Optimization

Anson Li

Contents

Listings

List of Figures

Barrier Method Solver

The barrier method solver `barrier_method.py` closely follows outline that was shown, it implements gradient search with a logarithmic barrier. A snippet of the code is shown in listing 1.

Listing 1: `barrier_method()` in Part 1

```
198 """
199 A is a matrix whose number of rows < number of columns.
200 Equality constrained to be on the affine set A_eq @ x = b_eq.
201 """
202 def barrier_method(f,x,t,mu,eps,alpha,beta,A,b,f_ineq,max_iterations=
      MAX_ITERATIONS):
203     """
204     number of inequalities = m
205     have to start from a strictly feasible point.
206 
207     Approximates the ideal barrier function by using a logarithmic barrier
          function.
208     As t -> \infty, the perturbed objective function apporaches the ideal
          objective function.
209     However, newton's method becomes increasingly difficult to solve for large
          values of t.
210     Use a centering technique.
211     """
212     # construct barrier
213 
214     # [ ln(-f_ineq_1(x)), ... , ln(-f_ineq_m(x)) ]
215     # so that if f_ineq_1 is positive (not feasible), then ln of a negative
          number is -infinity
216     # ln(0) gives us nan as well...
217 
218     # this is a function of x
219     log_vector = compose_entrywise(lambda x: jnp.log(-x), f_ineq)
220     phi = lambda x: (-1)*jnp.sum(log_vector(x))
221 
222     # deform the original objective function by log barrier and t
223     F = lambda t: lambda x: t*f(x) + phi(x)
224     Ft = F(t)
225 
226     xt = newtons_method(Ft,x,eps,A,b,alpha,beta,f_ineq) # ?? what is f_ineq
          doing here
227 
228     # compute m = number of contraints.
229     m = jnp.shape(f_ineq(x))[0] # to compute m, get length of f_ineq applied to
          a point
230 
231     # What if A = None? (unconstrained optimization)
232     counter = 0
233     while not barrier_stop_test(m,t,eps):
234         counter+=1
235         t*=mu
236         Ft = F(t) # new objective function
237         xt = newtons_method(Ft,xt,eps,A,b,alpha,beta,f_ineq) # ?? what is f_ineq
              doing here
238 
239         # print(f'Barrier({counter}): ({xt}, {f(xt)})')
240 
241         if counter>=max_iterations:
242             logger.warning(f'Barrier method failed to converge after
                  max_iterations = {max_iterations}.\nReturning x_last = xt.')
243             break
244     return xt
```

Test Cases

We have written a test generator to compare our performance against that of `cvxpy`. Let n, m, p be the dimension of the domain, number of inequality and equality constraints. We will generate objective functions of the form

$$f_0 = 2^{-1}\langle x,\ Px\rangle_{\mathbb{R}^n} + \langle q,\ x\rangle_{\mathbb{R}^n} + r,$$

with equality constraint $A_{\text{eq}}x = b_{\text{eq}}$, and inequality (affine) constraints: $A_{\text{ineq}}x \lesssim b_{\text{ineq}}$. The matrix P is positive definite and is generated using Cholesky's method and lower triangularization because

every positive operator admits a 'square-root' decomposition.

To determine the starting point of the algorithm, we have to get creative, because the equality-constraint set has measure zero in the inequality-constraint set. We used `cvxpy`'s solver to obtain a feasible (and optimal) point x^* to the problem, and we perturbed this point by some linear combination of the spanning vectors of the kernel of A_{eq}. This perturbation however may make us go out of the inequality-constrained region, and hence it is run on a while-loop until we get a solution that also satisfies the inequality constraints.

Comparison with `cvxpy`

One can use the command `compare_against_cvxpy(num_tests)`. The dimension of the domain is programmed to be less than 6 due to limitations in computer resources. ,

```
user@s-Laptop ecse507-final % python3 -i barrier_method.py
>>> compare_against_cvxpy(10, False)
Generating 10 test cases.
   cvxpy_result barrier_result  percentage_error  n_dom  m_ineq  p_eq
0     -5.881036      -5.880759          0.004711      4       1     1
1    -18.002843     -18.002817          0.000148      6       7     1
2      0.213485     0.21378164          0.139013      6       3     3
3      0.271904     0.27196267          0.021494      5       5     2
4   -375.237190     -375.24002          0.000756      3       4     0
5     -2.476908      -2.476316          0.023881      3       3     0
6     -0.200263    -0.20022428          0.019242      3       5     0
7     -0.851441     -0.8514215          0.002338      3       4     1
8     10.516634      10.516672          0.000363      3       4     0
9   -118.436623     -118.43613          0.000419      5       2     0

On average = 0.021236485423195884%
```

Preliminaries for Grid-Optimization

Line numbers are given for `tetris.py`. Tables are printed at the end of each section.

We introduce the n-dimensional setting. Let $(N_{\underline{n}}) = (N_1, \dots, N_n)$ be a list of *counting numbers* (meaning $N_i \in \mathbb{N}^+$), we define our space $X = \prod[0,\ N_i]$ which is the union of $\overline{N} = \prod N_i$ cubes. If $m \in (\mathbb{Z})^n$, the *m-cube* is the set $\mathfrak{C}_m = \bigcap_1^n \{x \in \mathbb{R}^n,\ m_i \le x_i \le m_i + 1\}$. The set of all indices m where $\mathfrak{C}_m \subseteq X$ will be denoted by $\mathbb{N}_X$. Each m-cubes is equal to the intersection of $2n$ affine hyperplanes, an element $x \in X$ iff

$$f_{\text{ineq}}(x) = \begin{bmatrix} \text{id}_{\mathbb{R}^n} \\ -\text{id}_{\mathbb{R}^n} \end{bmatrix} x - \begin{bmatrix} m + \mathbb{1}_{\mathbb{R}^n} \\ -m \end{bmatrix} \lesssim 0, \tag{1}$$

where $\mathbb{1}_{\mathbb{R}^n} = (1, \dots, 1)$ and $\lesssim$ denotes an entrywise inequality. The first (resp. last) n rows in the equation above represent the upper (resp. lower) bound for the coordinates of x, and this is easily achieved using the tensor product, see Listing 2.

Listing 2: Realizing Hyperplanes

```
351         def cell_hyperplanes(m):
352             n = len(m)
353             id_Rn = np.eye(n)
354             A_ineq = np.kron( np.array([[1],[-1]]), id_Rn ) # first (resp. last)
                        n rows serve as the upper (resp. lower) bound for the
                       coordinate functions.
355             b_ineq = np.concatenate([m + np.ones_like(m), - m])
356             f_ineq = lambda x: A_ineq @ x - b_ineq
357             return f_ineq
```

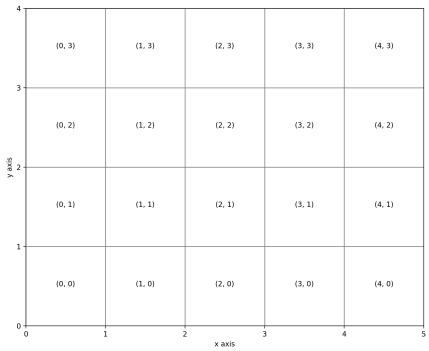

Figure 1: 5×4 Grid in cube coordinates.

Let $f_0 : \mathbb{R}^n \to \mathbb{R}$ be a smooth, convex objective function, and let X, m, $\mathbb{N}_X$ be define as above. Suppose we are given a disjoint union $X = \bigcup_{\text{finite}} \mathcal{A}_k$ where $\mathcal{A}_k = \bigcup_{\text{finite}} \mathfrak{C}_{m,k}$ and we wish to compute the *maximin* of the function $R_{\text{total}}(\{\mathcal{A}_k\}) = \sum_k \inf \mathcal{A}_k$.

For the two-dimensional case:

$$f_0(x, y) = (x-2)^2 + 3(y-1)^2 + e^{2x-3y}, \tag{2}$$

and $X = [0,5] \times [0,4]$ is divided into 20 cubes shown in fig. 1

Optimization over m-cubes

It is clear that $\inf \mathcal{A}_k = \min_m \inf \mathfrak{C}_{m,k}$ at every k. For each of the cubes we can run our unconstrained optimization algorithm, with f_{ineq} as in Equation (1). For f_0 as in eq. (2) we have **??**. For each $\mathfrak{C}_m$, the algorithm begins at the center point s and terminates at some point $x^* \in \mathfrak{C}_m$, the *minimizing value* is the number $f_0(x^*)$.[1]

Listing 3: Minimizing f_0 over X

```
>>> df
    starting_point                minimizer  minimizing_value
0 0     [0.5, 0.5]   [0.9999851, 0.9999829]          1.367917
  1     [0.5, 1.5]   [0.9999875, 1.1260438]          1.299729
  2     [0.5, 2.5]    [0.9999904, 2.000003]          4.018353
  3     [0.5, 3.5]    [0.9999882, 3.000002]         13.000958
1 0     [1.5, 0.5]  [1.3125881, 0.99999094]          1.159964
  1     [1.5, 1.5]    [1.5103639, 1.244948]          0.909367
  2     [1.5, 2.5]    [1.891078, 2.0000033]          3.120726
  3     [1.5, 3.5]   [1.9921248, 3.0000012]         12.006709
2 0     [2.5, 0.5]  [2.0000033, 0.99999774]          2.718318
  1     [2.5, 1.5]   [2.0000117, 1.4042233]          1.298631
  2     [2.5, 2.5]   [2.0000687, 2.0000036]          3.135374
  3     [2.5, 3.5]   [2.0010076, 3.0000017]         12.006772
3 0     [3.5, 0.5]  [3.0000005, 0.99999964]         21.085575
  1     [3.5, 1.5]   [3.0000036, 1.8307542]          4.732004
  2     [3.5, 2.5]   [3.0000048, 2.0000062]          5.000038
  3     [3.5, 3.5]    [3.0000105, 3.000002]         13.049831
4 0     [4.5, 0.5]   [4.0000005, 0.9999998]        152.413380
  1     [4.5, 1.5]   [4.0000014, 1.9999985]         14.389109
  2     [4.5, 2.5]    [4.0000024, 2.338457]         12.051242
  3     [4.5, 3.5]    [4.0000043, 3.000002]         16.367920
```

A plot of of the minimizers is shown in Figure 2.

[1] Results of Listing 3 can be seen in python by inspecting the variable `df` after running `python3 -i tetris.py`.

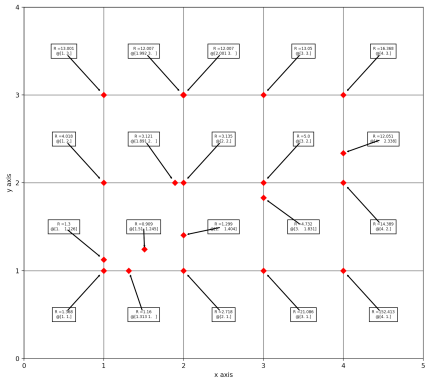

Figure 2: Optimizing over every cube. Red dots are the minimizers.

Tetrominos Schemes

A *scheme* is a disjoint union of X made of tetrominos. The method of converting an ASCII string to a scheme is too complicated to be described in this section (however see later sections for more). The user can define their own scheme to be used in the program by modifying the relevant lines shown in listing 4.[2]

Listing 4: User defined schemes. Tilings A, B are given. Tiling C is user defined.

```
Tiling_A_string = """
        a a b b c
        d a a b c
        d d e b c
        d e e e c"""
Tiling_A_colouring = ['red', 'yellow', 'blue', 'green', 'green']
Tiling_B_string = """
        a a b b b
        c a a b d
        c e e e d
        c c e d d """
Tiling_B_colouring = ['red', 'green', 'yellow', 'yellow', 'green']
# Custom Tilings
Tiling_C_string = """
        a b c d d
        a b c d d
        a b c e e
        a b c e e """
```

Computation of Rent

For every tetromino $\mathcal{T} = \{\mathfrak{C}_m\}$ in a fixed scheme $\mathcal{S}$, it is obvious that we can compute its *minimizer* and *minimizing value*. This is accomplished using the method `compute_rent_for_scheme()`, which for a given scheme $\mathcal{S}$ iterates over all such $\mathcal{T}$ and computes the *rent* of $\mathcal{S}$. Listing 5 contains the results of this when applied to Schemes A, B, C. , which also returns the rent of the individual schemes. We consider this computation to be 'elementary', as we can read off the minimizers for each tetromino for a given Scheme from fig. 2 or listing 5, and summing the minimizing values to obtain the rent. We see that **Scheme B produces a higher rent than Scheme A**. A comprehensive plot with all three schemes, and their minimizers is shown in fig. 3, one can generate this using the command `generate_plots()`.

[2]There is a prompt for user defined schemes using the method `prompted_custom_scheme()`. To use this, replace the `Scheme_C` variable by the result of the prompt. See Listing 6 as well. Alternatively, the user can modify the variable `Scheme_C` and rerun the program.

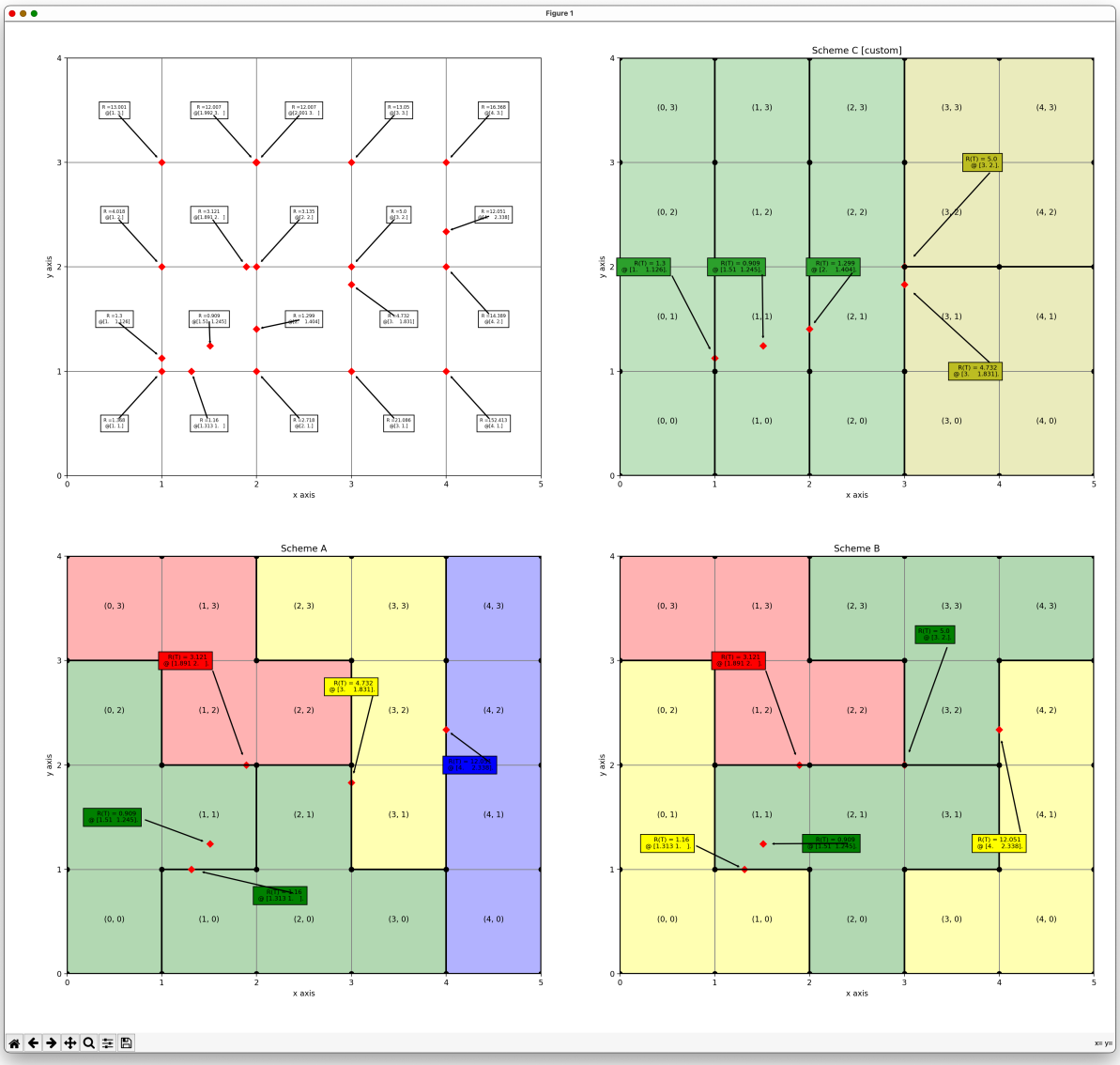

Figure 3: Optimizing over tetrominos for Schemes A, B, and C. Use `generate_plots()` to see this plot.

Listing 5: Results of optimization over all schemes.

```
>>> pprint.pprint(all_schemes)
[{'frame':                                cells  colour char is_valid   type                minimizer starting_point  minimizing_value
a  [[0, 3], [1, 3], [1, 2], [2, 2]]     red    a     True  S2_R1     [1.891078, 2.0000033]     [1.5, 2.5]           3.120726
b  [[2, 3], [3, 3], [3, 2], [3, 1]]  yellow    b     True  L2_R1    [3.0000036, 1.8307542]     [3.5, 1.5]           4.732004
c  [[4, 3], [4, 2], [4, 1], [4, 0]]    blue    c     True   I_R1     [4.0000024, 2.338457]     [4.5, 2.5]          12.051242
d  [[0, 2], [0, 1], [1, 1], [0, 0]]   green    d     True   T_R2     [1.5103639, 1.244948]     [1.5, 1.5]           0.909367
e  [[2, 1], [1, 0], [2, 0], [3, 0]]   green    e     True   T_R1   [1.3125881, 0.99999094]     [1.5, 0.5]           1.159964,
  'name': 'Scheme A',
  'rent': 21.97330317},
 {'frame':                                cells  colour char is_valid   type                minimizer starting_point  minimizing_value
a  [[0, 3], [1, 3], [1, 2], [2, 2]]     red    a     True  S2_R1     [1.891078, 2.0000033]     [1.5, 2.5]           3.120726
b  [[2, 3], [3, 3], [4, 3], [3, 2]]   green    b     True   T_R3    [3.0000048, 2.0000062]     [3.5, 2.5]           5.000038
c  [[0, 2], [0, 1], [0, 0], [1, 0]]  yellow    c     True  L2_R3   [1.3125881, 0.99999094]     [1.5, 0.5]           1.159964
d  [[4, 2], [4, 1], [3, 0], [4, 0]]  yellow    d     True  L1_R3     [4.0000024, 2.338457]     [4.5, 2.5]          12.051242
e  [[1, 1], [2, 1], [3, 1], [2, 0]]   green    e     True   T_R3     [1.5103639, 1.244948]     [1.5, 1.5]           0.909367,
  'name': 'Scheme B',
  'rent': 22.24133687},
 {'frame':                                cells     colour char is_valid    type                minimizer starting_point  minimizing_value
a  [[0, 3], [0, 2], [0, 1], [0, 0]]  tab:green    a     True    I_R1  [0.9999875, 1.1260438]     [0.5, 1.5]           1.299729
b  [[1, 3], [1, 2], [1, 1], [1, 0]]  tab:green    b     True    I_R1   [1.5103639, 1.244948]     [1.5, 1.5]           0.909367
c  [[2, 3], [2, 2], [2, 1], [2, 0]]  tab:green    c     True    I_R1  [2.0000117, 1.4042233]     [2.5, 1.5]           1.298631
d  [[3, 3], [4, 3], [3, 2], [4, 2]]  tab:olive    d     True  SQUARE  [3.0000048, 2.0000062]     [3.5, 2.5]           5.000038
e  [[3, 1], [4, 1], [3, 0], [4, 0]]  tab:olive    e     True  SQUARE  [3.0000036, 1.8307542]     [3.5, 1.5]           4.732004,
  'name': 'Scheme C [custom]',
  'rent': 13.23976867}]
```

Listing 6: Prompted user defined scheme

```
Frame_C = prompted_custom_scheme()
compute_rent_for_scheme(Frame_C)
all_schemes
```

High-dimension Generalizations

The program **tetris.py** generalizes easily to arbitrary n-dimensional domains. We can take the *tensor product* of 1-dimensional indexing sets to obtain some kind of lexicographical data structure.[3]

Example

Postponing further discussion regarding the implementation, we offer an example in $\mathbb{R}^3$ listing 7, where

$$f_0(x, y, z) = (x-1)^2 + (y-2)^2 + (z-3)^2 + 3y \quad \text{on} \quad X = [0,\ 3] \times [0,\ 3] \times [0,\ 3].$$

Listing 7: Optimizing f_0 in higher dimensions using a tensored data structure.

```
user@s-Laptop ecse507-final % python3 -i tetris.py
Dataframe detected. Loading results from /Users/user/ecse507-final/part2_optimal_points_tensored.csv.
Dataframe loaded.
>>> current_path =pathlib.Path('./').resolve()
>>> dpath_v3 = current_path/'a.csv'
>>> f0 = lambda x: (x[0]-1)**2 + (x[1]-2)**2 + (x[2]-3)**2 + 3*x[1]
>>> GRID_N = [3,3,3]
>>> optimize_over_each_cell(dpath_v3,f0,GRID_N)
NewtonTest unsatisfied after 100 iterations. Are we at the boundary?
NewtonTest unsatisfied after 100 iterations. Are we at the boundary?
NewtonTest unsatisfied after 100 iterations. Are we at the boundary?
NewtonTest unsatisfied after 100 iterations. Are we at the boundary?
NewtonTest unsatisfied after 100 iterations. Are we at the boundary?
NewtonTest unsatisfied after 100 iterations. Are we at the boundary?
NewtonTest unsatisfied after 100 iterations. Are we at the boundary?
NewtonTest unsatisfied after 100 iterations. Are we at the boundary?
NewtonTest unsatisfied after 100 iterations. Are we at the boundary?
NewtonTest unsatisfied after 100 iterations. Are we at the boundary?
NewtonTest unsatisfied after 100 iterations. Are we at the boundary?
NewtonTest unsatisfied after 100 iterations. Are we at the boundary?
Finished optimizing over each cell, saving to /Users/user/ecse507-final/a.csv
        starting_point                            minimizer minimizing_value
0 0 0  [0.5, 0.5, 0.5]          [0.996863, 0.5, 0.99999505]        7.7500296
    1  [0.5, 0.5, 1.5]        [0.99693114, 0.5, 1.9999902]         4.750029
    2  [0.5, 0.5, 2.5]          [0.9968636, 0.5, 2.9968636]        3.7500196
  1 0  [0.5, 1.5, 0.5]   [0.9969312, 1.0000174, 0.9999951]         8.000046
    1  [0.5, 1.5, 1.5]  [0.99686724, 1.0000198, 1.9999901]        5.0000496
    2  [0.5, 1.5, 2.5]   [0.99693114, 1.0000174, 2.996931]        4.0000362
  2 0  [0.5, 2.5, 0.5]  [0.99693155, 2.0000052, 0.9999952]        10.000044
    1  [0.5, 2.5, 1.5]   [0.9969312, 2.0000064, 1.9999902]         7.000048
    2  [0.5, 2.5, 2.5]  [0.99693155, 2.0000057, 2.9969316]         6.000036
1 0 0  [1.5, 0.5, 0.5]          [1.003137, 0.5, 0.99999505]        7.7500296
    1  [1.5, 0.5, 1.5]          [1.0030689, 0.5, 1.9999902]         4.750029
    2  [1.5, 0.5, 2.5]          [1.0031364, 0.5, 2.9968636]        3.7500196
  1 0  [1.5, 1.5, 0.5]   [1.0030688, 1.0000174, 0.9999951]         8.000046
    1  [1.5, 1.5, 1.5]   [1.0031327, 1.0000198, 1.9999901]        5.0000496
    2  [1.5, 1.5, 2.5]    [1.0030689, 1.0000174, 2.996931]        4.0000362
  2 0  [1.5, 2.5, 0.5]   [1.0030684, 2.0000052, 0.9999952]        10.000044
    1  [1.5, 2.5, 1.5]   [1.0030688, 2.0000064, 1.9999902]         7.000048
    2  [1.5, 2.5, 2.5]   [1.0030684, 2.0000057, 2.9969316]         6.000036
2 0 0  [2.5, 0.5, 0.5]          [2.0000098, 0.5, 0.9999951]         8.750038
    1  [2.5, 0.5, 1.5]          [2.0000098, 0.5, 1.9999901]         5.750039
    2  [2.5, 0.5, 2.5]           [2.0000098, 0.5, 2.996931]         4.750029
  1 0  [2.5, 1.5, 0.5]     [2.000012, 1.0000234, 0.999994]         9.000071
    1  [2.5, 1.5, 1.5]   [2.0000098, 1.0000191, 1.9999905]        6.0000577
    2  [2.5, 1.5, 2.5]   [2.0000098, 1.0000198, 2.9968672]         5.000049
  2 0  [2.5, 2.5, 0.5]  [2.0000098, 2.0000064, 0.99999523]        11.000057
    1  [2.5, 2.5, 1.5]   [2.0000095, 2.0000062, 1.9999906]         8.000056
    2  [2.5, 2.5, 2.5]   [2.0000098, 2.0000064, 2.9969313]         7.000048
>>>
```

[3] The *lexicographical ordering* (or dictionary ordering) of basis vectors in $\otimes_1^N \mathbb{R}(J_i)$ is the correspondence between $\{1, \ldots, \prod_1^N J_i\}$ and $\left\{\otimes_1^N e_{\mathbb{R}(J_i)}^{\alpha_i},\ 1 \le \alpha_i \le J_i,\ 1 \le i \le N\right\}$ such that indices α_i corresponding to a larger i value (closer to N) take precedence during enumeration from 1 to $\prod_1^N J_i$. If $\alpha = (\alpha_1, \ldots, \alpha_N) \in \otimes_{i=1}^N \mathbb{N}^+(J_i)$,

$$\alpha \cong (\alpha_1 - 1)\prod_2^N J_i + (\alpha_2 - 1)\prod_3^N J_i + \cdots + (\alpha_{N-1} - 1)J_N + \alpha_N = \sum_{i=1}^N (\alpha_i - 1) \prod_{j=i+1}^N J_j$$

It is easy to generalize the algorithm for minimizing over each $\mathcal{A}_k$ (as in the notation of the beginning of this document) to higher dimensions. In particular, we have included an example in the function `higher_dimensional_example()`.[4] Keep in mind that dataframes are cached to minimize loading times. If the function f_0 or any other parameters are modified (other than the scheme), one should delete the cached `.csv` file, found in the same folder as the program.

```
>>> result, df_new = higher_dimensional_example()
Dataframe detected. Loading results from /Users/user/ecse507-final/high_dim.csv.
Dataframe loaded.
{'frame': [{'cells': [(2, 0, 0), (2, 1, 1), (0, 0, 1), (1, 0, 2)],
            'minimizer': array([1.0031364, 0.5      , 2.9968636]),
            'minimizing_value': 3.7500196},
           {'cells': [(1, 2, 2),
                      (2, 2, 1),
                      (2, 2, 0),
                      (1, 2, 0),
                      (2, 1, 0),
                      (0, 1, 2)],
            'minimizer': array([0.99693114, 1.0000174 , 2.996931  ]),
            'minimizing_value': 4.0000362},
           {'cells': [(0, 1, 0),
                      (0, 0, 0),
                      (1, 0, 1),
                      (0, 0, 2),
                      (0, 2, 1),
                      (2, 1, 2),
                      (2, 2, 2)],
            'minimizer': array([0.9968636, 0.5      , 2.9968636]),
            'minimizing_value': 3.7500196},
           {'cells': [(2, 0, 2),
                      (0, 1, 1),
                      (0, 2, 2),
                      (0, 2, 0),
                      (1, 1, 2),
                      (1, 1, 1),
                      (1, 0, 0),
                      (2, 0, 1),
                      (1, 2, 1),
                      (1, 1, 0)],
            'minimizer': array([1.0030689, 1.0000174, 2.996931 ]),
            'minimizing_value': 4.0000362}],
 'name': 'High Dimensional Example',
 'rent': 15.5001116}
```

ASCII Converter

In this section, we discuss some of the methods of converting an ASCII input to a 2D partition. First, we prompt the user for the number of rows and columns of the cell. The dimensions of the grid has to satisfy $\text{rows} \times \text{cols} \equiv 0 \mod 4$. It is also easy to write a function that extracts the positioning of the occurences of a specific character (i.e 'a') in a string like

```
a b c d d
a b c d d
a b c e e
a b c e e
```

by scanning each row, and perhaps by trimming whitespaces and tabs if necessary.

Since each letter a, b, c, d, e must occur exactly 4 times, and the cells spanned by each corresponding letter have to be connected, we must check if the user is even giving us a tetromino-union of X. This is also easy to verify as the L^1 norm is designed precisely for this purpose, one would use something like:

> For every $c \in \{c_1, c_2, c_3, c_4\} = \mathcal{T}$, is there some $c' \in \mathcal{T}$ where $|c - c'|_{L^1} = 1$?

[4] simply run `python3 -i tetris.py` then `higher_dimensional_example()`.

If the answer to the above is positive, then $\mathcal{T}$ is a tetromino.[5]

Generalization to Regions that are Disks

Suppose we are given a finite collection of closed disks $\{D_j = \overline{B}(x^j, r^j)\}_{j=1}^M$ in the grid $X = \prod_{j=1}^n [0,\ N_j]$ the necessary and sufficient conditions for admissibility of a configuration are

$$\begin{cases} |x^j - x^k| > r_j + r_k & \forall 1 \le j,k \le M \\ \min\left(|x_i^j|,\ |x_i^j - N_i|\right) > r_j & \forall 1 \le j \le M,\ 1 \le i \le n\,. \\ x_i^j \in \{1,\dots,N_i\} & \forall 1 \le j \le M,\ 1 \le i \le n \end{cases} \tag{3}$$

Suppose the conditions in eq. (3) are met, and f_0 is smooth and convex on X. We can use the same algorithm to maximize R_{total} over the the union of the balls. Denote the minimizer, minimum pairs of each ball by $\mathcal{M}$, where

$$\mathcal{M} = \left\{\left(y^j,\ f_0(y^j)\right),\ y^j = \operatorname{argmin}_{z\in D_j} f_0(z)\right\}. \tag{4}$$

It is clear that $R_{\text{total}} = \max \sum_{j=1}^M \inf_{|z-x^j|\le r^j} f_0(z) = \sum \mathcal{M}$. To compute $(y^j,\ f_0(y^j))$, we can define each closed disk D_j using the (closed) sublevel set of the distance function

$$D_j = \operatorname{dist}_{x^j}^{-1}([0, r_0]) \quad \text{where} \quad \operatorname{dist}_x(y) = |x - y|$$

is continuous almost by definition of the standard topology on $\mathbb{R}^n$.

Modification Ideas for Annuli instead of Disks

In this final section of the report, we consider the case where $X' = \bigcup_{j=1}^M A_j$, where $A_j = \{z \in \mathbb{R}^n,\ s_j \le |z - x^j| \le r_j\}$. We assume that similar conditions to eq. (3) and the configuration that we are concerned with is admissible.

Suppose we can minimize f_0 over every annuli, and obtain $\mathcal{M}$ as in eq. (4) but with A_j instead of D_j, then

$$R_{\text{total}}(\{A_j\}) = \sum_j f_0(y_j).$$

So that for the remainder of this section, we will only concern ourselves with optimizing f_0 over any admissible annuli $A = A_j$. We write $A = \{z \in \mathbb{R}^n,\ s \le |z - x_0| \le r\}$, $B = \{z \in \mathbb{R}^n,\ |z - x_0| \le r\}$, let $p_0 = \operatorname{argmin}_{z\in A} f_0(z)$ and $p_1 = \operatorname{argmin}_{z\in B} f_0(z)$. We can run our algorithm on the full-disk B, this gives us an approximation of p_1. If $p_1 \in A$, then there is nothing to do, as $p_0 = p_1$ (as measured by f_0 or *modulo fibers of* f_0). It is the case of $p_1 \notin A$ that is of interest to us. For this, we need a geometric lemma.

[5] in practice, we also have to check that $c_i \in X$ and are lattice points.

Lemma 1.1: Where is the minimizer?

Let f_0 be a non-constant, smooth, convex function on X, and A, B, p_0, p_1 be as above. If $p_1 \notin A$, then $p_0 \in \partial A \setminus \partial B = \{z \in \mathbb{R}^n,\ |z - x_0| = s\}$.

Proof. Suppose for contradiction that $|p_0 - x_0| = s + \varepsilon$, the line segment starting at p_0 in the direction of p_1 can be made so small that it lies entirely within A. Let $t > 0$ where $p_t = p_0 + t(p_1 - p_0)$ such that $s \le |p_t - x_0| \le s + \varepsilon$. By the convexity of f_0, we have our desired contradiction:

$$f_0(p_t) \le f_0(p_0) + t(f_0(p_1) - f_0(p_0)) < f_0(p_0).$$

■

Let $C = \{z \in \mathbb{R}^n,\ |z - x_0| = s\}$ be the inner-boundary of A, I do not know of an easy way for us to find p_0, so I will propose a few methods.

Vanishing Tangential Gradient

We can view C' as a the boundary of a positively oriented (? I forgot the word) submanifold, so that C' is equipped with a unit-tangential vector field (? word), we can produce a smooth function $g = \langle \operatorname{grad} f_0, \mathcal{T}_{C'} \rangle_{T\mathbb{R}^n} \in C^\infty(C', \mathbb{R})$ and solve for $g = 0$. [6] The benefits of this is that we can use some intermediate value theorem argument when g changes sign.

Let me throw something random out there... By precomposing f_0 by a diffeomorphism (and destroying all of its convex structure), we can assume that $x_0 = 0$ and $s = 1$. To find a minimizing solution to $f_0|_C$, we write $f_0(e^{i2\pi\theta})$ for all $\theta \in [0, 1]$. Suppose that f_0 changes sign between $[\theta_0,\ \theta_1]$, then Rolle's Theorem will guarantee a critical point (along the submanifold).

Approximate C' using a sequence of convex submanifolds

We can probably, approximate the non-convex set by patching together convex bodies and do a search that way. Not sure how effective this is.

Change the objective function

We can bend the objective function f_0 such that it discourages going inside $E = \{z \in \mathbb{R}^n,\ |z - x_0| \le s\}$ in the first place. Not sure how possible this is to accomplish without destroying the convexity property. A simple appeal of adding some $g_0 = 2^{-1}\rho(z)|z - x_0|^2$ where $\rho(z)$ is some smooth mollifying function (that is possibly compactly supported) will not work because $h_0 = f_0 + g_0$ may fail to be convex (draw a picture and you will see that the line segment from p_0 to any point $p_2 \in \{z \in \mathbb{R}^n,\ |z - x_0| = r\}$ punctures the epigraph.

We can also try adding two quadratics in the form of $(x - q_0)^2 + (x - q_1)^2$, where $q_0 \in \partial B$ and $q_1 \in C'$, perhaps we can also write:

$$q_0(\theta) = re^{i2\pi\theta} \quad \text{and} \quad q_1(\theta) = se^{i2\pi\theta}.$$

Try plotting this on desmos*in the one dimensional case, try* $g(x) = (x + 1)^2 + (x - a)^2$, where $a \in [-10,\ +10]$ and run the animation on desmos, seems promising!.

[6] sorry for notation

Finally, we can try optimizing for $g_0 = f_0 + \sum_{j=1}^{L} (sL)^{-1}(x - q_j)^2$ where $q_j = se^{i\theta_j}$ and $\theta_j = 2\pi jL^{-1}$ are equaly spaced points on the circle.

Noteworthy Things

We mention a few of the interesting things.

Tensor Products using `pandas.MultiIndex`

There is a built-in way of computing tensor products of indexing sets using

`pandas.MultiIndex.from_product(q_MultiIndex),`

where `q_MultiIndex = list(map( range, [3,4,5] ) )` is a list of intervals. This gives us the iterator shown in listing 8.

Listing 8: easy tensor products

```
>>> q_MultiIndex = list(map(range, [3,4,3]))
>>> q_MultiIndex = pd.MultiIndex.from_product(q_MultiIndex)
>>> q_MultiIndex = list(q_MultiIndex)
>>> pprint.pprint(q_MultiIndex)
[(0, 0, 0),
 (0, 0, 1),
 (0, 0, 2),
 (0, 1, 0),
 (0, 1, 1),
 (0, 1, 2),
 (0, 2, 0),
 (0, 2, 1),
 (0, 2, 2),
 (0, 3, 0),
 (0, 3, 1),
 (0, 3, 2),
 (1, 0, 0),
 (1, 0, 1),
 (1, 0, 2),
 (1, 1, 0),
 (1, 1, 1),
 (1, 1, 2),
 (1, 2, 0),
 (1, 2, 1),
 (1, 2, 2),
 (1, 3, 0),
 (1, 3, 1),
 (1, 3, 2),
 (2, 0, 0),
 (2, 0, 1),
 (2, 0, 2),
 (2, 1, 0),
 (2, 1, 1),
 (2, 1, 2),
 (2, 2, 0),
 (2, 2, 1),
 (2, 2, 2),
 (2, 3, 0),
 (2, 3, 1),
 (2, 3, 2)]
```

Borders of Tetrominos

For a cell in a tetromino $c \in \mathcal{T}$, an easy way to determine whether or not a border should be drawn is to check whether or not the adjacent cell $c' = c + e_k$ where e_k is a standard basis vector is occupied by a cell is in your own tetromino $\mathcal{T}$.

If $c' \notin \mathcal{T}$, then we can draw the 'border' (a $n-1$ submanifold with corners) centered at the midpoint $m = (c + c')2^{-1}$, which is just a L^∞ cube in $\mathbb{R}^{n-1}$ with radius 1.

Classification of Tetrominos

There is probably an easy way of classifying all of the tetrominos and their rotated versions, we went with the rote and brute-force way of writing out all possible rotations and reflections. Shown in Listing 9.

```
shapes = {
# Square Shape
'SQUARE': """
            a a
            a a
""",
# L1 Shape
'L1_R1': """
            a a
            a b
            a b
""",
'L1_R2': """
            a a a
            b b a
""",
'L1_R3': """
            b a
            b a
            a a
""",
'L1_R4': """
            a b b
            a a a
""",
# L2 Shape
'L2_R1': """
            a a
            b a
            b a
""",
'L2_R2': """
            b b a
            a a a
""",
'L2_R3': """
            a b
            a b
            a a
""",
'L2_R4': """
            a a a
            a b b
""",
# T Shape
'T_R1': """
            b a b
            a a a
""",
'T_R2': """
            a b
            a a
            a b
""",
'T_R3': """
            a a a
            b a b
""",
'T_R4': """
            b a
            a a
            b a
""",
# S1 Shape
'S1_R1': """
            b a a
            a a b
""",
'S1_R2': """
            a b
            a a
            b a
""",
# S2 Shape
'S2_R1': """
            a a b
            b a a
""",
'S2_R2': """
            b a
            a a
            a b
""",
# I Shape
'I_R1': """
            a
            a
            a
            a
""",
'I_R2': """
            a a a a
"""
}
```

Listing 9: Classification = Huge Pain