Отчет по лабораторной работе 1

Полнодоступная двухсервисная модель Эрланга с одинаковыми интенсивностями обслуживания

Генералов Даниил, 1032212280

0. теоретическая информация

Исследуется сота сети связи емкостью C. Пусть пользователям сети предоставляются услуги двух типов. Запросы в виде двух пуассоновский потоков (ПП) с интенсивностями λ_1, λ_2 поступают в соту. Среднее время обслуживания запросов на предоставление услуг каждого типа μ_1^{-1}, μ_2^{-1} соответственно. Исследуются основные характеристики модели для случая $\mu_1 = \mu_2 = \mu$.

В классификации Башарина-Кендалла: $M(\lambda_1)M(\lambda_2)|M(\mu_1)M(\mu_2)|C|0$.

- C -- пиковая пропускная способность соты;
- λ_1, λ_2 -- интенсивность поступления запросов на предоставление услуги 1, 2-го типа;
- μ^{-1} -- среднее время обслуживания запроса на предоставление услуги 1, 2-го типа;
- ρ_1, ρ_2 -- интенсивность предложенной нагрузки, создаваемой запросами на предоставление услуги 1, 2-го типа;
- $X(t)$ -- число запросов, обслуживаемых в системе в момент времени $t, t \geq 0$ (случайный процесс (СП), описывающий функционирование системы в момент времени $t, t \geq 0$);
- X -- пространство состояний системы;
- n -- число обслуживаемых в системе запросов;
- B_1, B_2 -- множество блокировок запросов на предоставление услуги 1, 2-го типа;
- S_1, S_2 -- множество приема запросов на предоставление услуги 1, 2-го типа.

Пространство состояний системы: $X = \{0, \ldots, C\}$; $|X| = C + 1$

Множество блокировок запросов на предоставление услуги i-типа, $i = 1, 2$:
$B_1 = B_2 = \{C\}$

Множество приема запросов на предоставление услуги i-типа, $i=1,2$:
$S_i=\overline{B_i}=X\setminus B_i=\{0,\ldots C-1\}$

Система уравнений глобального баланса (СУГБ):

- $(\lambda_1+\lambda_2)p_0=\mu p_1$
- $(\lambda_1+\lambda_2+n\mu)p_n=(\lambda_1+\lambda_2)p_{n-1}+(n+1)\mu p_{n+1}, n=\overline{1,C-1}$
- $(\lambda_1+\lambda_2)p_{C-1}=C\mu p_C$

Обозначим $\rho_1=\frac{\lambda_1}{\mu},\rho_2=\frac{\lambda_2}{\mu}$.

Система уравнений локального баланса (СУЛБ):
$(\lambda_1+\lambda_2)p_{n-1}=n\mu p_n, n=\overline{1,C-1}$.

Стационарное распределение вероятностей состояний системы:
$p_n=\left(\sum_{i=0}^{C}\frac{(\rho_1+\rho_2)^i}{i!}\right)^{-1}\times\frac{(\rho_1+\rho_2)^n}{n!}, n=\overline{0,C}$

Используя СУЛБ, найдем стационарное распределение вероятностей состояний системы $p_n, n=\overline{1,C}$:

$p_n=p_{n-1}\frac{\lambda_1+\lambda_2}{n\mu}=p_{n-1}\frac{\rho_1+\rho_2}{n}=\ldots=p_0\frac{(\rho_1+\rho_2)^n}{n!}, n=\overline{1,C}$

Для нахождения вероятности p_0 воспользуемся условием нормировки $\sum_{n=0}^{C}p_n=1$:

$p_o=\left(\sum_{n=0}^{C}\frac{(\rho_1+\rho_2)^n}{n!}\right)^{-1}$

Основные вероятностные характеристики (ВХ) модели:

- Вероятность блокировки по времени E_i запроса на предоставление услуги i-типа, $i=1,2$: $E_1=E_2=E=\sum_{n\in B_i}p_n=p_C$
- Вероятность блокировки по вызовам B_i запроса на предоставление услуги i-типа, $i=1,2$: $B_i=\frac{\lambda_i}{\lambda_1+\lambda_2}E$ где $\frac{\lambda_i}{\lambda_1+\lambda_2}$ -- вероятность того, что поступит запрос на предоставление услуги i-типа
- Вероятность блокировки по нагрузке C_i запроса на предоставление услуги i-типа, $i=1,2$: $C_1=C_2=E$
- Среднее число $\overline{N}$ обслуживаемых в системе запросов: $\overline{N}=\sum_{n\in X}np_n$

1. подключение библиотек, определение функций

Для расчета больших факториалов нам потребуется длинная арифметика, а для рисования графиков -- библиотека для визуализации данных.

In [2]:
```
:dep num = { version = "^0.4.3" }
:dep plotters = { version = "^0.3.6", default-features = false, features = [

extern crate num;
use num::BigRational as R;
use num::BigInt as I;
use num::BigUint as U;
use num::Integer;
use num::traits::ConstZero;
use num::FromPrimitive;
use num::ToPrimitive;

extern crate plotters;
use plotters::prelude::*;
```

Для удобства конвертации стандартных чисел в числа длинной арифметики используются helper-функции.

In [3]:
```
fn u(i: usize) -> U {
    U::from_usize(i).unwrap()
}

fn rr(i: f64) -> R {
    R::from_float(i).unwrap()
}
```

Для вычисления факториала нет стандартной функции, и очевидные подходы не работают с длинной арифметикой, поэтому эта функция считает это за нас.

In [4]:
```
fn factorial(n: &U) -> R {
    let mut c = n.clone();
    let one = I::from_i8(1).unwrap();
    let mut out = R::new(one.clone(), one.clone());
    while c > U::ZERO {
        out *= R::new(I::from_biguint(num::bigint::Sign::Plus, c.clone()), o
        c -= 1u32;
    }
    out
}
```

Эта функция отображает график функции, принимая на вход список X-Y пар.

In [5]:
```
fn draw_chart(data: &Vec<(f32, f32)>, name: impl ToString) -> plotters::evcx
    let minx = data.iter().min_by(|a, b| a.0.partial_cmp(&b.0).unwrap_or(std
    let maxx = data.iter().max_by(|a, b| a.0.partial_cmp(&b.0).unwrap_or(std
    let miny = data.iter().min_by(|a, b| a.1.partial_cmp(&b.1).unwrap_or(std
    let maxy = data.iter().max_by(|a, b| a.1.partial_cmp(&b.1).unwrap_or(std
    let figure = evcxr_figure((640, 480), |root| {
        root.fill(&WHITE)?;
```

```
        let mut chart = ChartBuilder::on(&root)
            .caption(name.to_string(), ("Arial", 50).into_font())
            .margin(5)
            .x_label_area_size(30)
            .y_label_area_size(30)
            .build_cartesian_2d(minx..maxx, miny..maxy)?;

        chart.configure_mesh().draw()?;

        chart.draw_series(LineSeries::new(
            data.clone(),
            &RED,
        )).unwrap();

        // chart.configure_series_labels()
        //     .background_style(&WHITE.mix(0.8))
        //     .border_style(&BLACK)
        //     .draw()?;
        Ok(())
    });
    return figure;
}
```

Эта функция считает стационарное распределение вероятностей для рассматриваемой модели. Она принимает ρ_1 и ρ_2, которые соответствуют интенсивностям поступления заявок на два типа услуг, а также C -- максимальную пропускную способность, а именно количество заявок, которые могут одновременно выполняться.

Функция возвращает список: i-й элемент списка равен вероятности, что система будет найдена в состоянии i (то есть это p_i).

In [6]:
```
fn stationary_prob_distribution(rho1: &R, rho2: &R, max_c: u8) -> Vec<R> {
    let prob = |x: i32| (rho1 + rho2).pow(x) / (factorial(&u(x as usize)));

    let mut v = Vec::with_capacity(max_c as usize);

    let S = (0..=max_c).map(|x| prob(x as i32)).sum::<R>();
    let Sinv = rr(1.0)/S;

    for c in 0..=max_c {
        let res = &Sinv * prob(c as i32);
        v.push(res);
    }
    v
}
```

2. входные параметры

Здесь задаются параметры, которые определяют модель. Чтобы попробовать запустить вычисления с другими значениями, вы можете поменять эту ячейку и перезапустить ее и все ячейки ниже.

При разработке я использовал следующие значения для теста:

- $\lambda 1 = 30$
- $\lambda 2 = 10$
- $\mu_1 = \mu_2 = \frac{1}{2}$
- $C = 20$

In [7]:
```
let lambda1: R = rr(30.0);
let lambda2: R = rr(10.0);
let mu: R = rr(0.5);
let mu1: R = mu.clone();
let mu2: R = mu.clone();
let rho1: R = &lambda1 / &mu1;
let rho2: R = &lambda2 / &mu2;
let c: u8 = 20;
```

3. расчет

Распределение вероятностей можно посчитать с помощью нашей функции сверху. По свойству распределения вероятностей, сумма всех элементов должна быть равна 1, и мы здесь проверяем это.

In [8]:
```
let dist: Vec<R> = stationary_prob_distribution(&rho1, &rho2, c);
println!("probability distribution: {dist:?}");
println!("Sum should be 1: {}", dist.iter().sum::<R>());
```

```
probability distribution: [Ratio { numer: 14849255421, denom: 933279301282433
386345338108301 }, Ratio { numer: 1187940433680, denom: 933279301282433386345
338108301 }, Ratio { numer: 47517617347200, denom: 93327930128243338634533810
8301 }, Ratio { numer: 1267136462592000, denom: 9332793012824333863453381083O
1 }, Ratio { numer: 25342729251840000, denom: 933279301282433386345338108301
}, Ratio { numer: 405483668029440000, denom: 933279301282433386345338108301
}, Ratio { numer: 5406448907059200000, denom: 933279301282433386345338108301
}, Ratio { numer: 61787987509248000000, denom: 933279301282433386345338108301
}, Ratio { numer: 617879875092480000000, denom: 9332793012824333863453381083O
1 }, Ratio { numer: 5492265556377600000000, denom: 93327930128243338634533810
8301 }, Ratio { numer: 43938124451020800000000, denom: 933279301282433386345 3
38108301 }, Ratio { numer: 319549996007424000000000, denom: 93327930128243338
6345338108301 }, Ratio { numer: 2130333306716160000000000, denom: 93327930128
2433386345338108301 }, Ratio { numer: 13109743425945600000000000, denom: 9332
79301282433386345338108301 }, Ratio { numer: 74912819576832000000000000, deno
m: 933279301282433386345338108301 }, Ratio { numer: 399535037743104000000000
00, denom: 933279301282433386345338108301 }, Ratio { numer: 1997675188715520O
0000000000, denom: 933279301282433386345338108301 }, Ratio { numer: 94008244
174848000000000000000, denom: 933279301282433386345338108301 }, Ratio { numer:
417814418554880000000000000000, denom: 933279301282433386345338108301 }, Ratio
{ numer: 1759218604441600000000000000000, denom: 9332793012824333863453381083O
1 }, Ratio { numer: 7036874417766400000000000000000, denom: 933279301282433386
345338108301 }]
Sum should be 1: 1
```

Среднее число запросов, которые находятся в обработке -- это математическое ожидание набора вероятностей: ((вероятность того, что там 1 заявка) * 1 + (вероятность того, что там 2 заявки) * 2 + ...) / (макс. заявок)

Чтобы посчитать это, мы считаем сумму значений вероятности, помноженных на свой индекс (который равен количеству заявок, соответствующих этой ячейке).

In [9]:
```
let avg: R = dist.iter().enumerate().map(|(i,v)| rr(i as f64) * v).sum();
println!("Average requests in flight: {avg}");
```

```
Average requests in flight: 18367348760463470907627048664080/9332793012824333
86345338108301
```

Вероятность блокировки по времени равна между двумя типами запросов -- это просто вероятность того, что мы окажемся в последнем состоянии системы, потому что после этого уже нельзя обработать ни одну заявку, пока нагрузка не спадет.

In [10]:
```
let time_blocking_prob: R = dist.last().cloned().unwrap();
println!("time blocking probability (E): {}", time_blocking_prob.to_f64().un
```

```
time blocking probability (E): 0.7539944803336925
```

Вероятность блокировки по вызовам -- это вероятность, что определенный из двух типов запросов будет заблокирован; поэтому это связано с вероятностью поступления каждого из двух запросов.

In [11]:
```
let req_blocking_prob1: R = &lambda1 / (&lambda1 + &lambda2) * &time_blockin
let req_blocking_prob2: R = &lambda2 / (&lambda1 + &lambda2) * &time_blockin
println!("request blocking probability: {}, {}", req_blocking_prob1.to_f64()
```

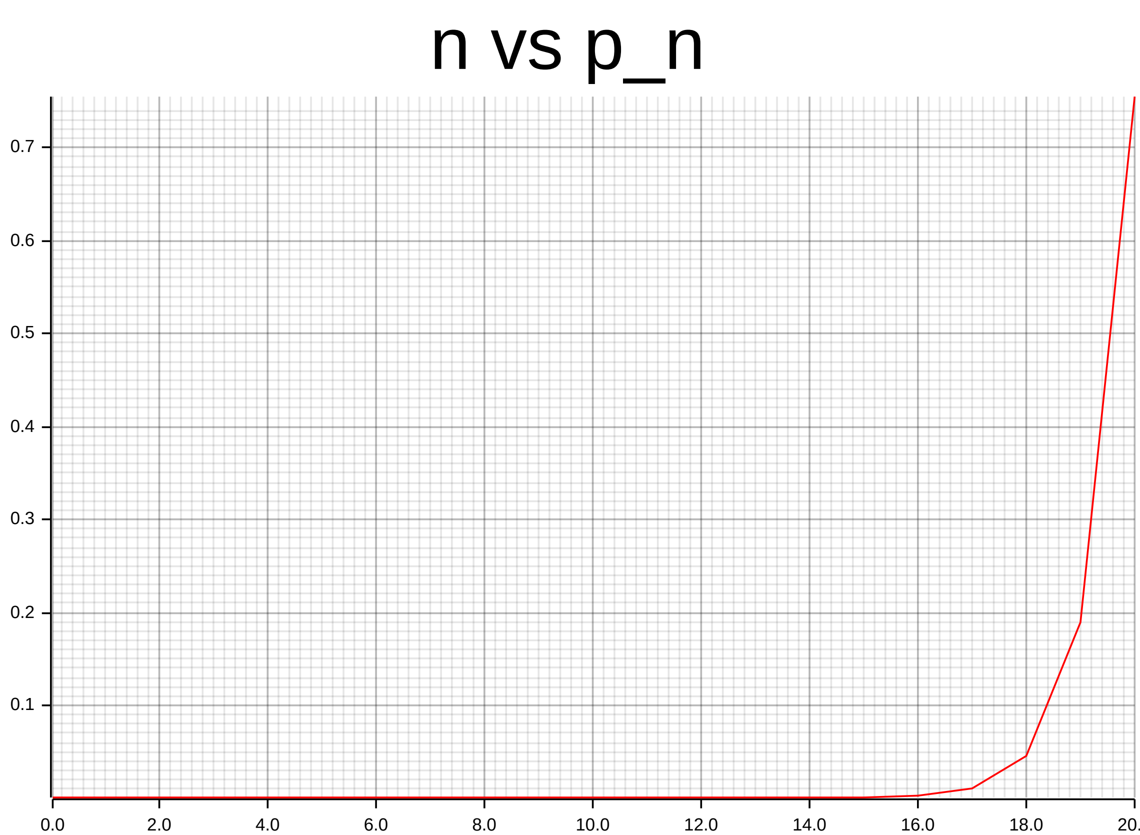
```
request blocking probability: 0.5654958602502693, 0.18849862008342314
```

4. графики

График распределения вероятности показывает, какие состояния системы являются самыми частыми. Если пик этого графика находится на правой границе, значит система перегружена; иначе он показывает среднее количество ожидающих своей очереди запросов (таким образом это значение имеет похожий смысл на E).

In [12]:
```
draw_chart(&dist.iter().enumerate().map(|(a,b)| (a as f32, b.to_f32().unwrap
```

Out[12]:



Для того, чтобыв посчитать график зависимости вероятности блокировки от интенсивности, надо зафиксировать значение одной из переменных λ и варьировать другую; для каждой такой конфигурации нужно вычислить значение E и затем отобразить на графике.

Сначала делаем это для λ_1.

In [13]:

```
let mut block_prob = vec![];
for lambda1 in 0..=100 {
    let rho1 = rr(lambda1 as f64) / &mu1;
    let dist = stationary_prob_distribution(&rho1, &rho2, c);
    block_prob.push((lambda1 as f32, dist.last().cloned().unwrap().to_f32().
}
block_prob
```

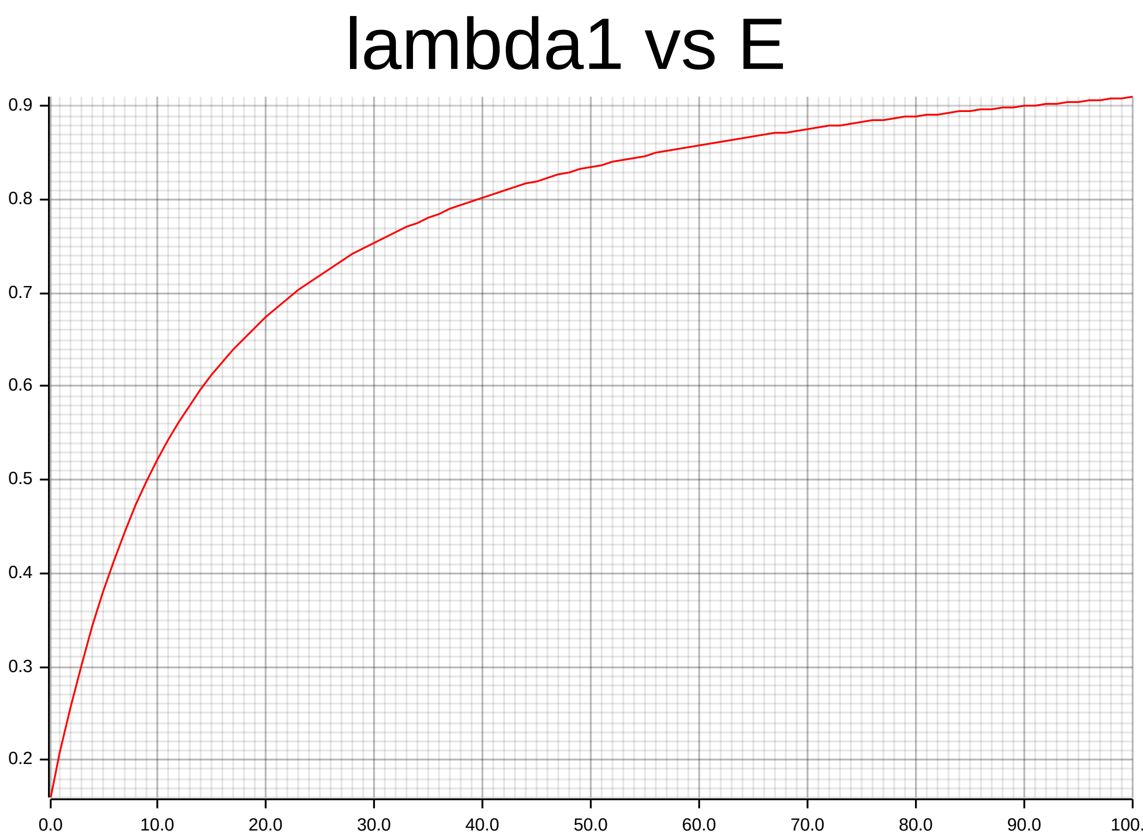
Out[13]:

```
[(0.0, 0.15889196), (1.0, 0.20904599), (2.0, 0.25708255), (3.0, 0.3017891),
(4.0, 0.34277654), (5.0, 0.38008487), (6.0, 0.41395226), (7.0, 0.44469154),
(8.0, 0.47262853), (9.0, 0.498073), (10.0, 0.521307), (11.0, 0.54258144),
(12.0, 0.5621168), (13.0, 0.5801058), (14.0, 0.59671634), (15.0, 0.6120946
4), (16.0, 0.6263683), (17.0, 0.6396486), (18.0, 0.65203315), (19.0, 0.6636
0754), (20.0, 0.67444724), (21.0, 0.6846187), (22.0, 0.69418097), (23.0, 0.
70318633), (24.0, 0.7116815), (25.0, 0.7197081), (26.0, 0.72730345), (27.0,
0.734501), (28.0, 0.741331), (29.0, 0.7478206), (30.0, 0.75399446), (31.0,
0.75987494), (32.0, 0.7654823), (33.0, 0.77083504), (34.0, 0.77594995), (3
5.0, 0.78084254), (36.0, 0.7855269), (37.0, 0.79001594), (38.0, 0.7943216),
(39.0, 0.7984548), (40.0, 0.80242574), (41.0, 0.8062437), (42.0, 0.8099173
3), (43.0, 0.8134547), (44.0, 0.81686306), (45.0, 0.8201495), (46.0, 0.8233
203), (47.0, 0.82638156), (48.0, 0.8293388), (49.0, 0.8321971), (50.0, 0.83
496153), (51.0, 0.8376365), (52.0, 0.8402263), (53.0, 0.8427349), (54.0, 0.
8451661), (55.0, 0.8475234), (56.0, 0.8498101), (57.0, 0.8520294), (58.0,
0.8541841), (59.0, 0.85627705), (60.0, 0.8583109), (61.0, 0.860288), (62.0,
0.8622108), (63.0, 0.8640815), (64.0, 0.86590207), (65.0, 0.8676746), (66.
0, 0.8694009), (67.0, 0.8710829), (68.0, 0.8727221), (69.0, 0.87432015), (7
0.0, 0.87587863), (71.0, 0.877399), (72.0, 0.8788826), (73.0, 0.88033074),
(74.0, 0.88174474), (75.0, 0.88312566), (76.0, 0.8844748), (77.0, 0.8857931
5), (78.0, 0.8870818), (79.0, 0.88834167), (80.0, 0.88957375), (81.0, 0.890
779), (82.0, 0.89195824), (83.0, 0.8931123), (84.0, 0.894242), (85.0, 0.895
3481), (86.0, 0.89643127), (87.0, 0.8974923), (88.0, 0.8985318), (89.0, 0.8
9955044), (90.0, 0.9005488), (91.0, 0.9015276), (92.0, 0.9024873), (93.0,
0.9034285), (94.0, 0.9043517), (95.0, 0.9052574), (96.0, 0.90614617), (97.
0, 0.9070184), (98.0, 0.9078746), (99.0, 0.9087152), (100.0, 0.90954053)]
```

In [14]:

```
draw_chart(&block_prob, "lambda1 vs E")
```

Out[14]:



После этого -- для λ_2.

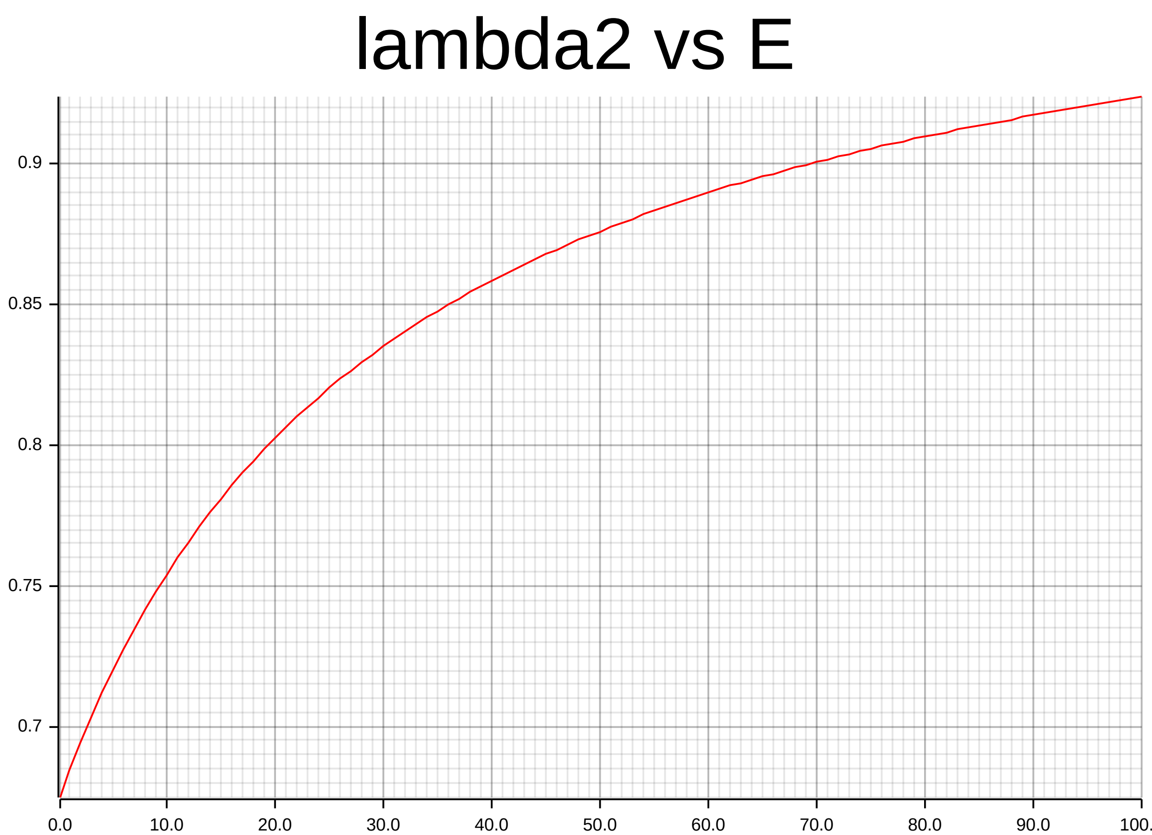
In [15]:

```
let mut block_prob = vec![];
for lambda2 in 0..=100 {
    let rho2 = rr(lambda2 as f64) / &mu2;
    let dist = stationary_prob_distribution(&rho1, &rho2, c);
    block_prob.push((lambda2 as f32, dist.last().cloned().unwrap().to_f32().
}
block_prob
```

Out[15]:
```
[(0.0, 0.67444724), (1.0, 0.6846187), (2.0, 0.69418097), (3.0, 0.70318633),
(4.0, 0.7116815), (5.0, 0.7197081), (6.0, 0.72730345), (7.0, 0.734501), (8.
0, 0.741331), (9.0, 0.7478206), (10.0, 0.75399446), (11.0, 0.75987494), (1
2.0, 0.7654823), (13.0, 0.77083504), (14.0, 0.77594995), (15.0, 0.7808425
4), (16.0, 0.7855269), (17.0, 0.79001594), (18.0, 0.7943216), (19.0, 0.7984
548), (20.0, 0.80242574), (21.0, 0.8062437), (22.0, 0.80991733), (23.0, 0.8
134547), (24.0, 0.81686306), (25.0, 0.8201495), (26.0, 0.8233203), (27.0,
0.82638156), (28.0, 0.8293388), (29.0, 0.8321971), (30.0, 0.83496153), (31.
0, 0.8376365), (32.0, 0.8402263), (33.0, 0.8427349), (34.0, 0.8451661), (3
5.0, 0.8475234), (36.0, 0.8498101), (37.0, 0.8520294), (38.0, 0.8541841),
(39.0, 0.85627705), (40.0, 0.8583109), (41.0, 0.860288), (42.0, 0.8622108),
(43.0, 0.8640815), (44.0, 0.86590207), (45.0, 0.8676746), (46.0, 0.869400
9), (47.0, 0.8710829), (48.0, 0.8727221), (49.0, 0.87432015), (50.0, 0.8758
7863), (51.0, 0.877399), (52.0, 0.8788826), (53.0, 0.88033074), (54.0, 0.88
174474), (55.0, 0.88312566), (56.0, 0.8844748), (57.0, 0.88579315), (58.0,
0.8870818), (59.0, 0.88834167), (60.0, 0.88957375), (61.0, 0.890779), (62.
0, 0.89195824), (63.0, 0.8931123), (64.0, 0.894242), (65.0, 0.8953481), (6
6.0, 0.89643127), (67.0, 0.8974923), (68.0, 0.8985318), (69.0, 0.89955044),
(70.0, 0.9005488), (71.0, 0.9015276), (72.0, 0.9024873), (73.0, 0.9034285),
(74.0, 0.9043517), (75.0, 0.9052574), (76.0, 0.90614617), (77.0, 0.907018
4), (78.0, 0.9078746), (79.0, 0.9087152), (80.0, 0.90954053), (81.0, 0.9103
5116), (82.0, 0.91114736), (83.0, 0.9119295), (84.0, 0.91269803), (85.0, 0.
9134533), (86.0, 0.9141956), (87.0, 0.9149253), (88.0, 0.9156427), (89.0,
0.91634804), (90.0, 0.9170418), (91.0, 0.9177241), (92.0, 0.9183952), (93.
0, 0.9190555), (94.0, 0.9197052), (95.0, 0.9203446), (96.0, 0.92097384), (9
7.0, 0.92159325), (98.0, 0.92220306), (99.0, 0.9228034), (100.0, 0.9233945
6)]
```

In [16]:
```
draw_chart(&block_prob, "lambda2 vs E")
```

Out[16]:



Для того, чтобы нарисовать график среднего числа обслуживаемых запросов, нужно сделать то же самое: по разным значениям λ посчитать распределение вероятности, из него -- среднее количество, и отобразить для него точку. Сначала делаем это, варьируя λ_1, а затем λ_2.

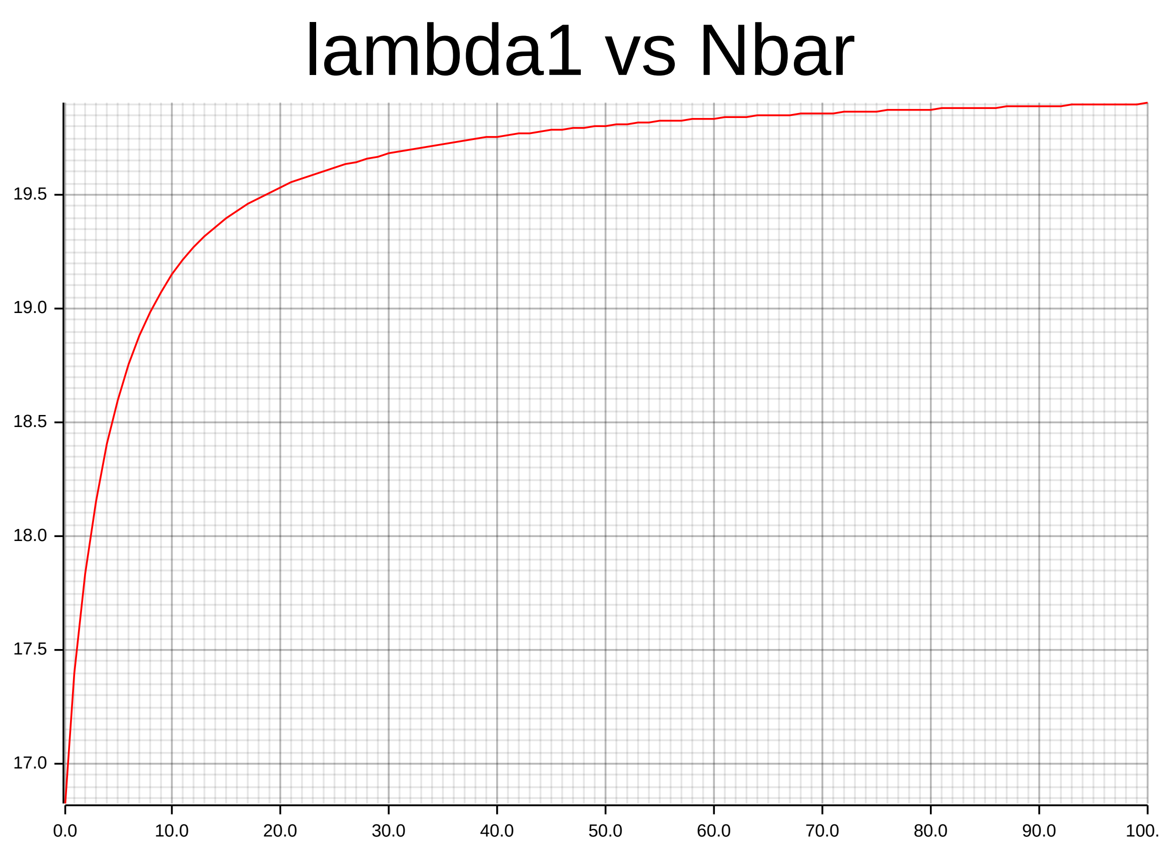
In [17]:

```
let mut avg_req_counts = vec![];
for lambda1 in 0..=100 {
    let rho1 = rr(lambda1 as f64) / &mu1;
    let dist = stationary_prob_distribution(&rho1, &rho2, c);
    let avg: R = dist.iter().enumerate().map(|(i,v)| rr(i as f64) * v).sum()
    avg_req_counts.push((lambda1 as f32, avg.to_f32().unwrap()));
}
avg_req_counts
```

Out[17]: [(0.0, 16.82216), (1.0, 17.400988), (2.0, 17.830019), (3.0, 18.153484), (4.0, 18.402256), (5.0, 18.597454), (6.0, 18.753529), (7.0, 18.880487), (8.0, 18.985373), (9.0, 19.073225), (10.0, 19.14772), (11.0, 19.21158), (12.0, 19.266861), (13.0, 19.315134), (14.0, 19.357616), (15.0, 19.395267), (16.0, 19.42885), (17.0, 19.458977), (18.0, 19.486143), (19.0, 19.510761), (20.0, 19.533167), (21.0, 19.55364), (22.0, 19.572418), (23.0, 19.5897), (24.0, 19.605658), (25.0, 19.620434), (26.0, 19.634153), (27.0, 19.646925), (28.0, 19.658846), (29.0, 19.669992), (30.0, 19.68044), (31.0, 19.690254), (32.0, 19.699486), (33.0, 19.708187), (34.0, 19.716402), (35.0, 19.72417), (36.0, 19.731527), (37.0, 19.738504), (38.0, 19.745129), (39.0, 19.751429), (40.0, 19.757425), (41.0, 19.763142), (42.0, 19.768597), (43.0, 19.773806), (44.0, 19.778788), (45.0, 19.783554), (46.0, 19.78812), (47.0, 19.7925), (48.0, 19.796703), (49.0, 19.80074), (50.0, 19.804619), (51.0, 19.80835), (52.0, 19.811943), (53.0, 19.815403), (54.0, 19.818739), (55.0, 19.821957), (56.0, 19.825062), (57.0, 19.828062), (58.0, 19.830961), (59.0, 19.833763), (60.0, 19.836475), (61.0, 19.8391), (62.0, 19.841642), (63.0, 19.844105), (64.0, 19.846493), (65.0, 19.848808), (66.0, 19.851055), (67.0, 19.853237), (68.0, 19.855356), (69.0, 19.857414), (70.0, 19.859415), (71.0, 19.86136), (72.0, 19.863255), (73.0, 19.865095), (74.0, 19.866888), (75.0, 19.868633), (76.0, 19.870335), (77.0, 19.871992), (78.0, 19.873608), (79.0, 19.875183), (80.0, 19.876719), (81.0, 19.878218), (82.0, 19.879683), (83.0, 19.881111), (84.0, 19.882505), (85.0, 19.883867), (86.0, 19.8852), (87.0, 19.886501), (88.0, 19.887774), (89.0, 19.889017), (90.0, 19.890234), (91.0, 19.891424), (92.0, 19.89259), (93.0, 19.89373), (94.0, 19.894846), (95.0, 19.895939), (96.0, 19.89701), (97.0, 19.898058), (98.0, 19.899086), (99.0, 19.900093), (100.0, 19.901081)]

In [18]:
```
draw_chart(&avg_req_counts, "lambda1 vs Nbar")
```

Out[18]:



In [20]:

```
let mut avg_req_counts = vec![];
for lambda2 in 0..=100 {
    let rho2 = rr(lambda2 as f64) / &mu2;
    let dist = stationary_prob_distribution(&rho1, &rho2, c);
    let avg: R = dist.iter().enumerate().map(|(i,v)| rr(i as f64) * v).sum()
    avg_req_counts.push((lambda2 as f32, avg.to_f32().unwrap()));
}
avg_req_counts
```

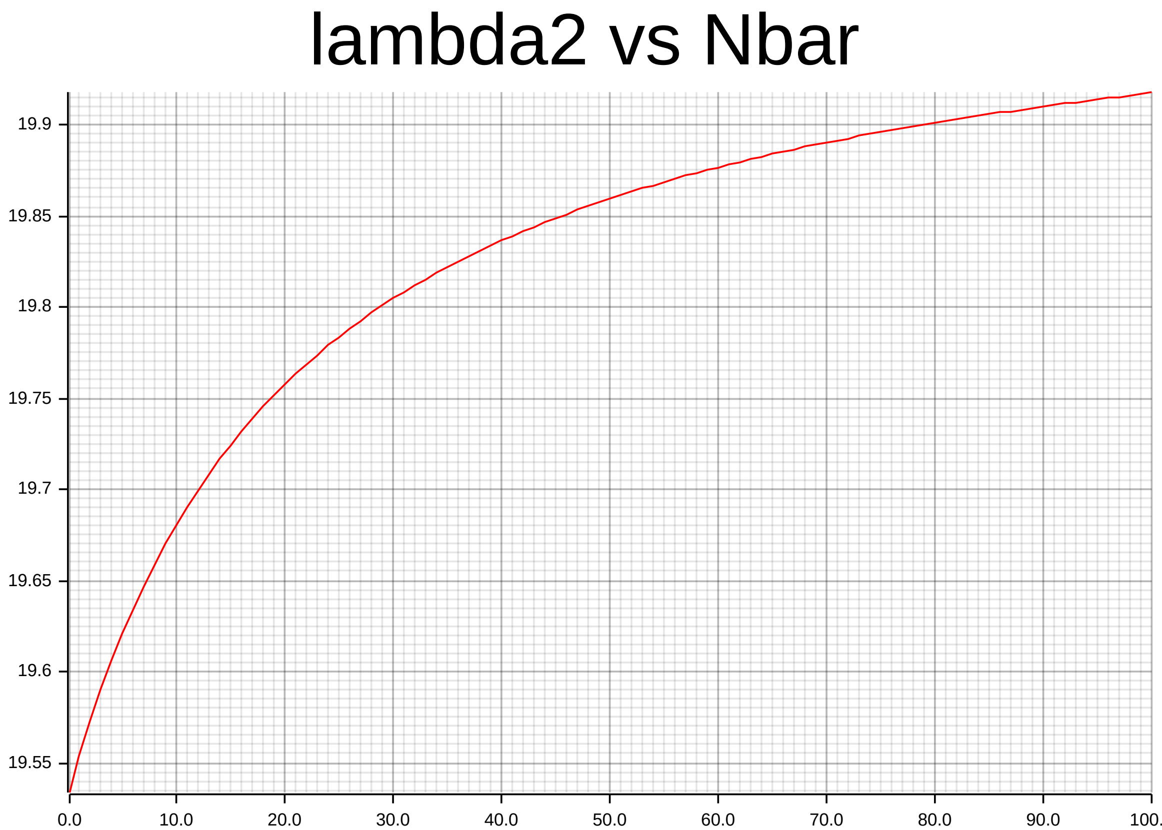
Out[20]:

```
[(0.0, 19.533167), (1.0, 19.55364), (2.0, 19.572418), (3.0, 19.5897), (4.0,
19.605658), (5.0, 19.620434), (6.0, 19.634153), (7.0, 19.646925), (8.0, 19.
658846), (9.0, 19.669992), (10.0, 19.68044), (11.0, 19.690254), (12.0, 19.6
99486), (13.0, 19.708187), (14.0, 19.716402), (15.0, 19.72417), (16.0, 19.7
31527), (17.0, 19.738504), (18.0, 19.745129), (19.0, 19.751429), (20.0, 19.
757425), (21.0, 19.763142), (22.0, 19.768597), (23.0, 19.773806), (24.0, 1
9.778788), (25.0, 19.783554), (26.0, 19.78812), (27.0, 19.7925), (28.0, 19.
796703), (29.0, 19.80074), (30.0, 19.804619), (31.0, 19.80835), (32.0, 19.8
11943), (33.0, 19.815403), (34.0, 19.818739), (35.0, 19.821957), (36.0, 19.
825062), (37.0, 19.828062), (38.0, 19.830961), (39.0, 19.833763), (40.0, 1
9.836475), (41.0, 19.8391), (42.0, 19.841642), (43.0, 19.844105), (44.0, 1
9.846493), (45.0, 19.848808), (46.0, 19.851055), (47.0, 19.853237), (48.0,
19.855356), (49.0, 19.857414), (50.0, 19.859415), (51.0, 19.86136), (52.0,
19.863255), (53.0, 19.865095), (54.0, 19.866888), (55.0, 19.868633), (56.0,
19.870335), (57.0, 19.871992), (58.0, 19.873608), (59.0, 19.875183), (60.0,
19.876719), (61.0, 19.878218), (62.0, 19.879683), (63.0, 19.881111), (64.0,
19.882505), (65.0, 19.883867), (66.0, 19.8852), (67.0, 19.886501), (68.0, 1
9.887774), (69.0, 19.889017), (70.0, 19.890234), (71.0, 19.891424), (72.0,
19.89259), (73.0, 19.89373), (74.0, 19.894846), (75.0, 19.895939), (76.0, 1
9.89701), (77.0, 19.898058), (78.0, 19.899086), (79.0, 19.900093), (80.0, 1
9.901081), (81.0, 19.90205), (82.0, 19.902998), (83.0, 19.903929), (84.0, 1
9.904842), (85.0, 19.905739), (86.0, 19.906618), (87.0, 19.907482), (88.0,
19.908329), (89.0, 19.90916), (90.0, 19.909979), (91.0, 19.910782), (92.0,
19.91157), (93.0, 19.912344), (94.0, 19.913105), (95.0, 19.913853), (96.0,
19.914589), (97.0, 19.915312), (98.0, 19.916021), (99.0, 19.916721), (100.
0, 19.917408)]
```

In [21]:

```
draw_chart(&avg_req_counts, "lambda2 vs Nbar")
```

Out[21]:



lambda2 vs Nbar
19.9
19.85
19.8
19.75
19.7
19.65
19.6
19.55
0.0
10.0
20.0
30.0
40.0
50.0
60.0
70.0
80.0
90.0
100.

In [ ]: