Acquisition Function

I have used the "Analytic LogEI" in section 4.1 of the article "Unexpected Improvements to Expected Improvement for Bayesian Optimization" for the acquisition function.

From a mathematical perspective, the values and gradients of EI are nonzero across the entire real line, except in cases without noise where points are perfectly correlated with earlier observations. However, simple implementations of h yield numerically zero when $z = (\mu(x) - y*)/\sigma(x)$ is small. This situation arises when the model is highly confident that there is minimal potential for improvement at x.

The implementation of "log_h" was proposed in this article which can be computed for a much larger range of inputs. Specifically, we compute

$\text{LogEI}(x) = \text{log\_h}((\mu(x) - y^*) / \sigma(x)) + \log(\sigma(x))$

where log_h can be computed by:

- **if $z > -1$:**

  $\text{log\_h}(z) = \log(\phi(z) + z\Phi(z))$

- **if $-1/\sqrt{\epsilon} < z \leq -1$:**

  $\text{log\_h}(z) = -z^2/2 - c1 + \text{log1mexp}(\log(\text{erfcx}(-z / \sqrt{2}) |z|) + c2)$

- **if $z \leq -1/\sqrt{\epsilon}$:**
  $\text{log\_h}(z) = -z^2/2 - c1 - 2\log(|z|)$

Parameters	Description
ϵ	numerical precision (very small)
$\mu(x)$	The mean
$\sigma(x)$	Standard deviation
$y*$	Best observed y
Z	$(\mu(x) - y*) / \sigma(x)$
c1	$\log(2\pi) / 2$
C2	$\log(\pi / 2) / 2$
log1mexp (z)	$\log(1 - \exp(z))$
erfcx(z)	$\exp(z^2)\, \text{erfc}(z)$
erfc(z)	complementary error function
ϕ (z)	standard Normal density function
Φ (z)	standard Normal distribution function

Before using the “Analytic LogEI” above, I have used the “Probability Improvement” for the acquisition function which is:

$$PI(x)= \Phi\ (\mu(x) - y^* \sigma(x))$$

However, I have noticed that using the “Analytic LogEI” got me better results. There are other techniques in that paper, but I have selected this one for its simplicity (compared to the other proposed techniques in the paper). I have uploaded the paper as a reference in my respiratory.

Adding New Queries to the Input and Output Data

Every time I obtain the next input query from the acquisition function above and its corresponding output from the feedback, I add the query to the original input data X and its corresponding output to the original output data Y to get a new query, and so on. The code line for that is below (from function 1):

```
X = np.append(X,[ [0, 0.818182]  , [0.6, 0.65] , [0, 1] , [1, 0] ], axis=0)

Y = np.append(Y, [ 6.48e-234 , 0.0707318, 0 ], axis=0)
```

After adding a query and its corresponding output, I then fit the “new” X and Y data in the Gaussian distribution process regressor to get a new mean and standard deviation, and then I use them for the acquisition above.

In function 1 to 3, I used the code below to create the grid for grid search:

```
X_grid = []

for i in range(len(x1)):

  for j in range(len(x2)):

    X_grid.append([x1[i], x2 [j] ] )
```

However, for function 4 to 8, this way of creating a grid would cost a lot of computation time especially if we are creating multi dimensions (let’s say above 3 dimensions). Therefore, I have used the code below to create the grid:

```
import itertools as it

dim = 4
```

```
X_grid = np.fromiter(it.chain(*it.product(x1, repeat=dim)), dtype=float).reshape(-1,dim)
```

Where “dim” is the dimension which is chosen depending on the function (e.g. function 4 is 4-dimensional so dim is 4 in this case). This code line will create a high-dimensional grid with much less computational time.

Results and Entries of Queries

Function 1	
Entries	**Output Result**
[0, 0.818182]	6.48E-234
[0.6, 0.65]	0.070731809
[0, 0.999999]	0
[0.999999, 0]	0

Function 2	
Entries	**Output Result**
[0, 0.424242]	-0.005814902
[0, 0.383838]	-0.068368297
[0 , 0]	0.067523435
[0.272727, 0.999999]	-0.030545674

Function 3	
Entries	**Output Result**
[0.373737, 1, 0]	-0.152684678
[1, 0, 0.59596]	-0.115212322
[0, 0, 0.949495]	-0.277077714
[0.494949, 0, 0.717172]	-0.19009924

Function 4	
Entries	**Output Result**
[0.22449, 0.326531, 0.346939, 0.387755]	-2.70557175
[0.310345, 0.310345, 0.344828, 0.413793]	-1.197112837
[0.344828, 0.37931, 0.37931, 0.413793]	0.418945648
[0.344828, 0.37931, 0.586207, 0.448276]	-3.469662475

Function 5	
Entries	**Output Result**

[0, 0.693878, 0.9999, 0.9999]	2162.457311
[0.482759, 0.9999, 0.9999, 0.9999]	4647.763251
[0, 0, 0, 0]	163.1225
[0.9999, 0.9999, 0.9999, 0.9999]	8662.405001

Function 6	
Entries	**Output Result**
[0, 0, 0, 0.555556, 0]	-1.781393162
[0, 0, 0, 0, 0]	-2.318386965
[0, 0, 0, 0.666667, 0]	-1.531533405
[0.3333, 0, 0.777778, 0.9999, 0.1111]	-0.794190523

Function 7	
Entries	**Output Result**
[0, 0, 0, 0, 0, 0.555556]	0.157417827
[0, 0, 0, 0, 0, 0]	0.005089113
[0, 0, 0, 0, 0, 0.714286]	0.169681608
[0, 0.285714, 0.571429, 0, 0.428571 0.9999]	0.592457683

Function 8	
Entries	**Output Result**
[0.166667, 0.166667, 0.166667, 0.3333, 0.3333, 0.5, 0.3333, 0.5]	9.807744416
[0.25, 0.25, 0.25, 0.25, 0.5, 0.5, 0.25, 0.5]	9.8408
[0.25, 0.25, 0.25, 0.25, 0.5, 0.75, 0.5, 0.75]	9.60205
[0.25, 0.25, 0.25, 0.25, 0.5, 0.5, 0.5, 0.5]	9.6658

Actions might be taken for Improvement

If there were more weeks available, I would have tested other techniques proposed in the paper, such as MonteCarlo Parallel LogEI, to obtain the acquisition function, and then compared it to the results of “Analytic LogEI” technique used in this capstone. Alternatively, it would have been worth looking up for other methods that could be effective for optimization such as reinforcement learning, but I would have needed more time to learn reinforcement learning.