Amplifier Optimization

Application of Simulated Annealing to Circuit Design

CONRAD JENSEN

Math 4630

Contents

1 Description

1.1 Solution to "Brute Force Engineering"

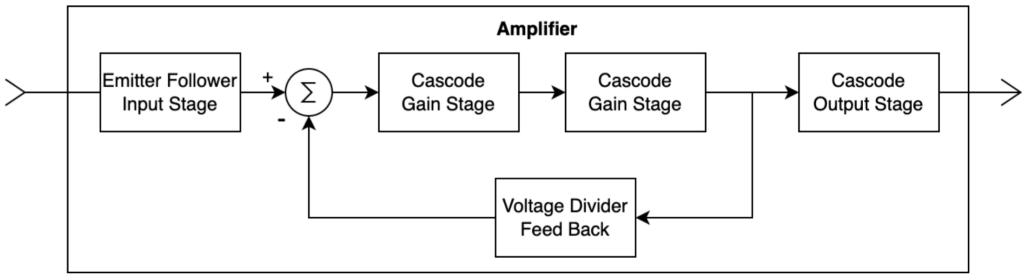

Figure 1: Top Level Design

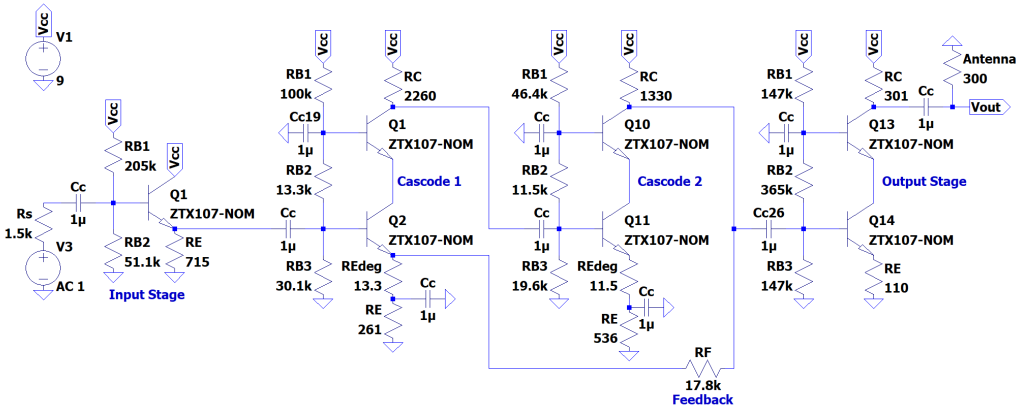

Figure 2: Optimized Amplifier Circuit

Electrical engineers have developed powerful design patterns that allow for confident decision making on a circuit's structure. Common design equations will produce a working, but in no way optimized, circuit. Optimizing a circuit by hand proves difficult due to the high dimensionality, unexpected interactions, and tedious analysis. In my experience this has lead to brute force trial and error instead of informed decision making.

Electronics II students are charged with designing an amplifier circuit to meet bandwidth and gain specifications within certain constraints. A program using simulated annealing is able to generate an design (Figure 2) that exceeds requirements where humans have failed. This program is written in Python and uses PySpice to simulate circuits in NgSpice.

1.2 Algorithm

Algorithm 1 Simulated Annealing [1]

```
1:  function SIMULATED-ANNEALING(T, s_0, kMax, σ)
2:      s ← s_0
3:      e ← −goodness(s)
4:      sbest ← s
5:      ebest ← e
6:      b ← T^{−2÷kMax}
7:      k ← 0
8:      while k < kMax do
9:          T ← temperature(T, b)
10:         snew ← neighbour(s, σ)
11:         enew ← −goodness(s)
12:         if P(e, enew, T) > random(0, 1) then
13:             s ← snew; e ← enew
14:         if enew < ebest then
15:             sbest ← snew; ebest ← enew
16:         k ← k + 1
17:     return sbest
```

Algorithm 1 is a common implementation of simulated annealing. In this case it takes four parameters: T, the initial temperature, s_0, the beginning state of the circuit, $kMax$, the number of iterations to run, and σ the standard deviation used to find neighbouring states. Energies are generally negative to conform with finding the lowest energy state. If $T = 0$ is input, simulated annealing behaves as a greedy random walk.

Algorithm 2 Annealing Schedule [1]

```
1:  function TEMPERATURE(T, b)
2:      return T · b
```

The annealing or cooling schedule determines how temperature decreases over time. In this case a geometric schedule is used.

Algorithm 3 Generate Random Neighbour

1: **function** NEIGHBOUR($s,\ \sigma$)
2: $\quad$ *snew* $\leftarrow s$
3: $\quad$ **for all** $snew_i \in snew$ **do**
4: $\quad\quad$ $snew_i \leftarrow snew_i + \text{normal}(\mu = 0,\ \sigma)$ discrete steps
5: $\quad$ **return** *snew*

Simulated annealing does not specify how to generate neighbouring states so the simplest approach is used. The state of the circuit is a set of resistances which take discrete values (see Figure 3). Each resistor is changed a discrete number of steps determined by a normal distribution centered at zero.

Algorithm 4 Probability of Accepting Higher Energy State [1]

1: **function** P($e,\ enew,\ T$)
2: $\quad$ **if** $enew < e$ **then**
3: $\quad\quad$ **return** 1
4: $\quad$ **return** $(1 + \exp(100(1 - enew \div e) \div T))^{-1}$

For a simulated annealing algorithm there are many approaches to calculating the probability of accepting a worse state. A normalized inverse exponential is used for its consistency across a range of energies. This method yields a 50% chance to accept an infinitesimally worse state with temperature moderating how the probability rolls off.

The objective function (see Algorithm 5) determines the goodness of an amplifier circuit state by finding the operational bandwidth while factoring in penalties for not meeting the design specifications (see Section 1.3.3).

1.3 Optimization Problem

1.3.1 Variables

Design Variables

Cascode 1	Cascode 2	Output Stage	Feedback
R_{B1}	R_{B1}		
R_{B2}	R_{B2}		
R_{B3}	R_{B3}		
R_C	R_C		
$R_{E,deg}$	$R_{E,deg}$		
R_E	R_E	R_E	
			R_F

The design variables above represent full freedom over all factors that affect gain and bandwidth without changing the nature of the amplifier's pattern. The R_B's and R_E set the bias point which affects quiescent current through the transistors which changes their transconductance. The gain of cascodes are strongly influenced by transconductance, R_C, and $R_{E,deg}$. R_F controls the negative feedback, which modulates the overall gain of the amplifier. The output stage's R_E also has a strong influence on the gain as it the final stage so a large overall increase can be achieved by a small local gain ($\sim 20\%$). See Figure 2 for each resistor's location in the amplifier circuit.

1.3.2 Objective Function

Algorithm 5 Bandwidth Scoring

function CALCULATE-GOODNESS(resistor values)
  for all extremes of transistor tolerance **do**
    map of ($freq : gain$), $current\ draw \leftarrow$ simulate circuit(resistor values, transistor)
    if $current\ draw > 12$ mA **then**
      $goodness \leftarrow 0$
      break
    else if $current\ draw > 10$ mA **then**
      $punish \leftarrow \max(0,\ 1 - (current\ draw\ - 10\text{mA})^2 \div 10\text{mA})$
    else
      $punish \leftarrow 1$
    $DC \leftarrow gain\left[freq = 0\right]$
    $bandwidth \leftarrow$ find first $freq$ where $gain\left[freq\right] \notin DC \pm 1.5\text{dB}$
    $badness \leftarrow$ mean-square-distance($gain\left[freq\right],\ 1000 \pm 1.5\text{dB}$) $\forall\ freq \in [0, 7.2\text{MHz}]$
    $goodness \leftarrow \min(goodness,\ punish \cdot bandwidth \div \max(badness,\ 1))$
  return goodness

1.3.3 Constraints

Design Variables

2% Standard Values (EIA E48) Decade multiples are available from 10 Ω through 22 MΩ											
10.0	10.5	11.0	11.5	12.1	12.7	13.3	14.0	14.7	15.4	16.2	16.9
17.8	18.7	19.6	20.5	21.5	22.6	23.7	24.9	26.1	27.4	28.7	30.1
31.6	33.2	34.8	36.5	38.3	40.2	42.2	44.2	46.4	48.7	51.1	53.6
56.2	59.0	61.9	64.9	68.1	71.5	75.0	78.7	82.5	86.6	90.9	95.3

Figure 3: Resistor are only commercially available in discrete values.

The original Electronics II project required the use of only commercially available 2% tolerance resistor values. To keep with the spirit of the project and add complexity, the same requirements are imposed on this optimization problem.

Objective Function

The design specifications for the Electronics II project stipulated a max current draw of 12mA and a 60 ± 1.5dB bandwidth of > 7.2MHz. To allow the program to start in bad state and work towards these requirements a penalty function is implemented for each constraint as seen in Algorithm 5. On line 8, current above 10mA is punished with a quadratic function. On line 13, the mean square distance of the gain from the desired gain is used to penalize the goodness. These functions allow simulated annealing to search over hills by providing nonzero goodness values to circuits outside specifications.

References

[1] Heikki Orsila, Erno Salminen, and Timo D. Hämäläinen. “Best Practices for Simulated Annealing in Multiprocessor Task Distribution Problems”. In: 2008.

1.4 Program Efficacy

1.4.1 Best Result Further Analysis

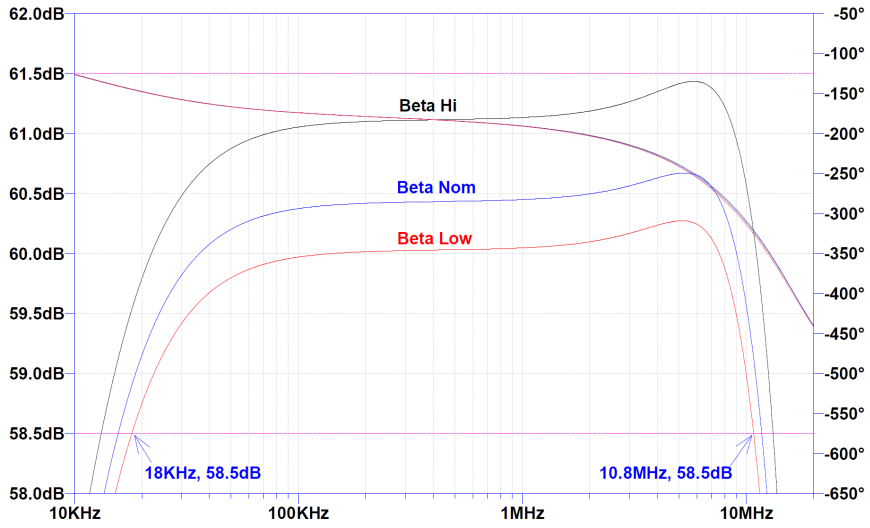

Figure 4: Bode plot of optimized circuit at different transistor β's including phase.

A more sophisticated analysis of the best result from the optimization program (not sample run) compared to my original Electronics II project submission shows how beneficial even simplistic optimization can be. I spent about equal time working on each result.

60 ± 1.5 dB bandwidth:

- Goal: 500 kHz to 7.2 MHz
- Greedy Random Walk: 18 kHz to **10.8 MHz**
- Human: 15.3 kHz to 6.0 MHz

Idle max power:

- Goal: < 108 mW
- Greedy Random Walk: 95 mW
- Human: 79 mW

1.4.2 Improvements

Simulated annealing is not well suited to this application as the search space is not fractal or particularly hilly. Design equations can likely place your starting state within a convex region where more sophisticated algorithms excel. However accessing the derivative information is difficult & the design variables are discrete so options are limited.

1.4.3 Iteration Speed

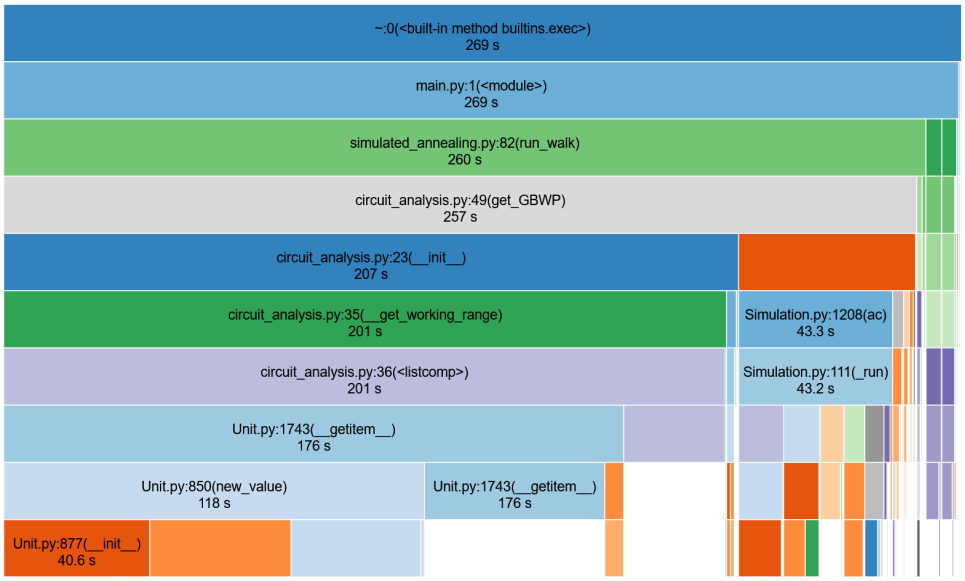

Figure 5: Run time profile of program. O(n^2) run time complexity in the getWorkingRange method proved extremely detrimental to performance.

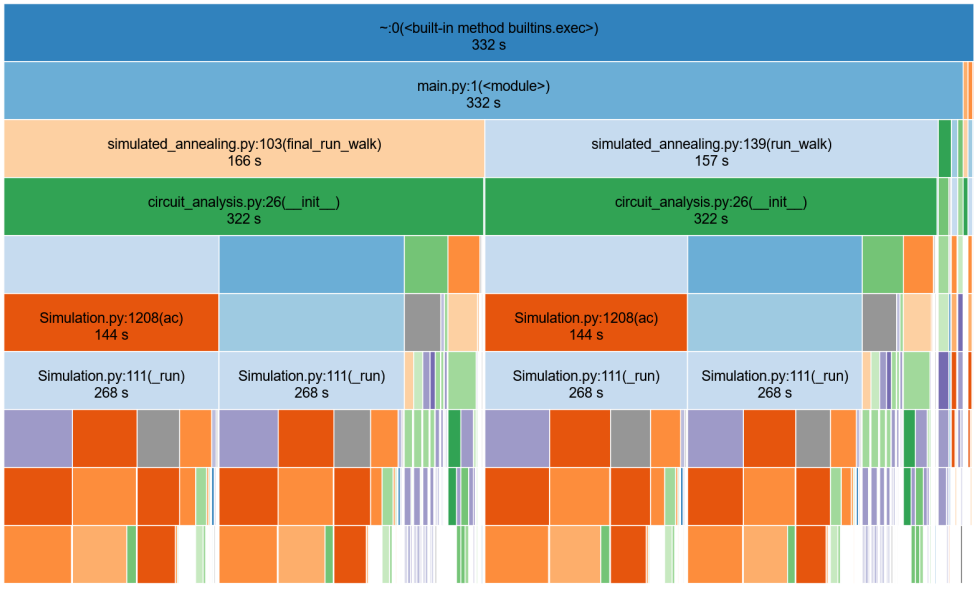

Figure 6: Simulation.py took 80% of the runtime. Since NgSpice is optimized, this result is acceptable. This results in ~60 simulations per second.

2 User Manual

2.1 Installation

Windows

1. Install Python 3.10
2. Download project from GitHub

   https://github.com/ComradeCon/NLO-Circuit-Optimization.git
3. (optional) Create a virtual environment
4. Open terminal, CD to project folder
5. Run

```
pip install -r requirements.txt
pyspice-post-installation --install-ngspice-dll
```

If these steps do not work please consult https://pyspice.fabrice-salvaire.fr/releases/v1.5/installation.html.

2.2 Interface

If installed correctly running main.py should prompt you in the terminal to enter the optimization parameters. The prompts are as follows:

1. **Default Behavior? ~5min (y/n)** starts optimizing with default parameters
2. **Verbose? (y/n)** prints run time progress updates
3. **Random Starting Point? (y/n)** generates a worse starting point
4. **(y) Check transistor tolerances or (n) assume nominal (y/n)** Simulates transistors with high & low β's instead of just nominal
5. **Neighbor Standard Deviation (recommend <1)** sets standard deviation for normal distribution that picks neighbouring states

Simulated Annealing

6. **Num steps per walk** number of iterations for each simulated annealing round/walk
7. **Num walks** number of times to run simulated annealing with the same starting conditions
8. **Starting Temperature (recommend <30)** higher temperatures allow worse states to be accepted more often

Greedy Random Walk

9. **Num steps** number of iterations for the greedy random walk

3 Results

3.1 Sample Run

Parameters

Neigbour σ: 0.2

SA Steps: 20,000, SA Walks: 1, Temperature: 30

GRW Steps: 20,000

Check Transistor Tolerance: True

Sample Run Results

Method	BW (MHz)	Gain (V/V)	Current (mA)	Run Time (s)
Goal	>7.20	~1000	<12	N/A
Start	0.917	880	9.6	N/A
Simulated Annealing	6.97	1150	11.0	808
Greedy Random Walk	7.65	1150	11.4	922

Cascode 1

Resistor	Start (Ω)	Best (Ω)
R_{B1}	100000	105000
R_{B2}	12100	10000
R_{B3}	21500	23700
R_C	3010	2050
$R_{E,deg}$	19.6	18.7
R_E	237	147

Cascode 2

Resistor	Start (Ω)	Best (Ω)
R_{B1}	59000	56200
R_{B2}	10500	12700
R_{B3}	18700	20500
R_C	1620	1620
$R_{E,deg}$	11.0	11.0
R_E	383	332

Feedback & Output Stage

Resistor	Start (Ω)	Best (Ω)
R_F	30100	31600
R_E	133	133

See Figure 2 for each resistor's location in the amplifier circuit.

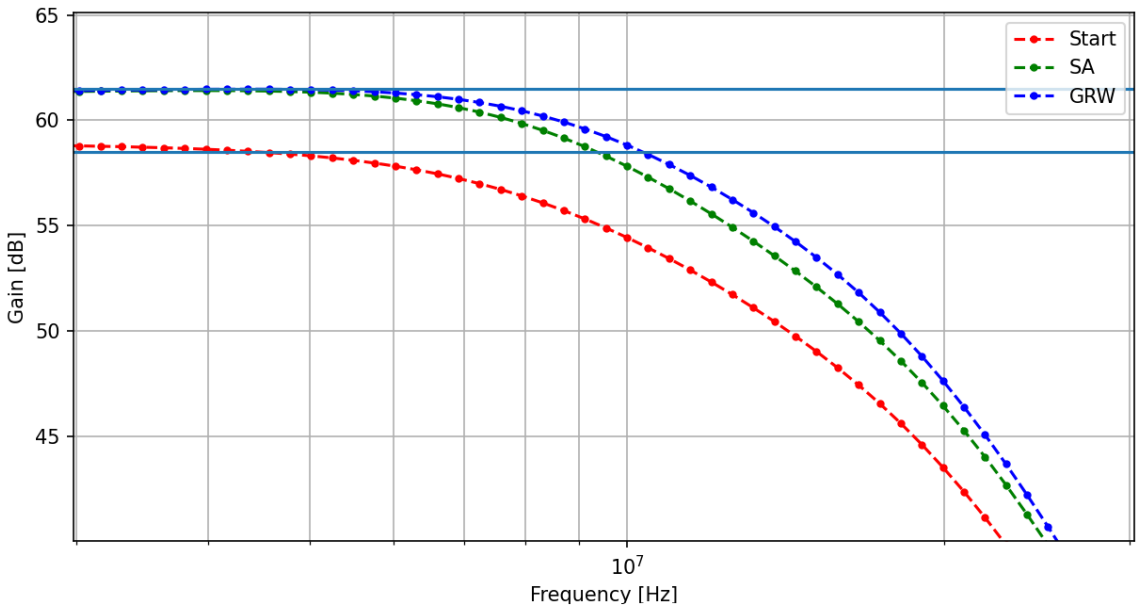

Figure 7: Example Bode plot comparing results of simulated annealing vs a greedy random walk. Two horizontal blue lines show the range of acceptable gain.

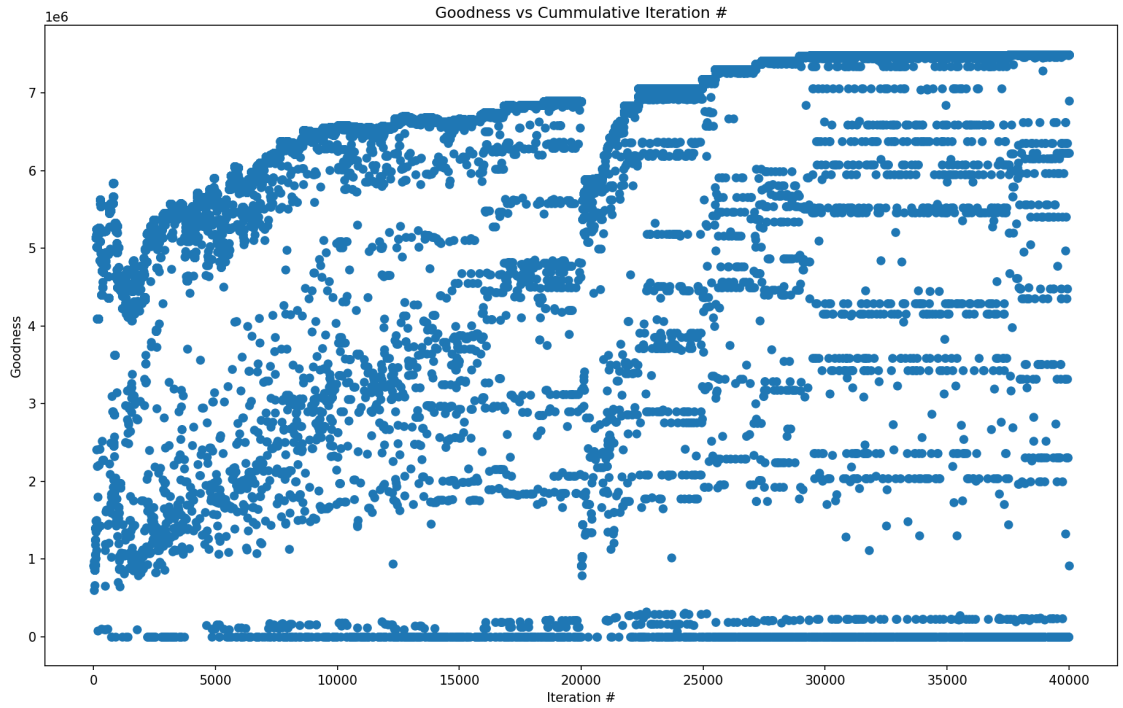

Figure 8: Scatter plot showing how each walk evolves over time. The first 20k iterations are one simulated annealing walk. The second 20k iterations are a greedy random walk.

4 Source Code

4.1 main.py

```
# PySpice imports

import PySpice.Logging.Logging as Logging
logger = Logging.setup_logging()
from PySpice.Spice.Netlist import Circuit, SubCircuitFactory
from PySpice.Unit import *

#####################################
# other libraries

from tqdm import tqdm
import csv
import atexit
import time

#####################################
# project files

from subcircuit_def import SubCircuitDictionaries
from circuit_analysis import *
from graphing import *
from simulated_annealing import run_walk, neighbour, free_vars
from helper_funcs import single

#####################################
# data recording

sbest_list = []
ebest_list = []

def capture_data():
    with open('data.csv', mode='w') as data:
        data = csv.writer(data, delimiter=',', quotechar='"', quoting=csv.
    QUOTE_MINIMAL)

        for e, s in zip(ebest_list, sbest_list):
            data.writerow([-e,s])

atexit.register(capture_data)

if __name__ == "__main__":

    trans = {} # holds the transistor tolerances to be tested
    if single.dBeta: # tests high and low beta
        trans = {"LO" :"ZTX107-LO", "HI" : "ZTX107-HI"}
    else: # nominal case
        trans = {"NOM" : "ZTX107-NOM"}

    # stores the starting configuration of the circuit
    sub_dicts = SubCircuitDictionaries()
    fb_dict = sub_dicts.get_feedbackamp_dict(trans)

    # randomizes the starting configuration within bounds
    # very bad starts tend to get stuck
```

```
    # practically designing a semi functional circuit is trivial
    # given more time I would add design equations to calculate a default state
    if single.isRand:
        temp = neighbour(fb_dict, free_vars, 10)
        gain_start = CircuitAnalyzer(temp).DC_gain
        while(gain_start > 944 or gain_start < 400): # bounds
            temp = neighbour(fb_dict, free_vars, 10)
            gain_start = CircuitAnalyzer(temp).DC_gain
        fb_dict = temp

    # grabs parameters from user's input
    v = single.isVerbose
    kMax = single.kAnnealing
    numWalk = single.nWalk
    Tinit = single.T

    # begins timing
    startSA = time.time()
    if v: # run conditions for verbose mode
        if single.isRand and single.isVerbose:
            make_bode_plot(CircuitAnalyzer(fb_dict).get_AC_analysis())
        print("Running Simulated Annealing in Verbose Mode...")
        for i in range(numWalk):
            print(f"Walk #{i+1}...")
            # run_walk computes one round of simulated annealing
            sbest, ebest = run_walk(T=Tinit, kMax=kMax, s_0=fb_dict)
            sbest_list.append(sbest)
            ebest_list.append(ebest)
    else: # non verbose
        print("Running Simulated Annealing...")
        with tqdm(total=kMax*numWalk) as pbar:
            for i in range(numWalk):
                # run_walk computes one round of simulated annealing
                sbest, ebest = run_walk(T=Tinit, kMax=kMax, s_0=fb_dict, pbar=pbar)
                sbest_list.append(sbest)
                ebest_list.append(ebest)
    # ends timer
    endSA = time.time()

    if v:
        print(f'Simulated Annealing Ebest: {min(ebest_list):0.2E}')
        print()

    # this run_walk is a greedy walk, i.e. SA with T = 0
    sbest = None
    ebest = None
    if v: # run conditions for verbose mode
        print("Running Greedy Walk in Verbose Mode...")
        sbest, ebest = run_walk(T=0, kMax=single.kGreedy, s_0=fb_dict)
    else: # non verbose
        print("Running Greedy Walk...")
        with tqdm(total=kMax*numWalk) as pbar:
            sbest, ebest = run_walk(T=0, kMax=single.kGreedy, s_0=fb_dict, pbar=pbar)
    endGRW = time.time()

    # reanalyses the start, SA best, and GRW best for comparison
```

```
    sGRW_analyser = CircuitAnalyzer(sbest)
    sSA_analyser = CircuitAnalyzer(sbest_list[np.argmin(ebest_list)])
    sstart_analyser = CircuitAnalyzer(fb_dict)

    # large printout
    print()
    print(f"        Neigbour SD: {single.sigma:.2}")
    print(f"           SA Steps: {kMax}, SA Walks: {numWalk}, Temp: {single.T}")
    print(f"          GRW Steps: {single.kGreedy}")
    print(f"    Check Trans Tol: {single.dBeta}")
    print()
    print(f"               Goal: BW:>{7.2*10**6:.2E}, Gain:~{1000:.2E}, Current:<{0.012:.2E}")
    print(f"              Start: BW: {sstart_analyser.BW:.2E}, Gain: {sstart_analyser.DC_gain:.2E}, Current: {sstart_analyser.OP_current:.2E}")
    print(f"Simulated Annealing: BW: {sSA_analyser.BW:.2E}, Gain: {sSA_analyser.DC_gain:.2E}, Current: {sSA_analyser.OP_current:.2E}, Time: {round(endSA - startSA)}s")
    print(f" Greedy Random Walk: BW: {sGRW_analyser.BW:.2E}, Gain: {sGRW_analyser.DC_gain:.2E}, Current: {sGRW_analyser.OP_current:.2E}, Time: {round(endGRW - endSA)}s")

    # making pretty plots
    make_bode_plot_from_list([sstart_analyser.get_AC_analysis(), sSA_analyser.get_AC_analysis(), sGRW_analyser.get_AC_analysis()])

    plt.scatter([i for i in range(len(mega_goodness))],mega_goodness)
    plt.xlabel("Iteration #")
    plt.ylabel("Goodness")
    plt.title("Goodness vs Cummulative Iteration #")
    plt.show()

    # adds GRW to list so it can be saved when program exits
    sbest_list.append(sbest)
    ebest_list.append(ebest)
```

4.2 simulated_annealing.py

```
import numpy as np
from PySpice.Unit import *
from circuit_analysis import *
from tqdm import tqdm
from helper_funcs import single

# annealing schedule
def temperature(T, b):
    return T*b

# acceptance function
def P(E_past, E_new, T):
    if(E_new >= -1): # rejects bad energies
        return False
    ratio = E_new/E_past
    if ratio >= 1: # if E_new is better
        return True
    if T != 0 and 4 > 100*(1-ratio)/T: # throws out below 2% chance
        return 1/(1 + np.exp(100*(1-ratio)/T)) > np.random.random()
```

```
    else:
        return False

# specify design variables
free_vars = ['RF','cascode1','cascode2','outStage']
free_vars_dict = {
    'cascode1' : ['RB1','RB2','RB3','RC','RE_deg','RE'],
    'cascode2' : ['RB1','RB2','RB3','RC','RE_deg','RE'],
    'outStage' : ['RE']
}

# generates a random state near "s"
def neighbour(s, free_vars, sd):
    s_new = s.copy()
    for key in free_vars:
        if type(s[key]) == dict: # recursion to handle sub circuits
            s_new[key] = neighbour(s[key], free_vars=free_vars_dict[key], sd
    =sd)
        else: # moves each component of the state by a random number of
    discrete steps
            s_new[key] = iter_resistor(s[key], round(np.random.normal(0,sd))
    )
    return s_new

# smallest valid 2% resistor values
valid_res = [10.0, 10.5, 11.0, 11.5, 12.1, 12.7, 13.3, 14.0, 14.7, 15.4,
    16.2, 16.9, 17.8, 18.7, 19.6, 20.5, 21.5, 22.6, 23.7, 24.9, 26.1, 27.4,
    28.7, 30.1, 31.6, 33.2, 34.8, 36.5, 38.3, 40.2, 42.2, 44.2, 46.4, 48.7,
    51.1, 53.6, 56.2, 59.0, 61.9, 64.9, 68.1, 71.5, 75.0, 78.7, 82.5, 86.6,
    90.9, 95.3]
len_valid_res = len(valid_res)

# changes the resistor value by a given number of steps up or down
def iter_resistor(R,n):
    if n == 0: # no change
        return R

    # extract order of magnitude
    OOM = np.floor(np.log10(R))

    # find resistor's index in valid_res
    i = valid_res.index(float(str(float(round(R,3))).replace('.','')[0:3])
    /10)

    while i + n >= len_valid_res: # allows overflow wrapping
        OOM += 1
        n -= len_valid_res
    while i + n <= -len_valid_res: # allows underflow wrapping
        OOM -= 1
        n += len_valid_res

    # calculates the new resistor
    return_val = valid_res[i + n]*10**(OOM-1)

    if return_val < 10: # bounds on allowed resistor values
        return 10
    elif return_val > 22*10**6:
        return 21.5*10**6
    else:
```

```
        return return_val

# runs a simulated annealing walk
def run_walk(T : float, kMax : int, s_0 : dict, pbar=0):
    s = s_0.copy()
    sbest = s.copy()
    e = -CircuitAnalyzer(s).goodness
    ebest = e
    if T != 0: # accounts for zero temp (greedy random)
        b = T**(-2/kMax)
    else:
        b = 0

    for i in range(kMax):
        # annealing schedule
        T = temperature(T, b=b)
        # generates a nearby state
        snew = neighbour(s=s, free_vars=free_vars, sd=single.sigma)
        try: # simulates the circuit and calculates goodness
            enew = -CircuitAnalyzer(snew).goodness
        except: # catches errors in NgSpice
            print(f"Something went wrong with: {snew}")
            enew = 0

        if P(e, enew, T): # acceptance
            s = snew.copy()
            e = enew

        if enew < ebest: # records best
            sbest = snew.copy()
            ebest = enew

        if single.isVerbose and i%100 == 0: # verbose printing
            print(f"{round(i/kMax*100)}% Complete")
            print(f"Current Energy: {e:.2E}")
            print(f"    New Energy: {enew:.2E}")
            print()

        if not single.isVerbose: # non verbose output
            pbar.update()

    return sbest, ebest
```

4.3 circuit_analysis.py

```
import numpy as np

##########################################

from PySpice.Spice.Netlist import Circuit
from PySpice.Spice.NgSpice.Simulation import NgSpiceSharedCircuitSimulator
from PySpice.Unit import *

##########################################

from subcircuit_def import *

# records every valid goodness value
mega_goodness = []
```

```
15
16 # grabs the transistor paths
17 dicts = SubCircuitDictionaries()
18 trans_dict = dicts.get_trans_dict()
19
20 # takes in a circuit definition (state) and calculates goodness
21 class CircuitAnalyzer:
22     def __init__(self, curr_dict : dict[str, float | dict[str,str]]) -> None
    :
23         self.curr_dict = curr_dict
24         # PySpice circuit from input dictionary
25         self.circuits = self.__make_circuit()
26         # creates simulators for each transistor tolerance
27         self.simulators = self.__make_simulator()
28         # DC simulation of circuit
29         self.__DC_analyses = self.__make_DC_analysis()
30         # extracts the operating current
31         self.OP_current = self.__get_OP_current()
32         # calculates punishment for current usage
33         self.OP_current_goodness = max(min(1,1 if self.OP_current <= 0.01
    else 1-(100*(self.OP_current-0.01))**2),0.001)
34         if (self.OP_current_goodness > 0.5): # rejects high current circuits
35             # AC simulation
36             self.__AC_analyses = self.__make_AC_analysis()
37             self.frequencies = dict()
38             self.gains = dict()
39             # grabs the frequency and gain data from the analysis
40             for key, value in self.__AC_analyses.items():
41                 self.frequencies[key] = value.frequency
42                 self.gains[key] = np.absolute(value.AC_out)
43             # calculates & punishes the bandwidth
44             self.BW, self.DC_gain, self.key_min = self.__get_BW()
45             # calculates goodness (higher better)
46             self.goodness = self.__get_goodness()
47             # records valid goodness
48             mega_goodness.append(self.goodness)
49         else: # rejected circuit
50             self.BW = 0
51             self.DC_gain = 0
52             self.goodness = -1
53
54     # creates the circuit using the PySpice environment
55     def __make_circuit(self) -> dict[str, Circuit]:
56         circuits = {}
57         for key, value in self.curr_dict['trans'].items(): # type: ignore
58             circuit = get_base_circuit()
59             circuit.SinusoidalVoltageSource('AC_voltage', 'ac_in', circuit.
    gnd, amplitude=1@u_V)
60             circuit.R('RAC', 'ac_in', 'in_node', 1500@u_Ohm)
61             circuit.include(trans_dict[value])
62             circuit.subcircuit(FeedBackAmp(self.curr_dict, value))
63             circuit.X('fbamp1','feedbackamp','Vcc', circuit.gnd,'in_node','
    out')
64             circuits[key] = circuit
65         return circuits
66
67     # creates simulator objects for each transistor tolerance
68     def __make_simulator(self, temperature = 25) -> dict[str,
    NgSpiceSharedCircuitSimulator]:
```

```
        simulators = dict()
        for key, value in self.circuits.items():
            temp = value.simulator(temperature=temperature,
    nominal_temperature=temperature)
            temp.save(["AC_out","i(vvdc)"])
            simulators[key] = temp
        return simulators

    # calculates DC operating point of the circuit
    def __make_DC_analysis(self) -> dict:
        DC_analyses = dict()
        for key, value in self.simulators.items():
            DC_analyses[key] = value.operating_point()
        return DC_analyses

    # returns the max current draw of all the transistor tolerances
    def __get_OP_current(self) -> float:
        return max(max([abs(float(np.absolute(x[0]))) for x in DC_analysis.
    branches.values()]) for DC_analysis in self.__DC_analyses.values())

    # simulates the circuit between a range of frequencies on a log scale
    def __make_AC_analysis(self) -> dict:
        AC_analyses = dict()
        for key, value in self.simulators.items():
            AC_analyses[key] = value.ac(start_frequency=1@u_kHz,
    stop_frequency=100@u_MHz, number_of_points=50,  variation='dec')
        return AC_analyses

    # calculates the 1.5dB bandwidth and punishes bandwidth for being out of
     the desired gain range
    def __get_BW(self) -> tuple[float, float, str]:
        BW_min = None
        DC_gain_min = None
        key_min = ""
        for key, freq in self.frequencies.items():
            DC_gain_curr = self.gains[key][0].value
            if DC_gain_curr > 100: # only check minimum operational
    amplifiers
                index_3dB = None
                for i, x in enumerate(self.gains[key]):
                    if x.value > 1188.5 or x.value < 944: # finds the
    60+-1.5dB BW
                        index_3dB = i
                        break
                if index_3dB > 1: # if its valid, approximates the exact
    value
                    currBW = self.__lin_approx(freq[index_3dB].value,self.
    gains[key][index_3dB],freq[index_3dB-1].value,self.gains[key][index_3dB
    -1], DC_gain_curr*0.944)
                else: # invalid -> needs to be punished
                    if index_3dB == 0:
                        for i, x in enumerate(self.gains[key]):
                            # finds 1.5dB bandwidth
                            if x.value <= DC_gain_curr*0.944 or x.value >=
    DC_gain_curr/0.944:
                                index_3dB = i
                                break
                        # approximates exact value
                        currBW = self.__lin_approx(freq[index_3dB].value,
```

```
        self.gains[key][index_3dB],freq[index_3dB-1].value,self.gains[key][
        index_3dB-1], DC_gain_curr*0.944)
118                         i = 0
119                         adj = 0
120                         # punishes bandwidth based on mean square distance
        from desired gain
121                         while(freq[i].value < 7.2*10**6):
122                             adj += min((self.gains[key][i].value - 1188.5)
        **2,(self.gains[key][i].value - 944)**2)
123                             i += 1
124                         currBW /= np.sqrt(adj)/i
125                 # boot strapping and takes lower of BWs
126                 if index_3dB != None and currBW != None and (BW_min == None
        or currBW < BW_min):
127                     BW_min = currBW
128                     DC_gain_min = DC_gain_curr
129                     key_min = key
130         return (lambda: (BW_min, DC_gain_min, key_min), lambda: (0,0,self.
        curr_dict['trans'][list(self.curr_dict['trans'].keys())[0]]))[BW_min ==
        None]()
131
132     # simple linear approximation
133     def __lin_approx(self, x1, y1, x2, y2, y_target):
134         return (y_target-y1)*(x2-x1)/(y2-y1) + x1
135
136     # unused
137     def __get_GBWP(self) -> float:
138         return self.BW*self.DC_gain
139
140     # calculates goodness
141     def __get_goodness(self) -> float:
142         goodness = self.BW*self.OP_current_goodness
143         if goodness > 0:
144             return goodness
145         else:
146             return 0
147
148     # returns analysis for worst case of transistor tolerance
149     def get_AC_analysis(self):
150         return self.__AC_analyses[self.key_min]
```

4.4 subcircuit_def.py

```
1  from PySpice.Spice.Netlist import SubCircuitFactory, Circuit
2  from PySpice.Unit import *
3
4  ################################################
5
6  class SubCircuitDictionaries:
7      def __init__(self) -> None:
8          pass
9
10     def get_trans_dict(self) -> dict: # path to transistor models
11         return {
12             "2N2222A" : "library/2N2222A.lib",
13             "ZTX107-HI" : "library/ZTX107-HI.lib",
14             "ZTX107-NOM" : "library/ZTX107-NOM.lib",
15             "ZTX107-LO" : "library/ZTX107-LO.lib"
16         }
```

```
17
18      # resistors in ohms
19      # capacitors in micro farads
20      # default starting state for each resistor
21      def get_cascode1_dict(self) -> dict[str, float | dict[str,str]]:
22          return {
23              'Cc' : 10**6,
24              'RB1' : 100000,
25              'RB2' : 12100,
26              'RB3' : 21500,
27              'RC' : 3010,
28              'RE_deg' : 19.6,
29              'RE' : 237
30          }
31
32      def get_cascode2_dict(self) -> dict[str, float | dict[str,str]]:
33          return {
34              'Cc' : 10**6,
35              'RB1' : 59000,
36              'RB2' : 10500,
37              'RB3' : 18700,
38              'RC' : 1620,
39              'RE_deg' : 11.0,
40              'RE' : 383
41          }
42
43      def get_inStage_dict(self) -> dict[str, float | dict[str,str]]:
44          return {
45              'Cc' : 10**6,
46              'RB1' : 205000,
47              'RB2' : 51100,
48              'RE' : 715
49          }
50
51      def get_outStage_dict(self) -> dict[str, float | dict[str,str]]:
52          return {
53              'Cc' : 10**6,
54              'RB1' : 147000,
55              'RB2' : 365000,
56              'RB3' : 147000,
57              'RC' : 301,
58              'RE' : 133
59          }
60
61      # entire circuit definition packaged into dictionary
62      def get_feedbackamp_dict(self, trans):
63          return {
64              'cascode1' : self.get_cascode1_dict(),
65              'cascode2' : self.get_cascode2_dict(),
66              'inStage' : self.get_inStage_dict(),
67              'outStage' : self.get_outStage_dict(),
68              'RF' : 30100,
69              'trans' : trans
70          }
71
72  # Cc on input but not output
73  class Cascode1(SubCircuitFactory):
74      NODES = ('Vcc','gnd','in_node','out','VE')
75      NAME = 'cascode1'
```

```
76      def __init__(self, cascodeDict : dict[str, float | dict[str, str]],
        trans : str):
77          super().__init__()
78          # bias resistors
79          self.R('RB1','Vcc','VB1',cascodeDict['RB1']@u_Ohm)
80          self.R('RB2','VB1','VB2',cascodeDict['RB2']@u_Ohm)
81          self.R('RB3','VB2','gnd',cascodeDict['RB3']@u_Ohm)
82          # bias Cc
83          self.C('Cc1','in_node','VB2', cascodeDict['Cc']@u_uF)
84          self.C('Cc2','gnd','VB1', cascodeDict['Cc']@u_uF)
85          # transistors
86          self.R('RC','Vcc','out',cascodeDict['RC']@u_Ohm)
87          self.BJT('Q1','out','VB1','VC',model=trans) # type: ignore
88          self.BJT('Q2','VC','VB2','VE',model=trans) # type: ignore
89          self.R('RE_deg','VE','VE2',cascodeDict['RE_deg']@u_Ohm)
90          self.C('Cc3','VE2','gnd',cascodeDict['Cc']@u_uF)
91          self.R('RE','VE2','gnd',cascodeDict['RE']@u_Ohm)
92
93  # Cc on input but not output
94  class Cascode2(SubCircuitFactory): # need a second one with diff name
95      NODES = ('Vcc','gnd','in_node','out')
96      NAME = 'cascode2'
97      def __init__(self, cascodeDict : dict[str, float | dict[str, str]],
        trans : str):
98          super().__init__()
99          # bias resistors
100         self.R('RB1','Vcc','VB1',cascodeDict['RB1']@u_Ohm)
101         self.R('RB2','VB1','VB2',cascodeDict['RB2']@u_Ohm)
102         self.R('RB3','VB2','gnd',cascodeDict['RB3']@u_Ohm)
103         # bias Cc
104         self.C('Cc1','in_node','VB2', cascodeDict['Cc']@u_uF)
105         self.C('Cc2','gnd','VB1', cascodeDict['Cc']@u_uF)
106         # transistors
107         self.R('RC','Vcc','out',cascodeDict['RC']@u_Ohm)
108         self.BJT('Q1','out','VB1','VC',model=trans) # type: ignore
109         self.BJT('Q2','VC','VB2','VE',model=trans) # type: ignore
110         self.R('RE_deg','VE','VE2',cascodeDict['RE_deg']@u_Ohm)
111         self.C('Cc3','VE2','gnd',cascodeDict['Cc']@u_uF)
112         self.R('RE','VE2','gnd',cascodeDict['RE']@u_Ohm)
113
114 # Cc on input but not output
115 class InputStage(SubCircuitFactory):
116     NODES = ('Vcc','gnd','in_node','out')
117     NAME = 'inStage'
118     def __init__(self, inputDict : dict[str, float | dict[str, str]], trans
        : str):
119         super().__init__()
120         # Input Cap
121         self.C('Cc1','in_node','VB', inputDict['Cc']@u_uF)
122         # bias resistors
123         self.R('RB1', 'Vcc', 'VB', inputDict['RB1']@u_Ohm)
124         self.R('RB2', 'VB', 'gnd', inputDict['RB2']@u_Ohm)
125         # transistor
126         self.BJT('Q1','Vcc','VB','out',model=trans) # type: ignore
127         self.R('RE','out','gnd',inputDict['RE']@u_Ohm)
128
129 # Cc on input but not output
130 class OutStage(SubCircuitFactory):
131     NODES = ('Vcc','gnd','in_node','out')
```

```
    NAME = 'outStage'
    def __init__(self, outDict : dict[str, float | dict[str, str]], trans :
    str):
        super().__init__()
        # bias resistors
        self.R('RB1','Vcc','VB1',outDict['RB1']@u_Ohm)
        self.R('RB2','VB1','VB2',outDict['RB2']@u_Ohm)
        self.R('RB3','VB2','gnd',outDict['RB3']@u_Ohm)
        # bias Cc
        self.C('Cc1','in_node','VB2', outDict['Cc']@u_uF)
        self.C('Cc2','gnd','VB1', outDict['Cc']@u_uF)
        # transistors
        self.R('RC','Vcc','out',outDict['RC']@u_Ohm)
        self.BJT('Q1','out','VB1','VC',model=trans) # type: ignore
        self.BJT('Q2','VC','VB2','VE',model=trans) # type: ignore
        self.R('RE','VE','gnd',outDict['RE']@u_Ohm)

# Cc on input but not output
class FeedBackAmp(SubCircuitFactory):
    NAME = 'feedbackamp'
    NODES = ('Vcc','gnd','in_node','out')
    def __init__(self, fbDict : dict, trans : str):
        super().__init__()
        # Input Stage
        self.subcircuit(InputStage(fbDict['inStage'], trans))
        self.X('in_Stage','inStage','Vcc','gnd','in_node','gain_in')
        # Gain Stages
        self.subcircuit(Cascode1(fbDict['cascode1'], trans))
        self.X('cascode_1','cascode1','Vcc', 'gnd','gain_in','gain_int','
    FB_in')
        self.subcircuit(Cascode2(fbDict['cascode2'], trans))
        self.X('cascode_2','cascode2','Vcc', 'gnd','gain_int','gain_out')
        self.R('RF','gain_out','FB_in',fbDict['RF']@u_Ohm)
        # Output Stage
        self.subcircuit(OutStage(fbDict['outStage'], trans))
        self.X('out_stage','outStage','Vcc','gnd','gain_out','out')


def get_base_circuit() -> Circuit:
    # circuit setup
    circuit = Circuit('main')

    # general circuit definition
    circuit.V('VDC', 'Vcc', circuit.gnd, 9@u_V)
    circuit.C('Cc_load','out','AC_out', 1@u_F)
    circuit.R('R_load','AC_out',circuit.gnd, 300@u_Ohm)
    return circuit
```

4.5 helper_funcs.py

```
import atexit

# prevents windows from sleeping while program is running
class WindowsInhibitor:
    ES_CONTINUOUS = 0x80000000
    ES_SYSTEM_REQUIRED = 0x00000001

    def __init__(self):
        pass
```

```
10
11      def inhibit(self):
12          import ctypes
13          print("Preventing Windows from going to sleep")
14          ctypes.windll.kernel32.SetThreadExecutionState(
15              WindowsInhibitor.ES_CONTINUOUS | \
16              WindowsInhibitor.ES_SYSTEM_REQUIRED)
17
18      def uninhibit(self):
19          import ctypes
20          print("Allowing Windows to go to sleep")
21          ctypes.windll.kernel32.SetThreadExecutionState(
22              WindowsInhibitor.ES_CONTINUOUS)
23
24  # sleep inhibitor
25  osSleep = WindowsInhibitor()
26  osSleep.inhibit()
27
28  # un inhibits no sleep when program finishes
29  def exit_handler():
30      osSleep.uninhibit()
31
32  atexit.register(exit_handler)
33
34  # gets user input and stores in object for easy global access
35  class singleton:
36      def __init__(self) -> None:
37          print()
38          des = input("Default Behavior? ~5min (y/n): ").lower()
39          if (des == 'y'):
40              self.isVerbose = True
41              self.isRand = False
42              self.kAnnealing = 2000
43              self.nWalk = 3
44              self.T = 30
45              self.kGreedy = 6000
46              self.sigma = 0.3
47              self.dBeta = True
48              print()
49          elif (des == 'test'):
50              self.isVerbose = True
51              self.isRand = True
52              self.kAnnealing = 1000
53              self.nWalk = 1
54              self.T = 30
55              self.kGreedy = 250
56              self.sigma = 0.3
57              self.dBeta = False
58              print()
59          else:
60              print()
61              self.isVerbose = ("y" == input("Verbose? (y/n): ").lower())
62              print()
63              self.isRand = ("y" == input("Random Starting Point? (y/n): ").
    lower())
64              if self.isRand:
65                  print()
66                  print("!!!! Recommend higher SD, assume nominal transistors,
     and num steps !!!!")
```

```
            print()
            self.dBeta = ("y" == input("(y) Check transistor tolerances or (n) assume nominal (y/n): ").lower())
            print()
            self.sigma = float(input("Neighbor Standard Deviation (recommend <1): "))
            print()
            print("Simulated Annealing")
            self.kAnnealing = int(input("Num steps per walk: "))
            self.nWalk = int(input("Num walks: "))
            self.T = float(input("Starting Temperature (recommend <30): "))
            print()
            print("Greedy Random Walk")
            self.kGreedy = int(input("Num steps: "))

single = singleton()
```

4.6 graphing.py

```
import matplotlib.pyplot as plt
import numpy as np
import math

from PySpice.Probe.Plot import plot

# making bode plots
def bode_diagram(axes, frequency, gain, phase, **kwargs):
    bode_diagram_gain(axes[0], frequency, gain, **kwargs)
    bode_diagram_phase(axes[1], frequency, phase, **kwargs)

def bode_diagram_gain(axe, frequency, gain, **kwargs):
    axe.semilogx(frequency, gain, **kwargs)
    axe.grid(True)
    axe.grid(True, which='minor')
    axe.set_xlabel("Frequency [Hz]")
    axe.set_ylabel("Gain [dB]")

def bode_diagram_phase(axe, frequency, phase, **kwargs):
    axe.semilogx(frequency, phase, **kwargs)
    axe.set_ylim(-math.pi, math.pi)
    axe.grid(True)
    axe.grid(True, which='minor')
    axe.set_xlabel("Frequency [Hz]")
    axe.set_ylabel("Phase [rads]")
    # axe.set_yticks # Fixme:
    plt.yticks((-math.pi, -math.pi/2,0, math.pi/2, math.pi),
                   (r"$-\pi$", r"$-\frac{\pi}{2}$", "0", r"$\frac{\pi}{2}$", r"$\pi$"))

def make_bode_plot(analysis):
    figure, (ax1, ax2) = plt.subplots(2, figsize=(20, 10))
    plt.title("Bode Diagram of an Operational Amplifier")
    bode_diagram(axes=(ax1, ax2),
                 frequency=analysis.frequency,
                 gain=20*np.log10(np.absolute([x.value for x in np.absolute(analysis.AC_out)])),
                 phase=np.angle([x.value for x in np.absolute(analysis.AC_out)], deg=False),
                 marker='.',
```

```
                color='blue',
                linestyle='-',
                )
    ax1.hlines([61.5,58.5],[0,0],[100*10**6,100*10**6])
    plt.tight_layout()
    plt.show()

colors = ["red","green","blue"]
label = ["Start", "SA", "GRW"]

def make_bode_plot_from_list(analysis_list):
    figure, (ax1, ax2) = plt.subplots(2, figsize=(20, 10))
    plt.title("Bode Diagram of an Operational Amplifier")
    for i, analysis in enumerate(analysis_list):
        bode_diagram(axes=(ax1, ax2),
                     frequency=analysis.frequency,
                     gain=20*np.log10([x.value for x in np.absolute(analysis.
    AC_out)]),
                     phase=np.angle([x.value for x in analysis.AC_out], deg=
    False),
                     marker='.',
                     color=colors[i],
                     label=label[i],
                     linestyle='--',
                     )
    ax1.hlines([61.5,58.5],[0,0],[100*10**6,100*10**6])
    ax1.legend()
    plt.tight_layout()
    plt.show()

def make_2d_plot(x, y):
    plt.plot(x,y, 'o')
    plt.show()
```