Лабораторная работа № 8. Оптимизация

8.1. Цель работы

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

8.2. Предварительные сведения

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.

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:

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

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:

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

Воспользуемся JuMP и решателем линейного и смешанного целочисленного программирования GLPK:

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

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

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):

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

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)`:

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

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

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

Для решения задачи оптимизации необходимо вызывать функцию оптимизации:

```
# Вызов функции оптимизации:
optimize!(model)
```

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

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

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:

```
# Демонстрация первичных результирующих значений переменных x и y:
@show value(x);
@show value(y);
# Демонстрация результата оптимизации:
@show objective_value(model);
```

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:

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

где

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

Воспользуемся JuMP и решателем линейного и смешанного целочисленного программирования GLPK:

```
# Подключение пакетов:
import Pkg
Pkg.add("JuMP")
Pkg.add("GLPK")
using JuMP
using GLPK
```

Определим объект модели:

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

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

```
# Определение начальных данных:
A= [ 1 1 9  5;
   3 5 0  8;
   2 0 6 13]
b = [7; 3; 5]
c = [1; 3; 5; 2]
```

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

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

Затем зададим ограничения в соответствии с условиями модели:

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

Далее зададим целевую функцию:

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

Наконец, для решения задачи оптимизации вызовем функцию оптимизации:

```
# Вызов функции оптимизации:
optimize!(vector_model)
```

Проверим, что найдено оптимальное решение:

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

Посмотрим результат решения оптимизационной задачи:

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

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 ($\geqslant 91$), 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.

Таблица 8.1

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 :

```
# Подключение пакетов:
import Pkg
Pkg.add("JuMP")
Pkg.add("GLPK")
using JuMP
using GLPK
```

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

```
# 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"])
```

Введём массив данных с наименованиями продуктов:

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

Введём данные о стоимости продуктов:

```
# 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)
```

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

```
# 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"])
```

Определим объект модели:

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

Определим массив:

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

Далее зададим переменные:

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

Задаём целевую функцию минимизации цены:

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

Затем зададим ограничения в соответствии с условиями модели:

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

Наконец, для решения задачи оптимизации вызовем функцию оптимизации:

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

Для просмотра результата решения модно вывести значение переменной buy:

```
hcat(buy.data,JuMP.value.(buy.data))
```

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

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 табл. 8.2).

Таблица 8.2

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:

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

Затем требуется подключить пакеты для обработки табличных файлов:

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

using DelimitedFiles
using CSV
```

Можем считать данные из имеющегося файла:

```
# Считывание данных:
passportdata = readdlm(joinpath("passport-index-dataset","passport-
↪  index-matrix.csv"),',')
```

Для решения задачи используем возможности пакета JuMP и решателя линейного и смешанного целочисленного программирования GLPK:

```
# Подключение пакетов:
Pkg.add("JuMP")
Pkg.add("GLPK")

using JuMP
using GLPK
```

Далее просматриваем файл, задаём переменную для подсчёта числа паспортов, задаём переменную vf, в которую заносим сведения при отсутствии необходимости получать визу (если в поле указано число, «VF» или «VOA»):

```
# Задаём переменные:
cntr = passportdata[2:end,1]
vf = (x ->  typeof(x)==Int64 || x == "VF" || x == "VOA" ? 1 :
↪  0).(passportdata[2:end,2:end]);
```

Определяем объект модели:

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

Добавляем переменные, ограничения и целевую функцию:

```
# Переменные, ограничения и целевая функция:
@variable(model, pass[1:length(cntr)], Bin)
@constraint(model, [j=1:length(cntr)], sum( vf[i,j]*pass[i] for i in
↪  1:length(cntr)) >= 1)
@objective(model, Min, sum(pass))
```

Для решения задачи оптимизации вызываем функцию оптимизации:

```
# Вызов функции оптимизации:
JuMP.optimize!(model)
termination_status(model)
```

Просматриваем результат:

```
# Просмотр результата:
print(JuMP.objective_value(model)," passports:
↪  ",join(cntr[findall(JuMP.value.(pass) .== 1)],", "))
```

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:

```
# 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")
using DataFrames
using XLSX
using Plots
pyplot()
using Convex
using SCS
using Statistics
```

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

```
# Считываем данные и размещаем их во фрейм:
T = DataFrame(XLSX.readtable("data/stock_prices.xlsx","Sheet2")...)
```

Отображаем данные на графике (рис. 8.1):

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

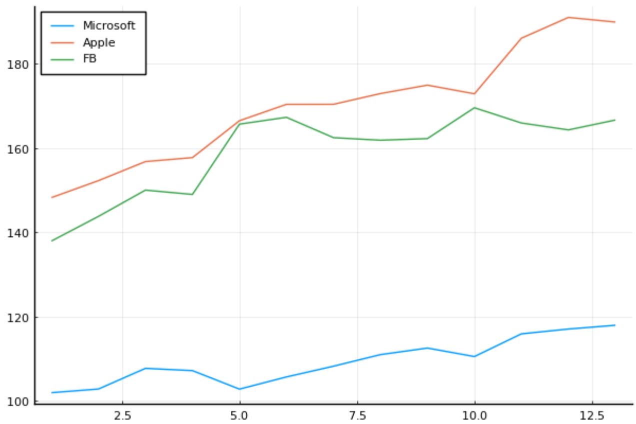

Рис. 8.1. Изменение цен на акции компаний за 13 дней

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

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

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

```
# Вычисление матрицы доходности за период времени:
M1 = prices_matrix[1:end-1,:]
M2 = prices_matrix[2:end,:]
# Матрица доходности:
R = (M2.-M1)./M1
```

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

```
# Матрица рисков:
risk_matrix = cov(R)
```

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

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

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

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:

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

при условии:

$$\sum_{i=1}^{3} x_i = 1, \quad \text{dot}(\vec{r}, \vec{x}) \geqslant 0,02, \quad x_i \geqslant 0,\ i = 1,2,3,$$

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

Формируем вектор инвестиций:

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

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`):

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

Решаем поставленную задачу:

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

Выводим значения компонент вектора инвестиций:

```
x
```

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

```
sum(x.value)
```

Проверяем выполнение условия на уровень доходности от 2%:

```
r'*x.value
```

Переводим процентные значения компонент вектора инвестиций в фактические денежные единицы:

```
x.value .* 1000
```

В результате, получаем: надо инвестировать 67.9 ден. ед. в Microsoft, 122.3 ден. ед. в Facebook, 809.7 ден. ед. в Apple.

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:

```
# Подключение необходимых пакетов:
import Pkg
Pkg.add("ImageMagick")
Pkg.add("Convex")
Pkg.add("SCS")
using Images
using Convex
using SCS
```

Загрузим изображение для последующей обработки (рис. 8.2):

```
# Считывание исходного изображения:
Kref = load("data/khiam-small.jpg")
```

Рис. 8.2. Исходное изображение

Преобразуем изображение в оттенки серого и испортим некоторые пиксели (рис. **??**):

```
K = copy(Kref)
p = prod(size(K))
missingids = rand(1:p,400)
```

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

Формируем матрицу со значениями цветов:

```
# Матрица цветов:
Y = Float64.(Gray.(K));
```

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:

```
correctids = findall(Y[:].!=0)
X = Convex.Variable(size(Y))
problem = minimize(nuclearnorm(X))
problem.constraints += X[correctids]==Y[correctids]
```

Решаем поставленную задачу:

```
# Находим решение:
solve!(problem, SCS.Optimizer(eps=1e-3, alpha=1.5))
```

Выводим значение нормы и исправленное изображение (рис. 8.3):

```
@show norm(float.(Gray.(Kref))-X.value)
@show norm(-X.value)
colorview(Gray, X.value)
```

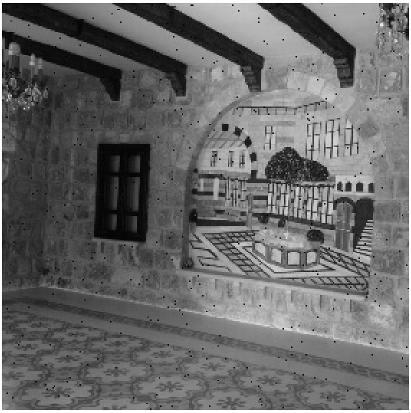

Рис. 8.3. Искусственно испорченное изображение в оттенках серого

Рис. 8.4. Исправленное изображение в оттенках серого

8.3. Задание

1. Используя Jupyter Lab, повторите примеры из раздела 8.2.
2. Выполните задания для самостоятельной работы (раздел 8.4).

8.4. Задания для самостоятельного выполнения

8.4.1. Линейное программирование

Решите задачу линейного программирования:

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

при заданных ограничениях:

$$-x_1 + x_2 + 3x_3 \leqslant -5, \quad x_1 + 3x_2 - 7x_3 \leqslant 10, \quad 0 \leqslant x_1 \leqslant 10, \quad x_2 \geqslant 0, \quad x_3 \geqslant 0.$$

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 $\vec{c}^T\vec{x}$.

8.4.3. Выпуклое программирование

Решите 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 $\vec{b}$ задайте случайным образом.

Для решения задачи используйте пакет `Convex` и решатель `SCS`.

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.

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`.

8.5. Содержание отчёта

1. Титульный лист с указанием номера лабораторной работы и ФИО студента.
2. Формулировка задания работы.
3. Описание выполнения задания:
   - подробное пояснение выполняемых в соответствии с заданием действий;
   - скриншоты (снимки экрана), фиксирующие выполнение лабораторной работы;
   - листинги (исходный код) программ и результаты его выполнения;
4. Выводы, согласованные с заданием работы.