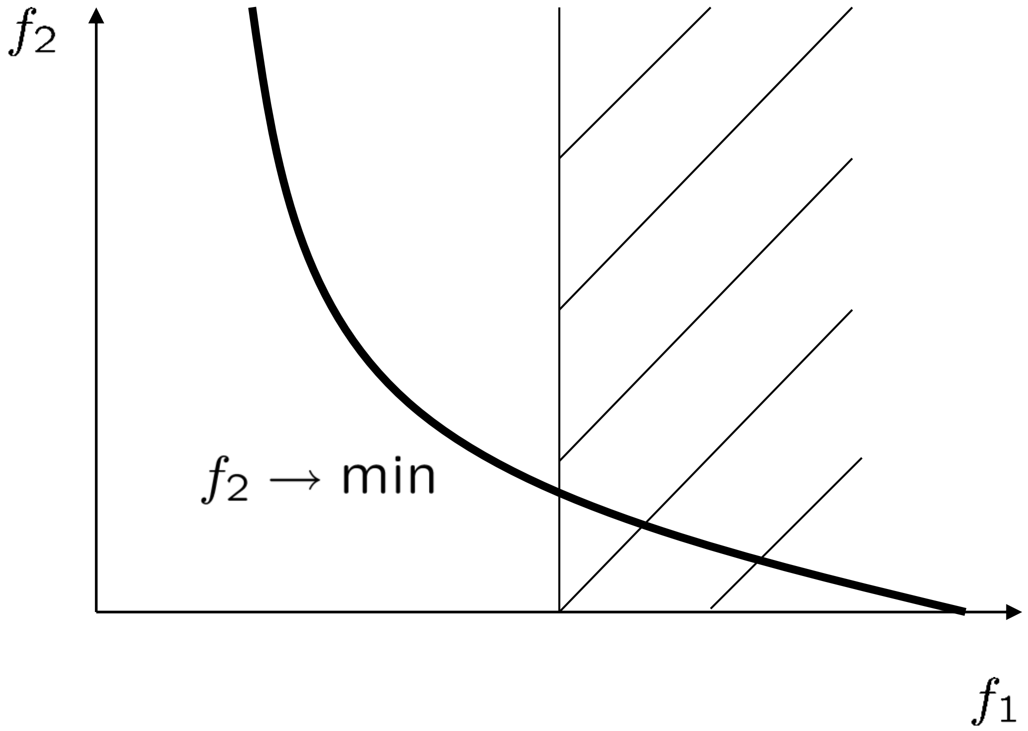
Exercise: Hill-climbing Methods (Local Optimization)

- Exercise (based on slides and python examples)
- Pareto optimization
- Adding objective function to Desdeo

Universiteit Leiden

ε-Constraint method

$$f_m(\mathbf{x}) \rightarrow \text{min}, \ \text{s.t.} f_i(\mathbf{x}) \leq \epsilon_i, i = 1, \ldots, m$$



With the dimension the number of ϵ combinations grows exponentially.

Basic strategy in Black-box optimization

Black-box Optimierungs-software

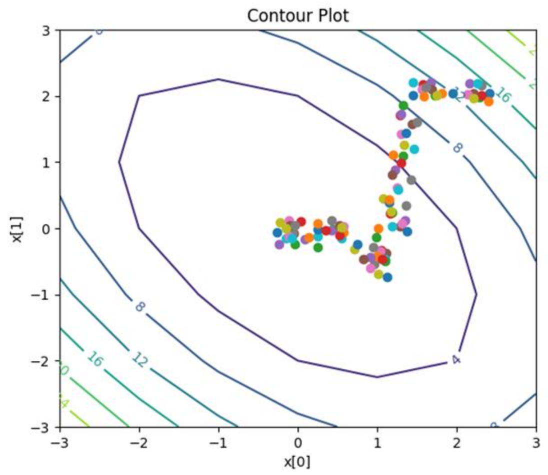
Input Parameters x(t)

$$\begin{tikzcd}\text{Black-box Optimierungs-software} \arrow[r, shift left, "\text{Input Parameters } x(t)"] & \text{Simulator/Evaluator} \arrow[l, shift left, "\text{Zielfunktionswerte, Restriktionsverletzungen}"]\end{tikzcd}$$

Simulator/Evaluator

Zielfunktionswerte,
Restriktionsverletzungen
f(x(t))+penalty(r(x(t))

1. Stochastic Hillclimbing
2. Gradient Descent
3. Newton Method
4. Simulated Annealing
5. Evolutionary Algorithm
6. Bayesian Optimization
7. Etc.



Universiteit Leiden

Hill-climbing Methods for Single-Objective Optimization

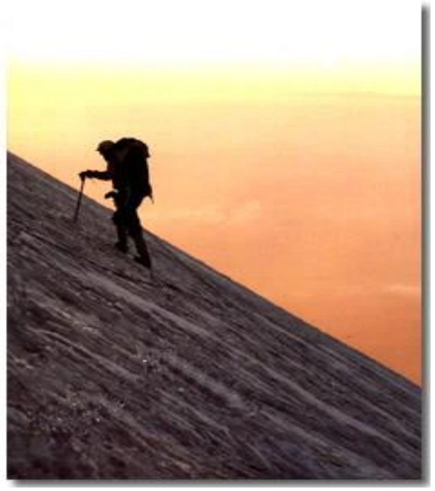
Path oriented (hill climbers) can be defined by a general iterative formula:

$$\mathbf{x_{t+1}} = \mathbf{x_t} + \sigma_\mathbf{t}\mathbf{d_t}$$

$\mathbf{x_t}$: Current search point

σ_t: Step size

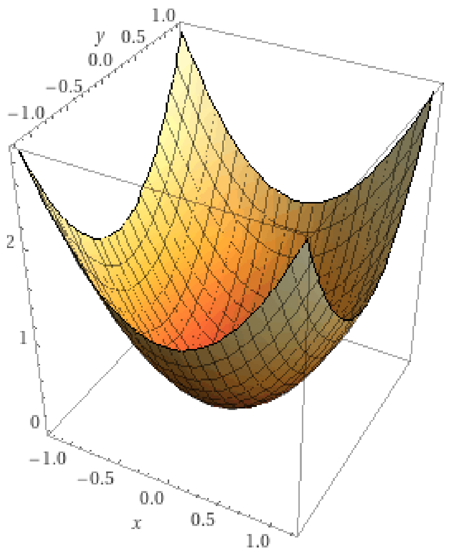
$\mathbf{d}_t$: Current search direction

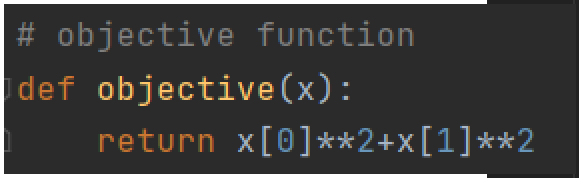


Hill-climbers generates a sequence of points $\{\mathbf{x}_t\}_{t=1,2,\ldots}$ that gradually improve the value of the objective function.

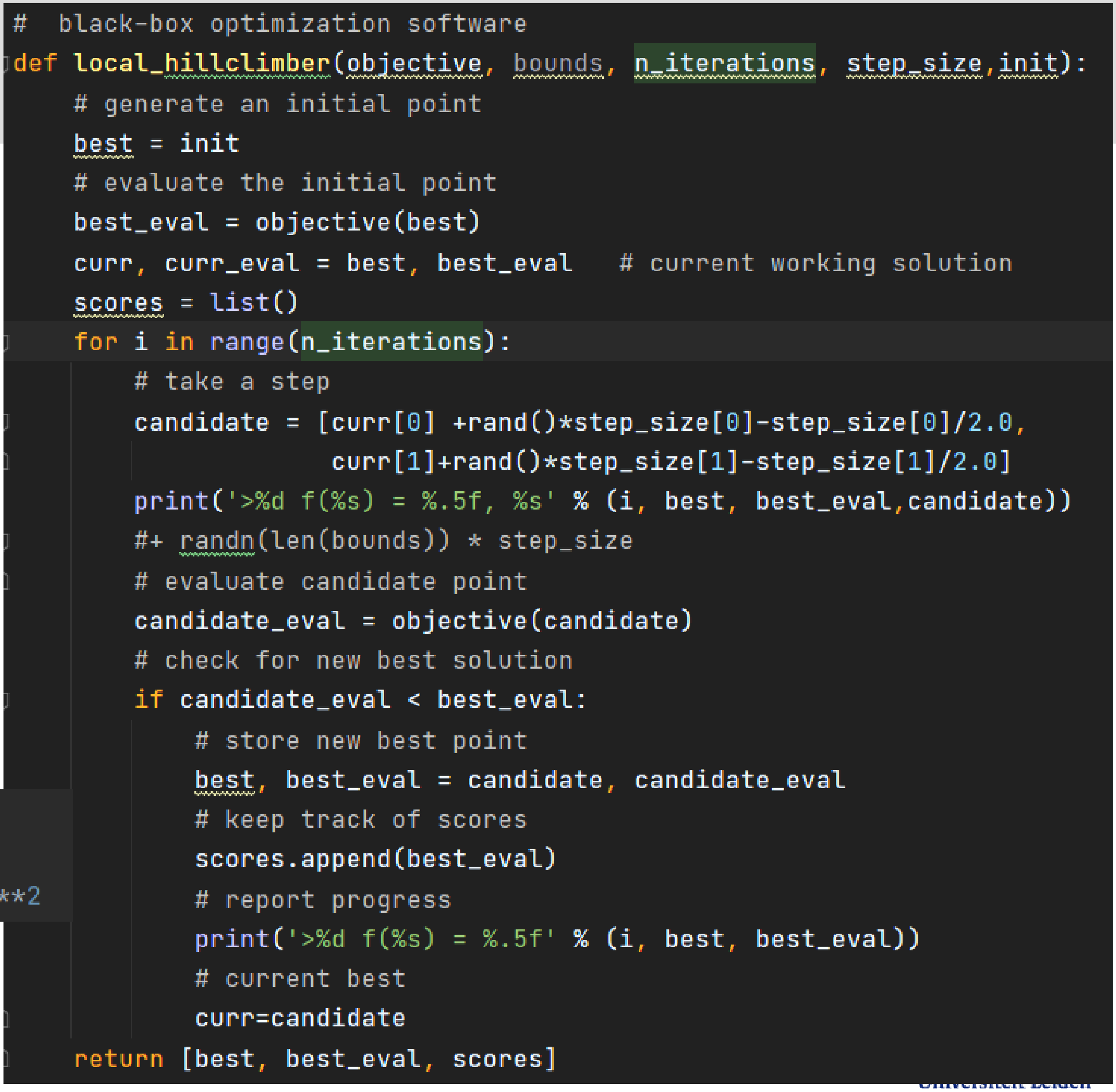
Universiteit Leiden

Simple 2-D stochastic hillclimber



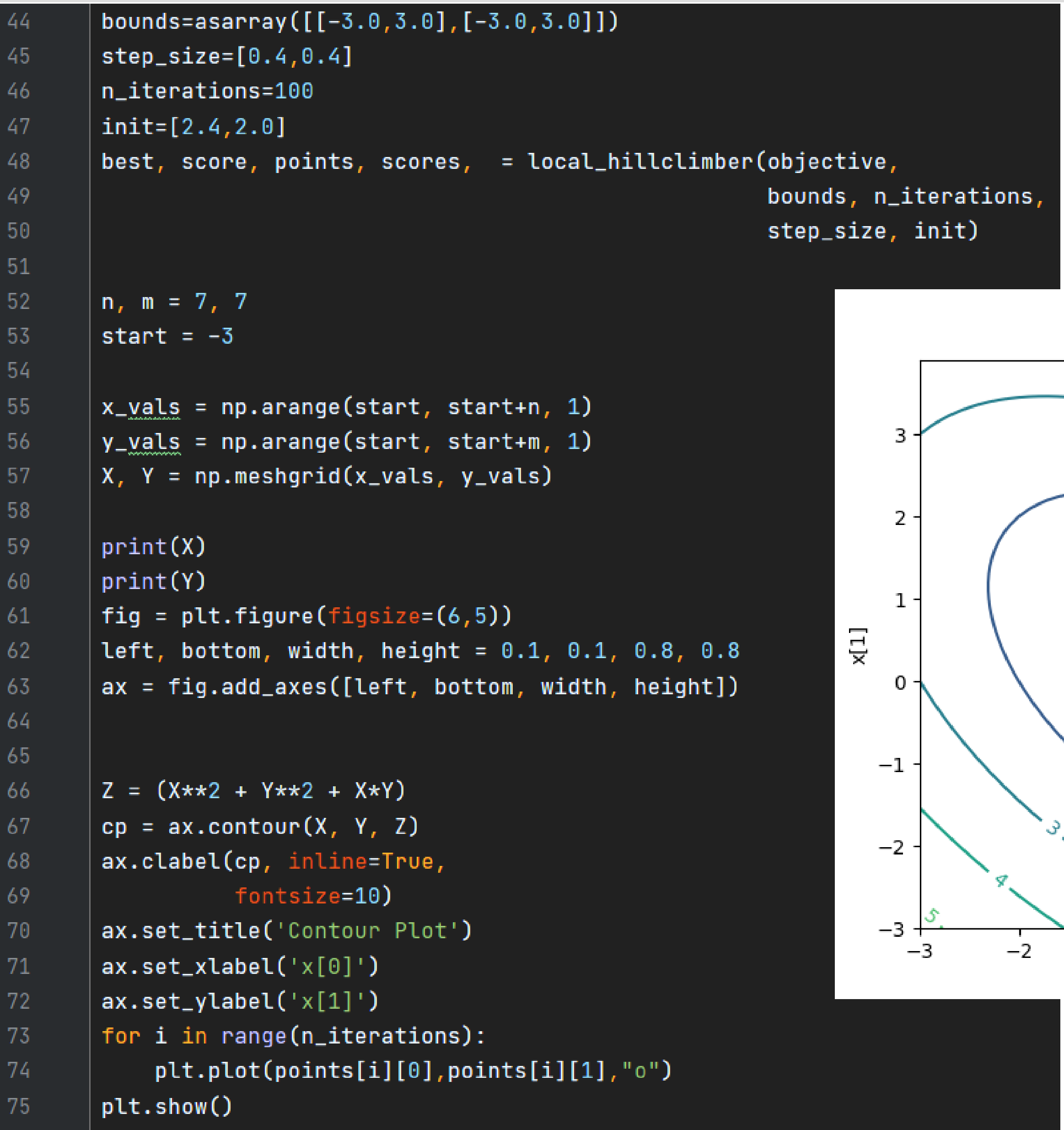


```
# objective function
def objective(x):
    return x[0]**2+x[1]**2
```

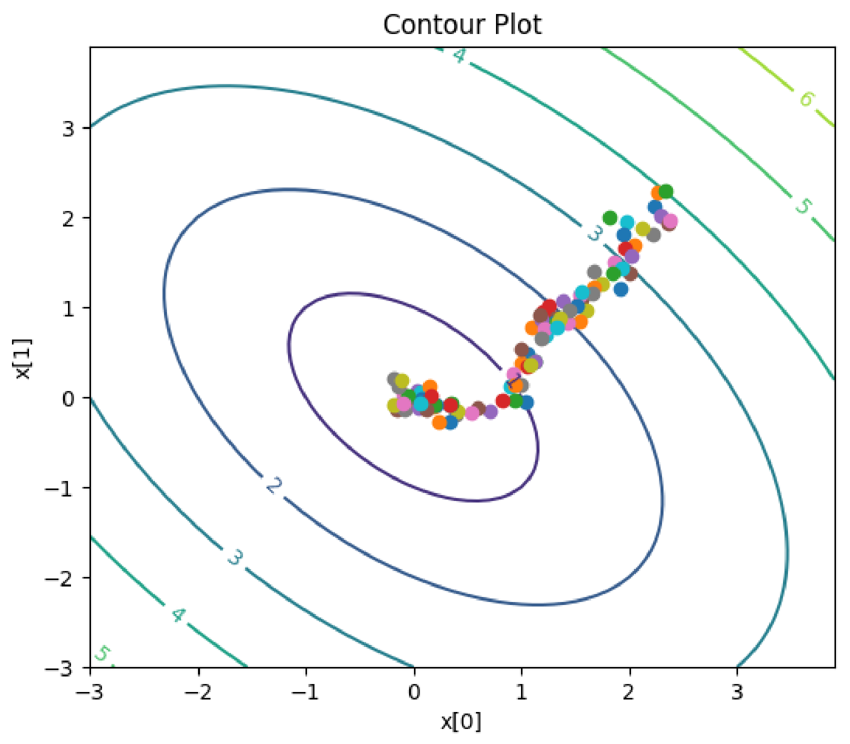


```
#  black-box optimization software
def local_hillclimber(objective, bounds, n_iterations, step_size,init):
    # generate an initial point
    best = init
    # evaluate the initial point
    best_eval = objective(best)
    curr, curr_eval = best, best_eval   # current working solution
    scores = list()
    for i in range(n_iterations):
        # take a step
        candidate = [curr[0] +rand()*step_size[0]-step_size[0]/2.0,
                     curr[1]+rand()*step_size[1]-step_size[1]/2.0]
        print('>%d f(%s) = %.5f, %s' % (i, best, best_eval,candidate))
        #+ randn(len(bounds)) * step_size
        # evaluate candidate point
        candidate_eval = objective(candidate)
        # check for new best solution
        if candidate_eval < best_eval:
            # store new best point
            best, best_eval = candidate, candidate_eval
            # keep track of scores
            scores.append(best_eval)
            # report progress
            print('>%d f(%s) = %.5f' % (i, best, best_eval))
            # current best
            curr=candidate
    return [best, best_eval, scores]
```

Plotting the history



```python
bounds=asarray([[-3.0,3.0],[-3.0,3.0]])
step_size=[0.4,0.4]
n_iterations=100
init=[2.4,2.0]
best, score, points, scores,  = local_hillclimber(objective,
                                                  bounds, n_iterations,
                                                  step_size, init)

n, m = 7, 7
start = -3

x_vals = np.arange(start, start+n, 1)
y_vals = np.arange(start, start+m, 1)
X, Y = np.meshgrid(x_vals, y_vals)

print(X)
print(Y)
fig = plt.figure(figsize=(6,5))
left, bottom, width, height = 0.1, 0.1, 0.8, 0.8
ax = fig.add_axes([left, bottom, width, height])


Z = (X**2 + Y**2 + X*Y)
cp = ax.contour(X, Y, Z)
ax.clabel(cp, inline=True,
          fontsize=10)
ax.set_title('Contour Plot')
ax.set_xlabel('x[0]')
ax.set_ylabel('x[1]')
for i in range(n_iterations):
    plt.plot(points[i][0],points[i][1],"o")
plt.show()
```



Universiteit Leiden

Penalty method for constraints

- In single objective black box optimization we can implement constraints in two ways
- Box constraints:
  - When hillclimbing method leaves the search region the point is projected back to the search region
- Implicit constraints:
  - Constraints that require the black-box function to be evaluated can be handled by a **(metric) penalty value**.

Metric Penalty method

$$\min f(\mathbf{x})$$

s.t. $c_i(\mathbf{x}) \leq 0 \; \forall i \in I$

Replace objective $f(\boldsymbol{x})$ by $\Phi_\mathrm{k}(\boldsymbol{x})$:

$$\min \Phi_k(\mathbf{x}) = f(\mathbf{x}) + \sigma_k \sum_{i \in I} g(c_i(\mathbf{x}))$$

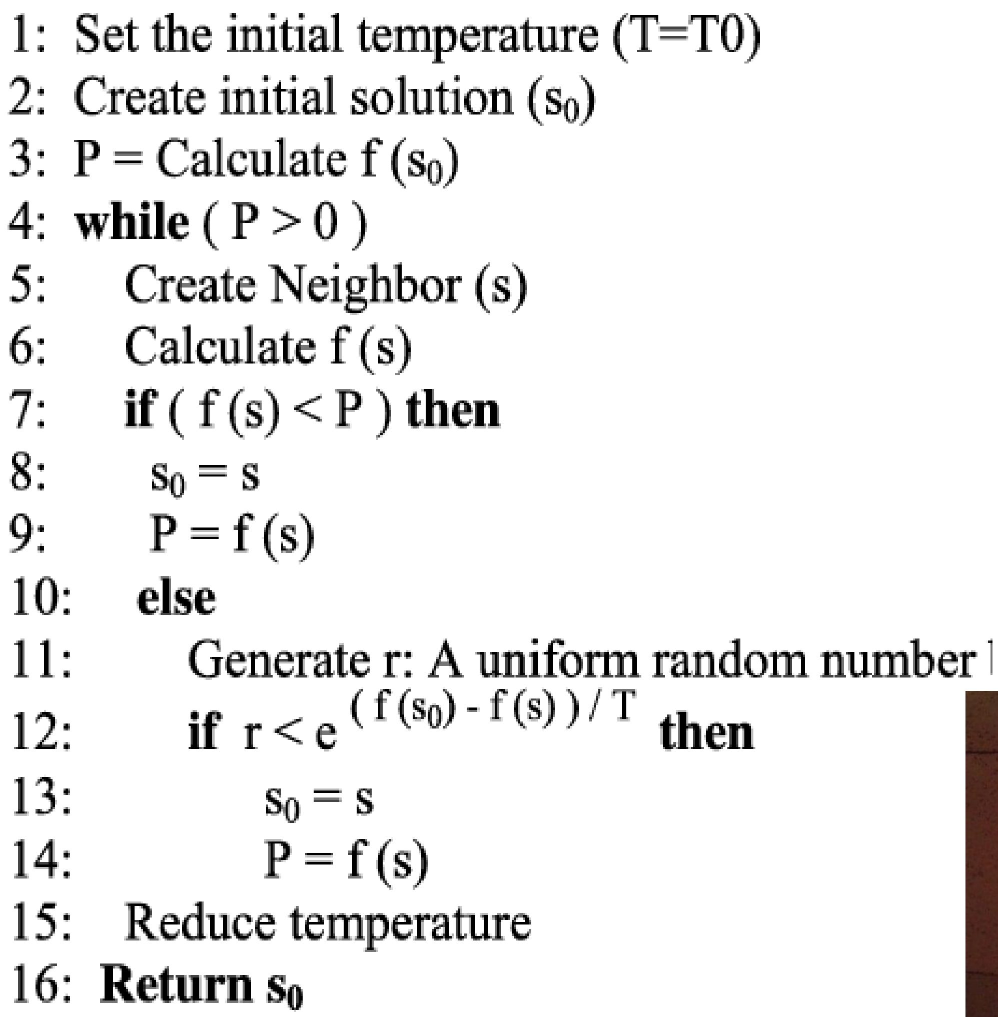
with: $g(c_i(\mathbf{x})) = \max(0, c_i(\mathbf{x}))^2$

σ_k can be constant or increasing over time (k).

Universiteit Leiden

Simulated Annealing

Kirkpatrick, S. Optimization by simulated annealing: Quantitative studies. *J Stat Phys* **34,** 975–986 (1984). https://doi.org/10.1007/BF01009452

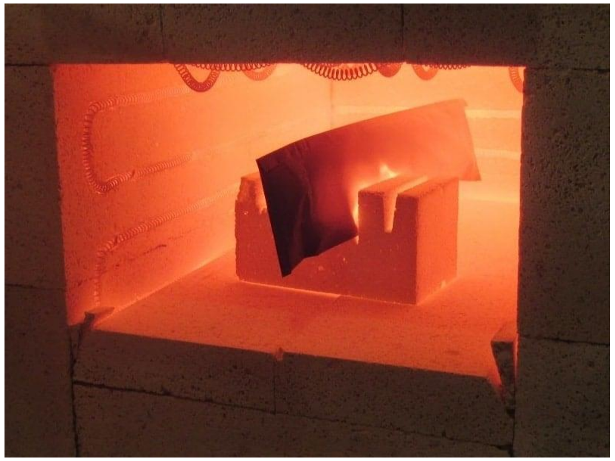


```
1:  Set the initial temperature (T=T0)
2:  Create initial solution ($s_0$)
3:  P = Calculate f ($s_0$)
4:  while ( P > 0 )
5:      Create Neighbor (s)
6:      Calculate f (s)
7:      if ( f (s) < P ) then
8:        $s_0 = s$
9:        $P = f(s)$
10:     else
11:         Generate r: A uniform random number
12:         if  $r < e^{(f(s_0) - f(s))/T}$  then
13:             $s_0 = s$
14:             $P = f(s)$
15:     Reduce temperature
16: Return $s_0$
```

Stochastic Hillclimbing inspired by Annealing process in crystals.

Simulated Annealing can always accepts improvements, but also worse solutions with some probability.

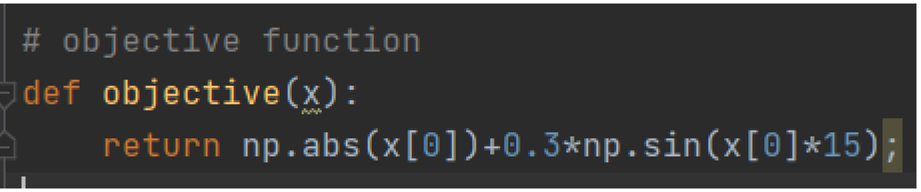
In order to get to global optima one might have to accept steps to get worse temporarily



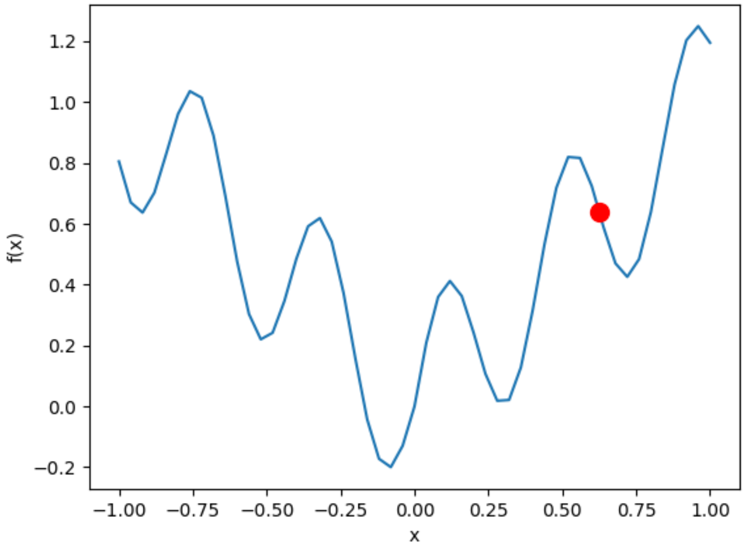
https://makeitfrommetal.com/beginners-guide-on-how-to-anneal-steel/

Universiteit Leiden

Simulated Annealing

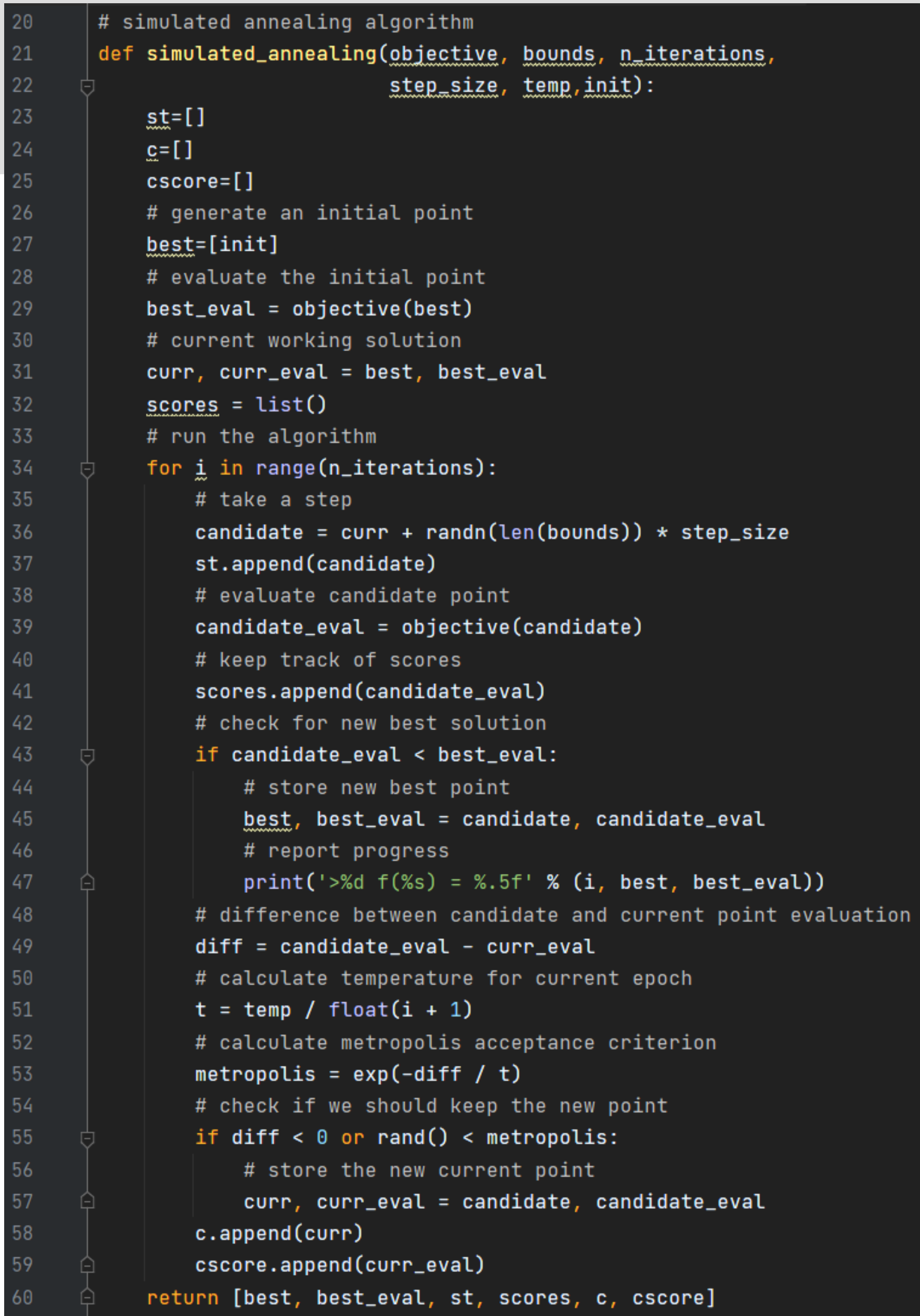


```python
# objective function
def objective(x):
    return np.abs(x[0])+0.3*np.sin(x[0]*15);
```

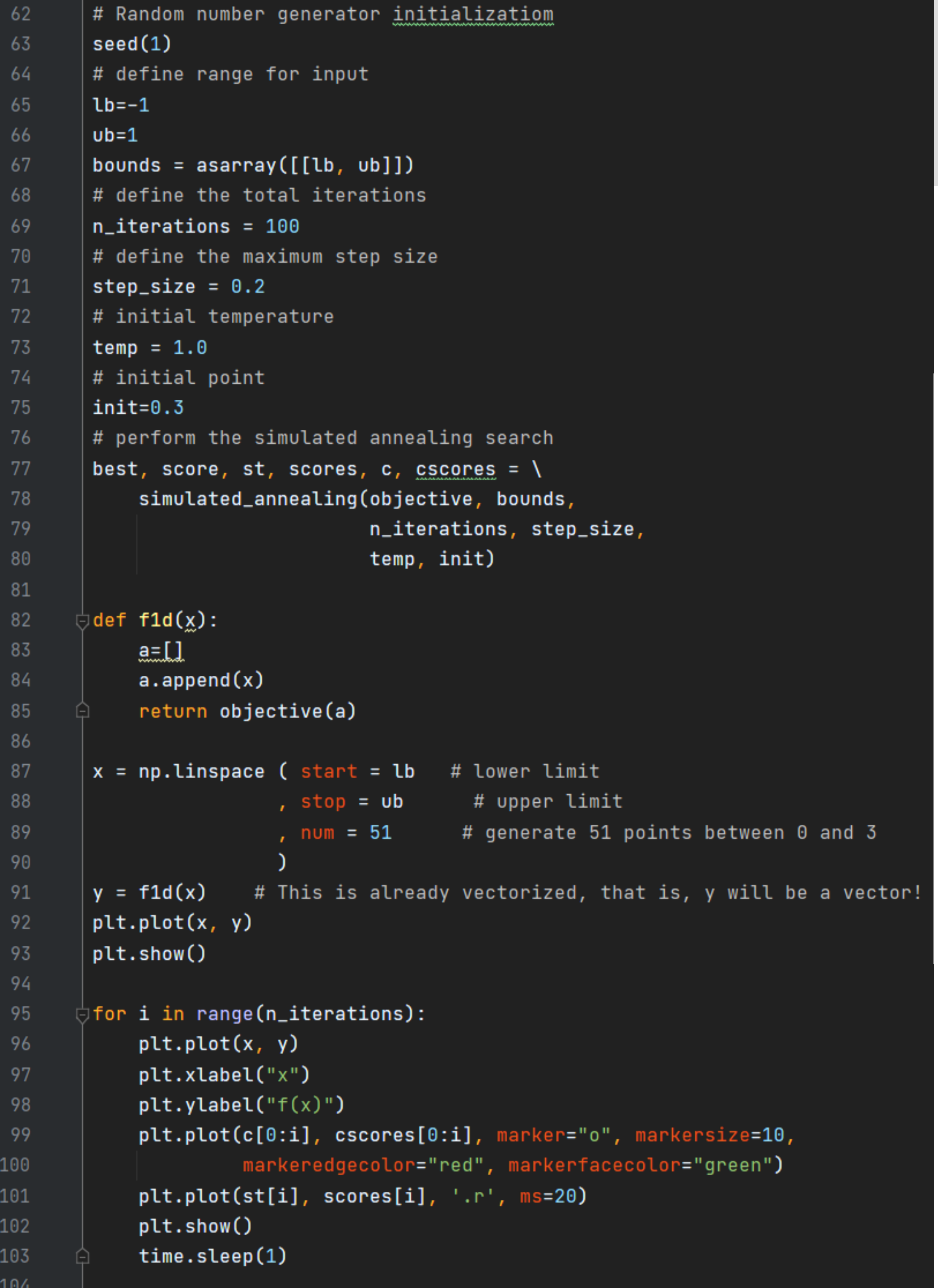


1-D Objective Function with local optima
$f(x) = |x| + 0.3\sin(15\,x), x \in [-1,1]$

Simulated Annealing can be implemented in 2-D and N-D (homework)



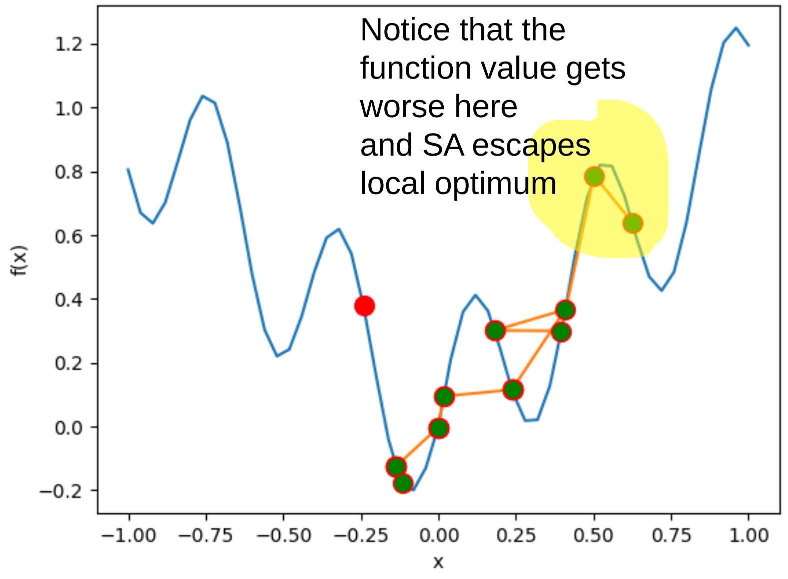
```python
# simulated annealing algorithm
def simulated_annealing(objective, bounds, n_iterations,
                        step_size, temp,init):
    st=[]
    c=[]
    cscore=[]
    # generate an initial point
    best=[init]
    # evaluate the initial point
    best_eval = objective(best)
    # current working solution
    curr, curr_eval = best, best_eval
    scores = list()
    # run the algorithm
    for i in range(n_iterations):
        # take a step
        candidate = curr + randn(len(bounds)) * step_size
        st.append(candidate)
        # evaluate candidate point
        candidate_eval = objective(candidate)
        # keep track of scores
        scores.append(candidate_eval)
        # check for new best solution
        if candidate_eval < best_eval:
            # store new best point
            best, best_eval = candidate, candidate_eval
            # report progress
            print('>%d f(%s) = %.5f' % (i, best, best_eval))
        # difference between candidate and current point evaluation
        diff = candidate_eval - curr_eval
        # calculate temperature for current epoch
        t = temp / float(i + 1)
        # calculate metropolis acceptance criterion
        metropolis = exp(-diff / t)
        # check if we should keep the new point
        if diff < 0 or rand() < metropolis:
            # store the new current point
            curr, curr_eval = candidate, candidate_eval
        c.append(curr)
        cscore.append(curr_eval)
    return [best, best_eval, st, scores, c, cscore]
```



```python
# Random number generator initializatiom
seed(1)
# define range for input
lb=-1
ub=1
bounds = asarray([[lb, ub]])
# define the total iterations
n_iterations = 100
# define the maximum step size
step_size = 0.2
# initial temperature
temp = 1.0
# initial point
init=0.3
# perform the simulated annealing search
best, score, st, scores, c, cscores = \
    simulated_annealing(objective, bounds,
                        n_iterations, step_size,
                        temp, init)

def f1d(x):
    a=[]
    a.append(x)
    return objective(a)

x = np.linspace ( start = lb   # lower limit
                , stop = ub      # upper limit
                , num = 51      # generate 51 points between 0 and 3
                )
y = f1d(x)    # This is already vectorized, that is, y will be a vector!
plt.plot(x, y)
plt.show()

for i in range(n_iterations):
    plt.plot(x, y)
    plt.xlabel("x")
    plt.ylabel("f(x)")
    plt.plot(c[0:i], cscores[0:i], marker="o", markersize=10,
             markeredgecolor="red", markerfacecolor="green")
    plt.plot(st[i], scores[i], '.r', ms=20)
    plt.show()
    time.sleep(1)
```

Simulated Annealing

- Plot shows the function and linked successful moves



- Homework:

Optimize design with N-D Simulated annealing

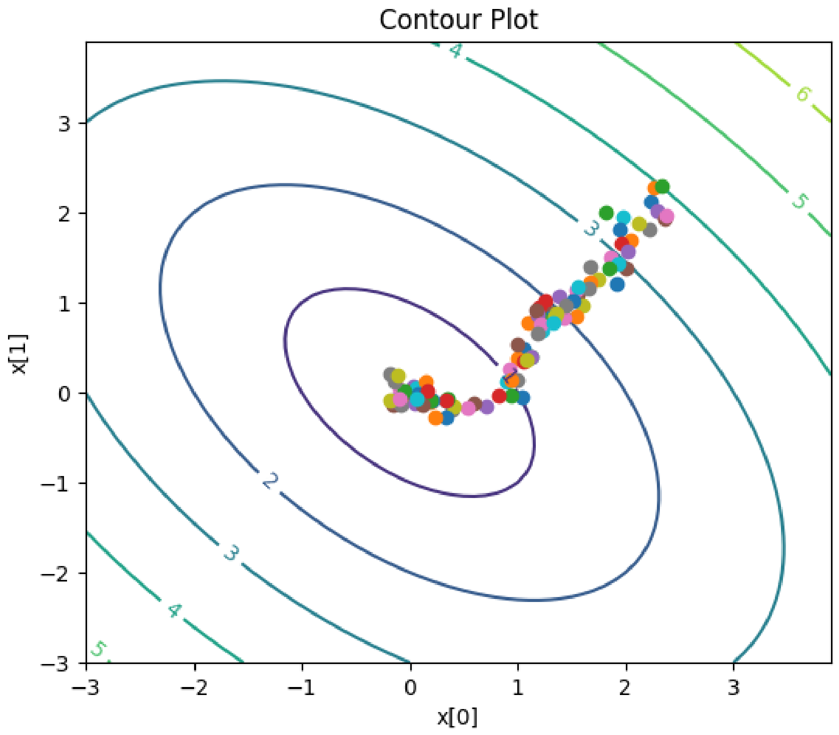
https://trinket.io/python3/b22300f21e

Universiteit Leiden

Task Local Optimization with Constraints/Penalties in Python

A

1. Modify simulated annealing example that it can be visualized in 2-D space
2. Implement a method that restricts the variables to the bounds (ranges) in the random steps
3. Implement a penalty method for the simulated annealing
4. Solve and visualize the two dimensional problem:
5. Visualize $|x| + 0.3\sin(15\, x)+$ $|y|+0.3\sin(15\, y)$ as a countour plot



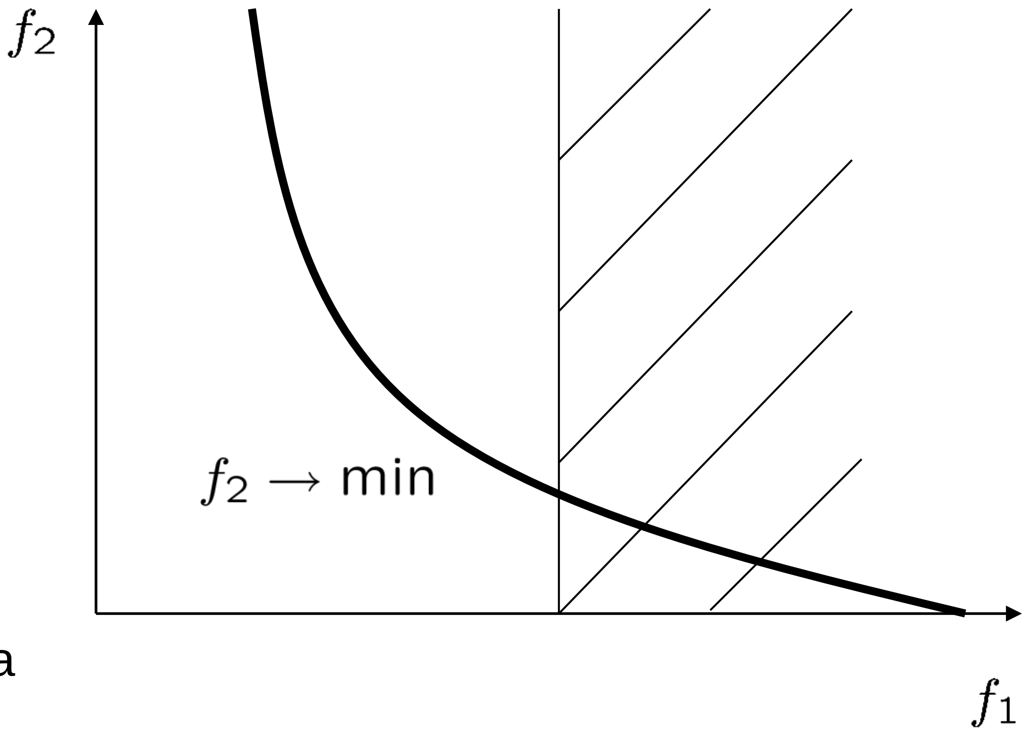
Save the source codes and add it to your homework folder.
You will be later asked to submit them to us.
You can name them after the number exday02A1

Universiteit Leiden

Exercise 2

B

1. Visualize the optimization problem of the optimal tin as a countour plot (or cone, if you prefer this problem)
2. Visualize the Area objective min)
3. Visualize the constraint $Volume(r,h) \geq L$, for level 330ml
4. Find the efficient points by solving a series of optimization problems where the volume is a constraint $Volume(r,h) \geq L$, for different levels (Pareto optimization, epsilon constraint method)


f_2
$f_2 \rightarrow \min$
f_1