



phmatray/TaLibStandard

The GitHub project **phmatray/TaLibStandard** is a **C# implementation of TA-Lib for .NET Standard** designed for conducting technical analysis of financial market data. It modernizes the original TA-Lib (created by Mario Fortier) with new C# and .NET features, including **Generic Math** and support for multiple numeric types.

Overview

TaLibStandard is written mostly in C# (about 93%) and distributed under the **GPL-3.0 license**. The library supports **.NET 8.0** and is available via NuGet. Its latest release, **v2.0.0 (June 17, 2025)**, includes major improvements like C# 13 compatibility, better exception handling, and generic numeric type support. [\[1\]](#) [\[2\]](#) [\[3\]](#)

Key Features

- Implements core TA-Lib functionality in native C#
- Supports **double**, **float**, and **decimal** through **.NET Generic Math**
- Provides automated **code quality checks via Codacy**
- Includes test coverage (≥80%), examples, and documentation
- Can be installed via:

```
dotnet add package Atypical.TechnicalAnalysis.Candles
dotnet add package Atypical.TechnicalAnalysis.Functions
```

- Planned features (from its roadmap) include:
 - More numeric types and trading functions
 - A high-level API
 - A **gRPC service** wrapper for remote analytics
 - Benchmark project and more sample datasets. [\[1\]](#)

Usage and Ecosystem

Developers can start by cloning the repository and exploring its `examples` folder. The project aligns closely with financial analysis workflows, making it particularly useful in **quantitative trading, backtesting, and market signal research** environments. [\[4\]](#) [\[1\]](#)

Repository Stats

- Stars: 85
- Forks: 25
- Watchers: 4
- Total issues: 6
- Total pull requests: 67. ^[5]

In short, **TaLibStandard** is a modern and actively maintained C# translation of TA-Lib for .NET developers, bridging the gap between technical analysis and modern .NET software architectures.



Переведи предыдущий пост на русский

Проект GitHub **phmatray/TaLibStandard** — это **реализация TA-Lib на C# для .NET Standard**, предназначенная для проведения технического анализа данных финансовых рынков. Он модернизирует оригинальный TA-Lib (созданный Марио Фортье), применяя новые возможности C# и .NET, такие как **Generic Math** и поддержка различных числовых типов.

Общие сведения

TaLibStandard в основном написан на C# (около 93%) и распространяется по лицензии **GPL-3.0**. Библиотека поддерживает **.NET 8.0** и доступна через NuGet. Последний релиз, **v2.0.0 (17 июня 2025)**, включает значительные улучшения, такие как совместимость с C# 13, улучшенная обработка исключений и поддержка обобщённых числовых типов. ^[8] ^[9] ^[10]

Главные преимущества

- Реализует основные функции TA-Lib в нативном C#
- Поддерживает **double**, **float** и **decimal** с помощью **.NET Generic Math**
- Автоматическая проверка качества кода через Codacy
- Покрытие тестами (не менее 80%), примеры и документация
- Установка через:

```
dotnet add package Atypical.TechnicalAnalysis.Candles
dotnet add package Atypical.TechnicalAnalysis.Functions
```

- В плане развития (roadmap):
 - Больше числовых типов и торговых функций
 - Высокоуровневое API
 - Обёртка **gRPC-сервиса** для удалённой аналитики
 - Проект бенчмарков и дополнительные примеры данных ^[8]

Использование и окружение

Разработчики могут начать работу, клонировав репозиторий и изучая папку `examples`. Проект тесно связан с рабочими процессами финансового анализа, что делает его особенно полезным для **алгоритмического трейдинга, тестирования стратегий и исследований торговых сигналов**.^[11] ^[8]

Статистика репозитория

- Звёзды: 85
- Форки: 25
- Слежение: 4
- Общее количество проблем (issues): 6
- Всего pull requests: 67^[12]

Вкратце: **TaLibStandard** — это современный и активно поддерживаемый перевод TA-Lib на C# для .NET-разработчиков, объединяющий технический анализ и современные архитектуры программного обеспечения .NET.



Приведите пример реализации индикаторов Использованияю

- SimpleMovingAvarage
- ExponentialMovingAvarage
- AverageTrueRange

и напиши для каждого индикатора Пример Использования

Вот пример реализации каждого из указанных индикаторов на C# с использованием библиотеки TaLibStandard, а также короткая инструкция по их использованию.

Simple Moving Average (SMA)

Реализация:

```
using Atypical.TechnicalAnalysis.Candles;
using Atypical.TechnicalAnalysis.Functions;

// Пусть prices – массив цен (double[])
double[] sma = TechnicalFunctions.SimpleMovingAverage(prices, period: 14);
```

Пример использования:

Рассчитайте среднее значение цены закрытия за последние 14 дней, чтобы определить направление тренда. SMA помогает отфильтровать рыночный «шум» — если цена выше SMA, это сигнал к восходящему тренду.

Exponential Moving Average (EMA)

Реализация:

```
using Atypical.TechnicalAnalysis.Candles;
using Atypical.TechnicalAnalysis.Functions;

// Пусть prices — массив цен (double[])
double[] ema = TechnicalFunctions.ExponentialMovingAverage(prices, period: 14);
```

Пример использования:

ЕМА сильнее реагирует на последние изменения цены. Постройте ЕМА для 14-дневного периода и сравните с SMA — более быстрое пересечение ЕМА и цены может служить сигналом для открытия позиции.

Average True Range (ATR)

Реализация:

```
using Atypical.TechnicalAnalysis.Candles;
using Atypical.TechnicalAnalysis.Functions;

// Пусть high, low, close — массивы цен
double[] atr = TechnicalFunctions.AverageTrueRange(high, low, close, period: 14);
```

Пример использования:

ATR оценивает рыночную волатильность. Используйте ATR для установки плавающих стоп-лоссов: чем выше значение ATR, тем шире допустимая амплитуда движения цены, чтобы избежать ложных срабатываний.

Эти функции помогут анализировать рынок и принимать торговые решения на основе различных аспектов динамики цен.

Если нужно больше подробностей или примеры с конкретными входными данными — уточните, и я подготовлю!

✱✱

В этой библиотеке все функции Технических Индикаторов Статические

?

Да, в библиотеке **TaLibStandard** все основные функции технических индикаторов реализованы как **статические методы**. Это значит, что для их использования не требуется создавать экземпляры классов — достаточно обратиться к функциям напрямую, например:

```
double[] sma = TechnicalFunctions.SimpleMovingAverage(prices, period: 14);
```

Такой подход повышает удобство и производительность при обработке данных для технического анализа, а также делает вызов функций максимально простым и интуитивным. [\[33\]](#) [\[34\]](#)

✱✱

Примеры использования индикаторов

Давайте рассмотрим, как можно инициализировать массив цен свечей (например, из 50 элементов), а затем реализовать регулярное обновление индикаторов — SMA, EMA и ATR — по мере добавления новых значений.

В примерах будет показан сдвиг начального массива (удаление первого элемента и добавление нового) и вызов индикаторов в "скользящем окне".

1. Инициализация данных

```
const int WindowSize = 50;
double[] close = new double[WindowSize];
double[] high = new double[WindowSize];
double[] low = new double[WindowSize];

// Пример: наполнение из источника данных (здесь — случайные числа)
Random rnd = new Random();
for (int i = 0; i < WindowSize; i++) {
    close[i] = rnd.NextDouble() * 100;
    high[i] = close[i] + rnd.NextDouble() * 5;
    low[i] = close[i] - rnd.NextDouble() * 5;
}
```

2. Основной цикл обработки новых данных

```
for (int t = 0; t < 100; t++) // 100 новых свечей
{
    // 1. Получаем новое значение (например, closeNew, highNew, lowNew)
    double closeNew = rnd.NextDouble() * 100;
    double highNew = closeNew + rnd.NextDouble() * 5;
    double lowNew = closeNew - rnd.NextDouble() * 5;

    // 2. Сдвигаем массивы и добавляем новые значения
    Array.Copy(close, 1, close, 0, WindowSize - 1);
    close[WindowSize - 1] = closeNew;
    Array.Copy(high, 1, high, 0, WindowSize - 1);
    high[WindowSize - 1] = highNew;
    Array.Copy(low, 1, low, 0, WindowSize - 1);
    low[WindowSize - 1] = lowNew;

    // 3. Вызываем индикаторы на текущем окне
    double[] sma = TechnicalFunctions.SimpleMovingAverage(close, period: 14);
    double[] ema = TechnicalFunctions.ExponentialMovingAverage(close, period: 14);
}
```

```
double[] atr = TechnicalFunctions.AverageTrueRange(high, low, close, period: 14);

// 4. Используем последнее значение каждого индикатора
Console.WriteLine($"Step {t}: SMA={sma[^1]:F2}, EMA={ema[^1]:F2}, ATR={atr[^1]:F2}");
}
```

Краткое пояснение

- **Array.Copy(...)** — реализует сдвиг массива, убирая самое старое значение и добавляя новое в конец массива.
- После каждого шага индикаторы рассчитываются на последнем актуальном окне данных.
- Выводим последнее значение каждого индикатора для текущего окна.

Если хотите интегрировать с реальным рынком — подключите реальные ценовые данные, а алгоритм останется таким же!

Если нужна версия из примера для другого языка или с чтением данных из файла — сообщите мне это.

Хотите разобраться подробнее, как работает какой-то конкретный индикатор или цикл? Могу задать подробные вопросы для закрепления материала!

Правильно ли я понял, что для того чтобы построить индикатор, например ЕМА,

мы должны сами создать свой класс с входящими параметрами для индикаторов и далее при появлении новых значений вызывать функцию Индикатора в этом классе с параметрами этого класса.

То есть для всех функций необходимо создавать обертки - classes, structures ?

Вы поняли **частично верно**. В библиотеке TaLibStandard индикаторы реализованы как статические функции: вы просто передаёте массив значений и нужные параметры, и функция возвращает значения индикатора для всего массива — исторические и последнее (актуальное) значение.

Однако если вы хотите рассчитывать индикаторы в режиме “реального времени” (online), по одному новому значению за раз с сохранением состояния между вызовами, то действительно, лучше создать свою обёртку-класс или структуру. Такая обёртка будет:

- Хранить внутренние данные (например, “скользящее окно” последних цен).
- Содержать параметры индикатора (период и т.д.).
- При появлении нового значения сдвигать массив и вызывать индикатор.

- Хранить последние вычисления, чтобы не перерасчитывать весь массив при каждом новом значении.

Итак:

- Для разового или пакетного расчёта по готовому массиву — достаточно стандартного вызова функции.
- Для потоковых данных (любое “online” обновление) — удобно сделать свою обёртку-класс, чтобы автоматизировать сдвиг, накопление и вызов функции.

Это общий подход для большинства технических индикаторов в любых библиотеках на C#, Python, C++ и т.д.

Если хотите, могу показать пример такого класса-обёртки для ЕМА в стиле “добавления новых цен по одной”. Скажите, требуется ли такой пример!

Да, пож-ста напишите такой пример такого класса-обёртки для ЕМА в стиле “добавления новых цен по одной”

Конечно! Вот пример простого класса-обёртки для ЕМА, который работает в режиме добавления новых цен по одной. Класс сам хранит последнее значение ЕМА и поддерживает динамическое обновление скользящего окна.

```
using System;
using System.Collections.Generic;

public class EmaOnline
{
    private readonly int _period;
    private readonly double _multiplier;
    private readonly Queue<double> _window;
    private double? _lastEma;

    public EmaOnline(int period)
    {
        _period = period;
        _multiplier = 2.0 / (period + 1);
        _window = new Queue<double>();
        _lastEma = null;
    }

    /// <summary>
    /// Добавляет новое значение и возвращает актуальный ЕМА.
    /// После накопления window.Count == _period — начнёт вычислять ЕМА.
    /// </summary>
    public double Add(double price)
    {
        _window.Enqueue(price);
        if (_window.Count > _period)
            _window.Dequeue();
    }
}
```

```

        if (_window.Count < _period)
        {
            // Недостаточно значений — ещё не вычисляем EMA
            return double.NaN;
        }

        if (_lastEma == null)
        {
            // Первая EMA — просто SMA
            double sma = 0.0;
            foreach (var p in _window) sma += p;
            _lastEma = sma / _period;
        }
        else
        {
            // Классическая EMA формула
            _lastEma = (price - _lastEma.Value) * _multiplier + _lastEma.Value;
        }

        return _lastEma.Value;
    }

    /// <summary>
    /// Возвращает последнее вычисленное значение EMA
    /// </summary>
    public double LastEma => _lastEma ?? double.NaN;
}

```

Пример использования:

```

var ema = new EmaOnline(period: 14);

double[] priceStream = { /* ваши новые цены */ };
foreach (double price in priceStream)
{
    double currentEma = ema.Add(price);
    Console.WriteLine($"Price={price:F2} EMA={currentEma:F2}");
}

```

Пояснения:

- Класс накапливает скользящее окно цен.
- Начинает вычислять EMA, когда накопится достаточно значений для первого SMA.
- Дальше пересчитывает EMA при каждом новом значении.
- Можно получить последнее значение через свойство **LastEma**.

Если хотите добавить новые индикаторы или сделать универсальную обёртку — шаблон этого класса легко расширяется!

Примедите, пож-ста такой же Пример с индикатором ATR

Вот пример класса-обёртки для **ATR (Average True Range)** в стиле “добавления новых свечей по одной”. Класс сам ведёт очереди High, Low, Close и по мере накопления данных рассчитывает и обновляет значение ATR.

```
using System;
using System.Collections.Generic;

public class AtrOnline
{
    private readonly int _period;
    private readonly Queue<double> _highs;
    private readonly Queue<double> _lows;
    private readonly Queue<double> _closes;
    private readonly Queue<double> _trueRanges;
    private double? _lastAtr;
    private double? _prevClose;

    public AtrOnline(int period)
    {
        _period = period;
        _highs = new Queue<double>();
        _lows = new Queue<double>();
        _closes = new Queue<double>();
        _trueRanges = new Queue<double>();
        _lastAtr = null;
        _prevClose = null;
    }

    /// <summary>
    /// Добавляет новые значения свечи и возвращает актуальный ATR
    /// </summary>
    public double Add(double high, double low, double close)
    {
        _highs.Enqueue(high);
        _lows.Enqueue(low);
        _closes.Enqueue(close);

        if (_highs.Count > _period)
        {
            _highs.Dequeue();
            _lows.Dequeue();
            _closes.Dequeue();
            _trueRanges.Dequeue();
        }

        double tr;
        if (_prevClose == null)
        {
            tr = high - low;
        }
        else
```

```

    {
        tr = Math.Max(high - low, Math.Abs(high - _prevClose.Value));
        tr = Math.Max(tr, Math.Abs(low - _prevClose.Value));
    }
    _trueRanges.Enqueue(tr);
    _prevClose = close;

    if (_trueRanges.Count < _period)
    {
        // Недостаточно данных для расчета ATR
        return double.NaN;
    }

    if (_lastAtr == null)
    {
        // Первый ATR – просто среднее TR (SMA)
        double sum = 0.0;
        foreach (var t in _trueRanges) sum += t;
        _lastAtr = sum / _period;
    }
    else
    {
        // Классическая формула сглаживания ATR
        _lastAtr = (_lastAtr.Value * (_period - 1) + tr) / _period;
    }

    return _lastAtr.Value;
}

/// <summary>
/// Получить последнее вычисленное значение ATR
/// </summary>
public double LastAtr => _lastAtr ?? double.NaN;
}

```

Пример использования:

```

var atr = new AtrOnline(period: 14);
double[] highs = { /* ваши новые high */ };
double[] lows = { /* ваши новые low */ };
double[] closes = { /* ваши новые close */ };
for (int i = 0; i < highs.Length; i++)
{
    double currentAtr = atr.Add(highs[i], lows[i], closes[i]);
    Console.WriteLine($"Step {i}: ATR={currentAtr:F2}");
}

```

Пояснения:

- Класс хранит очереди по High, Low, Close и вычисленные TrueRange.
- Считает первый ATR как простое среднее TrueRange, далее применяет классическую формулу сглаживания.
- После каждого добавления новых значений возвращает актуальный ATR.

При необходимости можно легко расширить или адаптировать этот класс под свои задачи!

Спасибо, я как раз это и хотел увидеть. Как использовать эти функции в своих структурах

"При необходимости можно легко расширить или адаптировать этот класс под свои задачи!"

Пожалуйста, рад помочь!

Ваши собственные структуры и классы позволяют гибко использовать любые индикаторы:

- Вы можете хранить **все параметры**, историю данных, результаты расчётов.
- Легко внедрять дополнительные методы — например, алерт-системы, визуализацию или интеграцию с торговым терминалом.
- Добавляйте поддержку других индикаторов или комбинируйте их в одну структуру (например, класс "MyStrategyIndicators", где будут EMA, ATR, SMA и другие, каждый как отдельное поле/метод).

Что можно делать дальше:

- Встроить логику принятия решений — прямо внутри класса индикатора анализировать пересечения, сигналы и выдавать рекомендации.
- Сохранять историю значений всех индикаторов и оперативно строить графики.
- Реализовать потоковую обработку (например, в асинхронном режиме для работы с потоками данных с биржи).

Если потребуется пример комплексной структуры, объединяющей несколько индикаторов или включающей логику трейдинга — просто дайте знать!

Могу подготовить пример реализации с сигналами, журналом событий, а также интеграцией с реальными источниками данных.

Давайте еще вот это почитаем

"Если потребуется пример комплексной структуры, объединяющей несколько индикаторов или включающей логику трейдинга — просто дайте знать!"

Привожу пример **комплексной структуры**, которая объединяет несколько индикаторов и на своём уровне реализует простую логику формирования сигналов для трейдинга (например, сигнал "купить", если цена пересекает EMA и ATR ниже порога).

```
using System;

public class MyStrategyIndicators
{
    public EmaOnline EmaShort { get; }
    public EmaOnline EmaLong { get; }
```

```

public AtrOnline Atr { get; }
public double? LastSignal { get; private set; }

public MyStrategyIndicators(int emaShortPeriod, int emaLongPeriod, int atrPeriod)
{
    EmaShort = new EmaOnline(emaShortPeriod);
    EmaLong = new EmaOnline(emaLongPeriod);
    Atr = new AtrOnline(atrPeriod);
    LastSignal = null;
}

// Добавить новую свечу и получить сигнал
public string AddCandle(double high, double low, double close)
{
    double emaShort = EmaShort.Add(close);
    double emaLong = EmaLong.Add(close);
    double atr = Atr.Add(high, low, close);

    // # Логика: если короткая EMA пересекает длинную снизу (buy), ATR ниже 2
    if (!double.IsNaN(emaShort) && !double.IsNaN(emaLong) && !double.IsNaN(atr))
    {
        if (emaShort > emaLong && atr < 2)
        {
            LastSignal = close;
            return "BUY";
        }
        else if (emaShort < emaLong)
        {
            LastSignal = close;
            return "SELL";
        }
        else
        {
            return "NONE";
        }
    }
    return "WAIT"; // Недостаточно данных для анализа
}
}

```

Пример использования:

```

var strategy = new MyStrategyIndicators(emaShortPeriod: 7, emaLongPeriod: 21, atrPeriod:

// Имитируем поток свечей:
for (int i = 0; i < candleCount; i++)
{
    double high = ...; // ваши данные
    double low = ...;
    double close = ...;
    string signal = strategy.AddCandle(high, low, close);
    Console.WriteLine($"Step {i}: Signal={signal}");
}

```

Пояснения:

- Все индикаторы хранятся внутри одного класса как поля (объекты-обёртки).
- Signal ("BUY"/"SELL"/"NONE"/"WAIT") генерируется на основании правил, которые можно расширять.
- Вы легко добавите новые индикаторы или усложните логику.

Такие структуры хорошо подходят для построения роботов, интеграции с API бирж и создания сложных стратегий!

Если нужен пример с хранением истории сигналов, алертами или журналом событий — дайте знать, напишу!

Сказали А, давайте скажем и В.

Я имею в виду Следующий слой "Signals" над слоем "Индикаторы".

Напиши мне Пример следующего слоя "Signals" над индикаторами.

Что такое Сигнал.

Сигнал - это когда несколько индикаторов возвращают Signal {1,-1,0}

ТО есть мы работаем со слоем Signals и не видим слой Индикаторов.

Мы уже торгуем исходя из значений Signals

Например Signal_EMA_RSI выдает Signal

+1 = EMA (currentPrice > previousPrice) AND RSI < 30 (0.3), то мы покупаем

-1 = EMA (currentPrice < previousPrice) AND RSI > 70 (0.7), то мы продаем.

Напиши слои

- Индикаторы
 - Сигналы
- Реализуй бизнес логику Buy/Sell

Главное условие, чтобы торговый слой был отделен от слоя индикаторов.

Торговый слой работает, только со слоями Сигналов.

Вот архитектурный пример, где слои **Индикаторов** и **Сигналов** полностью разделены:

- Слой индикаторов — только расчёт технических индикаторов и хранение их состояния
- Сигнальный слой — принимает значения индикаторов, преобразует их в торговые сигналы (+1, -1, 0)
- Торговый слой работает **только** с сигналами!

1. Слой индикаторов ("Indicators")

```
public class Indicators
{
    public EmaOnline Ema { get; }
    public RsiOnline Rsi { get; }
```

```

public Indicators(int emaPeriod, int rsiPeriod)
{
    Ema = new EmaOnline(emaPeriod);
    Rsi = new RsiOnline(rsiPeriod);
}

public void AddPrice(double price)
{
    Ema.Add(price);
    Rsi.Add(price);
}

public double CurrentEma => Ema.LastEma;
public double CurrentRsi => Rsi.LastRsi;
public double PreviousEma { get; private set; }

// Для сравнения – храним предыдущее значение EMA
public void ShiftEma(double newEma) => PreviousEma = newEma;
}

```

2. Слой сигналов ("Signals")

```

public class SignalLayer
{
    // Метод генерации сигнала на основании значений индикаторов:
    public int SignalEmaRsi(double currentEma, double previousEma, double currentRsi)
    {
        // BUY: EMA растет + RSI < 30
        if (currentEma > previousEma && currentRsi < 30) return +1;
        // SELL: EMA падает + RSI > 70
        if (currentEma < previousEma && currentRsi > 70) return -1;
        // Нет сигнала
        return 0;
    }
}

```

3. Торговый слой ("Trading")

```

public class TradingLayer
{
    public void OnSignal(int signal, double price)
    {
        if (signal == +1)
        {
            Console.WriteLine($"BUY at price {price:F2}");
            // Логика покупки
        }
        else if (signal == -1)
        {
            Console.WriteLine($"SELL at price {price:F2}");
        }
    }
}

```

```

        // Логика продажи
    }
    else
    {
        // Ожидание, нет торгового действия
    }
}
}

```

4. Пример использования (интеграция слоев)

```

Indicators indicators = new Indicators(emaPeriod: 14, rsiPeriod: 14);
SignalLayer signalLayer = new SignalLayer();
TradingLayer tradingLayer = new TradingLayer();
double? previousEma = null;

// Массив цен (например, приходящий от брокера)
double[] priceStream = { /* ...данные... */ };
foreach (double price in priceStream)
{
    indicators.AddPrice(price);

    // Используем только signalLayer для оценки ситуации!
    double currentEma = indicators.CurrentEma;
    double currentRsi = indicators.CurrentRsi;

    int signal = 0;
    if (previousEma != null)
    {
        signal = signalLayer.SignalEmaRsi(currentEma, previousEma.Value, currentRsi);
        tradingLayer.OnSignal(signal, price); // бизнес-логика только по сигналу!
    }

    // Сохраняем "предыдущую" ЕМА для анализа следующей свечи
    previousEma = currentEma;
}

```

Ключевые моменты

- Вы не торгуете напрямую по индикаторам — только по сигналам!
- Каждый слой занимается своим делом, нет смешивания бизнес-логики и расчетов показателей.
- Можете создавать сколько угодно слоёв сигнала (например, Signal_EMA_RSI_MACD) и добавить ещё правила для торговли на бизнес-уровне.

Если нужен пример "реального" RSI — тоже могу написать простой онлайн-класс! Или интеграцию сразу нескольких сигналов (например, голосование по индикаторам).

Напишите еще и это

Если нужен пример "реального" RSI — тоже могу написать простой онлайн-класс! Или интеграцию сразу нескольких сигналов (например, голосование по индикаторам).

Вот пример **реального онлайн-класса RSI** для обработки потоковых цен (алгоритм Wilder), а также пример "голосования сигналов" — интеграция нескольких сигнальных слоёв, когда окончательное торговое решение принимается на основе "голосов" разных сигналов.

1. Онлайн-класс для RSI

```
using System;
using System.Collections.Generic;

public class RsiOnline
{
    private readonly int _period;
    private readonly Queue<double> _deltas;
    private double? _gainAvg;
    private double? _lossAvg;
    private double? _prevPrice;
    private double? _lastRsi;

    public RsiOnline(int period)
    {
        _period = period;
        _deltas = new Queue<double>();
        _gainAvg = null;
        _lossAvg = null;
        _lastRsi = null;
        _prevPrice = null;
    }

    public double Add(double price)
    {
        if (_prevPrice != null)
        {
            double delta = price - _prevPrice.Value;
            _deltas.Enqueue(delta);
            if (_deltas.Count > _period) _deltas.Dequeue();

            if (_deltas.Count == _period)
            {
                if (_gainAvg == null || _lossAvg == null)
                {
                    // Первый RSI — на базе SMA gains/losses
                    double gainSum = 0.0, lossSum = 0.0;
                    foreach (var d in _deltas)
                    {
                        if (d > 0) gainSum += d;
                        else lossSum -= d;
                    }
                }
            }
        }
    }
}
```



```

        _gainAvg = gainSum / _period;
        _lossAvg = lossSum / _period;
    }
    else
    {
        // Wilder: EMA(AvgGain), EMA(AvgLoss)
        double lastDelta = _deltas.Last();
        _gainAvg = ((_gainAvg.Value * (_period - 1)) + Math.Max(lastDelta, 0)) / _period;
        _lossAvg = ((_lossAvg.Value * (_period - 1)) + Math.Max(-lastDelta, 0)) / _period;
    }

    // RSI формула
    double rs = (_lossAvg == 0) ? 100 : _gainAvg.Value / _lossAvg.Value;
    _lastRsi = 100 - (100 / (1 + rs));
}

}
_prevPrice = price;
return _lastRsi ?? double.NaN;
}

public double LastRsi => _lastRsi ?? double.NaN;
}

```

2. Голосование сигналов ("Signal Voting")

```

public class SignalVotingLayer
{
    // Массив функций-сигналов (например, реализованных как делегаты или лямбды)
    private readonly List<Func<double, double, double, int>> _signalGenerators;

    public SignalVotingLayer()
    {
        _signalGenerators = new List<Func<double, double, double, int>>();
    }

    public void AddSignal(Func<double, double, double, int> signalFunc)
    {
        _signalGenerators.Add(signalFunc);
    }

    // Получить итоговое решение
    public int VoteSignals(double currentEma, double previousEma, double currentRsi)
    {
        int votes = 0;
        foreach (var signalFunc in _signalGenerators)
        {
            votes += signalFunc(currentEma, previousEma, currentRsi);
        }
        // Return sign: +N = Buy, -N = Sell, 0 = None (если сигналы уравновешены)
        if (votes > 0) return +1;
        if (votes < 0) return -1;
        return 0;
    }
}

```

```
}  
}
```

Пример интеграции — голосование двумя сигналами:

```
SignalVotingLayer votingLayer = new SignalVotingLayer();  
  
// Добавим два разных правила в голосование  
votingLayer.AddSignal((currentEma, previousEma, currentRsi) =>  
    (currentEma > previousEma && currentRsi < 30) ? +1 : 0);  
votingLayer.AddSignal((currentEma, previousEma, currentRsi) =>  
    (currentEma < previousEma && currentRsi > 70) ? -1 : 0);  
  
// ... где-то в основном цикле:  
int signal = votingLayer.VoteSignals(currentEma, previousEma, currentRsi);  
// Торгуем только по финальному сигналу голосования!
```

Таким способом ваш торговый слой полностью отделён от слоя индикаторов и может гибко комбинировать логику принятия решений!

Может расширяться сколько угодно — например, добавить голосование для MACD, Stochastic, Bollinger Bands и любых других технических сигналов.

Спасибо большое

Пожалуйста, рад был помочь!

Если появятся дополнительные вопросы по архитектуре, реализации индикаторов, сигналов или торговых стратегий — обращайтесь в любое время. Удачи в разработке и трейдинге!

1. <https://github.com/phmatray/TaLibStandard>
2. <https://www.nuget.org/profiles/phmatray>
3. <https://repos.ecosyste.ms/hosts/GitHub/repositories/phmatray%2FTaLibStandard>
4. <https://www.github-zh.com/projects/126631575-talibstandard>
5. <https://issues.ecosyste.ms/hosts/GitHub/repositories/phmatray%2Ftalibstandard>
6. <https://www.github-zh.com/topics/technical-analysis?l=C%23>
7. https://www.linkedin.com/posts/nick-chapsas_github-opensource-activity-6821744895540121600-a_qW
8. <https://github.com/phmatray/TaLibStandard>
9. <https://www.nuget.org/profiles/phmatray>
10. <https://repos.ecosyste.ms/hosts/GitHub/repositories/phmatray%2FTaLibStandard>
11. <https://www.github-zh.com/projects/126631575-talibstandard>
12. <https://issues.ecosyste.ms/hosts/GitHub/repositories/phmatray%2Ftalibstandard>
13. <https://gist.github.com/troyscott/9d1ba51ca0fc2842b922732cc059dfa7>
14. <https://github.com/phmatray/TaLibStandard/blob/main/docs/functions.md>

15. <https://www.purple-trading.com/en/moving-average-the-most-commonly-used-tool-for-technical-analysis/>
16. <https://www.investopedia.com/terms/m/movingaverage.asp>
17. <https://www.axi.com/uk/blog/education/moving-averages>
18. <https://ru.scribd.com/document/198381770/Simple-Moving-Average-Technical-Analysis>
19. <https://hycm.com/en/blog/advanced-trading-concepts/technical-analysis>
20. <https://www.morpher.com/blog/exponential-moving-average>
21. <https://www.oanda.com/us-en/trade-tap-blog/analysis/technical/how-to-use-average-true-range-atr/>
22. <https://www.investopedia.com/terms/s/sma.asp>
23. <https://lightningchart.com/blog/trader/exponential-moving-average-indicator/>
24. <https://www.youtube.com/watch?v=khHtaJuwI2Q>
25. <https://ru.tradingview.com/scripts/averagetruerange/page-10/>
26. <https://ta-lib.github.io/ta-lib-python/>
27. <https://github.com/TA-Lib/ta-lib-python>
28. <https://stackoverflow.com/questions/45186262/exponential-moving-average-pandas-vs-ta-lib>
29. <https://machinelearning-basics.com/what-is-ta-lib-and-how-to-implement-technical-indicators-in-python/>
30. <https://www.kaggle.com/code/sajikim/technical-analysis-talib-wti-crude-oil-price>
31. <https://github.com/phmatray/TaLibStandard/blob/main/README.md>
32. <https://github.com/uyplayer/rusty-talib>
33. <https://github.com/phmatray/TaLibStandard/blob/main/README.md>
34. <https://github.com/phmatray/TaLibStandard/blob/main/docs/functions.md>