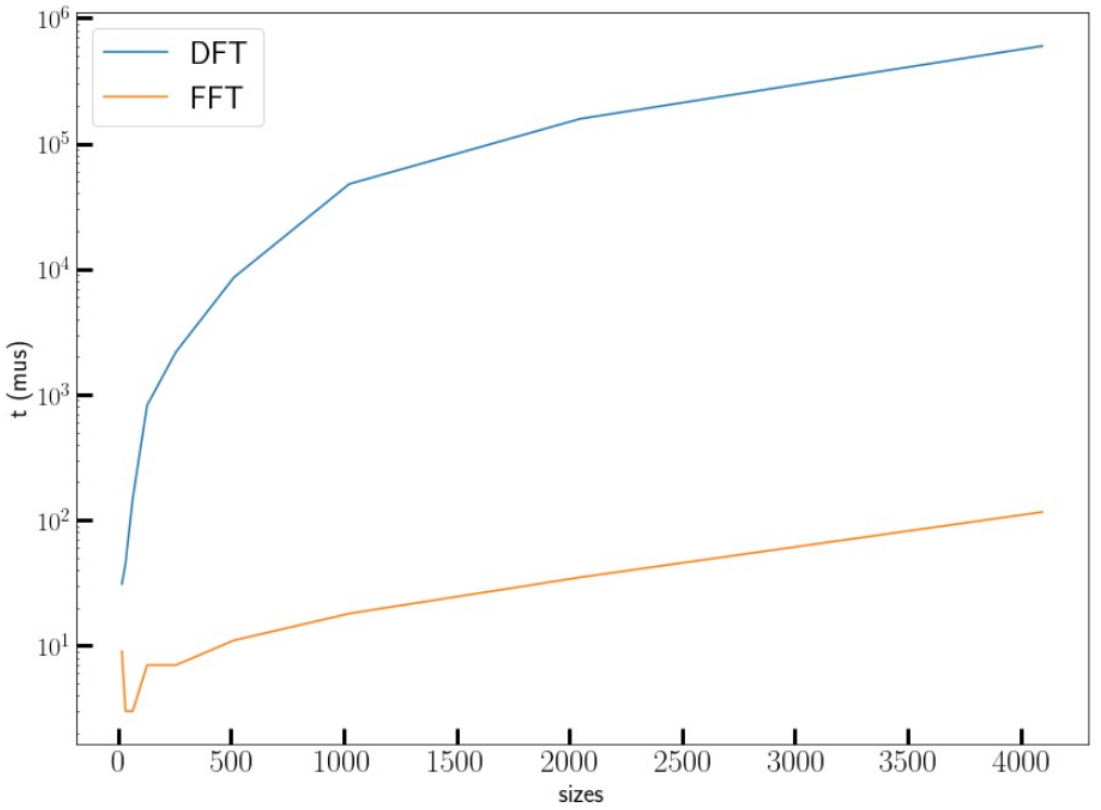
- **Repeat the steps above with the builtin FFT routine.**

- **In both cases, plot the execution time as a function of size and compare, then:**

  I. Plot the execution time for the DFT case as a function of size^2
  II. Plot the execution time for the FFT case as a function of size*log2(size)
  III. What can you infer from these graphs?

5. Exercises

```
import numpy as np
def dft(x):
    """
    Function to calculate the
    discrete Fourier Transform
    of a 1D real-valued signal x
    """
    N = x.shape[-1]
    n = np.arange(N)
    k = n.reshape((N, 1))
    e = np.exp(-2j * np.pi * k * n / N)
    return e @ x

x = np.random.random(100)
print(np.allclose(dft(x), np.fft.fft(x)))

import matplotlib.pyplot as plt
plt.rcParams['figure.figsize'] = (8,6)
plt.rcParams['font.size'] = 20

from timeit import Timer
defnum = 20
mysetup = 'import numpy as np\n'
mysetup += '''def dft(x):
                N = x.shape[-1]
                n = np.arange(N)
                k = n.reshape((N, 1))
                e = np.exp(-2j * np.pi * k * n / N)
                return e @ x'''

sizes = 2**np.arange(4,13)
times_dft = np.zeros_like(sizes)
times_fft = np.zeros_like(sizes)
for i,s in enumerate(sizes):
    dft = Timer(setup=mysetup, stmt="dft(np.random.random(%d))" % (s,))
    times_dft[i] = min(dft.repeat(number=defnum)) * 1e6 / defnum
    fft = Timer(setup=mysetup, stmt="np.fft.fft(np.random.random(%d))" % (s,))
    times_fft[i] = min(fft.repeat(number=defnum)) * 1e6 / defnum

plt.plot(sizes, times_dft,label="DFT")
plt.plot(sizes, times_fft, label="FFT")
plt.xlabel("sizes")
plt.ylabel("t (mus)")
plt.yscale('log')
plt.legend()
plt.show()
```

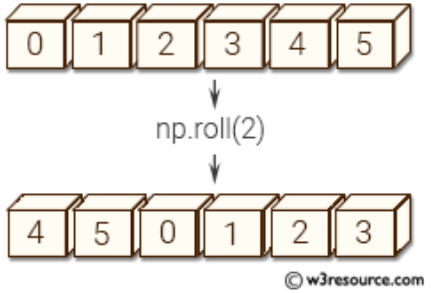


5. Exercises

4. Convolutions via FFT

- **Generate two vectors ‘f’ and ‘g’ of equal size (choose a large size, preferably a power of 2), consisting of random numbers, and evaluate the convolution vector $c = f \otimes g$, where the elements of c are given by**

$$c_l = \sum_j f_{l-j} \times g_j$$



  i. Imagine that the elements of ‘f’ lie on a ring. Then, the l^{th} component of ‘c involves the components of ‘f’ after ‘rolling’ this ring by l places.
  ii. In numpy, there is a function ‘numpy.roll’ that does this. Use that function.

- **Now, get the FFTs of the two vectors, compute their element-wise product, and do the IFFT of the result. Sum over this.**

- **Is this sum equal to the first result? Why?**

  i. Repeat for a few different sets of two random vectors.
  ii. Recall the convolution theorem

- **If they are indeed equal as expected, then which method is faster and why?**

  Use the %timeit magic in Jupyter to measure runtimes

5. Exercises

4. Convolutions via FFT

```
import numpy as np

N = 2**15
f = np.random.random(N)
g = np.random.random(N)

print("Normal Convolution:")
%timeit c = np.array([np.sum(np.roll(f,-l-1)[::-1]*g) for l in range(N)])
c = np.array([np.sum(np.roll(f,-l-1)[::-1]*g) for l in range(N)])

print("\nConvolution by FFT:")
%timeit c_fft = np.fft.ifft(np.fft.fft(f) * np.fft.fft(g))
c_fft = np.fft.ifft(np.fft.fft(f) * np.fft.fft(g))

print("\nAre they equal?", np.allclose(c, c_fft))
```

Normal Convolution:
1.91 s ± 10.7 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)

Convolution by FFT:
1.4 ms ± 23.7 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)

Are they equal? True

QuTiP: The Quantum Toolbox in Python

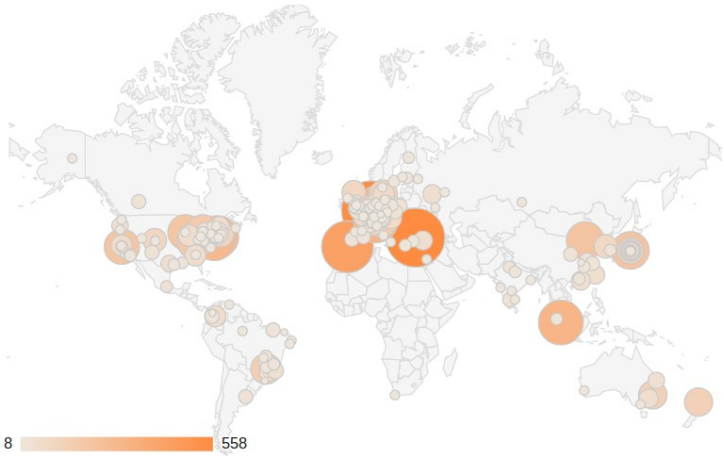
QuTiP: The Quantum Toolbox in Python

- Framework for computational quantum dynamics
  - Efficient and easy to use for quantum physicists
  - Thoroughly tested (100+ unit tests)
  - Well documented (200+ pages, 50+ examples)
  - Quite large number of users (>1000 downloads)
- Suitable for
  - theoretical modeling and simulations
  - modeling experiments
- 100% open source
- Implemented in Python/Cython using SciPy, Numpy, and matplotlib

QuTiP: The Quantum Toolbox in Python

Project information

- Authors: Paul Nation and Robert Johansson
- Web site: http://qutip.org
- License: GPLv3
- Repository: http://github.com/qutip
- Publication: Comp. Phys. Comm. **183**, 1760 (2012)
  arXiv:1211.6518 (2012)



QuTiP: The Quantum Toolbox in Python

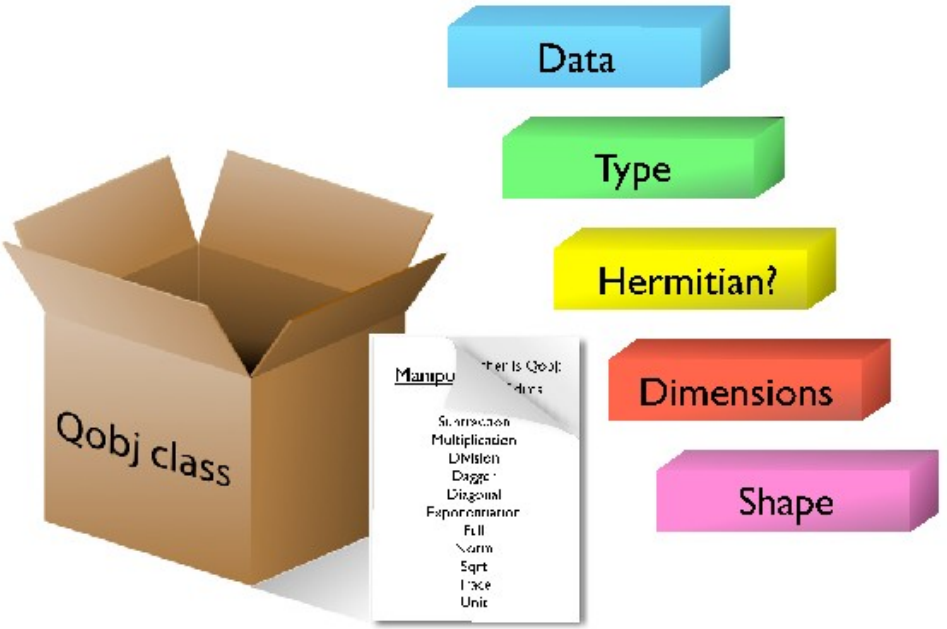
- Objectives

  To provide a powerful framework for quantum mechanics that closely resembles the standard mathematical formulation

  - Efficient and easy to use
  - General framework, able to handle a wide range of different problems

- Design and implementation
  - Object-oriented design
  - Qobj class used to represent quantum objects
    - Operators
    - State vectors
    - Density matrices
  - Library of utility functions that operate on Qobj instances

QuTiP core class:
Qobj

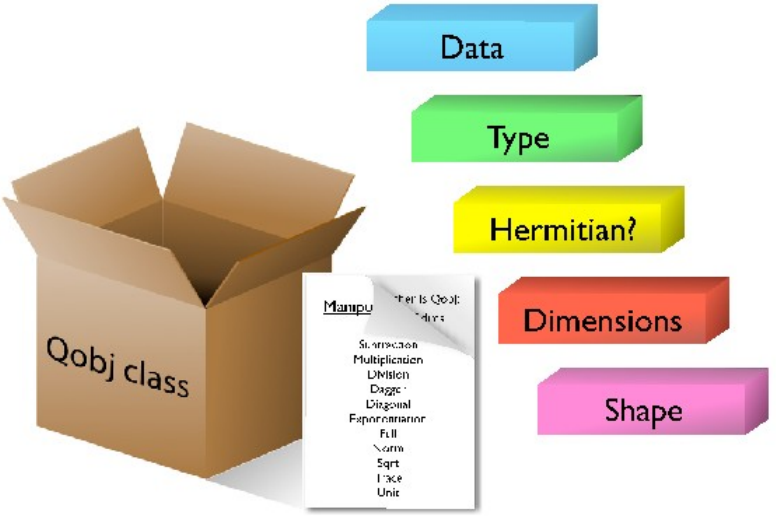


QuTiP: The Quantum Toolbox in Python

Quantum object class: `Qobj`

- Abstract representation of quantum states and operators
  - Matrix representation of the object
  - Structure of the underlaying state space, Hermiticity, type, etc.
  - Methods for performing all common operations on quantum objects:

    `eigs(),dag(),norm(),unit(),expm(),sqrt(),tr(), ...`
  - Operator arithmetic with implementations of: +. -, *, ...



Example: built-in operator $\hat{\sigma}_x$

```
>>> sigmax()

Quantum object: dims = [[2], [2]], shape = [2, 2],
type = oper, isHerm = True
Qobj data =
[[ 0.  1.]
 [ 1.  0.]]
```

Example: built-in state $|\alpha = 0.5\rangle$

```
>>> coherent(5, 0.5)

Quantum object: dims = [[5], [1]], shape = [5, 1], type = ket
Qobj data =
[[ 0.88249693]
 [ 0.44124785]
 [ 0.15601245]
 [ 0.04496584]
 [ 0.01173405]]
```

QuTiP: The Quantum Toolbox in Python

Evolution of quantum systems

The main use of QuTiP is quantum evolution. A number of solvers are available.

- Typical simulation workflow:
  i. Define parameters that characterize the system
  ii. Create Qobj instances for operators and states
  iii. Create Hamiltonian, initial state and collapse operators, if any
  iv. Choose a solver and evolve the system
  v. Post-process, visualize the data, etc.

- Available evolution solvers:
  - Unitary evolution: Schrödinger and von Neumann equations
  - Lindblad master equations
  - Monte-Carlo quantum trajectory method
  - Bloch-Redfield master equation
  - Floquet-Markov master equation
  - Propagators

Lindblad master equation

Equation of motion for the density matrix $\rho(t)$ for a quantum system that interacts with its environment:

$$\dot{\rho}(t) = -\frac{i}{\hbar}[H(t), \rho(t)] + \sum_n \frac{1}{2}\left[2c_n\rho(t)c_n^\dagger - \rho(t)c_n^\dagger c_n - c_n^\dagger c_n \rho(t)\right]$$

$H(t) =$ system Hamiltonian

$c_n = \sqrt{\gamma_n}a_n$ describes the effect of the environment on the system

$\gamma_n =$ rate of the environment-system interaction process

How do we solve this equation numerically?

I. Construct the matrix representation of all operators
II. Evolve the ODEs for the unknown elements in the density matrix
III. For example, calculate expectation values for some selected operators for each $\rho(t)$

QuTiP: The Quantum Toolbox in Python

Lindblad master equation

Equation of motion for the density matrix $\rho(t)$ for a quantum system that interacts with its environment:

$$\dot{\rho}(t) = -\frac{i}{\hbar}[H(t), \rho(t)] + \sum_n \frac{1}{2}\left[2c_n\rho(t)c_n^\dagger - \rho(t)c_n^\dagger c_n - c_n^\dagger c_n \rho(t)\right]$$

$$H(t) = \text{system Hamiltonian}$$

$$c_n = \sqrt{\gamma_n}a_n \text{ describes the effect of the environment on the system}$$

$$\gamma_n = \text{rate of the environment-system interaction process}$$

How do we solve this equation numerically in QuTiP?

```
from qutip import *

psi0 = ...              # initial state
H  = ...                # system Hamiltonian
c_op_list = [...]       # collapse operators
e_op_list = [...]       # expectation value operators

tlist = linspace(0, 10, 100)
result = mesolve(H, psi0, tlist, c_op_list, e_op_list)
```

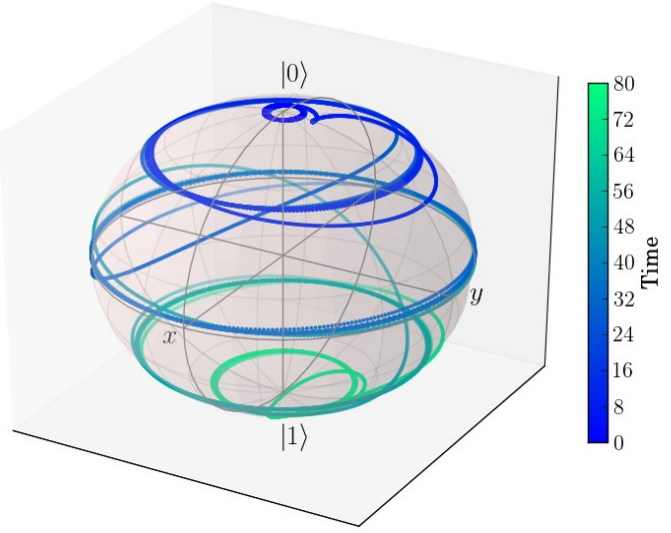
QuTiP: The Quantum Toolbox in Python

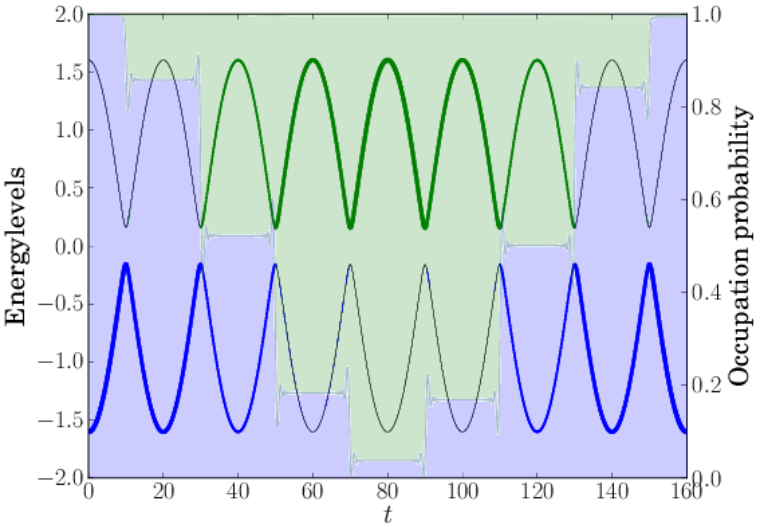
Example: time-dependence

Multiple Landau-Zener transitions

$$\hat{H}(t) = -\frac{\Delta}{2}\hat{\sigma}_z - \frac{\epsilon}{2}\hat{\sigma}_x - A\cos(\omega t)\hat{\sigma}_z$$

```
from qutip import *

# Parameters
epsilon = 0.0
delta = 1.0

# Initial state: start in ground state
psi0 = basis(2,0)

# Hamiltonian
H0 = - delta * sigmaz() - epsilon * sigmax()
H1 = - sigmaz()
h_t = [H0, [H1, 'A * cos(w*t)']]
args = {'A': 10.017, 'w': 0.025*2*pi}

# No dissipation
c_ops = []

# Expectation values
e_ops = [sigmax(), sigmay(), sigmaz()]

# Evolve the system
tlist = linspace(0, 160, 500)
output = mesolve(h_t, psi0, tlist, c_ops, e_ops, args)

# Process and plot result
# ...
```





QuTiP: The Quantum Toolbox in Python

QuTiP: Examples

```
from qutip import *

def evals_multw(w1list, w2, w3, g12, g13):

    # Pre-compute operators for the hamiltonian
    sz1 = tensor(sigmaz(), qeye(2), qeye(2))
    sx1 = tensor(sigmax(), qeye(2), qeye(2))

    sz2 = tensor(qeye(2), sigmaz(), qeye(2))
    sx2 = tensor(qeye(2), sigmax(), qeye(2))

    sz3 = tensor(qeye(2), qeye(2), sigmaz())
    sx3 = tensor(qeye(2), qeye(2), sigmax())

    evals_mat = np.zeros((len(w1list),2**3))
    for idx, w1 in enumerate(w1list):
        # evaluate the Hamiltonian
        H = w1 * sz1 + w2 * sz2 + w3 * sz3 +\
            g12 * sx1 * sx2 + g13 * sx1 * sx3
        # find the energy eigenvalues of the composite system
        evals = H.eigenenergies()
        evals_mat[idx,:] = np.real(evals)

    return evals_mat

w1  = 1.0 * 2 * np.pi   # atom 1 frequency: sweep this one
w2  = 0.9 * 2 * np.pi   # atom 2 frequency
w3  = 1.1 * 2 * np.pi   # atom 3 frequency
g12 = 0.05 * 2 * np.pi   # atom1-atom2 coupling strength
g13 = 0.05 * 2 * np.pi   # atom1-atom3 coupling strength

w1list = np.linspace(0.75, 1.25, 50) * 2 * np.pi # atom 1 frequency
range
evals_mat = evals_multw(w1list, w2, w3, g12, g13)
```

```
import matplotlib.pyplot as plt
plt.rcParams['font.size'] = 20
fig, ax = plt.subplots(figsize=(12,6))

for n in [1,2,3]:
    ax.plot(w1list / (2*np.pi), (evals_mat[:,n]-evals_mat[:,0]) / (2*np.pi), 'b')

ax.set_xlabel('Energy splitting of electron 1')
ax.set_ylabel('Eigenenergies')
ax.set_title('Energy spectrum of three coupled electrons')
plt.show()
```

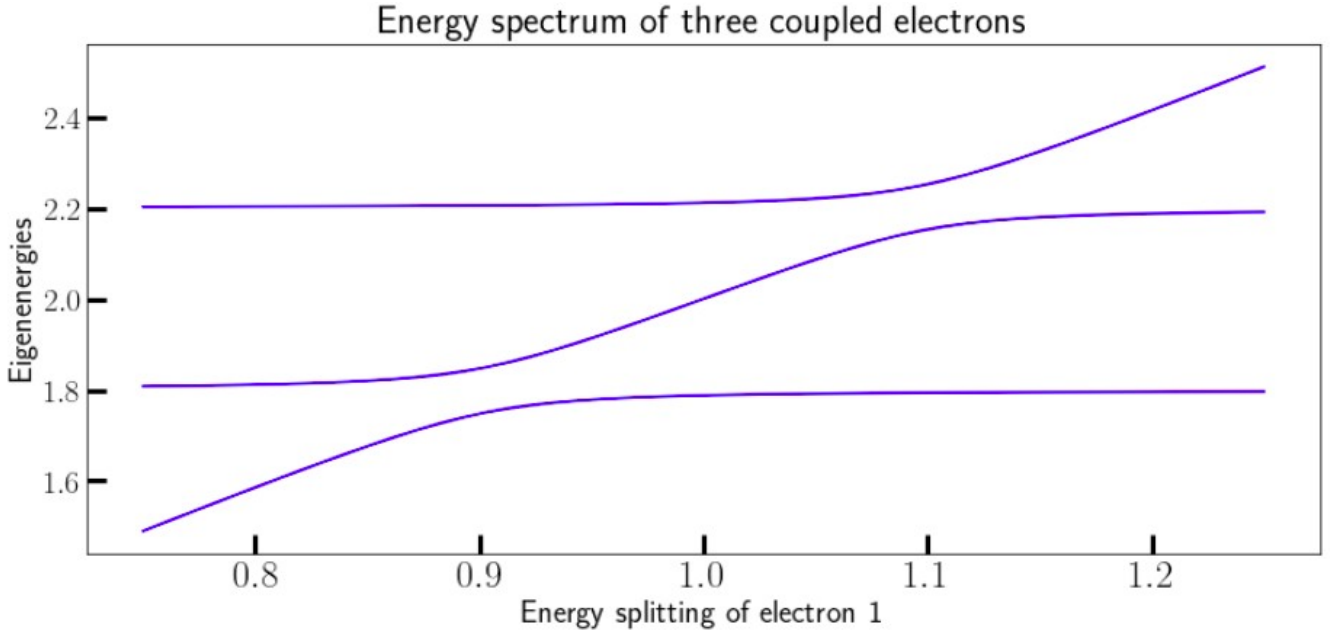
See Jupyter Notebook:
‘QUTIP.ipynb’

$$H = \sum_{i=0}^{N-1} \omega_i \sigma_i^z + g \sum_{i=0}^{N-1} \sigma_i^x \sigma_{i+1}^x$$

QuTiP: The Quantum Toolbox in Python

QuTiP: Examples

$$H = \sum_{i=0}^{N-1} \omega_i \sigma_i^z + g \sum_{i=0}^{N-1} \sigma_i^x \sigma_{i+1}^x$$



See Jupyter Notebook:
'Special_Topics.ipynb'

QuTiP: The Quantum Toolbox in Python

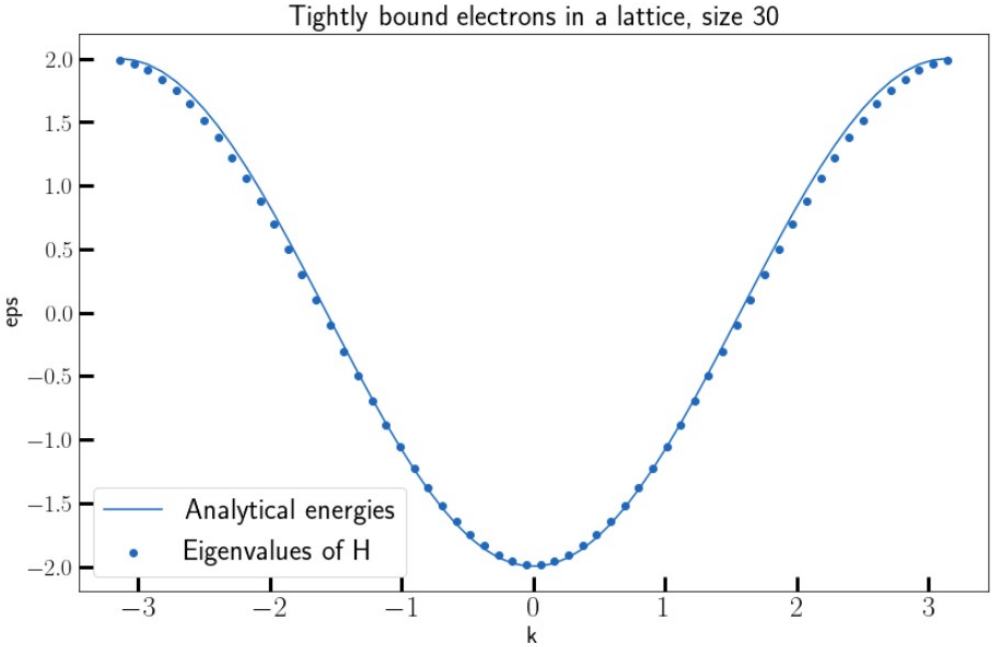
QuTiP: Examples

```
#N = 100
N = 30

H = sum([basis(N,i) * basis(N, i + 1).dag() for i in range(N - 1)]) +\
    sum([basis(N,i) * basis(N, i - 1).dag() for i in range(1, N)])

evals = H.eigenenergies()
# satisfy periodic boundary conditions we need E(-k) = E(k)
numerical = np.concatenate((np.flip(evals, 0), evals), axis=0)


import numpy as np
import matplotlib.pyplot as plt

plt.rcParams['figure.figsize'] = (12,8)
plt.rcParams['font.size'] = 20
k = np.linspace(-np.pi, np.pi, 2*N)

#plt.scatter(k[::3],numerical[::3].real, label='Eigenvalues of H')
plt.scatter(k,numerical.real, label='Eigenvalues of H')

analytical = -2 * np.cos(k)
plt.title(f"Tightly bound electrons in a lattice, size {N}")
plt.plot(k,analytical, label='Analytical energies')
plt.xlabel("k")
plt.ylabel("eps")
plt.legend()
plt.show()
```

See Jupyter Notebook:
'Special_Topics.ipynb'

$$H = \frac{1}{2}\sum_{i=0}^{N-1}\left(c_i^\dagger c_{i+1} + c_{i+1}^\dagger c_i\right)$$

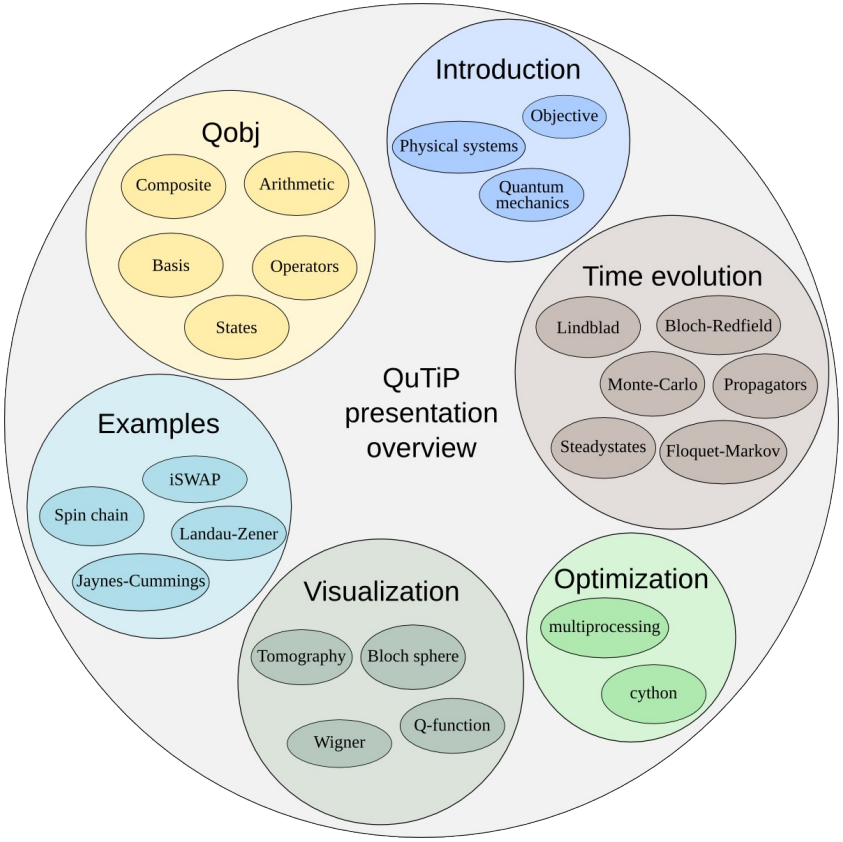
$$c_i^\dagger|0\rangle = |i\rangle \qquad c|i\rangle = |0\rangle \qquad c_i^\dagger|i\rangle = c|0\rangle = 0.$$



QuTiP: The Quantum Toolbox in Python

More information at: http://qutip.org

- QuTiP: framework for numerical simulations of quantum systems
  - Generic framework for representing quantum states and operators
  - Large number of dynamics solvers
- Main strengths:
  - Ease of use: complex quantum systems can programmed rapidly and intuitively
  - Flexibility: Can be used to solve a wide variety of problems
  - Performance: Near C-code performance due to use of Cython for time-critical functions



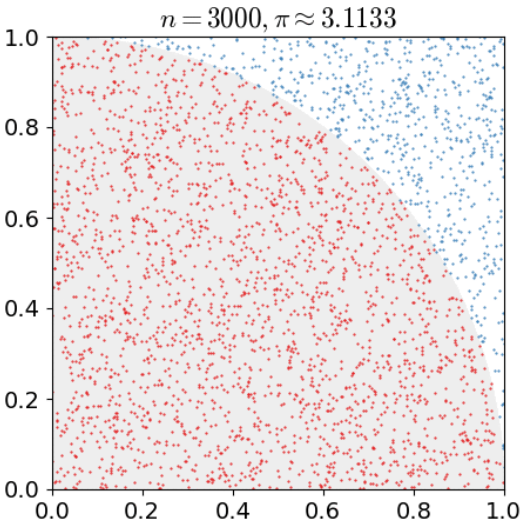
Monte Carlo Methods in Python

Monte Carlo Methods in Python

Many computer calculations employ Monte Carlo methods that include elements of chance to either:

1) Simulate random physical processes, such as thermal motion or radioactive decay, or
2) Estimate areas, volumes etc. by numerical integration

Monte Carlo Casino, Monaco, Cote-d-Azur, France



Monte Carlo Methods in Python

Randomness in deterministic computers???

Besides, how do even know if anything can be ‘random’??

Monte Carlo Methods in Python

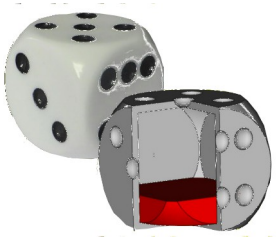
What is ‘randomness’, or a ‘random variable’?

Basically, it’s a special way of choosing the elements of a set. In our case, it involves choosing a single number from a particular subset of complex numbers , based on a probability of choice.

- Uniform randomness: When all numbers in the subset are equally likely to be chosen
- Non-uniform Randomness: When the probability of a particular choice depends on the value of the choice.

P(r) dr – Probability of a choice being in [r, r+dr]

Uniform: P(r) constant



Non- Uniform: P(r) depends on r

Monte Carlo Methods in Python

What is ‘randomness’, or a ‘random variable’?

Basically, it’s a special way of choosing the elements of a set. In our case, it involves choosing a single number from a particular subset of complex numbers, based on a probability of choice.

- Mathematically, “independent randomness” occurs when there are no correlations between any two sets of choices from the same subset

$$\langle r \rangle \equiv \int_D r\ P(r)\ \mathrm{d}r\ , \text{ with } \int_D P(r)\ \mathrm{d}r = 1$$

Independently Random:

$$\langle r_1\ r_2 \rangle = \langle r_1 \rangle\ \langle r_2 \rangle \text{ or } P(r_1|r_2) = P(r_1)P(r_2)$$

$$C(r_1, r_2) \equiv \langle r_1\ r_2 \rangle - \langle r_1 \rangle\ \langle r_2 \rangle = 0$$

Monte Carlo Methods in Python

What is ‘randomness’, or a ‘random variable’?

Basically, it’s a special way of choosing the elements of a set. In our case, it involves choosing a single number from a particular subset of complex numbers, based on a probability of choice.

- Mathematically, “independent randomness” occurs when there are no correlations between any two sets of choices from the same subset
- Obeys the ‘**Central Limit Theorem’**

Consider an arbitrarily large (infinite) array of independent random variables, called ‘A’.

By the law of large numbers, their average is a fixed constant μ and stdev σ

For a finite (but reasonably large) array of n independent random variables from A , the probability that their sample average (times $\sqrt{n}$) is a value ‘x’ (times $\sqrt{n}$) is

$$P(x) = \frac{1}{\sigma\sqrt{\pi}}\, e^{(x-\mu)^2/2\sigma^2}$$

Monte Carlo Methods in Python

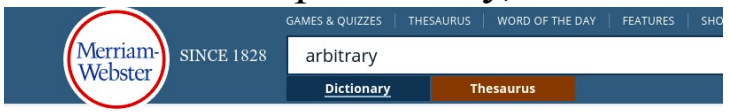
What is ‘randomness’, or a ‘random variable’?

Basically, it’s a special way of choosing the elements of a set. In our case, it involves choosing a single number from a particular subset of complex numbers, based on a probability of choice.

- **<u>Note: In science, ‘Random’ and ‘Arbitrary’ are two different things!</u>**

Although they are used interchangeably in common parlance

Arbitrary choices have no well-defined probability, whereas random choices do



arbitrary *adjective*

Save Word

ar·bi·trary | \ ˈär-bə-ˌtrer-ē, -ˌtre-rē \

Definition of *arbitrary*

1 a : existing or coming about seemingly at random or by chance or as a capricious and unreasonable act of will

// an *arbitrary* choice

// When a task is not seen in a meaningful context it is experienced as being *arbitrary*.

— Nehemiah Jordan

Monte Carlo Methods in Python

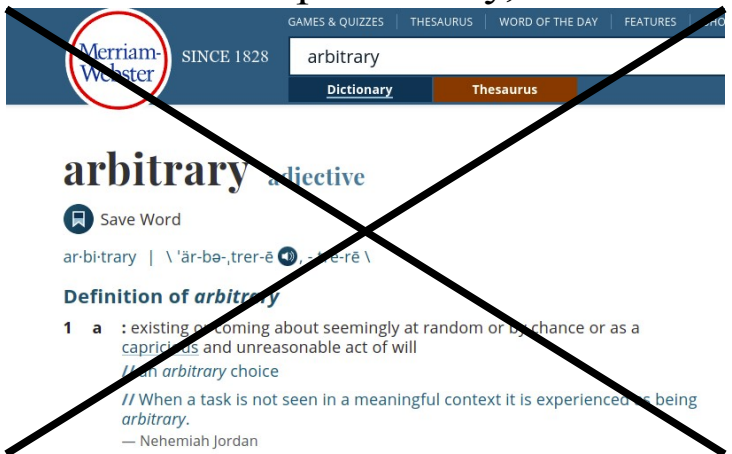
What is ‘randomness’, or a ‘random variable’?

Basically, it’s a special way of choosing the elements of a set. In our case, it involves choosing a single number from a particular subset of complex numbers, based on a probability of choice.

- **<u>Note: In science, ‘Random’ and ‘Arbitrary’ are two different things!</u>**

  Although they are used interchangeably in common parlance

  Arbitrary choices have no well-defined probability, whereas random choices do



Monte Carlo Methods in Python

What is ‘randomness’, or a ‘random variable’?

Basically, it’s a special way of choosing the elements of a set. In our case, it involves choosing a single number from a particular subset of complex numbers, based on a probability of choice.

- **Note: In science, ‘Random’ and ‘Arbitrary’ are two different things!**

  Although they are used interchangeably in common parlance

  Arbitrary choices have no well-defined probability, whereas random choices do

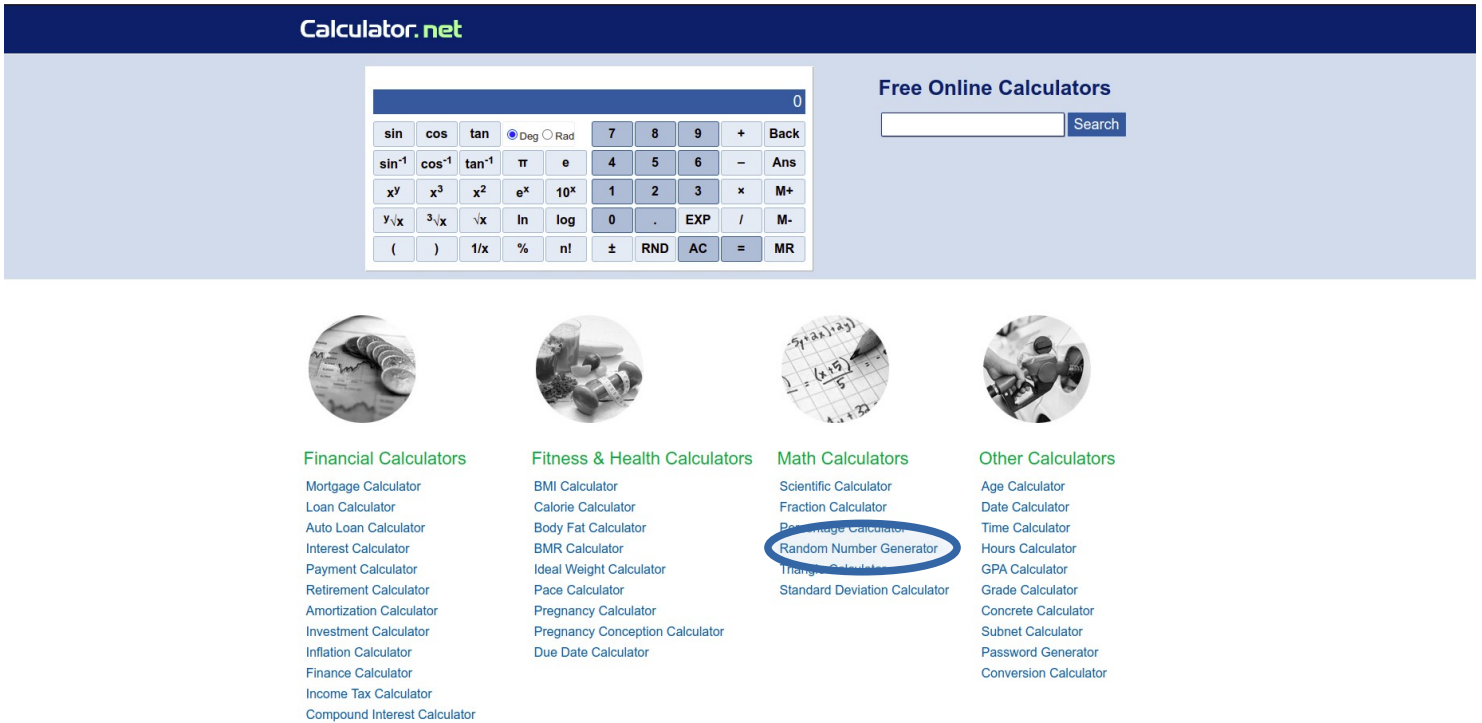
- However, in some cases, depending on context, we ignore this distinction

  In Python: “Prove” that Convolution Theorem works by creating 2 random arrays and using FFT to test the theorem

Monte Carlo Methods in Python

What is ‘randomness’, or a ‘random variable’?

Basically, it’s a special way of choosing the elements of a set. In our case, it involves choosing a single number from a particular subset of complex numbers, based on a probability of choice.

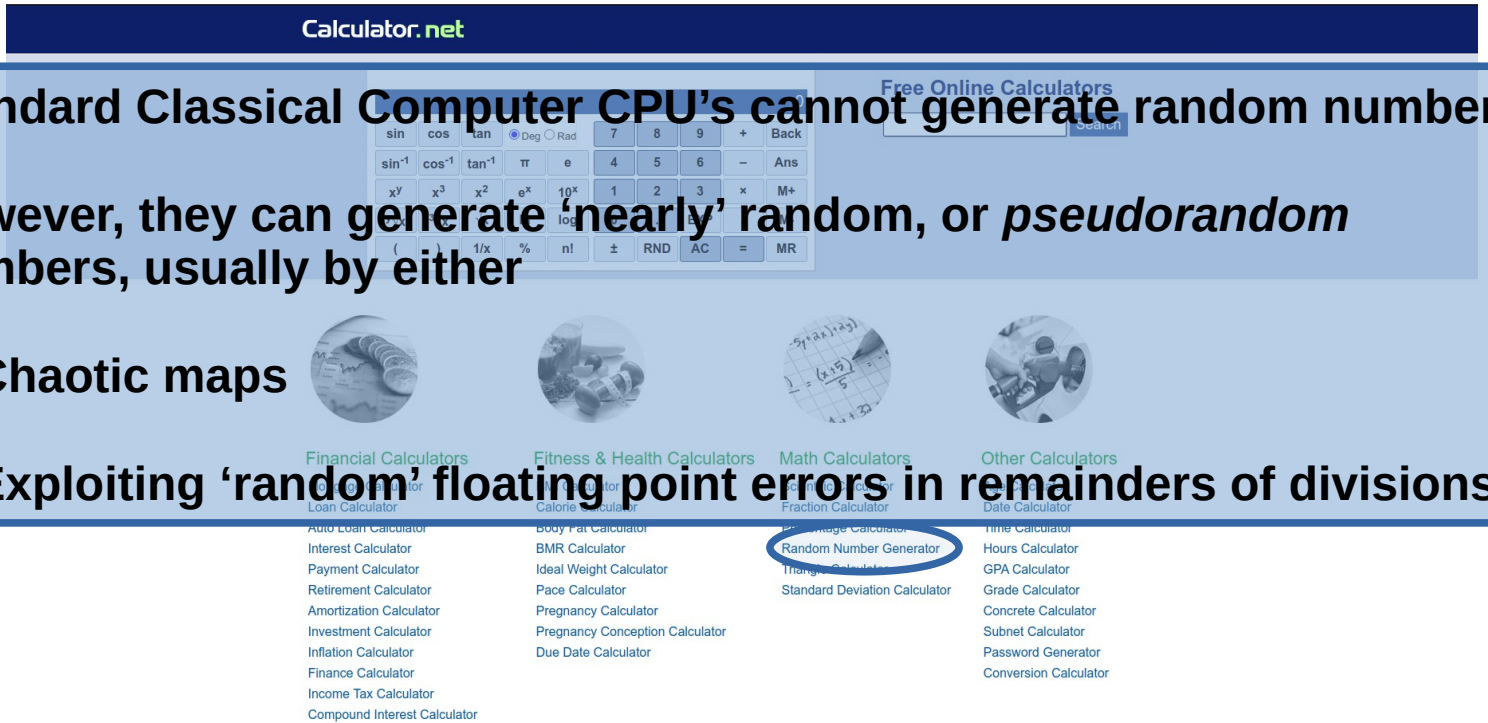


Monte Carlo Methods in Python

Monte Carlo Methods in Python

What is ‘randomness’, or a ‘random variable’?

Basically, it’s a special way of choosing the elements of a set. In our case, it involves choosing a single number from a particular subset of complex numbers, based on a probability of choice.



Standard Classical Computer CPU’s cannot generate random numbers

However, they can generate ‘nearly’ random, or *pseudorandom* numbers, usually by either

1) Chaotic maps

2) Exploiting ‘random’ floating point errors in remainders of divisions

Monte Carlo Methods in Python

Random numbers in NumPy: Uniform beteween 0 and 1 by default

```
Python 3.8.8 (default, Apr 13 2021, 19:58:26)
[GCC 7.3.0] :: Anaconda, Inc. on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import numpy as np
>>> print(np.random.random())
0.6226775696151192
>>> print(np.random.random())
0.792263425922030 5
>>> print(np.random.random())
0.14630228848931848
>>> a = np.random.random((3,2)); print(a)
[[0.69190953 0.36821051]
 [0.36047735 0.17258306]
 [0.00406954 0.26549197]]
```

Monte Carlo Methods in Python

Random numbers in NumPy: Uniform beteween 0 and 1 by default

How to make the probability of a number r be biased by P(r)?

```
import numpy as np

def weighted_choice(objects, weights):
    """ returns randomly an element from the sequence of 'objects',
        the likelihood of the objects is weighted according
        to the sequence of 'weights', i.e. percentages."""

    weights = np.array(weights, dtype=np.float64)
    # Normalize the weights:
    weights /= weights.sum()
    # “Integrate” the probabilities to find the cumulative probability
    weights = weights.cumsum()
    r = np.random.random()
    for i in range(len(weights)):
        if r < weights[i]:
            return objects[i]
```

$$\lambda(x_i) = \int_0^{x_i} P(r)\ \mathrm{d}r$$

Cumulative “Weights”

$$x_i = \{x_0, x_1, x_2, \ldots x_n\}$$

Objects to choose from

Choose r uniform-randomly

If $r < \lambda(x_i)$, then

return x_i

Monte Carlo Methods in Python

Random numbers in NumPy: Uniform beteween 0 and 1 by default

How to make the probability of a number r be biased by P(r)?

```
import numpy as np

def weighted_choice(objects, weights):
    """ returns randomly an element from the sequence of 'objects',
        the likelihood of the objects is weighted according
        to the sequence of 'weights', i.e. percentages."""

    weights = np.array(weights, dtype=np.float64)
    # Normalize the weights:
    weights /= weights.sum()
    # “Integrate” the probabilities to find the cumulative probability
    weights = weights.cumsum()
    r = np.random.random()
    for i in range(len(weights)):
        if r < weights[i]:
            return objects[i]
```

Note that:

$$\lambda(x_i) = \int_0^{x_i} P(r)\,\mathrm{d}r$$

$$\Longrightarrow \quad P(r) = \frac{d}{dr}\lambda(r)$$

Monte Carlo Methods in Python

Random numbers in NumPy: Uniform beteween 0 and 1 by default

How to make the probability of a number r be biased by P(r)?

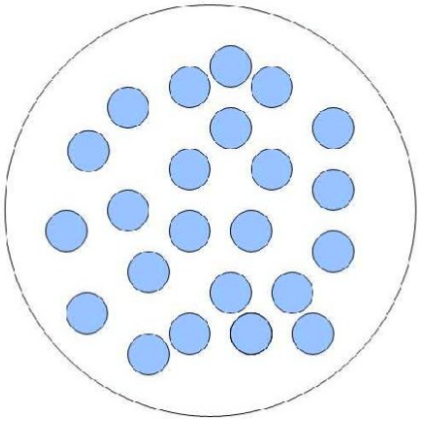
$$\lambda(x_i) = \int_0^{x_i} P(r)\,\mathrm{d}r$$

$$\Longrightarrow\ P(r) = \frac{d}{dr}\lambda(r)$$

Example problems: See Jupyter Notebook MC.ipynb

- **Radioactive Decay (exact)**
- **Random Walks**
- **Metropolis Algorithm and the Ising Model (1d and 2d)**

Monte Carlo Methods in Python



Example problems: See Jupyter Notebook MC.ipynb

Radioactive Decay:

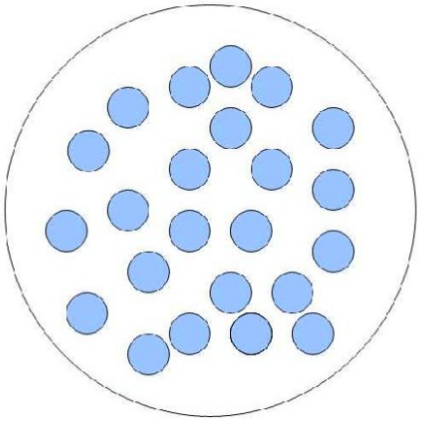
- **A sample of N radioactive atoms undergoing spontaneous decay**
- **The probability P of decay within a given time interval is proportional to the interval**

$$N(t) \xrightarrow{\Delta t} N - |\Delta N(t)|$$

$$P = -\lambda \Delta t = \frac{\Delta N}{N}$$

$$\Delta N \to dN \implies \frac{dN}{dt} = -\lambda N \implies N = N_0 e^{-\lambda t}$$

Monte Carlo Methods in Python



Example problems: See Jupyter Notebook MC.ipynb

Radioactive Decay:

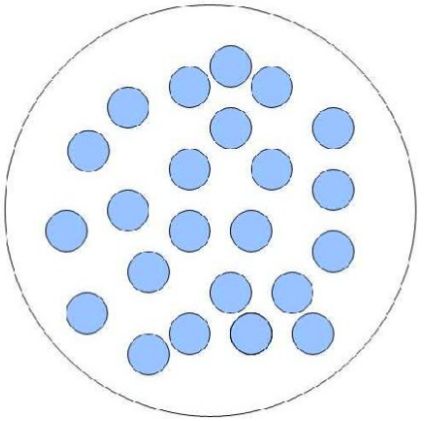
- **A sample of N radioactive atoms undergoing spontaneous decay**
- **The probability P of decay within a given time interval is proportional to the interval**

$$N(t) \xrightarrow{\Delta t} N - |\Delta N(t)|$$

$$P = -\lambda \Delta t = \frac{\Delta N}{N}$$

$$\Delta N \to dN \xrightarrow[\text{Continuum Approximation}]{\frac{dN}{dt} = -\lambda N} N = N_0 e^{-\lambda t}$$

Monte Carlo Methods in Python



Example problems: See Jupyter Notebook MC.ipynb

Radioactive Decay:

- **A sample of N radioactive atoms undergoing spontaneous decay**
- **The probability P of decay within a given time interval is proportional to the interval**

$$N(t) \xrightarrow{\Delta t} N + \Delta N(t)$$

$$P = -\lambda \Delta t = \frac{\Delta N}{N}$$

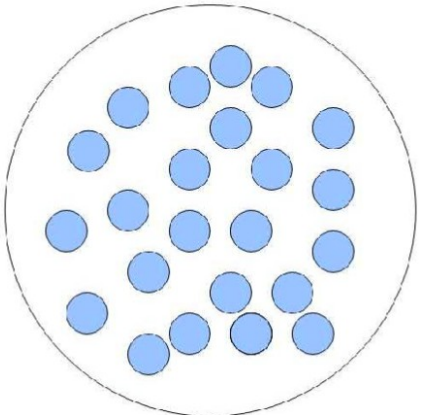
- **Exact Simulation: At each time, for each atom, choose a random number 'r' and decrement N only if**

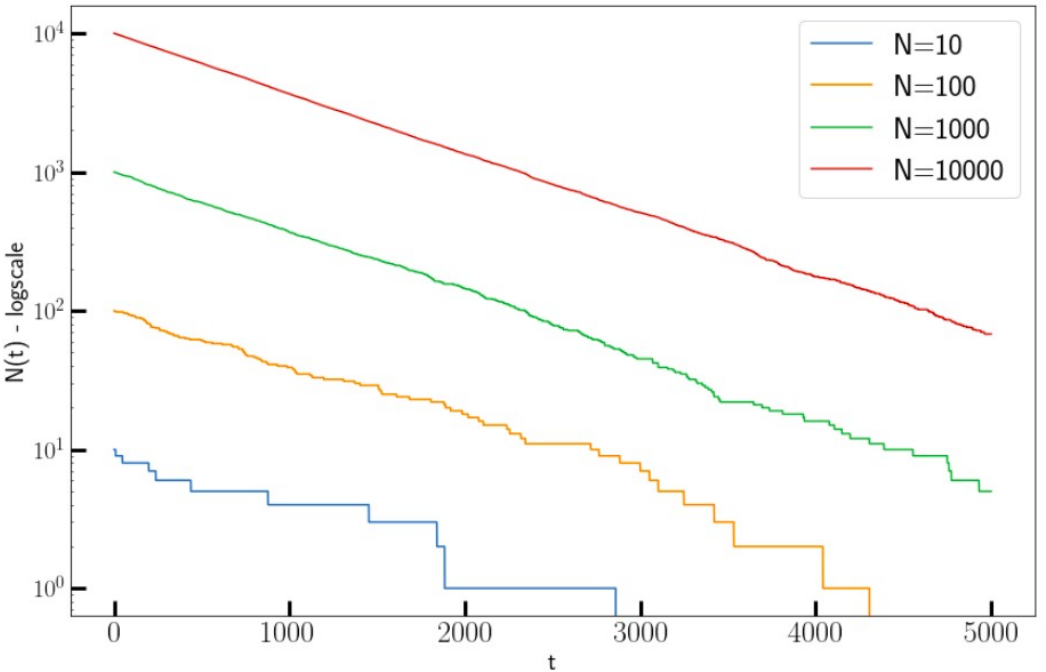
$$r < \lambda$$

Monte Carlo Methods in Python

Monte Carlo Methods in Python

```
import numpy as np
import matplotlib.pyplot as plt
plt.rcParams['figure.figsize'] = (12,8)
plt.rcParams['font.size'] = 20

# Decay constant
lambda1 = 1e-3

natoms_arr = [10, 100, 1000, 10000]
maxtime = 5000

timeline = np.arange(0, maxtime  + 1)
for natoms in natoms_arr:
    natoms_begin = natoms
    natoms_time = []
    for time in timeline:
        decays = np.random.random(natoms)
        natoms -= np.count_nonzero(decays < lambda1)
        natoms_time.append(natoms)

    plt.plot(timeline,
natoms_time,label=f'N={natoms_begin}')

plt.ylabel("N(t) - logscale")
plt.xlabel("t")
plt.yscale("log")
plt.legend()
plt.show()
```



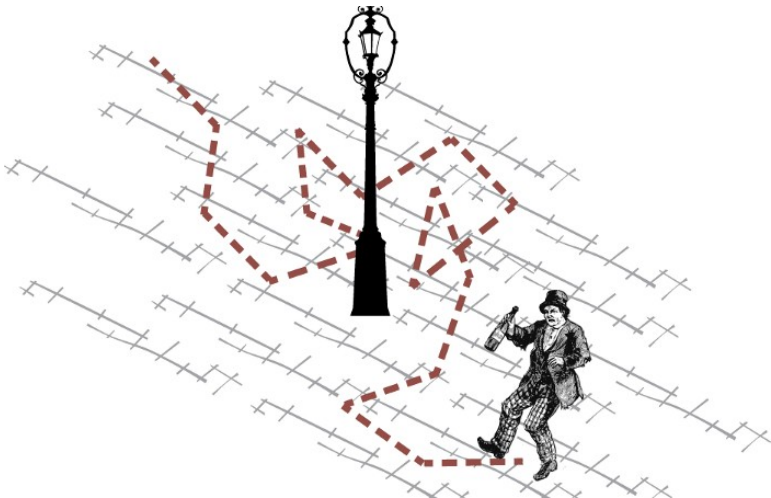


Monte Carlo Methods in Python

Example problems: See Jupyter Notebook MC.ipynb

Random Walk Problem:

- **Monte Carlo-like model of Diffusion, Brownian motion etc.**
- **A digital simulation of a 'drunk man', where a point advances along a graph in discrete steps**
- **Each step can be of a random stepsize in a random direction, or fixed stepsize in a random direction.**



$$R^2 = (\Delta x_1 + \Delta x_2 + \cdots + \Delta x_N)^2 + (\Delta y_1 + \Delta y_2 + \cdots + \Delta y_N)^2$$

Average over a large number of trials:

$$R_{rms}^2 \simeq \langle \Delta x_1^2 + \Delta y_1^2 \rangle + \langle \Delta x_2^2 + \Delta y_2^2 \rangle + \cdots = N\langle r^2 \rangle = N r_{rms}^2,$$

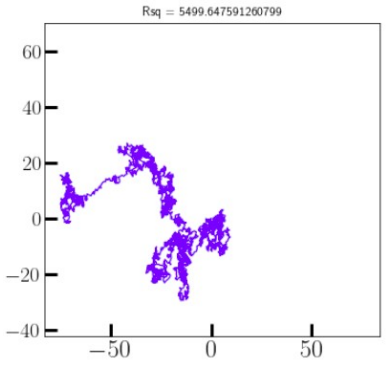
$$R_{rms} \simeq \sqrt{N} r_{rms},$$

Monte Carlo Methods in Python

Example problems: See Jupyter Notebook MC.ipynb

Random Walk Problem: Given the mean free path of air at room temperature (68 nm), how many collisions will it take for a single molecule suspended in air to travel 20 metres?

Molecule = Random Walker

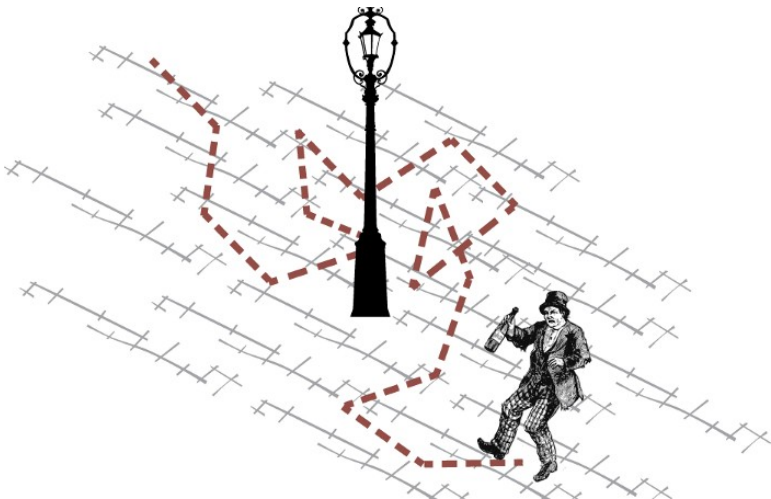


$$R^2 = (\Delta x_1 + \Delta x_2 + \cdots + \Delta x_N)^2 + (\Delta y_1 + \Delta y_2 + \cdots + \Delta y_N)^2$$

Average over a large number of trials:

$$R_{rms}^2 \simeq \langle \Delta x_1^2 + \Delta y_1^2 \rangle + \langle \Delta x_2^2 + \Delta y_2^2 \rangle + \cdots = N\langle r^2 \rangle = N r_{rms}^2,$$

$$R_{rms} \simeq \sqrt{N} r_{rms},$$

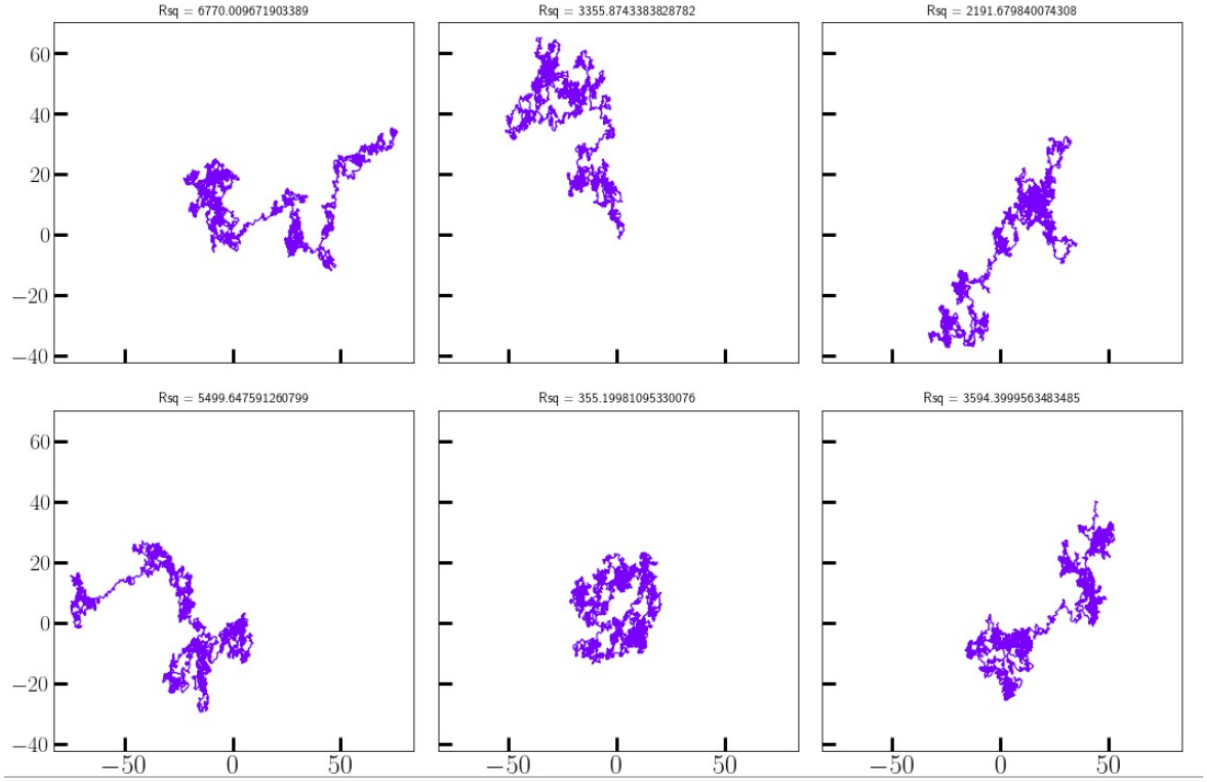


Monte Carlo Methods in Python

Example problems: See Jupyter Notebook MC.ipynb

Random Walk Problem: Given the mean free path of air at room temperature (68 nm), how many collisions will it take for a single molecule suspended in air to travel 20 metres?

```
import numpy as np
plt.rcParams['figure.figsize'] = (15,10)
plt.rcParams['font.size'] = 20

fig, axs = plt.subplots(2,3, sharex=True, sharey=True)
axs = axs.flatten()

jmax = 2500

for ax in axs:
    dr_vals = 2.0 * np.random.random((jmax,2)) -1
    Lvals = np.sum(dr_vals**2, axis=1)
    dr_vals /= np.sqrt(Lvals[:, None])

    points = np.cumsum(dr_vals, axis=0)
    Rsq = np.linalg.norm(points[-1])**2
    ax.set_title(f"Rsq = {Rsq}", fontsize=12)
    ax.plot(points[:,0], points[:,1], color='blue', alpha=0.6)
plt.show()
```

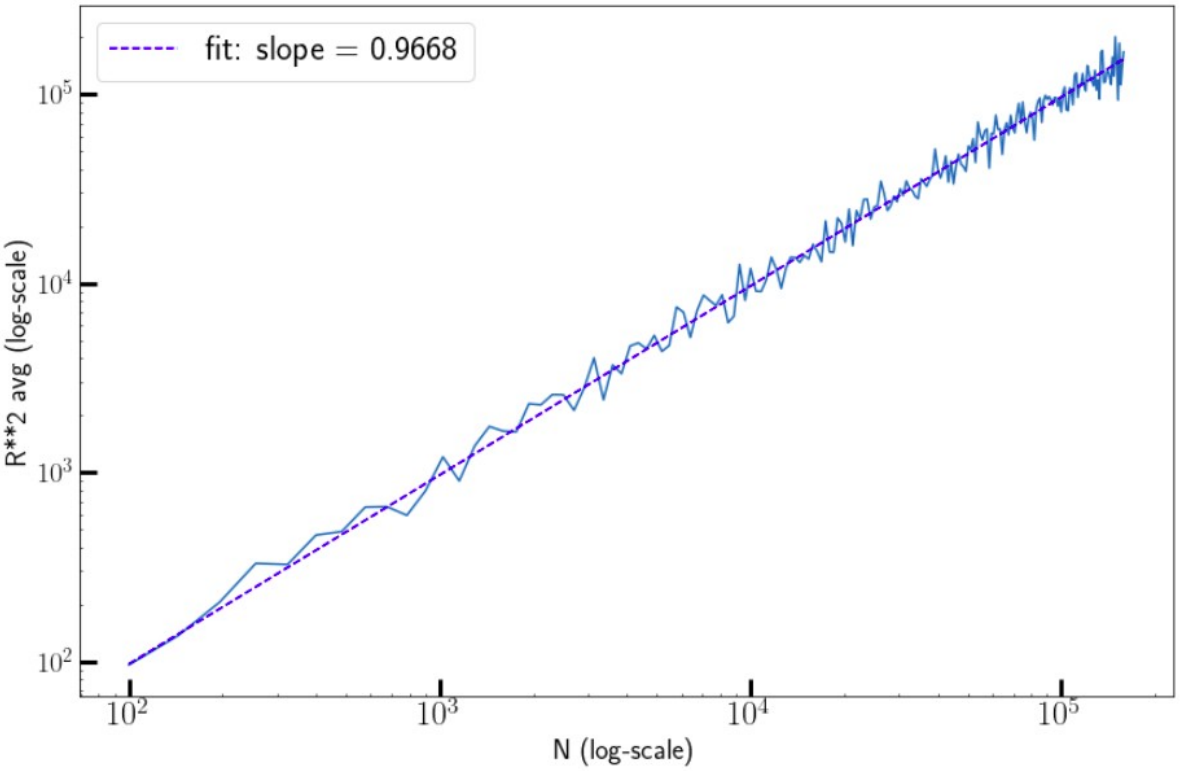


Monte Carlo Methods in Python

Example problems: See Jupyter Notebook MC.ipynb

Random Walk Problem: Given the mean free path of air at room temperature (68 nm), how many collisions will it take for a single molecule suspended in air to travel 20 metres?

```
import matplotlib.pyplot as plt
from scipy.optimize import curve_fit
import numpy as np
plt.rcParams['figure.figsize'] = (12,8)
plt.rcParams['font.size'] = 20

def randwalk2d(nsteps, repeat=6):
    points = []
    for i in range(repeat):
        dr_vals = 2.0 * np.random.random((nsteps,2)) -1
        Lvals = np.sum(dr_vals**2, axis=1)
        dr_vals /= np.sqrt(Lvals[:, None])
        points.append(np.cumsum(dr_vals, axis=0))
    return points

nsteps = np.arange(10,400,2)**2
Rsq_avs = []
for nstp in nsteps:
    walks = randwalk2d(nstp, repeat=50)
    Rsq_avs.append(np.average([np.linalg.norm(w[-1])**2 for w in walks]))
plt.loglog(nsteps, Rsq_avs)
popt, pcov = curve_fit(lambda x,a: a * x, nsteps, Rsq_avs)
plt.loglog(nsteps, popt[0] * nsteps, 'b--', label=f'fit: slope = {popt[0]:2.4f}')
plt.xlabel("N (log-scale)")
plt.ylabel("R**2 avg (log-scale)")
plt.legend(); plt.show()
```



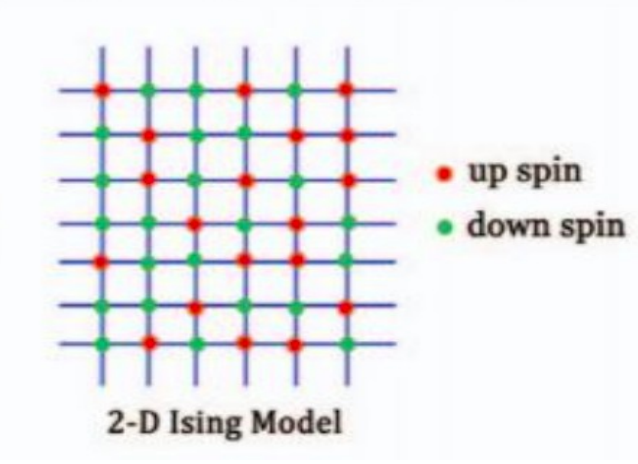
Monte Carlo Methods in Python

Demo: Ising Model and the Metropolis Algorithm

$$H = \begin{cases} -\sum_i s_i s_{i+1} - h \sum_i s_i & 1-D \\ -\sum_{ij} s_{i,j} \left(s_{i,j+1} + s_{i+1,j} \right) - h \sum_{i,j} s_{i,j} & 2-D \text{ square} \end{cases}$$

Cannot get the exact partition function numerically
Too many combinations of spins for a large size

$$N \rightarrow 2^N$$



Monte Carlo Methods in Python

Demo: Ising Model and the Metropolis Algorithm

$$H = \begin{cases} -\sum_i s_i s_{i+1} - h\sum_i s_i & 1-D \\ -\sum_{ij} s_{i,j}\left(s_{i,j+1} + s_{i+1,j}\right) - h\sum_{i,j} s_{i,j} & 2-D \text{ square} \end{cases}$$

The **Metropolis Algorithm** is a technique for computing the Monte Carlo calculation of averages that accurately simulates the fluctuations occurring during thermal equilibrium.

It changes individual spins randomly, but weights the changes such that on the average a Boltzmann distribution results (Canonical Ensemble)

$$\mathcal{P}(E_{\alpha_j}, T) = \frac{e^{-E_{\alpha_j}/k_B T}}{Z(T)}$$

Monte Carlo Methods in Python

Demo: Ising Model and the Metropolis Algorithm

$$H = \begin{cases} -\sum_i s_i s_{i+1} - h\sum_i s_i & 1-D \\ -\sum_{ij} s_{i,j}\left(s_{i,j+1} + s_{i+1,j}\right) - h\sum_{i,j} s_{i,j} & 2-D \text{ square} \end{cases}$$

1. Start with an arbitrary spin configuration $\alpha_k = \{s_1, s_2, \ldots, s_N\}$.
2. Generate a *trial* configuration α_{k+1}:
   a. Pick a particle i randomly and flip its spin.
   b. Calculate the energy $E_{\alpha_{\text{tr}}}$ of the trial configuration.
   c. If $E_{\alpha_{\text{tr}}} \leq E_{\alpha_k}$, accept the trial by setting $\alpha_{k+1} = \alpha_{\text{tr}}$.
   d. If $E_{\alpha_{\text{tr}}} > E_{\alpha_k}$, accept with relative probability $\mathcal{R} = \exp(-\Delta E/k_B T)$:
      a. Choose a uniform random number $0 \leq r_i \leq 1$.
      b. Set $\alpha_{k+1} = \begin{cases} \alpha_{\text{tr}}, & \text{if } \mathcal{R} \geq r_j \quad \text{(accept)}, \\ \alpha_k, & \text{if } \mathcal{R} < r_j \quad \text{(reject)}. \end{cases}$

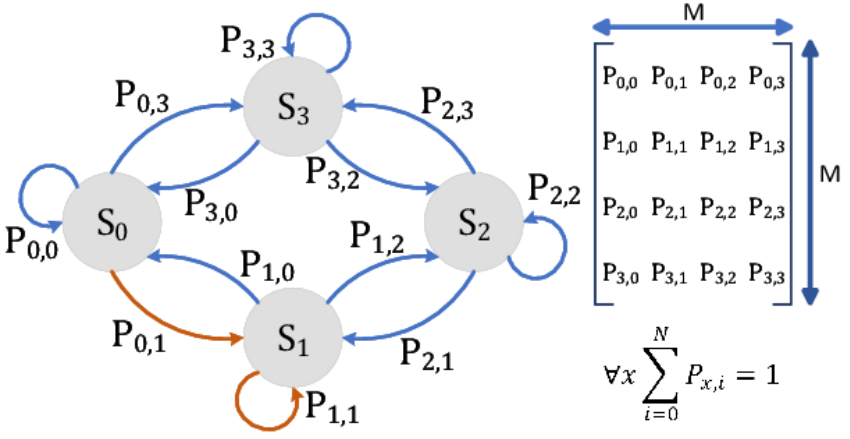
Monte Carlo Methods in Python

Monte Carlo Methods in Python

Demo: Ising Model and the Metropolis Algorithm

$$H = \begin{cases} -\sum_i s_i s_{i+1} - h\sum_i s_i & 1-D \\ -\sum_{ij} s_{i,j}\left(s_{i,j+1} + s_{i+1,j}\right) - h\sum_{i,j} s_{i,j} & 2-D \text{ square} \end{cases}$$

Will this equilibriate into a Canonical Ensemble?



Markov Chain:
Each AB transition depends only on

$$P_{AB}$$

Detailed Balance:
Happens when all transition probs balance out

$$p_{A\to B} = p_A P_{AB} = p_{B\to A} = p_B P_{BA},$$

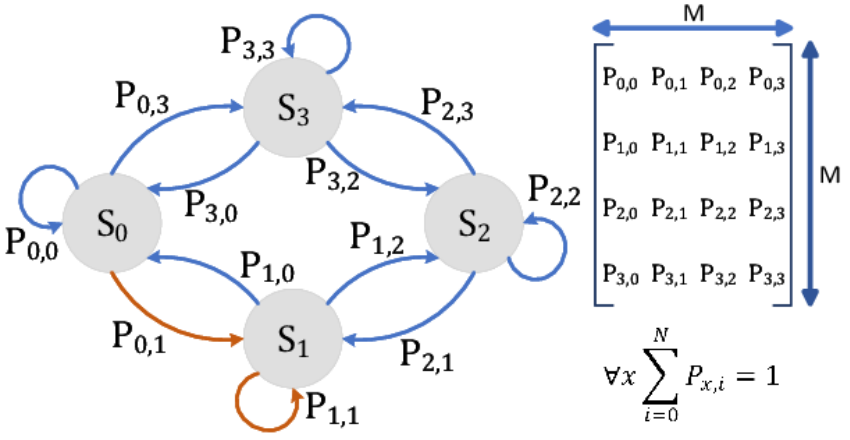
Monte Carlo Methods in Python

Monte Carlo Methods in Python

Demo: Ising Model and the Metropolis Algorithm

$$H = \begin{cases} -\sum_i s_i s_{i+1} - h\sum_i s_i & 1-D \\ -\sum_{ij} s_{i,j}\left(s_{i,j+1} + s_{i+1,j}\right) - h\sum_{i,j} s_{i,j} & 2-D \text{ square} \end{cases}$$

Will this equilibriate into a Canonical Ensemble?



Detailed Balance:
Happens when all transition probs balance out

$$p_{A\to B} = p_A P_{AB} = p_{B\to A} = p_B P_{BA},$$

$$\frac{p_A}{p_B} = \frac{\pi_{BA}}{\pi_{AB}} = \frac{p_{BA}^{acc}}{p_{AB}^{acc}}.$$

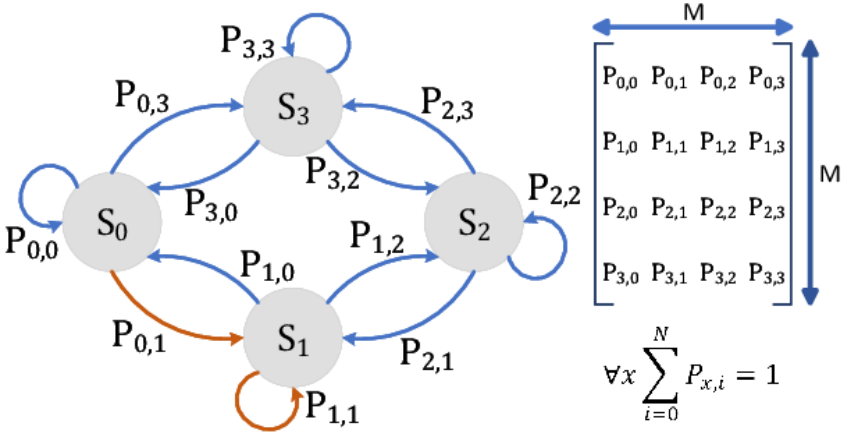
Monte Carlo Methods in Python

Monte Carlo Methods in Python

Demo: Ising Model and the Metropolis Algorithm

$$H = \begin{cases} -\sum_i s_i s_{i+1} - h\sum_i s_i & 1-D \\ -\sum_{ij} s_{i,j}\left(s_{i,j+1} + s_{i+1,j}\right) - h\sum_{i,j} s_{i,j} & 2-D \text{ square} \end{cases}$$

Will this equilibriate into a Canonical Ensemble?



Detailed Balance:
Happens when all transition probs balance out

$$\frac{p_A}{p_B} = \frac{\pi_{BA}}{\pi_{AB}} = \frac{p_{BA}^{acc}}{p_{AB}^{acc}}.$$

$$p_{AB}^{acc} \sim e^{\beta(E_A - E_B)}, \; p_{BA}^{acc} = 1$$

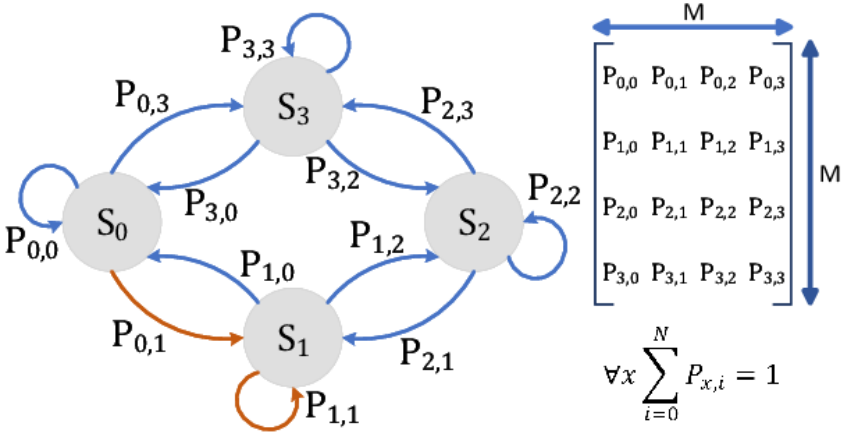
Monte Carlo Methods in Python

Monte Carlo Methods in Python

Demo: Ising Model and the Metropolis Algorithm

$$H = \begin{cases} -\sum_i s_i s_{i+1} - h\sum_i s_i & 1-D \\ -\sum_{ij} s_{i,j}\left(s_{i,j+1} + s_{i+1,j}\right) - h\sum_{i,j} s_{i,j} & 2-D \text{ square} \end{cases}$$

Will this equilibriate into a Canonical Ensemble?



Detailed Balance:
Happens when all transition probs balance out

$$p_r \sim \exp\left(-\beta E_r\right)$$

Monte Carlo Methods in Python

Demo: Ising Model and the Metropolis Algorithm

$$H = \begin{cases} -\sum_i s_i s_{i+1} - h\sum_i s_i & 1-D \\ -\sum_{ij} s_{i,j}\left(s_{i,j+1} + s_{i+1,j}\right) - h\sum_{i,j} s_{i,j} & 2-D \text{ square} \end{cases}$$

Videos: 2D without field

$$N = 150 \times 150,$$
$$T_H = 10J/k_B,$$

Monte Carlo Methods in Python

Demo: Ising Model and the Metropolis Algorithm

$$H = \begin{cases} -\sum_i s_i s_{i+1} - h\sum_i s_i & 1-D \\ -\sum_{ij} s_{i,j}\left(s_{i,j+1} + s_{i+1,j}\right) - h\sum_{i,j} s_{i,j} & 2-D \text{ square} \end{cases}$$

Videos: 2D without field

$$N = 150 \times 150,$$
$$T_L = 0.1 J/k_B$$

Partial Differential Equations

PDE: Imposes relations between the various partial derivatives of a multivariable function.

$$f\left(x, y, z \ldots \frac{\partial u}{\partial x}, \frac{\partial u}{\partial y}, \frac{\partial u}{\partial z} \ldots \frac{\partial^2 u}{\partial x^2}, \frac{\partial^2 u}{\partial y^2}, \frac{\partial^2 u}{\partial z^2} \ldots\right) = 0$$

Solve for u(x,y,z, ...), subject to **Boundary Conditions**

Partial Differential Equations

PDE: Imposes relations between the various partial derivatives of a multivariable function.

Types of Linear PDE:

1) Parabolic: $\Delta = 0$ $\quad \frac{\partial u}{\partial t} = \alpha \frac{\partial^2 u}{\partial x^2}$ Heat Equation

2) Hyperbolic: $\Delta > 0$ $\quad \frac{\partial^2 u}{\partial t^2} = c^2 \frac{\partial^2 u}{\partial x^2}$ Wave Equation

3) Elliptic: $\Delta < 0$ $\quad \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = 0$ Laplace Equation

$$A\frac{\partial^2}{\partial x^2} + 2B\frac{\partial^2}{\partial x \partial y} + C\frac{\partial^2}{\partial y^2} + \ldots \text{ lower order terms } \cdots = 0$$

$$\Delta \equiv B^2 - AC$$

Partial Differential Equations

PDE: Imposes relations between the various partial derivatives of a multivariable function.

Types of Boundary Conditions in PDE's:

1) Dirichlet: $u(x,y) \; \forall \, (x,y) \in \partial\mathcal{B}$

2) Neumann: $\vec{\nabla} u(x,y) \cdot \hat{n}(x,y) \; \forall \, (x,y) \in \partial\mathcal{B}$

3) Cauchy: Dirichlet + Neumann

$$A\frac{\partial^2}{\partial x^2} + 2B\frac{\partial^2}{\partial x \partial y} + C\frac{\partial^2}{\partial y^2} + \ldots \text{ lower order terms } \cdots = 0$$

$$\text{Domain } \; \mathcal{B} \; \text{ Boundary } \partial\mathcal{B}$$

Partial Differential Equations

PDE: Imposes relations between the various partial derivatives of a multivariable function.

Numerical Methods for PDE's

Finite Difference Methods (FDM):

Approximate the solutions to using finite difference equations to approximate derivatives. Reduce the ODE to a Linear Algebra Equation

Finite Volume Methods (FVM):

Divide the domain into subdomains and use Gauss/Stoke's theorem to reduce PDE to integral equations, solved using quadratures in alegebraic form

Finite Element Methods (FEM):

Divide the domain intosubdomains and use Gauss/Stoke's theorem to reduce PDE to integral equations, weighed by a choice of basis states, discretize them solve as linear eqns.

$$A\frac{\partial^2}{\partial x^2} + 2B\frac{\partial^2}{\partial x \partial y} + C\frac{\partial^2}{\partial y^2} + \ldots \text{ lower order terms } \cdots = 0$$

$$\text{Domain } \mathcal{B} \quad \text{Boundary } \partial\mathcal{B}$$

Partial Differential Equations

PDE: Imposes relations between the various partial derivatives of a multivariable function.

Numerical Methods for PDE's: Unline ODE's there are no general methods

- Numerical Solutions to PDEs are more complex than ODEs
- Each type of PDE requires its own algorithm
- Algorithms are also dependent on the boundary conditions

$$A\frac{\partial^2}{\partial x^2} + 2B\frac{\partial^2}{\partial x \partial y} + C\frac{\partial^2}{\partial y^2} + \ldots \text{ lower order terms } \cdots = 0$$

$$\text{Domain } \mathcal{B} \quad \text{Boundary } \partial\mathcal{B}$$

Finite Difference Methods: 2D Poisson Equation (Dirichlet)

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = -4\pi\rho(x, y)$$

1) Divide the 2D Domain into a discrete lattice.

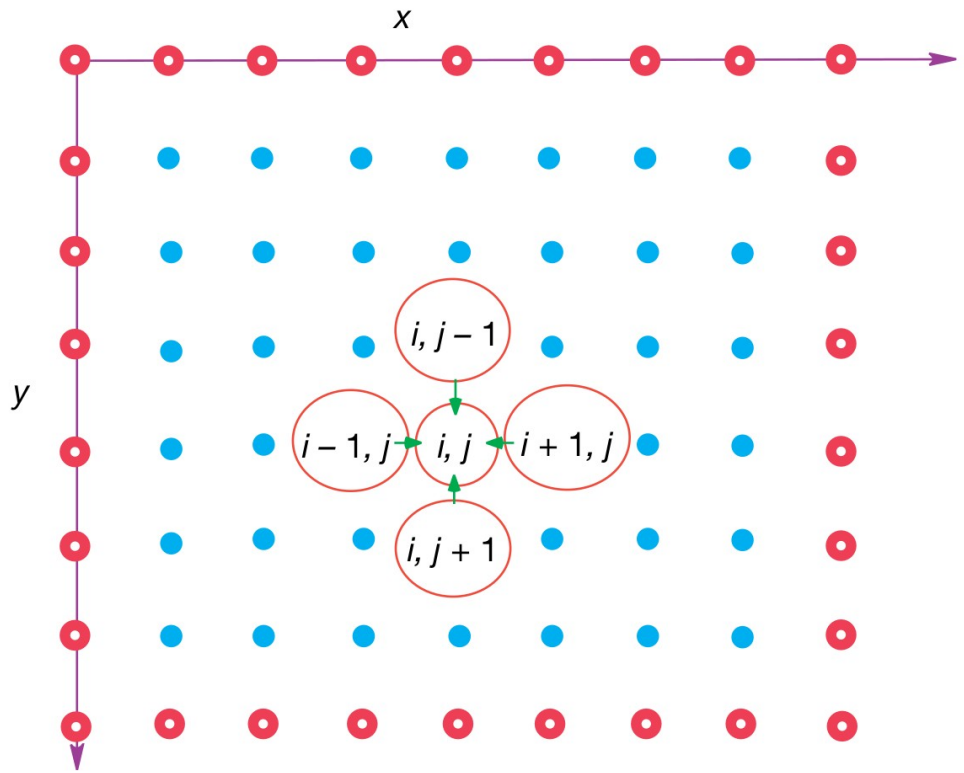
2) Replace the partial derivatives by finite difference approximations evaluated at the lattice points

3) Reduce the PDE to a system of linear algebra equations, with boundary conditions added as extra equations, then solve them (similar to the ‘matrix method’ for ODE’s)

4) Alternatively, reduce the PDE to a system of iterative equations, then update the domain values, starting from the boundary values.

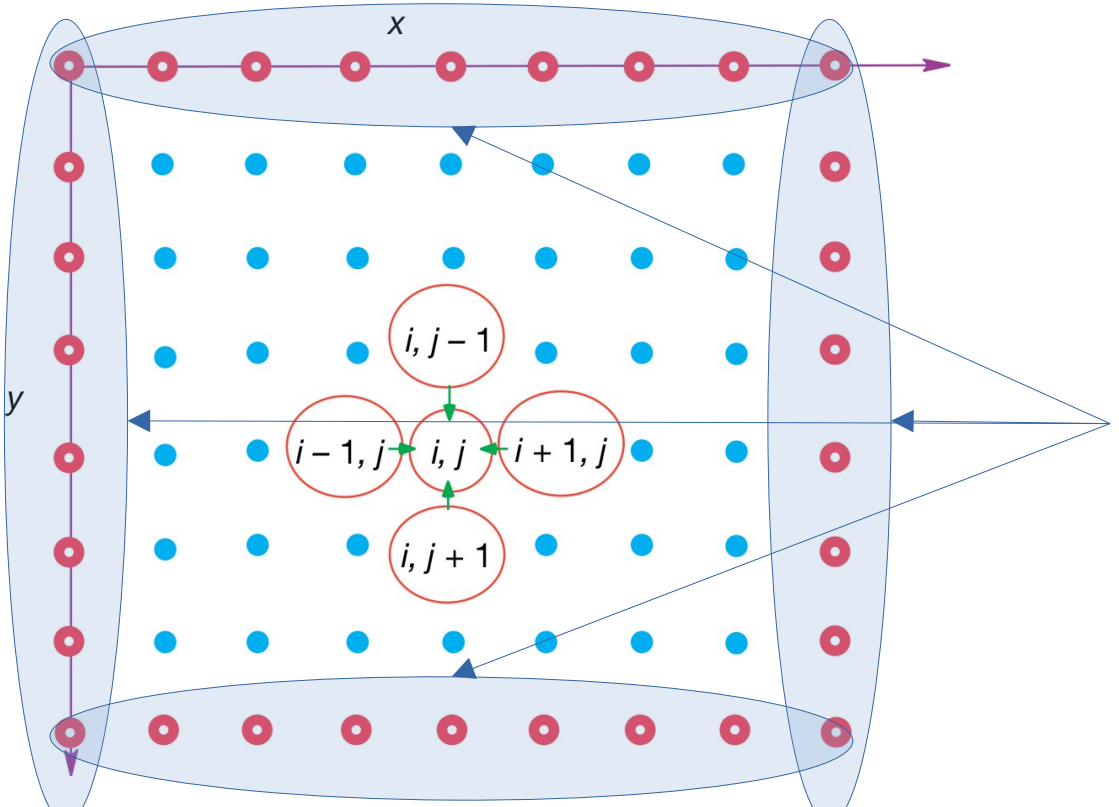
Finite Difference Methods: 2D Poisson Equation (Dirichlet)

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = -4\pi\rho(x, y)$$



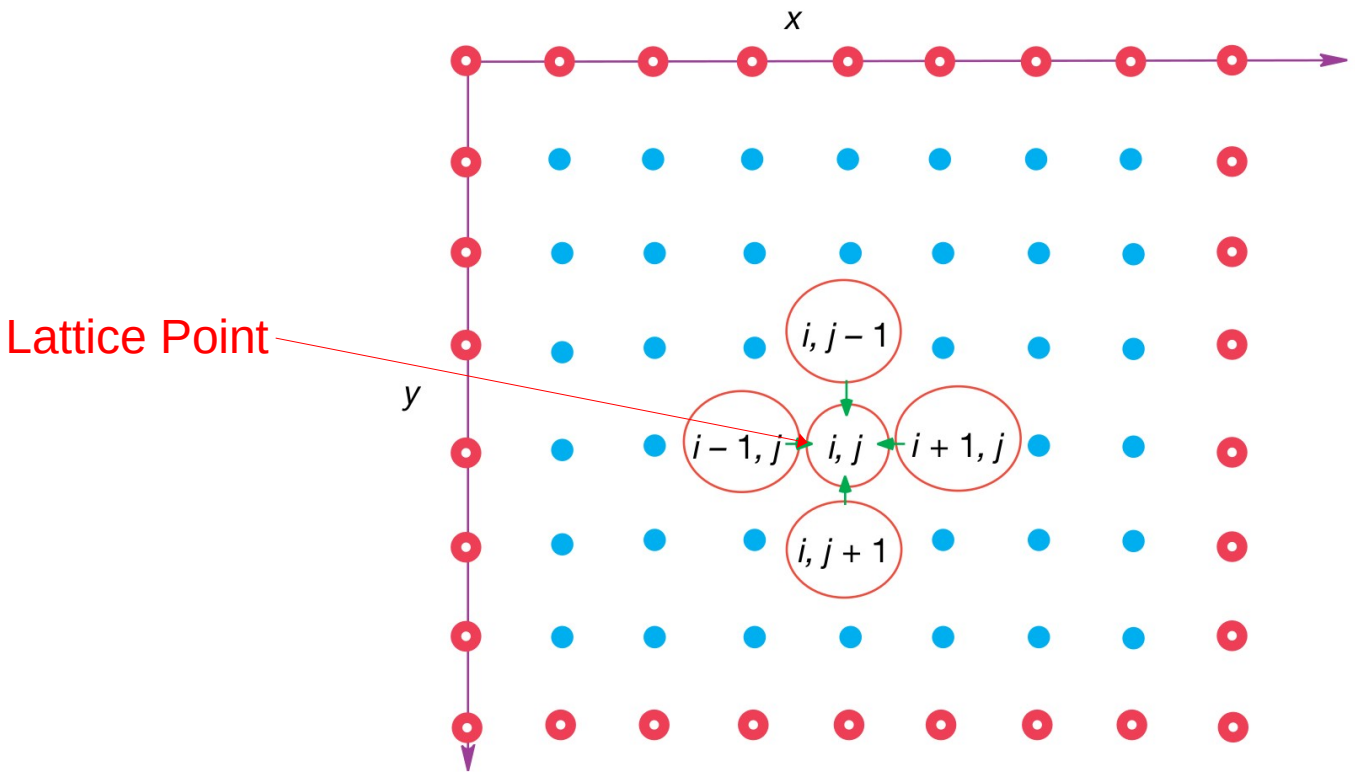
Finite Difference Methods: 2D Poisson Equation (Dirichlet)

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = -4\pi\rho(x, y)$$



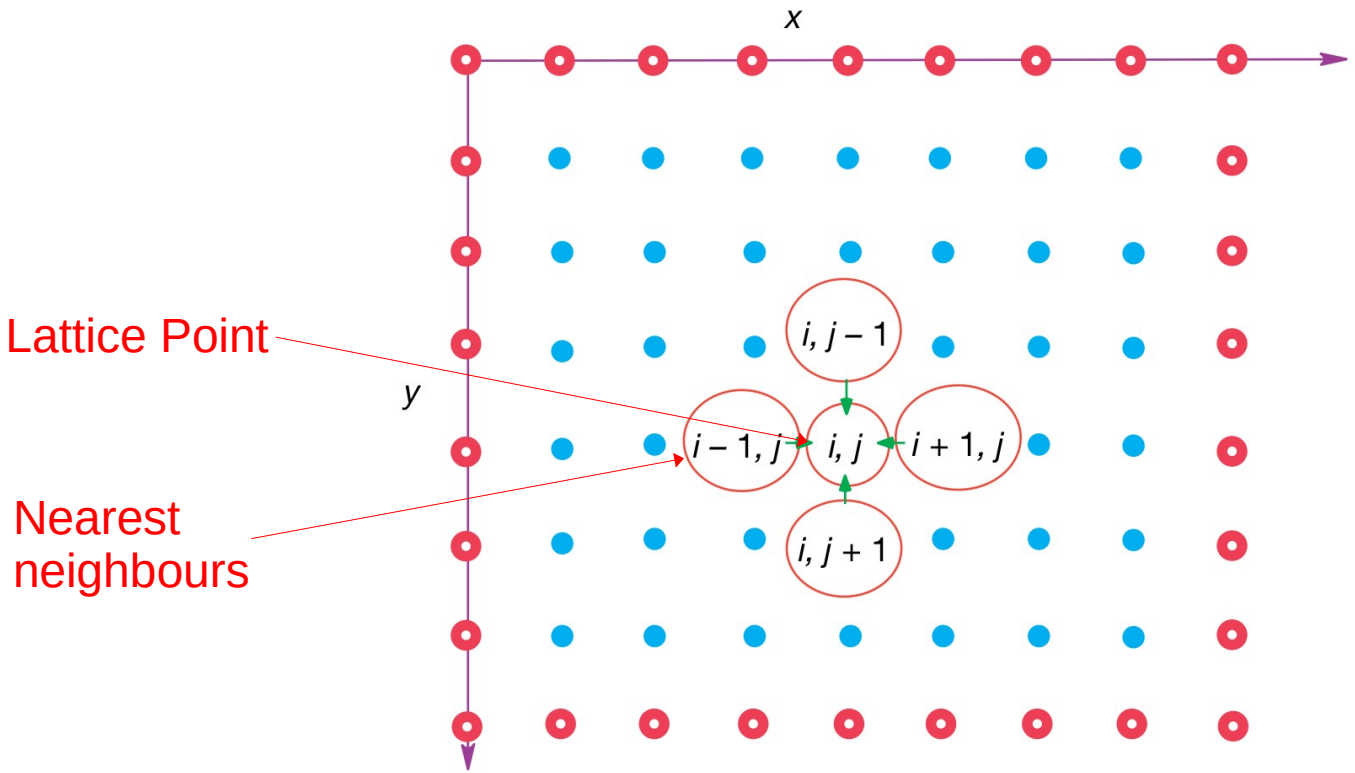
Finite Difference Methods: 2D Poisson Equation (Dirichlet)

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = -4\pi\rho(x, y)$$



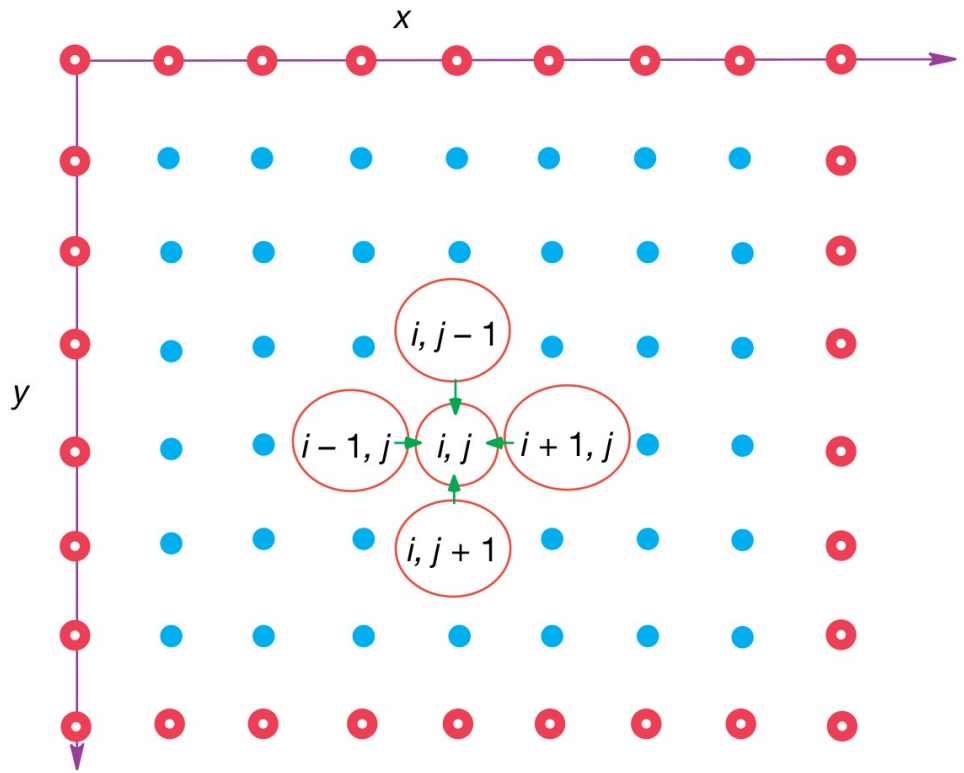
Finite Difference Methods: 2D Poisson Equation (Dirichlet)

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = -4\pi\rho(x, y)$$



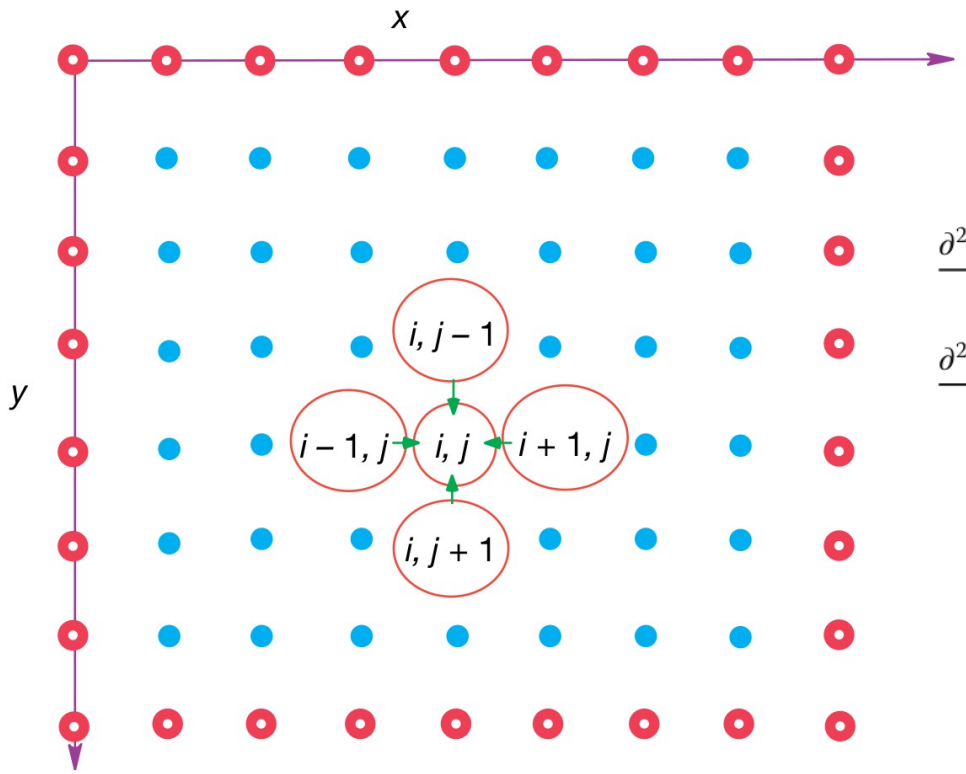
Finite Difference Methods: 2D Poisson Equation (Dirichlet)

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = -4\pi\rho(x, y)$$



Finite Difference Methods: 2D Poisson Equation (Dirichlet)

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = -4\pi\rho(x, y)$$



Three point difference formula!

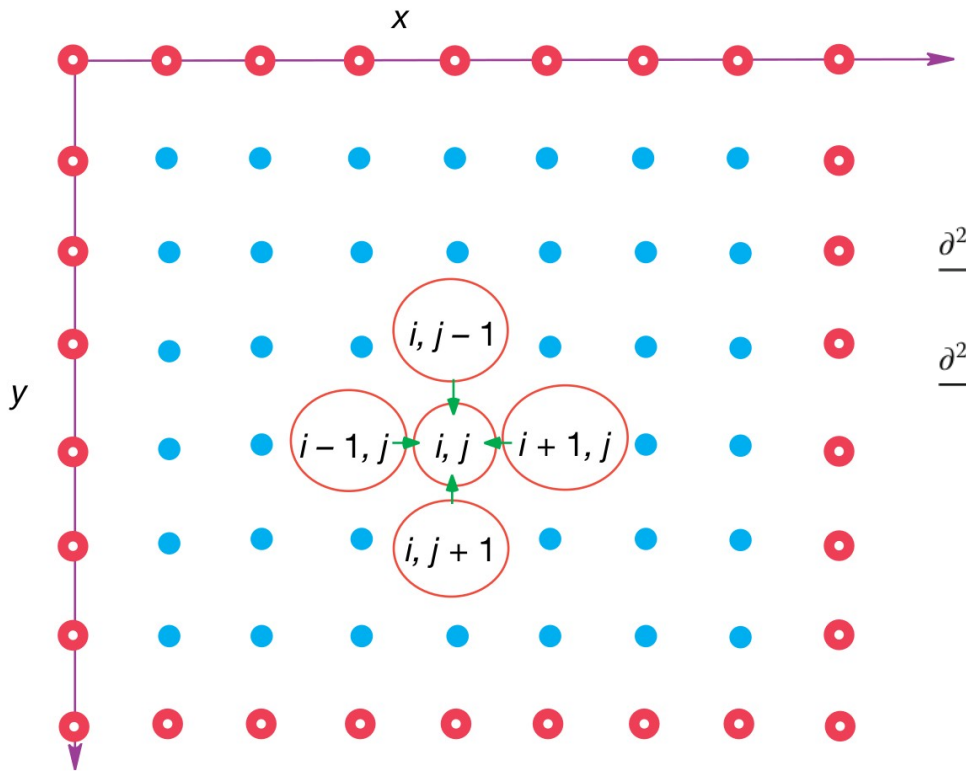
$$\frac{\partial^2 U(x, y)}{\partial x^2} \simeq \frac{U(x + \Delta x, y) + U(x - \Delta x, y) - 2U(x, y)}{(\Delta x)^2},$$

$$\frac{\partial^2 U(x, y)}{\partial y^2} \simeq \frac{U(x, y + \Delta y) + U(x, y - \Delta y) - 2U(x, y)}{(\Delta y)^2}.$$

Finite Difference Methods: 2D Poisson Equation (Dirichlet)

$$\frac{U(x+\Delta x, y)+U(x-\Delta x, y)-2U(x, y)}{(\Delta x)^2}$$

$$+\frac{U(x, y+\Delta y)+U(x, y-\Delta y)-2U(x, y)}{(\Delta y)^2}=-4\pi\rho\ .$$



Three point difference formula!

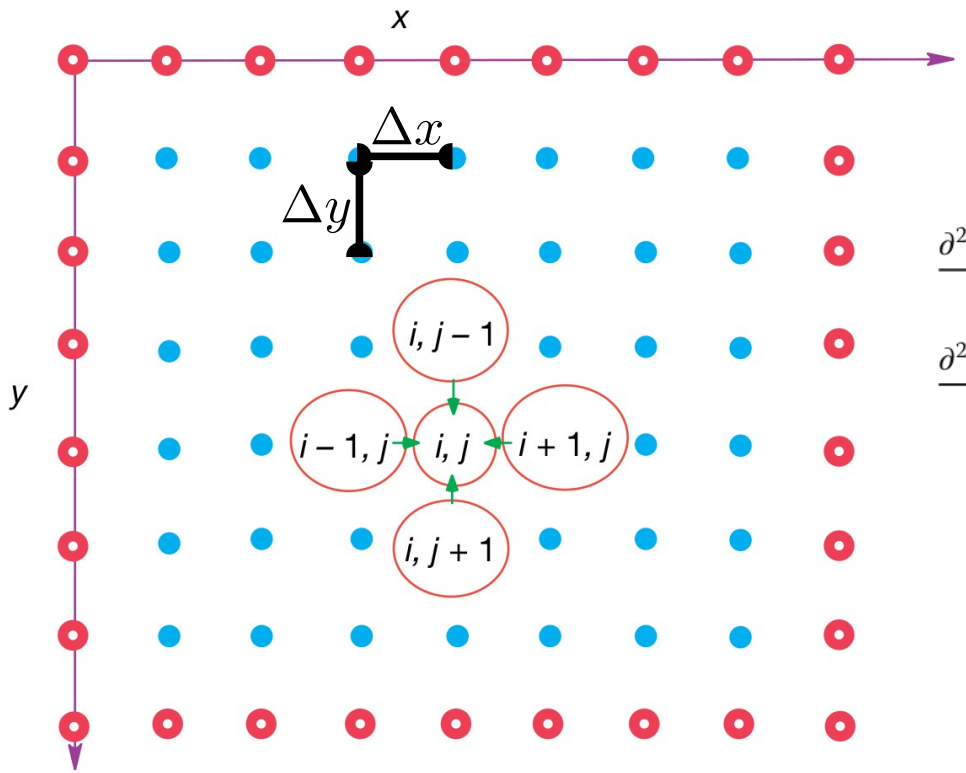
$$\frac{\partial^2 U(x, y)}{\partial x^2} \simeq \frac{U(x+\Delta x, y)+U(x-\Delta x, y)-2U(x, y)}{(\Delta x)^2}\ ,$$

$$\frac{\partial^2 U(x, y)}{\partial y^2} \simeq \frac{U(x, y+\Delta y)+U(x, y-\Delta y)-2U(x, y)}{(\Delta y)^2}\ .$$

Finite Difference Methods: 2D Poisson Equation (Dirichlet)

$$\frac{U(x+\Delta x, y)+U(x-\Delta x, y)-2U(x, y)}{(\Delta x)^2}$$

$$+\frac{U(x, y+\Delta y)+U(x, y-\Delta y)-2U(x, y)}{(\Delta y)^2}=-4\pi\rho\ .$$

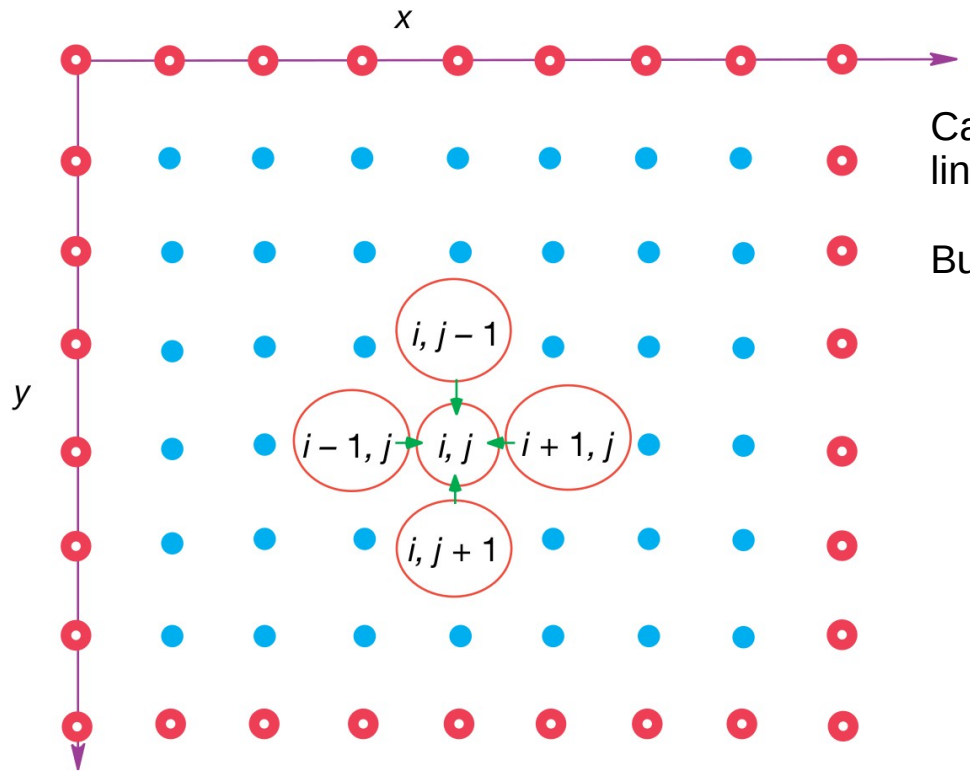


Three point difference formula!

$$\frac{\partial^2 U(x, y)}{\partial x^2} \simeq \frac{U(x+\Delta x, y)+U(x-\Delta x, y)-2U(x, y)}{(\Delta x)^2},$$

$$\frac{\partial^2 U(x, y)}{\partial y^2} \simeq \frac{U(x, y+\Delta y)+U(x, y-\Delta y)-2U(x, y)}{(\Delta y)^2}.$$

Finite Difference Methods: 2D Poisson Equation (Dirichlet)

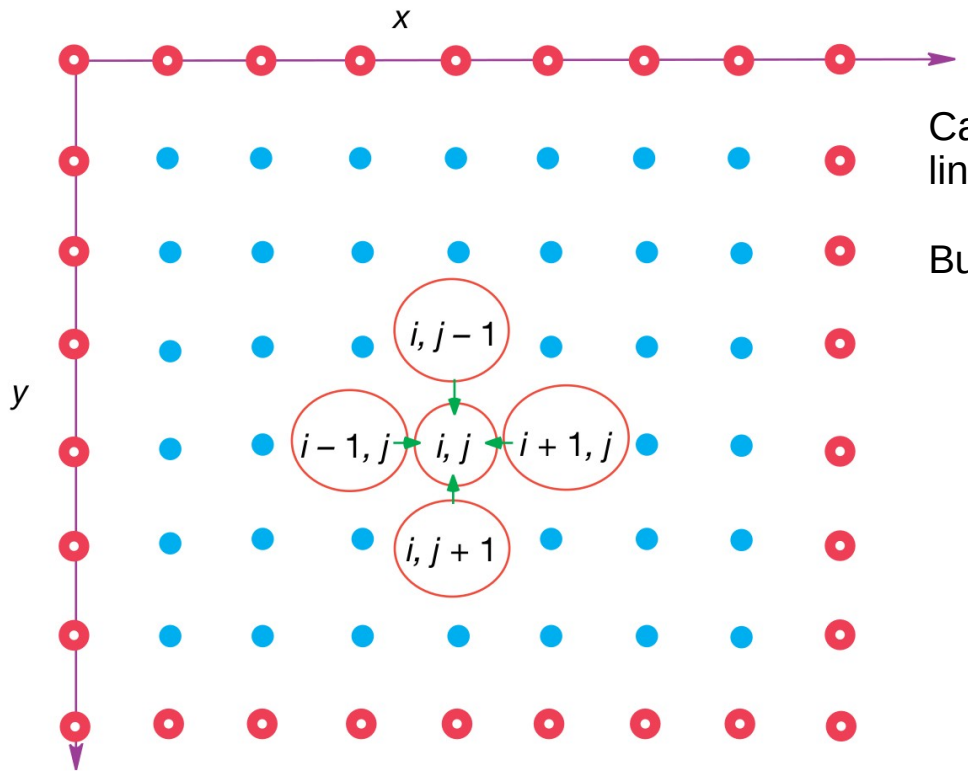


Can solve these equations numerically using linear algebra methods

But, there is a better way (less memory)

Finite Difference Methods: 2D Poisson Equation (Dirichlet)

$$U_{i,j} = \frac{1}{4}\left[U_{i+1,j} + U_{i-1,j} + U_{i,j+1} + U_{i,j-1}\right] + \pi\rho(i\Delta, j\Delta)\Delta^2$$

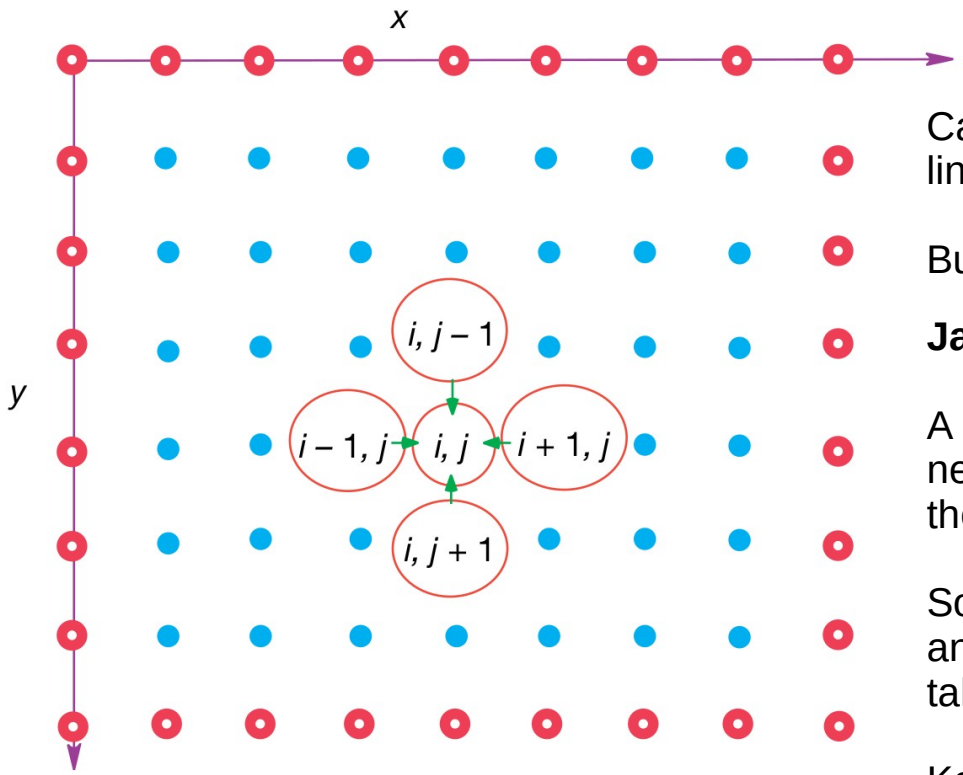


Can solve these equations numerically using linear algebra methods

But, there is a better way (less memory)

Finite Difference Methods: 2D Poisson Equation (Dirichlet)

$$U_{i,j} = \frac{1}{4}\left[U_{i+1,j} + U_{i-1,j} + U_{i,j+1} + U_{i,j-1}\right] + \pi\rho(i\Delta, j\Delta)\Delta^2$$



Can solve these equations numerically using linear algebra methods

But, there is a better way (less memory)

Jacobi Update:

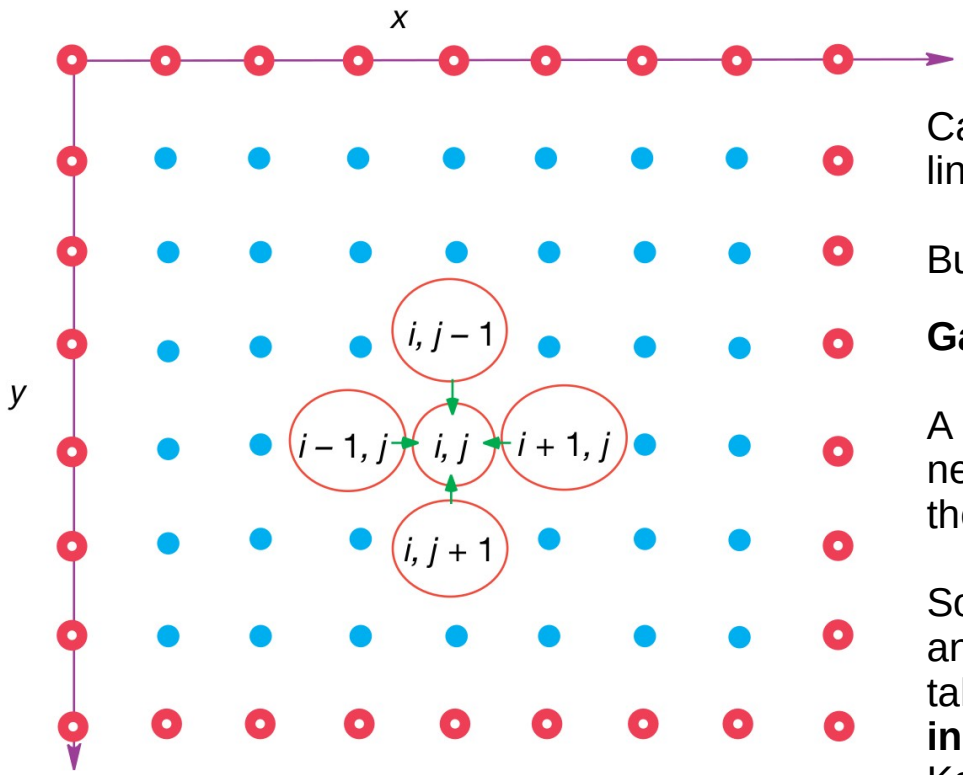
A proper solution will be the **average** of the nearest neighbors plus a contribution from the charge density

So, start with an initial guess for the potential and improve it by striding through the lattice taking the average over nearest neighbors

Keep doing this until solution converges

Finite Difference Methods: 2D Poisson Equation (Dirichlet)

$$U_{i,j}^{(\text{new})} = \frac{1}{4}\left[U_{i+1,j}^{(\text{old})} + U_{i-1,j}^{(\text{new})} + U_{i,j+1}^{(\text{old})} + U_{i,j-1}^{(\text{new})}\right]$$



Can solve these equations numerically using linear algebra methods

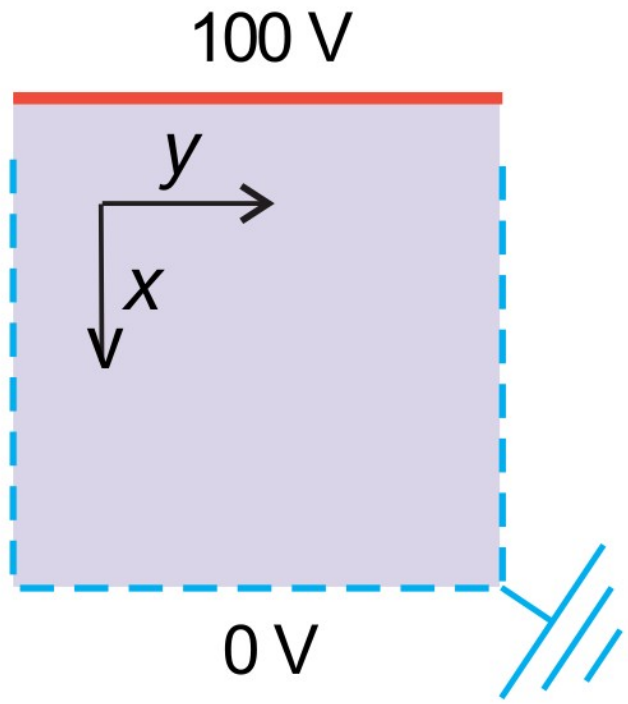
But, there is a better way (less memory)

Gauss-Seidel Update:

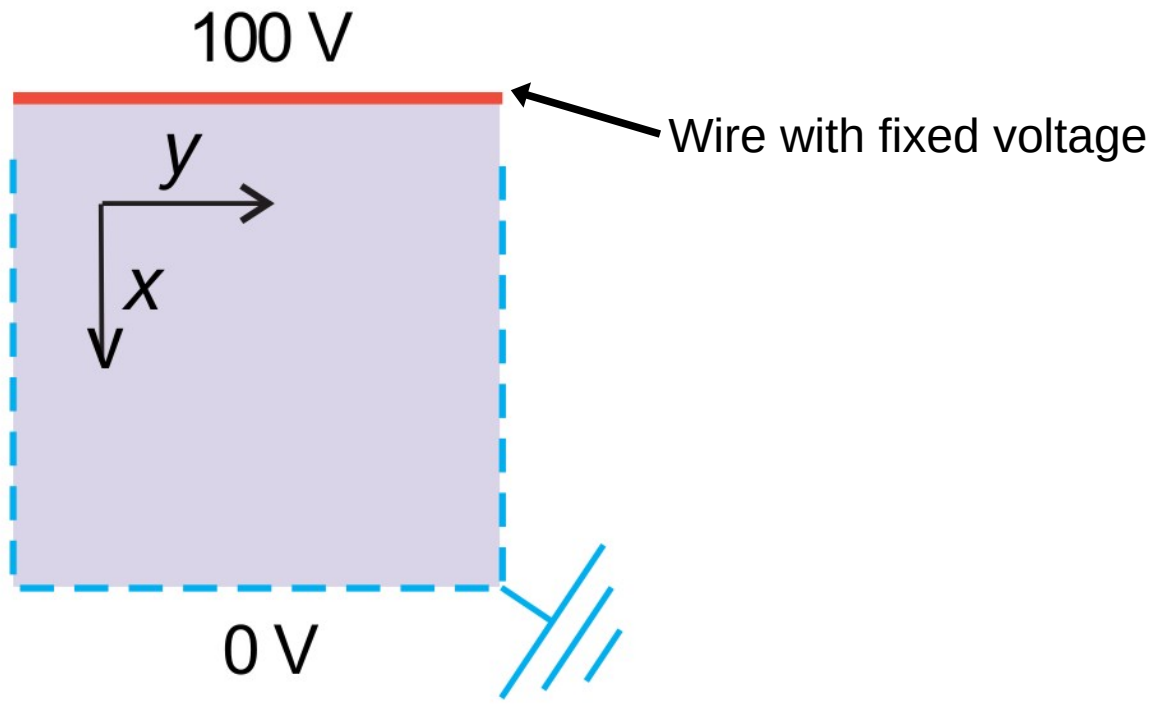
A proper solution will be the **average** of the nearest neighbors plus a contribution from the charge density

So, start with an initial guess for the potential and improve it by striding through the lattice taking the average over nearest neighbors **in real time.**
Keep doing this until solution converges

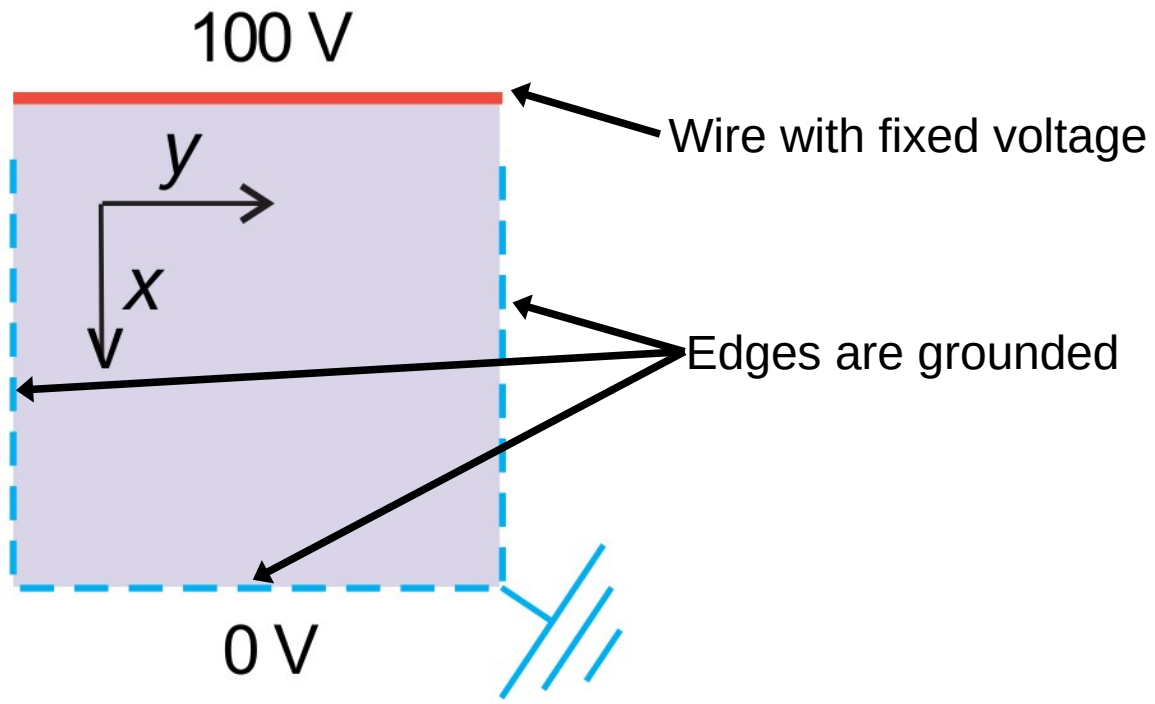
Finite Difference Methods: 2D Poisson Equation (Dirichlet)



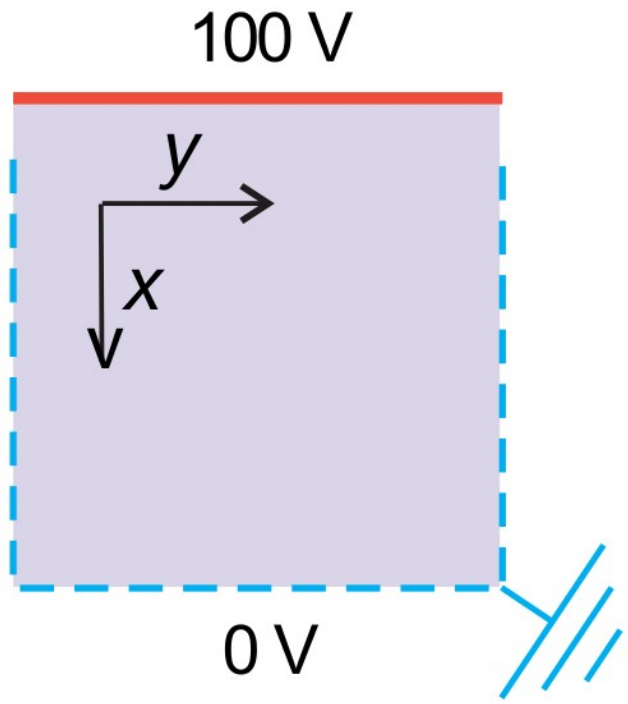
Finite Difference Methods: 2D Poisson Equation (Dirichlet)



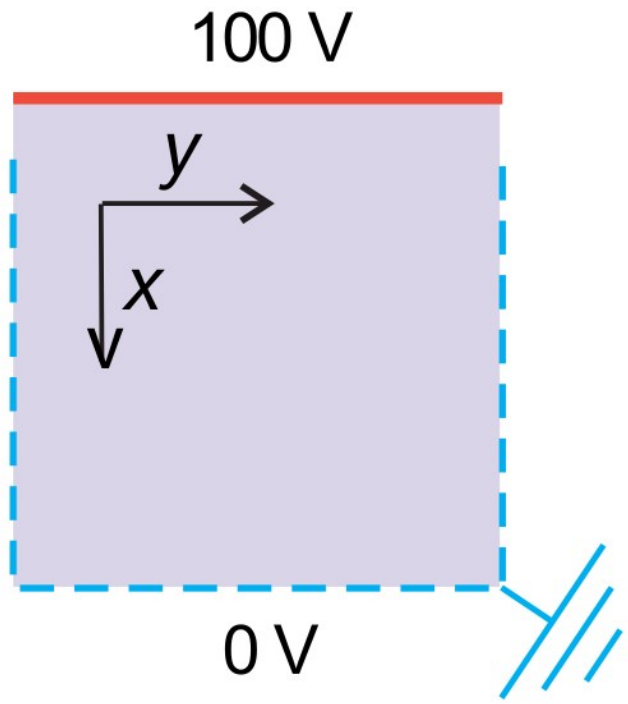
Finite Difference Methods: 2D Poisson Equation (Dirichlet)



Finite Difference Methods: 2D Poisson Equation (Dirichlet)



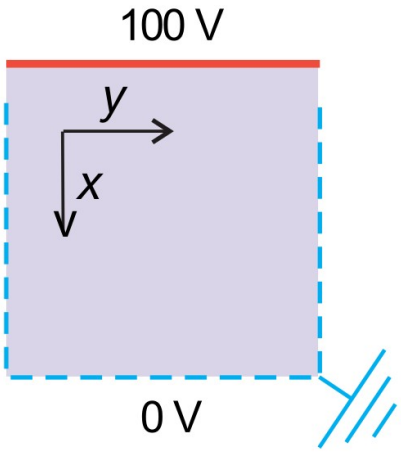
Finite Difference Methods: 2D Poisson Equation (Dirichlet)



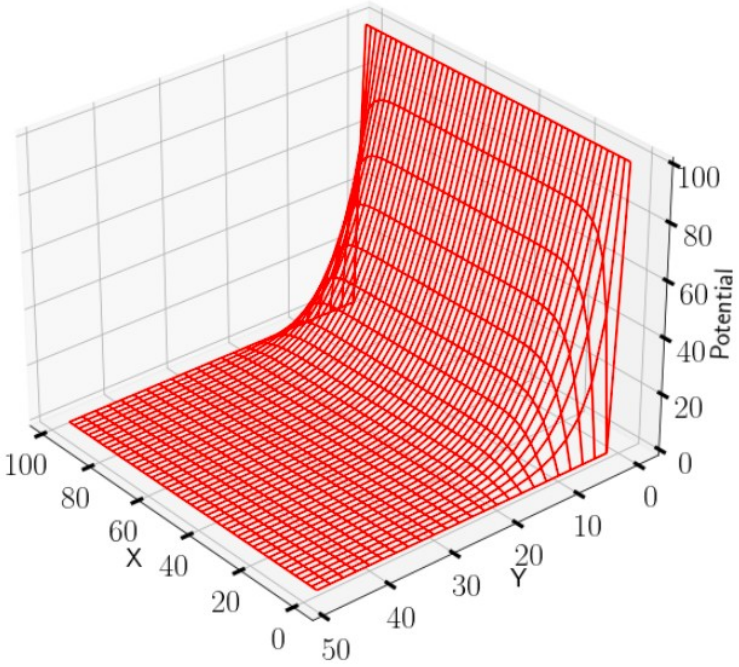

```
import numpy as np
import matplotlib.pyplot as plt

plt.rcParams['figure.figsize'] = (12,8)
plt.rcParams['font.size'] = 20


Nmax = 100
V_wire = 100

V = np.zeros((Nmax, Nmax))
V[:,0] = V_wire                                  # Line at V_wire volts

print ("Running the Jacobi Update Solver ...")

def jacobi_update(V, Niter=70):
    Nmax = V.shape[0]
    for iter in range(Niter):
        for i in range(1, Nmax-2):
            for j in range(1,Nmax-2):
                V[i,j] = 0.25*(V[i+1,j]+V[i-1,j]+V[i,j+1]+V[i,j-1])
    return V

V = jacobi_update(V)
x = np.arange(0, Nmax-1, 2);  y = np.arange(0, 50, 2)
X, Y = np.meshgrid(x,y)

def functz(V):                                   # V(x, y)
    z = V[X,Y]
    return z

Z = functz(V)
ax = plt.figure().add_subplot(projection='3d')
ax.plot_wireframe(X, Y, Z, color = 'r')          # Red wireframe
ax.set_xlabel('X')
ax.set_ylabel('Y')
ax.set_zlabel('Potential')
ax.view_init(30, 140)                            #Perspective
plt.show()                                       # Show fig
```

Partial Differential Equations

Finite Difference Methods: 2D Poisson Equation (Dirichlet)



$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = 0$$

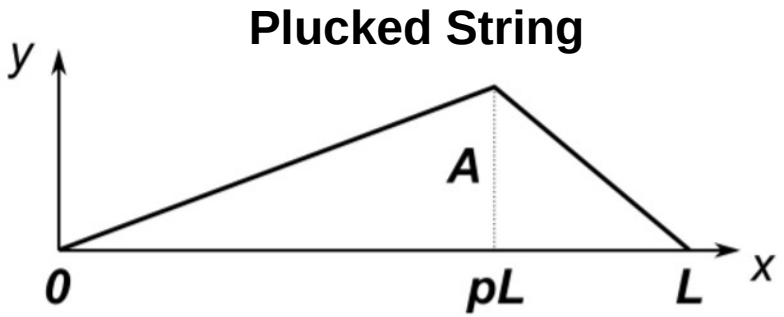


See Jupyter notebook: “Special_Topics.ipynb”

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

plt.rcParams['figure.figsize'] = (12,8)
plt.rcParams['font.size'] = 20


Nmax = 100
V_wire = 100

V = np.zeros((Nmax, Nmax))
V[:,0] = V_wire                                  # Line at V_wire volts

print ("Running the Jacobi Update Solver ...")

def jacobi_update(V, Niter=70):
    Nmax = V.shape[0]
    for iter in range(Niter):
        for i in range(1, Nmax-2):
            for j in range(1,Nmax-2):
                V[i,j] = 0.25*(V[i+1,j]+V[i-1,j]+V[i,j+1]+V[i,j-1])
    return V

V = jacobi_update(V)
x = np.arange(0, Nmax-1, 2);  y = np.arange(0, 50, 2)
X, Y = np.meshgrid(x,y)

def functz(V):                                   # V(x, y)
    z = V[X,Y]
    return z

Z = functz(V)
ax = plt.figure().add_subplot(projection='3d')
ax.plot_wireframe(X, Y, Z, color = 'r')          # Red wireframe
ax.set_xlabel('X')
ax.set_ylabel('Y')
ax.set_zlabel('Potential')
ax.view_init(30, 140)                            #Perspective
plt.show()                                       # Show fig
```

Finite Difference Methods: 1D Wave Equation (Dirichlet)

$$\frac{\partial^2 u}{\partial x^2} = \frac{1}{c^2}\frac{\partial^2 u}{\partial t^2}$$



Time-Stepping (Leapfrog Method):

1) Divide the space-time domain into a discrete lattice

2) Replace the partial derivatives by finite difference approximations evaluated at the lattice points

3) Reduce the PDE to a system of linear algebra equations, with initial/boundary conditions added as extra equations

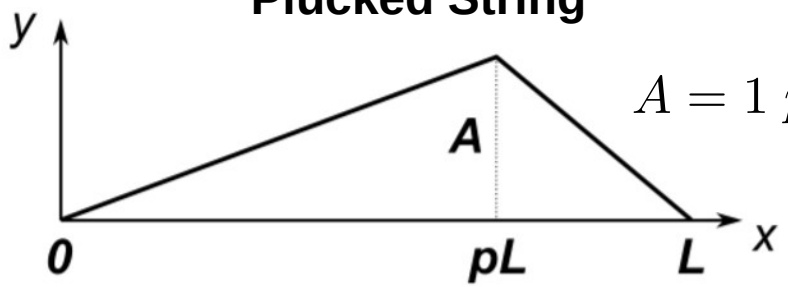
4) Jacobi-update one time-step at a time, updating space-steps from the previous time

5) Repeat for all times

Finite Difference Methods: 1D Wave Equation (Dirichlet)

$$\frac{\partial^2 u}{\partial x^2} = \frac{1}{c^2}\frac{\partial^2 u}{\partial t^2}$$

Plucked String



$A = 1 \; p = 0.8$

Time-Stepping (Leapfrog Method):

$$y(x, t=0) = \begin{cases} 1.25x/L\,, & x \le 0.8L\,, \\ (5 - 5x/L)\,, & x > 0.8L\,, \end{cases}$$

$$\frac{\partial y}{\partial t}(x, t=0) = 0$$

1) Divide the space-time domain into a discrete lattice

2) Replace the partial derivatives by finite difference approximations evaluated at the lattice points

3) Reduce the PDE to a system of linear algebra equations, with initial/boundary conditions added as extra equations

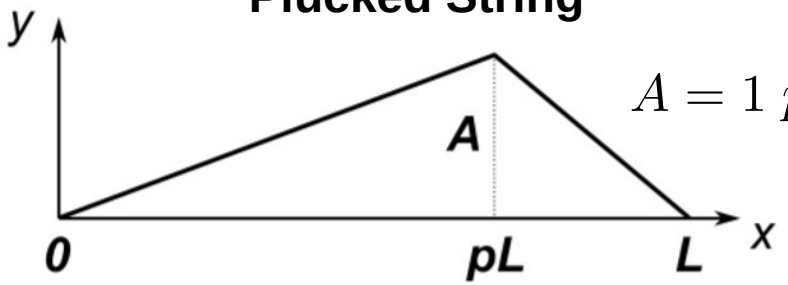
4) Jacobi-update one time-step at a time, updating space-steps from the previous time

5) Repeat for all times

Finite Difference Methods: 1D Wave Equation (Dirichlet)

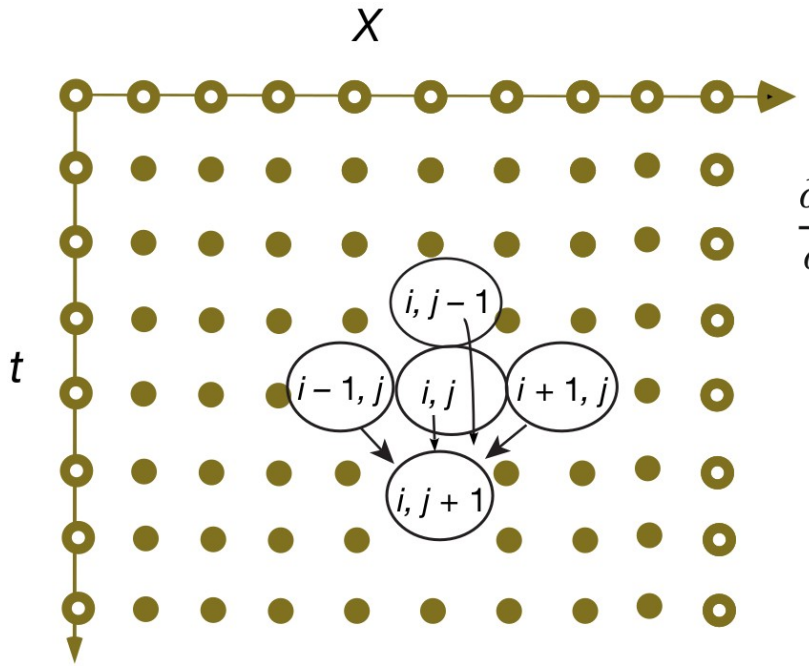
$$\frac{\partial^2 u}{\partial x^2} = \frac{1}{c^2}\frac{\partial^2 u}{\partial t^2}$$

Plucked String



$$A = 1 \; p = 0.8$$

Time-Stepping (Leapfrog Method):



$$y(x, t=0) = \begin{cases} 1.25x/L\,, & x \le 0.8L\,, \\ (5 - 5x/L)\,, & x > 0.8L\,, \end{cases}$$

$$\frac{\partial y}{\partial t}(x, t=0) = 0$$

$$\frac{\partial^2 y}{\partial t^2} \simeq \frac{y_{i,j+1} + y_{i,j-1} - 2y_{i,j}}{(\Delta t)^2}\,, \quad \frac{\partial^2 y}{\partial x^2} \simeq \frac{y_{i+1,j} + y_{i-1,j} - 2y_{i,j}}{(\Delta x)^2}$$

$$\frac{y_{i,j+1} + y_{i,j-1} - 2y_{i,j}}{c^2(\Delta t)^2} = \frac{y_{i+1,j} + y_{i-1,j} - 2y_{i,j}}{(\Delta x)^2}$$

Contains three time values:

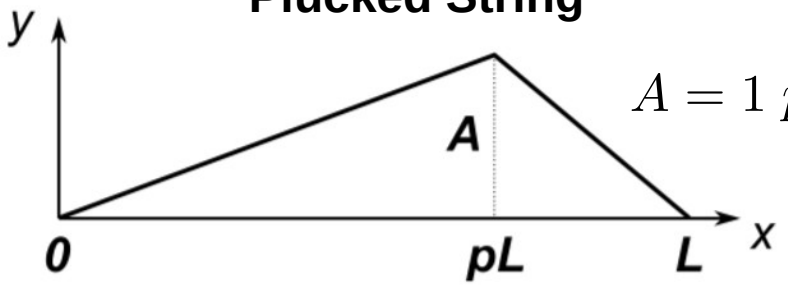
- j + 1 = the future,
- j = the present,
- j − 1 = the past

Predict the future solution from the present and past solutions

Finite Difference Methods: 1D Wave Equation (Dirichlet)

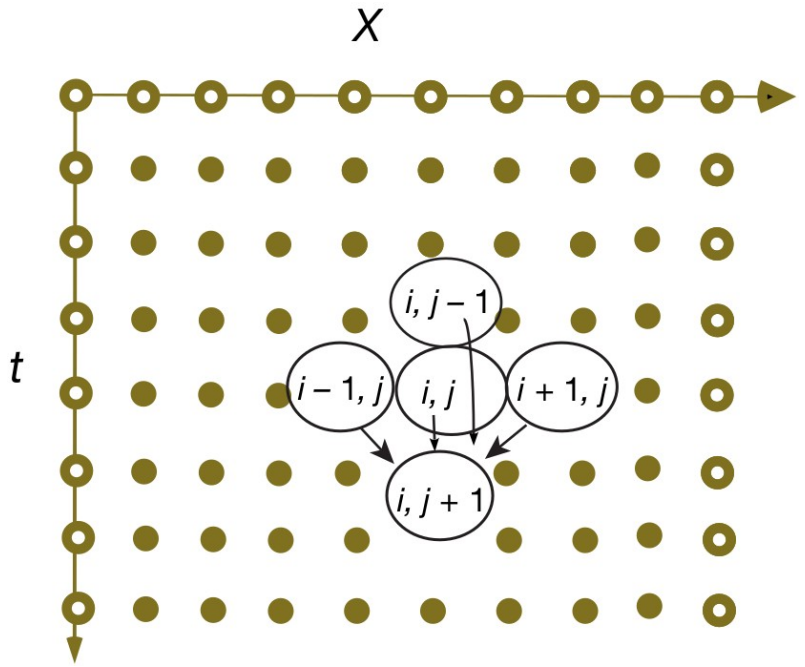
$$\frac{\partial^2 u}{\partial x^2} = \frac{1}{c^2}\frac{\partial^2 u}{\partial t^2}$$

Plucked String



$$A = 1 \; p = 0.8$$

Time-Stepping (Leapfrog Method):



$$y(x, t=0) = \begin{cases} 1.25x/L\,, & x \le 0.8L\,, \\ (5 - 5x/L)\,, & x > 0.8L\,, \end{cases}$$

$$\frac{\partial y}{\partial t}(x, t=0) = 0$$

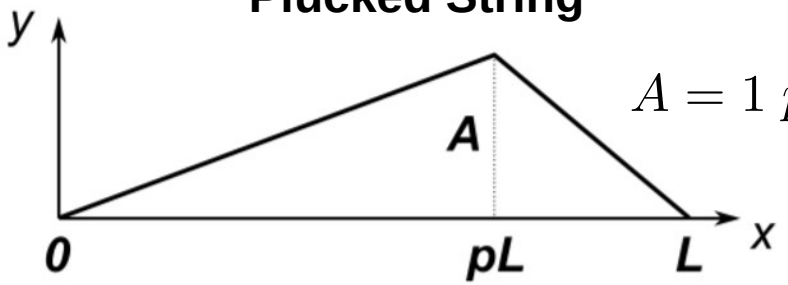
$$\frac{y_{i,j+1} + y_{i,j-1} - 2y_{i,j}}{c^2(\Delta t)^2} = \frac{y_{i+1,j} + y_{i-1,j} - 2y_{i,j}}{(\Delta x)^2}$$

$$y_{i,j+1} = 2y_{i,j} - y_{i,j-1} + \frac{c^2}{c'^2}\left[y_{i+1,j} + y_{i-1,j} - 2y_{i,j}\right]\,, \quad c' \overset{\text{def}}{=} \frac{\Delta x}{\Delta t}$$

Finite Difference Methods: 1D Wave Equation (Dirichlet)

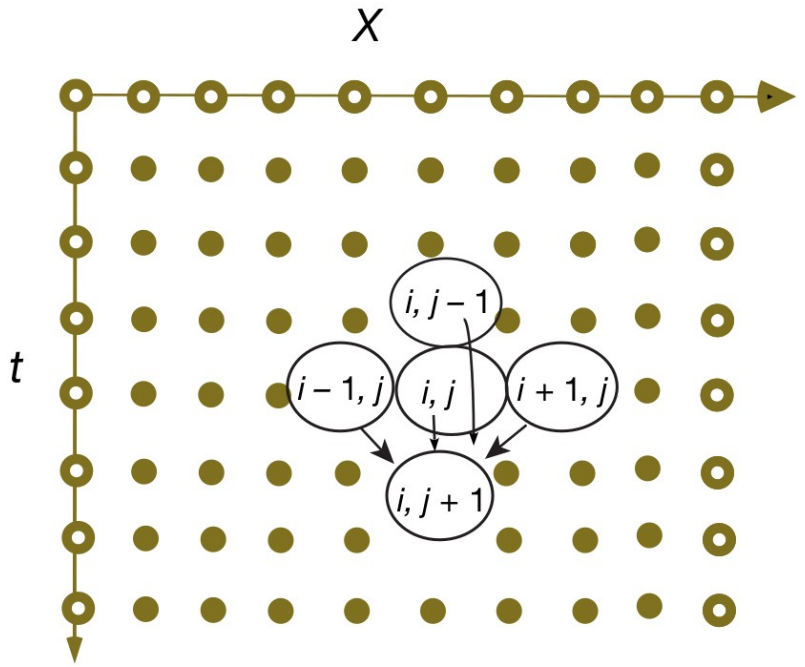
$$\frac{\partial^2 u}{\partial x^2} = \frac{1}{c^2}\frac{\partial^2 u}{\partial t^2}$$

Plucked String



$$A = 1 \; p = 0.8$$

Time-Stepping (Leapfrog Method):



$$y(x, t=0) = \begin{cases} 1.25x/L\,, & x \le 0.8L\,, \\ (5 - 5x/L)\,, & x > 0.8L\,, \end{cases}$$

$$\frac{\partial y}{\partial t}(x, t=0) = 0$$

$$y_{i,j+1} = 2y_{i,j} - y_{i,j-1} + \frac{c^2}{c'^2}\left[y_{i+1,j} + y_{i-1,j} - 2y_{i,j}\right]\,, \quad c' \overset{\text{def}}{=} \frac{\Delta x}{\Delta t}$$

Von-Neumann Stability Analysis
Ensures that the eigenfuncs don't blow up

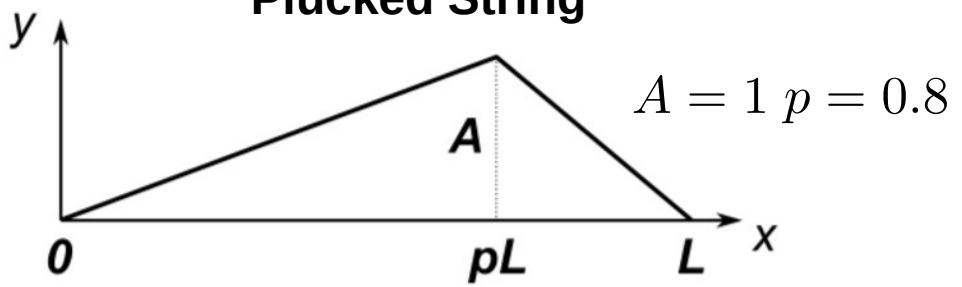
$$c \le c' = \frac{\Delta x}{\Delta t}$$ **Courant condition**

Solution gets better with smaller time steps
But gets worse for smaller space steps

Finite Difference Methods: 1D Wave Equation (Dirichlet)

$$\frac{\partial^2 u}{\partial x^2} = \frac{1}{c^2}\frac{\partial^2 u}{\partial t^2}$$

Plucked String



$A = 1 \; p = 0.8$

Time-Stepping (Leapfrog Method):

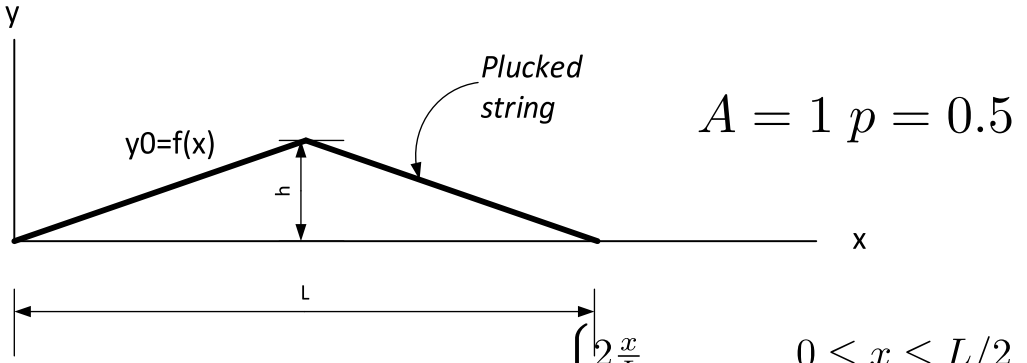
$$y(x, t=0) = \begin{cases} 1.25x/L\,, & x \le 0.8L\,, \\ (5 - 5x/L)\,, & x > 0.8L\,, \end{cases}$$

Homework: See if you can change the code to pluck the string in the middle

See Jupyter notebook: “Special_Topics.ipynb”

Finite Difference Methods: 1D Wave Equation (Dirichlet)

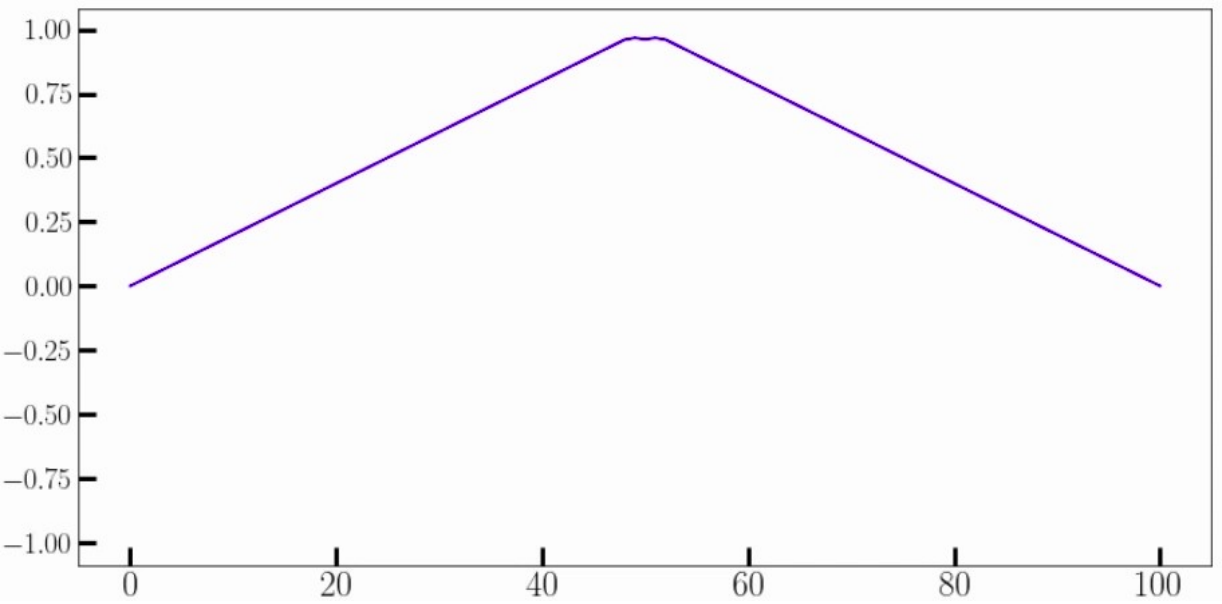
$$\frac{\partial^2 u}{\partial x^2} = \frac{1}{c^2}\frac{\partial^2 u}{\partial t^2}$$



$$A = 1 \; p = 0.5$$

Time-Stepping (Leapfrog Method):

$$y(x, t=0) = \begin{cases} 2\frac{x}{L} & 0 \le x \le L/2 \\ 2\left(1 - \frac{x}{L}\right) & L/2 \le x \le L \end{cases}$$



Homework: See if you can change the code to pluck the string in the middle

See Jupyter notebook: “Special_Topics.ipynb”

Finite Element Methods: 2D Poisson Equation (Dirichlet)

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = -4\pi\rho(x, y)$$

1) Reduce the PDE to an integral equation using Gauss/Stoke's theorem on the boundary

2) Divide the domain into a discrete mesh

3) Choose basis functions, usually polynomials, corresponding spatially to each mesh element

4) Write the RHS as a (known) linear superposition of these basis functions, and the LHS as a superposition with unknown coefficients

5) Substitute into the Integral equation and reduce it to a system of linear equations

6) Solve using BLAS or whatever.

Finite Element Methods: 2D Poisson Equation (Dirichlet)

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = -4\pi\rho(x, y)$$

1) Reduce the PDE to an integral equation using Gauss/Stoke's theorem on the boundary

2) Divide the domain into a discrete mesh

3) Choose basis functions, usually polynomials, corresponding spatially to each mesh element

4) Write the RHS as a (known) linear superposition of these basis functions, and the LHS as a superposition with unknown coefficients

5) Substitute into the Integral equation and reduce it to a system of linear equations

6) Solve using BLAS or whatever.