In [1]:
```
import torch
import functools
import matplotlib.pyplot as plt
```

Definition of the surrogate

In [2]:
```
class SimpleRBF:

    def __init__(self,
                 dim,
                 lb,
                 ub,
                 silent=False,
                 device="cpu",
                 eta=1e-6,
                 use_rnd=False):

        self._dim = dim
        self._lb = lb
        self._ub = ub
        self._silent = silent
        self._device = device

        self._eta = eta
        self._solver = torch.linalg.solve

        self._raw_x = torch.empty((0, self._dim), device=self._device)
        self._raw_y = torch.empty((0, 1), device=self._device)

        self._use_rnd = use_rnd

        self._lambda = None
        self._c = None

        self._k = None

        self._avg = None
        self._std = None

    def tell(self, x, y):


        self._raw_x = torch.cat((self._raw_x, x), 0)
        self._raw_y = torch.cat((self._raw_y, y), 0)

        if self._use_rnd:
            n, d = self._raw_x.shape

            if n < d + 1:
                return

        data_x = to_unit_box(self._raw_x, self._lb, self._ub)
        data_y, self._avg, self._std = to_normed(self._raw_y)

        n, d = data_x.shape

        if self._k is not None:
```

```
            with torch.no_grad():
                ker = self._k(data_x, data_x).evaluate()
        else:
            ker = torch.cdist(data_x, data_x).pow(3)

        reg = self._eta * torch.eye(n)
        Phi = ker + reg
        P = torch.cat((data_x, torch.ones(n, 1)), 1)
        O = torch.zeros((d + 1, d + 1))

        A = torch.cat((torch.cat((Phi, P), 1), torch.cat((P.T, O), 1)))

        B = torch.cat((data_y, torch.zeros((d + 1, 1))))
        if n > d + 1:
            tup = torch.linalg.lstsq(A, B)
        else:
            tup = self._solver(A, B)

        if isinstance(tup, tuple):
            X = tup.solution
        else:
            X = tup

        self._lambda = X[:n]
        self._c = X[n:]

        del A, B, O, P, Phi, X, reg, data_x, data_y, ker

    def ask(self, x):

        if self._use_rnd:
            n, d = self._raw_x.shape

            if n < d + 1:
                return torch.randn(x.shape[0], 1)

        s_x = to_unit_box(x, self._lb, self._ub)
        data_x = to_unit_box(self._raw_x, self._lb, self._ub)

        if self._k is not None:
            with torch.no_grad():
                ker = self._k(s_x, data_x).evaluate()
        else:
            ker = torch.cdist(s_x, data_x).pow(3)

        tail_x = torch.cat((s_x, torch.ones(s_x.shape[0], 1)), 1)

        output = ker @ self._lambda + tail_x @ self._c

        del tail_x, ker, data_x, s_x

        s_y = from_normed(output, self._avg, self._std)

        return s_y

    def has(self, x):
        s_x = to_unit_box(x, self._lb, self._ub)
        data_x = to_unit_box(self._raw_x, self._lb, self._ub)
        d_x = torch.cdist(s_x, data_x).pow(3)
```

```
        return d_x.lt(1e-6).any(1).int()

    def __str__(self):
        args = []
        args.append(f"eta: {self._eta}")
        args.append(f"device: {self._device}")
        args = ", ".join(args)
        color = u"\u001b[38;5;231m"

        return f"{color}SimpleRBF\033[0m Surrogate [{args}]"
```

Helper functions

In [3]:

```
def to_unit_box(x, lb, ub):
    return (x - lb) / (ub - lb)


def from_unit_box(x, lb, ub):
    return lb + (ub - lb) * x


def to_normed(y):
    avg, std = y.mean(), y.std()
    data_y = (y - avg) / std

    return data_y, avg, std


def from_normed(y, avg, std):
    return avg + std * y


def trunc_normal(mean, std, a, b, size):
    if len(mean.shape) > 1:
        mean = mean[0]
    if len(std.shape) > 1:
        std = std[0]
    if len(a.shape) > 1:
        a = a[0]
    if len(b.shape) > 1:
        b = b[0]

    sqrt_2 = 1.4142135623730951
    lb = (1+torch.erf(((a-mean)/std)/sqrt_2))/2
    ub = (1+torch.erf(((b-mean)/std)/sqrt_2))/2

    tensor = 2*(torch.rand(size) * (ub - lb) + lb) - 1

    eps = torch.finfo(tensor.dtype).eps
    tensor = clamp(tensor, min_val=-(1.-eps), max_val=(1.-eps))

    tensor.erfinv_().mul_(sqrt_2*std).add_(mean)
    tensor = clamp(tensor, min_val=a, max_val=b)

    return tensor

def clamp(tensor, min_val, max_val):
    if not isinstance(min_val, torch.Tensor):
        min_val = torch.Tensor([min_val])
```

```
    if not isinstance(max_val, torch.Tensor):
        max_val = torch.Tensor([max_val])

    return torch.min(torch.max(tensor, min_val), max_val)
```

Benchmark function

In [4]:
```
class Ackley:
    lb = -5.0
    ub = 10.0
    names = ("ack", "ackley")

    def __init__(self, noise=0):
        self._noise = noise

    def __call__(self, x):
        """https://www.sfu.ca/~ssurjano/ackley.html
        `x` is a 2D np.array with shape (n_points x n_dimensions)."""
        if len(x.shape) < 2:
            x = x[None]
        dim_sqrt = torch.sqrt(torch.Tensor([x.shape[1]]))
        e = 2.7182817459106445
        a, b, c = 20.0, 0.2, 2 * 3.141592653589793

        part1 = -a * torch.exp(-b / dim_sqrt * torch.norm(x, dim=-1))
        part2 = -(torch.exp(torch.mean(torch.cos(c * x), dim=-1)))
        y = part1 + part2 + a + e

        if self._noise > 0:
            y += torch.randn(y.shape).mul(self._noise)

        return y[:, None]
```

Algorithm functions

In [5]:
```
def phi_fn(evals_spent, n_evals):
    nds = [0.10, 0.10, 0.05, 0.025, 0.005, 0.005, 0, 0]
    val = nds[int(len(nds) * evals_spent / n_evals)]

    return val

def temp(k, n_evals, t_0=.1, t_K=1e-9):
    alpha = (t_K / t_0)**(1 / n_evals)
    t_k = alpha**(k) * t_0

    return torch.Tensor([t_k])
```

Example running ROSA on Ackley in 30 dimensions

In [6]:
```
dim = 30
f = Ackley()
f.lb = torch.zeros(dim) + f.lb
f.ub = torch.zeros(dim) + f.ub
```

In [7]:
```
rnd_seed = 42
torch.manual_seed(rnd_seed)
```

```
np.random.seed(rnd_seed)

n_cand = 50000
n_evals = dim*10
bsz = 1 # m in text
data_x = torch.empty((0, dim))

init_evals = int(0.02 * n_evals)
data_x = torch.empty((0, dim))
init_x = from_unit_box(torch.rand((init_evals, dim)), f.lb, f.ub)
data_x = torch.cat((data_x, init_x), 0)

data_y = f(data_x)

t_norm = functools.partial(trunc_normal,
                        std=(f.ub - f.lb) / 6.0,
                        a=f.lb,
                        b=f.ub)

T = functools.partial(temp, n_evals=n_evals)
phi = functools.partial(phi_fn, n_evals=n_evals)
model = SimpleRBF(dim=dim, lb=f.lb, ub=f.ub)
evals_spent = 0
hist = []
for idx in range(data_y.shape[0]):
    evals_spent += 1
    hist.append(data_y[idx].item())

x_best = data_x[data_y.argmin()][None, :]
y_best = data_y.min()

while evals_spent < n_evals:
    model.tell(data_x, data_y)
    p_select = phi(evals_spent)

    # neighbouring points generation
    dx = []
    for i in range(x_best.shape[0]):
        n_c = int(n_cand / x_best.shape[0])
        c = torch.repeat_interleave(x_best[i][None], n_c, axis=0)
        mask = torch.rand(c.shape) < p_select
        ind = torch.where(mask.sum(1).eq(0))[0]
        mask[ind, torch.randint(dim, size=ind.shape)] = True
        c[mask] = t_norm(x_best[i], size=c.shape)[mask]

        s_y = model.ask(c)
        b = int(bsz / x_best.shape[0])
        cand_order = s_y.argsort(0).squeeze()
        unique = model.has(c[cand_order[:10 * b]])
        best_idx = cand_order[unique.argsort(0).squeeze()[:b]]
        dx.append(c[best_idx])

    data_x = torch.cat(dx)
    data_y = f(data_x)
    curr_y = data_y.min()
    k = evals_spent
    u = torch.rand(1).item()
    sa_p = torch.exp(-(curr_y - y_best) / T(k)).item()
    hist.append(curr_y)
    if u < sa_p:
```

```
        x_best = data_x[data_y.argsort(0)[:1, 0]]
        y_best = data_y.min()
    evals_spent += data_x.shape[0]
print(f'Best objective value:{y_best:.2f}')
```

```
Best objective value:3.15
```

In [8]:
```
%matplotlib inline
import matplotlib.pyplot as plt

fig, ax = plt.subplots(figsize=(8,6))
best = np.minimum.accumulate(hist)
ax.plot(np.arange(len(best)), best, label="ROSA",
        lw=2, c=plt.cm.tab10(3), marker="s", markevery=30)
ax.tick_params(labeltop=False, labelright=True)

ax.set_xticks(range(0, len(best)+1, 30))
ax.set_xlim(0, len(best))

ax.grid(True, color="grey", alpha=0.15)
ax.set_title("Ackley $D=30$")
ax.set_xlabel("Number of function evaluations")
ax.set_ylabel("Best value found");
```