

# Метод инвариантного погружения

## Table of Contents

Школьная задача.....	1
Последовательность Фибоначчи.....	3
Процедурное решение и рекурсия.....	3
Memoization pattern.....	4
Класс dictionary.....	4
Класс-"запоминатель".....	6
Уравнение переноса излучения в двухпоточковом приближении.....	10
Классический подход.....	11
Метод инвариантного погружения для уравнения переноса излучения в двухпоточковом приближении.....	14
Аналитическое решение уравнения инвариантного погружения для функции отражения.....	22

```
[folder,name] = fileparts(matlab.desktop.editor.getActiveFilename);  
addpath(fullfile(folder,name+"_deps"));
```

## Школьная задача

$$\sqrt{3 + \sqrt{3 + \sqrt{3 \dots}}} - ?$$

$$F(n) = \sqrt{3+n} \sqrt{3+n-1} \dots \sqrt{3+1} \sqrt{3}$$

$$F(n \rightarrow \infty) = ?$$

```
function s = fval(N)  
% прямое суммирование  
a = 3;  
s = sqrt(3);  
for i = 1:N  
    s = sqrt(s + a);  
end  
end
```

```
fval(15) % вариант процедурный
```

```
ans = 2.3028
```

```
function s = fval_rec(s,i,N)  
% суммирование через рекурсию  
a=3;
```

```

    if i>=N
        return
    else
        s= sqrt(a + fval_rec(s,i+1,N));
    end
end

```

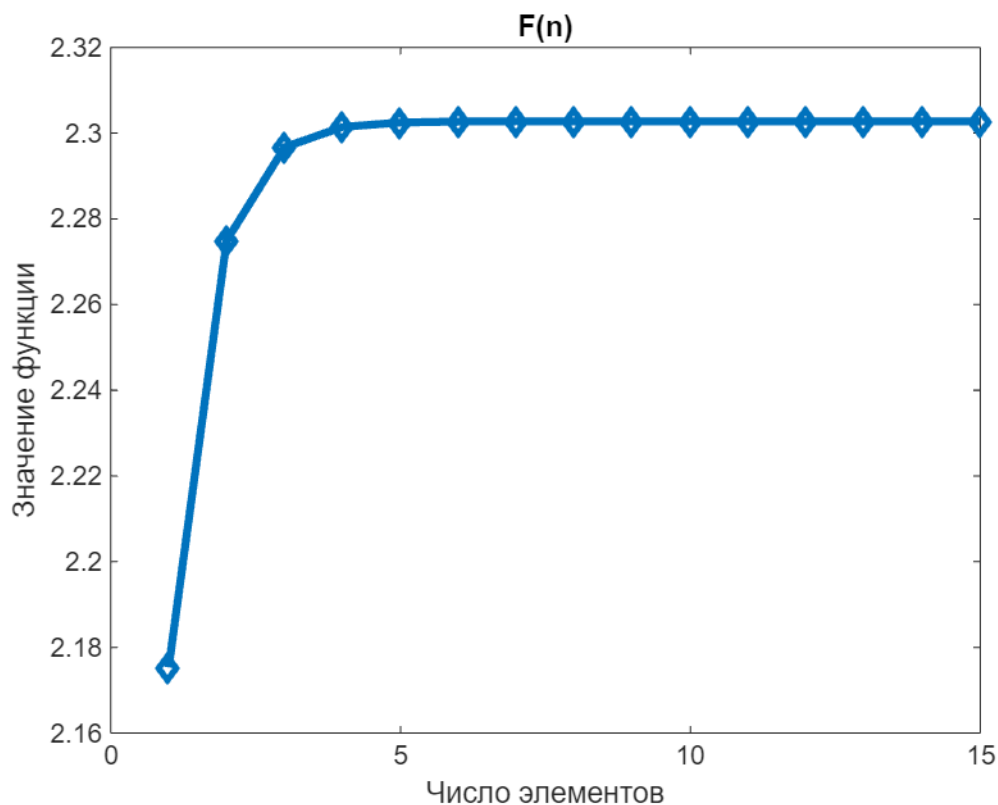
`fval_rec(0,0,15) % вариант через рекурсию`

`ans = 2.3028`

```

plot(arrayfun(@fval,1:15),Marker = 'diamond',linewidth=3)
title(" F(n) ")
xlabel("Число элементов")
ylabel("Значение функции")

```



$$F(n) = \sqrt{3 + \sqrt{3 + \sqrt{3 + \dots \sqrt{3 + \sqrt{3}}}}} = \sqrt{3 + F_{n-1}}$$

$$X = \sqrt{3 + X}, X^2 - X - 3 = 0 \quad (X > 0)$$

$$X = \frac{-1 \pm \sqrt{1+12}}{2}$$

```
0.5*(1+sqrt(13))
```

```
ans = 2.3028
```

Предполагаем, что решение нам известно и смотрим что будет если к нему добавить еще один элемент.

## Последовательность Фибоначчи

$$X_{n+1} = X_n + X_{n-1}, X_1 = 1, X_2 = 1$$

### Процедурное решение и рекурсия

Процедурное решение - суммирование в цикле.

Функциональное решение - суммирование при помощи рекурсии

```
fib_dir(7)
```

```
ans = 13
```

```
fib_rec(7)
```

```
ans = 13
```

```
tic;fib_dir(35);toc
```

```
Elapsed time is 0.000334 seconds.
```

```
tic;fib_rec(35);toc
```

```
Elapsed time is 0.149605 seconds.
```

Почему рекурсия медленно работает можно понять если посмотреть граф вызова

```
function val = fib_rec(N)
% рекурсивное решение
    if N<=2
        val = 1;
        return
    end
    val = fib_rec(N-1) + fib_rec(N-2);
end
```

```
plotFibTreeGraph(7)
```



```
% используют pass-by-value
```

```
f(d,"b",34)
```

```
ans =
```

```
dictionary (string & double) with 2 entries:
```

```
"a" & 10
```

```
"b" & 34
```

```
d
```

```
d =
```

```
dictionary (string & double) with 1 entry:
```

```
"a" & 10
```

Обертка ("wrapper") для стандартного словаря в матлаб, чтобы нам удобнее было писать данные

```
classdef Dict< handle
    % обертка вокруг стандартного словаря (dictionary) чтобы передавать его
    % по указателю (pass-by-handle)
    properties
        d % стандартный словарь
    end
    methods
        function obj = Dict()
            obj.d = dictionary;
        end
        function f = haskey(obj,k) % функция проверяет есть ли данный ключ в коллекции
            f = isConfigured(obj.d) && isKey(obj.d,k);
        end
        function add(obj,key,val) % добавляет в коллекцию элемент val с ключем key
            obj.d = insert(obj.d,key,val);
        end
        function val = get(obj,key) % достает содержимое по ключу
            val = obj.d(key);
        end
    end
end
```

Рекурсивная функция с запоминанием значений

```
function val = fib_mem(N)
    persistent M % хранилище для уже посчитанных элементов последовательности будет храниться
    % в памяти функции
    if isempty(M)
        M = Dict();
    end
    if N<=2 % первые два элемента равны единице
        val = 1;
        M.add(N,val);
        return
    end
    if M.haskey(N) % проверяем есть ли текущий элемент последовательности в хранилище
        % если есть, то возвращаем его
        val = M.get(N);
        return
    end
```

```

end
val = fib_mem(N-1) + fib_mem(N-2); % рекурсивно рассчитываем следующий
% элемент последовательности и добавляем его в словарь
M.add(N,val);
end

```

```
fib_mem(30)
```

```
ans = 832040
```

```
tic;fib_rec(35);toc
```

```
Elapsed time is 0.140574 seconds.
```

```
tic;fib_mem(35);toc
```

```
Elapsed time is 0.000133 seconds.
```

```
clear fib_mem
```

## Класс-"запоминатель"

Функция **fib\_mem** позволяет решать рекурсивную задачу фибоначчи с запоминанием, она специализированная, при помощи ООП можно сделать некоторый универсальный класс, который позволяет нам использовать произвольную рекурсивную функцию.

```

classdef memoizer_class<handle
    % позволяет имитировать паттерн closure для рекурсивной
    % функции, которая принимает функцию в качестве второго аргумента
    % f = @(x,f) f(N-2)+ f(N-1) - например для последовательности
    % Фибоначчи
    properties (SetAccess=private)
        d Dict % словарь, который будет хранить уже посчитанные пары аргумент-значение функции
        f function_handle
    end
    methods
        function obj = memoizer_class(f,init_keys,init_vals)
            arguments
                f function_handle
                init_keys double=[]
                init_vals double=[]
            end
            % конструктор
            % функция f - указатель на функцию, которая первым аргументом
            % принимает независимую переменную, а вторым - функцию, которая вызывается рекурсивно,
            % также она будет опционально принимать стартовые значения для последовательности
            % (первые элементы последовательности, стартовые условия, которые сразу записываются в словарь)
            % это сделано потому, что анонимные функции в матлаб не
            obj.d = Dict();
            if ~isempty(init_keys)
                arrayfun(@(i)add(obj.d,init_keys(i),init_vals(i)),1:length(init_keys)); % добавляем
                % стартовые условия в словарь
            end
        end
    end
end

```

```

        obj.f = f;
    end
    function val = fun(obj,N)
        if obj.d.haskey(N)
            val = get(obj.d,N);
        else
            val = obj.f(N,@obj.fun);
            add(obj.d,N,val)
        end
    end
end
end
end

```

```

fib2 = @(N,f) f(N-2) + f(N-1); % создаем анонимную функцию
m = memoizer_class(fib2,[1 2],[1 1]); % передаем мемоизатору функцию и начальные
условия
m.fun(45);

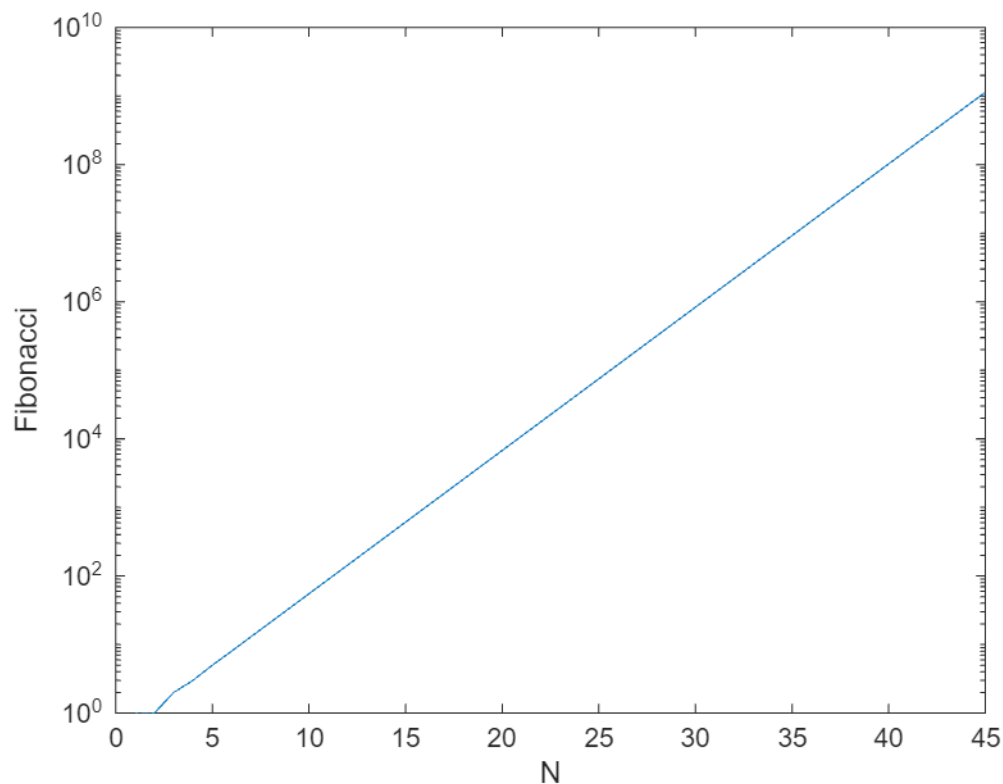
```

Одно из преимуществ способа запоминания при помощи классов, является то, что после выполнения функции все содержимое словаря может быть легко доступно. Построим график всех значений в словаре.

```

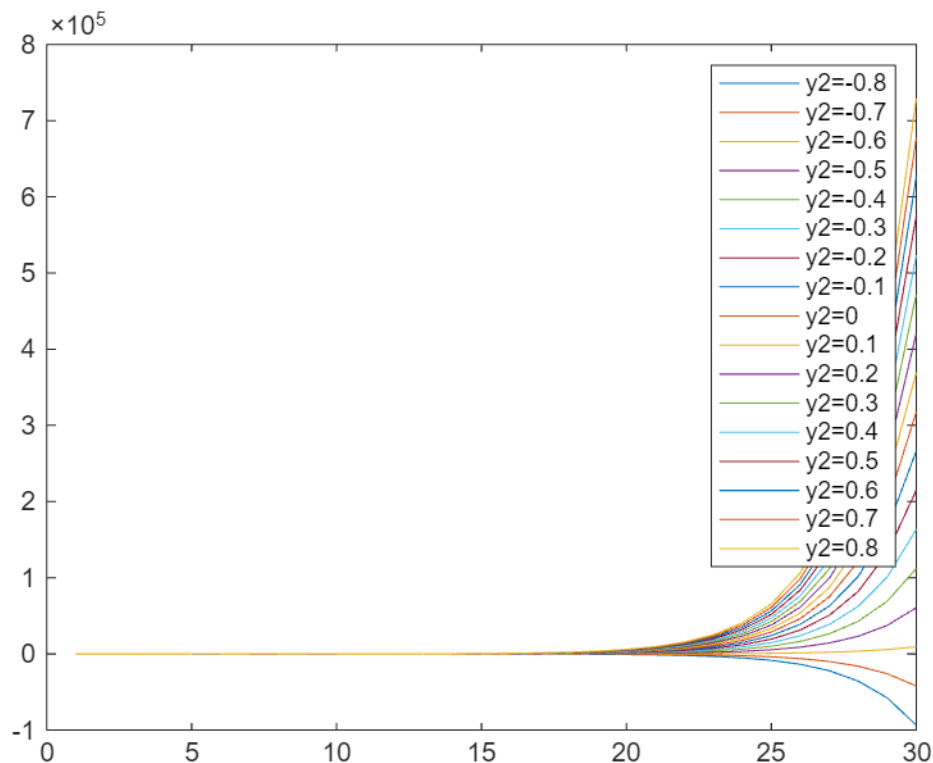
all_values = arrayfun(@(i)get(m.d,i),1:45);
plot(1:45,all_values);xlabel("N");ylabel("Fibonacci")
yscale log;

```



В отличие от функционального решения, класс -запоминатель является универсальным, например, мы можем проанализировать устойчивость рекуррентной последовательности в зависимости от начальных условий:

```
fib2 = @(N,f) f(N-2) + f(N-1); % создаем анонимную функцию
y2_ = -0.8:0.1:0.8;
NMAX = 30;
f_max = zeros(length(y2_),NMAX); % тут будут храниться полученные значения
counter = 0;
for y2 = y2_
    counter = counter+1;
    m = memoizer_class(fib2,[1 2],[1 y2]); % передаем мемоизатору функцию и
начальные условия
    m.fun(NMAX);
    f_max(counter,:) = arrayfun(@(i)get(m.d,i),1:NMAX);
end
plot(1:NMAX,f_max')
legend("y2=" + string(y2_))
```



Или изучить параметрическую устойчивость разностной схемы

```
f_new = @(N,f,a,b) a*f(N-2) + b*f(N-1)
```

```
f_new = function_handle with value:
    @(N,f,a,b)a*f(N-2)+b*f(N-1)
```



```

a = 1:-0.1:0.1;
b = -1:0.1:-0.1;
NMAX = 100;
f_max = zeros(length(a),length(b),NMAX); % тут будут храниться полученные значения

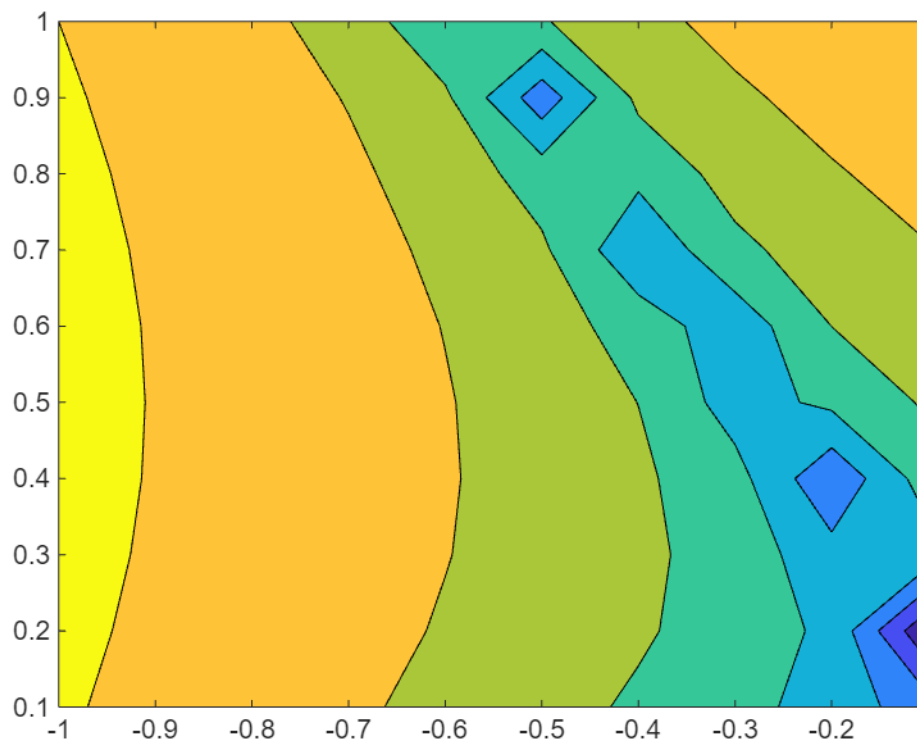
for i = 1:length(a)
    for j=1:length(b)
        ai = a(i);bj = b(j);
        fib2 = @(N,f) f_new(N,f,ai,bj);
        m = memoizer_class(fib2,[1 2],[1 1]); % передаем мемоизатору функцию и
начальные условия
        m.fun(NMAX);
        f_max(i,j,:) = arrayfun(@(i)get(m.d,i),1:NMAX);
    end
end

```

```

sc = 5;
contourf(b,a,log(abs(f_max(:, :, sc ))))

```



```

NMAX=35;
t_fib_dir= zeros([NMAX,1]);t_fib_rec = t_fib_dir;t_fib_mem_rec=t_fib_rec;
t_fib_mem_class = t_fib_dir;
fib_mem_class = memoizer_class(fib2,[1 2],[1 1])

```

```

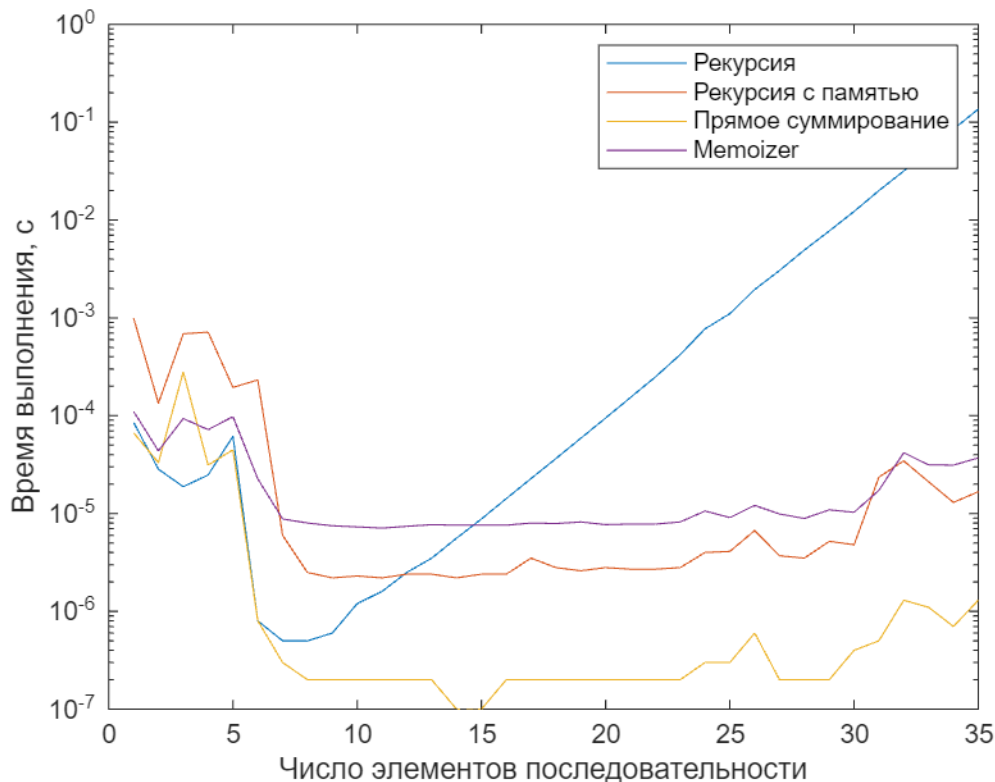
fib_mem_class =

```

memoizer\_class with properties:

```
d: [1x1 Dict]
f: @(N,f)f_new(N,f,ai,bj)
```

```
for n = 1:NMAX
    tic;fib_rec(n);t_fib_rec(n) = toc;
    tic;fib_mem(n);t_fib_mem_rec(n) = toc;
    tic;fib_dir(n);t_fib_dir(n) = toc;
    tic;fib_mem_class.fun(n);t_fib_mem_class(n)=toc;
end
plot(1:NMAX,t_fib_rec,1:NMAX,t_fib_mem_rec,1:NMAX,t_fib_dir,1:NMAX,t_fib_mem_class)
yscale log
legend(["Рекурсия" "Рекурсия с памятью" "Прямое суммирование" "Memoizer"])
xlabel("Число элементов последовательности")
ylabel("Время выполнения, с")
```



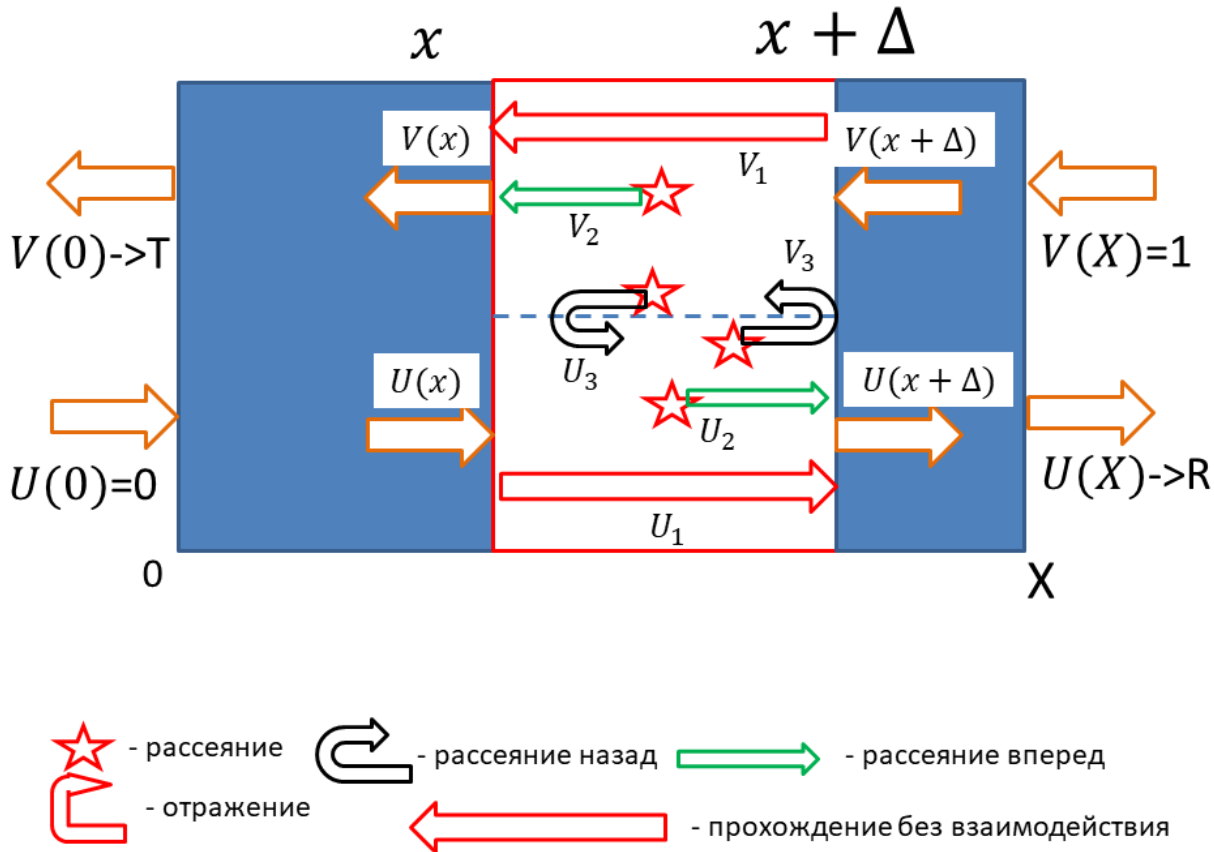
## Уравнение переноса излучения в двухпоточковом приближении

Будем решать следующую задачу:

Пусть у нас есть некоторая одномерная среда длиной  $X$ , в которой распространяется излучение (поток частиц). Вероятность того, что частица на участке пути длиной  $\Delta$  равна  $\sigma \cdot \Delta$ . Доля частиц, рассеиваемых вперед (продолжающих двигаться в том же направлении, что и до рассеяния) равна  $f$ , а в противоположном -  $b$ . Все параметры  $(\sigma, f, b)$  могут зависеть от координат. Излучение падает с левого края.

## Классический подход

Будем рассматривать задачу в двух потоковом приближении, то есть у нас есть плотность излучения движущегося вправо ( $U$ ) и влево -  $V$ .



Рассмотрим небольшой участок пути излучения длиной  $\Delta$  и, чтобы сформулировать дифференциальное уравнение переноса, рассмотрим потоки излучения на этом участке. Поток излучения в объеме этого элемента формулируется как сумма потоков движущихся влево и вправо:

Для излучения, движущегося вправо:

$$U(x + \Delta) = U_1 + V_3 + U_3$$

$U_1 = (1 - \sigma\Delta)U(x)$  - излучение, которое пролетает через слой не рассеиваясь ( $1 - \sigma\Delta$ ) - вероятность того, что рассеяния не произойдет (единица минус вероятность рассеяния  $\sigma\Delta$ ).

$U_2 = f\sigma\Delta U(x)$  - доля излучения, рассеянного вперед

$U_3 = b\sigma\Delta V(x + \Delta)$  - часть потока, идущего влево, рассеянная назад (для потока, идущего вправо - это попутное направление).

$$U(x + \Delta) - U(x) = \sigma\Delta[(f - 1)U(x) + bV(x + \Delta)] + o(\Delta)$$

Аналогично для потока вправо:  $V(x) = (1 - \sigma\Delta)V(x + \Delta) + f\sigma\Delta V(x + \Delta) + b\sigma\Delta U(x)$

$$V(x + \Delta) - V(x) = \sigma \Delta [-bU(x) - (f - 1)V(x + \Delta)] + o(\Delta)$$

Теперь можно поделить на  $\Delta$  и устремить его к нулю, что даст систему дифференциальных уравнений:

$$\begin{cases} \frac{dU}{dx} = \sigma[(f - 1)U(x) + bV(x)] \\ \frac{dV}{dx} = \sigma[-bU(x) - (f - 1)V(x)] \end{cases}$$

Удобно систему переписать в виде одного векторного уравнения:

$$\frac{d}{dx} \vec{Y} = \sigma \begin{bmatrix} f-1 & b \\ -b & -(f-1) \end{bmatrix} \vec{Y}; \quad \vec{Y} = \begin{bmatrix} U \\ V \end{bmatrix} \quad (1)$$

$$\frac{d}{dx} \vec{Y} = T \vec{Y}$$

Граничные условия :

$U(0) = 0$  - слева излучение не падает

$V(X) = 1$  - справа излучение падает

Это краевая задача

$$\frac{d}{dx} \vec{Y} = f(x, \vec{Y})$$

$G(\vec{Y}(0), \vec{Y}(X)) = \Gamma \cdot [\vec{Y}(0); \vec{Y}(X)] = \vec{0}$  - граничные условия ( $x = 0, x = X$  - границы области)

В нашем случае :

$f(x, \vec{Y}) = T(x) \vec{Y}$  (если  $\sigma, f, b$  не зависят от координат, то  $T$  - постоянная матрица)

Граничные условия:

$$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} U(0) & U(X) \\ V(0) & V(X) - 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

Матлаб имеет несколько численных методов для решения краевых задач.

Так как правая часть у нас имеет некоторые параметры, сделаем функцию, которая оборачивает конкретные значения параметров и возвращает анонимную функцию в **нужной** форме, правая часть нашего дифференциального **уравнения**:

```
function fun = dy(sigma,alfa,b,f)
% функция оборачивает конкретные значения параметров в анонимную функцию,
% которая будет использована в решателе краевых задач
T = zeros(2);
T(1) = sigma*(f-1)-alfa;
T(2) = sigma*b;
T(3) = -T(2);
T(4) = -T(1);
```

```

    fun = @(x,y)T*y(:);
    % первый аргумент - независимая переменная (сетка)
    % второй аргумент - само значение функции
end

```

Граничные условия в форме:

```

function res = bcfun(y0,yX) % y0 и yX - вектора решения уравнения слева и справа соответственно
res = [y0(1)
      yX(2)-1];
end

```

Также для решения краевых задач требуется стартовая аппроксимация, мы возьмем линейную зависимость:

```

function g = guess(x)
% стартовая аппроксимация
% x - сетка, на которой будет решаться уравнение
g = [0.5*x
     x];
end

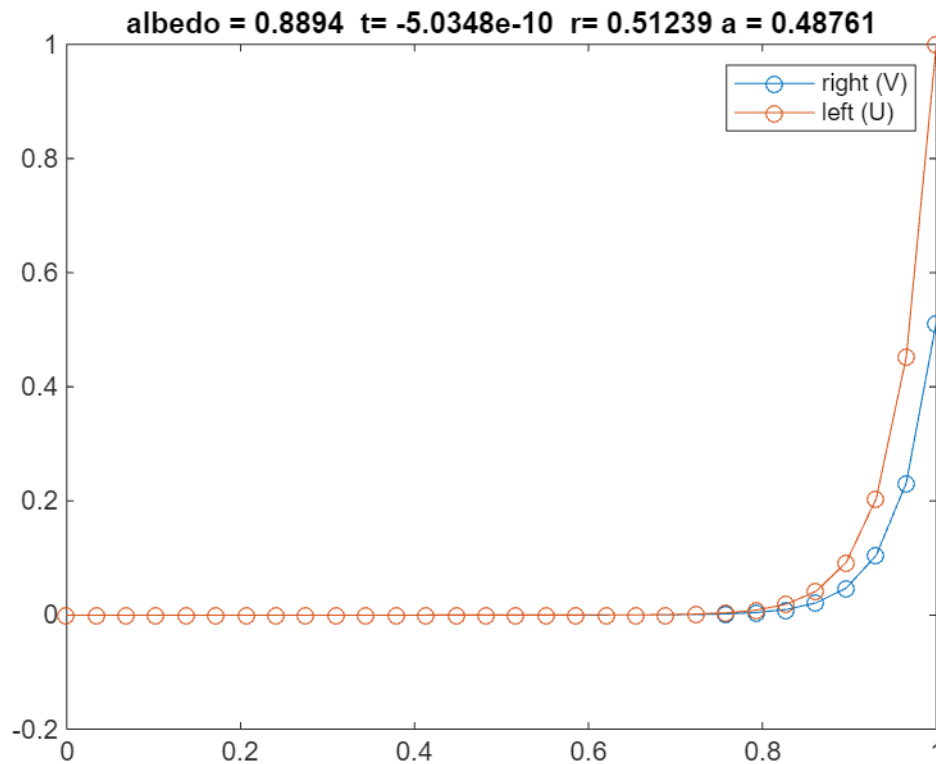
```

```

xmesh = linspace(0,1,30);% строим сетку, на которой будет
% искомое решение краевой задачи
solinit = bvpinit(xmesh, @guess); % исходное распределение
sigma=59.674;
% коэффициент f
forward_fraction = 0.464;
alfa = 7.421;
% коэффициент b в уравнении
backward_fraction = 1 - forward_fraction;
dy_fun = dy(sigma,alfa,backward_fraction,forward_fraction);
sol = bvp4c(dy_fun, @bcfun, solinit);
t = sol.y(2,1);
r = sol.y(1,end);
a = 1 - t - r;

plot(sol.x, sol.y,"-o" )
legend(["right (V)", "left (U)"])
title ( "albedo = "+ ...
        string(sigma/(sigma+alfa)) ...
        +" t= "+string(t)+ ...
        " r= "+string(r) + ...
        " a = "+string(a))

```



Когда потоки в объеме расчетной области найдены, коэффициент отражения это  $R = U(X) = \vec{Y}(X)(1)$  - последний элемент в функции для потока, движущегося вправо.

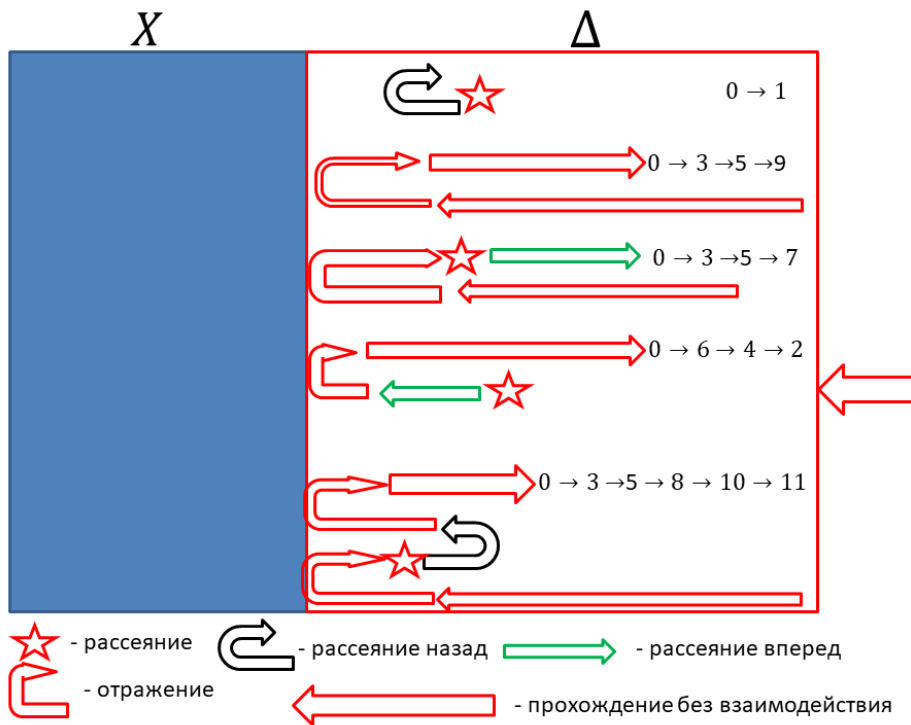
А коэффициент пропускания:

$T = V(0) = \vec{Y}(0)(2)$  - то есть второй элемент вектора решения на левой границе (поток, идущий с границы влево)

### Метод инвариантного погружения для уравнения переноса излучения в двухпоточковом приближении

В действительности нам, как правило, не нужны внутренние поля, нас интересуют коэффициенты отражения и пропускания на границе.

Поэтому будем писать уравнения сразу для отражения (для пропускания не будем, но логика там такая же).



Предположим, что нам уже известно отражения от слоя толщиной  $X$ , растянем слой, добавив к нему слой толщиной  $\Delta$  и рассмотрим как падающее излучение может взаимодействовать с этим добавленным слоем. На картинке выше каждому элементарному акту взаимодействия соответствует своя пиктограмма.

Излучение:

1. Рассеивается назад, падающее излучения сразу выходит, рассеявшись назад (путь 0-1 на графе ниже)
2. Проходит через слой нерассеявшись, отражается и опять нерассеявшись выходит из слоя (0-3-5-9)
3. Проходит не рассеявшись, отражается и рассеивается вперед (0-3-5-7)
4. Рассеивается вперед, отражается, проходит нерассеявшись (0-6-4-2)
5. Проходит, отражается, рассеивается назад, отражается, проходит без рассеяния (0-3-5-8-10-11)

Ниже приведен направленный граф рассеяний в добавленном слое. Каждый узел - единичный акт взаимодействия между излучением и добавленным слоем среды толщиной  $\Delta$ .

$(1 - \sigma\Delta)$  - прямое прохождение слоя без рассеяния

$f\sigma\Delta$  - рассеяние вперед

$b\sigma\Delta$  - рассеяние назад

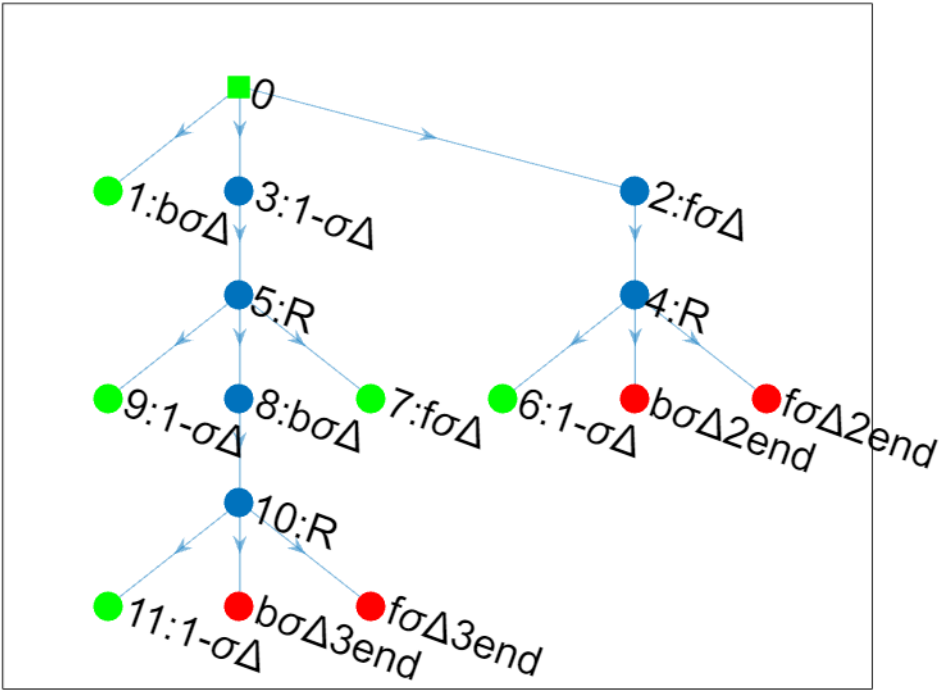
$R$  - отражение от слоя толщиной  $X$

Использованная для построения графа функция **inv\_emb\_refl\_graph** возвращает:

$G$  - сам граф

`real_leafNodes` - узлы, вносящие вклад первого порядка в отражение  
`dead_leafNodes` - узлы, вносящие вклад второго порядка в отражение  
`p` - картинка (для объекта типа направленный граф (класс **digraph** в матлаб))

```
[G,real_leafNodes,dead_leafNodes,p] = inv_emb_refl_graph()
```



`G =`  
 digraph with properties:

Edges: [15×1 table]  
 Nodes: [16×1 table]  
`real_leafNodes =` 5×1 table

	Name
1	'1:b\sigma\Delta'
2	'9:1-\sigma\Delta'
3	'7:f\sigma\Delta'
4	'6:1-\sigma\Delta'
5	'11:1-\sigma\Delta'

`dead_leafNodes =` 4×1 table

	Name
1	'b\sigma\Delta2end'
2	'f\sigma\Delta2end'
3	'b\sigma\Delta3end'



	Name
4	'\sigma\Delta3end'

p =

GraphPlot with properties:

```

NodeColor: [16x3 double]
MarkerSize: 10
Marker: {'square' 'o' 'o' 'o' 'o' 'o' 'o' 'o' 'o' 'o' 'o' 'o' 'o' 'o' 'o' 'o'}
EdgeColor: [0 0.4470 0.7410]
LineWidth: 0.5000
LineStyle: '- '
NodeLabel: {1x16 cell}
EdgeLabel: {}
XData: [2 2 5 1 2 5 1 2 3 4 5 6 2 1 2 3]
YData: [6 5 5 5 4 4 3 3 3 3 3 3 2 1 1 1]
ZData: [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]

```

Show all properties

Синие узлы - промежуточные взаимодействия

Красные - пути, которые далее ведут к событиям второго порядка малости ( $\Delta^2$ )

Зеленые узлы - те, которые приводят к выходу излучения на поверхность из добавленного слоя и дают вклад первого порядка.

Зеленый квадратик "0" - узел, соответствующий акту падения излучения на добавленный слой.

Таким образом, пути от зеленого квадратика к любому из зеленых кружочков дадут нам вклад первого порядка в отражение.

Матлаб позволяет анализировать пути в графах между узлами. Нас интересуют пути между **real\_leafNodes** зелеными кружочками и стартовой точкой.

```

pat = digitsPattern(1,2)+":";
disp("Ближайшие пути обратно на поверхность:")

```

Ближайшие пути обратно на поверхность:

```

for i =1:height(real_leafNodes)
    ppath = shortestpath(G,"0",real_leafNodes{i,1}{1}); % ищет пути между узлами
    out_str = join(ppath,[' ',char(8594),' ']);
    out_str = replace(out_str,pat,"");
    out_str = strrep(out_str, '\sigma', char(963)); % sigma  $\sigma$ 
    out_str = strrep(out_str, '\Delta', char(916)); % delta  $\Delta$ 
    disp(out_str)
end

```

```

0 → bσΔ
0 → 1-σΔ → R → 1-σΔ
0 → 1-σΔ → R → fσΔ
0 → fσΔ → R → 1-σΔ
0 → 1-σΔ → R → bσΔ → R → 1-σΔ

```

Просуммируем все полученные ветки (только надо выкинуть те, которые дают второй порядок по  $\Delta$ )

$$R(X + \Delta) = b\sigma\Delta + (1 - \sigma\Delta) \cdot R \cdot (1 - \sigma\Delta) + (1 - \sigma\Delta) \cdot R \cdot f\sigma\Delta + f\sigma\Delta \cdot R \cdot (1 - \sigma\Delta) + (1 - \sigma\Delta) \cdot R \cdot b\sigma\Delta \cdot R \cdot (1 - \sigma\Delta)$$

После простых преобразований, а также занулив все члены второго порядка получим выражение для коэффициента отражения "расширенного" слоя через коэффициент отражения исходного слоя:

$$R(X + \Delta) = R(X) + \sigma\Delta[b + 2(f - 1)R(X) + R^2(X)]$$

В форме дифференциального уравнения:

$$\frac{d}{dx} R(x) = \sigma[b + 2(f - 1)R(x) + R(x)^2]$$

Какое граничное условие? Если исследуемый слой полностью пропадет, то отражение от него будет равно нулю.

$$R(0) = 0$$

Аналогично можно сформулировать уравнение для пропускания (оно складывается из прохождения без изменений и прохождения излучения отраженного от слоя и рассеянного назад в направлении распространения):

$$\frac{d}{dx} T(x) = \sigma[(f - 1) + bR(x)]T(x)$$

Граничное условие для пропускания соответственно

$$T(0) = 1$$

Сравним одну и ту же задачу в постановке инвариантного подгружения и классической (локальной):

$$\left\{ \begin{array}{l} \frac{dU}{dx} = \sigma[(f - 1)U(x) + bV(x)] \\ \frac{dV}{dx} = \sigma[-bU(x) - (f - 1)V(x)] \\ V(X) = 1 \\ U(0) = 0 \end{array} \right.$$

И принципа инвариантности:

$$\left\{ \begin{array}{l} \frac{dR(x)}{dx} = \sigma\Delta[b + 2(f - 1)R(x) + R(x)^2] \\ \frac{dT(x)}{dx} = \sigma[(f - 1) + bR(x)]T(x) \\ R(0) = 0 \\ T(0) = 1 \end{array} \right.$$

С одной стороны, кажется, что уравнение для коэффициентов отражения и пропускания хуже, чем для потоков влево-вправо, так как оно нелинейно (в первом стоит квадрат отражения, во втором - произведение отражения и пропускания (это уравнение Рикатти, которое, вообще говоря, можно решить аналитически при помощи определенной подстановки)).

Но обратим внимание на граничные условия, первая система - это краевая задача слева от слоя нулевой координате мы знаем только поток, идущий вправо (слева на слой ничего не падает) а справа мы знаем поток, идущий влево (падает излучение). Вторая система - это принципиально другой и значительно более простой для решения тип дифференциального уравнения - задача с начальными условиями (задача Коши)!

Решение задачи с начальными условиями в матлаб. Как правило задачи с начальными условиями описывают эволюцию процессов во времени от некоторого начального состояния до нужного (сам диффур решается, например, методом Рунге-Кутты)

Задача с начальными условиями:

$$\frac{d}{dx} \vec{Y} = f(x, \vec{Y})$$

$$\vec{Y}(x) = \vec{Y}_0 - \text{начальные условия.}$$

В нашем случае:

$$\frac{d}{dx} \vec{Y} = \begin{bmatrix} \sigma \Delta [b + 2(f-1)Y(1) + Y(1)^2] \\ \sigma [(f-1) + bY(1)]Y(2) \end{bmatrix}; \quad \vec{Y} = \begin{bmatrix} R \\ T \end{bmatrix}$$

```
function fun = dRdx(sigma,alfa,b,f)
% производная функции отражения
    fun = @(x,R) sigma*(b + 2*(f-1)*R + R.*R*b) - 2*alfa*R; % в отличие от формул в тексте добавлено
    также поглощение в слое
end
function fun = dTdx(sigma,alfa,b,f)
% производная функции пропускания
    fun = @(x,R,T) (sigma*((f-1) + b*R)-alfa).*T;
end
function fun = iem(sigma,alfa,b,f)
% функция для векторной формы дифференциального уравнения для решателя
    funR = dRdx(sigma,alfa,b,f);
    funT = dTdx(sigma,alfa,b,f);
    fun = @(x,Y) [funR(x,Y(1));
        funT(x,Y(1),Y(2))];
end
```

```
sigma=6.04
```

```
sigma = 6.0400
```

```
% коэффициент f
forward_fraction = 0.45
```

```
forward_fraction = 0.4500
```

```
alfa = 0.819
```

```
alfa = 0.8190
```

```
% коэффициент b в уравнении
```

```
backward_fraction = 1 - forward_fraction
```

```
backward_fraction = 0.5500
```

```
%dR_fun = dRdx(sigma,alfa,backward_fraction,forward_fraction)
```

```
iem_fun = iem(sigma,alfa,backward_fraction,forward_fraction)
```

```
iem_fun = function_handle with value:
```

```
@(x,Y)[funR(x,Y(1));funT(x,Y(1),Y(2))]
```

```
F = ode; % создаем объект решателя
```

```
F.ODEFcn = iem_fun; %dR_fun;
```

```
F.InitialValue = [0;1];% начальные условия
```

```
sol_inv = solve(F,0,1) % решаем
```

```
sol_inv =
```

```
ODEResults with properties:
```

```
Time: [0 1.5123e-05 3.0245e-05 4.5368e-05 6.0491e-05 1.3610e-04 2.1172e-04 2.8733e-04 3.6295e-04 7.4101e-04
```

```
Solution: [2x61 double]
```

```
r_dir = zeros(size(sol_inv.Time));
```

```
t_dir = ones(size(sol_inv.Time));
```

```
for i =2:numel(sol_inv.Time)
```

```
    [r_dir(i),t_dir(i)] =
```

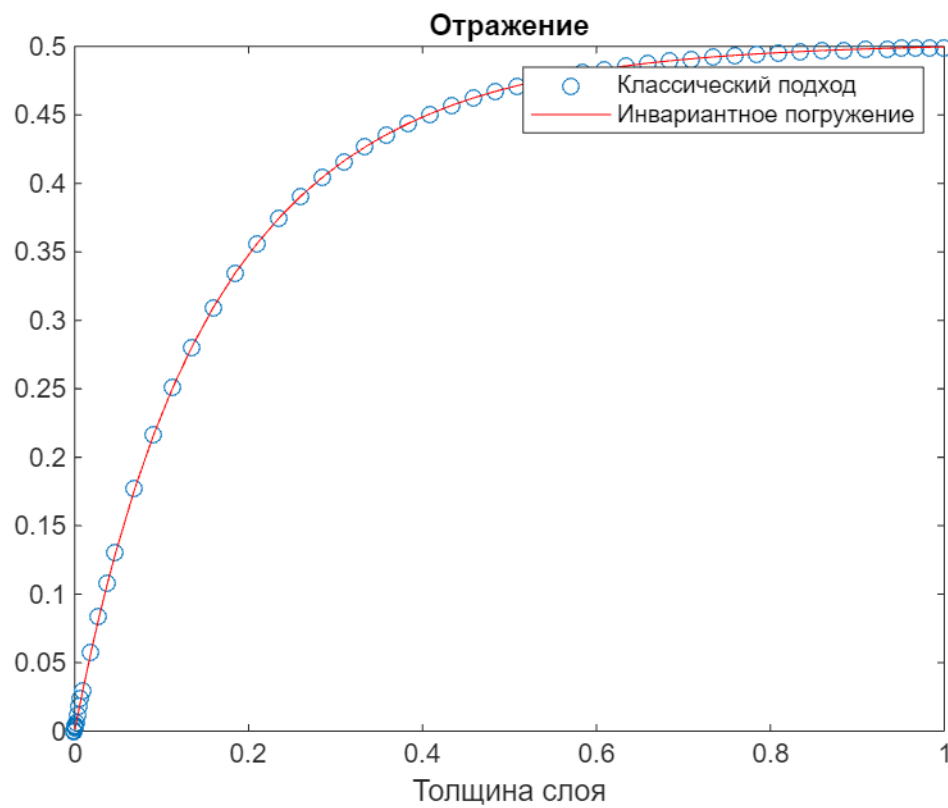
```
    r_direct(sol_inv.Time(i),alfa,sigma,forward_fraction,backward_fraction);
```

```
end
```

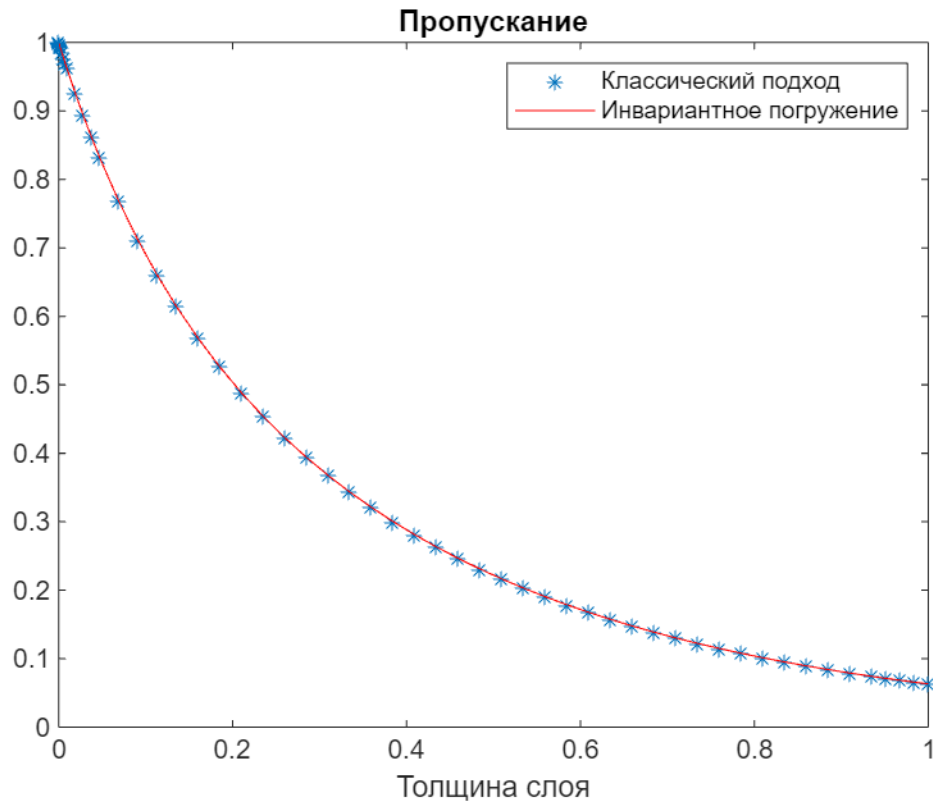
```
plot(sol_inv.Time,r_dir,'o',sol_inv.Time,sol_inv.Solution(1,:), 'r')
```

```
legend(["Классический подход", "Инвариантное погружение"])
```

```
title("Отражение");xlabel("Толщина слоя")
```



```
plot(sol_inv.Time,t_dir,'*',sol_inv.Time,sol_inv.Solution(2,:), 'r')  
legend(["Классический подход", "Инвариантное погружение"])  
title("Пропускание");xlabel("Толщина слоя")
```



#### Аналитическое решение уравнения инвариантного погружения для функции отражения.

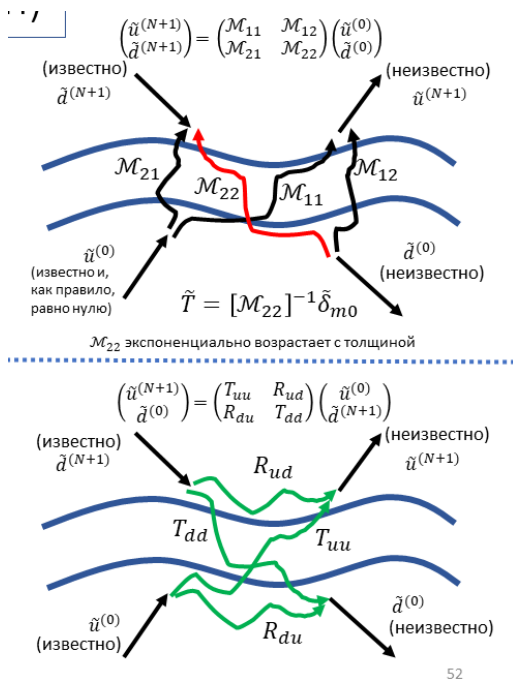
$$\frac{d}{dx} R(x) = \sigma b + 2[\sigma(f - 1) - \alpha]R(x) + R(x)^2 \quad (\text{здесь добавлено еще поглощение } \alpha - \text{поглощение на единице длины пути})$$

Следует отметить, что при выводе мы нигде не ограничивали материальные функции среды, поэтому все коэффициенты в уравнении могут зависеть от координаты.

Это уравнение общего вида:

$$\frac{d}{dx} Y(x) = A(x) + B(x)Y(x) + C(x)Y(x)^2$$

Хотел бы обратить внимание на глубокую связь между идеей принципа инвариантности и метода матриц рассеяния, который применяется для составления эффективных алгоритмов решения задачи рассеяния на многослойках гомогенных слоев и многослойных дифракционных решетках.



## Метод S-матриц (forward propagation)

$$\begin{pmatrix} \tilde{u}^{(l+1)} \\ \tilde{d}^{(0)} \end{pmatrix} = \begin{pmatrix} T_{uu}^{(l)} & R_{ud}^{(l)} \\ R_{du}^{(l)} & T_{dd}^{(l)} \end{pmatrix} \begin{pmatrix} \tilde{u}^{(0)} \\ \tilde{d}^{(l+1)} \end{pmatrix}$$

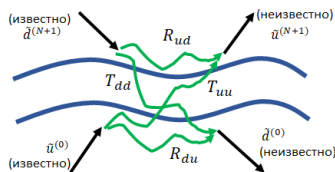
$$\Psi_{(l+1)l}^{-1} = [\Im_{21}^{(l+1)l} R_{ud}^{(l-1)} + \Im_{22}^{(l+1)l}]^{-1}$$

$$R_{ud}^{(l)} = [\Im_{11}^{(l+1)l} R_{ud}^{(l-1)} + \Im_{12}^{(l+1)l}] \Psi_{(l+1)l}^{-1} \text{ - зависит только от } R_{ud}^{(l-1)}$$

$$T_{dd}^{(l)} = T_{dd}^{(l-1)} \Psi_{(l+1)l}^{-1} \text{ - зависит от } R_{ud}^{(l-1)} \text{ и } T_{dd}^{(l-1)}$$

$$T_{uu}^{(l)} = \Im_{11}^{(l+1)l} - R_{ud}^{(l)} T_{uu}^{(l-1)} \text{ - зависит от } R_{ud}^{(l-1)} \text{ и } T_{uu}^{(l-1)}$$

$$R_{du}^{(l)} = [R_{du}^{(l-1)} - T_{dd}^{(l-1)} \Psi_{(l+1)l}^{-1} \Im_{21}^{(l+1)l} T_{uu}^{(l-1)}] \text{ - зависит от } R_{ud}^{(l-1)}, T_{uu}^{(l-1)} \text{ и } T_{dd}^{(l-1)}$$



Идея метода с-матриц в том, чтобы связь между падающим полем (на картинке выше стрелочки направлены к объекту) и рассеянным полем (стрелочки от объекта). В отличие от классического матричного метода, для которого ищется связь между полями идущими вверх и вниз на картинке в каждом слое.

```

%% задача
function s = fval(N)
% прямое суммирование
a = 3;
s = sqrt(3);
for i = 1:N
    s = sqrt(s + a);
end
end
function s = fval_rec(s,i,N)

```

```

% суммирование через рекурсию
a=3;
if i>=N
    return
else
    s= sqrt(a + fval_rec(s,i+1,N));
end
end
%% последовательность фибоначчи
function val = fib_dir(N)
% процедурное решение
    if N<=2
        val = 1;
        return
    end
    val = 0;
    x1 = 1;
    x2 = 1;
    for i = 3:N
        val = x1 + x2;
        x1 = x2;
        x2 = val;
    end
end
function val = fib_rec(N)
% рекурсивное решение
    if N<=2
        val = 1;
        return
    end
    val = fib_rec(N-1) + fib_rec(N-2);
end
function plotFibTreeGraph(n)
% для построения графа используем рекурсию в процедурном
% стиле
edges = [];
labels = string([]);
nodeCount = 0;

% рекурсивный вызов (как в простой версии
% рекурсивной функции, только каждый нод добавляется в граф)
% nested function (переменные edges,labels и nodeCount) для нее
% глобальные
function nodeID = buildFibGraph(num)
    nodeCount = nodeCount + 1;
    nodeID = nodeCount;
    val = fib_mem(num);
    labels(nodeID) = "#" + string(num) + "=" + string(val);
    if num <= 2
        return;
    end
end

```



```

    end
    % рекурсивный вызов
    leftID = buildFibGraph(num - 1);
    rightID = buildFibGraph(num - 2);

    edges(end+1, :) = [nodeID, leftID];
    edges(end+1, :) = [nodeID, rightID];
end

% запуск расчета
buildFibGraph(n);

% строим направленный граф
G = digraph(edges(:,1), edges(:,2));

% рисуем граф
figure;
h = plot(G, 'Layout', 'layered', 'NodeLabel', labels, 'Direction','down', ...
    'Interpreter','none', 'NodeFontSize', 10, 'MarkerSize', 7);
title(sprintf('Фибоначчи n = %d', n));
set(gca,'XTick',[], 'YTick', []);
end

%% Решение уравнения переноса в двух-потокном приближении

function fun = dy(sigma,alfa,b,f)
% функция оборачивает конкретные значения параметров в анонимную функцию,
% которая будет использована в решателе краевых задач
T = zeros(2);
T(1,1) = sigma*(f-1) - alfa;
T(1,2) = sigma*b;
T(2,1) = -T(1,2);
T(2,2) = -sigma*(f-1)+alfa;
fun = @(x,y)T*y(:);% первый аргумент - независимая переменная
% второй аргумент - само значение функции
end

function res = bcfun(y0,yX)
% граничные условия для решения краевой задачи
% для классической постановки
res = [y0(1)
    yX(2)-1];
end

function g = guess(x)
% стартовая аппроксимация
g = [0.5*x
    x];
end

function [r,t] = r_direct(X,alfa,sigma,f,b)
    dy_fun = dy(sigma,alfa,b,f);
    xmesh = linspace(0,X,10);

```

```

        solinit = bvpinit(xmesh, @guess);
        sol = bvp4c(dy_fun, @bcfun, solinit);
        t = sol.y(2,1);
        r = sol.y(1,end);
end
function fun = dRdx(sigma,alfa,b,f)
    fun = @(x,R) sigma*(b + 2*(f-1)*R + R.*R*b) - 2*alfa*R;
end
function fun = dTdx(sigma,alfa,b,f)
    fun = @(x,R,T) (sigma*((f-1) + b*R)-alfa).*T;
end
function fun = iem(sigma,alfa,b,f)
    funR = dRdx(sigma,alfa,b,f);
    funT = dTdx(sigma,alfa,b,f);
    fun = @(x,Y) [funR(x,Y(1));
        funT(x,Y(1),Y(2))];
end

```