ВСР 2.2

Подборка материалов по языку программирования Julia

1. The Julia Programming Language. — Текст : электронный // Julia : [сайт]. — URL: https://julialang.org/ (дата обращения: 23.09.2025).

Основной ресурс для начала работы. Содержит installer, вводное руководство, полную документацию и список пакетов. Раздел «Learn» является отправной точкой для новичков. Критически важен для понимания философии и возможностей языка из первых рук.

2. Lectures. — Текст : электронный // QuantEcon : [сайт]. — URL: quantecon.org/lectures/ (дата обращения: 23.09.2025).

Открытая образовательная платформа по количественной экономике, содержащая обширный курс по программированию на Julia. Лекции представляют собой углубленные интерактивные уроки, охватывающие как основы языка, так и продвинутые темы: производительность, параллельные вычисления, решение экономических моделей. Ресурс отличается академической строгостью и практической направленностью, что делает его ценным не только для экономистов, но и для всех специалистов в области вычислений.

3. Documentation. — Текст : электронный // Flux : [сайт]. — URL: https://fluxml.ai/Flux.jl/stable/ (дата обращения: 23.09.2025).

Документация ведущего фреймворка для машинного обучения в Julia.

4. DifferentialEquations.jl. — Текст : электронный // SciML : [сайт]. — URL: https://docs.sciml.ai/DiffEqDocs/stable/ (дата обращения: 23.09.2025).

Библиотека для решения различных типов дифференциальных уравнений.

Примеры решения задач с комментариями

1. Синтаксис и множественная диспетчеризация

```julia
# Определяем абстрактный тип и два конкретных подтипа
abstract type Shape end

struct Circle <: Shape
    radius::Float64
end

struct Rectangle <: Shape
    width::Float64
    height::Float64
end

# Определяем универсальную функцию area()
# Используем множественную диспетчеризацию: одна функция, разные реализации в зависимости от типа аргумента.

# Метод для Circle
area(shape::Circle) = π * shape.radius^2
```

```
# Метод для Rectangle
area(shape::Rectangle) = shape.width * shape.height

# Создаем объекты
circle = Circle(5.0)
rect = Rectangle(4.0, 6.0)

# Вызываем одну и ту же функцию с разными типами.
# Julia автоматически выберет корректную реализацию.
println("Площадь круга: ", area(circle))    # Вызов area(::Circle)
println("Площадь прямоугольника: ", area(rect)) # Вызов area(::Rectangle)

# Это мощная парадигма, позволяющая писать обобщенный и легко расширяемый
код.
# Новый тип Shape (например, Triangle) можно добавить, просто определив для
него новый метод area.
```

Этот пример иллюстрирует краеугольный камень Julia — множественную диспетчеризацию. В отличие от ООП-языков, где метод привязан к классу объекта, в Julia выбор метода определяется типами *всех* аргументов функции. Это делает код более модульным и понятным, что критически важно в научных вычислениях, где операции часто определяются для разных комбинаций математических объектов.

2. Матричные операции и производительность

```
using LinearAlgebra  # Используем стандартную библиотеку для линейной
алгебры

# Создаем две случайные матрицы 1000x1000
A = rand(1000, 1000)
B = rand(1000, 1000)

# Умножение матриц в Julia записывается так же просто, как и в Python
(с помощью NumPy), но...
# ...под капотом это высокооптимизированная операция, использующая
BLAS и LAPACK библиотеки.
C = A * B

# Вычисление собственных значений
eigen_vals = eigen(C).values
println("Сумма собственных значений: ", sum(eigen_vals))

# Сравним производительность с поэлементной операцией
function slow_matrix_mult(A, B)
    # Наивная, неоптимизированная реализация (для демонстрации)
    m, n = size(A)
    n2, p = size(B)
    @assert n == n2
    C = zeros(m, p)
    for i in 1:m
        for k in 1:n
            for j in 1:p
                C[i, j] += A[i, k] * B[k, j]
            end
        end
    end
    return C
```

```
end

# Используем встроенный макрос @time для замера времени
println("Оптимизированное умножение:")
@time A * B;

println("Наивное умножение:")
@time slow_matrix_mult(A, B);
```

Этот пример показывает две сильные стороны Julia:

1. Выразительность: Синтаксис для линейной алгебры чистый и математически естественный.
2. Производительность "из коробки": Стандартные операции, как * для матриц, делегируются оптимизированным compiled-библиотека. При этом пользователь может написать свой наивный алгоритм на самом Julia, и он все равно будет достаточно быстрым благодаря компиляции в машинный код, но, как видно, специализированные реализации значительно эффективнее.

3. Простое машинное обучение с Flux.jl

```
using Flux, Statistics, Random
Random.seed!(1234) # Для воспроизводимости результатов

# 1. Генерация синтетических данных: y = 2x - 3 + шум
x = rand(100) * 10 .- 5 # 100 точек от -5 до 5
y = 2 .* x .- 3 .+ randn(100) * 0.5 # Целевая функция с шумом

# 2. Определение модели: простая линейная регрессия
# Модель — это просто анонимная функция с обучаемыми параметрами (weights, bias).
model = Dense(1 => 1) # Полносвязный слой: 1 вход -> 1 выход
# Изначально model.weight и model.bias содержат случайные значения.

# 3. Функция потерь (Loss function) - Среднеквадратичная ошибка (MSE)
loss(x, y) = Flux.mse(model(x), y)

# 4. Подготовка данных и выбор оптимизатора
data = [(reshape([x_i], 1, 1), [y_i]) for (x_i, y_i) in zip(x, y)] # Упаковка данных в мини-батчи
optimizer = Descent(0.01) # Градиентный спуск с шагом 0.01

# 5. Тренировка модели
parameters = Flux.params(model) # Собираем все обучаемые параметры модели

for epoch in 1:1000
    Flux.train!(loss, parameters, data, optimizer)
    if epoch % 100 == 0
        current_loss = mean(loss(reshape([x_i], 1, 1), [y_i]) for (x_i, y_i) in zip(x, y))
        println("Epoch $epoch, Loss: $current_loss")
    end
end

```

```
# 6. Проверка результатов
println("Истинные параметры: k=2, b=-3")
println("Обученные параметры: k=$(round(model.weight[1]; digits=2)), b=$(round(model.bias[1]; digits=2))")
```

Этот пример демонстрирует элегантность Flux.jl. Модель определяется декларативно, а процесс обучения (включая автоматическое дифференцирование для вычисления градиентов с помощью Zygote.jl) инкапсулирован в простую функцию train!. Код выглядит почти так же, как и на Python с Keras, но является полностью нативным для Julia, что позволяет без проблем встраивать собственные функции и использовать всю мощь языка для пред-/пост-обработки данных.