Outer Limits

September 17, 2021

Contents

1 Introduction

`Outer_Limits` is an experimental arbitrary-precision semidefinite program solver, much like `SDPB`, but with a markedly different strategy. `SDPB` will find a solution by adding up polynomial matrices which are each guaranteed to be semidefinite. This sum becomes more and more ill-conditioned matrix as `SDPB` iterates towards a solution. This ill-conditioned matrix must be inverted in order to continue to the next iteration, causing numerical problems. `SDPB` overcomes this by requiring very large precision arithmetic.

`Outer_Limits`, on the other hand, starts by enforcing positive semidefiniteness only at a finite set of points. These are constant constraints, so there is no longer a problem with ill-conditioned matrices. This allows `Outer_Limits` to find a solution using much lower precision.

Initially, this will find a solution that is, in general, not positive semidefinite everywhere. So `Outer_Limits` will add more points to the places where the function is not positive semidefinite and find another solution. This iterative procedure continues until the solution is positive everywhere.

This strategy also allows `Outer_Limits` to be more easily applied to general problems. `SDPB` requires that all inputs be approximated as polynomials. `Outer_Limits`, in contrast, only requires evaluations of the functions at zeros of Chebyshev polynomials.

`Outer_Limits` is bundled with `SDPB`. See `Install.md` for up-to-date instructions on getting pre-made binaries or building from source.

2 Semidefinite Problems

Consider a system with J blocks. Within each block, there is a set of inequality constraints on an N-dimensional vector y. In addition, there is a known vector b, independent of the blocks. The goal of semidefinite programming is to find a solution to these constraints while trying to maximize the quantity $b \cdot y$.

Writing this more formally, consider a collection of symmetric $R_j \times R_j$ matrices

$$M_j^n(\Delta) \;=\; \begin{pmatrix} m_{j,11}^n(\Delta) & \dots & m_{j,1m_j}^n(\Delta) \\ \vdots & \ddots & \vdots \\ m_{j,R_j1}^n(\Delta) & \dots & m_{j,R_jR_j}^n(\Delta) \end{pmatrix} \tag{2.1}$$

where $0 \leq n \leq N$ and $1 \leq j \leq J$, and each element $m_{j,rs}^n(\Delta)$ is an arbitrary function in Δ. Define the matrix $F_j\left(\Delta\right)$

$$F_j\left(\Delta\right) = M_j^0(\Delta) + \sum_{n=1}^{N} y_n M_j^n(\Delta). \tag{2.2}$$

Then given $b \in \mathbb{R}^N$, `Outer_Limits` will solve for coefficients y_n that

$$\begin{array}{ll} \text{maximize} & b_0 + b \cdot y \quad \text{over} \quad y \in \mathbb{R}^N, \\ \text{such that} & F_j\left(\Delta\right) \succeq 0 \quad \text{for all } \Delta \geq 0 \text{ and } 1 \leq j \leq J. \end{array} \tag{2.3}$$

The notation $F_j\left(\Delta\right) \succeq 0$ means $F_j\left(\Delta\right)$ is positive semidefinite.

3 Input

Since `Outer_Limits` solves these problems on a computer, there are inevitable approximations that must occur. The main one is that the functions $m_{j,rs}^n(\Delta)$ must be well approximated by a series of Chebyshev polynomials. Specifically, the input to `Outer_Limits` is not the functions $m^n_{j,rs}(\Delta)$, but rather their values at Chebyshev zeros and their relative sizes near zero and at infinity.

The example in the `SDPB` Manual is

$$f_1 = 1 + \Delta^4, \tag{3.1}$$

$$f_2 = \Delta^2 + \frac{\Delta^4}{12}. \tag{3.2}$$

This is a system with a single block ($J = 1$) and two weights ($N = 2$). We will approximate these functions by Chebyshev polynomials covering 0 to $\Delta_{j,max}$. So we need to decide on a $\Delta_{j,max}$ that is sufficient to cover all of the places where the functions might go to zero. In general, we want to choose a large enough $\Delta_{j,max}$ such that the functions are dominated by the largest terms at infinity. In this case, the functions f_1 and f_2 have polynomial terms that are of equal size at $\Delta = 1$ and $\Delta = \sqrt{12} \simeq 3.464$ respectively. So to be conservative, we choose $\Delta_{j,max} = 100$. Currently, `Outer_Limits` requires $\Delta_{j,max}$ to be the same for each function in a block.

Now we need to evaluate f_1 and f_2 at scaled and translated zeros of Chebyshev polynomials. This means that for an approximation with N points, we need values at

$$\Delta_{j,k} = \frac{\Delta_{j,max}}{2}\left(1 + \cos\left(\frac{\pi\left(k + \frac{1}{2}\right)}{N}\right)\right),\ 0 \leq k < N \tag{3.3}$$

Since f_1 and f_2 are polynomials, we could use $N = 5$ to get a perfect approximation. For this exercise, we will pretend we do not know that and choose $N = 10$. `Outer_Limits` currently requires N to be the same for each function in a block.

After generating the value of these functions at these points, we must characterize the relative behavior of the different functions near zero and at infinity. This characterization turns out to be important for computing high precision answers for the objective.

In this case, the characterization is straightforward. f_1 and f_2 are both degree=4 polynomials. Near zero, f_1 goes to a constant term and f_2 goes to zero. So we can use $f_1(0) = 1$ and $f_2(0) = 0$ for the limiting behavior near zero. If f_1 did not have a constant term, then we would have to compare the relative sizes of the lowest degree non-zero coefficient. At infinity, we only need to know the relative sizes of the coefficients of Δ^4: 1 and $1/12 \sim 0.083333333333333$.

Now we can construct the `functions` input file shown in listing 1.

Listing 1: `functions` input file. The first block is f_1, and the second block is f_2.

```
{
 "objective": ["0", "-1"],
 "normalization": ["1", "0"],
 "functions":
 [
  [
   [
```

```
    [
     {
      "max_delta" : "100",
      "epsilon_value" : "1",
      "infinity_value" : "1",
      "chebyshev_values" :
      [
       "1.1435973374045645814919250171",
       "883.02729988511593507402953969",
       "45996.705504467834968336421068",
       "555496.84436116128227452603002",
       "3164867.0107610102130932508775",
       "11178001.455036876356972361751",
       "27933561.314132283966434754990",
       "53079005.294495532165031663578",
       "79919384.706784958935035978021",
       "97560312.498026486725689472807"
      ]
     },
     {
      "max_delta" : "100",
      "epsilon_value" : "0",
      "infinity_value" : "0.0833333333333333333333333333333333333333",
      "chebyshev_values" :
      [
       "0.3909088380370479104722553418",
       "103.20121941417842758517038120",
       "4047.4415527729152920238132784",
       "47036.636299366781486947578226",
       "265517.84125951408811424149736",
       "934843.38959957190879426845671",
       "2333081.9137776891397492374085",
       "4430535.8917805604180413095200",
       "6668888.4064183873419762575987",
       "8139903.2205172185244035518177"
      ]
     }
    ]
   ]
  ]
 ]
}
```

The exact format is specified in the JSON schema `docs/functions_schema.json`. At its heart, there is a set of nested arrays. The outermost array corresponds to the j index of $m^n_{j,rs}(\Delta)$, followed by r, s, and n. The toy example has $N = 2$, $J = 1$, and $R_1 = 1$, so $j = r = s = 1$ and $n = 1$ or 2.

The last thing to do is to create a `points` file that contains the values of Δ where `Outer_Limits` will initially apply positivity constraints. `Outer_Limits` will also apply constraints at epsilon and infinity. At a minimum, you will need to supply enough points such that there are at least as many constraints as there are degrees of freedom in y.

For the toy problem, there are only two functions, so in principle you only need two points. Epsilon and infinity are always used, so you not add any more points. In practice, `Outer_Limits` will get stuck if you do not exceed the minimum by a healthy margin.

With that in mind, we choose the two points, 0.1 and 1, resulting in the `points` input file in listing 2. Needless to say, the choice of points will depend strongly on the problem you are solving.

Listing 2: `poinst` input file

```
{
 "points": [["0.1", "1"]]
}
```

The format for this file is specified in the JSON schema `docs/points_schema.json`. The outer array corresponds to the j index of $m^n_{j,rs}(\Delta)$, and the inner array is a list of points for that index.

4 Running `Outer_Limits`

4.1 Initial Iteration and Scaling

When `Outer_Limits` is running, it will first try to solve equation 2.3, but with a duality gap of 1.1. This very relaxed constraint on the duality gap allows it to quickly get a very approximate solution. Also, `Outer_Limits` is only enforcing the solution at the initial points via constant constraints.

One wrinkle here is that `Outer_Limits` applies several transformations in order to scale the problem before making use of `SDPB`. In the usual formulation, `SDPB` maximizes

$$b \cdot y \tag{4.1}$$

subject to the constraints

$$Y + B \cdot y = c \tag{4.2}$$

$$Y \succeq 0 \tag{4.3}$$

B is a matrix corresponding to M^n_j, and c is M^0_j, both evaluated at the initial values of Δ. However, the relative sizes of c and the columns of B can be wildly different, leading to numerical difficulties. To ameliorate this, we first find c_{max}, the largest value of $|c|$. If c is completely zero, we use $c_{\text{max}} = 1$.

Next, we compute the singular value decomposition of B

$$B = U\Sigma V^T. \tag{4.4}$$

U and V are real orthogonal matrices, and Σ is a rectangular diagonal matrix with non-negative entries. We define new variables

$$\begin{aligned}
y' &= \Sigma V^T y/c_{\max} && (4.5)\\
b' &= bV\Sigma^\dagger c_{\max} && (4.6)\\
c' &= c/c_{\max} && (4.7)\\
Y' &= Y/c_{\max} && (4.8)\\
B' &= BV\Sigma^\dagger/c_{\max} && (4.9)
\end{aligned}$$

where $\Sigma^\dagger$ is the pseudo-inverse of Σ. In this case, $\Sigma^\dagger$ is the transpose of Σ with the diagonal entries inverted. So $\Sigma^\dagger$ is easy to compute once we have Σ. The system that we initially solve with `SDPB` routines is then

$$\text{maximize } b' \cdot y' \tag{4.10}$$

subject to the constraints

$$\begin{aligned}
Y' + B' \cdot y' &= c && (4.11)\\
Y' &\succeq 0 && (4.12)
\end{aligned}$$

Computing the singular value decomposition of B can be very expensive. To amortize this cost over the whole calculation, we save the expression $V\Sigma^\dagger$. Then, at each iteration, we compute a new B' using equation 4.9. With this method, B' is only guaranteed to be unitary on the first iteration. However, later iterations will only add rows to B, so it should still work reasonably well at equalizing the columns.

If your problem is very large, computing the SVD may be impractical. In that case, you can set the option `--useSVD=0` to skip it and solve equations 4.1, 4.3, 4.3 directly.

4.2 Adapted Mesh

With an initial solution in hand, the next step is to check for regions within each block where the functional is negative. To enable this, `Outer_Limits` creates a coarse mesh of cells over the range from 0 to $\Delta_{j,max}$. Within each cell, it evaluates the functional F_j using the Chebyshev approximations for M_j^n and the solution values for y_n. If the functional has too much curvature and the points are not so close together as to cause numerical difficulties, it splits the cell into smaller cells.

To be precise, for each block, we compute the largest value of the functional among all the Chebyshev zeros

$$F_{j,\max} = \max\left(\left|F_j\left(\Delta_{j,k}\right)\right|\right) \tag{4.13}$$

In addition, for a cell with a width δ, we define an interpolated averaged value

$$\overline{F}_j(\Delta) = (F_j(\Delta+\delta) + F_j(\Delta-\delta))/2 \tag{4.14}$$

Then we split the cell if the interpolated averaged value differs too much from the actual value at that point

$$\begin{aligned}\left|\overline{F}_j(\Delta) - F_j(\Delta)\right| &< \epsilon_{\text{relative}}\left|\overline{F}_j(\Delta) + F_j(\Delta)\right| && (4.15)\\ \left|\overline{F}_j(\Delta) - F_j(\Delta)\right| &< \epsilon_{\text{absolute}}F_{j,\max}. && (4.16)\end{aligned}$$

The default mesh tolerance $\epsilon_{\text{relative}} = \frac{1}{1000}$ is somewhat arbitrary. The goal is to catch all of the places where F_j varies strongly, and in this way we can find all of the places where it becomes negative. You can set $\epsilon_{\text{relative}}$ with the option `--meshThreshold`.

The tolerance $\epsilon_{\text{absolute}}$ is the smallest difference from 1 that can be represented, given the working precision. For example, with IEEE 754 double precision floating point numbers, this is 2.22045e-16.

4.3 Checking Positivity

With this adapted mesh in place, `Outer_Limits` checks every cell for positivity. For each cell, we compute the first and second derivative numerically

$$\begin{aligned}dF_j &= (F_j(\Delta+\delta) - F(\Delta-\delta))/(2\delta), && (4.17)\\ d^2F_j &= (F_j(\Delta+\delta) - 2F(\Delta) + F(\Delta-\delta))/\delta^2. && (4.18)\end{aligned}$$

Using these to form a quadratic approximation to F_j, we compute the location and value where F_j is minimum

$$\begin{aligned}\Delta_{\min} &= \Delta - dF_j/d^2F_j, && (4.19)\\ F_{\min} &= F(\Delta) - (dF_j)^2/\left(2d^2F_j\right). && (4.20)\end{aligned}$$

At first, we might think that only if $F_{\min} < 0$ do we need to add $\Delta_{\min}$ to the list of constraint points. However, if $F_{\min} > 0$, it may only be positive due to inaccuracies in the quadratic approximation. Conversely, if $F_{\min} < 0$, it may only be negative due to inaccuracies in the finite precision arithmetic. To handle these cases, we add $\Delta_{\min}$ to the list of constraint points if and only if

$$\begin{aligned}F_{\min} &< \left|F(\Delta) - \overline{F}_j(\Delta)\right|, && (4.21)\\ |F_{\min}| &> \epsilon_{\text{absolute}}F_{j,\max}. && (4.22)\end{aligned}$$

4.4 Further Iterations and Termination Criteria

With these new points, `Outer_Limits` creates and solves a new system as in section 4.1. Eventually, there will be no new points. Then `Outer_Limits` reduces the duality gap by dualityGapReduction and iterates again until there are no new points. Outer_Limits continues this cycle of finding new points and reducing the duality gap until the duality gap is less than dualityGapThreshold. This guarantees that the final result will fully satisfy all of the constraints in 2.3 without having to spend a lot of effort computing highly precise solutions for very approximate problems.

5 Output of SDPB

5.1 Terminal Output

The output from running Outer_Limits on the example problem in section 3 is in listing 3. The beginning is very similar to SDPB. When iterating, Outer_Limits first prints the current number of constraints, including all of the blocks, and the current gap threshold. After each iteration, Outer_Limits prints the full array of weights and the computed value of the objective. If Outer_Limits decides not to add points, it will reduce the duality gap threshold, print this new threshold, and continue with the current solution.

Figures 1, 2, and 3 show the behavior of F_0 during the iterations as it alternates between adding points and reducing the gap threshold.

Listing 3: Output of Outer_Limits for the input file in listing 2

```
Outer_Limits started at 2021-Jun-16 21:16:54
functions file  : "test/toy_functions.json"
out directory   : "test/toy_functions_out.json"

Parameters:
dualityGapReduction          = 1024
maxIterations                = 1000
maxRuntime                   = 9223372036854775807
checkpointInterval           = 3600
findPrimalFeasible           = false
findDualFeasible             = false
detectPrimalFeasibleJump     = false
detectDualFeasibleJump       = false
precision(actual)            = 128(128)
dualityGapThreshold          = 1e-10
primalErrorThreshold         = 1e-10
dualErrorThreshold           = 1e-10
initialMatrixScalePrimal     = 10
initialMatrixScaleDual       = 10
feasibleCenteringParameter   = 0.1
infeasibleCenteringParameter = 0.3
stepLengthReduction          = 0.7
maxComplementarity           = 1e+100
initialCheckpointDir         = "test/toy_functions.ck"
checkpointDir                = "test/toy_functions.ck"
writeSolution                =
verbosity                    = 1

num_constraints: 4
Threshold: 1.1
weight: [1, 17.92002209012499307550986390082525582465]
optimal: -17.92002209012499307550986390082525582465
Threshold: 0.001074218750000000086736173798840354720596
weight: [1, -1.845764788512892377453602431656624264456]
optimal: 1.845764788512892377453602431656624264454
```

```
Saving checkpoint to    : "test/toy_functions.ck"
num_constraints: 5
Threshold: 0.001074218750000000086736173798840354720596
weight: [1, -1.839611388308253894657545680555773464748]
optimal: 1.839611388308253894657545680555773464748
Threshold: 1.049041748046875084703294725430033906832e-06
weight: [1, -1.840275804402260393066356407589470103775]
optimal: 1.840275804402260393066356407589470103773
Saving checkpoint to    : "test/toy_functions.ck"
num_constraints: 6
Threshold: 1.049041748046875084703294725430033906832e-06
weight: [1, -1.840265160571370863263682954232296794857]
optimal: 1.840265160571370863263682954232296794854
Threshold: 1.024454832077026449905561255302767487141e-09
weight: [1, -1.840265762727976760495004292302021649703]
optimal: 1.840265762727976760495004292302021649702
Threshold: 1e-10
weight: [1, -1.840265763127732853472949500621348513756]
optimal: 1.840265763127732853472949500621348513754
Saving checkpoint to    : "test/toy_functions.ck"
optimal: 1.840265763127732853472949500621348513754
Saving solution to "test/toy_functions_out.json"
```

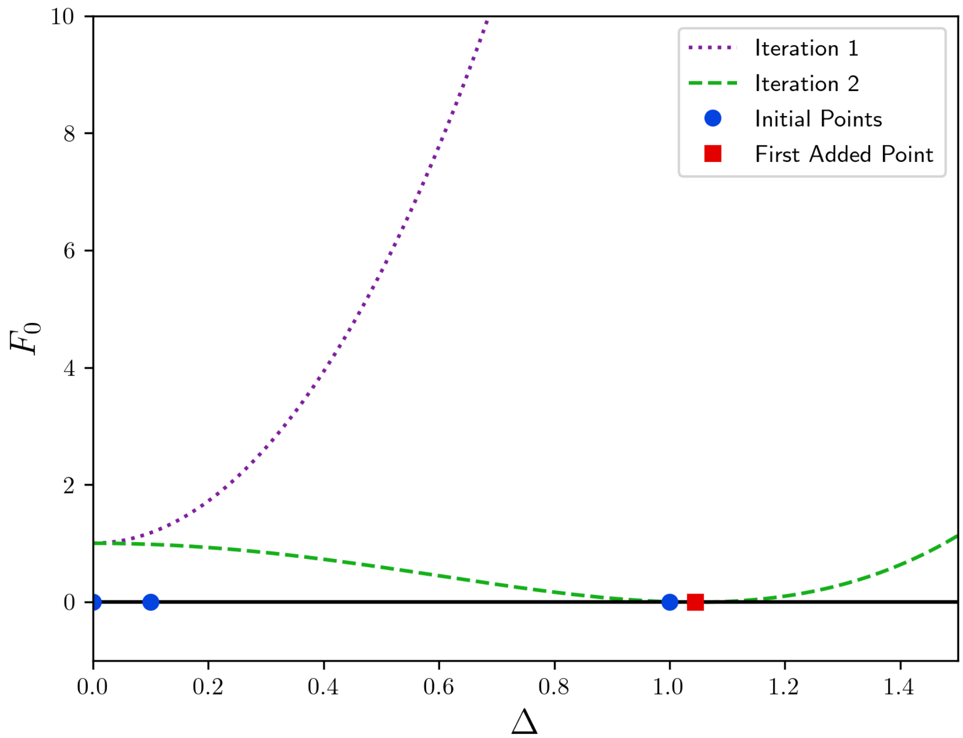

Figure 1: Plot of $F_0 = y_0\left(1+x^4\right) + y_1\left(\frac{x^4}{12}+x^2\right)$ after the first iteration, with duality gap threshold=1.1, and the second iteration, with duality gap threshold=1.07×10^{-3}. After the second iteration, `Outer_Limits` adds a point at 1.044.

5.2 Termination

`Outer_Limits` can fail when the inner `SDPB` routines fail, so `Outer_Limits` has the same failure termination criteria. `Outer_Limits`'s only successful termination criteria is if the duality gap is less than `dualityGapThreshold`.

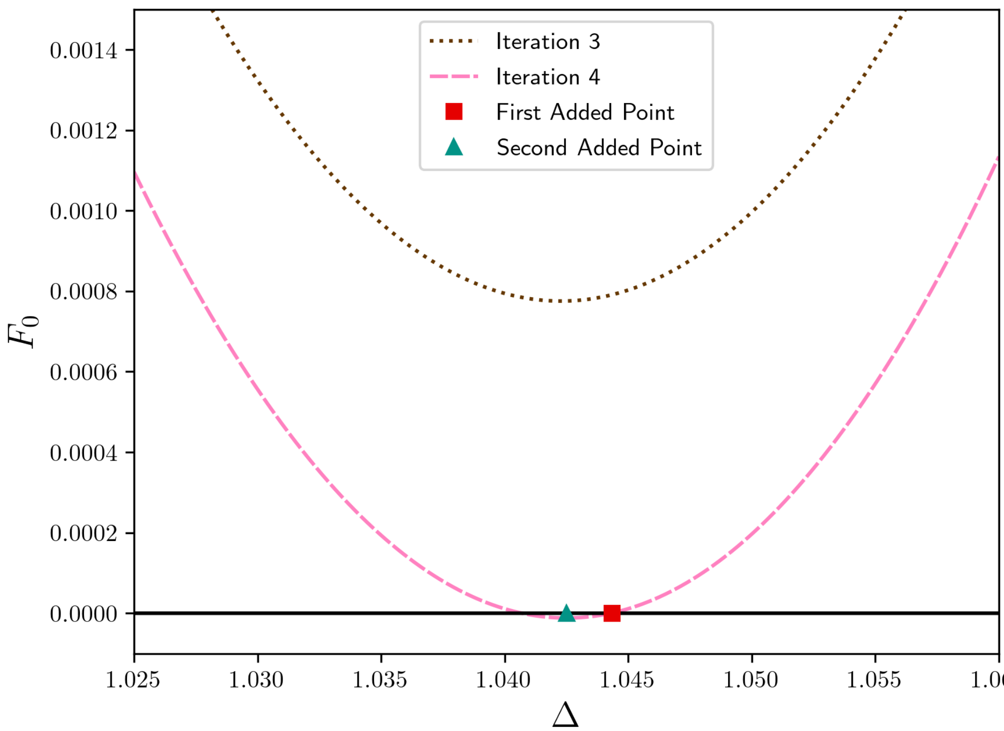

Figure 2: Plot of F_0 after the third iteration, with duality gap threshold=1.07×10^{-3}, and the fourth iteration, with duality gap threshold=1.05×10^{-6}. After the fourth iteration, `Outer_Limits` adds a point at 1.042500.

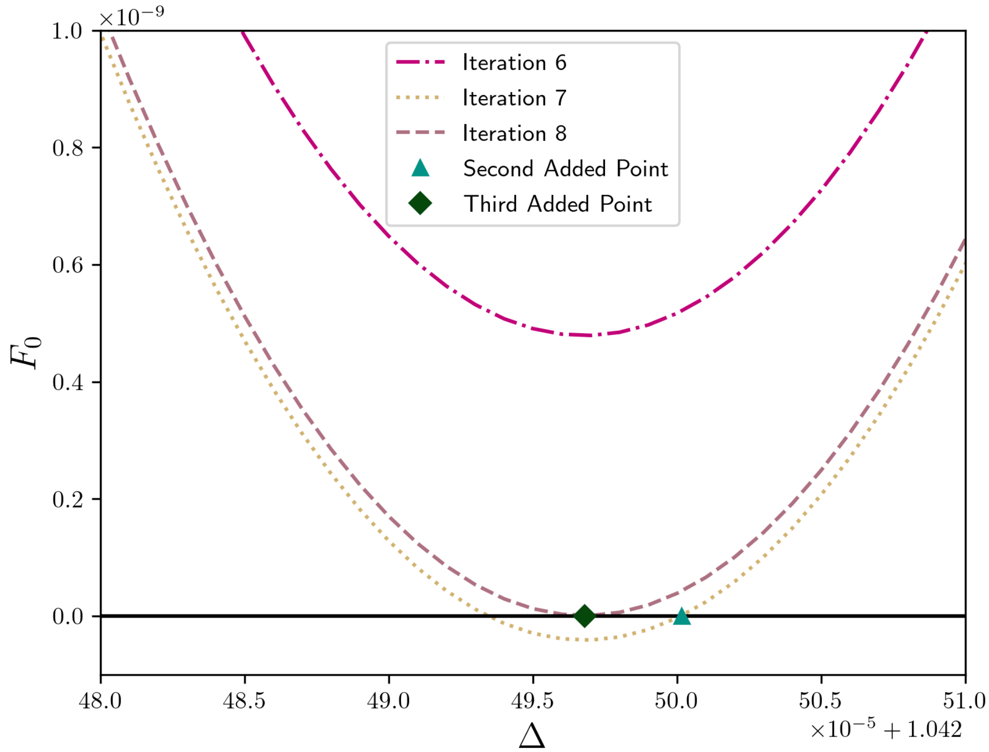

Figure 3: Plot of F_0 after the fifth, sixth, and seventh iterations, with duality gap thresholds of 1.05×10^{-6}, 1.02×10^{-9}, and 1.02×10^{-9}. After the sixth iteration, `Outer_Limits` adds a point at 1.042497.

5.3 Output File

If Outer_Limits completes successfully, it writes the solution to a single JSON file, as shown in listing 4.

Listing 4: Solution output of Outer_Limits for the input file in listing 2


```
{
  "optimal": "1.840265763127732853472949500621348513754",
  "y":
  [
    "1",
    "-1.840265763127732853472949500621348513756"
  ],
  "options":
{
    "maxIterations": "1000",
    "maxRuntime": "9223372036854775807",
    "checkpointInterval": "3600",
    "findPrimalFeasible": "false",
    "findDualFeasible": "false",
    "detectPrimalFeasibleJump": "false",
    "detectDualFeasibleJump": "false",
    "precision": "128",
    "precision_actual": "128",
    "dualityGapThreshold": "1e-10",
    "primalErrorThreshold": "1e-10",
    "dualErrorThreshold": "1e-10",
    "initialMatrixScalePrimal": "10",
    "initialMatrixScaleDual": "10",
    "feasibleCenteringParameter": "0.1",
    "infeasibleCenteringParameter": "0.3",
    "stepLengthReduction": "0.7",
    "maxComplementarity": "1e+100",
    "initialCheckpointDir": "test\/toy_functions.ck",
    "checkpointDir": "test\/toy_functions.ck",
    "functions": "test\/toy_functions.json",
    "points": "test\/toy_functions_points.json",
    "out": "test\/toy_functions_out.json",
    "dualityGapReduction": "1024",
    "writeSolution": "",
    "verbosity": "1"
}
}
```


References

[1] David Simmons-Duffin, "A Semidefinite Program Solver for the Conformal Bootstrap," arXiv:1502.02033 [hep-th].

[2] V. S. Rychkov and A. Vichi, "Universal Constraints on Conformal Operator Dimensions," Phys. Rev. D **80**, 045006 (2009) arXiv:0905.2211 [hep-th].

[3] D. Poland, D. Simmons-Duffin and A. Vichi, "Carving Out the Space of 4D CFTs," JHEP **1205**, 110 (2012) arXiv:1109.5176 [hep-th].

[4] F. Kos, D. Poland and D. Simmons-Duffin, "Bootstrapping the $O(N)$ vector models," JHEP **1406**, 091 (2014) arXiv:1307.6856 [hep-th].

[5] F. Kos, D. Poland and D. Simmons-Duffin, “Bootstrapping Mixed Correlators in the 3D Ising Model,” JHEP **1411**, 109 (2014) arXiv:1406.4858 [hep-th].

[6] M. Yamashita, K. Fujisawa, M. Fukuda, K. Nakata, and M. Nakata, “A high-performance software package for semidefinite programs: SDPA 7,” Research Report B-463, Dept. of Mathematical and Computing Science, Tokyo Institute of Technology, Tokyo, Japan (2010).

[7] M. Yamashita, K. Fujisawa, and M. Kojima, “Implementation and evaluation of SDPA 6.0 (SemiDefinite Programming Algorithm 6.0),” Optimization Methods and Software” 18 491-505 (2003).

[8] M. Nakata, “A numerical evaluation of highly accurate multiple-precision arithmetic version of semidefinite programming solver: SDPA-GMP, -QD and -DD.,” 2010 IEEE International Symposium on Computer-Aided Control System Design (CACSD), 29-34 Sept 2010.

[9] C++ Standards Committee Library Working Group and other contributors, “BOOST C++ Libraries,” http://www.boost.org.

[10] Gnome Project, Libxml2, http://www.xmlsoft.org/

[11] J. Poulson, B. Marker, R. van de Geijn, J. Hammond, and N. Romero, “Elemental: A new framework for distributed memory dense matrix computations, ACM Transactions on Mathematical Software,” ACM Trans. Math. Softw. 39 2 13:1-24 (2013), doi:10.1145/2427023.2427030

[12] The GNU Multiprecision Library, https://gmplib.org/

[13] The GNU MPFR Library, https://www.mpfr.org/