# Simulated Annealing And Its Applications

Pre-Master's Final Project
Artificial Intelligence
Rodrigo P. Coelho

November/2016

# Contents



- **Bibliography Review:**
  - Current State/ Applications / Main Parameters / Current Research
- **Parameter Study:**
  - Development of a test bench system in Python
  - Performance under different Functions and State Space sizes
  - Points per Temperature and other parameters (Temperature Schedule)
- **Implemented Features:**
  - Cutoff
  - State Space Narrowing
  - Recursive Application
  - Effectiveness of the Implemented Features
  - Combination with Quasi-Newton BFGS and CG Gradient Descent
- **Applications Implemented and Demonstrations:**
  - Optimization of tridimensional functions
  - Optimization N dimensional functional
  - Optimization of Routes (TSP Problem)
  - Neural Network Training:
    - Prototype with 6 Neurons
    - MNIST Digit Recognition System with 844 neurons – 50000 images

**SA is a heuristic method of optimization that simulates a set of atoms cooling to a minimum energy state.**

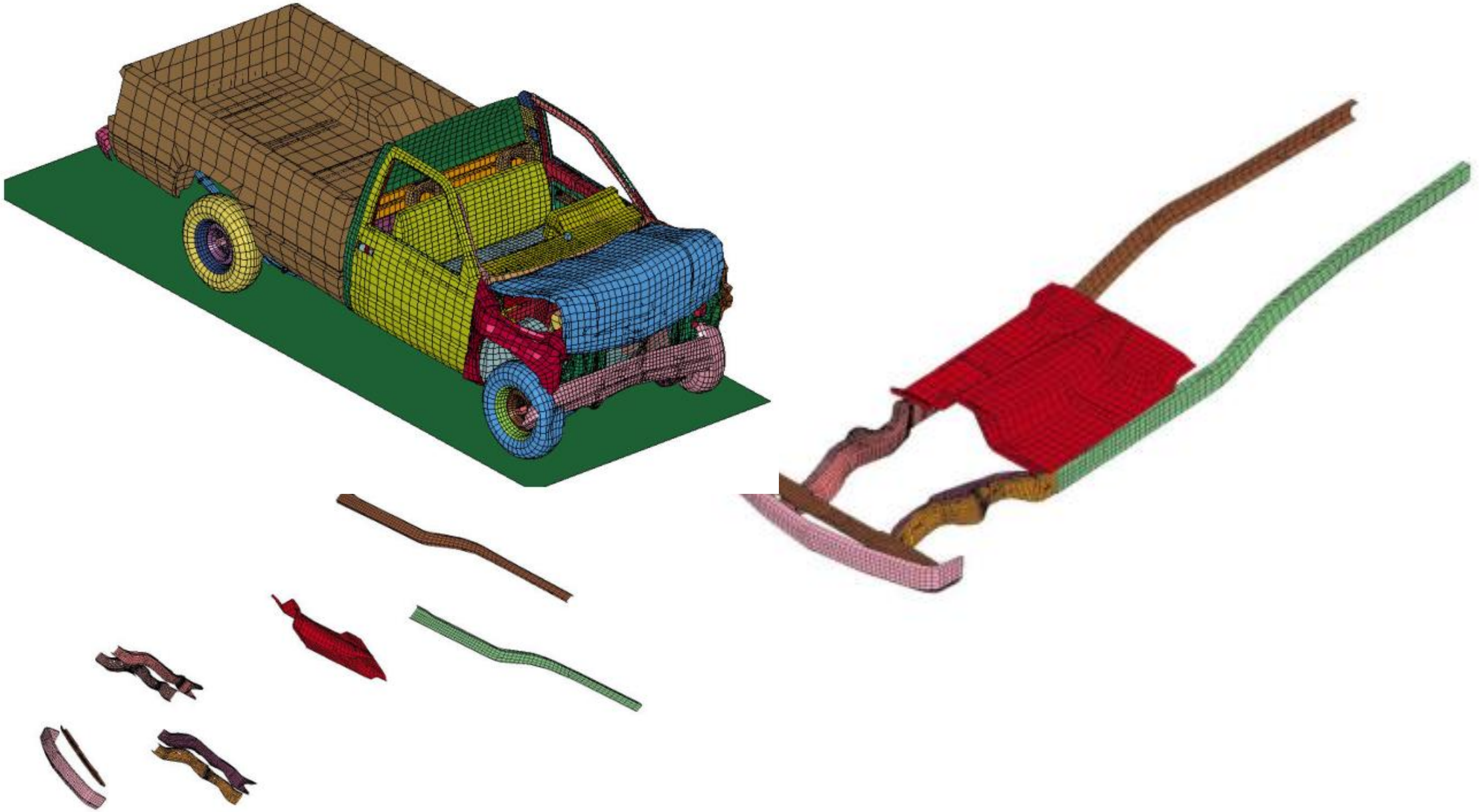
- Materials naturally settle in a state of minimum energy.
- Kirkpatrick, Gelett e Vecchi (1983):
  - Mechanical Statistics -> Combinatorial Optimization Problems;
  - Note: 1 $cm^3$ has $10^{23}$ atoms;
  - Cooling must be slow.
- In high temperatures -> Algorithm explores the search space;
- In low temperatures -> Transitions to a “hill climbing” behavior.

## SA has applications in many fields of science:

- Combinatorial Math, Decision Trees and Classification;
- Design of Electronic Circuits and Minimization of Wiring;
- Biostatistics (Optimization of Mixtures and Protein Chains);
- Geophysics e Physics (Mechanical Statistics);
- Finance (Portfolio Optimization and Risk Reduction);
- Optimization of Containers (Rucksack Problem);
- Optimization of Routes (TSP Problem);
- Application of filters in images and training of Neural Networks (2015);
- Optimization of Mechanical Components (Impact Reduction).

# SA: Applications – Impact Reduction





**SA is used in finite elements models to optimize the thickness of the different parts of an automobile to minimize crash impact.**

## Simulated Annealing has 3 main “components”:

- $g(\Delta x)$: a function or methodology to explore the search space;
- $h(\Delta E)$: a function to determine if we should or not accept the new value of $f(x)$;
- $T(k)$: Annealing or Cooling Schedule;
- Each type of problem will need an analysis of these components;
- Any improvement proposal to the algorithm will come in the shape of :
  - How to improve each component;
  - How to combine these components in the best way possible.

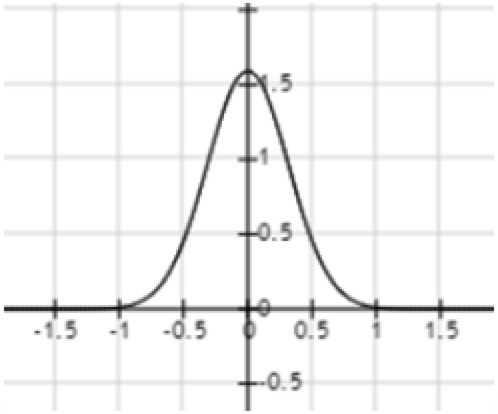
# The Search Space can be sampled in two ways:

- Uniform Sampling:
  - $g(x) = x_0 + \Delta x$, ($\Delta x$ = a uniform distribution between $x_{min}$ and $x_{max}$);

    xn[nd] = xbest[nd] + random.uniform(-self.DELTAObjFunc
    * self.OBJfunctionRANGE, self.DELTAObjFunc
    * self.OBJfunctionRANGE)

- Gaussian Sampling:
  - $g(x) = (2\pi T)^{-\frac{D}{2}} \exp\left[\frac{-(x-x_0)^2}{2T}\right]$
  - Radius gets smaller with lower temperatures

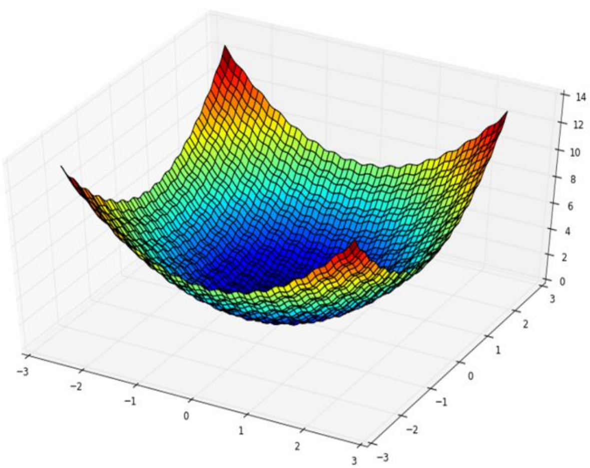
    xn[nd] = random.gauss(xbest[nd],math.sqrt(T))

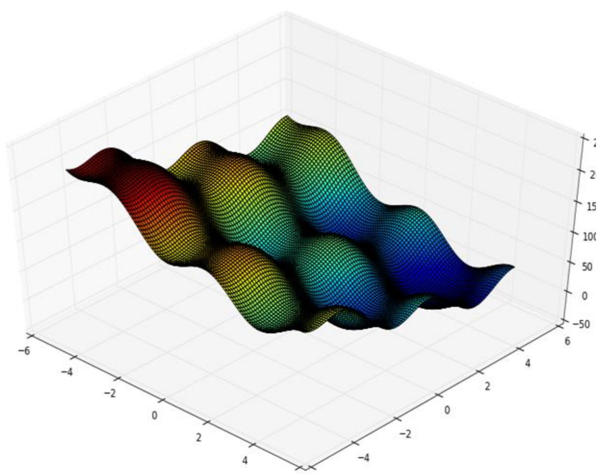


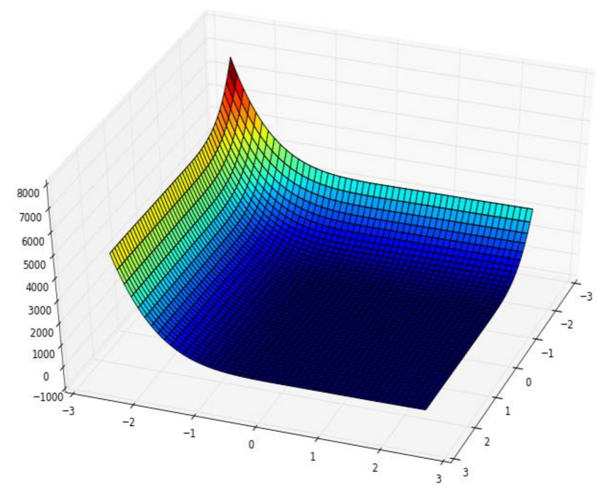

$x_o=0$, $D=2, T=0.1$

# SA: Studied Functions

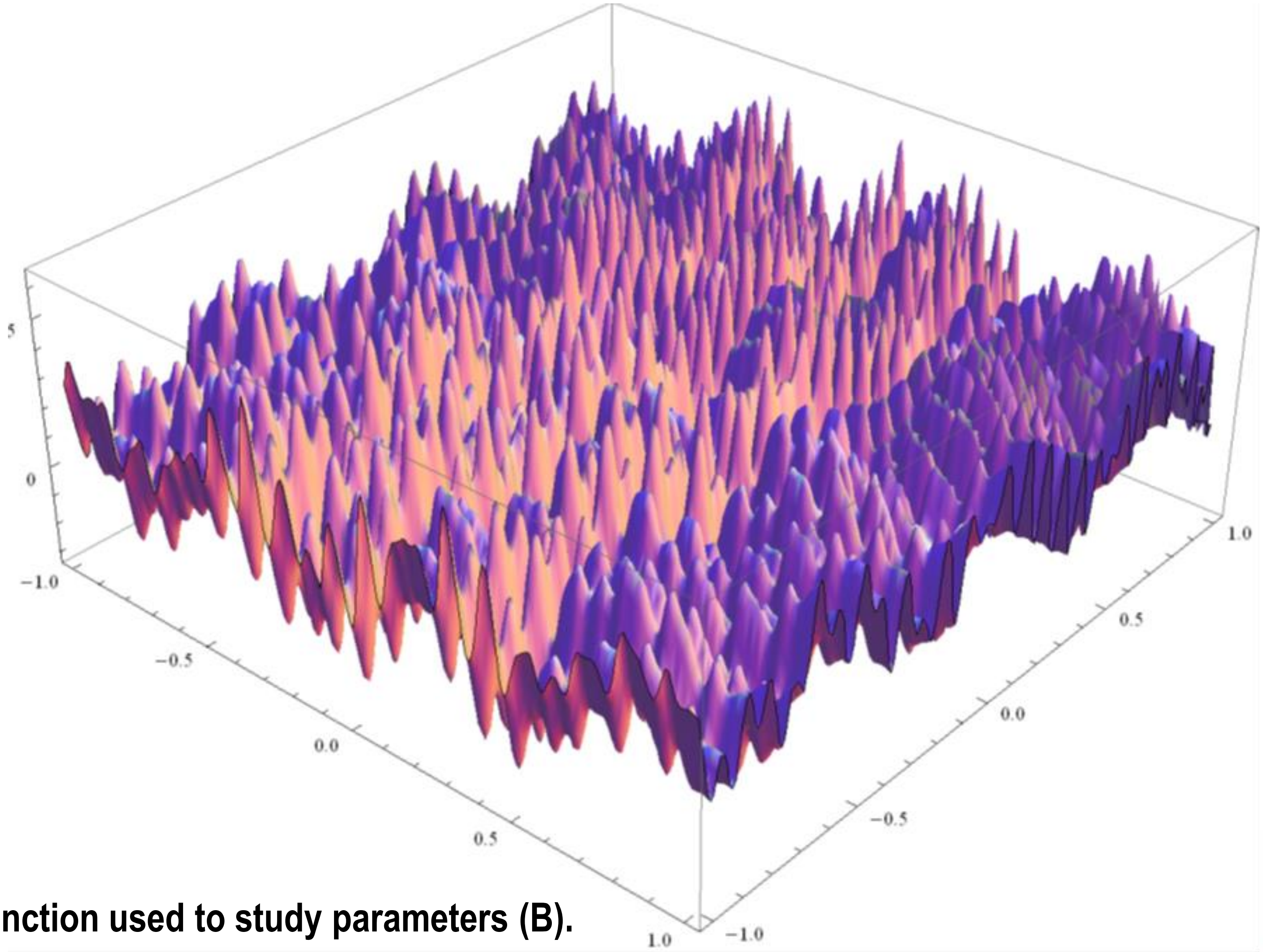
```
def f(self, x, y): # define objective functions
    Func = { 'A': 0.7 + x**2 + y**2 - 0.1*math.cos(6.0*3.1415*x) - 0.1*math.cos(6.0*3.1415*y),
             'B': (math.e**math.sin(50*x)+math.sin(60*math.e**y)+math.sin(70*math.sin(x))+math.sin(math.sin(80*y))-math.sin(10*(x + y))
                   + 0.25*(x**2 + y**2)),
             'C': x**2+ y**2 + 5,
             'D': 10000*(math.e**math.sin(50*x) + math.sin(60*math.e**y) + math.sin(70*math.sin(x)) + math.sin(math.sin(80*y))
                   - math.sin(10*(x + y)) + 0.25*(x**2 + y**2)),
             'E': (2*x**6-12.2*x**5+21.2*x**4+6.2*x-6.4*x**3-4.7*x**2+y**6-11*y**5+43.3*y**4-10*y-74.8*y**3+56.9*y**2-4.1*x*y-0.1*y**2*x**2
                   +0.4*x*y**2+0.4*x**2*y),
             'F': (1-x)**2 + 25*(y-x**2)**2 + (math.cos(3*math.pi*x+4*math.pi*y)+1),
             'G': 20*math.sin(x*np.pi/2-2*np.pi) + 20*math.sin(y*np.pi/2-2*np.pi)+(x-2*np.pi)**2+(y-2*np.pi)**2}
    return Func[self.FuncOption]
```







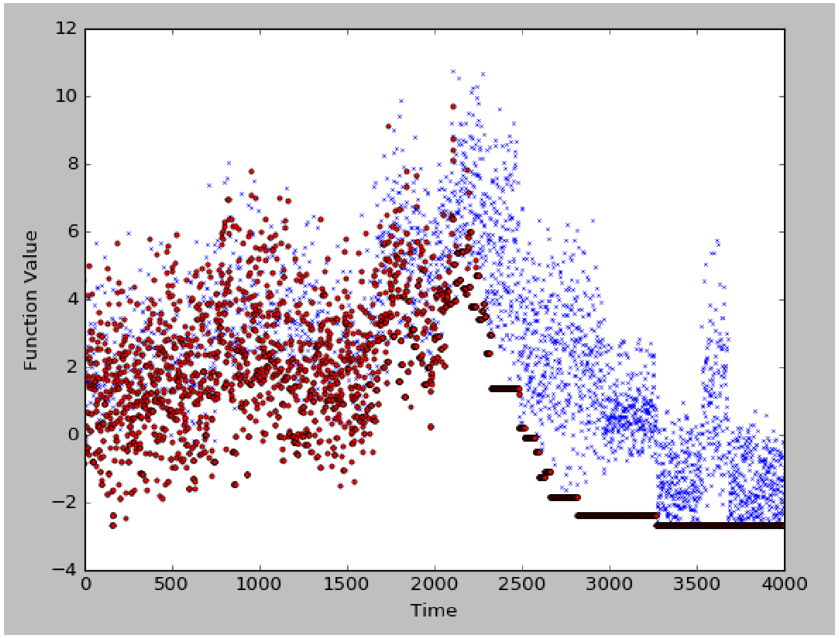
# SA: Studied Functions (Continued)

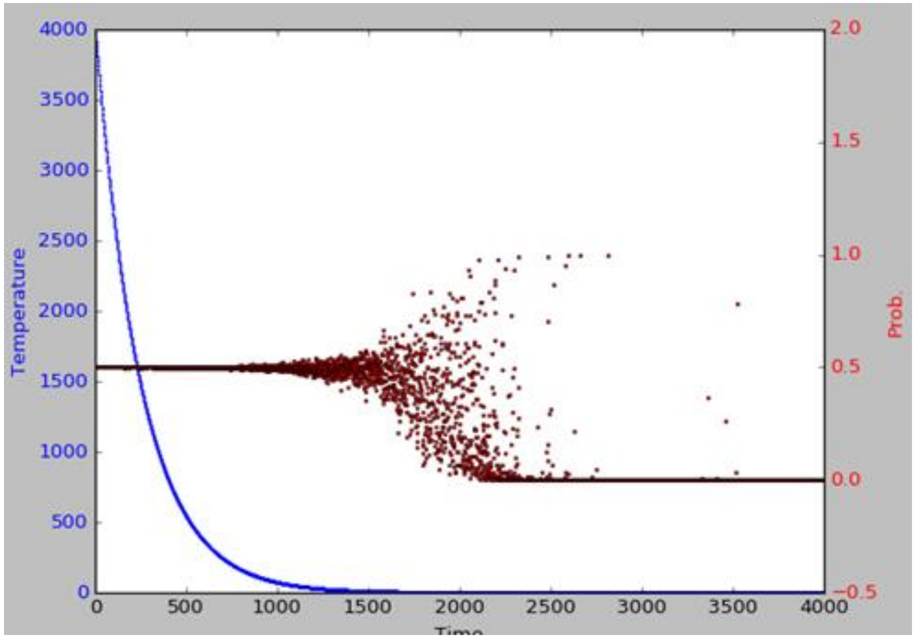



Main function used to study parameters (B).

# SA: Auxiliary Graphs and Implemented Features

**To help understand how the parameters function, it is useful to build these graphs:**

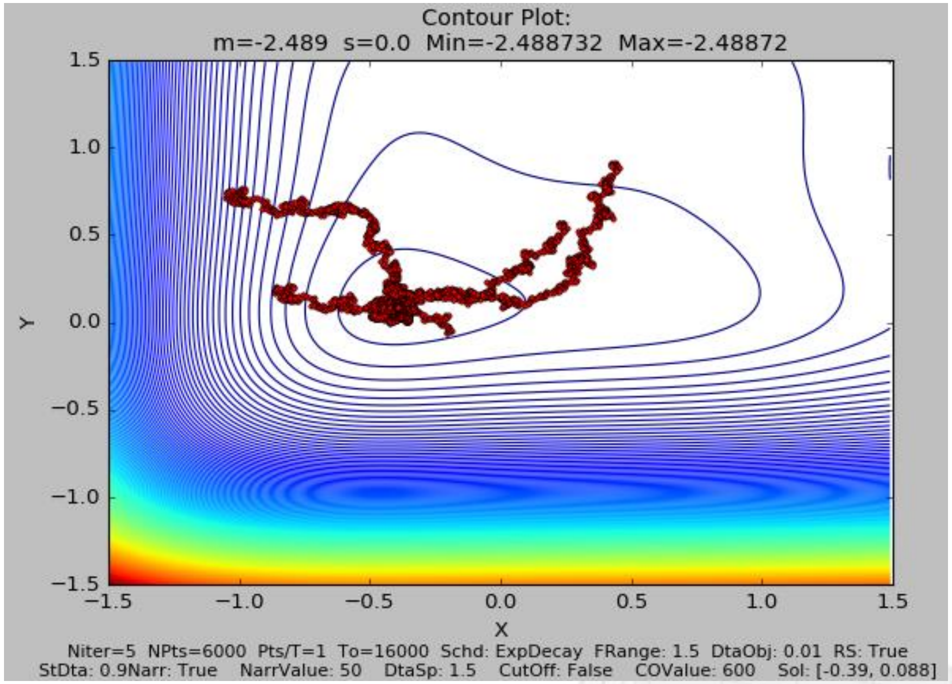


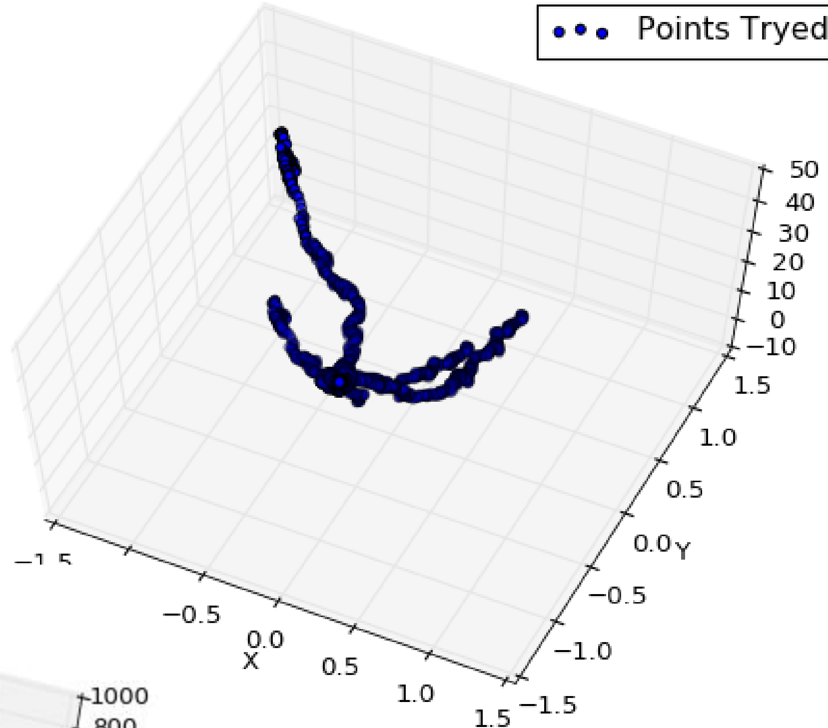



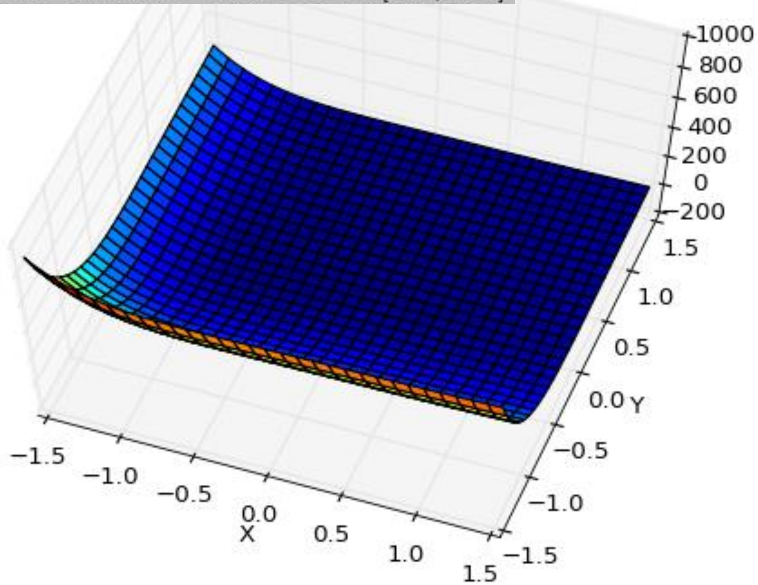

**Features Implemented:**

- Gaussian or Uniform Sampling;
- DELTAObjFunc: In Uniform Sampling, delta in X and Y that can be explored from the current state;
- State Space Narrowing: If a better solution is not found in SSNV iterations searches very near the current state.  Reopens the search space if a better solution is found near the current coordinate.  If still not found after SSNV * 2 iterations it goes to the best minimum found so far;
- Cutoff: Stops the search after CutoffValue iterations with no improvement;
- Points per Temperature: Number of times that f(x) is evaluated before lowering the temperature;
- Recursive Application: Takes the result of the last iteration and uses as the starting point for the next annealing run.

# SA: Auxiliary Graphs – Contour Plot

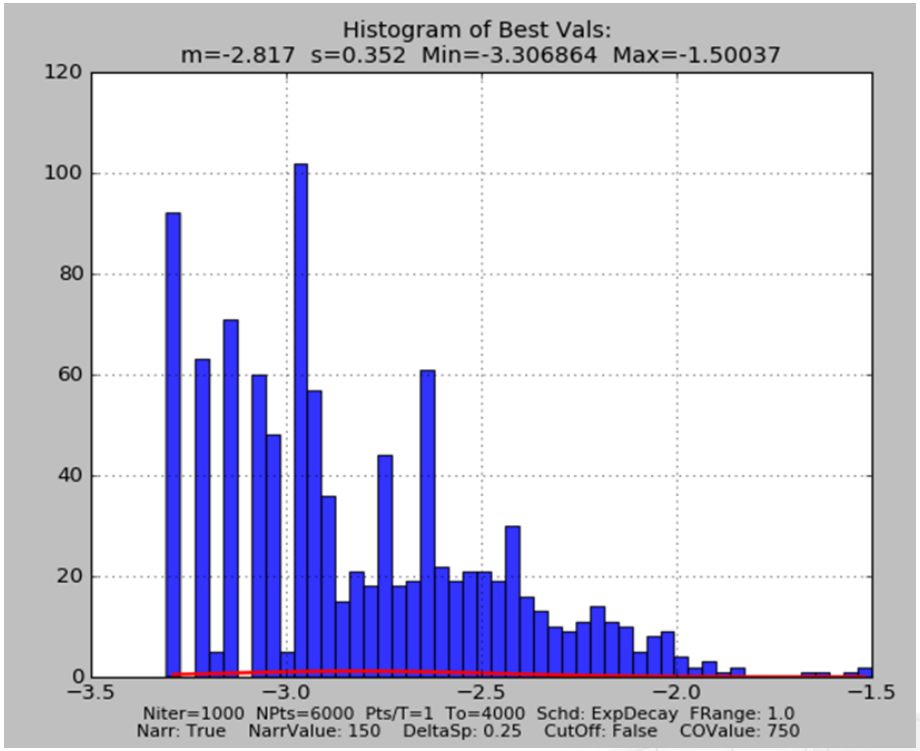


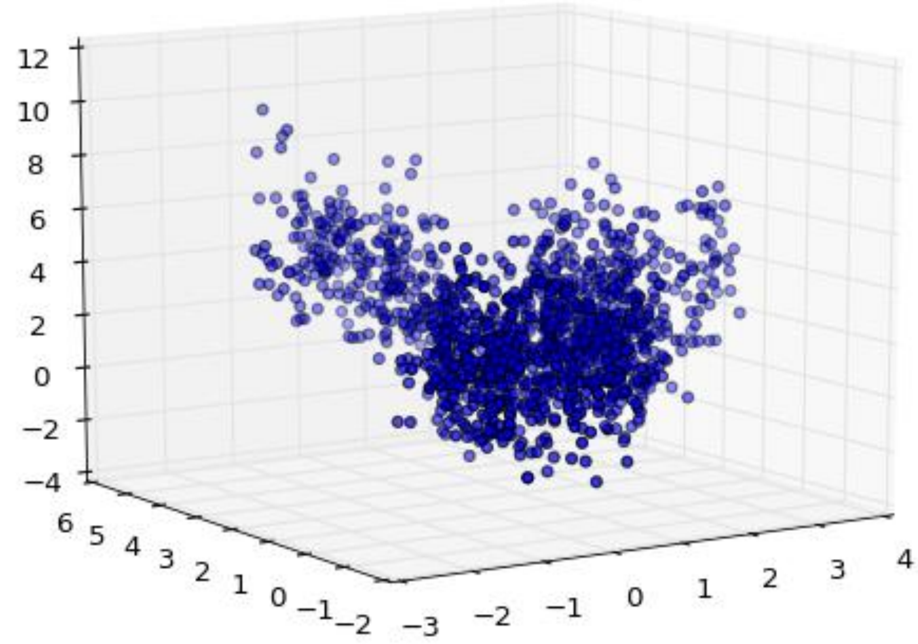



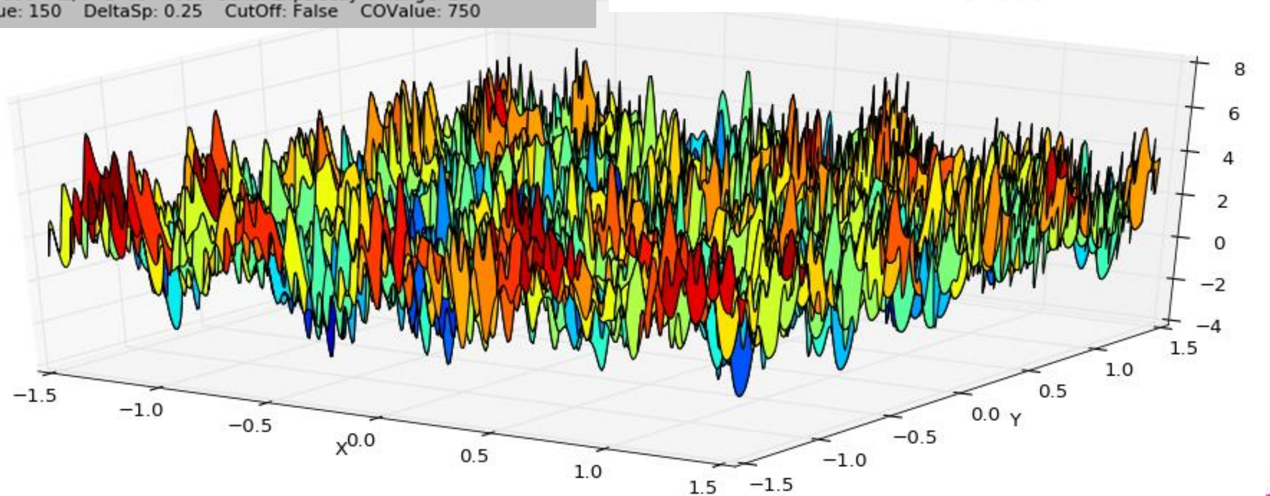



# SA: Auxiliary Graphs – Histogram & Plots





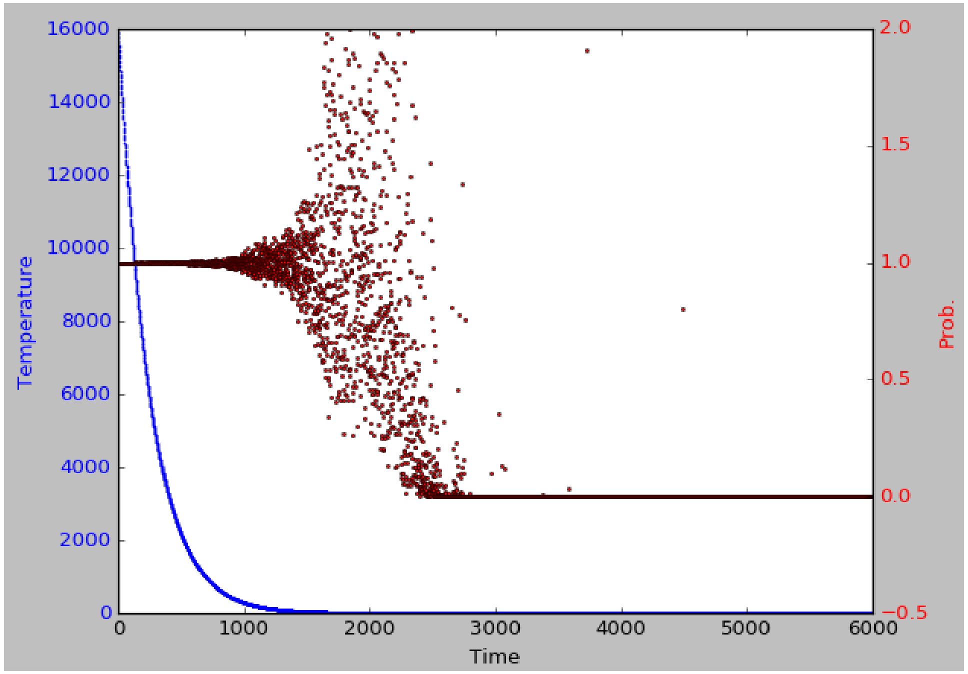



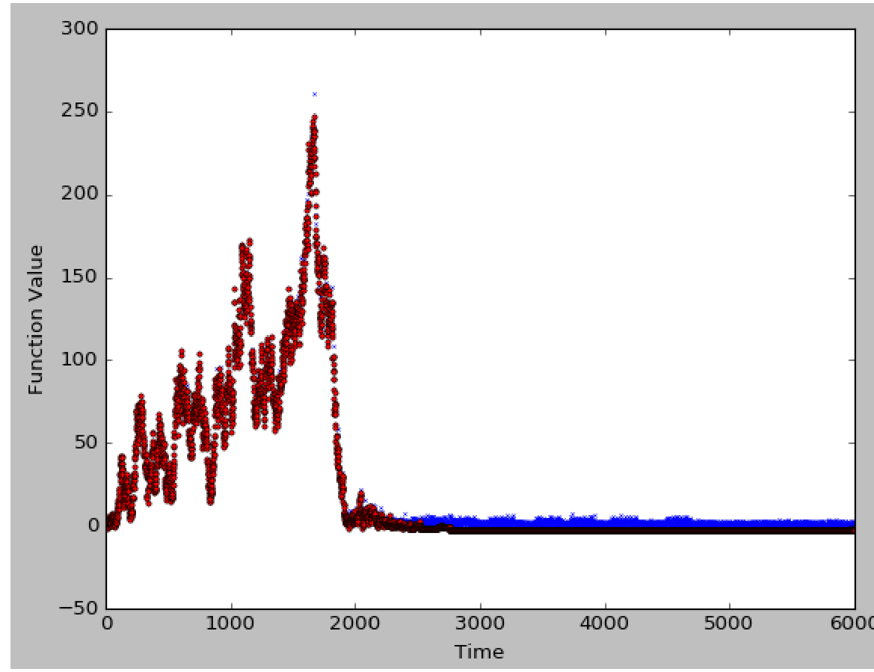



# SA – Choosing the Acceptance Function h(ΔE):

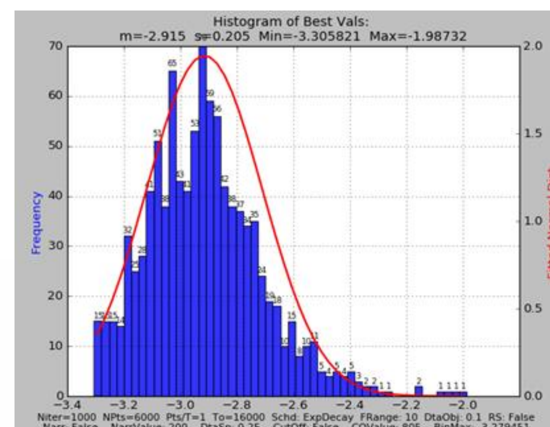
## Metropolis Criterion – Uniform Sampling:

*Graphs show behavior for 1 iteration.




```
Prob = math.exp(-(fn-fbest) / T)
```



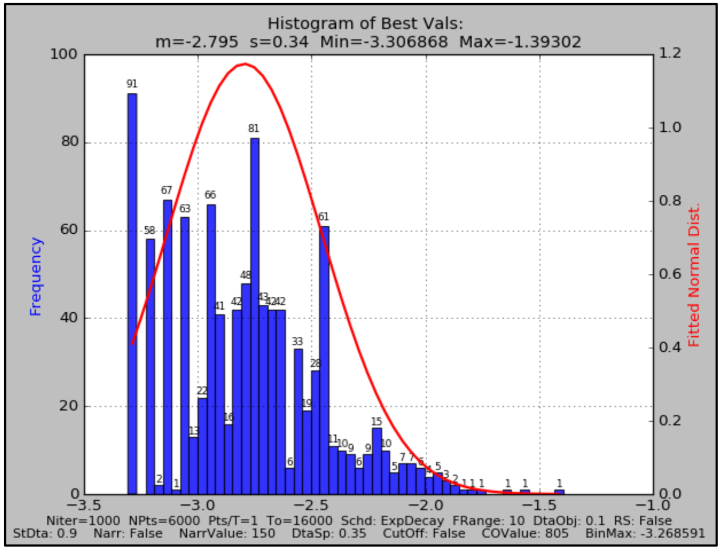

**Notes:**

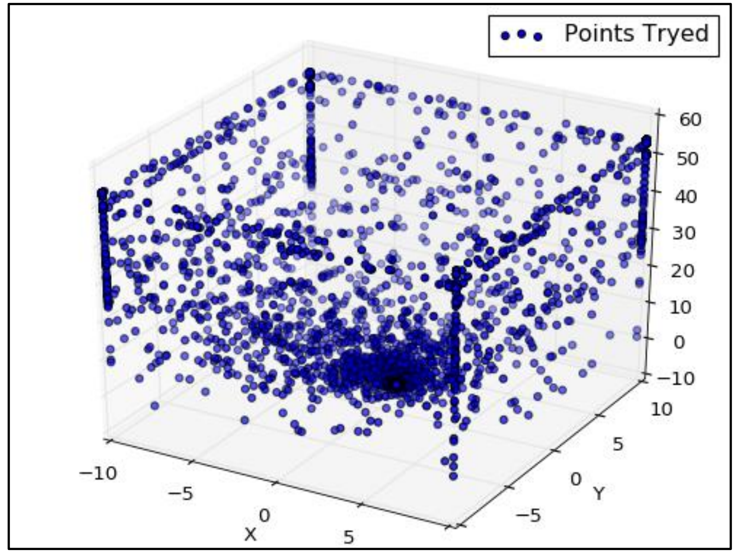
- Fn is the new Value;
- Fbest is the value computed in the prior iteration;
- Note that P is not normalized and a greater number of "bad" points end up being accepted;

*Histogram is showing the smallest value of a 1000 iterations (runs): -3.30582099378.

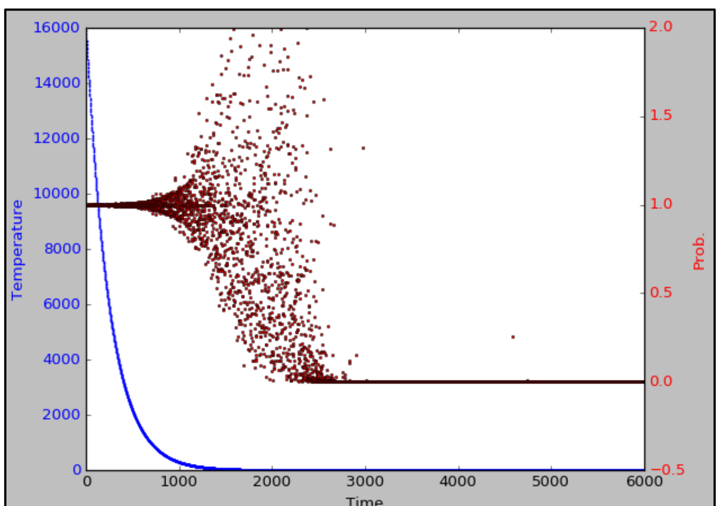
# SA – Choosing the Acceptance Function h(ΔE):

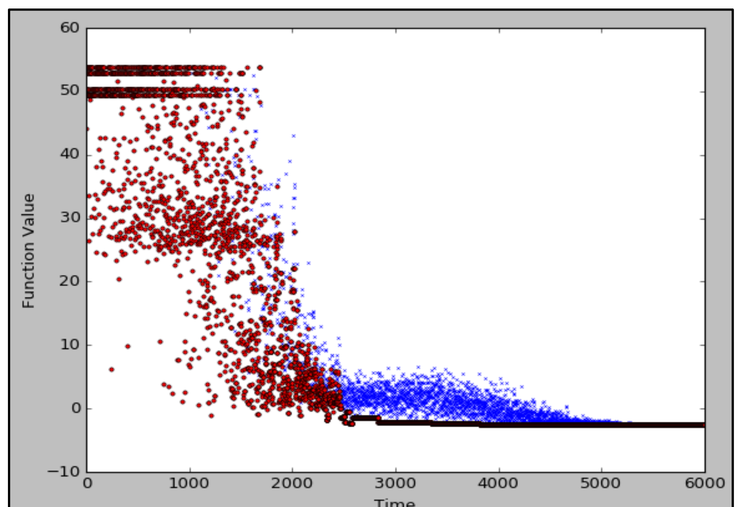
## Metropolis Criterion – Gaussian Sampling:




*The acceptance function does not impact how well the state space is searched.




*In Gaussian Sampling the points tried narrow with time.

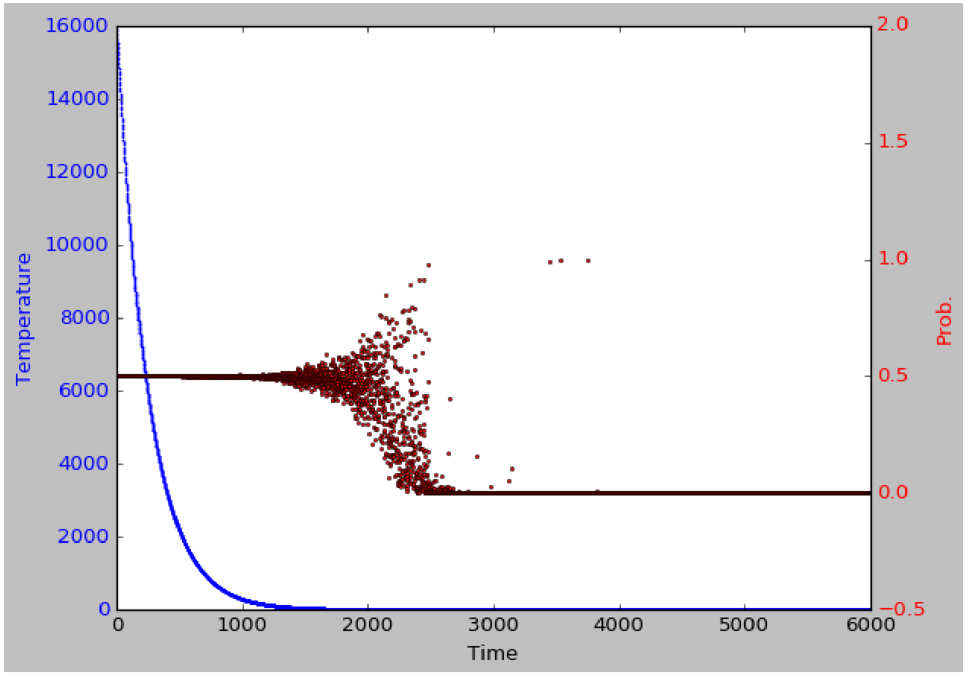
*These graphs show the behavior for 1 iteration.

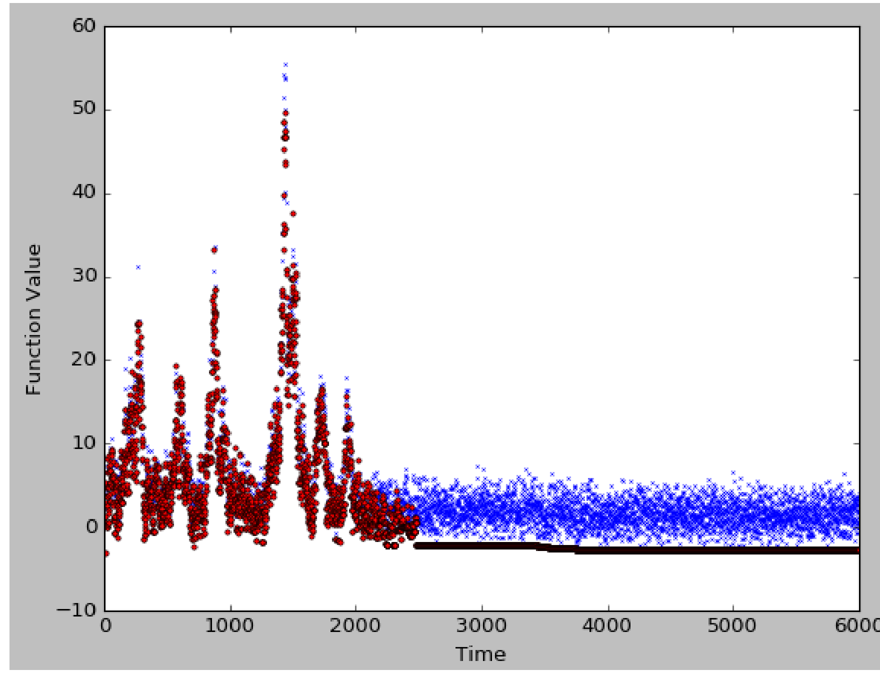
*All graphs produced wih Gaussian Sampling.

# SA - Choosing the Acceptance Function h(ΔE):

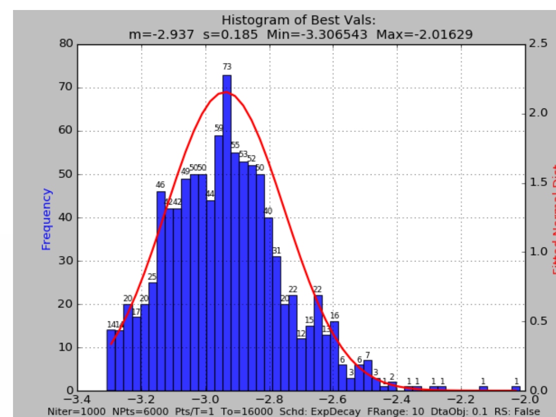
## Barker Criterion (Sigmoid) / Uniform Samp.:

*Graphs show behavior for 1 iteration.




```
Prob = 1/(1+math.exp((fn - fbest) / T))
```




**Notes:**

- The Sigmoid was chosen because it normalizes the probability;
- Note that in the beginning it accepts “bad” points with only a 50% chance;

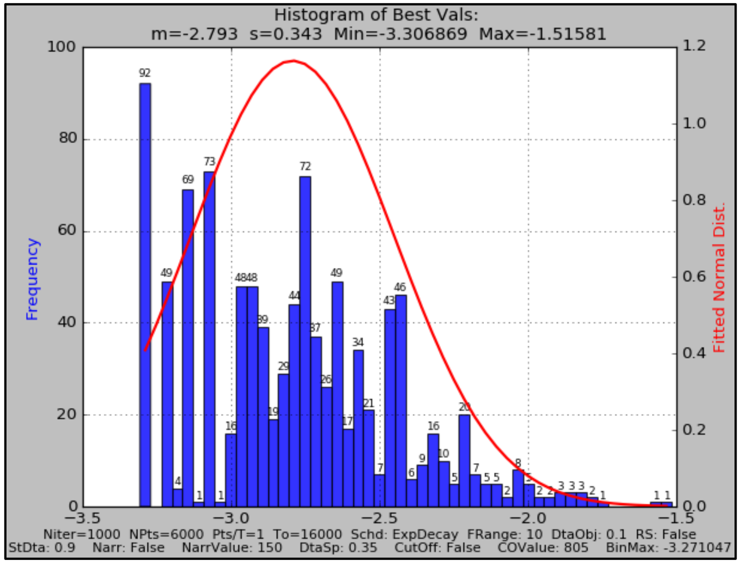
*For this comparison State Space Narrowing was off;

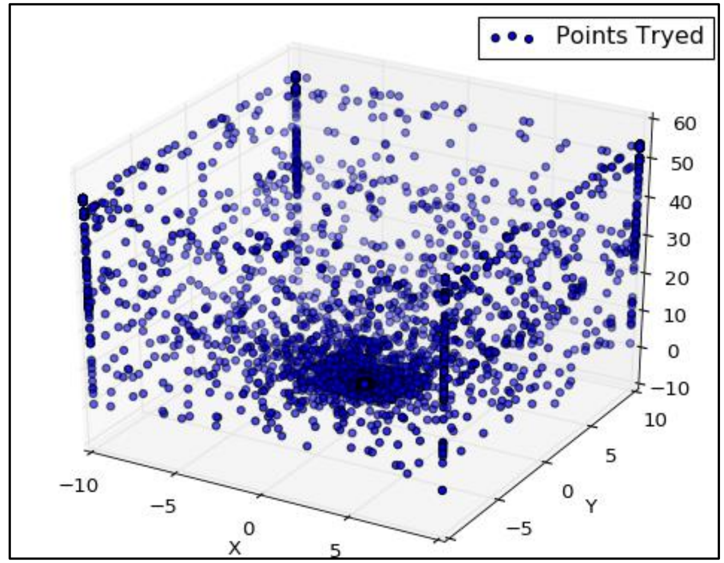
*Lowest value found in 1000 iterations: -3.30654299214.

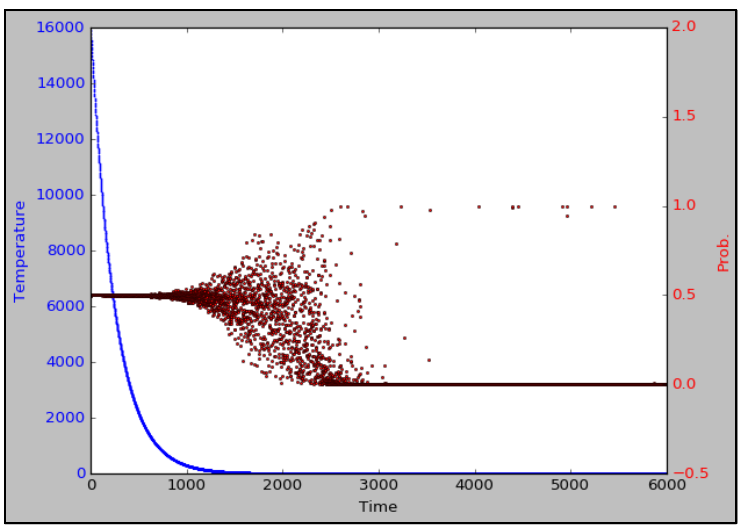
*T is Temperature not time.

# SA - Choosing the Acceptance Function h(ΔE):

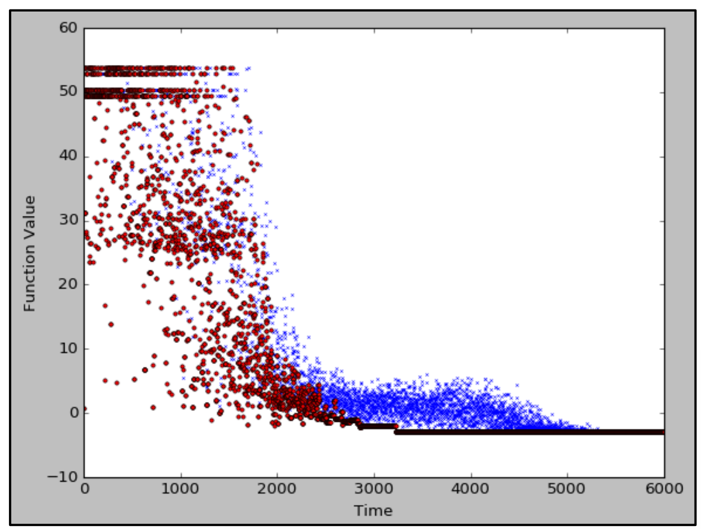
## Barker Criterion – Gaussian Sampling:




*These graphs show the behavior for 1 iteration.




*All graphs produced wih Gaussian Sampling.

*Although the Probability graph is normalized, no major difference is caused by the Barker criterion on either type of sampling.

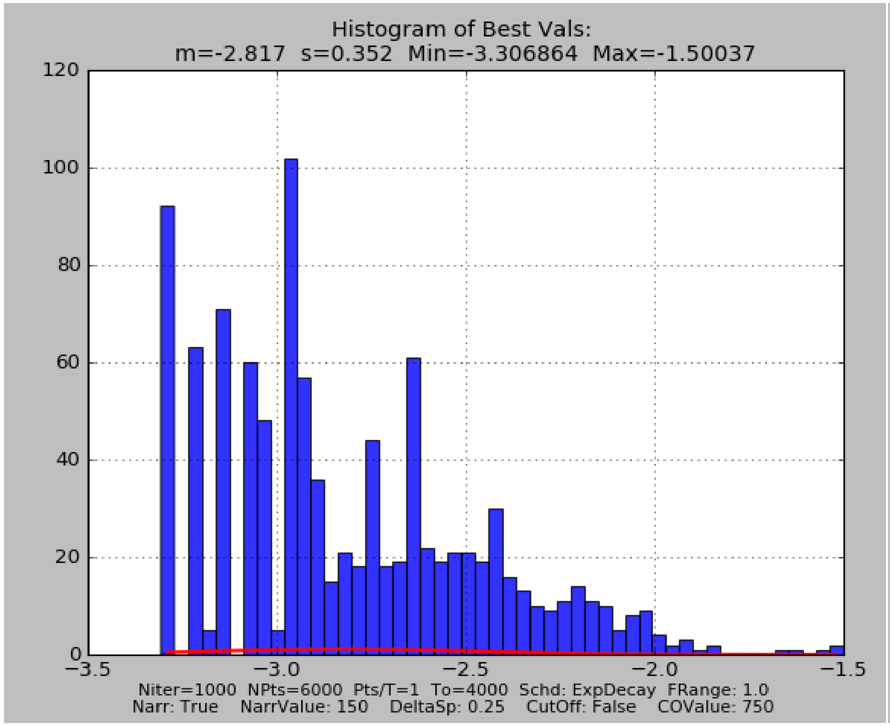
# SA: State Space Size Study (= 1)

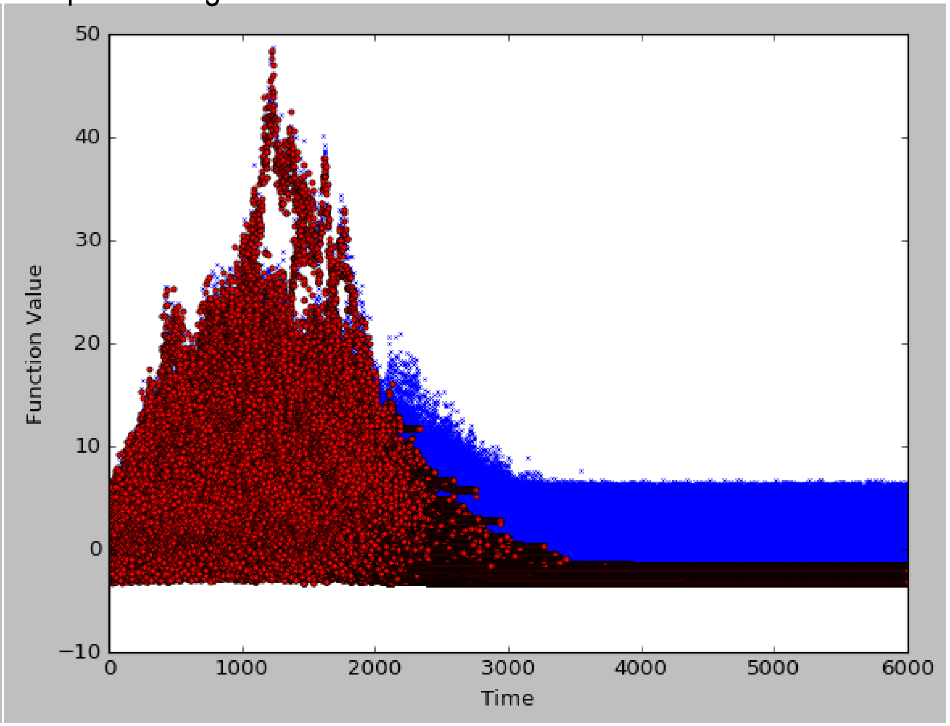


To = 4000, Npoints = 6000, **ObjRange = 1**, Niter=1000:

xn = xbest + random.uniform(-0.25*OBJfunctionRANGE, 0.25*OBJfunctionRANGE)

*Graph showing behavior for a 1000 iterations.




Time(s): 125.220999956

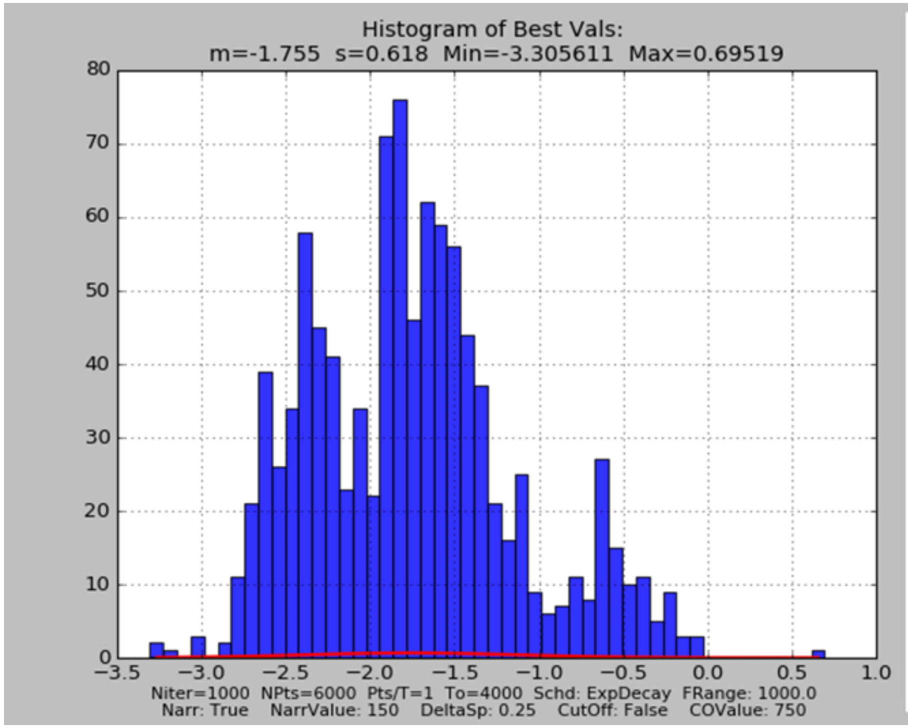
Best Solution: [-0.0244381315916016620, 0.210596769344324470]

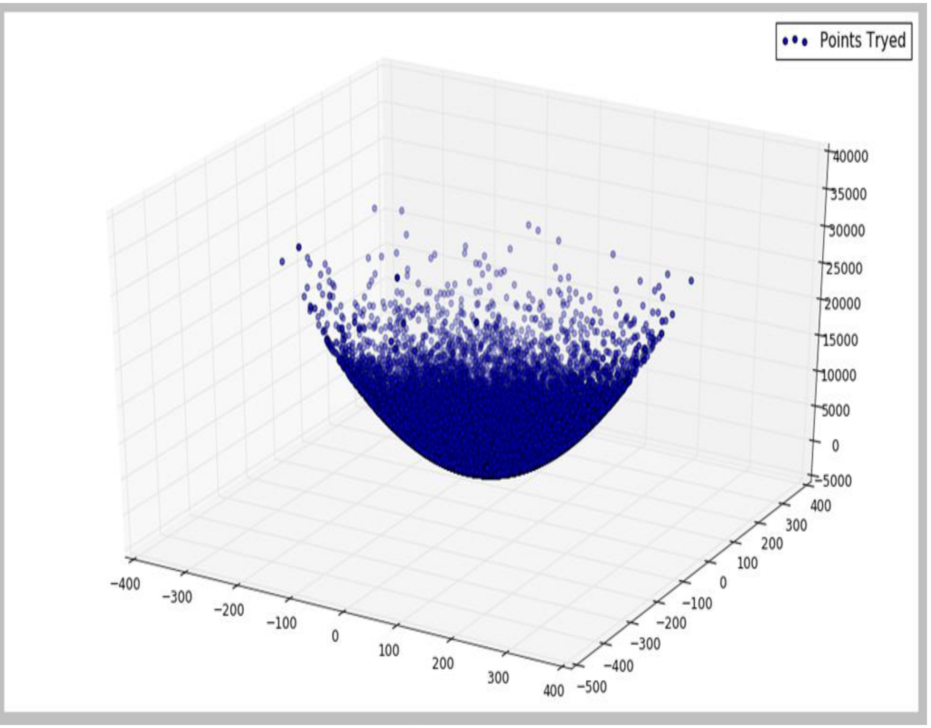
Best Minimum: -3.30686370834

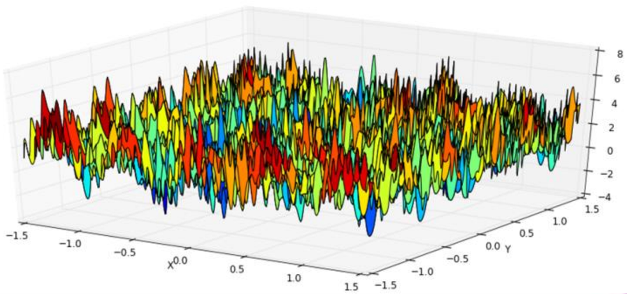
# SA: State Space Size Study (= 1000)

To = 4000, Npoints = 6000, **ObjRange = 1000,** Niter=1000:

xn = xbest + random.uniform(-0.25*OBJfunctionRANGE, 0.25*OBJfunctionRANGE)



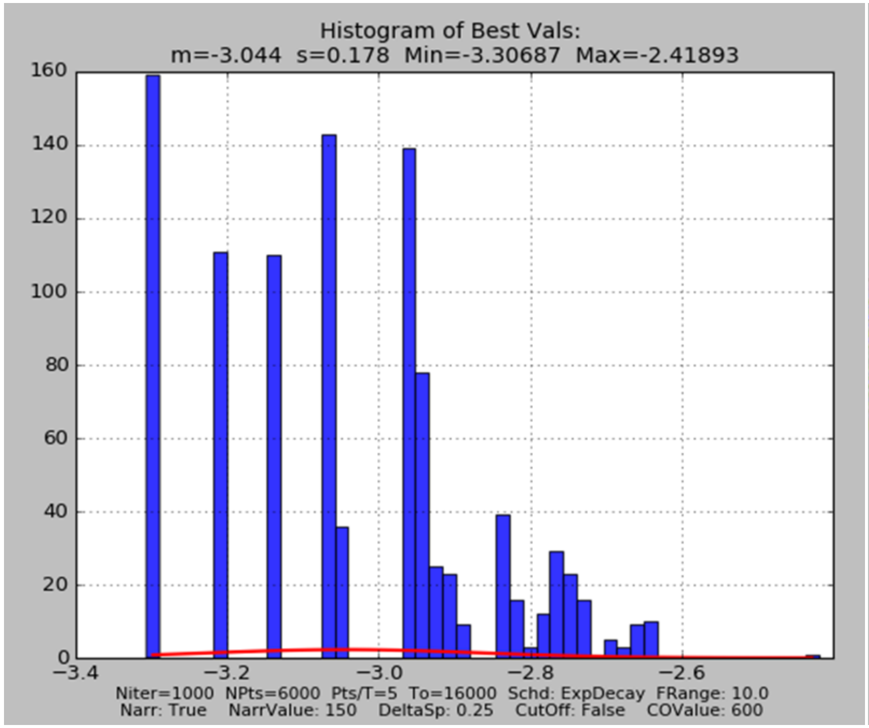



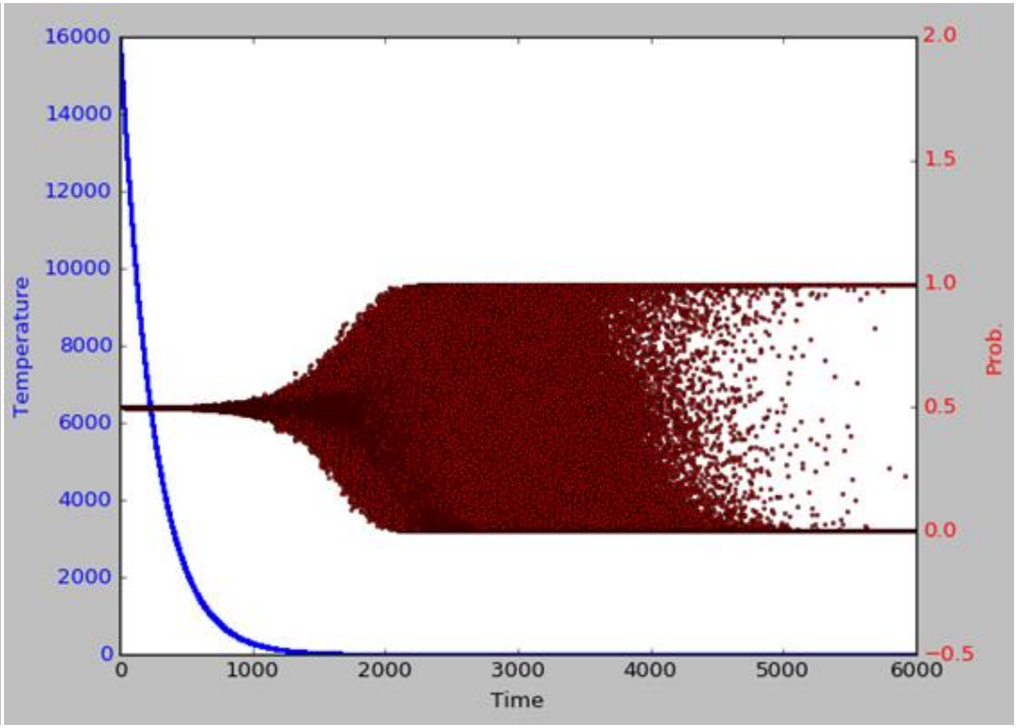
Time (s): 133.312000036
Best Solution: [-0.023821156890471266, 0.21082867317163206]
Best Minimum: -3.30561079366

With the State Space = 1000 we can see that the function has an overall parabolic shape.

To = 16000, Npoints = 6000, ObjRange = 10, Niter=1000, **Pts/T = 5**:

xn = xbest + random.uniform(-0.25*OBJfunctionRANGE, 0.25*OBJfunctionRANGE)



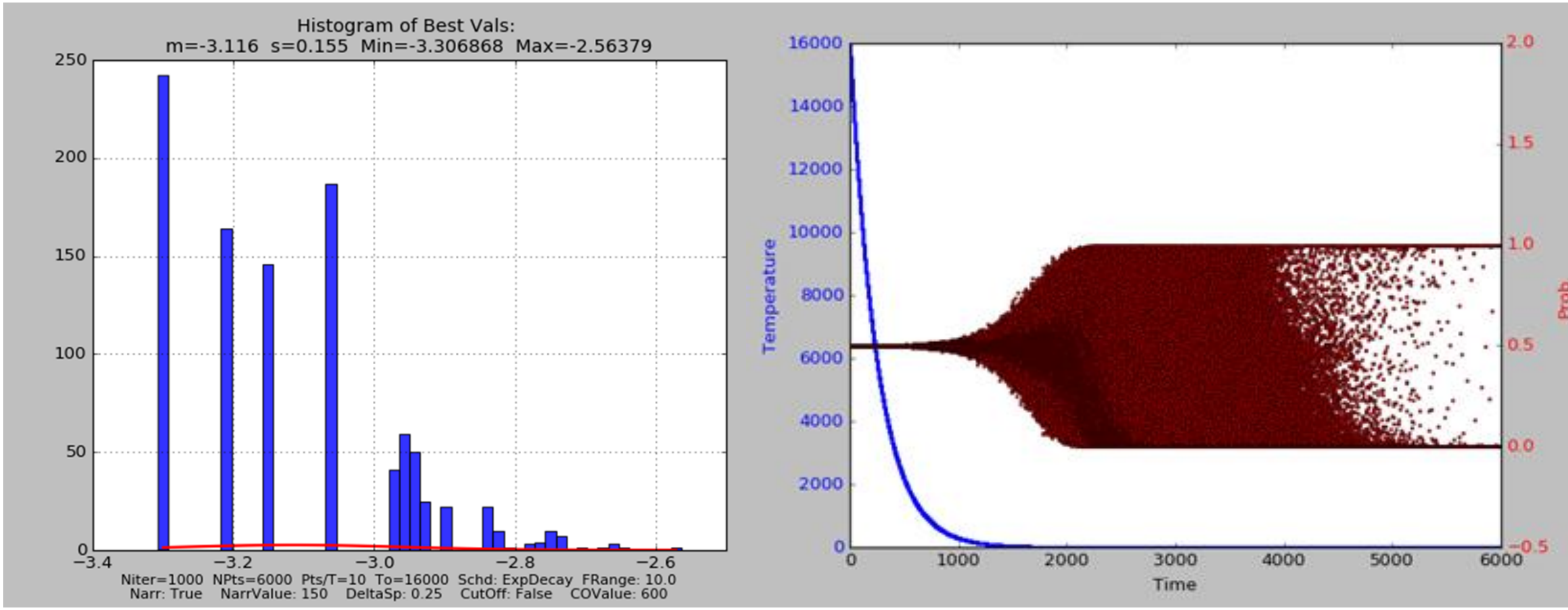




***The State Space Size of 10 was chosen for the parameter tests.**

# SA: Points Per Temperature (=10)

To = 16000, Npoints = 6000, ObjRange = 10, Niter=1000, **Pts/T = 10**:

xn = xbest + random.uniform(-0.25*OBJfunctionRANGE, 0.25*OBJfunctionRANGE)

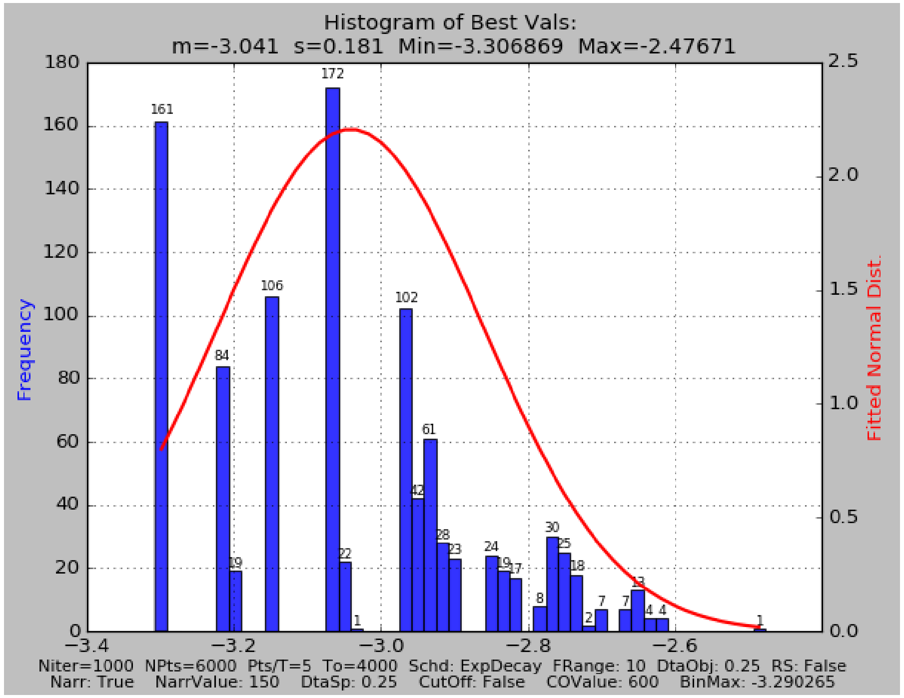


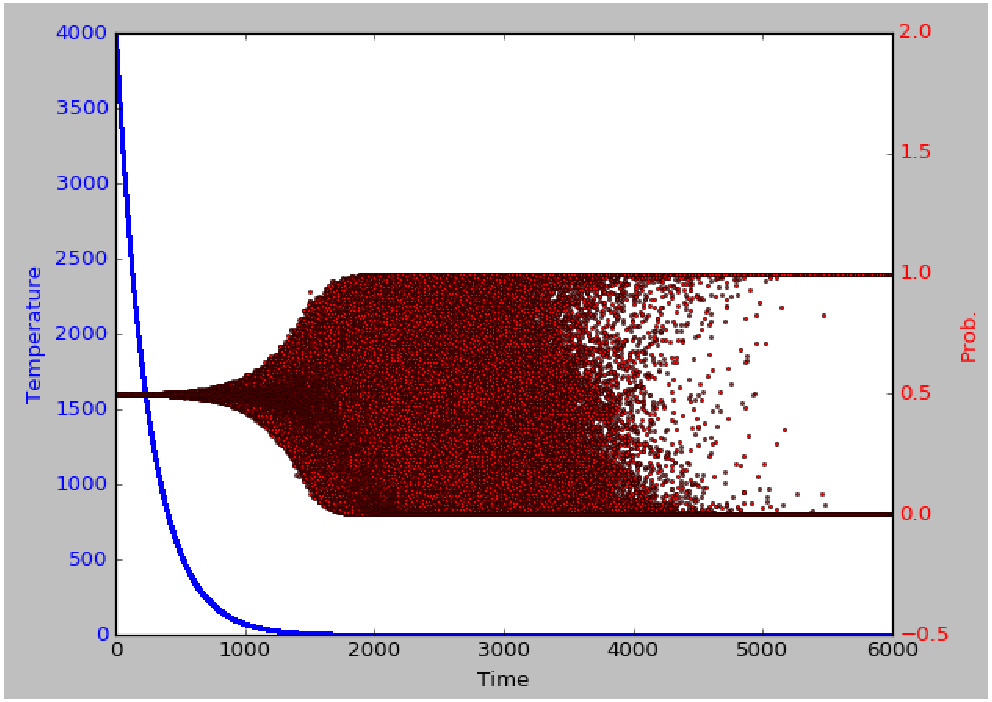

Time (s): 1065.08799982
Best Solution: [-0.024392895474989008, 0.2106125645880147]
Best Minimum: -3.3068683371

As expected, increasing the number of points per temperature tends to increase the number of times the global minimum is found, but at a cost in computational time.

## **To = 4000**, Npoints = 6000, ObjRange = 10, Niter=1000, Pts/T = 5:

xn = xbest + random.uniform(-0.25*OBJfunctionRANGE, 0.25*OBJfunctionRANGE)




Time (s): 902.938999891
Best Solution: [-0.024398969644433215, 0.2106135813698442]
Best Mínimum: -3.30686858992

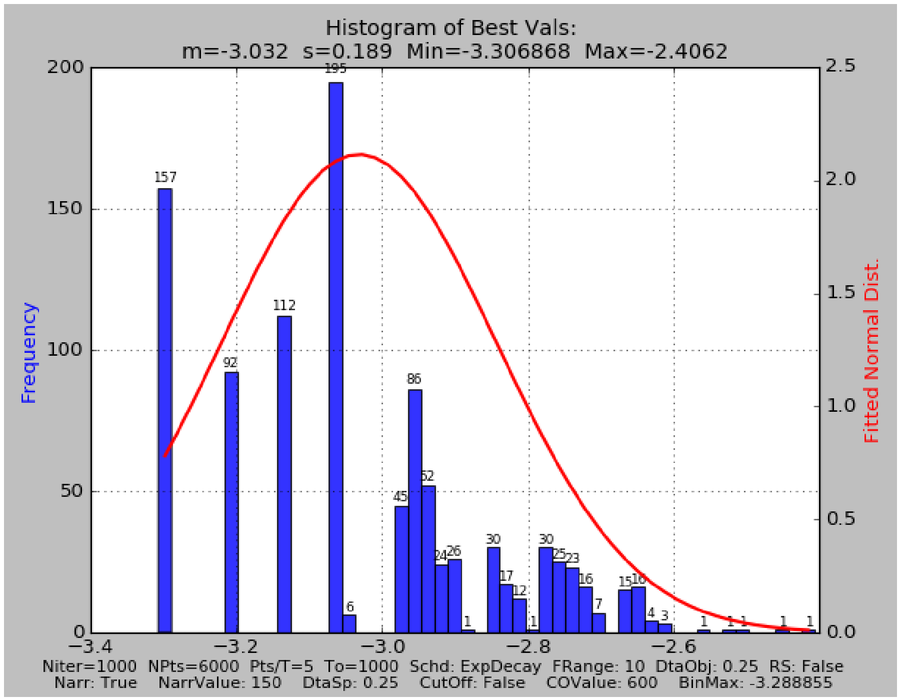
Lowering$T_0$ decreases the exploration phase of the agorithm.

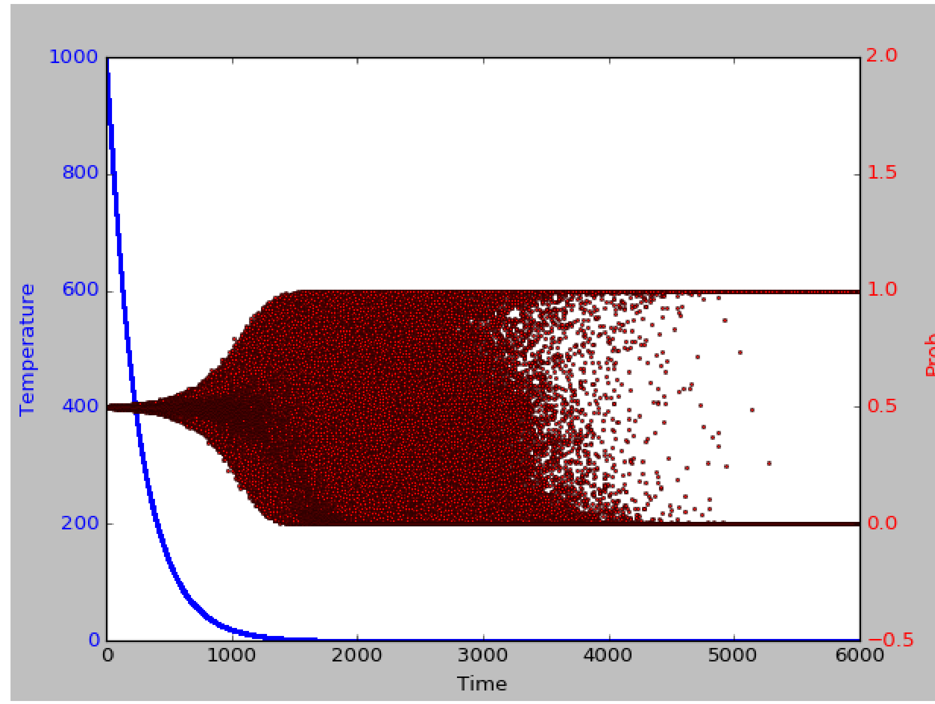
**Observação:**

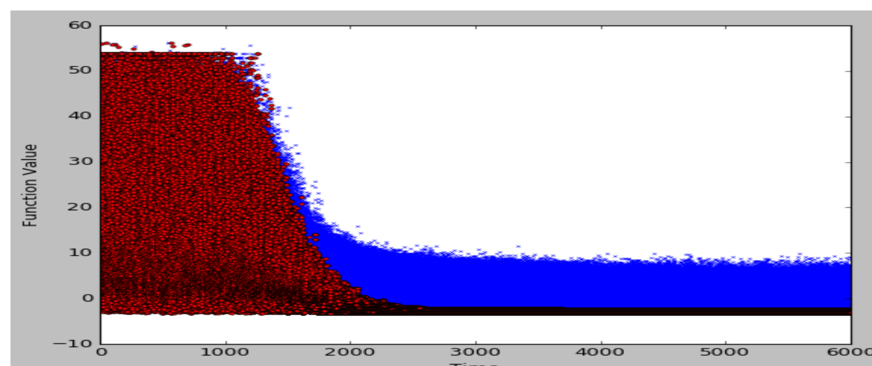
- Compare com os slides 16 e 19 (To = 16000, 1000).

**To = 1000**, Npoints = 6000, ObjRange = 10, Niter=1000, Pts/T = 5:

xn = xbest + random.uniform(-0.25*OBJfunctionRANGE, 0.25*OBJfunctionRANGE)



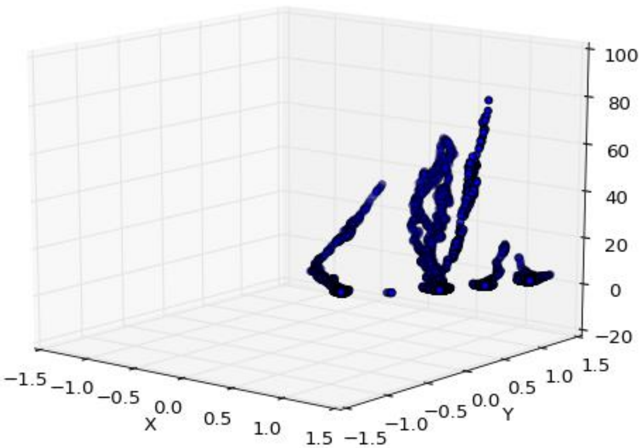

Time (s): 903.782000065

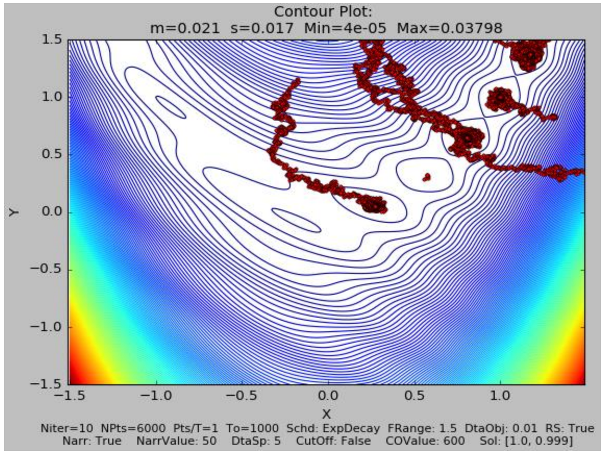
Best Solution: [-0.02441046481725372, 0.2106151431572286]

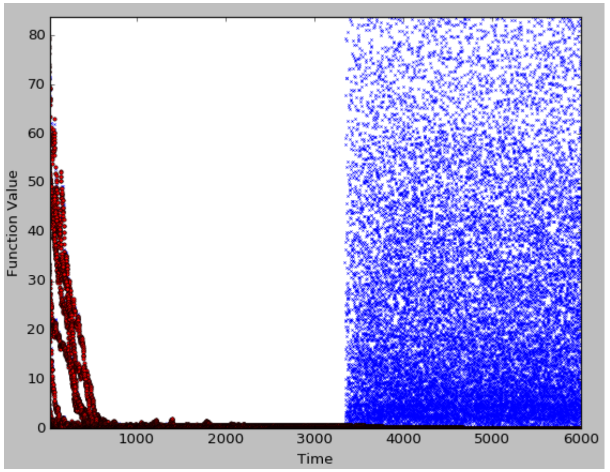
Best Minimum: -3.30686844982

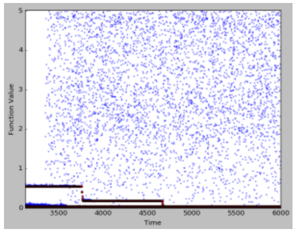
We can observe in the figures on the right, how lowering $T_0$ decreased the exploration phase. The lower $T_0$ makes the probability function converge to 0 faster.

# SA: Using SA as Stochastic Gradient Descent

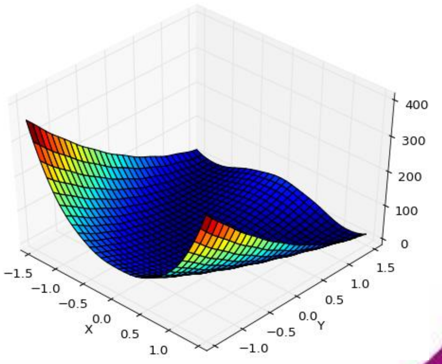


- If you use a small enough step size (DeltaOBjF), you can make SA behave like a Stochastic Gradient Descent.
- In this case, it is interesting to assign a high value to DeltaSp. When the SA finds a local minimum, the high DeltaSp opens the search space to try to find another minimum with a lower value then the one currently found.
- The Recursive Delta (RcDelta) can be used to approximate or spread the starting point of each iteration. (In this case a high Recursive Delta was used to show the distinct path of each iteration).
- Note in the graphs above that, each iteration goes directly to the nearst minimum found. To doesn't interfere a lot because the exploration ends up being limited by the small step size.




| Func | Niter | Npts | Pts/T | To | Sched | Frange | DeltaObjF | Narr | NarrVal | DeltaSp | CutOff | COValue |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| F | 10 | 6000 | 1 | 1000 | ExpDecay | 1.5 | 0.01 | TRUE | 50 | 5 | FALSE | 600 |
| **Min** | **Nmin** | **Recur.** | **RcDelta** | **Time** | **FixSeed** | **Mu** | **Sigma** | **Max** | **Sol** | | **SBin** | |
| 4.03E-05 | 4 | TRUE | 0.9 | 2.03 | FALSE | 0.0211 | 0.017361 | 0.04 | [1.000, 0.999] | | [ 4.0e-05 | 7.9e-04] |

# SA: Parameter Study



**The table below shows some of the entries in the experiment Log Book created:**

| Niter | Npts | Pts/T | To | Sched | Frange | DeltaObjF | Narr | NarrVal | DeltaSp | CutOff | COValue | Min | Nmin | Recur. | Tempo | Obs 1 | Obs 2 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1000 | 6000 | 1 | 16000 | To*exp(-L*t) | 10 | 0.1 | F | | | | | -3.30272 | 18 | | | | |
| 1000 | 6000 | 1 | 16000 | To*exp(-L*t) | 10 | 0.15 | F | | | | | -3.30586 | 9 | | | | DeltaObjF |
| 1000 | 6000 | 1 | 16000 | To*exp(-L*t) | 10 | 0.5 | F | | | | | -3.2991 | 4 | | | | |
| 1000 | 6000 | 1 | 16000 | To*exp(-L*t) | 10 | 0.25 | F | | | F | | -3.304316 | 6 | | | StateSpNarr | DeltaObjF |
| 1000 | 6000 | 1 | 16000 | To*exp(-L*t) | 10 | 0.25 | T | 150 | 0.25 | F | | -3.30686 | 62 | | | | |
| 1000 | 6000 | 1 | 16000 | To*exp(-L*t) | 10 | 0.25 | T | 200 | 0.25 | F | | -3.30683 | 65 | | | | |
| 1000 | 6000 | 1 | 16000 | To*exp(-L*t) | 10 | 0.25 | T | 150 | 0.15 | F | | -3.30686 | 43 | | | | |
| 1000 | 6000 | 5 | 16000 | To*exp(-L*t) | 10 | 0.25 | T | 150 | 0.25 | F | | -3.30687 | 160 | | | | Pts/T=5 |
| 1000 | 6000 | 1 | 4000 | To*exp(-L*t) | 1 | 0.25 | T | 150 | 0.25 | F | | -3.306864 | 92 | | 125.2 | | |
| 1000 | 6000 | 1 | 4000 | To*exp(-L*t) | 1000 | 0.25 | T | 150 | 0.25 | F | | -3.305611 | 2 | | 133.3 | | |
| 1000 | 6000 | 10 | 16000 | To*exp(-L*t) | 10 | 0.25 | T | 150 | 0.25 | F | | -3.306868 | 245 | | 1065 | Cutoff | Pts/T=10 |
| 1000 | 6000 | 10 | 16000 | To*exp(-L*t) | 10 | 0.25 | T | 150 | 0.25 | **T** | 450 | -3.306853 | 72 | | 504.9 | | |
| 1000 | 6000 | 10 | 16000 | To*exp(-L*t) | 10 | 0.25 | T | 150 | 0.25 | **T** | 750 | -3.306862 | 225 | | 598.2 | | |
| 1000 | 6000 | 1 | 16000 | To*exp(-L*t) | 10 | 0.25 | F | | | | | 3.300611 | 885 | T | 158.2 | StDelta = 0 | |
| 1000 | 6000 | 1 | 16000 | To*exp(-L*t) | 10 | 0.25 | F | | | | | -3.306832 | 100 | T | 159 | StDelta = 0.005 | |
| 1000 | 6000 | 5 | 1 | To/(1+log(1+t)) | 10 | 0.25 | F | | | F | | -3.306533 | 12 | | 720.1 | | |
| 1000 | 6000 | 5 | 1 | To/(1+log(1+t)) | 10 | 0.25 | T | 150 | 0.25 | F | | -3.306866 | 374 | | 720 | | |

**CONCLUSIONS (*Above examples used uniform sampling *g(Δx)*):**

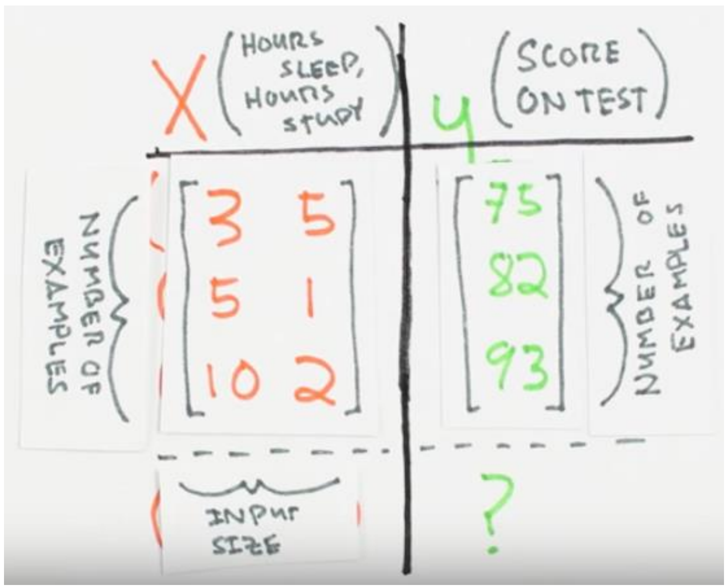
- Delta Objective Function: Smaller step sizes tend to transform the SA in a Gradient Descent. The ideal balance needs to be found for each fuction.
- State Space Narrowing is effective. An improvement of 10x was found, from 0.6% to ~ 6% in the number of times the global minimum was found;
- Cutoff: Is effective to reduce the computational time but you need to choose wisely the cuttoff value (ideal ~750 a 800);
- Points / temperature: Increases the rate the global is found but with a cost in execution time;
- Recursive Application: It can be interesting to apply some random noise so that we don't always find the same local minimum.
- An increase in $T_0$ shifts the probability graph to the right increasing the exploration phase.

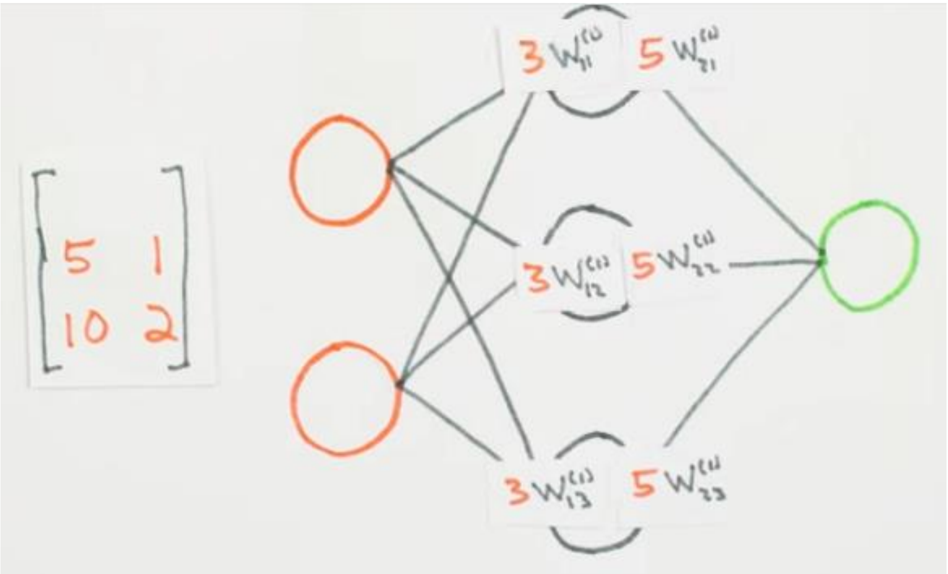
# SA: Cooling Schedules



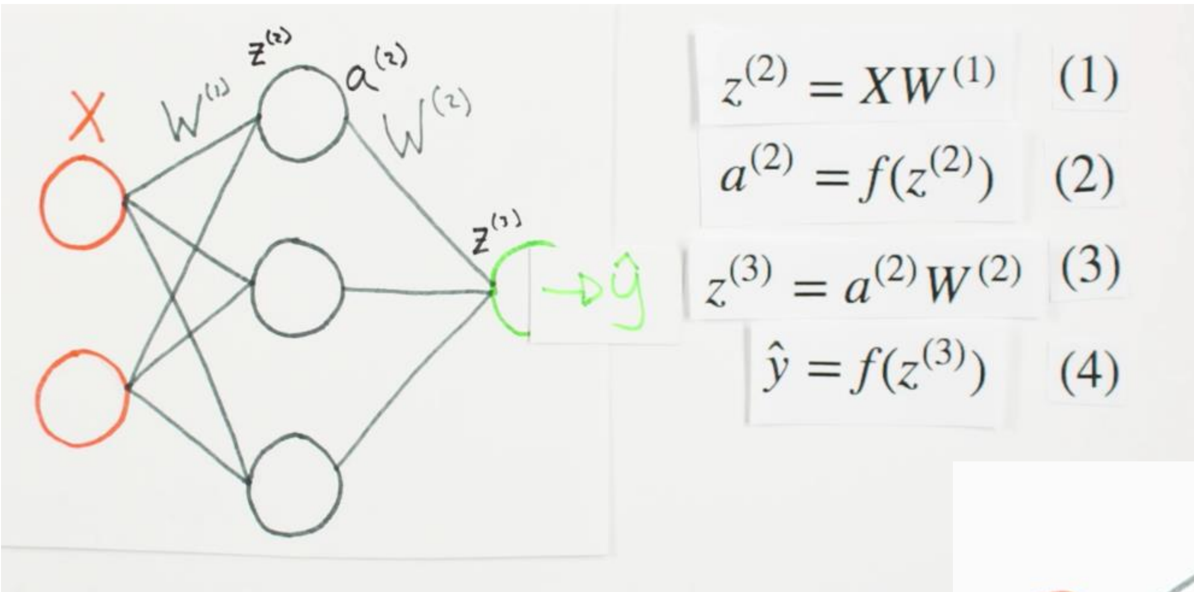
```python
def PickSched(SchedType = 'ExpDecay', limit = 4000):
    global T0
    limit += 1
    if SchedType == 'ExpDecay':
        To = 16000.0
        lam = 0.004
        SchedFunc = lambda t: (To * math.exp(-lam * t) if t < limit else 0)
    elif SchedType == 'ConvCond-Log':
        To = 1.0
        SchedFunc = lambda t: (To/(1+math.log(1+t)) if t < limit else 0)
    elif SchedType == 'ExpMultiplicative':
        To = 10000.0
        SchedFunc = lambda t: (To * 0.996**t) if t < limit else 0
    elif SchedType == 'LogMultip':
        To = 35.0
        Alpha = 1.1 # Alpha>1
        SchedFunc = lambda t: (To/(1+Alpha*math.log(1+t)) if t < limit else 0)
    elif SchedType == 'Logarithmic':
        To = 1.0
        SchedFunc = lambda t: (To/(math.log(t)) if t < limit else 0)
    elif SchedType == 'LinearMultip':
        To = 35.0
        Alpha = 2  # Alpha>0
        SchedFunc = lambda t: (To/(1+Alpha*t) if t < limit else 0)
    elif SchedType == 'QuadraticMultip':
        To = 35.0
        Alpha = 2  # Alpha>0
        SchedFunc = lambda t: (To/(1+Alpha*t**2) if t < limit else 0)
    elif SchedType == 'LinearInvTime':
         To = 1.0
         SchedFunc = lambda t: (To/t if t < limit else 0)
    global TSCHEDULE
    TSCHEDULE = SchedType
    T0 = To   #T0 is the initial temperature which is recorded in the .csv file
    return SchedFunc
```

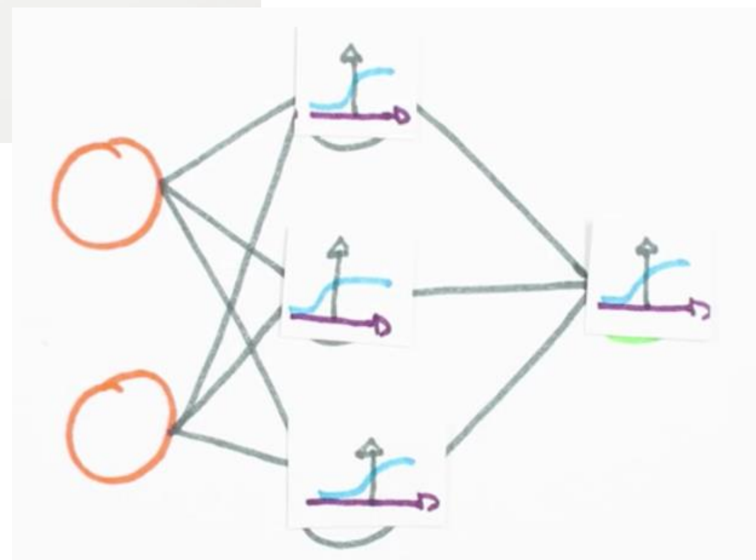
- Supervised Regression Problem
- Initial problem with two layers, 2 neurons in the visible layer and 3 neurons in the hidden layer;
- This neural network will be used to predict the score of a student in a test given the number of hours he slept and the number of hours he studied.



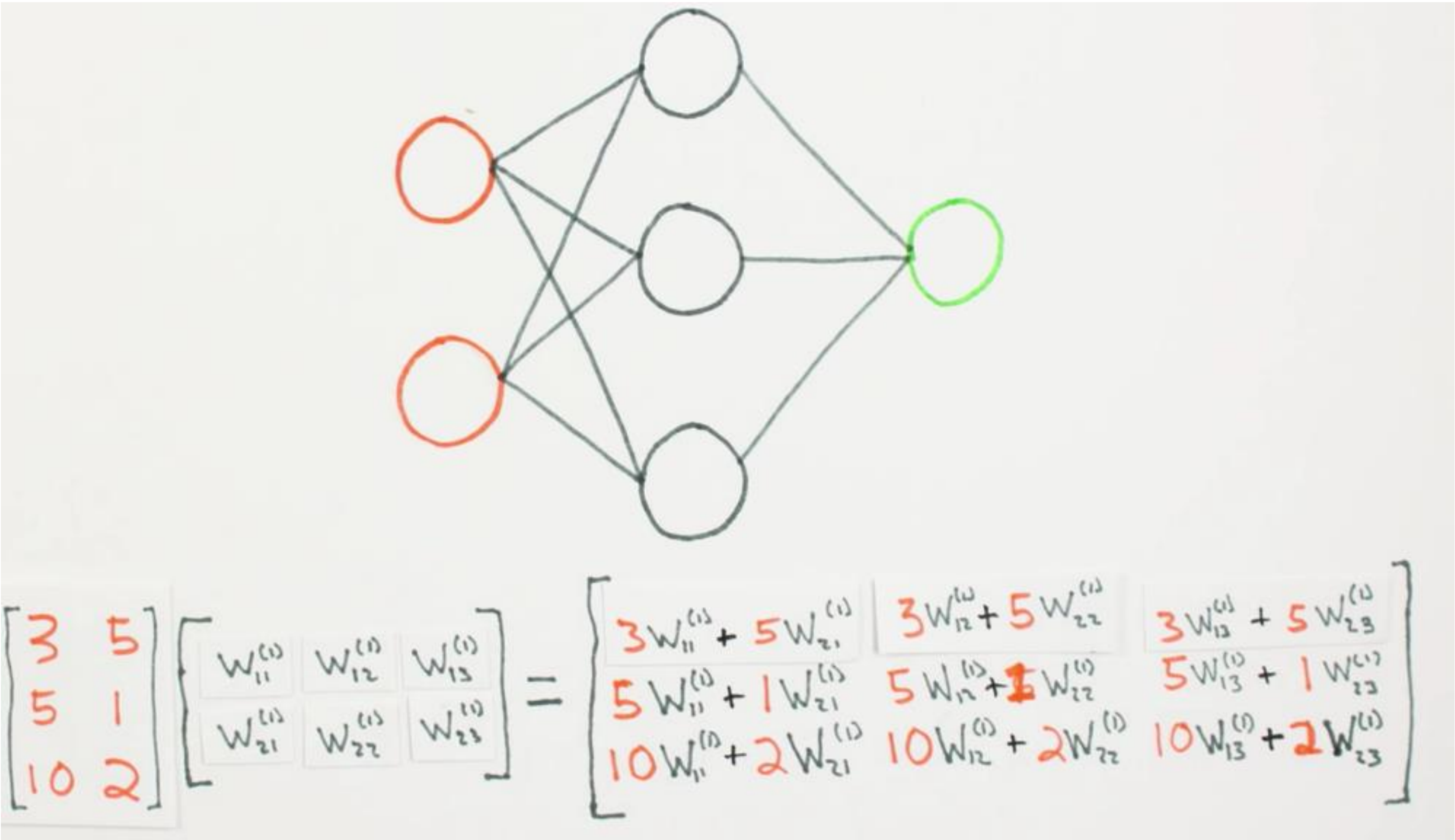



## Building the prediction equations:

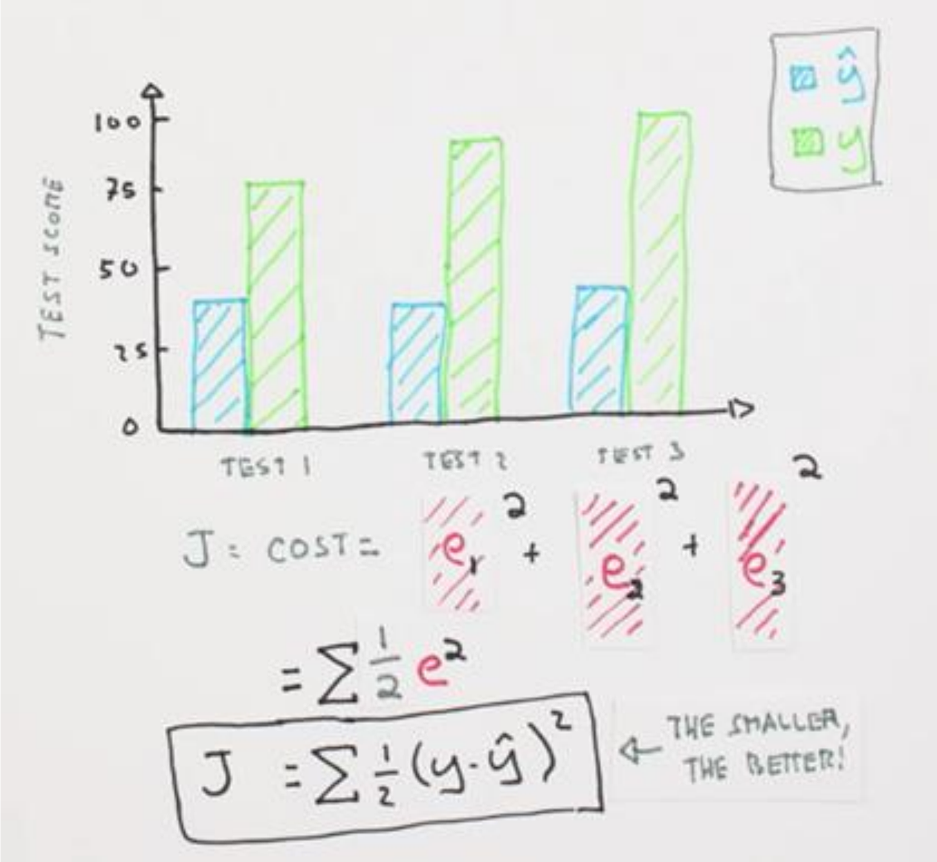


## Building the prediction equations (Continued):



## Building the error equation:



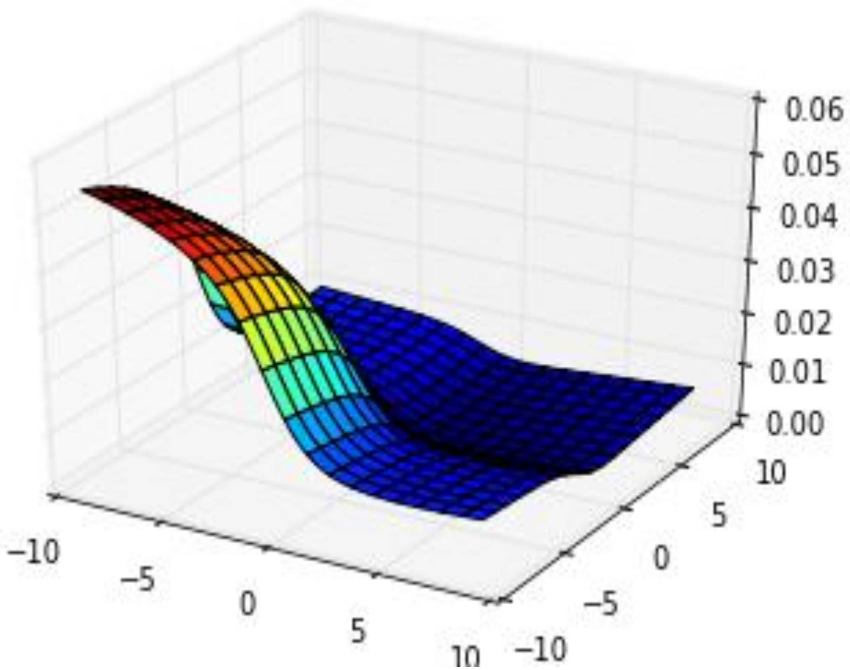
# Objective Function to be Optimized:

```
def sigmoid(self, z):
    # Apply sigmoid activation function to scalar, vector, or matrix
    return 1 / (1 + np.exp(-z))

def forward(self, X):
    # Propogate inputs though network
    self.z2 = np.dot(X, self.W1)  # multiplica a input layer pelos pesos
    self.a2 = self.sigmoid(self.z2)  # aplica a activation function
    self.z3 = np.dot(self.a2, self.W2) # multiplica a secunda camada layer pelos pesos W2
    yHat = self.sigmoid(self.z3)  # aplica a segunda activation function e calcula a previsao
    return yHat

def costFunction(self, X, y):
    # Compute cost for given X,y, use weights already stored in class.
    self.yHat = self.forward(X)
    J = 0.5 * sum((y - self.yHat) ** 2) / X.shape[0] + (self.Lambda / 2) * (
    np.sum(self.W1 ** 2) + np.sum(self.W2 ** 2))
    #print(J)
    return float(J)
```

# SA: Training a Neural Network - Challenges




```
Nota na Prova real normalizada:
 [[ 0.75]
 [ 0.82]
 [ 0.93]]
Previsao depois da Otimizacao de Annealing:
 [[ 0.80226458]
 [ 0.85876793]
 [ 0.88449846]]
∑(yHat-y)^2: 0.00630492879233


Nota na Prova real normalizada:
 [[ 0.75]
 [ 0.82]
 [ 0.93]]
Previsao depois da Ot. com Gradiente e com training data:
 [[ 0.75072298]
 [ 0.82699527]
 [ 0.91334669]]
∑(yHat-y)^2: 0.000326789084183
```

**Challenges:**

- Objective Function has many plateaus (as shown above);

**Solution:**

- Explore the State Space with SA;
- Apply a Gradient Descent Method (BFGS or Conjugate Gradient);
- For this problem, the combination of both methods yields a result better than using each method alone.

# SA: Training a Neural Network – MNIST Data Set

## Challenges:

Training set with 50.000 images
Test set with 10.000 images
Size: 28x28 = vector com 784 dimensions

Input Layer Size: 784 Hidden Layer: 50 Output Layer: 10

$Z^2$ has dimension $X \bullet W^1$ = [50.000 x 784] • [784 x 50] = [50.000 x 50]
$Z^3$ has dimension $a^2 \bullet W^2$ = [50.000 x 50] • [50 x 10] = [50.000 x 10]

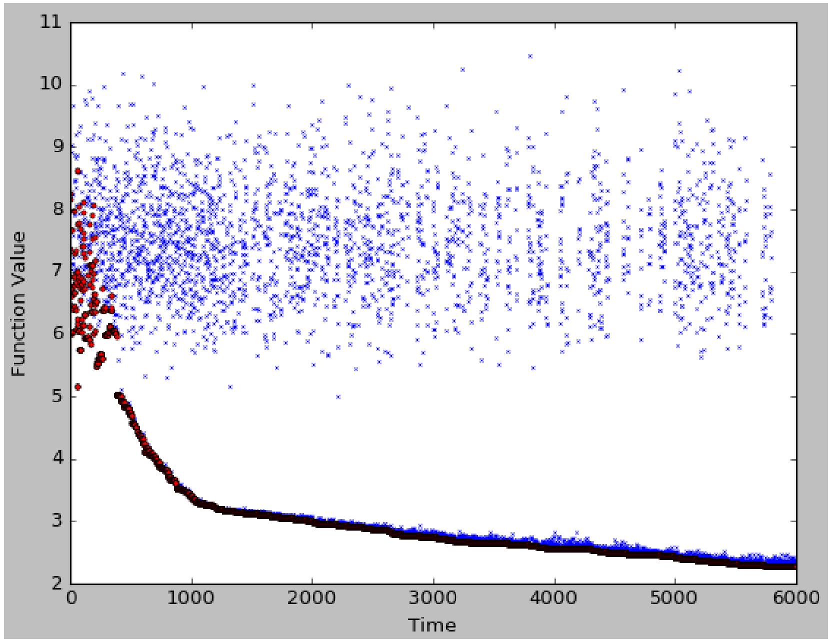
T Schedule: ExpDecay Pts/T =1 Npoints: 6000

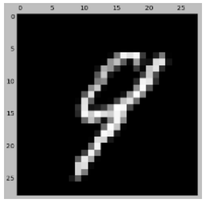
Execution Time: 4733.71(s)
Best Minimum Found with just Annealing: 2.44735099426

Gradient Method: Conjugate Gradient
Best Minimum Found with Annealing and Gradient: 0.418892021738
Execution Time: 4806.97(s)
Training set Accuracy: 93.238%
Test set Accuracy: 93.16%

Best Minimum Found with just the Gradient: 0.341415581947
Execution Time: 82.39(s)
Training set Accuracy: 94.508%
Test set Accuracy: 94.27%

## Annealing Evolution Graph:



Choose a test image index (0 to 9999): 9000
Test Label: 9
Predicted Label: 9
Both training methods correctly recognized this image, that with lower accuracies is confused with a 4.
However, SA took 58 times longer, and had lower accuracy.

# SA: Training a Neural Network – MNIST Data Set



**Parameter values used in the MNIST study and their functions:**

OBJfunctionRANGE = 6.0 *Explores the objective function from -OBJfunctionRANGE to +OBJfunctionRANGE in each xn.
DELTAObjFunc = 0.15 *Step size to the next point:
xn = xbest + random.uniform(-DeltaObjFunc*OBJfunctionRANGE, DeltaObjFunc*OBJfunctionRANGE)
WRITEtoFILE = False *Exports the result of each simulation to a .csv file.
GRAPH=0 *Creates a 3D plot of the accepted points.
GRAPHtemperSCHED = 0 *Creates a graph of the cooling schedule.
GRAPHProb = 0 *Plots the probabilities in the temperature graph.
GRAPHsurface = 0 *Creates a 3D graph of the function being optimized.
GRAPHobjValue = 1 *Plots the evolution of the cost of the objective function with time.
GRAPHcontour = 0 *Creates a contour plot of the function being optimized.
GRAPHhistogram = 0 *Creates a histogram of the results of all the iterations (simulations).
PRINTiter = False *Used for debugging. Prints parameters and results on the screen for each f(x) evaluated.
NUMpoints = 6000 *Number of random points in each iteration.
POINTSperTEMPERAT = 1 *Number of points per temperature.
NUMITER = 1 *Number of iterations to run.
APPLYcutoff = False *Turns Cutoff on and off.
CUTOFFvalue = 405 *Stops simulation after 405 evaluations with no improvement in the objective value.
STATEspaceNARROWING = True *Turns on State Space Narrowing.
STATEspaceNARvalue = 10 *Starts State Space Narrowing at each 10 points with no improvement.
DELTASpace = 0.2 *xn[nd] = xbest[nd] + random.uniform(-DELTASpace * abs(xbest[nd]), DELTASpace * abs(xbest[nd]))
TSCHEDULE = 'ExpDecay‘ *Sets which temperature schedule will be used.
T0 = 1 *Configures the initial temperature.
RECURSIVEstart = False *When ON, uses the result of this iteration as a starting point for the next iteration.
STARTdelta = 0.1 *Applies a random noise to the best result found in the previous simulation.
USEANNEAL = True *Turns on the use of Annealing to train the neural network.

## Conclusions up to now:

- Conjugate Gradient Descent is more efficient than BFGS for large dimensions.
- The computation time for the Cost Function is high for 6000 points and 50.000 images.
- I believe that SA can find a good global optimum, but with a very high computational cost, since it depends on “blind” guesses.
- Many of these guesses, which take a long time to compute, are wasted…
- Each iteration, with a method like gradient descent, goes towards a minimum in all dimensions at once. This ends up being more efficient when the gradient can be computed.
- SA seems to end up throwing the gradient descent in a local minimum that is not the best optimum. This is why in large Neural Networks it is better to use just the gradient.

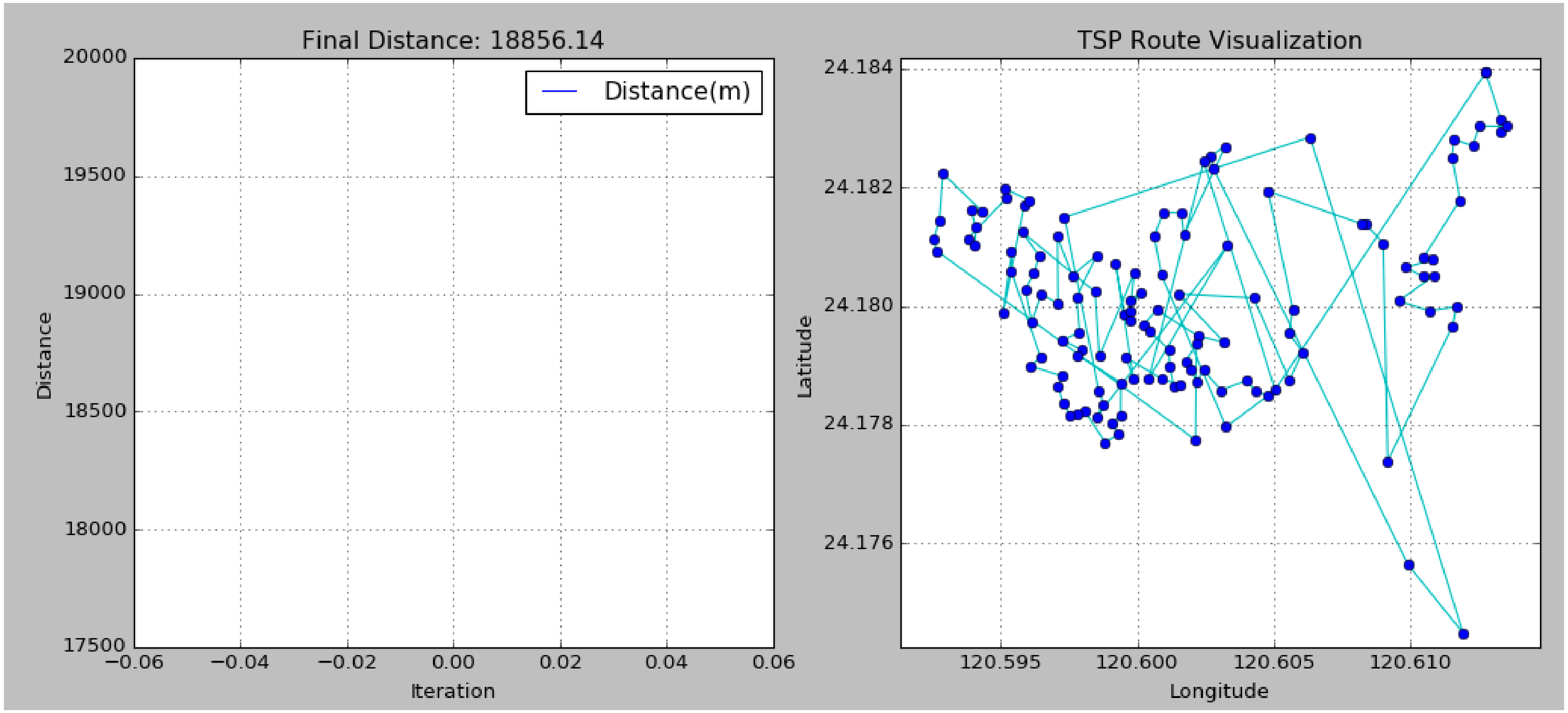
## Solutions being investigated: (*Concluded and described in the Final Report)

- Creating a smaller representative sample from the 50.000 images and run the SA on this sample.
- Searching further for articles about using SA to train Neural Networks.

# SA: Route Optimization – The TSP Problem



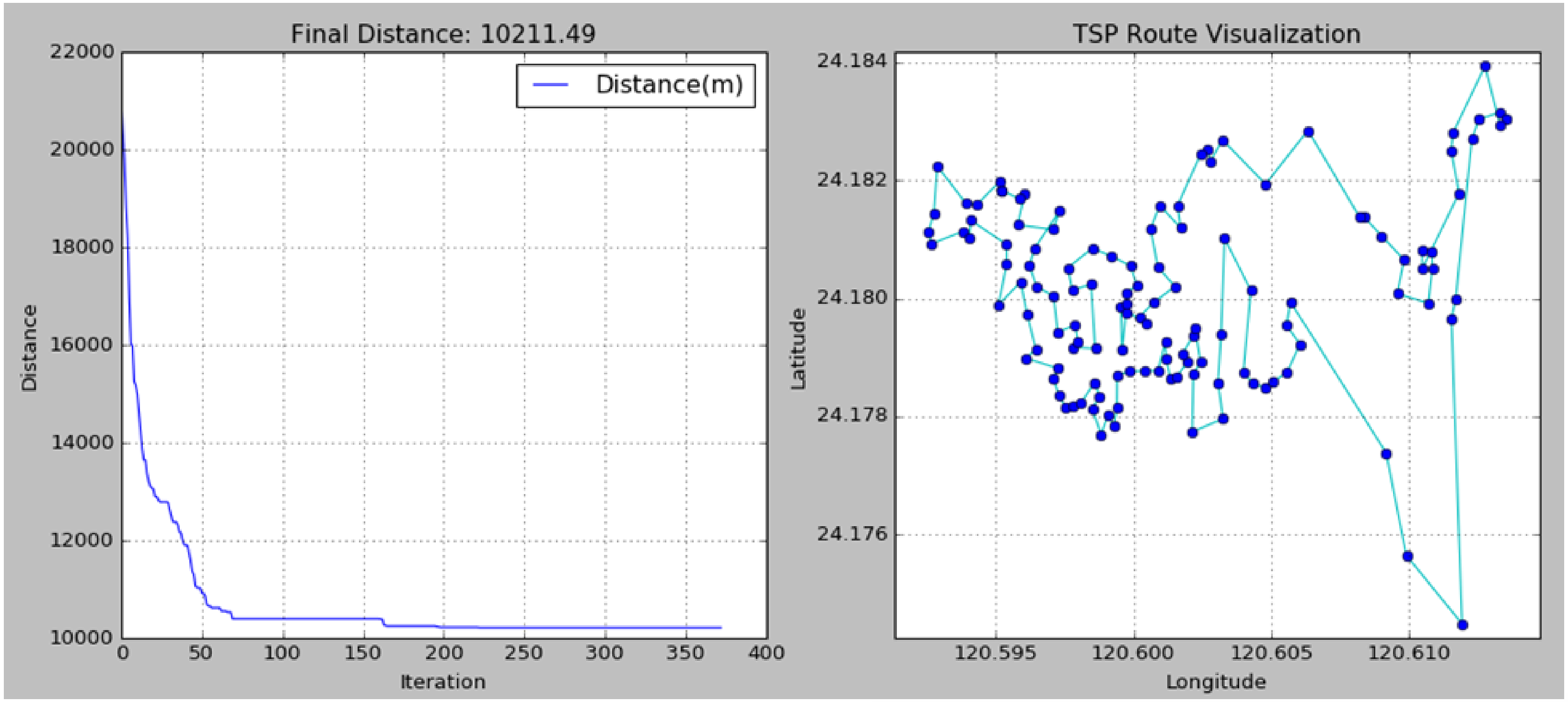
- Uses the same logic, but the next point will be based on a change in the sequence of cities to visit.  Then you calculate the distance for the new route:



# SA: Route Optimization



- It takes only 90 seconds to optimize 123 cities!
- From 18856 km to 10211km in 375 iterations!



## Simulated Annealing: Current Research



- Changes in the Cooling Schedule
- Adaptive Simulated Annealing (ASA)
  - Non-uniform sampling between the variables;
  - Different Cooling Rates for each variable;
  - Updates the cooling rate acording to the sensitivities;
- Mixing with other algorithms: GA, Tabu Search
- Optimization of Multiple Objectives
- Quantum Annealing
- Optimization of Neural Networks and Boltzmann Machines.

### Conclusions up to now:

- Conjugate Gradient Descent is more efficient than BFGS for large dimensions.
- The computation time for the Cost Function is high for 6000 points and 50.000 images.
- I believe that SA can find a good global optimum, but with a very high computational cost, since it depends on “blind” guesses.
- Many of these guesses, which take a long time to compute, are wasted…
- Each iteration, with a method like gradient descent, goes towards a minimum in all dimensions at once. This end up being more efficient when the gradient can be computed.
- SA seems to end up throwing the gradient descent in a local minimum that is not the best optimum. This is why in large Neural Networks it is better to use just the gradient.

### Solutions being investigated:

- Creating a smaller representative sample from the 50.000 images and run the SA on this sample.
- Searching further for articles about using SA to train Neural Networks.

**The alternative to use will depend on the type of problem you are solving, but some of the most interesting  are:**

- Gradient Descent (only continuous functions, can get stuck on a local minimum).
- Stochastic Gradient Descent
- Conjugate Gradient Descent
- Basin Hopping
- BFGS Quasi-newton (uses an approximation of the inversed Hessian)
- Genetic Algorithms.

# Simulated Annealing: Questions?