Лабораторная работа №8

Компьютерный практикум по статистическому анализу данных

Николаев Дмитрий Иванович

Содержание

Список иллюстраций

Список таблиц

1 Цель работы

Основная цель работа — освоить пакеты Julia для решения задач оптимизации.

2 Выполнение лабораторной работы

Под оптимизацией в математике и информатике понимается решение задачи нахождения экстремума (минимума или максимума) целевой функции в некоторой области конечномерного векторного пространства, ограниченной набором линейных и/или нелинейных равенств и/или неравенств.

Оптимизационной задачей называется задача определения наилучших с точки зрения структуры или значений параметров объектов.

2.1 Повторение примеров

Повторим примеры, представленные в лабораторной работе ([1]).

2.1.1 Линейное программирование

Линейное программирование рассматривает решения экстремальных задач на множествах n–мерного векторного пространства, задаваемых системами линейных уравнений и неравенств.

Общей (стандартной) задачей линейного программирования называется задача нахождения минимума линейной целевой функции вида:

$$f(\vec{x}) = \sum_{j=1}^{n} c_j x_j,$$

где $\vec{c}$ — некоторые коэффициенты, $\vec{x} \in \mathbb{R}^n$.

Основной задачей линейного программирования называется задача, в которой есть ограничения в форме неравенств:

$$\sum_{j=1}^{n} a_{ij} x_j \geq b_i, \quad i = 1, 2, \dots, m, \quad x_j \geq 0, \quad j = 1, 2, \dots, n.$$

Задачи линейного программирования со смешанными ограничениями, такими как равенства и неравенства, с наличием переменных, свободных от ограничений, могут быть сведены к эквивалентным с тем же множеством решений путём замены переменных и замены равенств на пару неравенств.

В Julia есть несколько средств, предназначенных для решения оптимизационных задач. Одним из таких средств является JuMP — язык моделирования и вспомогательные пакеты для формулирования и решения задач математической оптимизации в Julia.

JuMP включает пакет `Convex.jl`, позволяющий описать задачу оптимизации, используя естественный математический синтаксис, и решать её с помощью одного из решателей (COSMO, ECOS, SCS, GLPK, MathOptInterface).

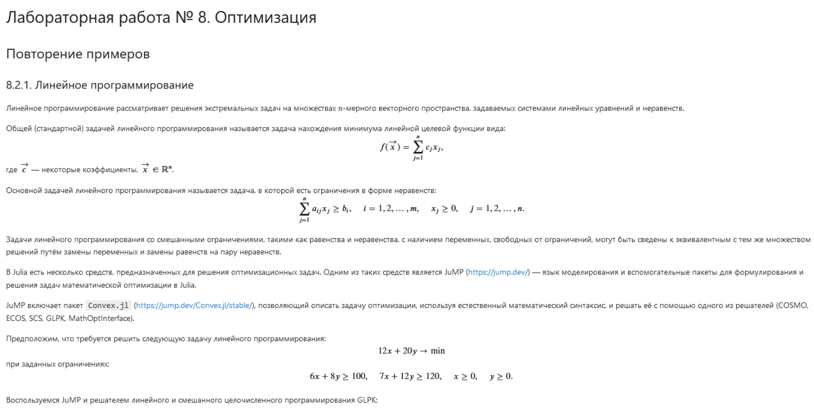
Предположим, что требуется решить следующую задачу линейного программирования:

$$12x + 20y \to \min$$

при заданных ограничениях:

$$6x + 8y \geq 100, \quad 7x + 12y \geq 120, \quad x \geq 0, \quad y \geq 0.$$

Решим задачу линейного программирования ([2.1-2.4]).



Лабораторная работа № 8. Оптимизация

Повторение примеров

8.2.1. Линейное программирование

Линейное программирование рассматривает решения экстремальных задач на множествах n-мерного векторного пространства, задаваемых системами линейных уравнений и неравенств.

Общей (стандартной) задачей линейного программирования называется задача нахождения минимума линейной целевой функции вида:

$$f(\vec{x}) = \sum_{j=1}^{n} c_j x_j,$$

где $\vec{c}$ — некоторые коэффициенты, $\vec{x} \in \mathbb{R}^n$.

Основной задачей линейного программирования называется задача, в которой есть ограничения в форме неравенств:

$$\sum_{j=1}^{n} a_{ij} x_j \geq b_i, \quad i = 1, 2, \dots, m, \quad x_j \geq 0, \quad j = 1, 2, \dots, n.$$

Задачи линейного программирования со смешанными ограничениями, такими как равенства и неравенства, с наличием переменных, свободных от ограничений, могут быть сведены к эквивалентным с тем же множеством решений путём замены переменных и замены равенств на пару неравенств.

В Julia есть несколько средств, предназначенных для решения оптимизационных задач. Одним из таких средств является JuMP (https://jump.dev/) — язык моделирования и вспомогательные пакеты для формулирования и решения задач математической оптимизации в Julia.

JuMP включает пакет `Convex.jl` (https://jump.dev/Convex.jl/stable/), позволяющий описать задачу оптимизации, используя естественный математический синтаксис, и решать её с помощью одного из решателей (COSMO, ECOS, SCS, GLPK, MathOptInterface).

Предположим, что требуется решить следующую задачу линейного программирования:

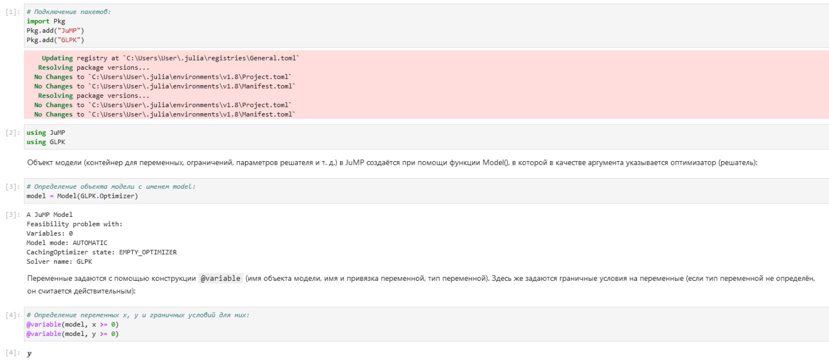
$$12x + 20y \to \min$$

при заданных ограничениях:

$$6x + 8y \geq 100, \quad 7x + 12y \geq 120, \quad x \geq 0, \quad y \geq 0.$$

Воспользуемся JuMP и решателем линейного и смешанного целочисленного программирования GLPK:

Рис. 2.1: Линейное программирование (1)



```
[1]: # Подключение пакетов:
import Pkg
Pkg.add("JuMP")
Pkg.add("GLPK")
```

```
    Updating registry at `C:\Users\User\.julia\registries\General.toml`
   Resolving package versions...
  No Changes to `C:\Users\User\.julia\environments\v1.8\Project.toml`
  No Changes to `C:\Users\User\.julia\environments\v1.8\Manifest.toml`
   Resolving package versions...
  No Changes to `C:\Users\User\.julia\environments\v1.8\Project.toml`
  No Changes to `C:\Users\User\.julia\environments\v1.8\Manifest.toml`
```

```
[2]: using JuMP
using GLPK
```

Объект модели (контейнер для переменных, ограничений, параметров решателя и т. д.) в JuMP создаётся при помощи функции Model(), в которой в качестве аргумента указывается оптимизатор (решатель):

```
[3]: # Определение объекта модели с именем model:
model = Model(GLPK.Optimizer)
```

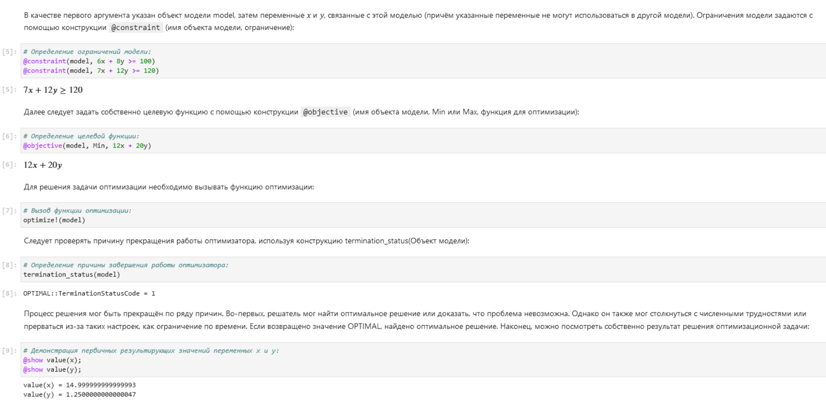
```
[3]: A JuMP Model
Feasibility problem with:
Variables: 0
Model mode: AUTOMATIC
CachingOptimizer state: EMPTY_OPTIMIZER
Solver name: GLPK
```

Переменные задаются с помощью конструкции `@variable` (имя объекта модели, имя и привязка переменной, тип переменной). Здесь же задаются граничные условия на переменные (если тип переменной не определён, он считается действительным):

```
[4]: # Определение переменных x, y и граничных условий для них:
@variable(model, x >= 0)
@variable(model, y >= 0)
```

[4]: y

Рис. 2.2: Линейное программирование (2)



В качестве первого аргумента указан объект модели model, затем переменные x и y, связанные с этой моделью (причём указанные переменные не могут использоваться в другой модели). Ограничения модели задаются с помощью конструкции `@constraint` (имя объекта модели, ограничение):

```
[5]: # Определение ограничений модели:
@constraint(model, 6x + 8y >= 100)
@constraint(model, 7x + 12y >= 120)
```

[5]: $7x + 12y \geq 120$

Далее следует задать собственно целевую функцию с помощью конструкции `@objective` (имя объекта модели, Min или Max, функция для оптимизации):

```
[6]: # Определение целевой функции:
@objective(model, Min, 12x + 20y)
```

[6]: $12x + 20y$

Для решения задачи оптимизации необходимо вызывать функцию оптимизации:

```
[7]: # Вызов функции оптимизации:
optimize!(model)
```

Следует проверять причину прекращения работы оптимизатора, используя конструкцию termination_status(Объект модели):

```
[8]: # Определение причины завершения работы оптимизатора:
termination_status(model)
```

```
[8]: OPTIMAL::TerminationStatusCode = 1
```

Процесс решения мог быть прекращён по ряду причин. Во-первых, решатель мог найти оптимальное решение или доказать, что проблема невозможна. Однако он также мог столкнуться с численными трудностями или прерваться из-за таких настроек, как ограничение по времени. Если возвращено значение OPTIMAL, найдено оптимальное решение. Наконец, можно посмотреть собственно результат решения оптимизационной задачи:

```
[9]: # Демонстрация первичных результирующих значений переменных x и y:
@show value(x);
@show value(y);
```

```
value(x) = 14.999999999999993
value(y) = 1.2500000000000047
```

Рис. 2.3: Линейное программирование (3)

```
[9]: # Демонстрация первичных результирующих значений переменных x и y:
@show value(x);
@show value(y);

value(x) = 14.999999999999993
value(y) = 1.2500000000000047

[10]: # Демонстрация результата оптимизации:
@show objective_value(model);

objective_value(model) = 205.0
```

Рис. 2.4: Линейное программирование (4)

2.1.2 Векторизованные ограничения и целевая функция оптимизации

Часто бывает полезно создавать коллекции переменных JuMP внутри более сложных структур данных. Можно добавить ограничения и цель в JuMP, используя векторизованную линейную алгебру.

Предположим, что требуется решить следующую задачу:

$$\vec{c}^T\vec{x} \to \min$$

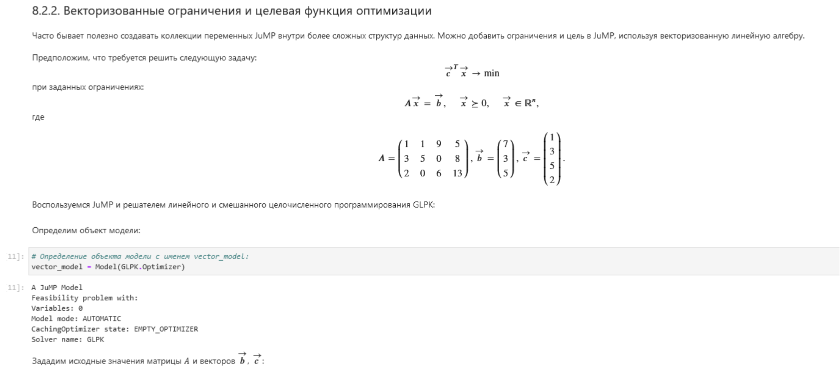
при заданных ограничениях:

$$A\vec{x} = \vec{b}, \quad \vec{x} \succeq 0, \quad \vec{x} \in \mathbb{R}^n,$$

где

$$A = \begin{pmatrix} 1 & 1 & 9 & 5 \\ 3 & 5 & 0 & 8 \\ 2 & 0 & 6 & 13 \end{pmatrix}, \vec{b} = \begin{pmatrix} 7 \\ 3 \\ 5 \end{pmatrix}, \vec{c} = \begin{pmatrix} 1 \\ 3 \\ 5 \\ 2 \end{pmatrix}.$$

Решим задачу линейного программирования, используя векторизированную линейную алгебру ([2.5-2.7]).



8.2.2. Векторизованные ограничения и целевая функция оптимизации

Часто бывает полезно создавать коллекции переменных JuMP внутри более сложных структур данных. Можно добавить ограничения и цель в JuMP, используя векторизованную линейную алгебру.

Предположим, что требуется решить следующую задачу:

$$\vec{c}^T \vec{x} \to \min$$

при заданных ограничениях:

$$A\vec{x} = \vec{b}, \quad \vec{x} \geq 0, \quad \vec{x} \in \mathbb{R}^n,$$

где

$$A = \begin{pmatrix} 1 & 1 & 9 & 5 \\ 3 & 5 & 0 & 8 \\ 2 & 0 & 6 & 13 \end{pmatrix}, \vec{b} = \begin{pmatrix} 7 \\ 3 \\ 5 \end{pmatrix}, \vec{c} = \begin{pmatrix} 1 \\ 3 \\ 5 \\ 2 \end{pmatrix}.$$

Воспользуемся JuMP и решателем линейного и смешанного целочисленного программирования GLPK:

Определим объект модели:

```
[11]: # Определение объекта модели с именем vector_model:
vector_model = Model(GLPK.Optimizer)
```

```
[11]: A JuMP Model
Feasibility problem with:
Variables: 0
Model mode: AUTOMATIC
CachingOptimizer state: EMPTY_OPTIMIZER
Solver name: GLPK
```

Зададим исходные значения матрицы A и векторов $\vec{b}$, $\vec{c}$:

Рис. 2.5: Векторизованные ограничения и целевая функция оптимизации (1)

Зададим исходные значения матрицы A и векторов $\vec{b}$, $\vec{c}$:

```
[12]: # Определение начальных данных:
A= [ 1 1 9 5;
3 5 0 8;
2 0 6 13]
```

```
[12]: 3×4 Matrix{Int64}:
 1  1  9   5
 3  5  0   8
 2  0  6  13
```

```
[13]: b = [7; 3; 5]
```

```
[13]: 3-element Vector{Int64}:
 7
 3
 5
```

```
[14]: c = [1; 3; 5; 2]
```

```
[14]: 4-element Vector{Int64}:
 1
 3
 5
 2
```

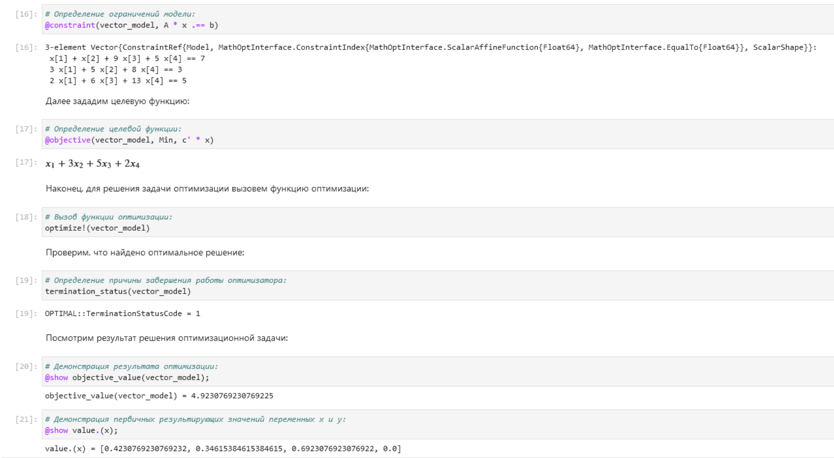
Далее зададим массив переменных для компонент вектора $\vec{x}$:

```
[15]: # Определение вектора переменных:
@variable(vector_model, x[1:4] >= 0)
```

```
[15]: 4-element Vector{VariableRef}:
 x[1]
 x[2]
 x[3]
 x[4]
```

Затем зададим ограничения в соответствии с условиями модели:

Рис. 2.6: Векторизованные ограничения и целевая функция оптимизации (2)



```
[16]: # Определение ограничений модели:
      @constraint(vector_model, A * x .== b)

[16]: 3-element Vector{ConstraintRef{Model, MathOptInterface.ConstraintIndex{MathOptInterface.ScalarAffineFunction{Float64}, MathOptInterface.EqualTo{Float64}}, ScalarShape}}:
       x[1] + x[2] + 9 x[3] + 5 x[4] == 7
       3 x[1] + 5 x[2] + 8 x[4] == 3
       2 x[1] + 6 x[3] + 13 x[4] == 5
```

Далее зададим целевую функцию:

```
[17]: # Определение целевой функции:
      @objective(vector_model, Min, c' * x)
```

[17]: $x_1 + 3x_2 + 5x_3 + 2x_4$

Наконец, для решения задачи оптимизации вызовем функцию оптимизации:

```
[18]: # Вызов функции оптимизации:
      optimize!(vector_model)
```

Проверим, что найдено оптимальное решение:

```
[19]: # Определение причины завершения работы оптимизатора:
      termination_status(vector_model)

[19]: OPTIMAL::TerminationStatusCode = 1
```

Посмотрим результат решения оптимизационной задачи:

```
[20]: # Демонстрация результата оптимизации:
      @show objective_value(vector_model);

      objective_value(vector_model) = 4.9230769230769225

[21]: # Демонстрация первичных результирующих значений переменных x и y:
      @show value.(x);

      value.(x) = [0.4230769230769232, 0.34615384615384615, 0.6923076923076922, 0.0]
```

Рис. 2.7: Векторизованные ограничения и целевая функция оптимизации (3)

2.1.3 Оптимизация рациона питания

В некоторых задачах требуется использование массивов, в которых индексы не являются целыми диапазонами с отсчётом от единицы. Например, требуется использовать переменную, индексируемую по названию продукта или местоположению. Тогда необходимо в качестве индекса использовать произвольный вектор. В Julia это можно реализовать с помощью DenseAxisArrays.

Рассмотрим применение DenseAxisArrays на примере решения задачи оптимизации рациона питании в заведении быстрого питания при условии, что задано ограничение на количество потребляемых калорий (1800–2200), белков (≥ 91), жиров (0–65) и соли (0–1779), а также перечень определённых продуктов питания с указанием их стоимости — гамбургер (2.49 ден.ед.), курица (2.89 ден.ед.), сосиска в тесте (1.50 ден.ед.), жареный картофель (1.89 ден.ед.), макароны (2.09 ден.ед.), пицца (1.99 ден.ед.), салат (2.49 ден.ед.), молочный коктейль (0.89 ден.ед.), мороженное (1.59 ден.ед.). Также известно содержание калорий, белков, жиров и соли в указанных продуктах.

Решим задачу оптимизации описанного выше рациона питания ([2.8-2.14]).

8.2.3. Оптимизация рациона питания

В некоторых задачах требуется использование массивов, в которых индексы не являются целыми диапазонами с отсчётом от единицы. Например, требуется использовать переменную, индексируемую по названию продукта или местоположению. Тогда необходимо в качестве индекса использовать произвольный вектор. В Julia это можно реализовать с помощью DenseAxisArrays.

Рассмотрим применение DenseAxisArrays на примере решения задачи оптимизации рациона питании в заведении быстрого питания при условии, что задано ограничение на количество потребляемых калорий (1800–2200), белков (≥ 91), жиров (0–65) и соли (0–1779), а также перечень определённых продуктов питания с указанием их стоимости — гамбургер (2.49 ден.ед.), курица (2.89 ден.ед.), сосиска в тесте (1.50 ден.ед.), жареный картофель (1.89 ден.ед.), макароны (2.09 ден.ед.), пицца (1.99 ден.ед.), салат (2.49 ден.ед.), молочный коктейль (0.89 ден.ед.), мороженное (1.59 ден.ед.). Также известно содержание калорий, белков, жиров и соли в указанных продуктах.

Содержание калорий, белков, жиров и соли в продуктах питания

продукт	калории	белки	жиры	соль
гамбургер	10	24	26	730
курица	420	32	10	1190
сосиска в тесте	560	20	32	1800
жареный картофель	380	4	19	270
макароны	320	12	10	930
пицца	320	15	12	820
салат	320	31	12	1230
молочный коктейль	100	8	2.5	125
мороженное	330	8	10	180

Воспользуемся JuMP и решателем линейного и смешанного целочисленного программирования GLPK :

Создадим контейнер JuMP для хранения информации об ограничениях (минимальное, максимальное) на количество потребляемых калорий, белков, жиров и соли:

Рис. 2.8: Оптимизация рациона питания (1)

```
[22]: # Контейнер для хранения данных об ограничениях на количество потребляемых калорий, белков, жиров и соли:
      category_data = JuMP.Containers.DenseAxisArray(
      [1800 2200;
      91 Inf;
      0 65;
      0 1779],
      ["calories", "protein", "fat", "sodium"],
      ["min", "max"])
```

```
[22]: 2-dimensional DenseAxisArray{Float64,2,...} with index sets:
          Dimension 1, ["calories", "protein", "fat", "sodium"]
          Dimension 2, ["min", "max"]
      And data, a 4×2 Matrix{Float64}:
       1800.0  2200.0
         91.0    Inf
          0.0    65.0
          0.0  1779.0
```

Введём массив данных с наименованиями продуктов:

```
[23]: # массив данных с наименованиями продуктов:
      foods = ["hamburger", "chicken", "hot dog", "fries", "macaroni", "pizza","salad", "milk", "ice cream"]
```

```
[23]: 9-element Vector{String}:
       "hamburger"
       "chicken"
       "hot dog"
       "fries"
       "macaroni"
       "pizza"
       "salad"
       "milk"
       "ice cream"
```

Рис. 2.9: Оптимизация рациона питания (2)

Введём данные о стоимости продуктов:

```
[24]: # Массив стоимости продуктов:
cost = JuMP.Containers.DenseAxisArray(
[2.49, 2.89, 1.50, 1.89, 2.09, 1.99, 2.49, 0.89, 1.59],
foods)
```

```
[24]: 1-dimensional DenseAxisArray{Float64,1,...} with index sets:
    Dimension 1, ["hamburger", "chicken", "hot dog", "fries", "macaroni", "pizza", "salad", "milk", "ice cream"]
And data, a 9-element Vector{Float64}:
 2.49
 2.89
 1.5
 1.89
 2.09
 1.99
 2.49
 0.89
 1.59
```

Рис. 2.10: Оптимизация рациона питания (3)

Введём сведения о содержании калорий, белков, жиров и соли в продуктах питания:

```
[25]: # Массив данных о содержании калорий, белков, жиров и соли в продуктах питания:
food_data = JuMP.Containers.DenseAxisArray(
[410 24 26 730;
420 32 10 1190;
560 20 32 1800;
380 4 19 270;
320 12 10 930;
320 15 12 820;
320 31 12 1230;
100 8 2.5 125;
330 8 10 180],
foods,
["calories", "protein", "fat", "sodium"])
```

```
[25]: 2-dimensional DenseAxisArray{Float64,2,...} with index sets:
    Dimension 1, ["hamburger", "chicken", "hot dog", "fries", "macaroni", "pizza", "salad", "milk", "ice cream"]
    Dimension 2, ["calories", "protein", "fat", "sodium"]
And data, a 9×4 Matrix{Float64}:
 410.0  24.0  26.0   730.0
 420.0  32.0  10.0  1190.0
 560.0  20.0  32.0  1800.0
 380.0   4.0  19.0   270.0
 320.0  12.0  10.0   930.0
 320.0  15.0  12.0   820.0
 320.0  31.0  12.0  1230.0
 100.0   8.0   2.5   125.0
 330.0   8.0  10.0   180.0
```

Определим объект модели:

```
[26]: # Определение объекта модели с именем model:
model = Model(GLPK.Optimizer)
```

```
[26]: A JuMP Model
Feasibility problem with:
Variables: 0
Model mode: AUTOMATIC
CachingOptimizer state: EMPTY_OPTIMIZER
Solver name: GLPK
```

Рис. 2.11: Оптимизация рациона питания (4)

Определим массив:

```
[27]: # Определим массив:
categories = ["calories", "protein", "fat", "sodium"]
```

```
[27]: 4-element Vector{String}:
 "calories"
 "protein"
 "fat"
 "sodium"
```

Далее зададим переменные:

```
[28]: # Определение переменных:
@variables(model, begin
    category_data[c, "min"] <= nutrition[c = categories] <= category_data[c, "max"]
    # Сколько покупать продуктов:
    buy[foods] >= 0
end)
```

```
[28]: (1-dimensional DenseAxisArray{VariableRef,1,...} with index sets:
    Dimension 1, ["calories", "protein", "fat", "sodium"]
And data, a 4-element Vector{VariableRef}:
 nutrition[calories]
 nutrition[protein]
 nutrition[fat]
 nutrition[sodium], 1-dimensional DenseAxisArray{VariableRef,1,...} with index sets:
    Dimension 1, ["hamburger", "chicken", "hot dog", "fries", "macaroni", "pizza", "salad", "milk", "ice cream"]
And data, a 9-element Vector{VariableRef}:
 buy[hamburger]
 buy[chicken]
 buy[hot dog]
 buy[fries]
 buy[macaroni]
 buy[pizza]
 buy[salad]
 buy[milk]
 buy[ice cream])
```

Рис. 2.12: Оптимизация рациона питания (5)

Задаём целевую функцию минимизации цены:

```
[29]: # Определение целевой функции:
@objective(model, Min, sum(cost[f] * buy[f] for f in foods))
```

[29]: $2.49 buy_{hamburger} + 2.89 buy_{chicken} + 1.5 buy_{hotdog} + 1.89 buy_{fries} + 2.09 buy_{macaroni} + 1.99 buy_{pizza} + 2.49 buy_{salad} + 0.89 buy_{milk} + 1.59 buy_{icecream}$

Затем зададим ограничения в соответствии с условиями модели:

```
[30]: # Определение ограничений модели:
@constraint(model, [c in categories],
sum(food_data[f, c] * buy[f] for f in foods) == nutrition[c])
```

```
[30]: 1-dimensional DenseAxisArray{ConstraintRef{Model, MathOptInterface.ConstraintIndex{MathOptInterface.ScalarAffineFunction{Float64}, MathOptInterface.EqualTo{Float64}}, ScalarShape},1,...} with index sets:
    Dimension 1, ["calories", "protein", "fat", "sodium"]
And data, a 4-element Vector{ConstraintRef{Model, MathOptInterface.ConstraintIndex{MathOptInterface.ScalarAffineFunction{Float64}, MathOptInterface.EqualTo{Float64}}, ScalarShape}}:
 -nutrition[calories] + 410 buy[hamburger] + 420 buy[chicken] + 560 buy[hot dog] + 380 buy[fries] + 320 buy[macaroni] + 320 buy[pizza] + 320 buy[salad] + 100 buy[milk] + 330 buy[ice cream] == 0
 -nutrition[protein] + 24 buy[hamburger] + 32 buy[chicken] + 20 buy[hot dog] + 4 buy[fries] + 12 buy[macaroni] + 15 buy[pizza] + 31 buy[salad] + 8 buy[milk] + 8 buy[ice cream] == 0
 -nutrition[fat] + 26 buy[hamburger] + 10 buy[chicken] + 32 buy[hot dog] + 19 buy[fries] + 10 buy[macaroni] + 12 buy[pizza] + 12 buy[salad] + 2.5 buy[milk] + 10 buy[ice cream] == 0
 -nutrition[sodium] + 730 buy[hamburger] + 1190 buy[chicken] + 1800 buy[hot dog] + 270 buy[fries] + 930 buy[macaroni] + 820 buy[pizza] + 1230 buy[salad] + 125 buy[milk] + 180 buy[ice cream] == 0
```

Рис. 2.13: Оптимизация рациона питания (6)

Наконец, для решения задачи оптимизации вызовем функцию оптимизации:

```
[31]: # Вызов функции оптимизации:
      JuMP.optimize!(model)
      term_status = JuMP.termination_status(model)
```

```
[31]: OPTIMAL::TerminationStatusCode = 1
```

Для просмотра результата решения модно вывести значение переменной buy:

```
[32]: hcat(buy.data, JuMP.value.(buy.data))
```

```
[32]: 9×2 Matrix{AffExpr}:
       buy[hamburger]  0.6045138888888888
       buy[chicken]    0
       buy[hot dog]    0
       buy[fries]      0
       buy[macaroni]   0
       buy[pizza]      0
       buy[salad]      0
       buy[milk]       6.9701388888888935
       buy[ice cream]  2.591319444444441
```

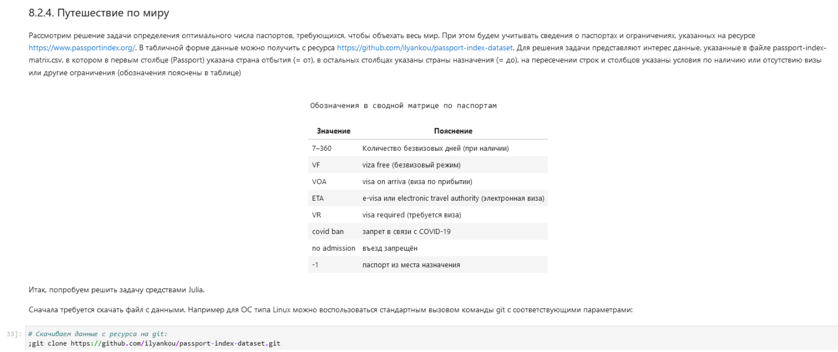
В результате оптимальным по цене и пищевой ценности будет предложено купить гамбургер, молочный коктейль и мороженное.

Рис. 2.14: Оптимизация рациона питания (7)

2.1.4 Путешествие по миру

Рассмотрим решение задачи определения оптимального числа паспортов, требующихся, чтобы объехать весь мир. При этом будем учитывать сведения о паспортах и ограничениях, указанных на ресурсе https://www.passportindex.org/. В табличной форме данные можно получить с ресурса https://github.com/ilyankou/passport-index-dataset. Для решения задачи представляют интерес данные, указанные в файле passport-index-matrix.csv, в котором в первым столбце (Passport) указана страна отбытия (= от), в остальных столбцах указаны страны назначения (= до), на пересечении строк и столбцов указаны условия по наличию или отсутствию визы или другие ограничения (обозначения пояснены в таблице)

Найдем наименьшее число паспортов, необходимых для путешествия по любой стране мира ([2.15-2.21]).



8.2.4. Путешествие по миру

Рассмотрим решение задачи определения оптимального числа паспортов, требующихся, чтобы объехать весь мир. При этом будем учитывать сведения о паспортах и ограничениях, указанных на ресурсе https://www.passportindex.org/. В табличной форме данные можно получить с ресурса https://github.com/ilyankou/passport-index-dataset. Для решения задачи представляют интерес данные, указанные в файле passport-index-matrix.csv, в котором в первым столбце (Passport) указана страна отбытия (= от), в остальных столбцах указаны страны назначения (= до), на пересечении строк и столбцов указаны условия по наличию или отсутствию визы или другие ограничения (обозначения пояснены в таблице)

Обозначения в сводной матрице по паспортам

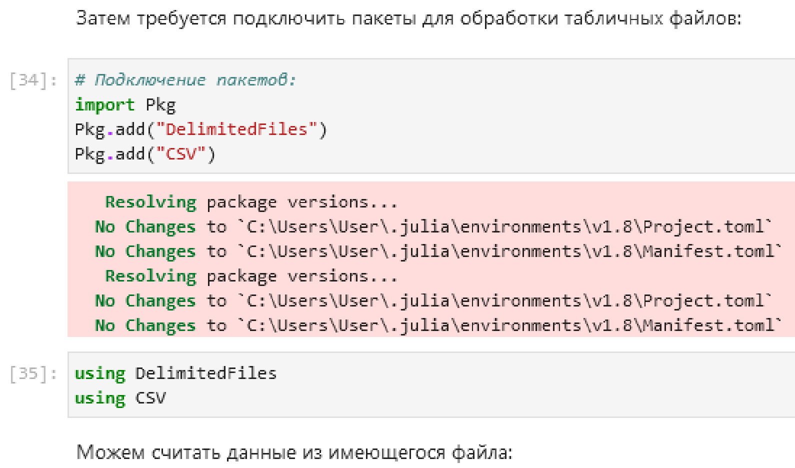
Значение	Пояснение
7–360	Количество безвизовых дней (при наличии)
VF	viza free (безвизовый режим)
VOA	visa on arriva (виза по прибытии)
ETA	e-visa или electronic travel authority (электронная виза)
VR	visa required (требуется виза)
covid ban	запрет в связи с COVID-19
no admission	въезд запрещён
-1	паспорт из места назначения

Итак, попробуем решить задачу средствами Julia.

Сначала требуется скачать файл с данными. Например для ОС типа Linux можно воспользоваться стандартным вызовом команды git с соответствующими параметрами:

```
33]: # Скачиваем данные с ресурса на git:
;git clone https://github.com/ilyankou/passport-index-dataset.git
```

Рис. 2.15: Путешествие по миру (1)



Затем требуется подключить пакеты для обработки табличных файлов:

```
[34]: # Подключение пакетов:
import Pkg
Pkg.add("DelimitedFiles")
Pkg.add("CSV")
```

```
   Resolving package versions...
  No Changes to `C:\Users\User\.julia\environments\v1.8\Project.toml`
  No Changes to `C:\Users\User\.julia\environments\v1.8\Manifest.toml`
   Resolving package versions...
  No Changes to `C:\Users\User\.julia\environments\v1.8\Project.toml`
  No Changes to `C:\Users\User\.julia\environments\v1.8\Manifest.toml`
```

```
[35]: using DelimitedFiles
using CSV
```

Можем считать данные из имеющегося файла:

Рис. 2.16: Путешествие по миру (2)

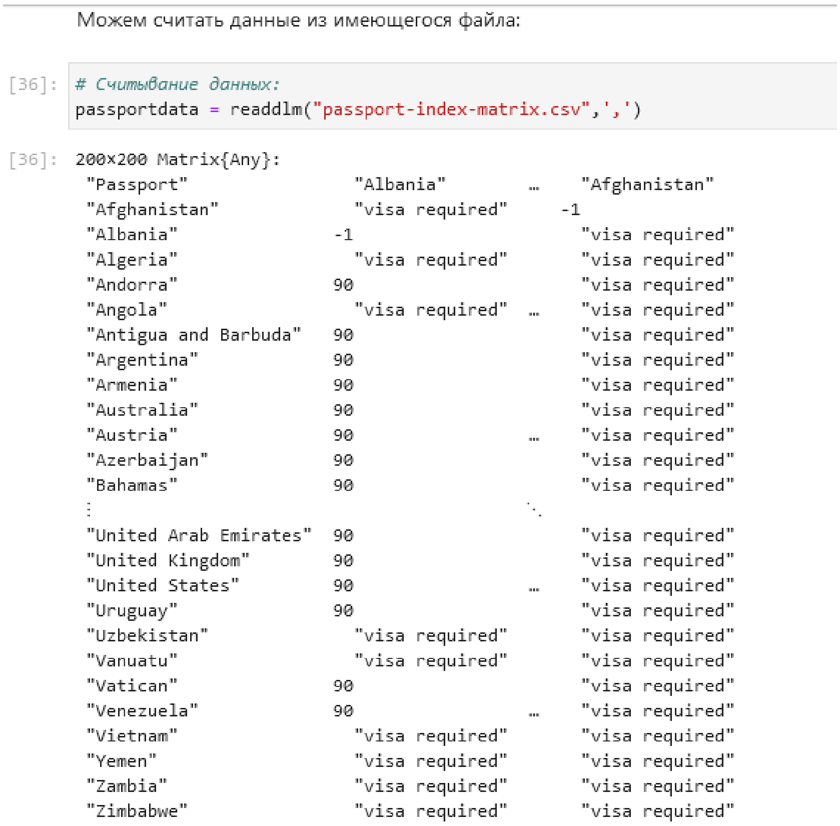


Рис. 2.17: Путешествие по миру (3)

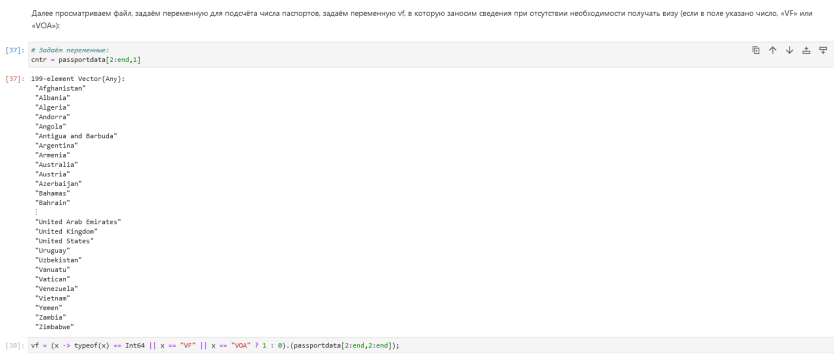


Рис. 2.18: Путешествие по миру (4)

Определяем объект модели:

```
[39]: # Определение объекта модели с именем model:
model = Model(GLPK.Optimizer)
```

```
[39]: A JuMP Model
Feasibility problem with:
Variables: 0
Model mode: AUTOMATIC
CachingOptimizer state: EMPTY_OPTIMIZER
Solver name: GLPK
```

Добавляем переменные, ограничения и целевую функцию:

```
[40]: # Переменные, ограничения и целевая функция:
@variable(model, pass[1:length(cntr)], Bin)
```

```
[40]: 199-element Vector{VariableRef}:
 pass[1]
 pass[2]
 pass[3]
 pass[4]
 pass[5]
 pass[6]
 pass[7]
 pass[8]
 pass[9]
 pass[10]
 pass[11]
 pass[12]
 pass[13]
 ⋮
 pass[188]
 pass[189]
 pass[190]
 pass[191]
 pass[192]
 pass[193]
 pass[194]
 pass[195]
 pass[196]
 pass[197]
 pass[198]
 pass[199]
```

Рис. 2.19: Путешествие по миру (5)

Рис. 2.20: Путешествие по миру (6)

Рис. 2.21: Путешествие по миру (7)

2.1.5 Портфельные инвестиции

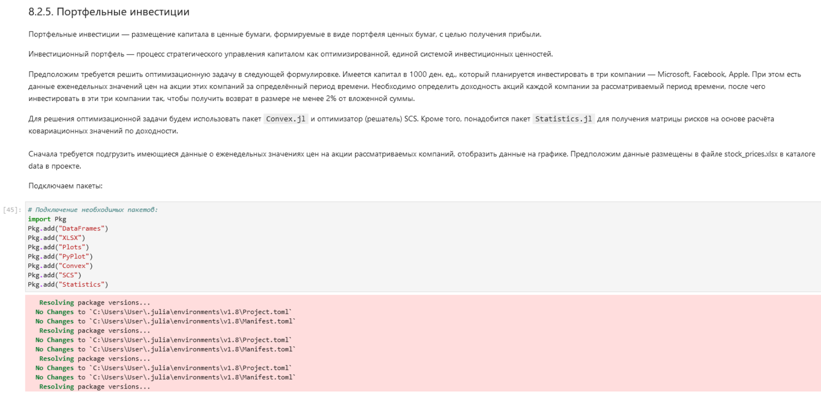
Портфельные инвестиции — размещение капитала в ценные бумаги, формируемые в виде портфеля ценных бумаг, с целью получения прибыли.

Инвестиционный портфель — процесс стратегического управления капиталом как оптимизированной, единой системой инвестиционных ценностей.

Предположим требуется решить оптимизационную задачу в следующей формулировке. Имеется капитал в 1000 ден. ед., который планируется инвестировать в три компании — Microsoft, Facebook, Apple. При этом есть данные еженедельных значений цен на акции этих компаний за определённый период времени. Необходимо определить доходность акций каждой компании за рассматриваемый период времени, после чего инвестировать в эти три компании так, чтобы получить возврат в размере не менее 2% от вложенной суммы.

Для решения оптимизационной задачи будем использовать пакет `Convex.jl` и оптимизатор (решатель) SCS. Кроме того, понадобится пакет `Statistics.jl` для получения матрицы рисков на основе расчёта ковариационных значений по доходности.

Найдем оптимальное число инвестиций в ценные бумаги, при получении хотя бы 2% возврата ([2.22-2.31]).



8.2.5. Портфельные инвестиции

Портфельные инвестиции — размещение капитала в ценные бумаги, формируемые в виде портфеля ценных бумаг, с целью получения прибыли.

Инвестиционный портфель — процесс стратегического управления капиталом как оптимизированной, единой системой инвестиционных ценностей.

Предположим требуется решить оптимизационную задачу в следующей формулировке. Имеется капитал в 1000 ден. ед., который планируется инвестировать в три компании — Microsoft, Facebook, Apple. При этом есть данные еженедельных значений цен на акции этих компаний за определённый период времени. Необходимо определить доходность акций каждой компании за рассматриваемый период времени, после чего инвестировать в эти три компании так, чтобы получить возврат в размере не менее 2% от вложенной суммы.

Для решения оптимизационной задачи будем использовать пакет `Convex.jl` и оптимизатор (решатель) SCS. Кроме того, понадобится пакет `Statistics.jl` для получения матрицы рисков на основе расчёта ковариационных значений по доходности.

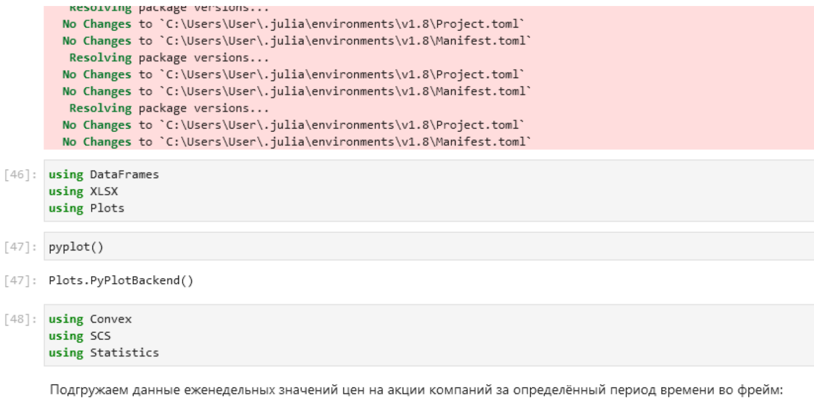
Сначала требуется подгрузить имеющиеся данные о еженедельных значениях цен на акции рассматриваемых компаний, отобразить данные на графике. Предположим данные размещены в файле stock_prices.xlsx в каталоге data в проекте.

Подключаем пакеты:

[45]:
```
# Подключение необходимых пакетов:
import Pkg
Pkg.add("DataFrames")
Pkg.add("XLSX")
Pkg.add("Plots")
Pkg.add("PyPlot")
Pkg.add("Convex")
Pkg.add("SCS")
Pkg.add("Statistics")
```

```
  Resolving package versions...
 No Changes to `C:\Users\User\.julia\environments\v1.8\Project.toml`
 No Changes to `C:\Users\User\.julia\environments\v1.8\Manifest.toml`
  Resolving package versions...
 No Changes to `C:\Users\User\.julia\environments\v1.8\Project.toml`
 No Changes to `C:\Users\User\.julia\environments\v1.8\Manifest.toml`
  Resolving package versions...
 No Changes to `C:\Users\User\.julia\environments\v1.8\Project.toml`
 No Changes to `C:\Users\User\.julia\environments\v1.8\Manifest.toml`
  Resolving package versions...
```

Рис. 2.22: Портфельные инвестиции (1)



```
  Resolving package versions...
 No Changes to `C:\Users\User\.julia\environments\v1.8\Project.toml`
 No Changes to `C:\Users\User\.julia\environments\v1.8\Manifest.toml`
  Resolving package versions...
 No Changes to `C:\Users\User\.julia\environments\v1.8\Project.toml`
 No Changes to `C:\Users\User\.julia\environments\v1.8\Manifest.toml`
  Resolving package versions...
 No Changes to `C:\Users\User\.julia\environments\v1.8\Project.toml`
 No Changes to `C:\Users\User\.julia\environments\v1.8\Manifest.toml`
```

[46]:
```
using DataFrames
using XLSX
using Plots
```

[47]:
```
pyplot()
```

[47]:
```
Plots.PyPlotBackend()
```

[48]:
```
using Convex
using SCS
using Statistics
```

Подгружаем данные еженедельных значений цен на акции компаний за определённый период времени во фрейм:

Рис. 2.23: Портфельные инвестиции (2)

Подгружаем данные еженедельных значений цен на акции компаний за определённый период времени во фрейм:

```
[49]: T = DataFrame(XLSX.readtable("stock_prices.xlsx","Sheet2"))
```

[49]: 13×3 DataFrame

Row	MSFT	FB	AAPL
	Any	Any	Any
1	101.93	137.95	148.26
2	102.8	143.8	152.29
3	107.71	150.04	156.82
4	107.17	149.01	157.76
5	102.78	165.71	166.52
6	105.67	167.33	170.41
7	108.22	162.5	170.42
8	110.97	161.89	172.97
9	112.53	162.28	174.97
10	110.51	169.6	172.91
11	115.91	165.98	186.12
12	117.05	164.34	191.05
13	117.94	166.69	189.95

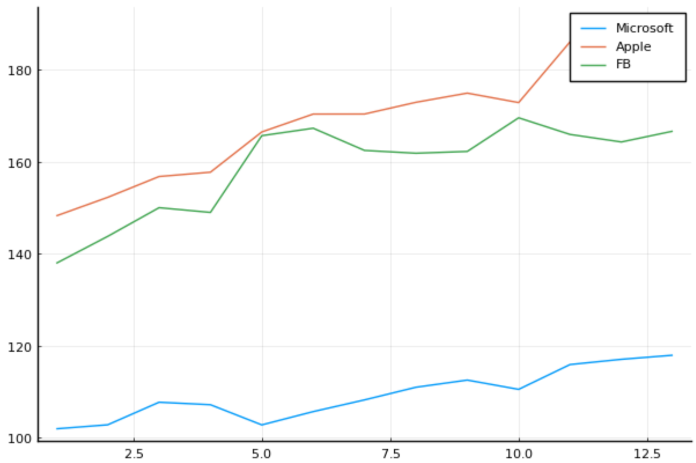
Отображаем данные на графике:

Рис. 2.24: Портфельные инвестиции (3)

Отображаем данные на графике:

```
[50]: # Построение графика:
plot(T[!,:MSFT],label="Microsoft")
plot!(T[!,:AAPL],label="Apple")
plot!(T[!,:FB],label="FB")
```

[50]:



Для дальнейших расчётов данные по ценам на акции переформируем из фрейма в матрицу:

Рис. 2.25: Портфельные инвестиции (4)

Для дальнейших расчётов данные по ценам на акции переформируем из фрейма в матрицу:

```
[51]: # Данные о ценах на акции размещаем в матрице:
      prices_matrix = Matrix(T)

[51]: 13×3 Matrix{Any}:
       101.93  137.95  148.26
       102.8   143.8   152.29
       107.71  150.04  156.82
       107.17  149.01  157.76
       102.78  165.71  166.52
       105.67  167.33  170.41
       108.22  162.5   170.42
       110.97  161.89  172.97
       112.53  162.28  174.97
       110.51  169.6   172.91
       115.91  165.98  186.12
       117.05  164.34  191.05
       117.94  166.69  189.95
```

Доходность акций i-й компании за период времени t определяется формулой: $R(i,t) = \frac{\mathrm{pr}(i,t) - \mathrm{pr}(i,t-1))}{\mathrm{pr}(i,t-1)}$, где $\mathrm{pr}(i,t)$ — цена акций i-й компании за период времени t:

```
[52]: # Вычисление матрицы доходности за период времени:
      M1 = prices_matrix[1:end-1,:]

[52]: 12×3 Matrix{Any}:
       101.93  137.95  148.26
       102.8   143.8   152.29
       107.71  150.04  156.82
       107.17  149.01  157.76
       102.78  165.71  166.52
       105.67  167.33  170.41
       108.22  162.5   170.42
       110.97  161.89  172.97
       112.53  162.28  174.97
       110.51  169.6   172.91
       115.91  165.98  186.12
       117.05  164.34  191.05
```

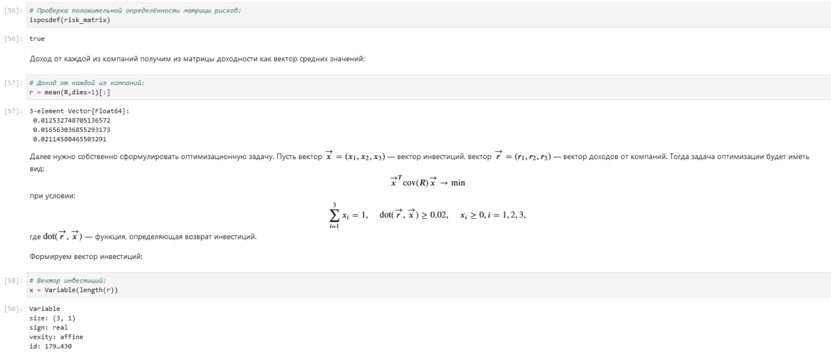
Рис. 2.26: Портфельные инвестиции (5)

```
[53]: M2 = prices_matrix[2:end,:]

[53]: 12×3 Matrix{Any}:
       102.8   143.8   152.29
       107.71  150.04  156.82
       107.17  149.01  157.76
       102.78  165.71  166.52
       105.67  167.33  170.41
       108.22  162.5   170.42
       110.97  161.89  172.97
       112.53  162.28  174.97
       110.51  169.6   172.91
       115.91  165.98  186.12
       117.05  164.34  191.05
       117.94  166.69  189.95

[54]: # Матрица доходности:
      R = (M2.-M1)./M1

[54]: 12×3 Matrix{Float64}:
        0.00853527   0.0424067    0.027182
        0.0477626    0.0433936    0.0297459
       -0.00501346  -0.00686484   0.00599413
       -0.040963     0.112073     0.0555274
        0.0281183    0.00977611   0.0233606
        0.0241317   -0.0288651    5.8682e-5
        0.0254112   -0.00375385   0.014963
        0.0140579    0.00240904   0.0115627
       -0.0179508    0.0451072   -0.0117734
        0.0488644   -0.0213443    0.0763981
        0.00983522  -0.00988071   0.0264883
        0.00760359   0.0142996   -0.00575766
```

Далее необходимо сформировать матрицу рисков — ковариационную матрицу рассчитанных цен доходности:

```
[55]: # Матрица рисков:
      risk_matrix = cov(R)

[55]: 3×3 Matrix{Float64}:
        0.000659383  -0.000630653  0.000139112
       -0.000630653   0.00152162   0.000192288
        0.000139112   0.000192288  0.000635503
```

Рис. 2.27: Портфельные инвестиции (6)



```
[56]: # Проверка положительной определённости матрицы рисков:
isposdef(risk_matrix)
```

```
[56]: true
```

Доход от каждой из компаний получим из матрицы доходности как вектор средних значений:

```
[57]: # Доход от каждой из компаний:
r = mean(R,dims=1)[:]
```

```
[57]: 3-element Vector{Float64}:
 0.012532748705136572
 0.016563036855293173
 0.02114580465503291
```

Далее нужно собственно сформулировать оптимизационную задачу. Пусть вектор $\vec{x} = (x_1, x_2, x_3)$ — вектор инвестиций, вектор $\vec{r} = (r_1, r_2, r_3)$ — вектор доходов от компаний. Тогда задача оптимизации будет иметь вид:

$$\vec{x}^T \operatorname{cov}(R) \vec{x} \to \min$$

при условии:

$$\sum_{i=1}^{3} x_i = 1, \quad \operatorname{dot}(\vec{r}, \vec{x}) \geq 0.02, \quad x_i \geq 0, i = 1, 2, 3,$$

где $\operatorname{dot}(\vec{r}, \vec{x})$ — функция, определяющая возврат инвестиций.

Формируем вектор инвестиций:

```
[58]: # Вектор инвестиций:
x = Variable(length(r))
```

```
[58]: Variable
size: (3, 1)
sign: real
vexity: affine
id: 179…430
```

Рис. 2.28: Портфельные инвестиции (7)

Определяем объект модели в соответствии с формулировкой оптимизационной задачи (при этом делаем задачу совместимой с DCP в соответствии с требованием пакета `Convex.jl` — см. http://cvxr.com/cvx/doc/dcp.html):

```
[59]: # Объект модели:
problem = minimize(Convex.quadform(x,risk_matrix),[sum(x)==1; r'*x >= 0.02; x.>=0])
```

```
[59]: minimize
└─ * (convex; positive)
   ├─ 1
   └─ qol_elem (convex; positive)
      ├─ norm2 (convex; positive)
      │  └─ …
      └─ [1.0;;]
subject to
├─ == constraint (affine)
│  ├─ sum (affine; real)
│  │  └─ 3-element real variable (id: 179…430)
│  └─ 1
├─ >= constraint (affine)
│  ├─ * (affine; real)
│  │  ├─ [0.0125327 0.016563 0.0211458]
│  │  └─ 3-element real variable (id: 179…430)
│  └─ 0.02
├─ >= constraint (affine)
│  ├─ index (affine; real)
│  │  └─ 3-element real variable (id: 179…430)
│  └─ 0
├─ >= constraint (affine)
│  ├─ index (affine; real)
│  │  └─ 3-element real variable (id: 179…430)
│  └─ 0
└─ >= constraint (affine)
   ├─ index (affine; real)
   │  └─ 3-element real variable (id: 179…430)
   └─ 0

status: `solve!` not called yet
```

Решаем поставленную задачу:

Рис. 2.29: Портфельные инвестиции (8)

Решаем поставленную задачу:

```
[60]: # Находим решение:
solve!(problem, SCS.Optimizer)
```

```
------------------------------------------------------------------
               SCS v3.2.4 - Splitting Conic Solver
        (c) Brendan O'Donoghue, Stanford University, 2012
------------------------------------------------------------------
problem:  variables n: 6, constraints m: 14
cones:    z: primal zero / dual free vars: 2
          l: linear vars: 5
          q: soc vars: 7, qsize: 2
settings: eps_abs: 1.0e-004, eps_rel: 1.0e-004, eps_infeas: 1.0e-007
          alpha: 1.50, scale: 1.00e-001, adaptive_scale: 1
          max_iters: 100000, normalize: 1, rho_x: 1.00e-006
          acceleration_lookback: 10, acceleration_interval: 10
lin-sys:  sparse-direct-amd-qdldl
          nnz(A): 24, nnz(P): 0
------------------------------------------------------------------
 iter | pri res | dua res |   gap   |   obj   |  scale  | time (s)
------------------------------------------------------------------
     0|1.71e+001 1.00e+000 1.62e+001 -8.03e+000 1.00e-001 2.63e-003
    75|8.16e-005 1.46e-004 5.60e-005 5.56e-004 1.00e-001 2.68e-003
------------------------------------------------------------------
status:  solved
timings: total: 2.68e-003s = setup: 2.59e-003s + solve: 8.86e-005s
         lin-sys: 1.97e-005s, cones: 1.43e-005s, accel: 2.40e-006s
------------------------------------------------------------------
objective = 0.000556
------------------------------------------------------------------
```

Выводим значения компонент вектора инвестиций:

Рис. 2.30: Портфельные инвестиции (9)

Выводим значения компонент вектора инвестиций:

```
[61]: x
```

```
[61]: Variable
size: (3, 1)
sign: real
vexity: affine
id: 179…430
value: [0.06922834751660402, 0.117301582202275, 0.8134695146542507]
```

Проверяем выполнение условия $\sum_{i=1}^{3} x_i = 1$:

```
[62]: sum(x.value)
```

```
[62]: 0.9999994443731297
```

Проверяем выполнение условия на уровень доходности от 2%:

```
[63]: r'*x.value
```

```
[63]: 1×1 adjoint(::Vector{Float64}) with eltype Float64:
 0.02001195936160116
```

Переводим процентные значения компонент вектора инвестиций в фактические денежные единицы:

```
[64]: x.value .* 1000
```

```
[64]: 3×1 Matrix{Float64}:
   69.22834751660402
  117.301582202275
  813.4695146542507
```

В результате, получаем: надо инвестировать 69.2 ден. ед. в Microsoft, 117.3 ден. ед. в Facebook, 813.5 ден. ед. в Apple.

Рис. 2.31: Портфельные инвестиции (10)

2.1.6 Восстановление изображения

Предположим есть изображение, на котором были изменены некоторые пиксели. Требуется восстановить неизвестные пиксели путём решения задачи оптимизации.

Восстановим изображения, решая задачу оптимизации ([2.32-2.40]).

8.2.6. Восстановление изображения

Предположим есть изображение, на котором были изменены некоторые пиксели. Требуется восстановить неизвестные пиксели путём решения задачи оптимизации.

Будем использовать пакет `Convex.jl` и оптимизатор (решатель) SCS, пакет `ImageMagick.jl` для работы с изображениями:

```
[65]: # Подключение необходимых пакетов:
import Pkg
Pkg.add("ImageMagick")
Pkg.add("Convex")
Pkg.add("SCS")
```

```
  Resolving package versions...
 No Changes to `C:\Users\User\.julia\environments\v1.8\Project.toml`
 No Changes to `C:\Users\User\.julia\environments\v1.8\Manifest.toml`
  Resolving package versions...
 No Changes to `C:\Users\User\.julia\environments\v1.8\Project.toml`
 No Changes to `C:\Users\User\.julia\environments\v1.8\Manifest.toml`
  Resolving package versions...
 No Changes to `C:\Users\User\.julia\environments\v1.8\Project.toml`
 No Changes to `C:\Users\User\.julia\environments\v1.8\Manifest.toml`
```

```
[66]: using ImageMagick
using Images
using Convex
using SCS
```

Рис. 2.32: Восстановление изображения (1)

Загрузим изображение для последующей обработки:

[67]:
```
# Считывание исходного изображения:
Kref = load("khiam-small.jpg")
```

[67]:

Преобразуем изображение в оттенки серого и испортим некоторые пиксели:

[68]:
```
K = copy(Kref)
p = prod(size(K))
```

[68]:
```
80089
```

Рис. 2.33: Восстановление изображения (2)

[69]:
```
missingids = rand(1:p, 400)
```

[69]:
```
400-element Vector{Int64}:
 36792
  3288
 59416
 54373
  1517
 10504
 56393
 22732
 10895
 42671
 21157
 54120
 55163
     ⋮
 65874
 10345
  6570
 56929
 36460
 77048
 22267
 58659
 51205
 38175
 79959
 30372
```

[70]:
```
K[missingids] .= RGBX{N0f8}(0.0,0.0,0.0)
```

[70]:

Рис. 2.34: Восстановление изображения (3)

[71]:

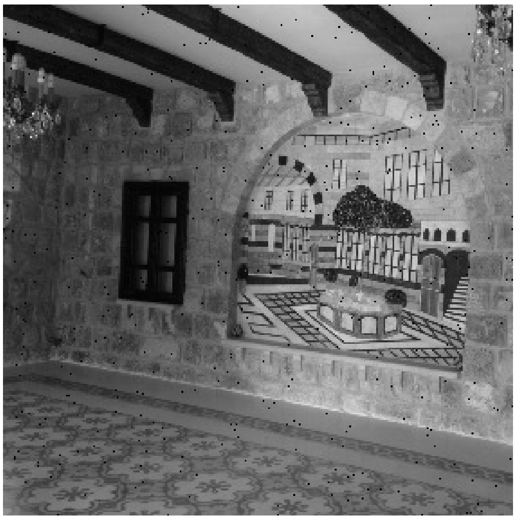
```
K
```

[71]:

[72]:

```
Gray.(K)
```

[72]:



Формируем матрицу со значениями цветов:

Рис. 2.35: Восстановление изображения (4)

Формируем матрицу со значениями цветов:

```
[73]: # Матрица цветов:
      Y = Float64.(Gray.(K))
```

```
[73]: 283×283 Matrix{Float64}:
 0.101961   0.0627451  0.0784314  0.0941176  …  0.509804  0.552941  0.666667
 0.0666667  0.0980392  0.0745098  0.054902      0.505882  0.584314  0.501961
 0.0784314  0.0862745  0.0784314  0.0901961     0.6       0.701961  0.615686
 0.0862745  0.0666667  0.0745098  0.0941176     0.658824  0.705882  0.145098
 0.0784314  0.101961   0.0901961  0.0745098     0.713725  0.682353  0.231373
 0.0745098  0.0745098  0.0784314  0.0862745  …  0.729412  0.701961  0.168627
 0.12549    0.0980392  0.0862745  0.0862745     0.627451  0.466667  0.192157
 0.439216   0.447059   0.305882   0.137255      0.231373  0.184314  0.137255
 0.458824   0.454902   0.45098    0.454902      0.196078  0.101961  0.117647
 0.45098    0.466667   0.458824   0.45098       0.584314  0.121569  0.137255
 0.458824   0.458824   0.458824   0.454902   …  0.521569  0.513725  0.12549
 0.466667   0.45098    0.458824   0.47451       0.576471  0.741176  0.117647
 0.45098    0.45098    0.462745   0.458824      0.560784  0.67451   0.117647
 ⋮                                           ⋱  ⋮
 0.494118   0.47451    0.47451    0.462745      0.427451  0.435294  0.443137
 0.47451    0.482353   0.470588   0.470588      0.439216  0.431373  0.431373
 0.494118   0.501961   0.470588   0.45098       0.447059  0.447059  0.45098
 0.470588   0.494118   0.490196   0.482353      0.431373  0.419608  0.419608
 0.458824   0.482353   0.47451    0.466667   …  0.454902  0.435294  0.423529
 0.443137   0.458824   0.45098    0.45098       0.309804  0.32549   0.34902
 0.47451    0.478431   0.462745   0.462745      0.341176  0.345098  0.360784
 0.482353   0.478431   0.458824   0.458824      0.423529  0.372549  0.321569
 0.552941   0.552941   0.541176   0.533333      0.447059  0.411765  0.372549
 0.552941   0.545098   0.576471   0.552941   …  0.435294  0.423529  0.407843
 0.564706   0.552941   0.54902    0.505882      0.439216  0.431373  0.419608
 0.568627   0.552941   0.517647   0.462745      0.439216  0.431373  0.427451
```

Далее необходимо сформировать новую матрицу X, в которой минимизируется норма ядра матрицы (т.е. сумма сингулярных чисел элементов матрицы) так, что элементы, которые уже известны в матрице Y, остаются теми же самыми в матрице X:

Рис. 2.36: Восстановление изображения (5)

```
[74]: correctids = findall(Y[:].!=0)

[74]: 79690-element Vector{Int64}:
          1
          2
          3
          4
          5
          6
          7
          8
          9
         10
         11
         12
         13
          ⋮
      80078
      80079
      80080
      80081
      80082
      80083
      80084
      80085
      80086
      80087
      80088
      80089

[75]: X = Convex.Variable(size(Y))

[75]: Variable
      size: (283, 283)
      sign: real
      vexity: affine
      id: 915…144
```

Рис. 2.37: Восстановление изображения (6)

```
[75]: X = Convex.Variable(size(Y))

[75]: Variable
      size: (283, 283)
      sign: real
      vexity: affine
      id: 915…144

[76]: problem = minimize(nuclearnorm(X))

[76]: minimize
      └─ nuclearnorm (convex; positive)
         └─ 283×283 real variable (id: 915…144)

      status: `solve!` not called yet

[77]: problem.constraints += X[correctids]==Y[correctids]

[77]: 1-element Vector{Constraint}:
       == constraint (affine)
      ├─ index (affine; real)
      │  └─ 283×283 real variable (id: 915…144)
      └─ 79690-element Vector{Float64}
```

Решаем поставленную задачу:

Рис. 2.38: Восстановление изображения (7)

Решаем поставленную задачу:

```
[78]: # Находим решение:
solve!(problem, SCS.Optimizer())
```

```
------------------------------------------------------------------
	       SCS v3.2.4 - Splitting Conic Solver
	(c) Brendan O'Donoghue, Stanford University, 2012
------------------------------------------------------------------
problem:  variables n: 240268, constraints m: 400047
cones: 	  z: primal zero / dual free vars: 239586
	  s: psd vars: 160461, ssize: 1
settings: eps_abs: 1.0e-004, eps_rel: 1.0e-004, eps_infeas: 1.0e-007
	  alpha: 1.50, scale: 1.00e-001, adaptive_scale: 1
	  max_iters: 100000, normalize: 1, rho_x: 1.00e-006
	  acceleration_lookback: 10, acceleration_interval: 10
lin-sys:  sparse-direct-amd-qdldl
	  nnz(A): 400330, nnz(P): 0
------------------------------------------------------------------
 iter | pri res | dua res |   gap   |   obj   |  scale  | time (s)
------------------------------------------------------------------
     0|1.50e+001 9.96e-001 8.34e+003 1.76e+002 1.00e-001 7.16e-001
   250|7.57e-004 2.82e-005 1.62e-005 4.46e+002 3.44e-001 6.14e+001
   350|3.25e-004 1.87e-005 3.16e-006 4.46e+002 3.44e-001 8.68e+001
------------------------------------------------------------------
status:  solved
timings: total: 8.68e+001s = setup: 4.16e-001s + solve: 8.64e+001s
	 lin-sys: 4.39e+000s, cones: 7.95e+001s, accel: 4.40e-001s
------------------------------------------------------------------
objective = 445.574978
------------------------------------------------------------------
```

Рис. 2.39: Восстановление изображения (8)

Выводим значение нормы и исправленное изображение:

```
[79]: @show norm(float.(Gray.(Kref))-X.value)
      @show norm(-X.value)

norm(float.(Gray.(Kref)) - X.value) = 1.208040379109865
norm(-(X.value)) = 124.3440619248571

[79]: 124.3440619248571
```

```
[80]: colorview(Gray, X.value)
```

[80]:

Рис. 2.40: Восстановление изображения (9)

2.2 Самостоятельная работа

2.2.1 Линейное программирование

Решим задачу линейного программирования:

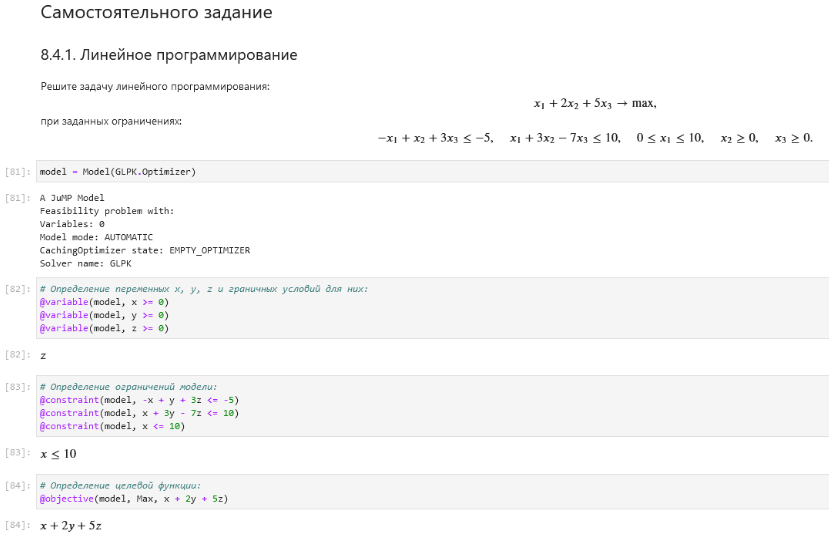
$$x_1 + 2x_2 + 5x_3 \to \max,$$

при заданных ограничениях:

$$-x_1+x_2+3x_3 \leq -5, \quad x_1+3x_2-7x_3 \leq 10, \quad 0 \leq x_1 \leq 10, \quad x_2 \geq 0, \quad x_3 \geq 0.$$

Найдем значения переменных x_1, x_2, x_3 в данной оптимизационной задаче

([2.41,2.42])



Самостоятельного задание

8.4.1. Линейное программирование

Решите задачу линейного программирования:

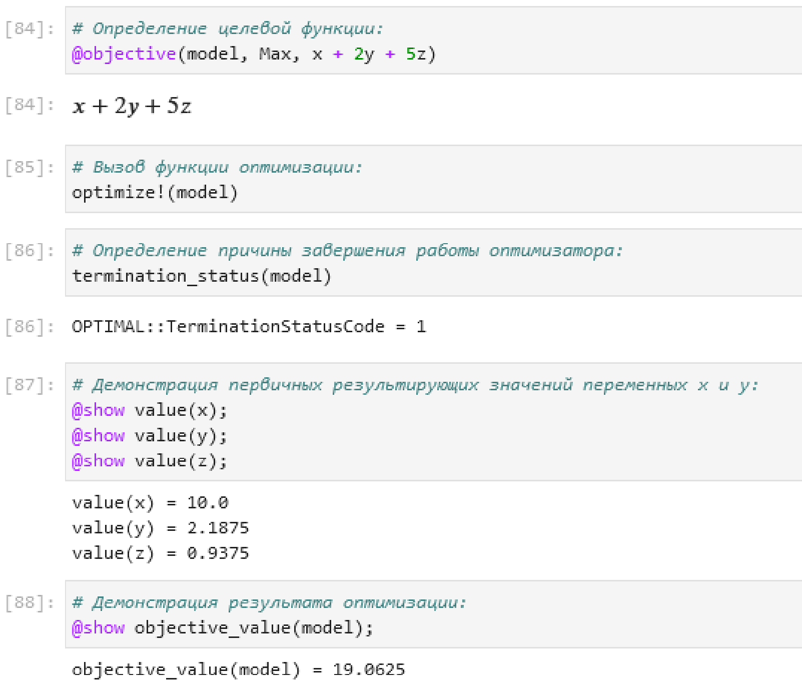
$$x_1 + 2x_2 + 5x_3 \to \max,$$

при заданных ограничениях:

$$-x_1 + x_2 + 3x_3 \le -5, \quad x_1 + 3x_2 - 7x_3 \le 10, \quad 0 \le x_1 \le 10, \quad x_2 \ge 0, \quad x_3 \ge 0.$$

```
[81]: model = Model(GLPK.Optimizer)

[81]: A JuMP Model
      Feasibility problem with:
      Variables: 0
      Model mode: AUTOMATIC
      CachingOptimizer state: EMPTY_OPTIMIZER
      Solver name: GLPK

[82]: # Определение переменных x, y, z и граничных условий для них:
      @variable(model, x >= 0)
      @variable(model, y >= 0)
      @variable(model, z >= 0)

[82]: z

[83]: # Определение ограничений модели:
      @constraint(model, -x + y + 3z <= -5)
      @constraint(model, x + 3y - 7z <= 10)
      @constraint(model, x <= 10)

[83]: x ≤ 10

[84]: # Определение целевой функции:
      @objective(model, Max, x + 2y + 5z)

[84]: x + 2y + 5z
```

Рис. 2.41: Задание 8.4.1. Линейное программирование (1)



```
[84]: # Определение целевой функции:
      @objective(model, Max, x + 2y + 5z)

[84]: x + 2y + 5z

[85]: # Вызов функции оптимизации:
      optimize!(model)

[86]: # Определение причины завершения работы оптимизатора:
      termination_status(model)

[86]: OPTIMAL::TerminationStatusCode = 1

[87]: # Демонстрация первичных результирующих значений переменных x и y:
      @show value(x);
      @show value(y);
      @show value(z);

      value(x) = 10.0
      value(y) = 2.1875
      value(z) = 0.9375

[88]: # Демонстрация результата оптимизации:
      @show objective_value(model);

      objective_value(model) = 19.0625
```

Рис. 2.42: Задание 8.4.1. Линейное программирование (2)

2.2.2 Линейное программирование. Использование массивов

Решим предыдущее задание, используя массивы вместо скалярных переменных ([2.43,2.44]).

8.4.2. Линейное программирование. Использование массивов

Решите предыдущее задание, используя массивы вместо скалярных переменных.

Рекомендация. Запишите систему ограничений в виде $A\vec{x} = \vec{b}$, а целевую функцию как $\vec{c}^T\vec{x}$.

```
[89]: # Определение объекта модели с именем vector_model:
      vector_model = Model(GLPK.Optimizer)

[89]: A JuMP Model
      Feasibility problem with:
      Variables: 0
      Model mode: AUTOMATIC
      CachingOptimizer state: EMPTY_OPTIMIZER
      Solver name: GLPK

[90]: # Определение начальных данных:
      A= [-1 1 3;
          1 3 -7;
          1 0 0]

[90]: 3×3 Matrix{Int64}:
       -1  1   3
        1  3  -7
        1  0   0

[91]: b = [-5; 10; 10]

[91]: 3-element Vector{Int64}:
       -5
       10
       10

[92]: c = [1; 2; 5]

[92]: 3-element Vector{Int64}:
       1
       2
       5
```

Рис. 2.43: Задание 8.4.2. Линейное программирование. Использование массивов (1)

```
[93]: # Определение вектора переменных:
      @variable(vector_model, x[1:3] >= 0)

[93]: 3-element Vector{VariableRef}:
       x[1]
       x[2]
       x[3]

[94]: # Определение ограничений модели:
      @constraint(vector_model, A * x .<= b)

[94]: 3-element Vector{ConstraintRef{Model, MathOptInterface.ConstraintIndex{MathOptInterface.ScalarAffineFunction{Float64}, MathOptInterface.LessThan{Float64}}, ScalarShape}}:
       -x[1] + x[2] + 3 x[3] <= -5
       x[1] + 3 x[2] - 7 x[3] <= 10
       x[1] <= 10

[95]: # Определение целевой функции:
      @objective(vector_model, Max, c' * x)

[95]: x1 + 2x2 + 5x3

[96]: # Вызов функции оптимизации:
      optimize!(vector_model)

[97]: # Определение причины завершения работы оптимизатора:
      termination_status(vector_model)

[97]: OPTIMAL::TerminationStatusCode = 1

[98]: @show value.(x);

      value.(x) = [10.0, 2.1875, 0.9375]

[99]: # Демонстрация результата оптимизации:
      @show objective_value(vector_model);

      objective_value(vector_model) = 19.0625

      Результаты совпали
```

Рис. 2.44: Задание 8.4.2. Линейное программирование. Использование массивов (2)

2.2.3 Выпуклое программирование

Решим задачу оптимизации:

$$||A\vec{x} - \vec{b}||_2^2 \to \min$$

при заданных ограничениях:

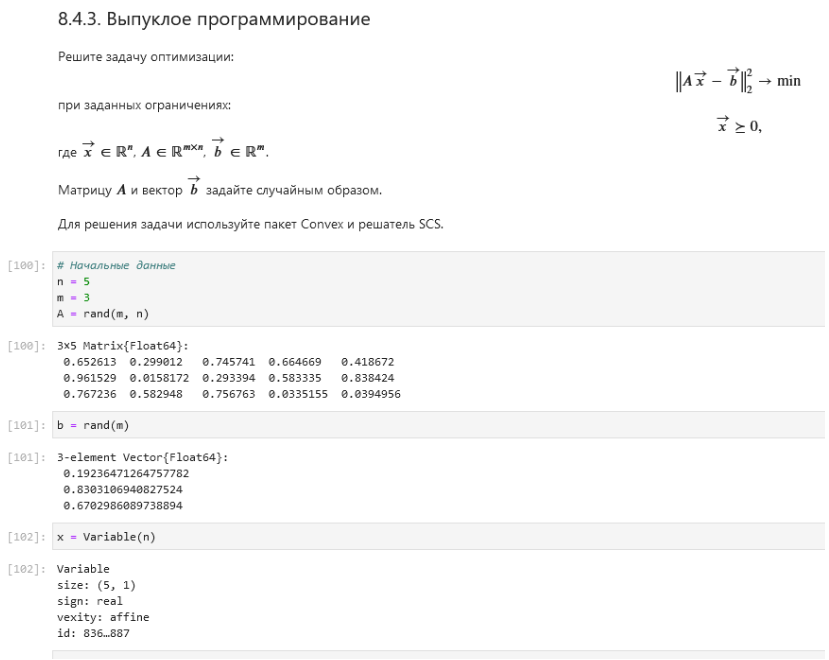
$$\vec{x} \succeq 0,$$

где $\vec{x} \in \mathbb{R}^n, A \in \mathbb{R}^{m \times n}, \vec{b} \in \mathbb{R}^m$.

Матрицу A и вектор $\vec{b}$ зададим случайным образом.

Для решения задачи используем пакет Convex и решатель SCS.

Найдем значения вектора $\vec{x}$ в данной оптимизационной задаче ([2.45-2.48])



8.4.3. Выпуклое программирование

Решите задачу оптимизации:

$$\left\| A\vec{x} - \vec{b} \right\|_2^2 \to \min$$

при заданных ограничениях:

$$\vec{x} \geq 0,$$

где $\vec{x} \in \mathbb{R}^n$, $A \in \mathbb{R}^{m \times n}$, $\vec{b} \in \mathbb{R}^m$.

Матрицу A и вектор $\vec{b}$ задайте случайным образом.

Для решения задачи используйте пакет Convex и решатель SCS.

```
[100]: # Начальные данные
n = 5
m = 3
A = rand(m, n)

[100]: 3×5 Matrix{Float64}:
 0.652613  0.299012   0.745741  0.664669   0.418672
 0.961529  0.0158172  0.293394  0.583335   0.838424
 0.767236  0.582948   0.756763  0.0335155  0.0394956

[101]: b = rand(m)

[101]: 3-element Vector{Float64}:
 0.19236471264757782
 0.8303106940827524
 0.6702986089738894

[102]: x = Variable(n)

[102]: Variable
size: (5, 1)
sign: real
vexity: affine
id: 836…887
```

Рис. 2.45: Задание 8.4.3. Выпуклое программирование (1)

```
[103]: I = zeros(m, m)
       for i ∈ 1:m
           I[i, i] = 1
       end
       I
```

```
[103]: 3×3 Matrix{Float64}:
        1.0  0.0  0.0
        0.0  1.0  0.0
        0.0  0.0  1.0
```

```
[104]: Y = A*x - b
```

```
[104]: + (affine; real)
       ├─ * (affine; real)
       │  ├─ 3×5 Matrix{Float64}
       │  └─ 5-element real variable (id: 836…887)
       └─ [-0.192365, -0.830311, -0.670299]
```

```
[105]: problem3 = minimize(Convex.quadform(Y, I))
```

```
[105]: minimize
       └─ * (convex; positive)
          ├─ 1
          └─ qol_elem (convex; positive)
             ├─ norm2 (convex; positive)
             │  └─ …
             └─ [1.0;;]

       status: `solve!` not called yet
```

Рис. 2.46: Задание 8.4.3. Выпуклое программирование (2)

```
[106]: problem3.constraints += x .>= 0
```

```
[106]: 5-element Vector{Constraint}:
        >= constraint (affine)
       ├─ index (affine; real)
       │  └─ 5-element real variable (id: 836…887)
       └─ 0
        >= constraint (affine)
       ├─ index (affine; real)
       │  └─ 5-element real variable (id: 836…887)
       └─ 0
        >= constraint (affine)
       ├─ index (affine; real)
       │  └─ 5-element real variable (id: 836…887)
       └─ 0
        >= constraint (affine)
       ├─ index (affine; real)
       │  └─ 5-element real variable (id: 836…887)
       └─ 0
        >= constraint (affine)
       ├─ index (affine; real)
       │  └─ 5-element real variable (id: 836…887)
       └─ 0
```

Рис. 2.47: Задание 8.4.3. Выпуклое программирование (3)

```
[107]: # Находим решение:
       solve!(problem3, SCS.Optimizer)
```

```
------------------------------------------------------------------
               SCS v3.2.4 - Splitting Conic Solver
        (c) Brendan O'Donoghue, Stanford University, 2012
------------------------------------------------------------------
problem:  variables n: 8, constraints m: 14
cones:    z: primal zero / dual free vars: 1
          l: linear vars: 6
          q: soc vars: 7, qsize: 2
settings: eps_abs: 1.0e-004, eps_rel: 1.0e-004, eps_infeas: 1.0e-007
          alpha: 1.50, scale: 1.00e-001, adaptive_scale: 1
          max_iters: 100000, normalize: 1, rho_x: 1.00e-006
          acceleration_lookback: 10, acceleration_interval: 10
lin-sys:  sparse-direct-amd-qdldl
          nnz(A): 26, nnz(P): 0
------------------------------------------------------------------
 iter | pri res | dua res |   gap   |   obj   |  scale  | time (s)
------------------------------------------------------------------
     0|1.71e+001 1.00e+000 1.62e+001 -8.04e+000 1.00e-001 6.40e-005
   125|2.48e-005 1.38e-005 4.18e-005 1.09e-001 6.28e-001 1.36e-004
------------------------------------------------------------------
status:  solved
timings: total: 1.37e-004s = setup: 5.11e-005s + solve: 8.56e-005s
         lin-sys: 2.87e-005s, cones: 1.32e-005s, accel: 4.70e-006s
------------------------------------------------------------------
objective = 0.109032
------------------------------------------------------------------
```

```
[108]: x
```

```
[108]: Variable
       size: (5, 1)
       sign: real
       vexity: affine
       id: 836…887
       value: [0.7416883143029624, -1.7272943063149415e-6, -6.629980264234386e-6, -7.2867904909678845e-6, -4.327451306140682e-7]
```

Решения почти удовлетворяют ограничениям (до 6 знака)

Рис. 2.48: Задание 8.4.3. Выпуклое программирование (4)

2.2.4 Оптимальная рассадка по залам

Проводится конференция с 5 разными секциями. Забронировано 5 залов различной вместимости: в каждом зале не должно быть меньше 180 и больше 250 человек, а на третьей секции активность подразумевает, что должно быть точно 220 человек.

В заявке участник указывает приоритет посещения секции: 1 — максимальный приоритет, 3 — минимальный, а значение 10000 означает, что человек не пойдёт на эту секцию.

Организаторам удалось собрать 1000 заявок с указанием приоритета посещения трёх секций. Необходимо дать рекомендацию слушателю, на какую же секцию ему пойти, чтобы хватило места всем.

Для решения задачи используем пакет Convex и решатель GLPK.

Приоритеты по слушателям распределим случайным образом.

Согласно заданию ([2.49]), будем считать, что приоритеты по секциям у участников, кроме отказа от секции, не могут повторяться ([2.50,2.51]). После чего зададим и решим оптимизационную задачу ([2.52-2.54]), найдя оптимальную рассадку по секциям ([2.55]).

8.4.4. Оптимальная рассадка по залам

Проводится конференция с 5 разными секциями. Забронировано 5 залов различной вместимости: в каждом зале не должно быть меньше 180 и больше 250 человек, а на третьей секции активность подразумевает, что должно быть точно 220 человек.

В заявке участник указывает приоритет посещения секции: 1 — максимальный приоритет, 3 — минимальный, а значение 10000 означает, что человек не пойдёт на эту секцию.

Организаторам удалось собрать 1000 заявок с указанием приоритета посещения трёх секций. Необходимо дать рекомендацию слушателю, на какую же секцию ему пойти, чтобы хватило места всем.

Для решения задачи используйте пакет Convex и решатель GLPK.

Приоритеты по слушателям распределите случайным образом.

Вектор значений $\vec{x} = (x_1, x_2, x_3, x_4, x_5)^T$ означает число человек в залах

$$180 \le \vec{x} \le 250, \quad x_3 = 220,$$

Матрица A размерности 1000×5 обозначает приоритет посещения секций людьми (так как приоритет по 3 секциям, то 2 столбца будут иметь значение 10000)

Рис. 2.49: Задание 8.4.4. Оптимальная рассадка по залам (1)

```
[177]: function Priority(Vector)
           A = zeros(length(Vector))
           counter = 1
           for v in Vector
               if counter <= 3
                   A[v] = counter
               else
                   A[v] = 10000
               end
               counter += 1
           end
           return A
       end

[177]: Priority (generic function with 1 method)

[178]: function Generate_Priorities(N)
           # N - число генераций заявок
           A = []
           for i ∈ 1:N
               t = [rand() for _ in 1:5]
               t_indexes = sortperm(t)
               if i == 1
                   A = Priority(t_indexes)'
               else
                   A = [A; Priority(t_indexes)']
               end
           end
           return A
       end

[178]: Generate_Priorities (generic function with 1 method)
```

Рис. 2.50: Задание 8.4.4. Оптимальная рассадка по залам (2)

```
[179]: A = Generate_Priorities(1000)

[179]: 1000×5 Matrix{Float64}:
     1.0  10000.0      3.0      2.0  10000.0
     3.0  10000.0      1.0  10000.0      2.0
     2.0  10000.0  10000.0      1.0      3.0
 10000.0      1.0      3.0      2.0  10000.0
 10000.0      2.0      1.0      3.0  10000.0
 10000.0      2.0  10000.0      1.0      3.0
     3.0      1.0  10000.0  10000.0      2.0
 10000.0      1.0      3.0  10000.0      2.0
     2.0      3.0      1.0  10000.0  10000.0
 10000.0      1.0      2.0  10000.0      3.0
     3.0      2.0  10000.0  10000.0      1.0
     1.0  10000.0  10000.0      2.0      3.0
 10000.0      1.0      2.0  10000.0      3.0
     ⋮
 10000.0      1.0  10000.0      2.0      3.0
 10000.0      1.0      3.0  10000.0      2.0
     3.0      1.0  10000.0  10000.0      2.0
 10000.0      2.0  10000.0      3.0      1.0
 10000.0      2.0      1.0      3.0  10000.0
 10000.0  10000.0      3.0      2.0      1.0
 10000.0      2.0  10000.0      1.0      3.0
 10000.0      2.0  10000.0      1.0      3.0
     1.0      2.0      3.0  10000.0  10000.0
     2.0  10000.0  10000.0      1.0      3.0
     3.0  10000.0  10000.0      2.0      1.0
 10000.0      2.0      1.0      3.0  10000.0

[180]: x = Variable(size(A), :Bin)

[180]: Variable
size: (1000, 5)
sign: real
vexity: affine
id: 180…299
```

Рис. 2.51: Задание 8.4.4. Оптимальная рассадка по залам (3)

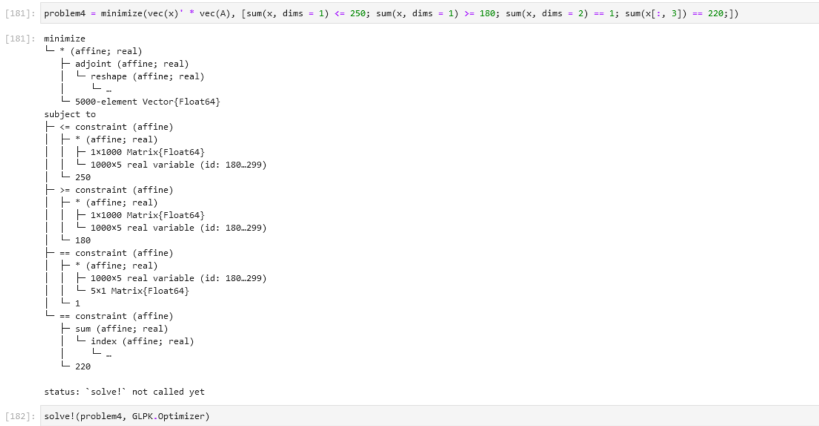


Рис. 2.52: Задание 8.4.4. Оптимальная рассадка по залам (4)

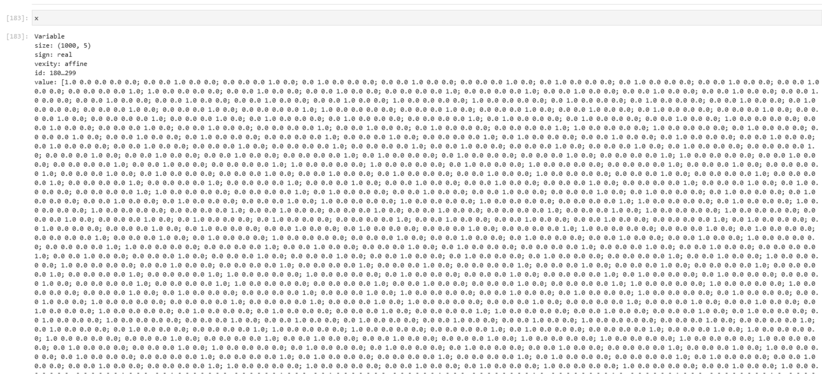


Рис. 2.53: Задание 8.4.4. Оптимальная рассадка по залам (5)

```
[184]: x.value
```

```
[184]: 1000×5 Matrix{Float64}:
 1.0  0.0  0.0  0.0  0.0
 0.0  0.0  1.0  0.0  0.0
 0.0  0.0  0.0  1.0  0.0
 0.0  1.0  0.0  0.0  0.0
 0.0  0.0  1.0  0.0  0.0
 0.0  0.0  0.0  1.0  0.0
 0.0  1.0  0.0  0.0  0.0
 0.0  1.0  0.0  0.0  0.0
 0.0  0.0  1.0  0.0  0.0
 0.0  0.0  1.0  0.0  0.0
 0.0  0.0  0.0  0.0  1.0
 1.0  0.0  0.0  0.0  0.0
 0.0  0.0  1.0  0.0  0.0
 ⋮
 0.0  1.0  0.0  0.0  0.0
 0.0  1.0  0.0  0.0  0.0
 0.0  1.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  1.0
 0.0  0.0  1.0  0.0  0.0
 0.0  0.0  0.0  0.0  1.0
 0.0  0.0  0.0  1.0  0.0
 0.0  0.0  0.0  1.0  0.0
 1.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  1.0  0.0
 0.0  0.0  0.0  0.0  1.0
 0.0  0.0  1.0  0.0  0.0
```

Рис. 2.54: Задание 8.4.4. Оптимальная рассадка по залам (6)

```
[185]: recommendations = [findfirst(x.value[i, :] .== 1.0) for i ∈ 1:size(A)[1]]

[185]: 1000-element Vector{Int64}:
 1
 3
 4
 2
 3
 4
 2
 2
 3
 3
 5
 1
 3
 ⋮
 2
 2
 2
 5
 3
 5
 4
 4
 1
 4
 5
 3

[186]: seat_distribution = [count(x -> x == j, recommendations) for j ∈ 1:5]

[186]: 5-element Vector{Int64}:
 190
 191
 220
 189
 210
```

Получили оптимальную рассадку по залам (согласно рекомендациям)

Рис. 2.55: Задание 8.4.4. Оптимальная рассадка по залам (7)

2.2.5 План приготовления кофе

Кофейня готовит два вида кофе «Раф кофе» за 400 рублей и «Капучино» за 300. Чтобы сварить 1 чашку «Раф кофе» необходимо: 40 гр. зёрен, 140 гр. молока и 5 гр. ванильного сахара. Для того чтобы получить одну чашку «Капучино» необходимо потратить: 30 гр. зёрен, 120 гр. молока. На складе есть: 500 гр. зёрен, 2000 гр. молока и 40 гр. ванильного сахара.

Необходимо найти план варки кофе, обеспечивающий максимальную выручку от их реализации. При этом необходимо потратить весь ванильный сахар.

Для решения задачи используем пакет JuMP и решатель GLPK.

Переформулируем задачу. Введем обозначения: x — число приготовленных "Раф кофе", y — число приготовленных "Каппучино". Тогда необходимо решить следующую оптимизационную задачу:

$$400x + 300y \to \max,$$

при заданных ограничениях

$$40x + 30y \leq 500, \quad 140x + 120y \leq 2000, \quad 5x = 40, \quad x \geq 0, \quad y \geq 0,$$

где первое ограничение обозначает массу затраченных зёрен, второе — массу затраченного молока, третье — ванильного сахара.

Теперь решим полученную задачу, найдя x и y ([2.56,2.57]).

8.4.5. План приготовления кофе

Кофейня готовит два вида кофе «Раф кофе» за 400 рублей и «Капучино» за 300. Чтобы сварить 1 чашку «Раф кофе» необходимо: 40 гр. зёрен, 140 гр. молока и 5 гр. ванильного сахара. Для того чтобы получить одну чашку «Капучино» необходимо потратить: 30 гр. зёрен, 120 гр. молока. На складе есть: 500 гр. зёрен, 2000 гр. молока и 40 гр. ванильного сахара.

Необходимо найти план варки кофе, обеспечивающий максимальную выручку от их реализации. При этом необходимо потратить весь ванильный сахар.

Для решения задачи используйте пакет JuMP и решатель GLPK.

$$400x + 300y \to \max,$$

где x - число приготовленных "Раф кофе", y - число приготовленных "Каппучино".

Граничные условия:

$$40x + 30y \leq 500,$$
$$140x + 120y \leq 2000,$$
$$5x = 40, \quad x \geq 0, \quad y \geq 0.$$

[116]:
```
# Определение объекта модели с именем model5:
model5 = Model(GLPK.Optimizer)
```

[116]:
```
A JuMP Model
Feasibility problem with:
Variables: 0
Model mode: AUTOMATIC
CachingOptimizer state: EMPTY_OPTIMIZER
Solver name: GLPK
```

[117]:
```
# Определение переменных:
@variable(model5, x >= 0)
@variable(model5, y >= 0)
```

[117]: y

Рис. 2.56: Задание 8.4.5. План приготовления кофе (1)

Рис. 2.57: Задание 8.4.5. План приготовления кофе (2)

3 Выводы

В ходе выполнения лабораторной работы я освоил пакеты Julia для решения задач оптимизации.

Список литературы

1. Королькова А. В., Кулябов Д. С. Лабораторная работа № 8. Оптимизация [Электронный ресурс]. RUDN, 2023. URL: https://esystem.rudn.ru/pluginfile.php/2231367/mod_resource/content/2/008-lab_optimization.pdf.