Федеральное государственное автономное образовательное учреждение высшего образования «Национальный Исследовательский Университет ИТМО»

ЛАБОРАТОРНАЯ РАБОТА №5
ПРЕДМЕТ «ЧАСТОТНЫЕ МЕТОДЫ»
ТЕМА «СВЯЗЬ НЕПРЕРЫВНОГО И ДИСКРЕТНОГО»

Лектор: Перегудин А. А.
Практик: Пашенко А. В.
Студент: Румянцев А. А.
Поток: ЧАСТ.МЕТ. 1.3

Факультет: СУиР
Группа: R3241

Санкт-Петербург
2024

Содержание

1 Задание 1. Непрерывное и дискретное преобразование Фурье

Рассмотрим прямоугольную функцию $\Pi : \mathbb{R} \to \mathbb{R}$:

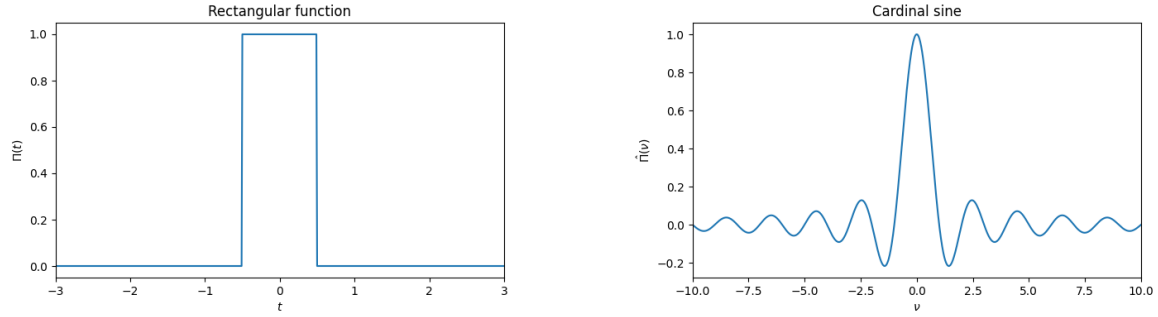
$$\Pi(t) = \begin{cases} 1, & |t| \leq 1/2, \\ 0, & |t| > 1/2. \end{cases}$$

1.1 Истинный Фурье-образ

Найдем аналитическое выражение для Фурье-образа прямоугольной функции

$$\hat{\Pi}(\nu) = \int_{-\infty}^{+\infty} \Pi(t) e^{-2\pi i \nu t}\, dt = \int_{-\frac{1}{2}}^{\frac{1}{2}} e^{-2\pi i \nu t}\, dt = -\frac{e^{-\pi i \nu} - e^{\pi i \nu}}{2\pi i \nu} = \frac{\sin(\pi\nu)}{\pi\nu} = \operatorname{sinc}(\nu)$$

Построим графики $\Pi(t)$ и $\hat{\Pi}(\nu)$



(a) Прямоугольная функция (b) Кардинальный синус

Рис. 1: Исходный сигнал и его Фурье-образ

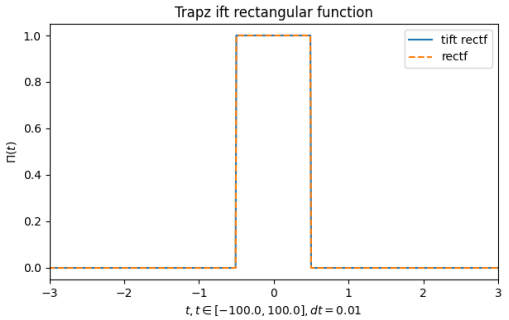
1.2 Численное интегрирование

Зададим функцию $\Pi(t)$ в `Python`. Найдем ее Фурье-образ с помощью численного интегрирования (функция `trapz`). Вновь используя численное интегрирование, выполним обратное преобразование Фурье от найденного Фурье-образа с целью восстановить исходную функцию. Схематично наши действия будут выглядеть так:

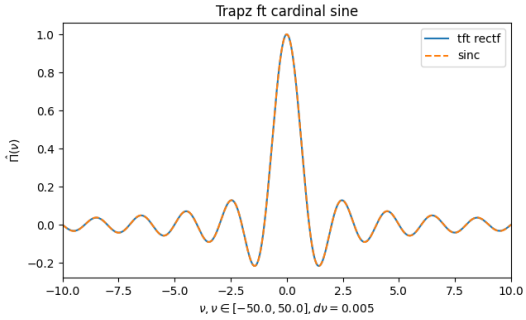
$$\Pi(t) \xrightarrow{\texttt{trapz}} \hat{\Pi}(\nu) \xrightarrow{\texttt{trapz}} \Pi(t)$$

Построим график найденной функции $\hat{\Pi}(\nu)$ и *восстановленной* функции $\Pi(t)$. Сравним результат с истинной функцией и Фурье-образом. Исследуем влияние величины шага интегрирования и размера промежутка, по которому вычисляется интеграл, на результат. Сделаем выводы о точности и быстродействии метода.

Далее приведены соответствующие графики. Оранжевым цветом выделены оригинальные функции, синим – найденные через преобразования. Каждый график подписан сверху. Под временной шкалой также указаны рассматриваемый промежуток времени или частот и шаг дискретизации во временной или частотной областях.

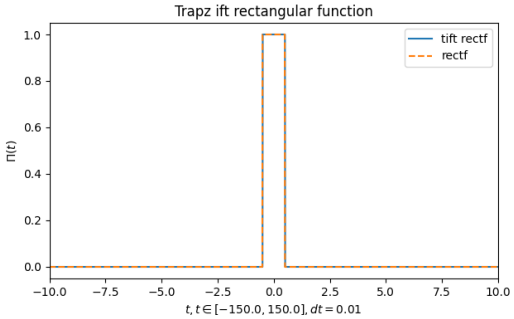


(a) $\Pi(t)$, восстановленная `trapz`

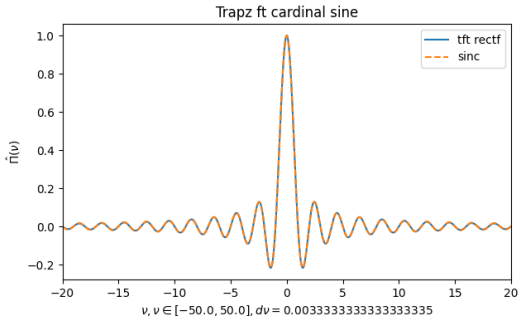


(b) $\hat{\Pi}(t)$ восстановленной `trapz` $\Pi(t)$

Рис. 2: Интеграл по всей области определения функции от -100 до 100

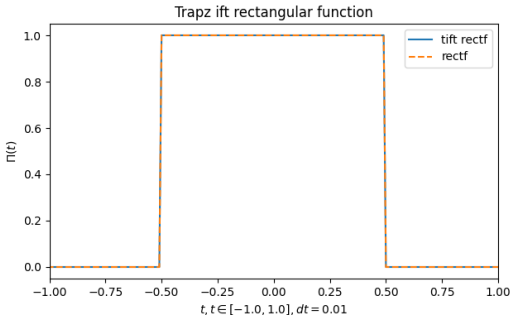


(a) $\Pi(t)$, восстановленная `trapz`

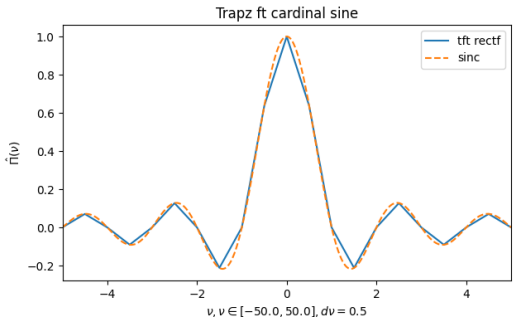


(b) $\hat{\Pi}(t)$ восстановленной `trapz` $\Pi(t)$

Рис. 3: Интеграл на увеличенном промежутке от -150 до 150

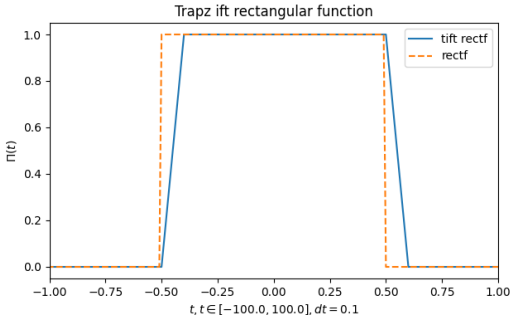


(a) $\Pi(t)$, восстановленная `trapz`

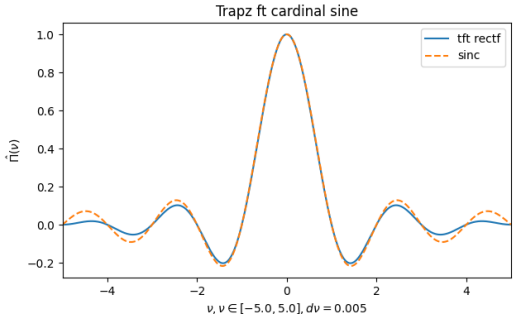


(b) $\hat{\Pi}(t)$ восстановленной `trapz` $\Pi(t)$

Рис. 4: Интеграл на уменьшенном промежутке от -1 до 1

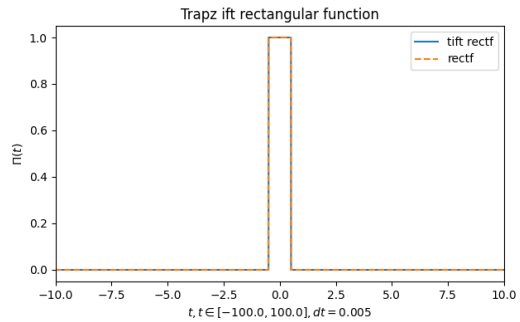


(a) $\Pi(t)$, восстановленная `trapz`

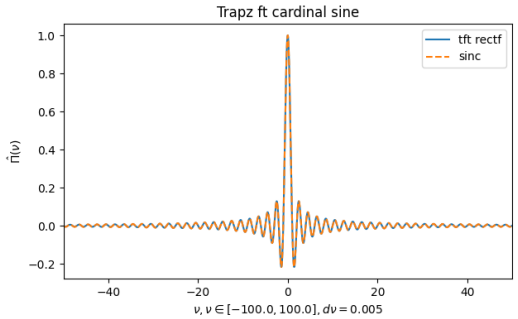


(b) $\hat{\Pi}(t)$ восстановленной `trapz` $\Pi(t)$

Рис. 5: Увеличение шага интегрирования $dt = 0.1$, интеграл аналогично рис. 2



(a) $\Pi(t)$, восстановленная `trapz`



(b) $\hat{\Pi}(t)$ восстановленной `trapz` $\Pi(t)$

Рис. 6: Уменьшение шага интегрирования $dt = 0.005$, интеграл аналогично рис. 2

Исходя из графиков можно сделать вывод, что `trapz` имеет высокую точность аппроксимации исходного сигнала и его Фурье-образа, однако ее мы достигаем при маленьком шаге dt и большом промежутке T (см. рис. 2, 3, 6), что увеличивает вычислительную сложность метода. В итоге приходится долго ждать получения результата. При уменьшении промежутка интегрирования `trapz` выдает точный результат прямоугольной функции (см. рис. 4), однако Фурье-образ становится негладким. При большом шаге интегрирования подсчеты происходят быстро, но теряется точность функций во временной и частотных областях (см. рис. 5). Таким образом, метод точный, но затрачивает много времени на вычисления.

1.3 Использование DFT

Найдем Фурье-образ функции $\Pi(t)$ с помощью дискретного преобразования Фурье (конструкция `fftshift(fft())`), используя его так, чтобы преобразование было унитарным. Выполним обратное преобразование от найденного Фурье-образа с помощью обратного дискретного преобразования (конструкция `ifft(ifftshift())`). Схематично наши действия можно представить так:

$$\Pi(t) \xrightarrow{\texttt{fftshift(fft())}} \hat{\Pi}(\nu) \xrightarrow{\texttt{ifft(ifftshift())}} \Pi(t)$$

Для того, чтобы преобразование было унитарным, необходимо домножить ряд дискретного преобразования Фурье на коэффициент

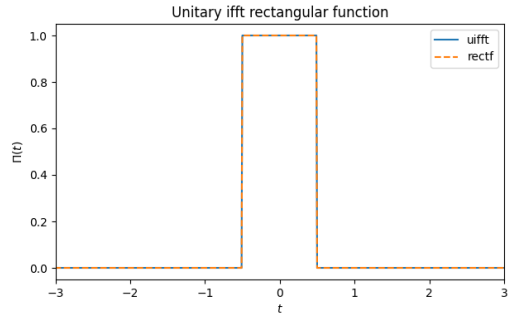
$$\frac{1}{\sqrt{N}}, \quad N \text{ – количество членов ряда.}$$

Аналогично для обратного преобразования Фурье. Таким образом, формулы DFT и IDFT будут иметь вид:

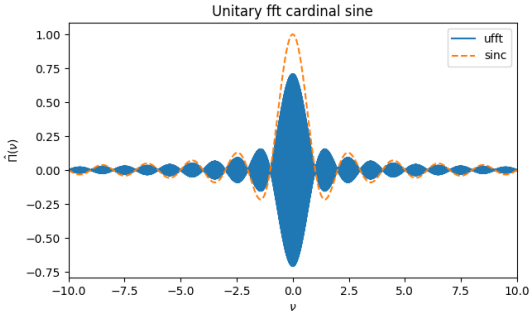
$$\mathcal{F}_m = \frac{1}{\sqrt{N}} \sum_{n=0}^{N-1} f_n e^{-2\pi i \frac{mn}{N}}, \quad f_n = \frac{1}{\sqrt{N}} \sum_{m=0}^{N-1} \mathcal{F}_m e^{2\pi i \frac{mn}{N}}$$

Далее приведены сравнительные графики найденной $\hat{\Pi}(\nu)$ и *восстановленной* $\Pi(t)$ функций с исходными. Цвета и обозначения аналогичны предыдущему пункту.

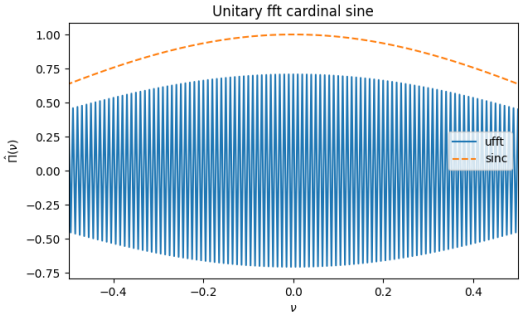
Исходя из графиков можно сделать вывод, что унитарное быстрое преобразование Фурье точно восстанавливает исходный сигнал, затрачивая малое количество времени на вычисления. Однако Фурье-образ не совпадает с истинным (кардинальным синусом). Функция приобрела лишние амплитуды на всей частотной области (см. рис. 7).



(a) $\Pi(t)$, восстановленная `ufft`



(b) $\hat{\Pi}(t)$ восстановленной `ufft` $\Pi(t)$



(c) Рис. (b) в приближении

Рис. 7: Унитарное быстрое преобразование Фурье `ufft`

1.4 Выводы о trapz и fft

Метод `trapz` работает медленно, но точно – он смог приблизиться к истинному Фурье-образу прямоугольной функции. `trapz` выполняет численное интегрирование с помощью трапециевидного метода – он аппроксимирует интегрирование на интервале путем разламывания области на трапецоиды с более легко вычислимыми областями. Для интеграции с $N+1$ равномерно распределенными точками приближение будет иметь вид:

$$\int_a^b f(x)\,dx \approx \frac{b-a}{2N}\sum_{n=1}^{N}\left(f(x_n)+f(x_{n+1})\right), \quad \frac{b-a}{N} \text{ – интервал между каждой точкой.}$$

К выражению выше добавляется умножение на экспоненту, в степени которой находится переменная, по которой вычисляется интеграл. Обратное преобразование находится аналогично.

Метод `fft` работает быстро, но Фурье-образ не похож на кардинальный синус. В предыдущем пункте были приведены формулы, которые используются для дискретного преобразования Фурье – через них работает быстрое преобразование Фурье. `fft` основывается на «симметриях» корней из единицы матрицы `dft`, которую можно составить из элементов одноименного ряда. Эта матрица квадратная и обратимая, иначе мы не смогли бы найти `idft`. Более того, она унитарная – обратная матрица является транспонированной комплексно-сопряженной – воздействует как «поворот». Симметрия корней из 1 связана с тем, что значения $\omega_N^k = e^{2\pi ik \div N}$ делятся на две группы: те, у которых k четное, и те, у которых k нечетное. При этом $\omega_N^{N \div 2}$ является корнем из -1.

Фурье-образ получается лучше у метода трапеций, так как он проходит по каждому значению времени и вычисляет в приближении сложный интеграл. Однако у `fft` по времени лучше получается восстановленный сигнал несмотря на то, что Фурье-образ не

совпадает с истинным. Причина в тех симметриях, о которых написано в абзаце выше – алгоритмы `fft` и `ifft` взаимно обратны. Ошибки округления и другие численные погрешности, возникающие при прямом преобразовании, компенсируются при обратном преобразовании.

1.5 Приближение непрерывного с помощью DFT

Чтобы исправить ситуацию, попробуем совместить достоинства обоих подходов: точность и быстродействие. Найдем способ получить правильный Фурье-образ, соответствующий непрерывному преобразованию Фурье, используя функцию `fft` и не прибегая к численному интегрированию. Найдем способ восстановить исходный сигнал по полученному Фурье-образу – тоже с помощью `fft`. Схема нашего успеха:

$$\Pi(t) \xrightarrow{\text{умное использование } \texttt{fft}} \hat{\Pi}(\nu) \xrightarrow{\text{умное использование } \texttt{ifft}} \Pi(t)$$

Ранее мы выяснили, что метод трапеций работает наиболее точно, но он ресурсозатратен. Вспомним вид интеграла Фурье для ограниченной по времени функции:

$$\mathcal{F}(\nu) = \int_{-t_0}^{t_0} f(t) e^{-2\pi i \nu t}\, dt$$

Аппроксимируем интеграл Фурье с помощью суммы Римана – бесконечного числа маленьких прямоугольников, аппроксимирующих кривую в интеграле. Представим $t = n \cdot \Delta t + t_0$ – время на n-ом шаге с масштабированием Δt и некоторым положением относительно t_0.

$$\mathcal{F}(\nu) \approx \sum_{n=0}^{N-1} f(n \cdot \Delta t + t_0) e^{-2\pi i \nu (n \cdot \Delta t + t_0)}\, \Delta t$$

Разделив временной диапазон на небольшие интервалы и задав постоянное значение сигнала для каждого интервала, мы, по сути, просто выполняем временную дискретизацию сигнала. Упростим выражение, собрав как можно больше слагаемых, не зависящих от n:

$$\mathcal{F}(\nu) \approx \Delta t \sum_{n=0}^{N-1} f(n \cdot \Delta t + t_0) e^{-2\pi i \nu n \Delta t} e^{-2\pi i \nu t_0}$$

Так как Δt и t_0 – постоянные, обозначим:

$$f[n] = f(n \cdot \Delta t + t_0)$$

Частоты ν нам известны. Оставим их как есть для константы экспоненты вне ряда, а для экспоненты со степенью, содержащей n, выразим их как

$$\nu_k = k \cdot \Delta\nu = \frac{k}{N \cdot \Delta t}$$

Δt в экспоненте сократятся, получим:

$$\mathcal{F}\left(\nu_k = \frac{k}{N \cdot \Delta t}\right) \approx \Delta t\, e^{-2\pi i \nu t_0} \sum_{n=0}^{N-1} f[n] e^{-2\pi i \frac{kn}{N}}$$

Формула дискретного преобразования Фурье может быть представлена как

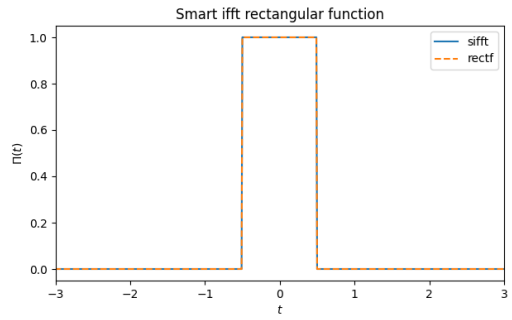
$$\mathcal{F}[k] = \sum_{n=0}^{N-1} f[n] e^{-2\pi i \frac{kn}{N}},$$

подставляя ее в формулу $\mathcal{F}(\nu)$ получим

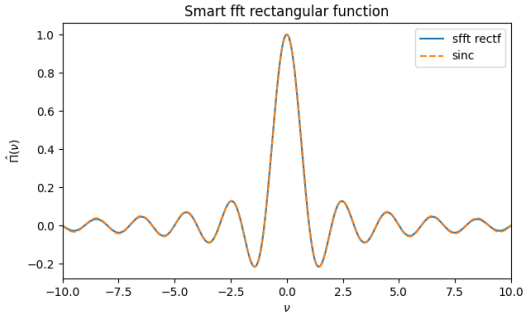
$$\mathcal{F}\left(\nu_k = \frac{k}{N \cdot \Delta t}\right) \approx \Delta t \, e^{-2\pi i \nu t_0} \mathcal{F}[k],$$

где $\mathcal{F}[k]$ – дискретное преобразование Фурье, которое можно заменить на быстрое преобразование Фурье. То есть результат `fft` необходимо домножить на $\Delta t \, e^{-2\pi i \nu t_0}$, чтобы получить Фурье-образ, соответствующий эталону. Для обратного преобразования перед применением `ifft` просто разделим умный `fft` на то же слагаемое, так как оно является константой, а `ifft` в точности быстро восстанавливает исходный сигнал без дополнительных преобразований `fft`. Кратко будем обозначать такое преобразование как `sfft` и `sifft`.

Далее приведены сравнительные графики исходного сигнала и его Фурье-образа с результатами после умного применения `fft` и `ifft`.



(a) $\Pi(t)$, восстановленная `sfft`



(b) $\hat{\Pi}(t)$ восстановленной `sfft` $\Pi(t)$

Рис. 8: Умное использование быстрого преобразования Фурье `sfft` и `sifft`

1.6 Используемые программы

Далее всегда будут использоваться язык программирования `Python` и библиотеки `numpy` и `matplotlib`.

Для удобного построения графиков с точной настройкой я написал программу, представленную на листинге ниже.

```
    # Method to build 1 function
    def showf(x,
              y,
              xlim=(None, None),
              ylim=(None, None),
              title=None,
              label=None,
              xlabel=None,
              ylabel=None,
              legend=False,
              grid=False,
              linest='-',
              color=None,
```

```
              figsize_x=7,
              figsize_y=4):
        plt.plot(x, y, label=label, color=color, linestyle=linest)
        plt.xlim(xlim)
        plt.ylim(ylim)
        plt.xlabel(xlabel)
        plt.ylabel(ylabel)
        plt.title(title)
        plt.grid(grid)
        if legend:
            plt.legend()
        plt.gcf().set_size_inches(figsize_x, figsize_y)
        plt.show()

    # Needed if we want to build more than 1 function
    def showfs(x,
               y: list,
               xlim=(None, None),
               ylim=(None, None),
               title=None,
               labels: list = None,
               xlabel=None,
               ylabel=None,
               legend=False,
               grid=False,
               linest: list = None,
               colors: list = None,
               figsize_x=7,
               figsize_y=4):
        if (labels is None):
            labels = [None] * len(y)
        if (linest is None):
            linest = ['-'] * len(y)
        if (colors is None):
            colors = [None] * len(y)
        if isinstance(x, list):
            for k in range(len(y)):
                plt.plot(x[k],
                         y[k],
                         label=labels[k],
                         linestyle=linest[k],
                         color=colors[k])
        else:
            for k in range(len(y)):
                plt.plot(x,
                         y[k],
                         label=labels[k],
                         linestyle=linest[k],
                         color=colors[k])
        plt.xlim(xlim)
        plt.ylim(ylim)
        plt.xlabel(xlabel)
        plt.ylabel(ylabel)
        plt.title(title)
        plt.grid(grid)
        if legend:
            plt.legend()
        plt.gcf().set_size_inches(figsize_x, figsize_y)
        plt.show()
```

Листинг 1: Методы для построения графиков

```
# Usage example

# Building rectangular function
sh.showf(t,
         rectf_,
         xlim=(-3, 3),
         title=(name1[0].upper()) + name1[1:],
         xlabel=rf'$t$',
         ylabel=r'$\Pi(t)$')

# Building comparison graph: trapz inverse Ft and rectangular function
sh.showfs([new_t, t], [tift_.real, rectf_],
          xlim=(-3, 3),
          title=f'Trapz ift {name1}',
          linest=['-', '--'],
          labels=['tift rectf', 'rectf'],
          xlabel=rf'$t,t\in{[new_t[0],round(new_t[-1],2)]},dt={new_dt}$',
          ylabel=r'$\Pi(t)$',
          legend=True)
```

Листинг 2: Пример использования методов для построения графиков

Вспомогательные функции.

```
# Rectangular function
def rectf(t):
    return np.where((-0.5 <= t) & (t <= 0.5), 1, 0)

# Cardinal sine
def sinc(v):
    return np.where(v == 0, 1, np.sin(np.pi * v) / (np.pi * v))
```

Листинг 3: Методы для нахождения прямоугольной функции и кардинального синуса

Основные математические операции.

```
# trapz Fourier transform
def tft(f, t, v):
    F = []
    for k in v:
        F_k = np.trapz(f * np.exp(-2j * np.pi * k * t), t)
        F.append(F_k)
    return np.array(F)

# trapz inverse Fourier transform
def tift(F, t, v):
    f = []
    for k in t:
        f_k = np.trapz(F * np.exp(2j * np.pi * k * v), v)
        f.append(f_k)
    return np.array(f)

# DFT. For smart FFT special coeff is needed
def dft(f, norm=None, coeff=1):
    fft_ = coeff * np.fft.fftshift(np.fft.fft(f, norm=norm))
    ifft_ = np.fft.ifft(np.fft.ifftshift(fft_ / coeff), norm=norm)
    return fft_, ifft_
```

Листинг 4: Методы, реализующие `trapz`, `dft` и умный `fft`

Использование предыдущих наработок для выполнения задания 1.

```
# Set the t array and calculate rectangular function
T = 200
dt = 0.01
t = np.arange(-T / 2, T / 2 + dt, dt)
rectf_ = hp.rectf(t)
name1 = 'rectangular function'

# Set the v array and calculate cardinal sine
V = 1 / dt
dv = 1 / T
v = np.arange(-V / 2, V / 2 + dv, dv)
sinc_ = hp.sinc(v)
name2 = 'cardinal sine'

# Adjusting new t array to integrate on this interval
new_T = 200
new_dt = 0.01
new_t = np.arange(-new_T / 2, new_T / 2 + new_dt, new_dt)
new_rectf = hp.rectf(new_t)

# Adjusting new v array to integrate on this interval
new_V = 1 / new_dt
new_dv = 1 / new_T
new_v = np.arange(-new_V / 2, new_V / 2 + new_dv, new_dv)

# Calculating trapz
# Firstly direct transform, secondly inverse transform
tft_ = fm.tft(new_rectf, new_t, new_v)
tift_ = fm.tift(tft_, new_t, new_v)

# Calculating unitary fast Fourier transform
ufft, uifft = fm.dft(rectf_, norm='ortho')

# Finding special coefficient for smart FFT, then calculating smart FFT
c = dt * np.exp(-2 * np.pi * 1j * v * t[0])
sfft, sifft = fm.dft(rectf_, coeff=c)

# Build graphs using showf and showfs methods as in the example
```

Листинг 5: Реализация задания 1

2 Задание 2. Сэмплирование

В этом задании будем исследовать теорему Найквиста-Шеннона-Котельникова на двух примерах.

2.1 Сэмплирование синусов

Зададим параметры $a_1 = 1$, $a_2 = 2$, $\omega_1 = 3$, $\omega_2 = 4$, $\varphi_1 = \pi \div 5$, $\varphi_2 = \pi \div 6$, $B = 7$ и рассмотрим функцию

$$y(t) = a_1 \sin(\omega_1 t + \varphi_1) + a_2 \sin(\omega_2 t + \varphi_2)$$

Зададим в `Python` массивы времени `t` и значений `y`. Выберем малый шаг дискретизации $dt = 10^{-4}$ для имитирования непрерывной функции. Построим график полученной функции.

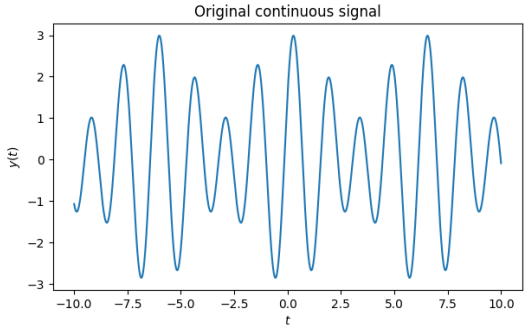


Рис. 9: График непрерывного сигнала

Теперь зададим сэмплированный вариант указанной функции: рассмотрим разреженный вариант массива времени и соответствующий ему массив значений. Сначала рассмотрим большой шаг $dt = 0.2$. Построим дискретный график поверх непрерывного.

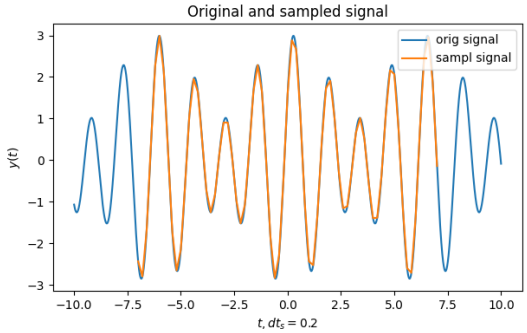


Рис. 10: График сэмплированного сигнала поверх непрерывного

Применим интерполяционную формулу из лекции к сэмплированным данным с целью восстановить непрерывную функцию.

$$f(t) = \sum_{n=-\infty}^{\infty} f(t_n) \cdot \operatorname{sinc}\left(2B(t - t_n)\right), \quad t_n = \frac{n}{2B}$$

В результате получатся новые массивы времени и значений – той же размерности, что и исходные.

Построим график восстановленной функции поверх исходной.

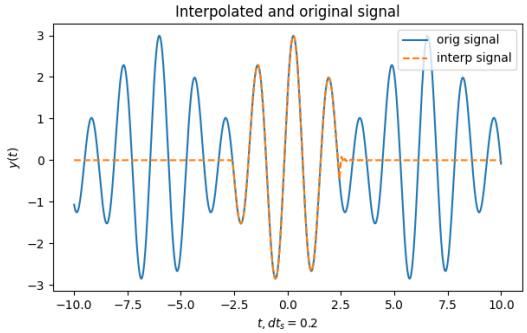
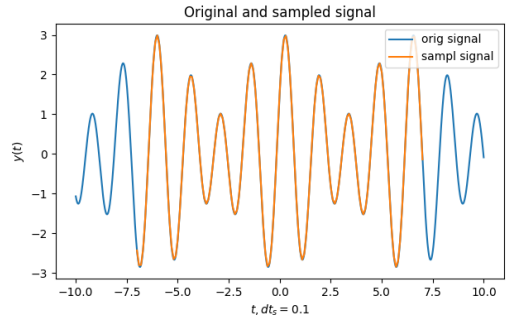
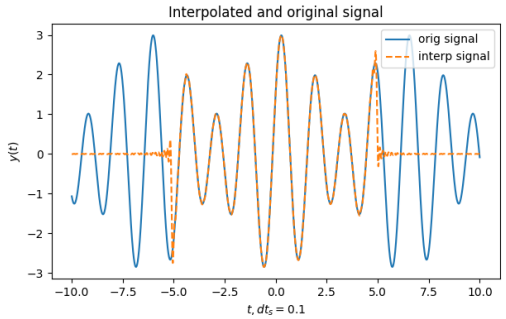


Рис. 11: График интерполированного сигнала поверх непрерывного

Далее исследуем влияние шага дискретизации на вид восстановленной функции и соотнесем результаты с теоремой Найквиста-Шеннона-Котельникова. Синим цветом выделены исходные сигналы, оранжевым – сэмплированные и восстановленные. Под шкалой времени на каждом графике указан шаг дискретизации сэмплирования dt_s.

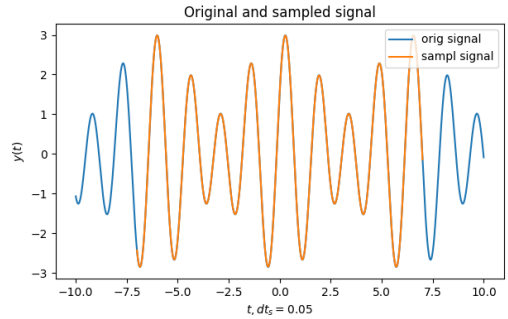


(a) Сэмплированный поверх непр-ого

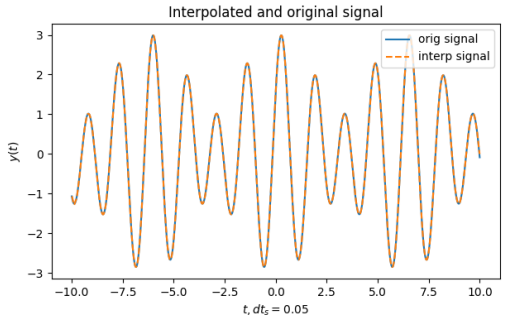


(b) Интерполированный поверх непр-ого

Рис. 12: Уменьшение шага дискретизации $dt = 0.1$



(a) Сэмплированный поверх непр-ого



(b) Интерполированный поверх непр-ого

Рис. 13: Уменьшение шага дискретизации $dt = 0.05$

Теорема Найквиста-Шеннона-Котельникова гласит: если образ Фурье исходной функции содержится внутри отрезка $[-B, B]$, то функция может быть безошибочно восстановлена по дискретным данным, если для шага дискретизации Δt выполнено условие

$$\Delta t < \frac{1}{2B}$$

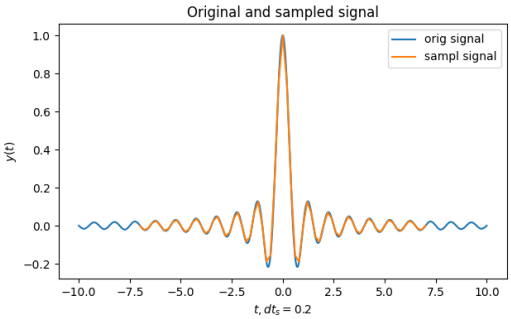
В данном случае $B = 7$, значит при $\Delta t < 1 \div 14 \approx 0.071$ теорема должна выполняться. Видим, что на рис. 11 и 12 интерполированный сигнал не совпал с исходным при $\Delta t > 0.071$. На рис. 13 результат в точности повторяет изначальный сигнал при $\Delta t = 0.05 < 0.071$, следовательно теорема выполняется.

2.2 Сэмплирование sinus cardinalis

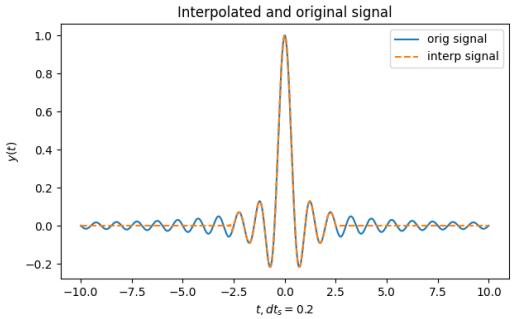
Зададим параметр $b = 2$ и рассмотрим функцию

$$y(t) = \operatorname{sinc}(bt) = \operatorname{sinc}(2t)$$

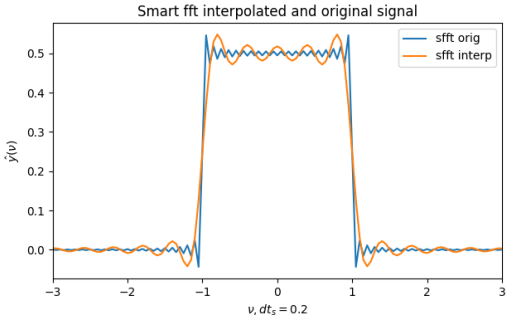
Оставим $B = 7$. Выполним все шаги из предыдущего пункта. Дополнительно для каждой величины шага дискретизации построим Фурье-образ исходного и восстановленного сигналов. Дадим объяснение увиденному, сделаем выводы.



(a) Сэмплированный поверх непр-ого

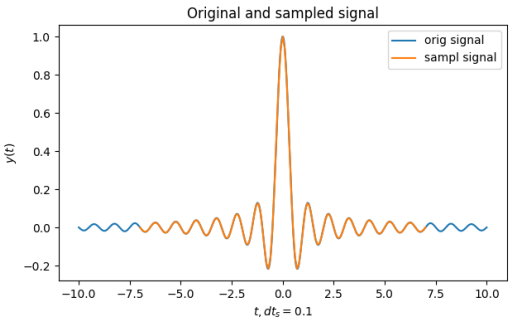


(b) Интерполированный поверх непр-ого

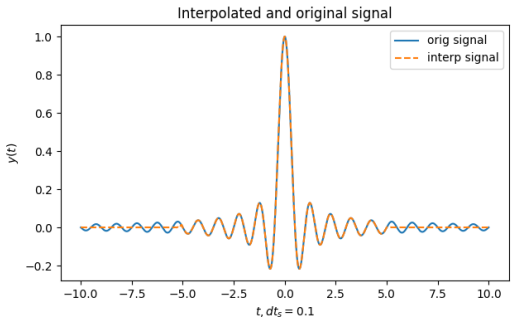


(c) Фурье-образы восст. и исх. сигналов

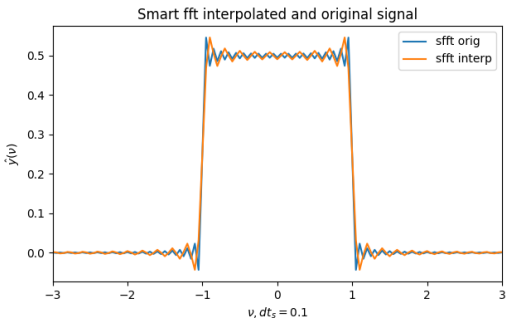
Рис. 14: Сэмплирование sinc при шаге дискретизации $dt = 0.2$



(a) Сэмплированный поверх непр-ого

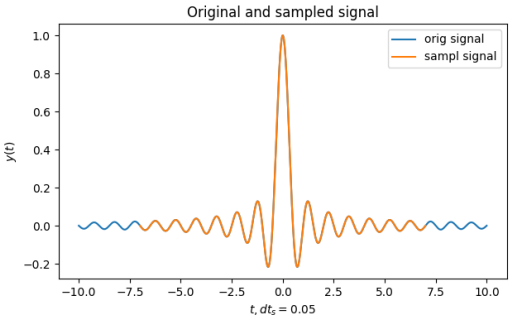


(b) Интерполированный поверх непр-ого

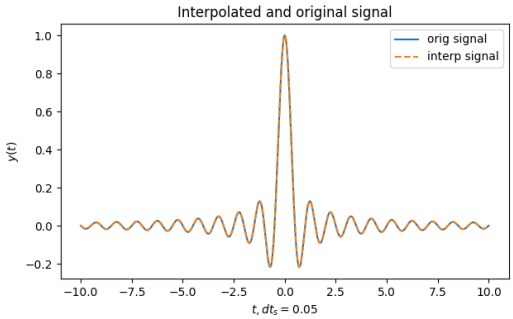


(c) Фурье-образы восст. и исх. сигналов

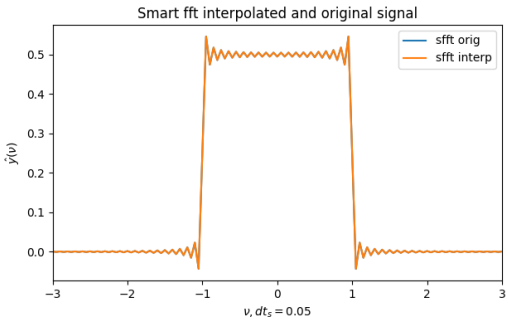
Рис. 15: Уменьшение шага дискретизации $dt = 0.1$



(a) Сэмплированный поверх непр-ого



(b) Интерполированный поверх непр-ого



(c) Фурье-образы восст. и исх. сигналов

Рис. 16: Уменьшение шага дискретизации $dt = 0.05$

Мы получили аналогичные предыдущему пункту результаты – для $\Delta t < 1 \div 14 \approx 0.071$ теорема Найквиста-Шеннона-Котельникова выполнилась и интерполированный сигнал совпадает с иходным (см. рис. 16). Фурье-образ интерполированного сигнала в точности повторяет образ изначального сигнала. При $\Delta t > 0.071$ восстановленный сигнал не совпадает с исходным (см. рис. 14, 15). Фурье-образы тоже не одинаковые, но при уменьшении шага дискретизации Фурье-образ восстановленного сигнала приближается к образу исходного.

2.3 Используемые программы

Вспомогательные функции.

```
# First paragraph function
def y_t(t, a1, a2, w1, w2, p1, p2):
    return a1 * np.sin(w1 * t + p1) + a2 * np.sin(w2 * t + p2)

# Second paragraph function
def y_t_sinc(t, b):
    return np.sinc(b * t)
```

Листинг 6: Методы для нахождения двух функций из задания 2

Основная математическая операция.

```
def interp(f, t, t_s, B):
    ans = np.zeros_like(t)
    for n in range(-len(t_s) // 2, len(t_s) // 2):
        t_n = n / (2 * B)
        ans += f(t_n) * np.sinc(2 * B * (t - t_n))
    return ans
```

Листинг 7: Метод для вычисления интерполированного сигнала

Программа для выполнения задания 2, реализующая предыдущие наработки.

```
# Calculating t array
T = 20
dt = 0.0001
t = np.arange(-T / 2, T / 2 + dt, dt)

# Calculating v array
V = 1 / dt
dv = 1 / T
v = np.arange(-V / 2, V / 2 + dv, dv)

# Calculating the first function (signal)
a1 = 1
a2 = 2
w1 = 3
w2 = 4
p1 = np.pi / 5
p2 = np.pi / 6
y = hp.y_t(t, a1, a2, w1, w2, p1, p2)

# ideal dt_s < 1/2B (Nyquist-Shannon-Kotelnikov theorem)
B = 7
dt_s = 0.05

# Calculating sampled signal
t_s = np.arange(-B, B + dt_s, dt_s)
y_s = hp.y_t(t_s, a1, a2, w1, w2, p1, p2)

def f(t):
    return a1 * np.sin(w1 * t + p1) + a2 * np.sin(w2 * t + p2)

# Calculating interpolated signal
y_ip = fm.interp(f, t, t_s, B)

# Calculating the second function (signal)
b = 2
y_2 = hp.y_t_sinc(t, b)
y_2s = hp.y_t_sinc(t_s, b)

def g(t):
    return np.sinc(b * t)

# Calculating sampled signal
y_2ip = fm.interp(g, t, t_s, B)

# Using smart FFT to calculate Fourier image
const = dt * np.exp(-2 * np.pi * 1j * v * t[0])
y_2_dft = fm.dft(y_2, coeff=const)
y_2ip_dft = fm.dft(y_2ip, coeff=const)

# Build graphs using showf and showfs methods as in the example
```

Листинг 8: Реализация задания 2