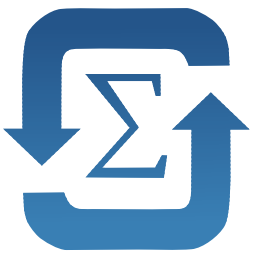
Министерство образования и науки Российской Федерации

Федеральное государственное бюджетное образовательное учреждение
высшего образования
«НОВОСИБИРСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ»

Кафедра прикладной математики

Лабораторная работа №4
по дисциплине «Численные методы»

Решение систем нелинейных уравнений методом Ньютона



Факультет: ПМИ
Группа: ПМ-63
Студент: Шепрут И.И.
Вариант: Все
Преподаватель: Задорожный А.Г.

Новосибирск
2018

1 Цель работы

Разработать программу решения системы нелинейных уравнений (СНУ) методом Ньютона. Провести исследования метода для нескольких систем размерности от 2 до 10.

2 Исследования

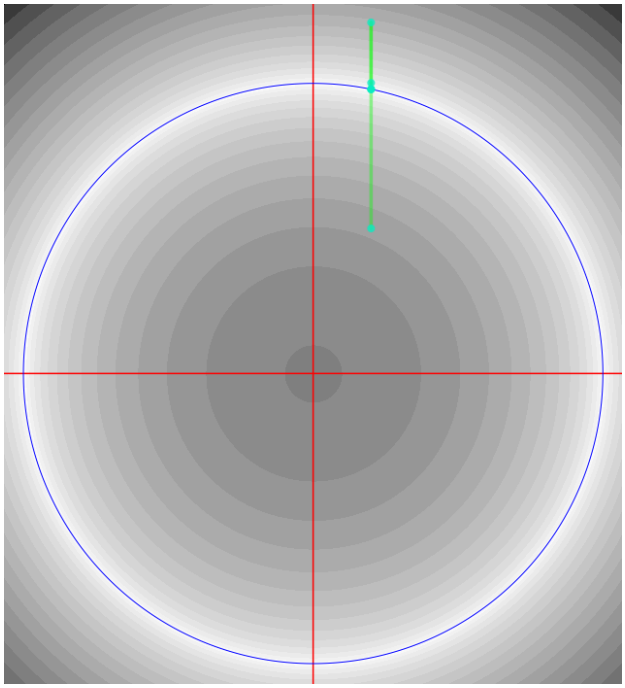
Описание визуализации: невязка в каждой точке рисуется после приведения СНУ к квадратному виду, сделано это для большей наглядности, потому что норма невязки от всех $m > n$ уравнений не настолько точно показывает куда будет двигаться метод. Так же, из-за того, что изображение получалось слишком светлым возле точки решения, невязка нормируется и возводится в квадрат, поэтому изображения стали более темными.

Для всех запусков заданые следующие значения: максимальное количество итераций — 30, минимальная невязка — 10^{-10},

2.1 Одна окружность

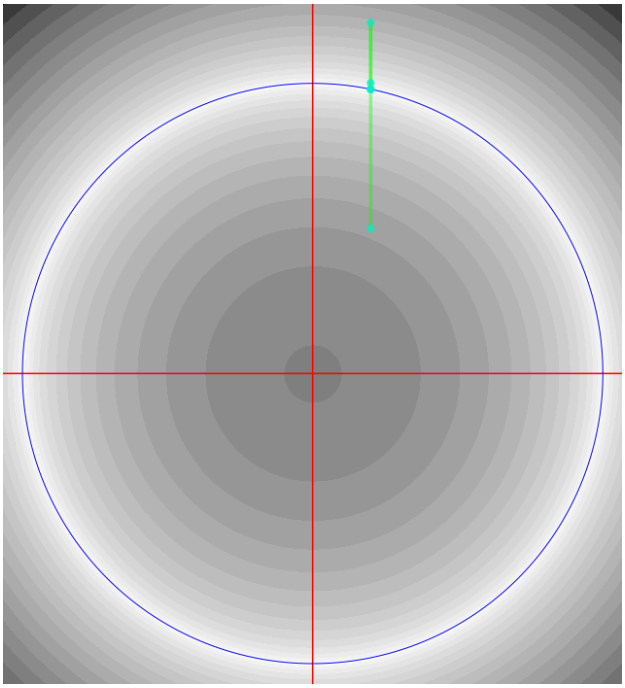
Вариант приведения к квадратному виду: 1

Аналитическое вычисление матрицы Якоби



k	β	$\frac{\|f(x_k)\|}{f(x_0)}$	x	y
1	1	$7.1 \cdot 10^{-1}$	0.20000000	1.21000000
2	1	$6.1 \cdot 10^{-2}$	0.20000000	1.00169421
3	1	$6.6 \cdot 10^{-4}$	0.20000000	0.98003526
4	1	$8.1 \cdot 10^{-8}$	0.20000000	0.97979593
5	1	$1.3 \cdot 10^{-15}$	0.20000000	0.97979590

Численное вычисление матрицы Якоби



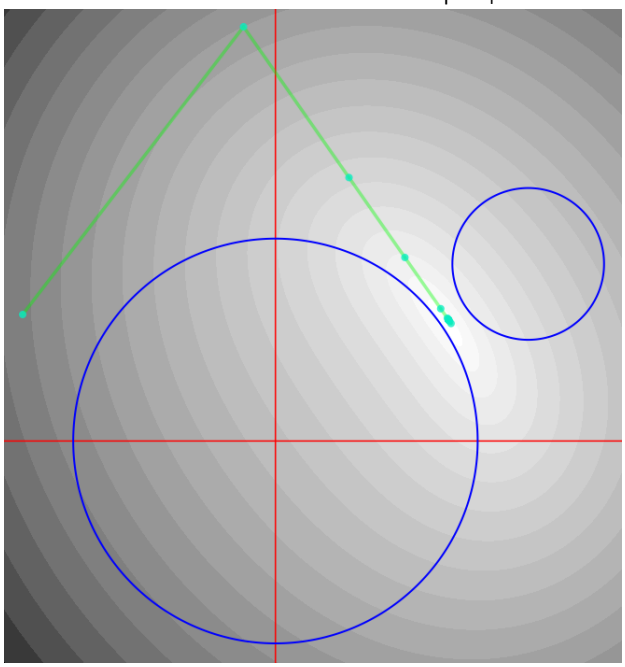
k	β	$\frac{\|f(x_k)\|}{f(x_0)}$	x	y
1	1	$7.1 \cdot 10^{-1}$	0.20000000	1.20999994
2	1	$6.1 \cdot 10^{-2}$	0.20000000	1.00169420
3	1	$6.6 \cdot 10^{-4}$	0.20000000	0.98003526
4	1	$8.1 \cdot 10^{-8}$	0.20000000	0.97979593
5	1	$2.8 \cdot 10^{-15}$	0.20000000	0.97979590

2.2 Две окружности

2.2.1 Не пересекаются

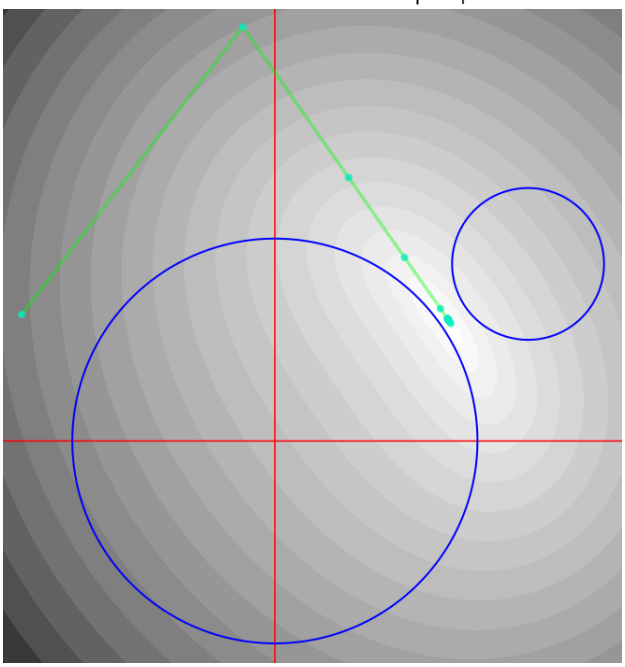
Вариант приведения к квадратному виду: —

Аналитическое вычисление матрицы Якоби



k	β	$\frac{\|\|f(\mathbf{x}_k)\|\|}{f(\mathbf{x}_0)}$	x	y
1	1	$7.3 \cdot 10^{-1}$	−0.12625000	1.63750000
2	1	$1.9 \cdot 10^{-1}$	0.29097075	1.04147036
3	1	$5.3 \cdot 10^{-2}$	0.51210419	0.72556544
4	1	$2.2 \cdot 10^{-2}$	0.65387654	0.52303351
5	$1.3 \cdot 10^{-1}$	$2.1 \cdot 10^{-2}$	0.69481257	0.46455347
6	$1.6 \cdot 10^{-2}$	$2.1 \cdot 10^{-2}$	0.68012890	0.48553014
7	$3.9 \cdot 10^{-3}$	$2.1 \cdot 10^{-2}$	0.68857468	0.47346474
8	$2.0 \cdot 10^{-3}$	$2.1 \cdot 10^{-2}$	0.68390632	0.48013382
9	$6.1 \cdot 10^{-5}$	$2.1 \cdot 10^{-2}$	0.68479563	0.47886339
10	$7.6 \cdot 10^{-6}$	$2.1 \cdot 10^{-2}$	0.68448045	0.47931364
11	$9.5 \cdot 10^{-7}$	$2.1 \cdot 10^{-2}$	0.68459010	0.47915700
12	$1.2 \cdot 10^{-7}$	$2.1 \cdot 10^{-2}$	0.68454676	0.47921892
13	$6.0 \cdot 10^{-8}$	$2.1 \cdot 10^{-2}$	0.68458033	0.47917095
14	$3.0 \cdot 10^{-8}$	$2.1 \cdot 10^{-2}$	0.68456311	0.47919556
15	$5.8 \cdot 10^{-11}$	$2.1 \cdot 10^{-2}$	0.68456397	0.47919433

Численное вычисление матрицы Якоби



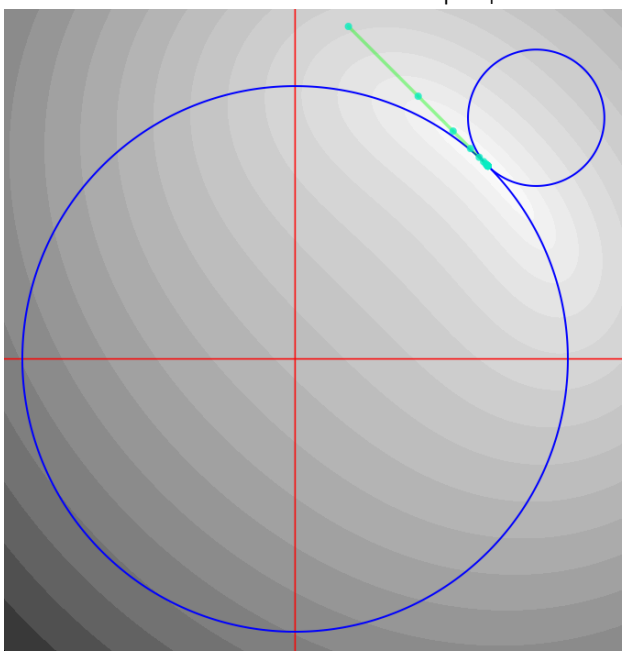
k	β	$\frac{\|\|f(\mathbf{x}_k)\|\|}{f(\mathbf{x}_0)}$	x	y
1	1	$7.3 \cdot 10^{-1}$	−0.12625007	1.63749991
2	1	$1.9 \cdot 10^{-1}$	0.29097138	1.04147030
3	1	$5.3 \cdot 10^{-2}$	0.51210433	0.72556522
4	1	$2.2 \cdot 10^{-2}$	0.65387666	0.52303330
5	$1.3 \cdot 10^{-1}$	$2.1 \cdot 10^{-2}$	0.69481287	0.46455301
6	$1.6 \cdot 10^{-2}$	$2.1 \cdot 10^{-2}$	0.68012964	0.48552904
7	$3.9 \cdot 10^{-3}$	$2.1 \cdot 10^{-2}$	0.68857685	0.47346160
8	$2.0 \cdot 10^{-3}$	$2.1 \cdot 10^{-2}$	0.68391104	0.48012705
9	$6.1 \cdot 10^{-5}$	$2.1 \cdot 10^{-2}$	0.68480681	0.47884737
10	$7.6 \cdot 10^{-6}$	$2.1 \cdot 10^{-2}$	0.68450617	0.47927686
11	$4.8 \cdot 10^{-7}$	$2.1 \cdot 10^{-2}$	0.68458553	0.47916350
12	$6.0 \cdot 10^{-8}$	$2.1 \cdot 10^{-2}$	0.68455933	0.47920092
13	$3.7 \cdot 10^{-9}$	$2.1 \cdot 10^{-2}$	0.68456743	0.47918935
14	$1.9 \cdot 10^{-9}$	$2.1 \cdot 10^{-2}$	0.68456258	0.47919627
15	$2.3 \cdot 10^{-10}$	$2.1 \cdot 10^{-2}$	0.68456451	0.47919352
16	$1.2 \cdot 10^{-10}$	$2.1 \cdot 10^{-2}$	0.68456306	0.47919559
17	$5.8 \cdot 10^{-11}$	$2.1 \cdot 10^{-2}$	0.68456388	0.47919441

2.2.2 Пересекаются в одной точке

2.2.2.1 Начальное приближение лежит на оси симметрии

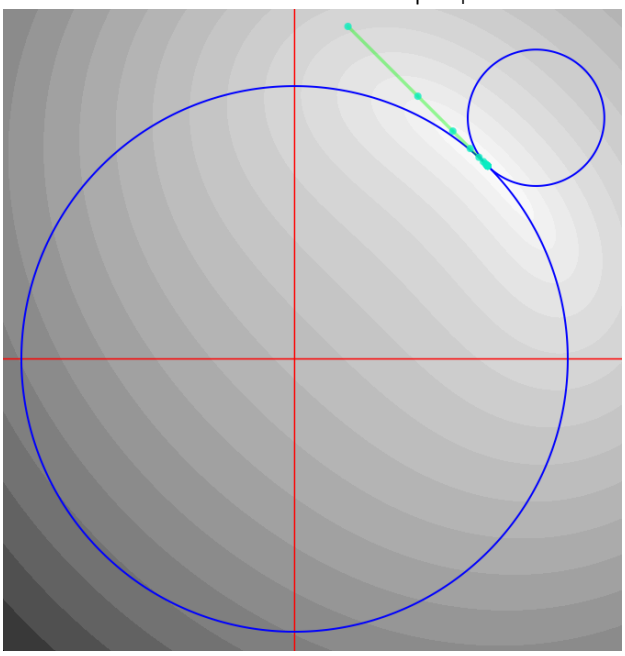
Вариант приведения к квадратному виду: —

Аналитическое вычисление матрицы Якоби



k	β	$\frac{\|\|f(\mathbf{x}_k)\|\|}{f(\mathbf{x}_0)}$	x	y
1	1	$2.5 \cdot 10^{-1}$	0.36104517	0.77032567
2	1	$6.3 \cdot 10^{-2}$	0.46336530	0.66800555
3	1	$1.6 \cdot 10^{-2}$	0.51452536	0.61684549
4	1	$3.9 \cdot 10^{-3}$	0.54010539	0.59126546
5	1	$9.8 \cdot 10^{-4}$	0.55289541	0.57847544
6	1	$2.4 \cdot 10^{-4}$	0.55929042	0.57208043
7	1	$6.1 \cdot 10^{-5}$	0.56248792	0.56888293
8	1	$1.5 \cdot 10^{-5}$	0.56408667	0.56728418
9	1	$3.8 \cdot 10^{-6}$	0.56488605	0.56648480
10	1	$9.5 \cdot 10^{-7}$	0.56528574	0.56608511
11	1	$2.4 \cdot 10^{-7}$	0.56548558	0.56588527
12	1	$6.0 \cdot 10^{-8}$	0.56558550	0.56578535
13	1	$1.5 \cdot 10^{-8}$	0.56563546	0.56573539
14	1	$3.7 \cdot 10^{-9}$	0.56566044	0.56571041
15	1	$9.3 \cdot 10^{-10}$	0.56567293	0.56569792
16	1	$2.3 \cdot 10^{-10}$	0.56567918	0.56569167
17	1	$5.8 \cdot 10^{-11}$	0.56568230	0.56568855

Численное вычисление матрицы Якоби



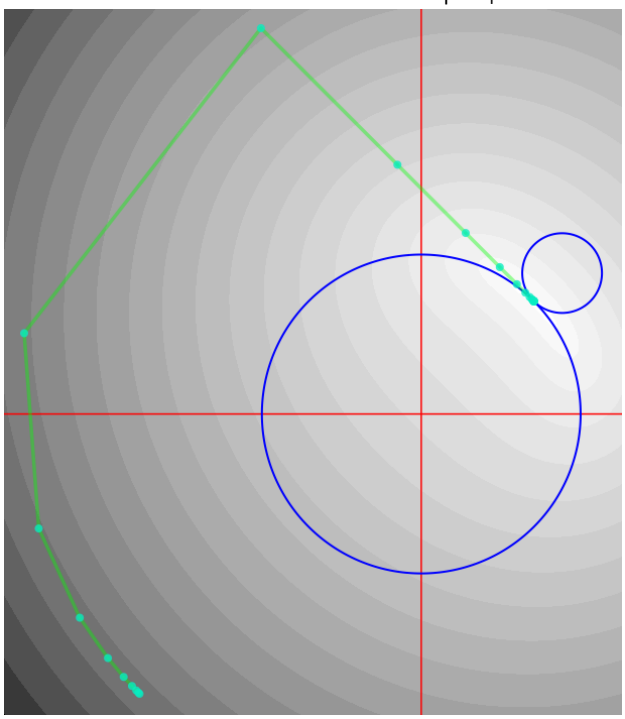
k	β	$\frac{\|\|f(\mathbf{x}_k)\|\|}{f(\mathbf{x}_0)}$	x	y
1	1	$2.5 \cdot 10^{-1}$	0.36104513	0.77032573
2	1	$6.3 \cdot 10^{-2}$	0.46336528	0.66800558
3	1	$1.6 \cdot 10^{-2}$	0.51452535	0.61684550
4	1	$3.9 \cdot 10^{-3}$	0.54010539	0.59126545
5	1	$9.8 \cdot 10^{-4}$	0.55289541	0.57847544
6	1	$2.4 \cdot 10^{-4}$	0.55929042	0.57208043
7	1	$6.1 \cdot 10^{-5}$	0.56248792	0.56888293
8	1	$1.5 \cdot 10^{-5}$	0.56408667	0.56728418
9	1	$3.8 \cdot 10^{-6}$	0.56488605	0.56648480
10	1	$9.5 \cdot 10^{-7}$	0.56528573	0.56608512
11	1	$2.4 \cdot 10^{-7}$	0.56548558	0.56588527
12	1	$6.0 \cdot 10^{-8}$	0.56558551	0.56578534
13	1	$1.5 \cdot 10^{-8}$	0.56563546	0.56573539
14	1	$3.7 \cdot 10^{-9}$	0.56566044	0.56571041
15	1	$9.3 \cdot 10^{-10}$	0.56567294	0.56569791
16	1	$2.3 \cdot 10^{-10}$	0.56567918	0.56569167
17	1	$5.8 \cdot 10^{-11}$	0.56568231	0.56568854

2.2.2.2 Начальное приближение лежит на оси, соединяющей центры окружностей

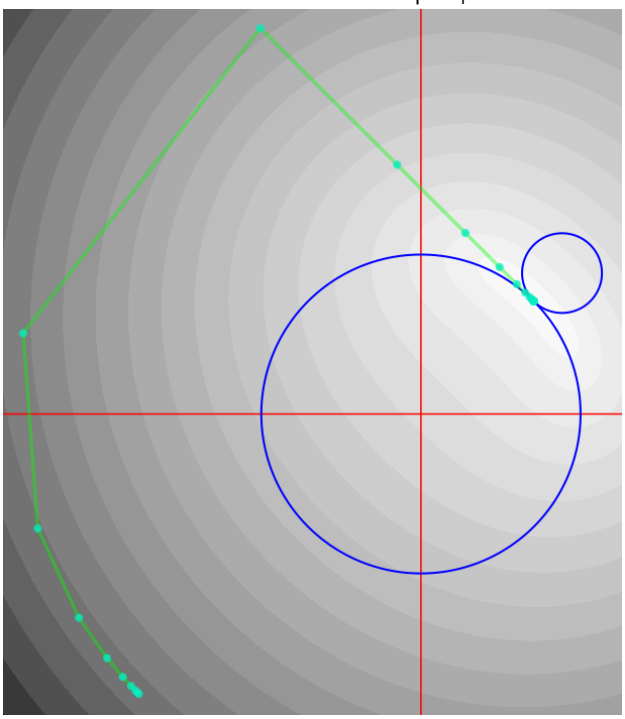
Комментарий: добавлено немного смещения, потому что на этой оси метод не сходится.

Вариант приведения к квадратному виду: —

Аналитическое вычисление матрицы Якоби



Численное вычисление матрицы Якоби



k	β	$\frac{\|f(x_k)\|}{f(x_0)}$	x	y
1	$1.5 \cdot 10^{-5}$	1	−1.42013466	−1.39823219
2	$6.1 \cdot 10^{-5}$	1	−1.43088214	−1.38724364
3	$2.4 \cdot 10^{-4}$	1	−1.45221430	−1.36494725
4	$9.8 \cdot 10^{-4}$	$1.0 \cdot 10^{0}$	−1.49388254	−1.31942302
5	$3.9 \cdot 10^{-3}$	$1.0 \cdot 10^{0}$	−1.57310973	−1.22478694
6	$1.6 \cdot 10^{-2}$	$9.9 \cdot 10^{-1}$	−1.71419326	−1.02230861
7	$6.3 \cdot 10^{-2}$	$9.6 \cdot 10^{-1}$	−1.92036679	−0.57439303
8	$2.5 \cdot 10^{-1}$	$8.6 \cdot 10^{-1}$	−1.99353987	0.40531272
9	1	$5.6 \cdot 10^{-1}$	−0.80483512	1.93620597
10	1	$1.4 \cdot 10^{-1}$	−0.11957485	1.25094570
11	1	$3.5 \cdot 10^{-2}$	0.22305529	0.90831556
12	1	$8.7 \cdot 10^{-3}$	0.39437036	0.73700049
13	1	$2.2 \cdot 10^{-3}$	0.48002789	0.65134296
14	1	$5.5 \cdot 10^{-4}$	0.52285666	0.60851419
15	1	$1.4 \cdot 10^{-4}$	0.54427104	0.58709981
16	1	$3.4 \cdot 10^{-5}$	0.55497823	0.57639262
17	1	$8.5 \cdot 10^{-6}$	0.56033183	0.57103902
18	1	$2.1 \cdot 10^{-6}$	0.56300863	0.56836222
19	1	$5.3 \cdot 10^{-7}$	0.56434703	0.56702382
20	1	$1.3 \cdot 10^{-7}$	0.56501623	0.56635462
21	1	$3.3 \cdot 10^{-8}$	0.56535083	0.56602002
22	1	$8.3 \cdot 10^{-9}$	0.56551813	0.56585272
23	1	$2.1 \cdot 10^{-9}$	0.56560178	0.56576907
24	1	$5.2 \cdot 10^{-10}$	0.56564360	0.56572725
25	1	$1.3 \cdot 10^{-10}$	0.56566451	0.56570634
26	1	$3.3 \cdot 10^{-11}$	0.56567497	0.56569588

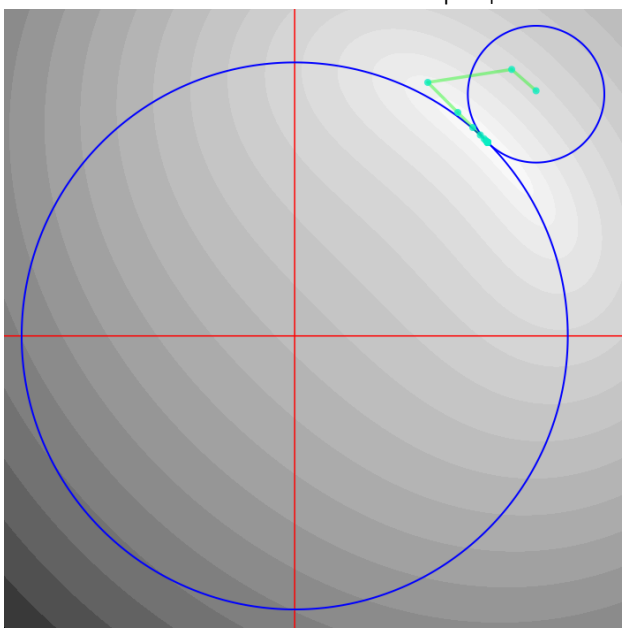
k	β	$\frac{\|f(x_k)\|}{f(x_0)}$	x	y
1	$1.5 \cdot 10^{-5}$	1	−1.42013487	−1.39823199
2	$6.1 \cdot 10^{-5}$	1	−1.43088224	−1.38724354
3	$2.4 \cdot 10^{-4}$	1	−1.45221533	−1.36494619
4	$9.8 \cdot 10^{-4}$	$1.0 \cdot 10^{0}$	−1.49388220	−1.31942335
5	$3.9 \cdot 10^{-3}$	$1.0 \cdot 10^{0}$	−1.57310819	−1.22478867
6	$1.6 \cdot 10^{-2}$	$9.9 \cdot 10^{-1}$	−1.71419400	−1.02230790
7	$6.3 \cdot 10^{-2}$	$9.6 \cdot 10^{-1}$	−1.92036738	−0.57439211
8	$2.5 \cdot 10^{-1}$	$8.6 \cdot 10^{-1}$	−1.99354020	0.40531465
9	1	$5.6 \cdot 10^{-1}$	−0.80483420	1.93620626
10	1	$1.4 \cdot 10^{-1}$	−0.11957460	1.25094544
11	1	$3.5 \cdot 10^{-2}$	0.22305570	0.90831553
12	1	$8.7 \cdot 10^{-3}$	0.39437046	0.73700036
13	1	$2.2 \cdot 10^{-3}$	0.48002795	0.65134290
14	1	$5.5 \cdot 10^{-4}$	0.52285669	0.60851416
15	1	$1.4 \cdot 10^{-4}$	0.54427106	0.58709979
16	1	$3.4 \cdot 10^{-5}$	0.55497824	0.57639261
17	1	$8.5 \cdot 10^{-6}$	0.56033183	0.57103902
18	1	$2.1 \cdot 10^{-6}$	0.56300862	0.56836223
19	1	$5.3 \cdot 10^{-7}$	0.56434702	0.56702383
20	1	$1.3 \cdot 10^{-7}$	0.56501623	0.56635462
21	1	$3.3 \cdot 10^{-8}$	0.56535083	0.56602002
22	1	$8.3 \cdot 10^{-9}$	0.56551813	0.56585272
23	1	$2.1 \cdot 10^{-9}$	0.56560177	0.56576908
24	1	$5.2 \cdot 10^{-10}$	0.56564360	0.56572725
25	1	$1.3 \cdot 10^{-10}$	0.56566452	0.56570633
26	1	$3.2 \cdot 10^{-11}$	0.56567497	0.56569588

2.2.2.3 Начальное приближение лежит в центре одной из окружностей

Комментарий: добавлено немного смещения, потому что на этой оси метод не сходится.

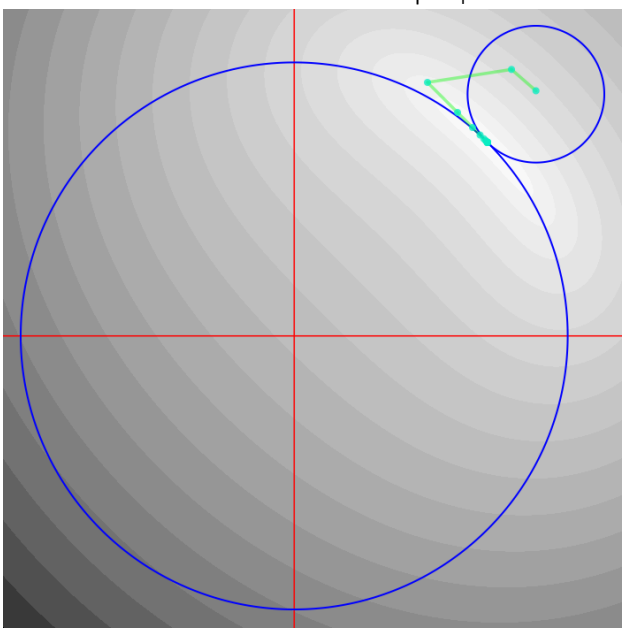
Вариант приведения к квадратному виду: —

Аналитическое вычисление матрицы Якоби



k	β	$\frac{\|\|f(\mathbf{x}_k)\|\|}{f(\mathbf{x}_0)}$	x	y
1	$3.1 \cdot 10^{-2}$	$9.9 \cdot 10^{-1}$	0.63561170	0.77945053
2	1	$2.3 \cdot 10^{-1}$	0.38984555	0.74152530
3	1	$5.8 \cdot 10^{-2}$	0.47776549	0.65360536
4	1	$1.5 \cdot 10^{-2}$	0.52172546	0.60964539
5	1	$3.6 \cdot 10^{-3}$	0.54370544	0.58766541
6	1	$9.1 \cdot 10^{-4}$	0.55469543	0.57667542
7	1	$2.3 \cdot 10^{-4}$	0.56019043	0.57118042
8	1	$5.7 \cdot 10^{-5}$	0.56293793	0.56843292
9	1	$1.4 \cdot 10^{-5}$	0.56431168	0.56705917
10	1	$3.6 \cdot 10^{-6}$	0.56499855	0.56637230
11	1	$8.9 \cdot 10^{-7}$	0.56534199	0.56602886
12	1	$2.2 \cdot 10^{-7}$	0.56551371	0.56585714
13	1	$5.5 \cdot 10^{-8}$	0.56559957	0.56577128
14	1	$1.4 \cdot 10^{-8}$	0.56564250	0.56572835
15	1	$3.5 \cdot 10^{-9}$	0.56566396	0.56570689
16	1	$8.7 \cdot 10^{-10}$	0.56567469	0.56569616
17	1	$2.2 \cdot 10^{-10}$	0.56568006	0.56569079
18	1	$5.4 \cdot 10^{-11}$	0.56568274	0.56568811

Численное вычисление матрицы Якоби

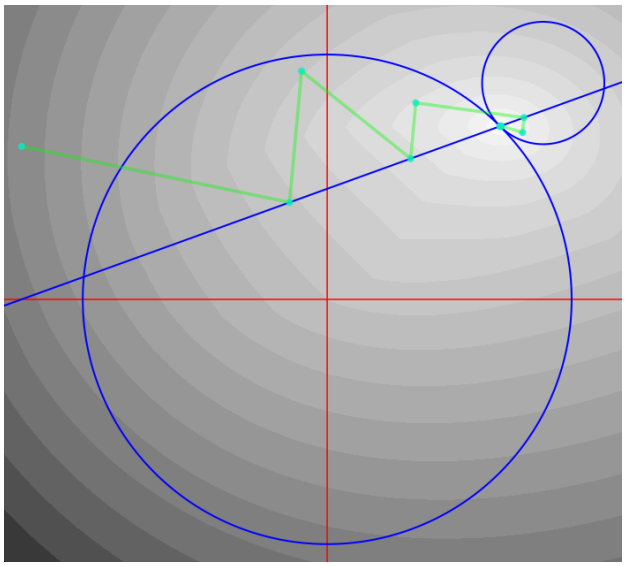


k	β	$\frac{\|\|f(\mathbf{x}_k)\|\|}{f(\mathbf{x}_0)}$	x	y
1	$3.1 \cdot 10^{-2}$	$9.9 \cdot 10^{-1}$	0.63561166	0.77945055
2	1	$2.3 \cdot 10^{-1}$	0.38984560	0.74152523
3	1	$5.8 \cdot 10^{-2}$	0.47776552	0.65360534
4	1	$1.5 \cdot 10^{-2}$	0.52172547	0.60964538
5	1	$3.6 \cdot 10^{-3}$	0.54370545	0.58766540
6	1	$9.1 \cdot 10^{-4}$	0.55469543	0.57667542
7	1	$2.3 \cdot 10^{-4}$	0.56019043	0.57118042
8	1	$5.7 \cdot 10^{-5}$	0.56293793	0.56843292
9	1	$1.4 \cdot 10^{-5}$	0.56431168	0.56705917
10	1	$3.6 \cdot 10^{-6}$	0.56499855	0.56637230
11	1	$8.9 \cdot 10^{-7}$	0.56534199	0.56602886
12	1	$2.2 \cdot 10^{-7}$	0.56551371	0.56585714
13	1	$5.5 \cdot 10^{-8}$	0.56559957	0.56577128
14	1	$1.4 \cdot 10^{-8}$	0.56564250	0.56572835
15	1	$3.5 \cdot 10^{-9}$	0.56566396	0.56570689
16	1	$8.7 \cdot 10^{-10}$	0.56567469	0.56569616
17	1	$2.2 \cdot 10^{-10}$	0.56568006	0.56569079
18	1	$5.5 \cdot 10^{-11}$	0.56568274	0.56568811

2.2.3 Добавлена ещё прямая

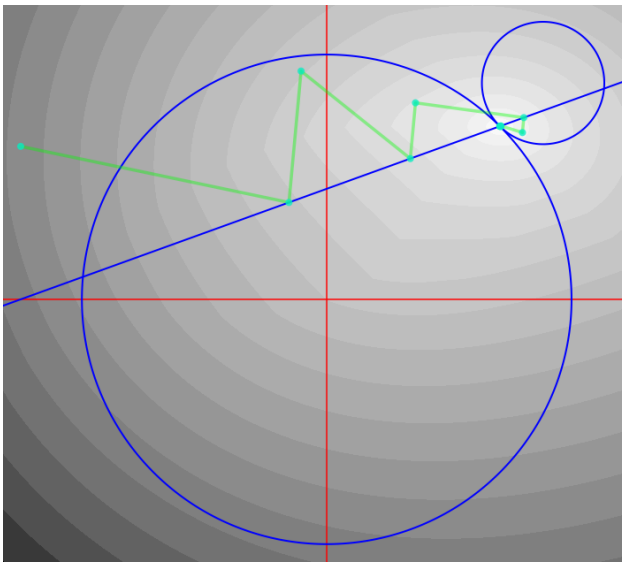
Вариант приведения к квадратному виду: 2

Аналитическое вычисление матрицы Якоби



k	β	$\frac{\|\|f(x_k)\|\|}{f(x_0)}$	x	y
1	1	$3.1 \cdot 10^{-1}$	-0.12336484	0.31673012
2	$5.0 \cdot 10^{-1}$	$3.1 \cdot 10^{-1}$	-0.08375629	0.74612436
3	1	$1.4 \cdot 10^{-1}$	0.27323391	0.46002208
4	$5.0 \cdot 10^{-1}$	$1.1 \cdot 10^{-1}$	0.29004484	0.64226857
5	1	$4.3 \cdot 10^{-2}$	0.64406414	0.59400382
6	$5.0 \cdot 10^{-1}$	$3.5 \cdot 10^{-2}$	0.63955871	0.54516069
7	1	$1.8 \cdot 10^{-3}$	0.56919908	0.56695492
8	$5.0 \cdot 10^{-1}$	$1.5 \cdot 10^{-3}$	0.56899710	0.56476532
9	1	$4.0 \cdot 10^{-6}$	0.56569307	0.56568819
10	$5.0 \cdot 10^{-1}$	$3.3 \cdot 10^{-6}$	0.56569263	0.56568342
11	1	$1.9 \cdot 10^{-11}$	0.56568543	0.56568543

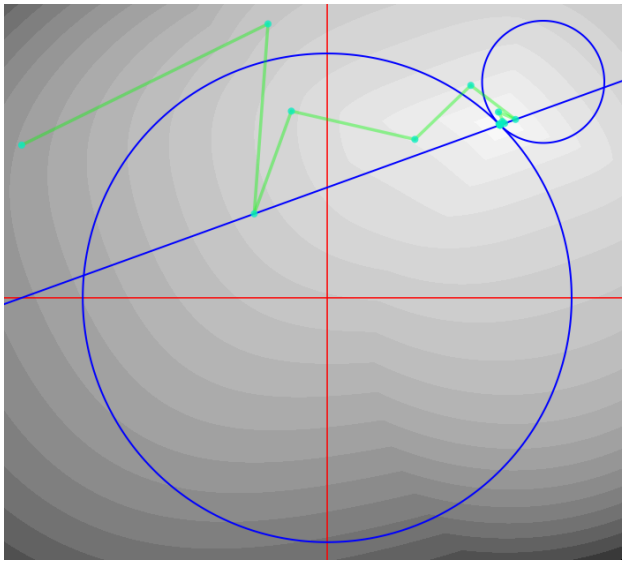
Численное вычисление матрицы Якоби



k	β	$\frac{\|\|f(x_k)\|\|}{f(x_0)}$	x	y
1	1	$3.1 \cdot 10^{-1}$	-0.12336491	0.31673017
2	$5.0 \cdot 10^{-1}$	$3.1 \cdot 10^{-1}$	-0.08375628	0.74612442
3	1	$1.4 \cdot 10^{-1}$	0.27323425	0.46002225
4	$5.0 \cdot 10^{-1}$	$1.1 \cdot 10^{-1}$	0.29004521	0.64226847
5	1	$4.3 \cdot 10^{-2}$	0.64406392	0.59400373
6	$5.0 \cdot 10^{-1}$	$3.5 \cdot 10^{-2}$	0.63955850	0.54516075
7	1	$1.8 \cdot 10^{-3}$	0.56919906	0.56695490
8	$5.0 \cdot 10^{-1}$	$1.5 \cdot 10^{-3}$	0.56899709	0.56476532
9	1	$4.0 \cdot 10^{-6}$	0.56569307	0.56568819
10	$5.0 \cdot 10^{-1}$	$3.3 \cdot 10^{-6}$	0.56569264	0.56568341
11	1	$1.9 \cdot 10^{-11}$	0.56568543	0.56568543

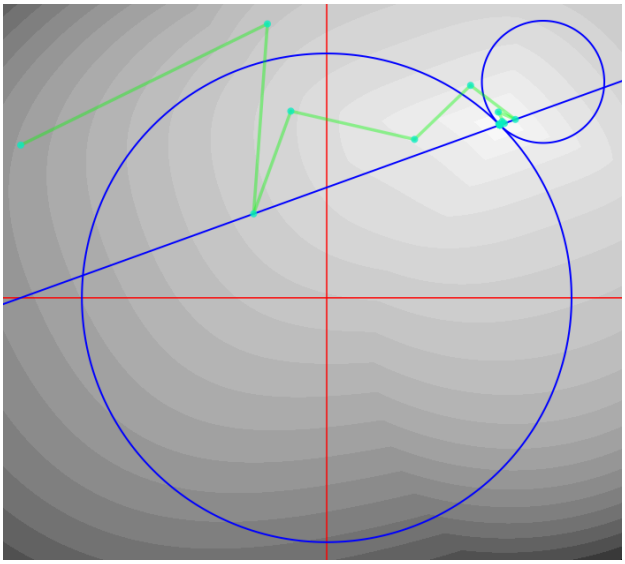
Вариант приведения к квадратному виду: 3

Аналитическое вычисление матрицы Якоби



k	β	$\frac{\|\|f(x_k)\|\|}{f(x_0)}$	x	y
1	1	$4.1 \cdot 10^{-1}$	-0.19374305	0.89683239
2	1	$3.4 \cdot 10^{-1}$	-0.23892040	0.27497965
3	$5.0 \cdot 10^{-1}$	$2.3 \cdot 10^{-1}$	-0.11725999	0.61129566
4	1	$9.3 \cdot 10^{-2}$	0.28698224	0.51878350
5	1	$9.3 \cdot 10^{-2}$	0.47006887	0.69556211
6	1	$2.6 \cdot 10^{-2}$	0.61640145	0.58400923
7	$5.0 \cdot 10^{-1}$	$2.5 \cdot 10^{-2}$	0.56051701	0.60831170
8	1	$7.4 \cdot 10^{-3}$	0.58066384	0.57109716
9	$6.3 \cdot 10^{-2}$	$7.0 \cdot 10^{-3}$	0.57328317	0.57732528
10	1	$2.8 \cdot 10^{-3}$	0.56218743	0.56932626
11	1	$3.7 \cdot 10^{-5}$	0.56576048	0.56571254
12	$2.4 \cdot 10^{-4}$	$3.7 \cdot 10^{-5}$	0.56573105	0.56574195
13	1	$1.3 \cdot 10^{-5}$	0.56566827	0.56570258
14	1	$1.0 \cdot 10^{-9}$	0.56568543	0.56568543
15	$4.8 \cdot 10^{-7}$	$1.0 \cdot 10^{-9}$	0.56568543	0.56568543
16	1	$4.5 \cdot 10^{-10}$	0.56568542	0.56568543
17	1	$2.3 \cdot 10^{-17}$	0.56568542	0.56568542

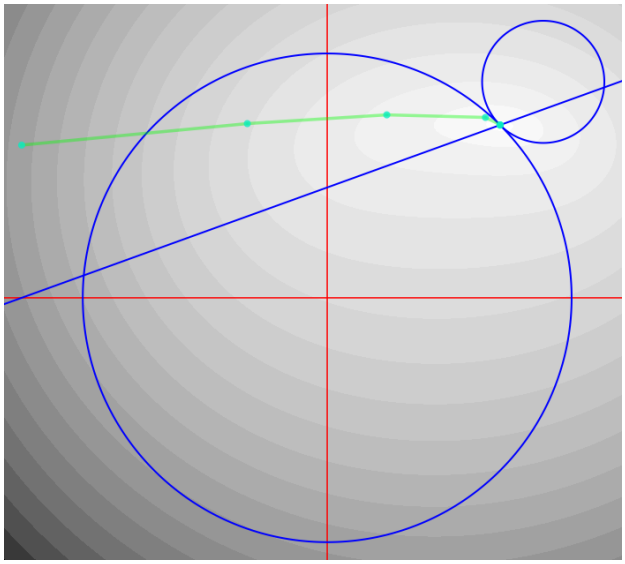
Численное вычисление матрицы Якоби



k	β	$\frac{\|\|f(x_k)\|\|}{f(x_0)}$	x	y
1	1	$4.1 \cdot 10^{-1}$	-0.19374310	0.89683235
2	1	$3.4 \cdot 10^{-1}$	-0.23892037	0.27497963
3	$5.0 \cdot 10^{-1}$	$2.3 \cdot 10^{-1}$	-0.11725985	0.61129543
4	1	$9.3 \cdot 10^{-2}$	0.28698234	0.51878318
5	1	$9.3 \cdot 10^{-2}$	0.47006835	0.69556248
6	1	$2.6 \cdot 10^{-2}$	0.61640183	0.58400938
7	$5.0 \cdot 10^{-1}$	$2.5 \cdot 10^{-2}$	0.56051859	0.60831034
8	1	$7.4 \cdot 10^{-3}$	0.58066380	0.57109714
9	$6.3 \cdot 10^{-2}$	$7.0 \cdot 10^{-3}$	0.57328259	0.57732580
10	1	$2.8 \cdot 10^{-3}$	0.56218719	0.56932650
11	1	$3.7 \cdot 10^{-5}$	0.56576049	0.56571255
12	$2.4 \cdot 10^{-4}$	$3.7 \cdot 10^{-5}$	0.56572310	0.56574991
13	1	$1.6 \cdot 10^{-5}$	0.56566481	0.56570604
14	1	$1.1 \cdot 10^{-9}$	0.56568543	0.56568543
15	$3.9 \cdot 10^{-3}$	$1.1 \cdot 10^{-9}$	0.56568543	0.56568543
16	1	$3.9 \cdot 10^{-10}$	0.56568542	0.56568543
17	1	$1.4 \cdot 10^{-16}$	0.56568542	0.56568542

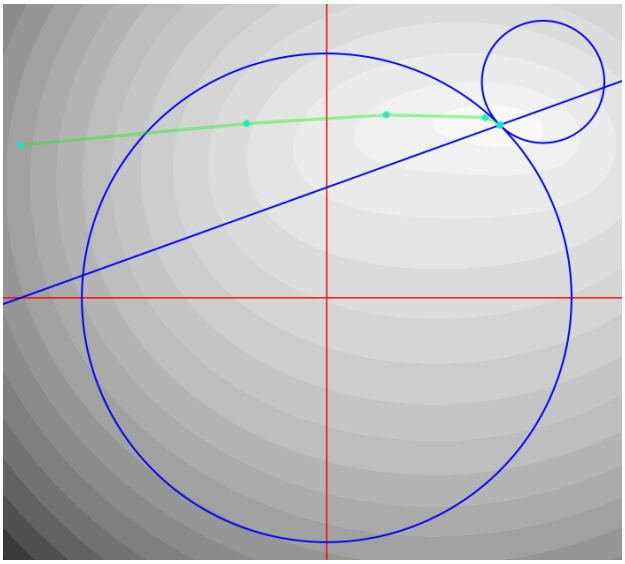
Вариант приведения к квадратному виду: 4

Аналитическое вычисление матрицы Якоби



k	β	$\frac{\|\|f(x_k)\|\|}{f(x_0)}$	x	y
1	1	$1.7 \cdot 10^{-1}$	-0.26143900	0.57036552
2	1	$4.7 \cdot 10^{-2}$	0.19508724	0.59905160
3	1	$1.1 \cdot 10^{-2}$	0.51796311	0.59063602
4	1	$2.9 \cdot 10^{-4}$	0.56679698	0.56624218
5	1	$1.6 \cdot 10^{-7}$	0.56568614	0.56568568
6	1	$5.8 \cdot 10^{-14}$	0.56568542	0.56568542

Численное вычисление матрицы Якоби

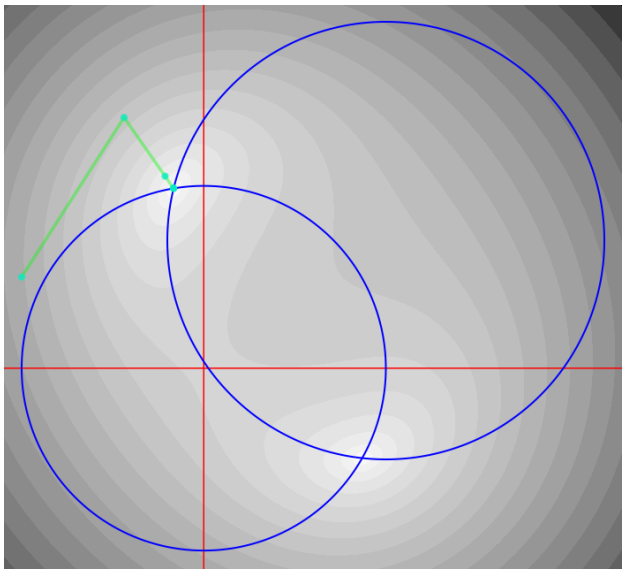


k	β	$\frac{\|\|f(x_k)\|\|}{f(x_0)}$	x	y
1	1	$1.7 \cdot 10^{-1}$	-0.26143906	0.57036549
2	1	$4.7 \cdot 10^{-2}$	0.19508720	0.59905164
3	1	$1.1 \cdot 10^{-2}$	0.51796311	0.59063601
4	1	$2.9 \cdot 10^{-4}$	0.56679698	0.56624218
5	1	$1.6 \cdot 10^{-7}$	0.56568614	0.56568568
6	1	$6.1 \cdot 10^{-14}$	0.56568542	0.56568542

2.2.4 Пересекаются в двух точках

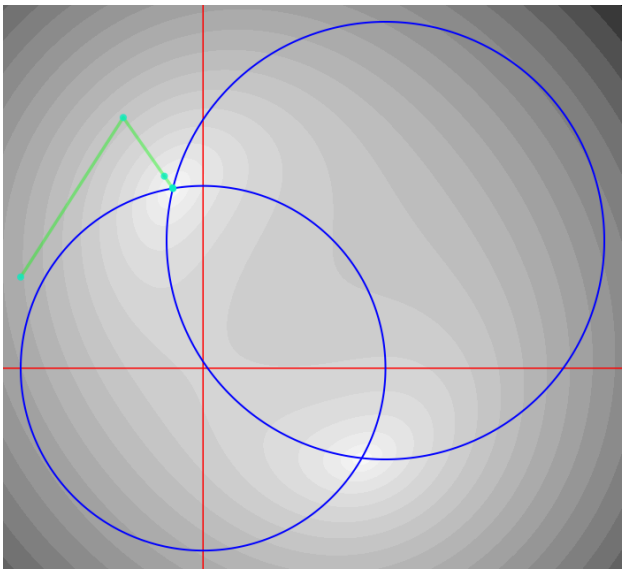
Вариант приведения к квадратному виду: $-$

Аналитическое вычисление матрицы Якоби



k	β	$\frac{\|\|f(x_k)\|\|}{f(x_0)}$	x	y
1	1	$5.9 \cdot 10^{-1}$	-0.43750000	1.37500000
2	1	$8.4 \cdot 10^{-2}$	-0.21224442	1.05320632
3	1	$3.3 \cdot 10^{-3}$	-0.16730937	0.98901338
4	1	$6.2 \cdot 10^{-6}$	-0.16536659	0.98623799
5	1	$2.2 \cdot 10^{-11}$	-0.16536295	0.98623278

Численное вычисление матрицы Якоби



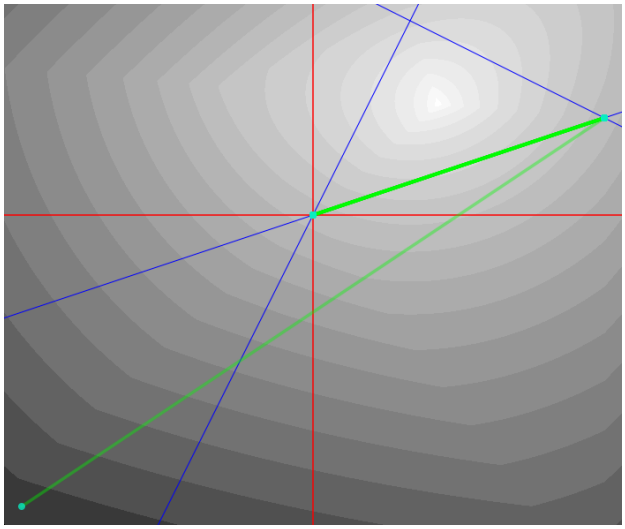
k	β	$\frac{\|\|f(x_k)\|\|}{f(x_0)}$	x	y
1	1	$5.9 \cdot 10^{-1}$	-0.43750005	1.37499993
2	1	$8.4 \cdot 10^{-2}$	-0.21224451	1.05320633
3	1	$3.3 \cdot 10^{-3}$	-0.16730936	0.98901339
4	1	$6.2 \cdot 10^{-6}$	-0.16536659	0.98623799
5	1	$2.2 \cdot 10^{-11}$	-0.16536295	0.98623278

2.3 Три попарно пересекающиеся прямые

2.3.1 Невзвешенный вариант

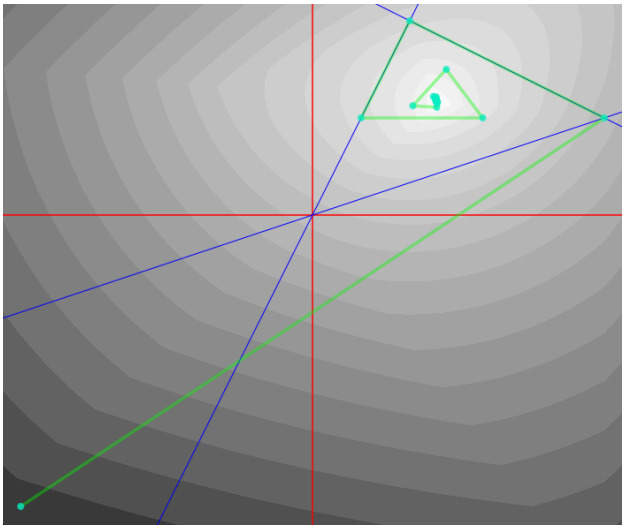
Вариант приведения к квадратному виду: 2

Аналитическое вычисление матрицы Якоби



k	β	$\frac{\|\|f(x_k)\|\|}{f(x_0)}$	x	y
1	1	$3.4 \cdot 10^{-1}$	3.00000000	1.00000000
2	1	$3.4 \cdot 10^{-1}$	0.00000000	0.00000000
3	1	$3.4 \cdot 10^{-1}$	3.00000000	1.00000000
4	1	$3.4 \cdot 10^{-1}$	0.00000000	0.00000000
5	1	$3.4 \cdot 10^{-1}$	3.00000000	1.00000000
6	1	$3.4 \cdot 10^{-1}$	0.00000000	0.00000000
7	1	$3.4 \cdot 10^{-1}$	3.00000000	1.00000000
8	1	$3.4 \cdot 10^{-1}$	0.00000000	0.00000000
9	1	$3.4 \cdot 10^{-1}$	3.00000000	1.00000000
10	1	$3.4 \cdot 10^{-1}$	0.00000000	0.00000000
11	1	$3.4 \cdot 10^{-1}$	3.00000000	1.00000000
12	1	$3.4 \cdot 10^{-1}$	0.00000000	0.00000000
13	1	$3.4 \cdot 10^{-1}$	3.00000000	1.00000000
14	1	$3.4 \cdot 10^{-1}$	0.00000000	0.00000000
15	1	$3.4 \cdot 10^{-1}$	3.00000000	1.00000000
16	1	$3.4 \cdot 10^{-1}$	0.00000000	0.00000000
17	1	$3.4 \cdot 10^{-1}$	3.00000000	1.00000000
18	1	$3.4 \cdot 10^{-1}$	0.00000000	0.00000000
19	1	$3.4 \cdot 10^{-1}$	3.00000000	1.00000000
20	1	$3.4 \cdot 10^{-1}$	0.00000000	0.00000000
21	1	$3.4 \cdot 10^{-1}$	3.00000000	1.00000000
22	1	$3.4 \cdot 10^{-1}$	0.00000000	0.00000000
23	1	$3.4 \cdot 10^{-1}$	3.00000000	1.00000000
24	1	$3.4 \cdot 10^{-1}$	0.00000000	0.00000000
25	1	$3.4 \cdot 10^{-1}$	3.00000000	1.00000000
26	1	$3.4 \cdot 10^{-1}$	0.00000000	0.00000000
27	1	$3.4 \cdot 10^{-1}$	3.00000000	1.00000000
28	1	$3.4 \cdot 10^{-1}$	0.00000000	0.00000000
29	1	$3.4 \cdot 10^{-1}$	3.00000000	1.00000000
30	1	$3.4 \cdot 10^{-1}$	0.00000000	0.00000000
31	1	$3.4 \cdot 10^{-1}$	3.00000000	1.00000000

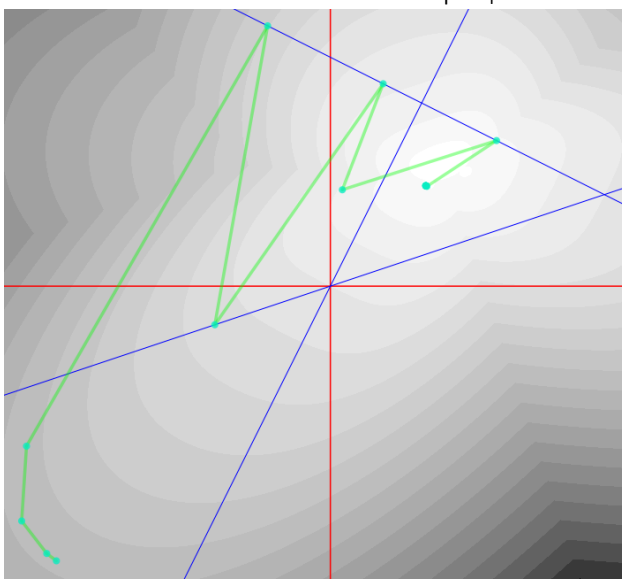
Численное вычисление матрицы Якоби



k	β	$\frac{\|\|f(x_k)\|\|}{f(x_0)}$	x	y
1	1	$3.4 \cdot 10^{-1}$	2.99999933	0.99999976
2	1	$2.3 \cdot 10^{-1}$	1.00000020	2.00000003
3	$5.0 \cdot 10^{-1}$	$2.1 \cdot 10^{-1}$	0.50000017	1.00000013
4	$5.0 \cdot 10^{-1}$	$1.9 \cdot 10^{-1}$	1.74999975	0.99999984
5	$5.0 \cdot 10^{-1}$	$1.7 \cdot 10^{-1}$	1.37499987	1.49999980
6	$2.5 \cdot 10^{-1}$	$1.6 \cdot 10^{-1}$	1.03124993	1.12499992
7	$1.3 \cdot 10^{-1}$	$1.4 \cdot 10^{-1}$	1.27734363	1.10937495
8	$1.3 \cdot 10^{-1}$	$1.4 \cdot 10^{-1}$	1.24267568	1.22070307
9	$1.6 \cdot 10^{-2}$	$1.4 \cdot 10^{-1}$	1.27013387	1.21725459
10	$7.8 \cdot 10^{-3}$	$1.4 \cdot 10^{-1}$	1.26021094	1.20774479
11	$3.9 \cdot 10^{-3}$	$1.4 \cdot 10^{-1}$	1.26700699	1.20693328
12	$3.9 \cdot 10^{-3}$	$1.4 \cdot 10^{-1}$	1.26205775	1.20221870
13	$3.9 \cdot 10^{-3}$	$1.4 \cdot 10^{-1}$	1.26884659	1.20142878
14	$3.9 \cdot 10^{-3}$	$1.4 \cdot 10^{-1}$	1.26389015	1.19673570
15	$3.9 \cdot 10^{-3}$	$1.4 \cdot 10^{-1}$	1.27067183	1.19596720
16	$3.9 \cdot 10^{-3}$	$1.4 \cdot 10^{-1}$	1.26570827	1.19129546
17	$3.9 \cdot 10^{-3}$	$1.4 \cdot 10^{-1}$	1.27248285	1.19054821
18	$3.9 \cdot 10^{-3}$	$1.4 \cdot 10^{-1}$	1.26751221	1.18589763
19	$3.9 \cdot 10^{-3}$	$1.4 \cdot 10^{-1}$	1.27427974	1.18517147
20	$3.9 \cdot 10^{-3}$	$1.4 \cdot 10^{-1}$	1.26930209	1.18054189
21	$3.9 \cdot 10^{-3}$	$1.4 \cdot 10^{-1}$	1.27606263	1.17983665
22	$3.9 \cdot 10^{-3}$	$1.4 \cdot 10^{-1}$	1.27107801	1.17522791
23	$3.9 \cdot 10^{-3}$	$1.4 \cdot 10^{-1}$	1.27783161	1.17454343
24	$3.9 \cdot 10^{-3}$	$1.4 \cdot 10^{-1}$	1.27284008	1.16995537
25	$3.9 \cdot 10^{-3}$	$1.4 \cdot 10^{-1}$	1.27958680	1.16929148
26	$3.9 \cdot 10^{-3}$	$1.4 \cdot 10^{-1}$	1.27458841	1.16472393
27	$3.9 \cdot 10^{-3}$	$1.4 \cdot 10^{-1}$	1.28132830	1.16408048
28	$3.9 \cdot 10^{-3}$	$1.4 \cdot 10^{-1}$	1.27632311	1.15953329
29	$3.9 \cdot 10^{-3}$	$1.4 \cdot 10^{-1}$	1.28305623	1.15891011
30	$2.0 \cdot 10^{-3}$	$1.4 \cdot 10^{-1}$	1.28055026	1.15664662
31	$4.9 \cdot 10^{-4}$	$1.4 \cdot 10^{-1}$	1.28138983	1.15657013

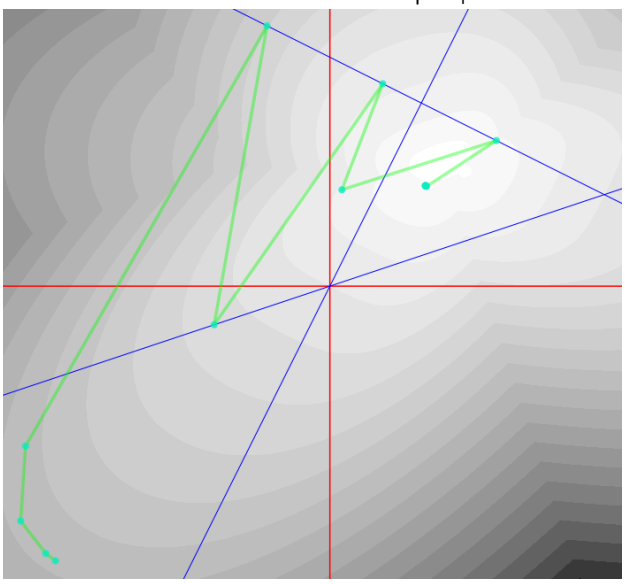
Вариант приведения к квадратному виду: 3

Аналитическое вычисление матрицы Якоби



k	β	$\frac{\|\|f(x_k)\|\|}{f(x_0)}$	x	y
1	$3.9 \cdot 10^{-3}$	$1.0 \cdot 10^{0}$	-3.10546875	-2.91992188
2	$3.1 \cdot 10^{-2}$	$1.0 \cdot 10^{0}$	-3.38021506	-2.56465321
3	$1.3 \cdot 10^{-1}$	$9.3 \cdot 10^{-1}$	-3.32693889	-1.74694621
4	1	$5.8 \cdot 10^{-1}$	-0.68686359	2.84343180
5	1	$4.0 \cdot 10^{-1}$	-1.26514770	-0.42171590
6	1	$2.2 \cdot 10^{-1}$	0.57828410	2.21085795
7	$5.0 \cdot 10^{-1}$	$1.9 \cdot 10^{-1}$	0.13099859	1.05271449
8	1	$1.5 \cdot 10^{-1}$	1.81980880	1.59009560
9	$5.0 \cdot 10^{-1}$	$1.3 \cdot 10^{-1}$	1.05742830	1.09009560
10	$4.9 \cdot 10^{-4}$	$1.3 \cdot 10^{-1}$	1.04176218	1.09835893
11	$6.1 \cdot 10^{-5}$	$1.3 \cdot 10^{-1}$	1.04684040	1.09587357
12	$9.5 \cdot 10^{-7}$	$1.3 \cdot 10^{-1}$	1.04616630	1.09621147
13	$1.2 \cdot 10^{-7}$	$1.3 \cdot 10^{-1}$	1.04646869	1.09606038
14	$1.2 \cdot 10^{-7}$	$1.3 \cdot 10^{-1}$	1.04618208	1.09620378
15	$6.0 \cdot 10^{-8}$	$1.3 \cdot 10^{-1}$	1.04635127	1.09611924
16	$7.5 \cdot 10^{-9}$	$1.3 \cdot 10^{-1}$	1.04627744	1.09615616
17	$3.7 \cdot 10^{-9}$	$1.3 \cdot 10^{-1}$	1.04631587	1.09613695
18	$1.5 \cdot 10^{-11}$	$1.3 \cdot 10^{-1}$	1.04631345	1.09613816

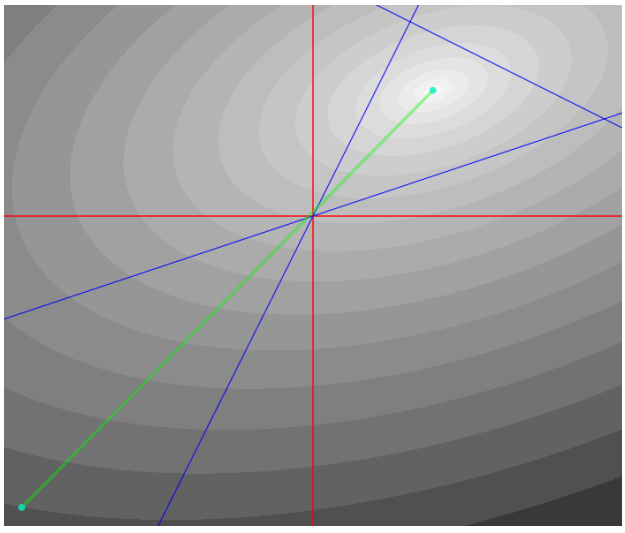
Численное вычисление матрицы Якоби



k	β	$\frac{\|\|f(x_k)\|\|}{f(x_0)}$	x	y
1	$3.9 \cdot 10^{-3}$	$1.0 \cdot 10^{0}$	-3.10546871	-2.91992190
2	$3.1 \cdot 10^{-2}$	$1.0 \cdot 10^{0}$	-3.38021499	-2.56465331
3	$1.3 \cdot 10^{-1}$	$9.3 \cdot 10^{-1}$	-3.32693878	-1.74694605
4	1	$5.8 \cdot 10^{-1}$	-0.68686422	2.84343073
5	1	$4.0 \cdot 10^{-1}$	-1.26514509	-0.42171483
6	1	$2.2 \cdot 10^{-1}$	0.57828515	2.21085830
7	$5.0 \cdot 10^{-1}$	$1.9 \cdot 10^{-1}$	0.13099663	1.05271370
8	1	$1.5 \cdot 10^{-1}$	1.81980515	1.59009501
9	$5.0 \cdot 10^{-1}$	$1.3 \cdot 10^{-1}$	1.05742535	1.09009293
10	$4.9 \cdot 10^{-4}$	$1.3 \cdot 10^{-1}$	1.04175894	1.09835640
11	$6.1 \cdot 10^{-5}$	$1.3 \cdot 10^{-1}$	1.04683655	1.09587135
12	$9.5 \cdot 10^{-7}$	$1.3 \cdot 10^{-1}$	1.04616094	1.09621000
13	$1.2 \cdot 10^{-7}$	$1.3 \cdot 10^{-1}$	1.04645801	1.09606157
14	$1.2 \cdot 10^{-7}$	$1.3 \cdot 10^{-1}$	1.04615594	1.09621271
15	$1.2 \cdot 10^{-7}$	$1.3 \cdot 10^{-1}$	1.04644307	1.09606925
16	$6.0 \cdot 10^{-8}$	$1.3 \cdot 10^{-1}$	1.04627477	1.09615345
17	$3.7 \cdot 10^{-9}$	$1.3 \cdot 10^{-1}$	1.04631312	1.09613428
18	$1.5 \cdot 10^{-11}$	$1.3 \cdot 10^{-1}$	1.04631055	1.09613557

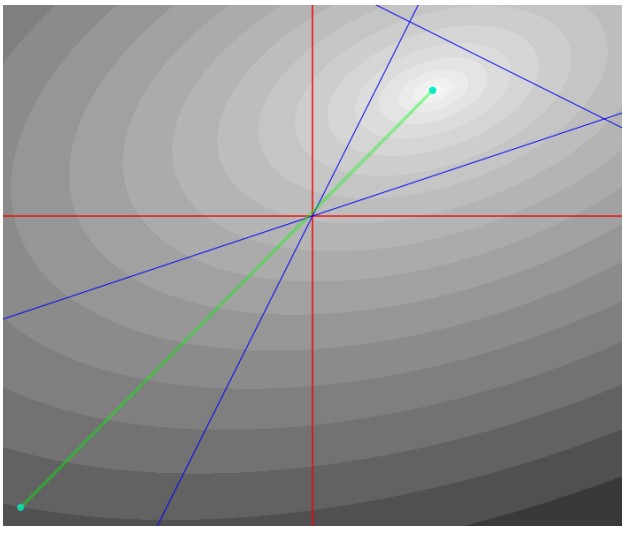
Вариант приведения к квадратному виду: 4

Аналитическое вычисление матрицы Якоби



k	β	$\frac{\|\|f(x_k)\|\|}{f(x_0)}$	x	y
1	1	$2.6 \cdot 10^{-16}$	1.23529412	1.29411765

Численное вычисление матрицы Якоби

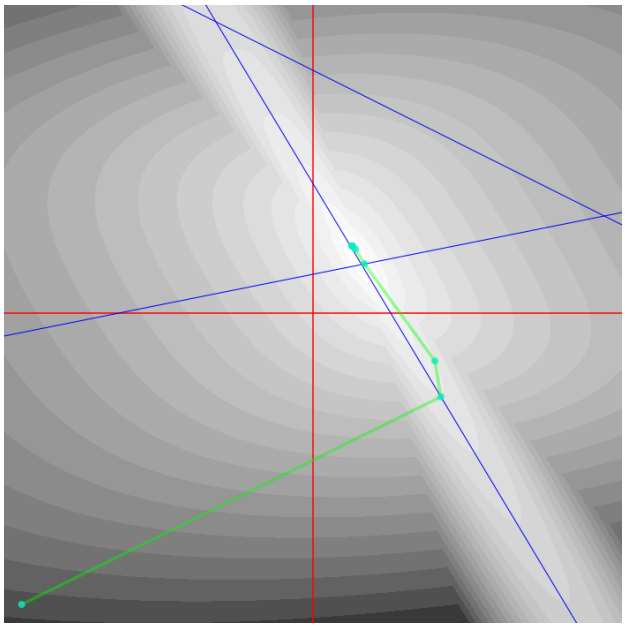


k	β	$\frac{\|\|f(x_k)\|\|}{f(x_0)}$	x	y
1	1	$4.8 \cdot 10^{-8}$	1.23529397	1.29411743
2	1	$5.7 \cdot 10^{-9}$	1.23529412	1.29411765
3	1	$4.7 \cdot 10^{-16}$	1.23529408	1.29411764

2.3.2 Взвешенный вариант

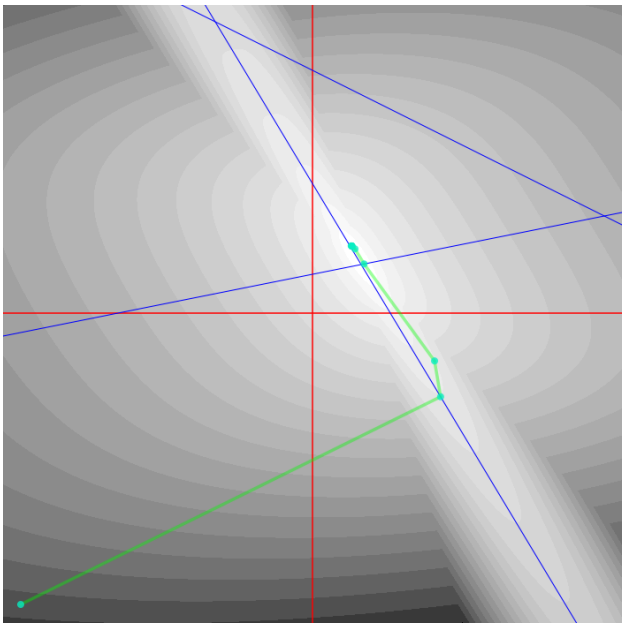
Вариант приведения к квадратному виду: 3

Аналитическое вычисление матрицы Якоби



k	β	$\frac{\|\|f(x_k)\|\|}{f(x_0)}$	x	y
1	1	$6.2 \cdot 10^{-2}$	1.31672598	−0.86120996
2	$2.5 \cdot 10^{-1}$	$5.9 \cdot 10^{-2}$	1.25489324	−0.49243772
3	1	$1.5 \cdot 10^{-2}$	0.53048450	0.50609690
4	$6.3 \cdot 10^{-2}$	$1.4 \cdot 10^{-2}$	0.43635344	0.66120373
5	$7.8 \cdot 10^{-3}$	$1.4 \cdot 10^{-2}$	0.39827532	0.69290388
6	$4.9 \cdot 10^{-4}$	$1.4 \cdot 10^{-2}$	0.40454734	0.69055301
7	$1.9 \cdot 10^{-9}$	$1.4 \cdot 10^{-2}$	0.40453532	0.69055902
8	$1.2 \cdot 10^{-10}$	$1.4 \cdot 10^{-2}$	0.40453123	0.69056107
9	$2.9 \cdot 10^{-11}$	$1.4 \cdot 10^{-2}$	0.40453320	0.69056008

Численное вычисление матрицы Якоби

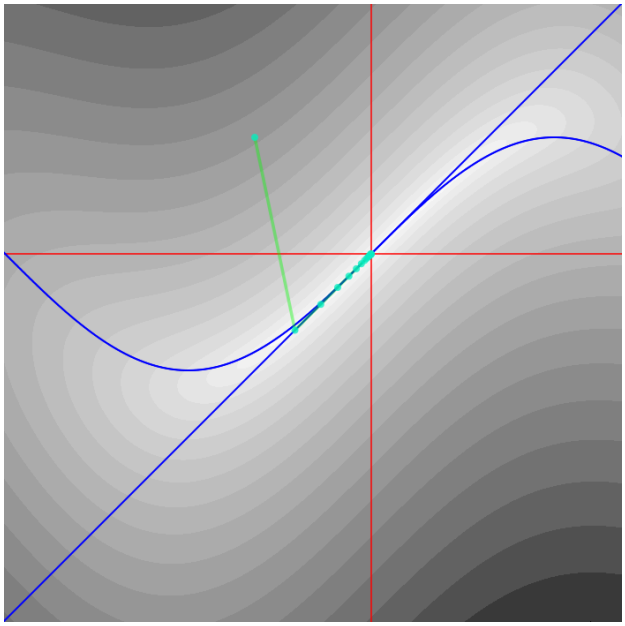


k	β	$\frac{\|\|f(x_k)\|\|}{f(x_0)}$	x	y
1	1	$6.2 \cdot 10^{-2}$	1.31672652	−0.86121005
2	$2.5 \cdot 10^{-1}$	$5.9 \cdot 10^{-2}$	1.25490123	−0.49243625
3	1	$1.5 \cdot 10^{-2}$	0.53049098	0.50609800
4	$6.3 \cdot 10^{-2}$	$1.4 \cdot 10^{-2}$	0.43636001	0.66120445
5	$7.8 \cdot 10^{-3}$	$1.4 \cdot 10^{-2}$	0.39829292	0.69289907
6	$4.9 \cdot 10^{-4}$	$1.4 \cdot 10^{-2}$	0.40458552	0.69053791
7	$3.0 \cdot 10^{-8}$	$1.4 \cdot 10^{-2}$	0.40453458	0.69056343
8	$2.3 \cdot 10^{-10}$	$1.4 \cdot 10^{-2}$	0.40452981	0.69056582
9	$2.3 \cdot 10^{-13}$	$1.4 \cdot 10^{-2}$	0.40452995	0.69056574

2.4 Прямая и синусоида

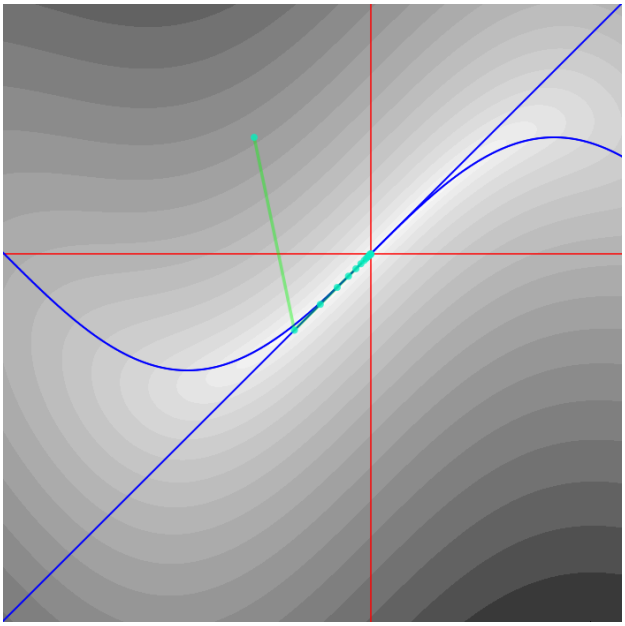
Вариант приведения к квадратному виду: —

Аналитическое вычисление матрицы Якоби



k	β	$\frac{\|f(x_k)\|}{f(x_0)}$	x	y
1	1	$2.2 \cdot 10^{-2}$	−0.65514507	−0.65514507
2	1	$6.4 \cdot 10^{-3}$	−0.43359037	−0.43359037
3	1	$1.9 \cdot 10^{-3}$	−0.28814840	−0.28814840
4	1	$5.6 \cdot 10^{-4}$	−0.19183231	−0.19183231
5	1	$1.7 \cdot 10^{-4}$	−0.12780967	−0.12780967
6	1	$4.9 \cdot 10^{-5}$	−0.08518323	−0.08518323
7	1	$1.5 \cdot 10^{-5}$	−0.05678195	−0.05678195
8	1	$4.3 \cdot 10^{-6}$	−0.03785260	−0.03785260
9	1	$1.3 \cdot 10^{-6}$	−0.02523446	−0.02523446
10	1	$3.8 \cdot 10^{-7}$	−0.01682280	−0.01682280
11	1	$1.1 \cdot 10^{-7}$	−0.01121515	−0.01121515
12	1	$3.3 \cdot 10^{-8}$	−0.00747675	−0.00747675
13	1	$9.9 \cdot 10^{-9}$	−0.00498449	−0.00498449
14	1	$2.9 \cdot 10^{-9}$	−0.00332299	−0.00332299
15	1	$8.7 \cdot 10^{-10}$	−0.00221533	−0.00221533
16	1	$2.6 \cdot 10^{-10}$	−0.00147689	−0.00147689
17	1	$7.6 \cdot 10^{-11}$	−0.00098459	−0.00098459

Численное вычисление матрицы Якоби



k	β	$\frac{\|f(x_k)\|}{f(x_0)}$	x	y
1	1	$2.2 \cdot 10^{-2}$	−0.65514518	−0.65514501
2	1	$6.4 \cdot 10^{-3}$	−0.43359046	−0.43359048
3	1	$1.9 \cdot 10^{-3}$	−0.28814855	−0.28814857
4	1	$5.6 \cdot 10^{-4}$	−0.19183240	−0.19183240
5	1	$1.7 \cdot 10^{-4}$	−0.12780984	−0.12780984
6	1	$4.9 \cdot 10^{-5}$	−0.08518376	−0.08518376
7	1	$1.5 \cdot 10^{-5}$	−0.05678290	−0.05678290
8	1	$4.3 \cdot 10^{-6}$	−0.03785306	−0.03785306
9	1	$1.3 \cdot 10^{-6}$	−0.02523546	−0.02523546
10	1	$3.8 \cdot 10^{-7}$	−0.01682381	−0.01682381
11	1	$1.1 \cdot 10^{-7}$	−0.01121390	−0.01121390
12	1	$3.3 \cdot 10^{-8}$	−0.00747179	−0.00747179
13	1	$9.9 \cdot 10^{-9}$	−0.00498678	−0.00498678
14	1	$2.9 \cdot 10^{-9}$	−0.00331311	−0.00331311
15	1	$8.6 \cdot 10^{-10}$	−0.00221063	−0.00221063
16	1	$2.4 \cdot 10^{-10}$	−0.00144581	−0.00144581
17	1	$7.2 \cdot 10^{-11}$	−0.00096602	−0.00096602

2.5 Влияние размера шага при численном вычислении производной на сходимость метода

Тест производится на системе из двух окружностей с прямой, вариант преобразования к квадратному виду: 3, вычисление матрицы Якоби: численное.

$h = 1$

k	β	$\frac{\|\|f(\mathbf{x}_k)\|\|}{f(\mathbf{x}_0)}$	x	y
1	1	$5.5 \cdot 10^{-1}$	−0.33111565	0.91372015
2	$5.0 \cdot 10^{-1}$	$5.1 \cdot 10^{-1}$	−0.54566562	0.50017738
3	1	$2.0 \cdot 10^{-1}$	−0.03233864	0.43078527
4	1	$1.3 \cdot 10^{-1}$	0.33028297	0.71906106
5	$5.0 \cdot 10^{-1}$	$8.1 \cdot 10^{-2}$	0.26029889	0.57456215
6	1	$4.4 \cdot 10^{-2}$	0.48266435	0.61429001
7	1	$5.2 \cdot 10^{-3}$	0.55501826	0.56183136
8	$1.6 \cdot 10^{-2}$	$5.1 \cdot 10^{-3}$	0.55933821	0.55771490
9	1	$1.7 \cdot 10^{-3}$	0.56352198	0.56313848
10	1	$5.6 \cdot 10^{-4}$	0.56490990	0.56489747
11	1	$1.9 \cdot 10^{-4}$	0.56538403	0.56546638
12	1	$6.3 \cdot 10^{-5}$	0.56554456	0.56565289
13	1	$2.2 \cdot 10^{-5}$	0.56567215	0.56564091
14	1	$1.3 \cdot 10^{-5}$	0.56565867	0.56567576
15	$3.1 \cdot 10^{-5}$	$1.3 \cdot 10^{-5}$	0.56566725	0.56566717
16	1	$4.4 \cdot 10^{-6}$	0.56567831	0.56568040
17	1	$1.5 \cdot 10^{-6}$	0.56568203	0.56568477
18	1	$4.9 \cdot 10^{-7}$	0.56568486	0.56568465
19	1	$1.6 \cdot 10^{-7}$	0.56568524	0.56568516
20	1	$5.4 \cdot 10^{-8}$	0.56568536	0.56568534
21	1	$1.8 \cdot 10^{-8}$	0.56568540	0.56568540
22	1	$6.0 \cdot 10^{-9}$	0.56568542	0.56568542
23	1	$2.0 \cdot 10^{-9}$	0.56568542	0.56568542
24	1	$6.7 \cdot 10^{-10}$	0.56568542	0.56568542
25	1	$2.3 \cdot 10^{-10}$	0.56568542	0.56568542
26	1	$1.4 \cdot 10^{-10}$	0.56568542	0.56568542
27	$1.2 \cdot 10^{-7}$	$1.4 \cdot 10^{-10}$	0.56568542	0.56568542
28	1	$4.7 \cdot 10^{-11}$	0.56568542	0.56568542

$h = 2^{-1}$

k	β	$\frac{\|\|f(\mathbf{x}_k)\|\|}{f(\mathbf{x}_0)}$	x	y
1	1	$4.8 \cdot 10^{-1}$	−0.27179090	0.90642712
2	$5.0 \cdot 10^{-1}$	$4.0 \cdot 10^{-1}$	−0.39294723	0.54099126
3	1	$1.5 \cdot 10^{-1}$	0.11002810	0.41613209
4	$5.0 \cdot 10^{-1}$	$1.2 \cdot 10^{-1}$	0.15015298	0.62407259
5	1	$6.5 \cdot 10^{-2}$	0.41714732	0.51201830
6	1	$4.0 \cdot 10^{-2}$	0.49234304	0.60998834
7	1	$4.3 \cdot 10^{-3}$	0.55690853	0.56251432
8	$3.1 \cdot 10^{-2}$	$4.2 \cdot 10^{-3}$	0.56058385	0.55917407
9	1	$1.1 \cdot 10^{-3}$	0.56546801	0.56362717
10	1	$4.7 \cdot 10^{-4}$	0.56472699	0.56533914
11	$3.9 \cdot 10^{-3}$	$4.7 \cdot 10^{-4}$	0.56519669	0.56487402
12	1	$1.6 \cdot 10^{-4}$	0.56570307	0.56540832
13	1	$5.4 \cdot 10^{-5}$	0.56557589	0.56564585
14	$4.9 \cdot 10^{-4}$	$5.4 \cdot 10^{-5}$	0.56563474	0.56558706
15	1	$2.1 \cdot 10^{-5}$	0.56569033	0.56565072
16	1	$6.2 \cdot 10^{-6}$	0.56567284	0.56568088
17	$6.1 \cdot 10^{-5}$	$6.2 \cdot 10^{-6}$	0.56568020	0.56567352
18	1	$2.6 \cdot 10^{-6}$	0.56568631	0.56568111
19	1	$7.1 \cdot 10^{-7}$	0.56568398	0.56568490
20	$7.6 \cdot 10^{-6}$	$7.1 \cdot 10^{-7}$	0.56568490	0.56568398
21	1	$3.3 \cdot 10^{-7}$	0.56568557	0.56568489
22	1	$8.2 \cdot 10^{-8}$	0.56568526	0.56568536
23	$4.8 \cdot 10^{-7}$	$8.2 \cdot 10^{-8}$	0.56568532	0.56568531
24	1	$1.7 \cdot 10^{-8}$	0.56568541	0.56568539
25	1	$9.4 \cdot 10^{-9}$	0.56568541	0.56568542
26	$1.2 \cdot 10^{-7}$	$9.4 \cdot 10^{-9}$	0.56568542	0.56568541
27	1	$3.2 \cdot 10^{-9}$	0.56568543	0.56568542
28	1	$1.1 \cdot 10^{-9}$	0.56568542	0.56568542
29	$1.2 \cdot 10^{-7}$	$1.1 \cdot 10^{-9}$	0.56568542	0.56568542
30	1	$3.7 \cdot 10^{-10}$	0.56568543	0.56568542
31	1	$1.2 \cdot 10^{-10}$	0.56568542	0.56568542

$h = 2^{-2}$

k	β	$\frac{\|\|f(\mathbf{x}_k)\|\|}{f(\mathbf{x}_0)}$	x	y
1	1	$4.4 \cdot 10^{-1}$	−0.23562082	0.90198059
2	1	$4.4 \cdot 10^{-1}$	−0.37948764	0.22419242
3	$5.0 \cdot 10^{-1}$	$3.2 \cdot 10^{-1}$	−0.25187015	0.66111384
4	1	$9.6 \cdot 10^{-2}$	0.19002880	0.57399703
5	1	$9.2 \cdot 10^{-2}$	0.46914539	0.69407201
6	1	$2.7 \cdot 10^{-2}$	0.61770022	0.58447848
7	$2.5 \cdot 10^{-1}$	$2.1 \cdot 10^{-2}$	0.62458817	0.56138841
8	1	$6.8 \cdot 10^{-3}$	0.58032312	0.55896536
9	1	$1.6 \cdot 10^{-3}$	0.56900194	0.56688369
10	$6.3 \cdot 10^{-2}$	$1.5 \cdot 10^{-3}$	0.56989335	0.56573805
11	1	$4.0 \cdot 10^{-4}$	0.56656292	0.56529223
12	1	$9.5 \cdot 10^{-5}$	0.56587832	0.56575512
13	$7.8 \cdot 10^{-3}$	$9.4 \cdot 10^{-5}$	0.56598337	0.56564822
14	1	$4.0 \cdot 10^{-5}$	0.56575944	0.56564043
15	1	$5.7 \cdot 10^{-6}$	0.56569695	0.56568959
16	$4.9 \cdot 10^{-4}$	$5.7 \cdot 10^{-6}$	0.56570350	0.56568304
17	1	$2.5 \cdot 10^{-6}$	0.56568995	0.56568264
18	1	$3.4 \cdot 10^{-7}$	0.56568612	0.56568568
19	$3.1 \cdot 10^{-5}$	$3.4 \cdot 10^{-7}$	0.56568653	0.56568527
20	1	$1.6 \cdot 10^{-7}$	0.56568570	0.56568525
21	1	$2.0 \cdot 10^{-8}$	0.56568547	0.56568544
22	$1.9 \cdot 10^{-6}$	$2.0 \cdot 10^{-8}$	0.56568549	0.56568541
23	1	$9.8 \cdot 10^{-9}$	0.56568544	0.56568541
24	1	$1.2 \cdot 10^{-9}$	0.56568543	0.56568543
25	$1.2 \cdot 10^{-7}$	$1.2 \cdot 10^{-9}$	0.56568543	0.56568543
26	1	$4.1 \cdot 10^{-10}$	0.56568542	0.56568543
27	1	$7.4 \cdot 10^{-11}$	0.56568543	0.56568543

$h = 2^{-3}$

k	β	$\frac{\|\|f(\mathbf{x}_k)\|\|}{f(\mathbf{x}_0)}$	x	y
1	1	$4.2 \cdot 10^{-1}$	−0.21547663	0.89950419
2	1	$3.9 \cdot 10^{-1}$	−0.30981685	0.24936462
3	$5.0 \cdot 10^{-1}$	$2.7 \cdot 10^{-1}$	−0.18342983	0.63371268
4	1	$9.5 \cdot 10^{-2}$	0.24043521	0.54179583
5	1	$8.6 \cdot 10^{-2}$	0.47869789	0.68677241
6	1	$2.2 \cdot 10^{-2}$	0.60989461	0.58165830
7	1	$1.9 \cdot 10^{-3}$	0.57015594	0.56645191
8	1	$3.2 \cdot 10^{-4}$	0.56634502	0.56534838
9	1	$6.1 \cdot 10^{-5}$	0.56580925	0.56573016
10	$9.8 \cdot 10^{-4}$	$6.1 \cdot 10^{-5}$	0.56576079	0.56577847
11	1	$2.0 \cdot 10^{-5}$	0.56566687	0.56571390
12	1	$1.9 \cdot 10^{-6}$	0.56568922	0.56568680
13	$3.1 \cdot 10^{-5}$	$1.9 \cdot 10^{-6}$	0.56568771	0.56568831
14	1	$6.2 \cdot 10^{-7}$	0.56568484	0.56568631
15	1	$5.7 \cdot 10^{-8}$	0.56568554	0.56568547
16	$9.5 \cdot 10^{-7}$	$5.7 \cdot 10^{-8}$	0.56568549	0.56568551
17	1	$1.9 \cdot 10^{-8}$	0.56568541	0.56568545
18	1	$1.8 \cdot 10^{-9}$	0.56568543	0.56568543
19	$2.4 \cdot 10^{-7}$	$1.8 \cdot 10^{-9}$	0.56568543	0.56568542
20	1	$5.4 \cdot 10^{-10}$	0.56568543	0.56568542
21	1	$5.4 \cdot 10^{-11}$	0.56568543	0.56568543

$h = 2^{-4}$

k	β	$\frac{\|\|f(x_k)\|\|}{f(x_0)}$	x	y
1	1	$4.2 \cdot 10^{-1}$	−0.20482004	0.89819413
2	1	$3.7 \cdot 10^{-1}$	−0.27450182	0.26212401
3	$5.0 \cdot 10^{-1}$	$2.5 \cdot 10^{-1}$	−0.15000629	0.62183222
4	1	$9.4 \cdot 10^{-2}$	0.26431759	0.52907491
5	1	$8.7 \cdot 10^{-2}$	0.47734798	0.68860447
6	1	$2.3 \cdot 10^{-2}$	0.61073123	0.58196057
7	$5.0 \cdot 10^{-1}$	$1.3 \cdot 10^{-2}$	0.57319801	0.59149384
8	1	$6.4 \cdot 10^{-3}$	0.57851535	0.57032090
9	$6.3 \cdot 10^{-2}$	$6.0 \cdot 10^{-3}$	0.57341708	0.57443263
10	1	$1.9 \cdot 10^{-3}$	0.56369716	0.56828415
11	1	$1.2 \cdot 10^{-4}$	0.56592259	0.56577111
12	$9.8 \cdot 10^{-4}$	$1.2 \cdot 10^{-4}$	0.56584057	0.56585285
13	1	$3.6 \cdot 10^{-5}$	0.56564563	0.56573504
14	1	$1.8 \cdot 10^{-6}$	0.56568911	0.56568676
15	$1.5 \cdot 10^{-5}$	$1.8 \cdot 10^{-6}$	0.56568783	0.56568804
16	1	$5.6 \cdot 10^{-7}$	0.56568480	0.56568620
17	1	$2.8 \cdot 10^{-8}$	0.56568548	0.56568545
18	$2.4 \cdot 10^{-7}$	$2.8 \cdot 10^{-8}$	0.56568546	0.56568547
19	1	$9.0 \cdot 10^{-9}$	0.56568542	0.56568544
20	1	$4.3 \cdot 10^{-10}$	0.56568543	0.56568543
21	$3.7 \cdot 10^{-9}$	$4.3 \cdot 10^{-10}$	0.56568543	0.56568542
22	1	$1.6 \cdot 10^{-10}$	0.56568543	0.56568542
23	1	$6.7 \cdot 10^{-12}$	0.56568543	0.56568543

$h = 2^{-5}$

k	β	$\frac{\|\|f(x_k)\|\|}{f(x_0)}$	x	y
1	1	$4.1 \cdot 10^{-1}$	−0.19933564	0.89751991
2	1	$3.5 \cdot 10^{-1}$	−0.25674166	0.26854079
3	$5.0 \cdot 10^{-1}$	$2.4 \cdot 10^{-1}$	−0.13354110	0.61638957
4	1	$9.4 \cdot 10^{-2}$	0.27581888	0.52361758
5	1	$8.9 \cdot 10^{-2}$	0.47456209	0.69135130
6	1	$2.4 \cdot 10^{-2}$	0.61277894	0.58270041
7	$5.0 \cdot 10^{-1}$	$1.9 \cdot 10^{-2}$	0.56615606	0.60003123
8	1	$6.8 \cdot 10^{-3}$	0.57931550	0.57061000
9	$6.3 \cdot 10^{-2}$	$6.4 \cdot 10^{-3}$	0.57309154	0.57578562
10	1	$2.3 \cdot 10^{-3}$	0.56296496	0.56879608
11	1	$7.8 \cdot 10^{-5}$	0.56584339	0.56574250
12	$9.8 \cdot 10^{-4}$	$7.8 \cdot 10^{-5}$	0.56574378	0.56584191
13	1	$4.0 \cdot 10^{-5}$	0.56563543	0.56573874
14	1	$6.1 \cdot 10^{-7}$	0.56568666	0.56568587
15	$7.6 \cdot 10^{-6}$	$6.1 \cdot 10^{-7}$	0.56568588	0.56568665
16	1	$3.2 \cdot 10^{-7}$	0.56568503	0.56568584
17	1	$4.7 \cdot 10^{-9}$	0.56568543	0.56568543
18	$1.2 \cdot 10^{-7}$	$4.7 \cdot 10^{-9}$	0.56568543	0.56568543
19	1	$2.0 \cdot 10^{-9}$	0.56568542	0.56568543
20	1	$3.7 \cdot 10^{-11}$	0.56568543	0.56568543

$h = 2^{-6}$

k	β	$\frac{\|\|f(x_k)\|\|}{f(x_0)}$	x	y
1	1	$4.1 \cdot 10^{-1}$	−0.19655306	0.89717784
2	1	$3.5 \cdot 10^{-1}$	−0.24783831	0.27175759
3	$5.0 \cdot 10^{-1}$	$2.4 \cdot 10^{-1}$	−0.12537658	0.61379821
4	1	$9.3 \cdot 10^{-2}$	0.28144493	0.52112169
5	1	$9.1 \cdot 10^{-2}$	0.47254462	0.69326079
6	1	$2.5 \cdot 10^{-2}$	0.61435845	0.58327109
7	$5.0 \cdot 10^{-1}$	$2.2 \cdot 10^{-2}$	0.56312569	0.60421397
8	1	$7.1 \cdot 10^{-3}$	0.57991697	0.57082731
9	$6.3 \cdot 10^{-2}$	$6.7 \cdot 10^{-3}$	0.57311868	0.57653079
10	1	$2.5 \cdot 10^{-3}$	0.56258367	0.56905894
11	1	$5.8 \cdot 10^{-5}$	0.56580244	0.56572770
12	$4.9 \cdot 10^{-4}$	$5.7 \cdot 10^{-5}$	0.56574814	0.56578193
13	1	$2.3 \cdot 10^{-5}$	0.56565624	0.56571585
14	1	$2.3 \cdot 10^{-7}$	0.56568589	0.56568559
15	$1.9 \cdot 10^{-6}$	$2.3 \cdot 10^{-7}$	0.56568567	0.56568580
16	1	$9.1 \cdot 10^{-8}$	0.56568531	0.56568554
17	1	$8.8 \cdot 10^{-10}$	0.56568543	0.56568543
18	$9.5 \cdot 10^{-7}$	$8.8 \cdot 10^{-10}$	0.56568543	0.56568542
19	1	$4.6 \cdot 10^{-10}$	0.56568543	0.56568542
20	1	$3.4 \cdot 10^{-12}$	0.56568543	0.56568543

$h = 2^{-7}$

k	β	$\frac{\|\|f(x_k)\|\|}{f(x_0)}$	x	y
1	1	$4.1 \cdot 10^{-1}$	−0.19515151	0.89700554
2	1	$3.5 \cdot 10^{-1}$	−0.24338113	0.27336798
3	$5.0 \cdot 10^{-1}$	$2.4 \cdot 10^{-1}$	−0.12131217	0.61253573
4	1	$9.3 \cdot 10^{-2}$	0.28422495	0.51993276
5	1	$9.2 \cdot 10^{-2}$	0.47136608	0.69436077
6	1	$2.5 \cdot 10^{-2}$	0.61531660	0.58361727
7	$5.0 \cdot 10^{-1}$	$2.4 \cdot 10^{-2}$	0.56176310	0.60627512
8	1	$7.2 \cdot 10^{-3}$	0.58027164	0.57095545
9	$6.3 \cdot 10^{-2}$	$6.8 \cdot 10^{-3}$	0.57318318	0.57692168
10	1	$2.7 \cdot 10^{-3}$	0.56238767	0.56919204
11	1	$4.7 \cdot 10^{-5}$	0.56578143	0.56572011
12	$4.9 \cdot 10^{-4}$	$4.7 \cdot 10^{-5}$	0.56572486	0.56577663
13	1	$2.3 \cdot 10^{-5}$	0.56565554	0.56571583
14	1	$9.4 \cdot 10^{-8}$	0.56568562	0.56568549
15	$9.5 \cdot 10^{-7}$	$9.4 \cdot 10^{-8}$	0.56568550	0.56568560
16	1	$4.6 \cdot 10^{-8}$	0.56568537	0.56568548
17	1	$1.8 \cdot 10^{-10}$	0.56568543	0.56568543
18	$1.9 \cdot 10^{-6}$	$1.8 \cdot 10^{-10}$	0.56568543	0.56568543
19	1	$7.0 \cdot 10^{-11}$	0.56568542	0.56568543

$h = 2^{-8}$

k	β	$\frac{\|\|f(x_k)\|\|}{f(x_0)}$	x	y
1	1	$4.1 \cdot 10^{-1}$	−0.19444815	0.89691907
2	1	$3.4 \cdot 10^{-1}$	−0.24115120	0.27417366
3	$5.0 \cdot 10^{-1}$	$2.3 \cdot 10^{-1}$	−0.11928454	0.61191288
4	1	$9.3 \cdot 10^{-2}$	0.28560647	0.51935316
5	1	$9.2 \cdot 10^{-2}$	0.47073257	0.69494855
6	1	$2.6 \cdot 10^{-2}$	0.61584243	0.58380726
7	$5.0 \cdot 10^{-1}$	$2.4 \cdot 10^{-2}$	0.56112472	0.60729671
8	1	$7.3 \cdot 10^{-3}$	0.58046297	0.57102458
9	$6.3 \cdot 10^{-2}$	$6.9 \cdot 10^{-3}$	0.57322866	0.57712186
10	1	$2.7 \cdot 10^{-3}$	0.56228812	0.56925901
11	1	$4.2 \cdot 10^{-5}$	0.56577089	0.56571630
12	$2.4 \cdot 10^{-4}$	$4.2 \cdot 10^{-5}$	0.56574203	0.56574514
13	1	$1.4 \cdot 10^{-5}$	0.56566800	0.56570308
14	1	$4.2 \cdot 10^{-8}$	0.56568551	0.56568546
15	$2.4 \cdot 10^{-7}$	$4.2 \cdot 10^{-8}$	0.56568548	0.56568548
16	1	$1.4 \cdot 10^{-8}$	0.56568541	0.56568544
17	1	$4.1 \cdot 10^{-11}$	0.56568543	0.56568543

$h = 2^{-9}$

k	β	$\frac{\|\|f(x_k)\|\|}{f(x_0)}$	x	y
1	1	$4.1 \cdot 10^{-1}$	−0.19409581	0.89687576
2	1	$3.4 \cdot 10^{-1}$	−0.24003591	0.27457662
3	$5.0 \cdot 10^{-1}$	$2.3 \cdot 10^{-1}$	−0.11827188	0.61160356
4	1	$9.3 \cdot 10^{-2}$	0.28629508	0.51906709
5	1	$9.2 \cdot 10^{-2}$	0.47040453	0.69525208
6	1	$2.6 \cdot 10^{-2}$	0.61611769	0.58390671
7	$5.0 \cdot 10^{-1}$	$2.5 \cdot 10^{-2}$	0.56081693	0.60780506
8	1	$7.4 \cdot 10^{-3}$	0.58056220	0.57106043
9	$6.3 \cdot 10^{-2}$	$7.0 \cdot 10^{-3}$	0.57325478	0.57722316
10	1	$2.7 \cdot 10^{-3}$	0.56223792	0.56929260
11	1	$3.9 \cdot 10^{-5}$	0.56576566	0.56571441
12	$2.4 \cdot 10^{-4}$	$3.9 \cdot 10^{-5}$	0.56573651	0.56574353
13	1	$1.3 \cdot 10^{-5}$	0.56566815	0.56570282
14	1	$2.0 \cdot 10^{-8}$	0.56568547	0.56568544
15	$1.2 \cdot 10^{-7}$	$2.0 \cdot 10^{-8}$	0.56568545	0.56568545
16	1	$6.4 \cdot 10^{-9}$	0.56568542	0.56568543
17	1	$9.9 \cdot 10^{-12}$	0.56568543	0.56568543

3 Выводы

● Метод Ньютона не находит все точки, или всё множество точек, где заданная СНУ равна нулю, он находит лишь одну точку, где невязка минимальна. Это проявляется в варианте с одной окржуностью (там имеется множество решений, а находится лишь точка), в варианте с двумя пересекающимися окружностями (там имеется 2 решения, а находится ближнее к начальной точке), в варианте с тремя прямыми (там нет точки, где невязка равна нулю, но решение сходится к точке с минимальной невязкой).
● 2 вариант приведения к диагональному виду делает итоговую СНУ выглядящей как ломаную, поэтому метод на ней может сходиться не плавно.
● 3 вариант приведения к диагональному виду так же делает итоговую СНУ выглядещей как ломаную, но при этом она более плавная.
● 4 вариант приведения к диагональному виду делает итоговую СНУ очень плавной, и видно, что на всех тестах невязка как будто непрерывна. И даже на тесте с тремя прямыми метод Ньютона сошелся за один шаг. Но на этот метод очень сильно влияет погрешность при вычислении производной, что опять же видно из теста с тремя прямыми (1 шаг против 7).
● Взвешинвание одного из уравнений смещает решение к точкам этого уравнения.
● Вычисление матрицы Якоби при помощи численного дифференцирования работает неплохо по сравнению с аналитическим вычислением. Оно влияет только на скорость сходимости, но на данных тестах незначительно.
● Чем меньше шаг при взятии производной, тем лучше и быстрее сходится метод.

4 Код программы

FILE **logic.h**

```h
#pragma once

#include <iostream>
#include <functional>
#include <vector>
#include "../1/matrix.h"

#define myassert(A) {if (!(A)) { std::cerr << "ERROR: " << #A << endl; throw std::exception(); }}

using namespace std;

typedef vector<double> xn_t; // x in R^n вектор
typedef vector<xn_t> matrix_t;
typedef function<double(const xn_t&)> fn_f; // f : R^n -> R одна функция нелинейной системы
typedef vector<fn_f> fnmv_f; // Vector<fn_f>
typedef function<xn_t(const xn_t&)> fnm_f; // f : R^n -> R^m нелинейная система
typedef function<matrix_t(const xn_t&)> jnm_f; // j : R^n -> R^(m*n) матрица якоби
typedef pair<matrix_t, xn_t> sle_t; // Тип СЛАУ
typedef function<sle_t(const xn_t&)> sle_f; // Функция, которая возвращает СЛАУ
typedef function<sle_f(const sle_f&)> sqr_f; // Функция, преобразующая СЛАУ к квадратному виду

Matrix to(const matrix_t& a);
Matrix to(const xn_t& a);
matrix_t to_matrix(const Matrix& a);
xn_t to_vec(const Matrix& a);

double length(const xn_t& x);
xn_t operator+(const xn_t& a, const xn_t& b);
xn_t operator-(const xn_t& a, const xn_t& b);
xn_t operator*(const xn_t& a, double b);
xn_t operator*(double b, const xn_t& a);
xn_t operator*(const xn_t& a, const xn_t& b);
ostream& operator<<(ostream& out, const xn_t& v);

void solve_gauss(const matrix_t& a, const xn_t& b, xn_t& dx);
void mul_t(const matrix_t& a, const matrix_t& b, matrix_t& result);
void mul_t(const matrix_t& a, const xn_t& b, xn_t& result);

extern double partial_derivative_step;
double calc_partial_derivative_numeric(const fn_f& f, const xn_t& x, int i);
matrix_t calc_jacobi_matrix_numeric(const fnmv_f& f, const xn_t& x);
jnm_f calc_jacobi_matrix_numeric_funcion(const fnmv_f& f);

xn_t calc_vector_function(const fnmv_f& f, const xn_t& x);
sle_f get_sle_function(const jnm_f& j, const fnmv_f& f);
sle_f square_cast_none(const sle_f& s);
sle_f square_cast_1(const sle_f& s);
sle_f square_cast_2(const sle_f& s);
sle_f square_cast_3(const sle_f& s);
sle_f square_cast_4(const sle_f& s);

fnm_f get_f(const sle_f& s);
//function<fnmv_f(const fnmv_f&)> function_mul(const xn_t& m);
sqr_f square_cast_mul(const xn_t& x);
sqr_f composition(const sqr_f& f, const sqr_f& g);

enum exit_type_t
{
    EXIT_ITER,
    EXIT_RESIDUAL,
    EXIT_BETA,
    EXIT_STEP,
    EXIT_ERROR
};

struct solved_t
{
    int iterations;
    double residual;
    xn_t point;
    vector<xn_t> x_process;
    vector<double> beta_process;
    vector<double> residual_process;
    exit_type_t exit_type;
};

solved_t solve(
    const sle_f& s,
    const xn_t& x_0,
    int maxiter,
    double eps,
    bool is_log
);
```

FILE **logic.cpp**

```cpp
#include <iostream>
#include <iomanip>
#include <algorithm>

#include "../1/matrix.h"
#include "logic.h"

//-----------------------------------------------------------------------------
Matrix to(const matrix_t& a) {
    Matrix result(a[0].size(), a.size());
    for (int i = 0; i < a.size(); ++i) {
        for (int j = 0; j < a[i].size(); ++j)
            result(i, j) = a[i][j];
    }
    return result;
}

//-----------------------------------------------------------------------------
Matrix to(const xn_t& a) {
    Matrix result(1, a.size());
    for (int i = 0; i < a.size(); ++i) {
        result(i, 0) = a[i];
    }
    return result;
}

//-----------------------------------------------------------------------------
matrix_t to_matrix(const Matrix& a) {
    matrix_t result(a.height(), vector<double>(a.width()));
    for (int i = 0; i < a.height(); ++i) {
        for (int j = 0; j < a.width(); ++j) {
            result[i][j] = a(i, j);
        }
    }
    return result;
}

//-----------------------------------------------------------------------------
xn_t to_vec(const Matrix& a) {
    xn_t result(a.height());
    for (int i = 0; i < a.height(); ++i)
        result[i] = a(i, 0);
    return result;
}

//-----------------------------------------------------------------------------
double length(const xn_t& x) {
    double sum = 0;
    for (const auto& i : x)
        sum += i*i;
    return sqrt(sum);
}

//-----------------------------------------------------------------------------
void solve_gauss(const matrix_t& a, const xn_t& b, xn_t& dx) {
    Matrix dx_m;
    solveSLAE_byGaussMethod(to(a), to(b), dx_m);
    dx = to_vec(dx_m);
}

//-----------------------------------------------------------------------------
void mul_t(const matrix_t& a, const matrix_t& b, matrix_t& result) {
    Matrix result_m;
    Matrix a_m = to(a);
    transpose(a_m);
    mul(a_m, to(b), result_m);
    result = to_matrix(result_m);
}

//-----------------------------------------------------------------------------
void mul_t(const matrix_t& a, const xn_t& b, xn_t& result) {
    Matrix result_m;
    Matrix a_m = to(a);
```

```
    transpose(a_m);
    mul(a_m, to(b), result_m);
    result = to_vec(result_m);
}

//-----------------------------------------------------------------------------
xn_t operator+(const xn_t& a, const xn_t& b) {
    myassert(a.size() == b.size());
    xn_t result(a.size());
    for (int i = 0; i < a.size(); ++i)
        result[i] = a[i] + b[i];
    return result;
}

//-----------------------------------------------------------------------------
xn_t operator-(const xn_t& a, const xn_t& b) {
    myassert(a.size() == b.size());
    xn_t result(a.size());
    for (int i = 0; i < a.size(); ++i)
        result[i] = a[i] - b[i];
    return result;
}

//-----------------------------------------------------------------------------
xn_t operator*(const xn_t& a, double b) {
    xn_t result(a.size());
    for (int i = 0; i < a.size(); ++i)
        result[i] = a[i] * b;
    return result;
}

//-----------------------------------------------------------------------------
xn_t operator*(double b, const xn_t& a) {
    return operator*(a, b);
}

//-----------------------------------------------------------------------------
xn_t operator*(const xn_t& a, const xn_t& b) {
    myassert(a.size() == b.size());
    xn_t result(a.size());
    for (int i = 0; i < a.size(); ++i)
        result[i] = a[i] * b[i];
    return result;
}

//-----------------------------------------------------------------------------
ostream& operator<<(ostream& out, const xn_t& v) {
    out << "(";
    for (int i = 0; i < v.size()-1; ++i)
        out << v[i] << ", ";
    out << v.back() << ")";
    return out;
}

//-----------------------------------------------------------------------------
double partial_derivative_step = 1e-9;
double calc_partial_derivative_numeric(const fn_f& f, const xn_t& x_in, int i) {
    myassert(i < x_in.size());

    double step = partial_derivative_step;
    if (x_in[i] != 0)
        step = x_in[i]*step;

    auto x = x_in;
    x[i] = x[i] +      0; double x0 = f(x); x[i] = x[i] -      0;
    x[i] = x[i] +   step; double x1 = f(x); x[i] = x[i] -   step;
    x[i] = x[i] -   step; double x2 = f(x); x[i] = x[i] +   step;
    x[i] = x[i] + 2*step; double x3 = f(x); x[i] = x[i] - 2*step;
    x[i] = x[i] - 2*step; double x4 = f(x); x[i] = x[i] + 2*step;

    //double result = (-x3 + 8*x1 - 8*x2 + x4)/(12.0*step);
    double result = (x1 - x0) / step;
    return result;
}

//-----------------------------------------------------------------------------
matrix_t calc_jacobi_matrix_numeric(const fnmv_f& f, const xn_t& x) {
    matrix_t result(f.size(), xn_t(x.size()));
    for (int i = 0; i < f.size(); ++i) {
        for (int j = 0; j < x.size(); ++j) {
            result[i][j] = calc_partial_derivative_numeric(f[i], x, j);
        }
    }
    return result;
}
```

```
//-----------------------------------------------------------------------------
jnm_f calc_jacobi_matrix_numeric_funcion(const fnmv_f& f) {
    return bind(calc_jacobi_matrix_numeric, f, placeholders::_1);
}

//-----------------------------------------------------------------------------
xn_t calc_vector_function(const fnmv_f& f, const xn_t& x) {
    xn_t result;
    for (const auto& i : f)
        result.push_back(i(x));
    return result;
}

//-----------------------------------------------------------------------------
sle_f get_sle_function(const jnm_f& j, const fnmv_f& f) {
    return [j, f] (const xn_t& x) -> sle_t {
        matrix_t a = j(x);
        xn_t b = calc_vector_function(f, x);
        return {a, b};
    };
}

sle_f square_cast_none(const sle_f& s) {
    return [s] (const xn_t& x) -> sle_t {
        auto res = s(x);
        myassert(res.first[0].size() == res.first.size());
        myassert(res.first.size() == res.second.size());
        return res;
    };
}

//-----------------------------------------------------------------------------
sle_f square_cast_1(const sle_f& s) {
    return [s] (const xn_t& x) -> sle_t {
        // Получаем значение текущей СЛАУ
        sle_t res = s(x);
        auto& A = res.first;
        auto& b = res.second;

        myassert(A[0].size() > A.size());

        int count = x.size() - A.size();

        // Находим номера элементов, для которых dF_i(x)/dx_j минимально при всех i
        vector<pair<double, int>> b_sorted;
        for (int j = 0; j < A[0].size(); ++j) {
            double max1 = fabs(A[0][j]);
            for (int i = 1; i < A.size(); ++i)
                max1 = max(max1, fabs(A[i][j]));
            b_sorted.push_back({max1, j});
        }

        sort(b_sorted.begin(), b_sorted.end(), [] (auto a, auto b) -> bool {
            return a.first < b.first;
        });

        vector<int> mins;
        for (int i = 0; i < count; ++i)
            mins.push_back(b_sorted[i].second);
        sort(mins.begin(), mins.end(), less<int>());

        int start = mins[0];

        // Добавляем к вектору нулевые элементы
        for (int i = 0; i < mins.size(); ++i)
            b.push_back(0);

        auto make_vec = [] (int size, int where_one) -> vector<double> {
            vector<double> result(size, 0);
            result[where_one] = 1;
            return result;
        };

        // Добавляем к матрице новые строки
        for (auto& i : mins)
            A.push_back(make_vec(x.size(), i));

        return {A, b};
    };
}

//-----------------------------------------------------------------------------
sle_f square_cast_2(const sle_f& s) {
    return [s] (const xn_t& x) -> sle_t {
        // Получаем значение текущей СЛАУ
        sle_t res = s(x);
        auto& A = res.first;
        auto& b = res.second;

        myassert(A[0].size() < A.size());

        int count = b.size() - x.size();

        // Находим номера элементов, для которых f_i(x) минимальны
        vector<pair<double, int>> b_sorted;
        for (int i = 0; i < b.size(); ++i)
            b_sorted.push_back({fabs(b[i]), i});

        sort(b_sorted.begin(), b_sorted.end(), [] (auto a, auto b) -> bool {
            return a.first < b.first;
        });

        vector<int> mins;
        for (int i = 0; i < count; ++i)
            mins.push_back(b_sorted[i].second);
        sort(mins.begin(), mins.end(), less<int>());

        int start = mins[0];

        // Удаляем лишние строки
        for (int i = mins.size()-1; i >= 0; --i) {
            A.erase(A.begin() + mins[i]);
            b.erase(b.begin() + mins[i]);
        }

        return {A, b};
    };
}

//-----------------------------------------------------------------------------
sle_f square_cast_3(const sle_f& s) {
    return [s] (const xn_t& x) -> sle_t {
        // Получаем значение текущей СЛАУ
        sle_t res = s(x);
        auto& A = res.first;
        auto& b = res.second;

        myassert(A[0].size() < A.size());

        int count = b.size() - x.size() + 1;

        // Находим номера элементов, для которых f_i(x) минимальны
        vector<pair<double, int>> b_sorted;
        for (int i = 0; i < b.size(); ++i)
            b_sorted.push_back({fabs(b[i]), i});

        sort(b_sorted.begin(), b_sorted.end(), [] (auto a, auto b) -> bool {
            return a.first < b.first;
        });

        vector<int> mins;
        for (int i = 0; i < count; ++i)
            mins.push_back(b_sorted[i].second);
        sort(mins.begin(), mins.end(), less<int>());

        int start = mins[0];

        // Строим новую матрицу Якоби
        for (int j = 0; j < A[start].size(); ++j)
            A[start][j] = 2*A[start][j] * b[start];

        for (int i = 1; i < mins.size(); ++i) {
            for (int j = 0; j < A[mins[i]].size(); ++j) {
                A[start][j] += 2 * A[mins[i]][j] * b[mins[i]];
            }
        }

        // Строим новый вектор правой части
        b[start] = b[start] * b[start];
        for (int i = 1; i < mins.size(); ++i)
            b[start] += b[mins[i]] * b[mins[i]];

        // Удаляем лишние строки
        for (int i = mins.size()-1; i > 0; --i) {
            A.erase(A.begin() + mins[i]);
            b.erase(b.begin() + mins[i]);
        }

        return {A, b};
    };
}

//-----------------------------------------------------------------------------
sle_f square_cast_4(const sle_f& s) {
    return [s] (const xn_t& x) -> sle_t {
        // Получаем значение текущей СЛАУ
        sle_t res = s(x);
```

```
        auto& A = res.first;
        auto& b = res.second;

        myassert(A[0].size() < A.size());

        matrix_t AR;
        xn_t bR;
        mul_t(A, A, AR);
        mul_t(A, b, bR);

        return {AR, bR};
    };
}

//-----------------------------------------------------------------------------
fnmv_f get_f(const sle_f& s) {
    return [s](const xn_t& x) -> xn_t {
        return s(x).second;
    };
}

//-----------------------------------------------------------------------------
sqr_f square_cast_mul(const xn_t& m) {
    return [m](const sle_f& s) -> sle_f {
        return [m, s] (const xn_t& x) -> sle_t {
            auto res = s(x);
            auto& A = res.first;
            auto& b = res.second;

            myassert(m.size() == A.size());
            myassert(m.size() == b.size());

            for (int i = 0; i < A.size(); i++) {
                for (int j = 0; j < A[i].size(); j++) {
                    A[i][j] *= m[i];
                }
            }
            b = b * m;

            return {A, b};
        };
    };
}

//-----------------------------------------------------------------------------
sqr_f composition(const sqr_f& f, const sqr_f& g) {
    return [f, g](const sle_f& t) -> sle_f {
        return f(g(t));
    };
}

//-----------------------------------------------------------------------------
solved_t solve(const sle_f& s, const xn_t& x_0, int maxiter, double eps, bool is_log) {
    vector<xn_t> x_process;
    vector<double> beta_process;
    vector<double> residual_process;
    x_process.push_back(x_0);
    beta_process.push_back(0);
    residual_process.push_back(0);
    xn_t x_k = x_0, x_kv, dx;
    exit_type_t exit_type;

    auto f = get_f(s);

    double f_0 = length(f(x_k));
    int it = 0;
    while (true) {
        if (it > maxiter) {
            exit_type = EXIT_ITER;
            break;
        }

        if (length(f(x_k)) / f_0 < eps) {
            exit_type = EXIT_RESIDUAL;
            break;
        }

        auto sle = s(x_k);
        auto& A = sle.first;
        auto& b = sle.second;
        for (auto& i : b) i = -i; // b = -b

        #ifdef _DEBUG
        int An = A.size();
        myassert(An == b.size());
        for (auto& i : A)
            myassert(An == i.size());
        #endif

        solve_gauss(A, b, dx);

        if (dx.size() == 0) {
            exit_type = EXIT_ERROR;
            break;
        }

        double beta = 1;
        x_kv = x_k + beta*dx;
        while (length(f(x_kv)) > length(f(x_k))) {
            beta /= 2.0;
            x_kv = x_k + beta * dx;
        }

        x_k = x_kv;
        it++;

        x_process.push_back(x_k);
        beta_process.push_back(beta);
        residual_process.push_back(length(f(x_k)) / f_0);

        if (is_log) {
            cout << "Iteration: " << setw(5) << it;
            cout << scientific << setprecision(2);
            cout << ", B: " << setw(8) << beta;
            cout << ", Residual: " << setw(8) << length(f(x_k)) << endl;
        }

        if (length(x_k - x_process[x_process.size() - 2]) < eps) {
            exit_type = EXIT_STEP;
            break;
        }

        if (fabs(beta) < eps) {
            exit_type = EXIT_BETA;
            break;
        }
    }

    return {it, length(f(x_k))/f_0, x_k, x_process, beta_process, residual_process, exit_type
};
}
```

FILE objects.h

```
#pragma once

#include "logic.h"

//-----------------------------------------------------------------------------
struct point
{
    point(double x, double y) : x(x), y(y) {}
    point(const xn_t& x) : x(x[0]), y(x[1]) {
        myassert(x.size() == 2);
    }
    double x, y;
};

struct circle {
    circle(double x, double y, double r) : c(x, y), r(r) {}
    circle(point c, double r) : c(c), r(r) {}
    circle(const xn_t& x) : c(x[0], x[1]), r(x[2]) {
        myassert(x.size() == 3);
    }
    point c; double r;
};

struct line { point a, b; };

//-----------------------------------------------------------------------------
pair<sle_f, fnmv_f> one_circle(circle a);
pair<sle_f, fnmv_f> two_circles(circle a, circle b);
pair<sle_f, fnmv_f> two_circles_and_line(circle a, circle b, line l);
pair<sle_f, fnmv_f> three_lines(line a, line b, line c);

pair<sle_f, fnmv_f> sin_and_line(point b); // a is {0, 0}

pair<sle_f, fnmv_f> three_circles(
    circle a, circle b, circle c,
    bool a_in, bool b_in, bool c_in
);

//-----------------------------------------------------------------------------
inline double sqr(double a) {return a*a;}
inline double sign(double a) { if (a == 0) return 0; else return (a<0)?-1:1; }

double dist(const point& a, const point& b);

double line_f(const line& l, const point& x);
double line_d_x(const line& l, const point& x);
double line_d_y(const line& l, const point& x);

double circle_f(const circle& a, const point& x);
double circle_d_x(const circle& a, const point& x);
double circle_d_y(const circle& a, const point& x);

double circles_f(const circle& a, const circle& x, bool in);
double circles_d_x(const circle& a, const circle& x, bool in);
double circles_d_y(const circle& a, const circle& x, bool in);
double circles_d_r(const circle& a, const circle& x, bool in);
```

FILE objects.cpp

```
#include "objects.h"

//-----------------------------------------------------------------------------
//-----------------------------------------------------------------------------
//-----------------------------------------------------------------------------

//-----------------------------------------------------------------------------
pair<sle_f, fnmv_f> one_circle(circle a) {
    //-------------------------------------------------------------------------
    fn_f f1 = [a] (const xn_t& x) -> double { return circle_f(a, x); };

    fnmv_f f = {f1};

    //-------------------------------------------------------------------------
    jnm_f j = [a] (const xn_t& x) -> matrix_t {
        matrix_t result(1, xn_t(2));
        result[0][0] = circle_d_x(a, x);
        result[0][1] = circle_d_y(a, x);

        return result;
    };

    return {get_sle_function(j, f), f};
}

//-----------------------------------------------------------------------------
pair<sle_f, fnmv_f> two_circles(circle a, circle b) {
    //-------------------------------------------------------------------------
    fn_f f1 = [a] (const xn_t& x) -> double { return circle_f(a, x); };
    fn_f f2 = [b] (const xn_t& x) -> double { return circle_f(b, x); };

    fnmv_f f = {f1, f2};

    //-------------------------------------------------------------------------
    jnm_f j = [a, b] (const xn_t& x) -> matrix_t {
        matrix_t result(2, xn_t(2));
        result[0][0] = circle_d_x(a, x);
        result[0][1] = circle_d_y(a, x);

        result[1][0] = circle_d_x(b, x);
        result[1][1] = circle_d_y(b, x);

        return result;
    };

    return {get_sle_function(j, f), f};
}

//-----------------------------------------------------------------------------
pair<sle_f, fnmv_f> two_circles_and_line(circle a, circle b, line l) {
    //-------------------------------------------------------------------------
    fn_f f1 = [a] (const xn_t& x) -> double { return circle_f(a, x); };
    fn_f f2 = [b] (const xn_t& x) -> double { return circle_f(b, x); };
    fn_f f3 = [l] (const xn_t& x) -> double { return line_f(l, x); };

    fnmv_f f = {f1, f2, f3};

    //-------------------------------------------------------------------------
    jnm_f j = [a, b, l] (const xn_t& x) -> matrix_t {
        matrix_t result(3, xn_t(2));
        result[0][0] = circle_d_x(a, x);
        result[0][1] = circle_d_y(a, x);

        result[1][0] = circle_d_x(b, x);
        result[1][1] = circle_d_y(b, x);

        result[2][0] = line_d_x(l, x);
        result[2][1] = line_d_y(l, x);

        return result;
    };

    return {get_sle_function(j, f), f};
}

//-----------------------------------------------------------------------------
pair<sle_f, fnmv_f> three_lines(line a, line b, line c) {
    //-------------------------------------------------------------------------
    fn_f f1 = [a] (const xn_t& x) -> double { return line_f(a, x); };
    fn_f f2 = [b] (const xn_t& x) -> double { return line_f(b, x); };
    fn_f f3 = [c] (const xn_t& x) -> double { return line_f(c, x); };

    fnmv_f f = {f1, f2, f3};

    //-------------------------------------------------------------------------
    jnm_f j = [a, b, c] (const xn_t& x) -> matrix_t {
        matrix_t result(3, xn_t(2));
        result[0][0] = line_d_x(a, x);
        result[0][1] = line_d_y(a, x);

        result[1][0] = line_d_x(b, x);
        result[1][1] = line_d_y(b, x);

        result[2][0] = line_d_x(c, x);
        result[2][1] = line_d_y(c, x);

        return result;
    };

    return {get_sle_function(j, f), f};
}

//-----------------------------------------------------------------------------
pair<sle_f, fnmv_f> sin_and_line(point b) {
    //-------------------------------------------------------------------------
    line l = {point(0, 0), b};
    fn_f f1 = [l] (const xn_t& x) -> double {return line_f(l, x); };
    fn_f f2 = [] (const xn_t& x) -> double {
        myassert(x.size() == 2);
        return x[1]-sin(x[0]);
    };

    fnmv_f f = {f1, f2};

    //-------------------------------------------------------------------------
    jnm_f j = [l] (const xn_t& x) -> matrix_t {
        myassert(x.size() == 2);
        matrix_t result(2, xn_t(2));
        result[0][0] = line_d_x(l, x);
        result[0][1] = line_d_y(l, x);

        result[1][0] = -cos(x[0]);
        result[1][1] = 1;

        return result;
    };

    return {get_sle_function(j, f), f};
}

//-----------------------------------------------------------------------------
pair<sle_f, fnmv_f> three_circles(circle a, circle b, circle c, bool a_in, bool b_in, bool
↪  c_in) {
    //-------------------------------------------------------------------------
    fn_f f1 = [a, a_in] (const xn_t& x) -> double { return circles_f(a, x, a_in); };
    fn_f f2 = [b, b_in] (const xn_t& x) -> double { return circles_f(b, x, b_in); };
    fn_f f3 = [c, c_in] (const xn_t& x) -> double { return circles_f(c, x, c_in); };

    fnmv_f f = {f1, f2, f3};

    //-------------------------------------------------------------------------
    jnm_f j = [a, b, c, a_in, b_in, c_in] (const xn_t& x) -> matrix_t {
        matrix_t result(3, xn_t(3));
        result[0][0] = circles_d_x(a, x, a_in);
        result[0][1] = circles_d_y(a, x, a_in);
        result[0][2] = circles_d_r(a, x, a_in);

        result[1][0] = circles_d_x(b, x, b_in);
```

```
        result[1][1] = circles_d_y(b, x, b_in);
        result[1][2] = circles_d_r(b, x, b_in);

        result[2][0] = circles_d_x(c, x, c_in);
        result[2][1] = circles_d_y(c, x, c_in);
        result[2][2] = circles_d_r(c, x, c_in);

        return result;
    };

    return {get_sle_function(j, f), f};
}

//-----------------------------------------------------------------------------
// sle_f three_circles_2(circle a, circle b, circle c, bool a_in, bool b_in, bool c_in) {
// }

//-----------------------------------------------------------------------------
//-----------------------------------------------------------------------------
//-----------------------------------------------------------------------------

//-----------------------------------------------------------------------------
double dist(const point& a, const point& b) {
    return sqrt(sqr(a.x-b.x) + sqr(a.y-b.y));
}

//-----------------------------------------------------------------------------
double line_f(const line& l, const point& x) {
    const double relative = 1e-10;
    double length = dist(l.a, l.b);

    bool first_zero = fabs(l.b.x-l.a.x)/length < relative;
    bool second_zero = fabs(l.b.y-l.a.y)/length < relative;

    myassert(!(first_zero && second_zero));

    if (first_zero)
        return x.x-l.a.x;
    else
    if (second_zero)
        return x.y-l.a.y;
    else
        return (x.x-l.a.x)/(l.b.x-l.a.x) - (x.y-l.a.y)/(l.b.y-l.a.y);
}

//-----------------------------------------------------------------------------
double line_d_x(const line& l, const point& x) {
    const double relative = 1e-10;
    double length = dist(l.a, l.b);

    bool first_zero = fabs(l.b.x-l.a.x)/length < relative;
    bool second_zero = fabs(l.b.y-l.a.y)/length < relative;

    myassert(!(first_zero && second_zero));

    if (first_zero)
        return 1;
    else
    if (second_zero)
        return 0;
    else
        return 1.0/(l.b.x-l.a.x);
}

//-----------------------------------------------------------------------------
double line_d_y(const line& l, const point& x) {
    const double relative = 1e-10;
    double length = dist(l.a, l.b);

    bool first_zero = fabs(l.b.x-l.a.x)/length < relative;
    bool second_zero = fabs(l.b.y-l.a.y)/length < relative;

    myassert(!(first_zero && second_zero));

    if (first_zero)
        return 0;
    else
    if (second_zero)
        return 1;
    else
        return -1.0/(l.b.y-l.a.y);
}

//-----------------------------------------------------------------------------
double circle_f(const circle& a, const point& x) {
    return sqr(a.c.x-x.x) + sqr(a.c.y-x.y) - sqr(a.r);
}

//-----------------------------------------------------------------------------
double circle_d_x(const circle& a, const point& x) {
    return -2*(a.c.x-x.x);
}

//-----------------------------------------------------------------------------
double circle_d_y(const circle& a, const point& x) {
    return -2*(a.c.y-x.y);
}

//-----------------------------------------------------------------------------
double circles_f(const circle& a, const circle& x, bool in) {
    double result = sqr(a.c.x-x.c.x) + sqr(a.c.y-x.c.y);
    if (in)
        return result - sqr(a.r-fabs(x.r));
    else
        return result - sqr(a.r+fabs(x.r));
}

//-----------------------------------------------------------------------------
double circles_d_x(const circle& a, const circle& x, bool in) {
    return -2*(a.c.x-x.c.x);
}

//-----------------------------------------------------------------------------
double circles_d_y(const circle& a, const circle& x, bool in) {
    return -2*(a.c.y-x.c.y);
}

//-----------------------------------------------------------------------------
double circles_d_r(const circle& a, const circle& x, bool in) {
    if (in)
        return -2*(x.r-a.r*sign(x.r));
    else
        return -2*(x.r+a.r*sign(x.r));
}
```

FILE find_borders.h

```
#pragma once

#include <vector>
#include "vector2.h"
#include "logic.h"
#include "objects.h"

class FindBorders
{
public:
    FindBorders(int maxSize, int border, bool isRevert);
    void process(vec2 point);
    void process(circle c);
    void process(const std::vector<vec2>& points);
    void finish(void);
    vec2 toImg(vec2 point) const;
    double toImg(double a) const;
    vec2 fromImg(vec2 point) const;
    vec2 getCalculatedSize(void) const;
    bool isInside(vec2 pos) const;
    vec2 min, max;
private:
    int m_maxSize, m_border;
    double m_scale;
    vec2 m_offset;
    vec2 m_size;
    bool m_isRevert;
};
```

FILE find_borders.cpp

```
#include "find_borders.h"

//-----------------------------------------------------------------------------
FindBorders::FindBorders(int maxSize, int border, bool isRevert) : m_maxSize(maxSize),
↪  m_border(border), m_isRevert(isRevert) {
    double inf = std::numeric_limits<double>::infinity();
    min = vec2(inf, inf);
    max = vec2(-inf, -inf);
}

//-----------------------------------------------------------------------------
void FindBorders::process(vec2 point) {
    if (point.x > max.x) max.x = point.x;
    if (point.y > max.y) max.y = point.y;
    if (point.x < min.x) min.x = point.x;
    if (point.y < min.y) min.y = point.y;
}

//-----------------------------------------------------------------------------
void FindBorders::process(circle cr) {
    vec2 c(cr.c.x, cr.c.y);
    vec2 sz(cr.r, cr.r);
    process(c + sz);
    process(c - sz);
}

//-----------------------------------------------------------------------------
void FindBorders::process(const std::vector<vec2>& points) {
    for (const auto& i : points)
        process(i);
}

//-----------------------------------------------------------------------------
void FindBorders::finish(void) {
    if (max.x - min.x > max.y - min.y) {
        m_size.x = m_maxSize;
        m_size.y = (max.y - min.y)/(max.x - min.x) * m_size.x;

        m_scale = m_maxSize/(max.x - min.x);
    } else {
        m_size.y = m_maxSize;
        m_size.x = (max.x - min.x)/(max.y - min.y) * m_size.y;

        m_scale = m_maxSize/(max.y - min.y);
    }
    m_offset = -min;
    m_size += vec2(m_border) * 2;
}

//-----------------------------------------------------------------------------
vec2 FindBorders::toImg(vec2 point) const {
    vec2 result = (point + m_offset) * m_scale + vec2(m_border, m_border);
    if (m_isRevert)
        result.y = m_size.y - result.y - 1;
    return result;
}

//-----------------------------------------------------------------------------
double FindBorders::toImg(double a) const {
    return m_scale * a;
}

//-----------------------------------------------------------------------------
vec2 FindBorders::fromImg(vec2 point) const {
    if (m_isRevert)
        point.y = m_size.y - point.y - 1;
    return (point - vec2(m_border, m_border)) / m_scale - m_offset;
}

//-----------------------------------------------------------------------------
vec2 FindBorders::getCalculatedSize(void) const {
    return m_size;
}

//-----------------------------------------------------------------------------
bool FindBorders::isInside(vec2 pos) const {
    return (pos.x > min.x) && (pos.y > min.y) && (pos.x < max.x) && (pos.y < max.y);
}
```

FILE vector2.h

```
#pragma once

//-----------------------------------------------------------------------------
class vec2
{
public:
    double x, y;

    vec2();
    explicit vec2(double a);
    vec2(double x, double y);

    vec2& negate(void);
    vec2& normalize(void);

    double length(void) const;
    double lengthSqr(void) const;

    //-------------------------------------------------------------------------
    vec2& operator+=(const vec2& a);
    vec2& operator-=(const vec2& a);
    vec2& operator*=(double a);
    vec2& operator/=(double a);
};

//-----------------------------------------------------------------------------
vec2 operator-(const vec2& a);
vec2 operator+(const vec2& a, const vec2& b);
vec2 operator-(const vec2& a, const vec2& b);
vec2 operator*(const vec2& a, double k);
vec2 operator/(const vec2& a, double k);
vec2 operator*(double k, const vec2& a);
vec2 operator/(double k, const vec2& a);

//=============================================================================
//=============================================================================
//=============================================================================

//-----------------------------------------------------------------------------
inline vec2::vec2() : vec2(0) {}

//-----------------------------------------------------------------------------
inline vec2::vec2(double a) : x(a), y(a) {}

//-----------------------------------------------------------------------------
inline vec2::vec2(double x, double y) : x(x), y(y) {}

//-----------------------------------------------------------------------------
inline vec2& vec2::negate(void) {
    x = -x;
    y = -y;
    return *this;
}

//-----------------------------------------------------------------------------
inline vec2& vec2::normalize(void) {
    double length1 = length();
    if (length1 != 0) {
        x /= length1;
        y /= length1;
    }
    return *this;
}

//-----------------------------------------------------------------------------
inline double vec2::length(void) const {
    return std::sqrt(x*x + y*y);
}

//-----------------------------------------------------------------------------
inline double vec2::lengthSqr(void) const {
    return x*x + y*y;
}

//-----------------------------------------------------------------------------
inline vec2& vec2::operator+=(const vec2& a) {
    x += a.x;
    y += a.y;
    return *this;
}

//-----------------------------------------------------------------------------
inline vec2& vec2::operator-=(const vec2& a) {
    x -= a.x;
```

```
    y -= a.y;
    return *this;
}

//-----------------------------------------------------------------------
inline vec2& vec2::operator*=(double a) {
    x *= a;
    y *= a;
    return *this;
}

//-----------------------------------------------------------------------
inline vec2& vec2::operator/=(double a) {
    #ifdef _DEBUG
    if (a == 0)
        throw std::exception("Divide by zero");
    #endif
    x /= a;
    y /= a;
    return *this;
}

//-----------------------------------------------------------------------
inline vec2 operator-(const vec2& a) {
    return vec2(-a.x, -a.y);
}

//-----------------------------------------------------------------------
inline vec2 operator+(const vec2& a, const vec2& b) {
    return vec2(a.x + b.x, a.y + b.y);
}

//-----------------------------------------------------------------------
inline vec2 operator-(const vec2& a, const vec2& b) {
    return vec2(a.x - b.x, a.y - b.y);
}

//-----------------------------------------------------------------------
inline vec2 operator*(const vec2& a, double k) {
    return vec2(a.x * k, a.y * k);
}

//-----------------------------------------------------------------------
inline vec2 operator/(const vec2& a, double k) {
    #ifdef _DEBUG
    if (k == 0)
        throw std::exception("Divide by zero");
    #endif
    return vec2(a.x / k, a.y / k);
}

//-----------------------------------------------------------------------
inline vec2 operator*(double k, const vec2& a) {
    return vec2(a.x * k, a.y * k);
}

//-----------------------------------------------------------------------
inline vec2 operator/(double k, const vec2& a) {
    #ifdef _DEBUG
    if (a.x == 0 || a.y == 0)
        throw std::exception("Divide by zero");
    #endif
    return vec2(k / a.x, k/ a.y);
}
```

FILE draw.cpp

```
#include <sstream>
#include <fstream>
#include <iomanip>
#include <twg/image/image_drawing.h>

#include "logic.h"
#include "objects.h"
#include "find_borders.h"

using namespace twg;

//-----------------------------------------------------------------------
Polygon_d calc_circle(circle a, const FindBorders& brd) {
    vec2 pos = brd.toImg(vec2(a.c.x, a.c.y));
    double r = brd.toImg(a.r);
    return computeEllipse({r, r}).move(Point_d(pos.x, pos.y));
}

//-----------------------------------------------------------------------
void draw_line(ImageDrawing_aa& img, const FindBorders& brd, vec2 a, vec2 b) {
    vec2 start, end;
    start = a + (b-a) * (-5);
    end =   a + (b-a) * (5);
    img.drawLine(brd.toImg(start), brd.toImg(end));
}

//-----------------------------------------------------------------------
class Picture
{
public:
    void init_solve(const pair<sle_f, fnmv_f>& p, const xn_t& x_0, bool is_numeric, const
    ↪  sqr_f& square_cast, int iterations, double residual) {
        m_x_0 = x_0;
        m_iterations = iterations;
        m_residual = residual;
        if (is_numeric) {
            auto j = calc_jacobi_matrix_numeric_functon(p.second);
            auto sle = get_sle_function(j, p.second);
            sle = square_cast(sle);
            m_solved = solve(sle, x_0, iterations, residual, false);
            m_f = get_f(sle);
        } else {
            auto sle = square_cast(p.first);
            m_solved = solve(sle, x_0, iterations, residual, false);
            m_f = get_f(sle);
        }
    }
    virtual void init_brd(FindBorders& brd) const {
        brd.process(vec2(0, 0));
        for (int i = 0; i < m_solved.x_process.size(); ++i)
            brd.process(vec2(m_solved.x_process[i][0], m_solved.x_process[i][1]));
    }
    virtual double getResidual(vec2 a) const {
        return length(m_f({ a.x, a.y }));
    }
    virtual void draw(ImageDrawing_aa& img, const FindBorders& brd) const = 0;
    vector<vec2> getProcess(void) const {
        vector<vec2> result;
        for (auto& i : m_solved.x_process)
            result.push_back(vec2(i[0], i[1]));
        return result;
    }

    xn_t get_x_0(void) const { return m_x_0; }
    int get_iterations(void) const { return m_iterations; }
    double get_residual(void) const { return m_residual; }
    solved_t get_solved(void) const { return m_solved; }
protected:
    solved_t m_solved;
    fnm_f m_f;

    xn_t m_x_0;
    int m_iterations;
    double m_residual;
};

//-----------------------------------------------------------------------
class Picture_two_circles : public Picture
{
public:
    Picture_two_circles(circle a, circle b, xn_t x_0, bool is_numeric = false, const sqr_f&
    ↪  square_cast = square_cast_none, int iterations = 100, double residual = 1e-10) :
    ↪  m_a(a), m_b(b) {
        init_solve(two_circles(m_a, m_b), x_0, is_numeric, square_cast, iterations, residual);
    }

    void init_brd(FindBorders& brd) const {
        brd.process(m_a);
        brd.process(m_b);
        Picture::init_brd(brd);
    }

    void draw(ImageDrawing_aa& img, const FindBorders& brd) const {
        img.drawPolyline(calc_circle(m_a, brd));
        img.drawPolyline(calc_circle(m_b, brd));
    }
private:
    circle m_a, m_b;
};

//-----------------------------------------------------------------------
class Picture_one_circle : public Picture
{
public:
    Picture_one_circle(circle a, xn_t x_0, bool is_numeric = false, const sqr_f& square_cast
    ↪  = square_cast_none, int iterations = 100, double residual = 1e-10) : m_a(a) {
        init_solve(one_circle(m_a), x_0, is_numeric, square_cast, iterations, residual);
    }

    void init_brd(FindBorders& brd) const {
        brd.process(m_a);
        Picture::init_brd(brd);
    }

    void draw(ImageDrawing_aa& img, const FindBorders& brd) const {
        img.setPen(Pen(1, Blue));
        img.drawPolyline(calc_circle(m_a, brd));
    }
private:
    circle m_a;
};

//-----------------------------------------------------------------------
class Picture_two_circles_and_line : public Picture
{
public:
    Picture_two_circles_and_line(circle a, circle b, line c, xn_t x_0, bool is_numeric =
    ↪  false, const sqr_f& square_cast = square_cast_none, int iterations = 100, double
    ↪  residual = 1e-10) : m_a(a), m_b(b), m_la(c.a.x, c.a.y), m_lb(c.b.x, c.b.y) {
        init_solve(two_circles_and_line(m_a, m_b, c), x_0, is_numeric, square_cast,
        ↪  iterations, residual);
    }

    void init_brd(FindBorders& brd) const {
        brd.process(m_a);
        brd.process(m_b);
        Picture::init_brd(brd);
    }

    void draw(ImageDrawing_aa& img, const FindBorders& brd) const {
        img.drawPolyline(calc_circle(m_a, brd));
        img.drawPolyline(calc_circle(m_b, brd));
        draw_line(img, brd, m_la, m_lb);
    }
private:
    circle m_a, m_b;
    vec2 m_la, m_lb;
};

//-----------------------------------------------------------------------
class Picture_three_circles : public Picture
{
public:
    Picture_three_circles(circle a, circle b, circle c, bool a_in, bool b_in, bool c_in, xn_t
    ↪  x_0, bool is_numeric = false, const sqr_f& square_cast = square_cast_none, int
    ↪  iterations = 100, double residual = 1e-10) : m_a(a), m_b(b), m_c(c), m_res_circle(a) {
        init_solve(three_circles(m_a, m_b, m_c, a_in, b_in, c_in), x_0, is_numeric,
        ↪  square_cast, iterations, residual);
        m_res_circle.c = point(m_solved.point[0], m_solved.point[1]);
        m_res_circle.r = m_solved.point[2];
    }

    void init_brd(FindBorders& brd) const {
        brd.process(m_a);
        brd.process(m_b);
        brd.process(m_c);
        brd.process(m_res_circle);
        Picture::init_brd(brd);
    }

    double getResidual(vec2 a) const {
        double r = fabs(sqrt(sqr(m_a.c.x-a.x)+sqr(m_a.c.y-a.y))-m_a.r);
        return length(m_f({a.x, a.y, r}));
    }

    void draw(ImageDrawing_aa& img, const FindBorders& brd) const {
        img.drawPolyline(calc_circle(m_a, brd));
        img.drawPolyline(calc_circle(m_b, brd));
        img.drawPolyline(calc_circle(m_c, brd));

        Pen oldPen = img.getPen();
        oldPen.clr = Orange;
        img.setPen(oldPen);
        img.drawPolyline(calc_circle(m_res_circle, brd));
    }
private:
    circle m_a, m_b, m_c, m_res_circle;
};

//-----------------------------------------------------------------------
class Picture_three_lines : public Picture
{
public:
    Picture_three_lines(vec2 a, vec2 b, vec2 c, xn_t x_0, bool is_numeric = false, const
    ↪  sqr_f& square_cast = square_cast_none, int iterations = 100, double residual = 1e-10)
    ↪  : m_a(a), m_b(b), m_c(c) {
        point pa(a.x, a.y), pb(b.x, b.y), pc(c.x, c.y);
        init_solve(three_lines({pa, pb}, {pb, pc}, {pa, pc}), x_0, is_numeric, square_cast,
        ↪  iterations, residual);
    }

    void init_brd(FindBorders& brd) const {
        brd.process(m_a);
        brd.process(m_b);
        brd.process(m_c);
        Picture::init_brd(brd);
    }

    void draw(ImageDrawing_aa& img, const FindBorders& brd) const {
        img.setPen(Pen(1, Blue));

        draw_line(img, brd, m_a, m_b);
        draw_line(img, brd, m_a, m_c);
        draw_line(img, brd, m_b, m_c);
    }
private:
    vec2 m_a, m_b, m_c;
};

//-----------------------------------------------------------------------
class Picture_sin_and_line : public Picture
{
public:
    Picture_sin_and_line(vec2 a, xn_t x_0, bool is_numeric = false, const sqr_f& square_cast
    ↪  = square_cast_none, int iterations = 100, double residual = 1e-10) : m_a(a), m_b(0,
    ↪  0) {
        init_solve(sin_and_line(point(a.x, a.y)), x_0, is_numeric, square_cast, iterations,
        ↪  residual);
    }

    void init_brd(FindBorders& brd) const {
        brd.process(m_a);
        brd.process(vec2(-3, -3));
        Picture::init_brd(brd);
    }

    void draw(ImageDrawing_aa& img, const FindBorders& brd) const {
        draw_line(img, brd, m_a, m_b);

        // Рисуем синусоиду
        vec2 last;
        for (int i = 0; i < 1000; ++i) {
            double x = (i - 500.0) / 100.0;
            double y = sin(x);
            vec2 current(x, y);
            if (i != 0)
                img.drawLine(brd.toImg(last), brd.toImg(current));
            last = current;
        }
    }
private:
    vec2 m_a, m_b;
};

//-----------------------------------------------------------------------
void draw_picture(const Picture& pic, wstring filename, int pic_size) {
    const double step_count = 20;
    FindBorders brd(pic_size - 40, 20, true);
    pic.init_brd(brd);
    brd.finish();

    // Создание изображения
    Point_i size(brd.getCalculatedSize().x, brd.getCalculatedSize().y);
    ImageDrawing_aa img(size);

    vector<vector<double>> mas(size.x, vector<double>(size.y));
    img.clear(White);

    // Заполняем массив невязкой по всем координатам
    double max1 = -100000000000;
    double min1 = 100000000000;
    for (int i = 0; i < size.x; i++) {
        for (int j = 0; j < size.y; j++) {
            mas[i][j] = pic.getResidual(brd.fromImg(vec2(i, j)));
            min1 = min(min1, mas[i][j]);
            max1 = max(max1, mas[i][j]);
```

```
        }
    }

    // Рисуем все нормированные точки из массива
    for (int i = 0; i < size.x; i++) {
        for (int j = 0; j < size.y; j++) {
            double v = mas[i][j];
            double pos = (v - min1)/(max1-min1);
            pos = sqrt(pos);
            pos = int(pos*step_count)/step_count;
            img[Point_i(i, j)] = getColorBetween(pos, White, Black);
        }
    }

    // Рисуем оси координат
    img.setPen(Pen(1.5, Red));
    vec2 xa = brd.toImg(vec2(-1, 0));
    vec2 xb = brd.toImg(vec2(1, 0));
    vec2 ya = brd.toImg(vec2(0, -1));
    vec2 yb = brd.toImg(vec2(0, 1));
    if (xa.y >= 0 && xa.y <= size.y) {
        xa.x = -1;
        xb.x = size.x;
        img.drawLine(xa, xb);
    }
    if (ya.x >= 0 && ya.x <= size.x) {
        ya.y = -1;
        yb.y = size.y;
        img.drawLine(ya, yb);
    }

    img.setPen(Pen(2, Blue));
    pic.draw(img, brd);

    // Рисуем линии поиска решения
    auto process = pic.getProcess();
    for (int i = 0; i < process.size()-1; i++) {
        auto a = brd.toImg(process[i]);
        auto b = brd.toImg(process[i+1]);

        img.setPen(Pen(3.5, setAlpha(Green, 100)));
        img.drawLine(a, b);
    }

    // Рисуем точки поиска решения
    for (int i = 0; i < process.size(); i++) {
        auto a = brd.toImg(process[i]);

        img.setBrush(Brush(setAlpha(twg::Miku, 200)));
        img.drawPolygon(computeEllipse(Point_d(4, 4)).move(a));
    }

    saveToBmp(&img, filename + L".bmp");

    // TODO добавить сюда вывод в файл процесса
    ofstream fout(filename + L".dat");

    auto solved = pic.get_solved();
    fout << "%\tbeta\tresIdual\t";
    if (solved.point.size() == 2)
        fout << "x\ty" << endl;
    else
        fout << "x\ty\tr" << endl;
    for (int i = 1; i < solved.x_process.size(); i++) {
        fout << i << setprecision(2) << scientific << "\t" << solved.beta_process[i] << "\t"
          << solved.residual_process[i] << "\t" << setprecision(10) << fixed;
        auto point = solved.x_process[i];
        for (int j = 0; j < point.size() - 1; j++)
            fout << point[j] << "\t";
        fout << point.back() << endl;
    }

    fout << setprecision(10) << fixed;
    fout << "# x_0 = " << pic.get_x_0() << endl;
    fout << "# max_iter = " << pic.get_iterations() << endl;
    fout << setprecision(2) << scientific;
    fout << "# required residual = " << pic.get_residual() << endl;

    fout << endl << "# ";

    switch (solved.exit_type) {
        case EXIT_ITER: fout << "Exit by iterations" << endl; break;
        case EXIT_RESIDUAL: fout << "Exit by residual" << endl; break;
        case EXIT_STEP: fout << "Exit by step" << endl; break;
        case EXIT_BETA: fout << "Exit by beta" << endl; break;
        case EXIT_ERROR: fout << "Exit by error" << endl; break;
    }

    fout << "# iter = " << pic.get_solved().iterations << endl;
    fout << setprecision(2) << scientific;
    fout << "# Residual = " << pic.get_solved().residual << endl;
    fout << setprecision(10) << fixed;
    fout << "# Result point = " << pic.get_solved().point << endl;

    fout.close();
}

int CALLBACK WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR lpCmdLine, int
  ↪  nCmdShow) {
    const double required_residual = 1e-10;
    const int maxiter = 30;
    #ifdef _DEBUG
        int pic_size = 50;
    #else
        int pic_size = 700;
    #endif

    // Делаем таблицы для сходимости метода в зависимости от размера шага при взятии производ
ной
    for (int i = 0; i < 10; i++) {
        partial_derivative_step = pow(2.0, double(-i));
        circle a(0, 0, 0.8), b(sqrt(2.0) / 2, sqrt(2.0) / 2, 0.2);
        line c = { point(sqrt(2.0)*(1.0 / 2.0 - 0.1), sqrt(2.0)*(1.0 / 2.0 - 0.1)), point(-1,
          ↪  0) };
        Picture_two_circles_and_line pic(a, b, c, { -1, 0.5 }, true, square_cast_3, maxiter,
          ↪  required_residual);
        draw_picture(pic, L"img/two_circles_and_line_3_step_" + to_wstring(i) +
          ↪  L"_41ab_numeric", pic_size);
    }

    auto make_images = [&](bool is_numeric, wstring append) {
        /*{
            circle a(0, 0, 1), b(5, 0, 1), c(10, 5, 5);
            Picture_three_circles pic(a, b, c, false, false, false, { -1, -1, 13 },
  ↪  is_numeric, square_cast_none, 100, 1e-10);
            draw_picture(pic, L"img/three_circles_41ab" + append, pic_size);
        }*/

        auto draw_three_lines = [&](int no, const sqr_f& sq) {
            vec2 a(0, 0), b(1, 2), c(3, 1);
            Picture_three_lines pic(a, b, c, { -3, -3 }, is_numeric, sq, maxiter,
              ↪  required_residual);
            draw_picture(pic, L"img/three_lines_41ab_" + to_wstring(no) + append, pic_size);
        };

        draw_three_lines(2, square_cast_2);
        draw_three_lines(3, square_cast_3);
        draw_three_lines(4, square_cast_4);

        {
            vec2 a(0.5, 0.5), b(3, 1), c(-1, 3);
            Picture_three_lines pic(a, b, c, { -3, -3 }, is_numeric,
              ↪  composition(square_cast_3, square_cast_mul({1, 1, 10})), 100,
              ↪  required_residual);
            draw_picture(pic, L"img/three_lines_41ab_3_weight" + append, pic_size);
        }

        {
            vec2 a(2, 2);
            Picture_sin_and_line pic(a, { -1, 1 }, is_numeric, square_cast_none, maxiter,
              ↪  required_residual);
            draw_picture(pic, L"img/sin_and_line_41ab" + append, pic_size);
        }

        {
            circle a(0, 0, 1), b(1, 0.7, 1.2); // Пересекаются в двух точках
            Picture_two_circles pic(a, b, { -1, 0.5 }, is_numeric, square_cast_none, maxiter,
              ↪  required_residual);
            draw_picture(pic, L"img/two_circles_1_41ab" + append, pic_size);
        }

        {
            circle a(0, 0, 0.8), b(sqrt(2.0) / 2, sqrt(2.0) / 2, 0.2); // Пересекаются в
              ↪  одной точке
            {
                // Начальное приближение лежит на оси симметрии
                Picture_two_circles pic(a, b, { 0.156405, 0.974966 }, is_numeric,
                  ↪  square_cast_none, maxiter, required_residual);
                draw_picture(pic, L"img/two_circles_2_1_41ab" + append, pic_size);
            }
            {
                // Начальное приближение лежит на линии, соединяющей центры окружностей
                Picture_two_circles pic(a, b, { -sqrt(2.0), -sqrt(2.0)+0.01 }, is_numeric,
                  ↪  square_cast_none, maxiter, required_residual);
                draw_picture(pic, L"img/two_circles_2_2_41ab" + append, pic_size);
            }
            {
                // Начальное приближение лежит в центре одной из окружностей
                Picture_two_circles pic(a, b, { b.c.x, b.c.y + 0.01 }, is_numeric,
                  ↪  square_cast_none, maxiter, required_residual);
                draw_picture(pic, L"img/two_circles_2_3_41ab" + append, pic_size);
            }
        }

        {
            circle a(0, 0, 0.8), b(1, 0.7, 0.3); // Не пересекаются
            Picture_two_circles pic(a, b, { -1, 0.5 }, is_numeric, square_cast_none, maxiter,
              ↪  required_residual);
            draw_picture(pic, L"img/two_circles_3_41ab" + append, pic_size);
        }

        auto draw_two_circles_and_line = [&](int no, const sqr_f& sq) {
            circle a(0, 0, 0.8), b(sqrt(2.0) / 2, sqrt(2.0) / 2, 0.2);
            line c = { point(sqrt(2.0)*(1.0 / 2.0 - 0.1), sqrt(2.0)*(1.0 / 2.0 - 0.1)),
              ↪  point(-1, 0) };
            Picture_two_circles_and_line pic(a, b, c, { -1, 0.5 }, is_numeric, sq, maxiter,
              ↪  required_residual);
            draw_picture(pic, L"img/two_circles_and_line_" + to_wstring(no) + L"_41ab" +
              ↪  append, pic_size);
        };

        draw_two_circles_and_line(2, square_cast_2);
        draw_two_circles_and_line(3, square_cast_3);
        draw_two_circles_and_line(4, square_cast_4);

        {
            circle a(0, 0, 1);
            Picture_one_circle pic(a, { 0.2, 0.5 }, is_numeric, square_cast_1, maxiter,
              ↪  required_residual);
            draw_picture(pic, L"img/one_circle_41ab_1" + append, pic_size);
        }
    };
    make_images(false, L"_analytic");
    make_images(true, L"_numeric");
}
```