Micro-data — Part 2

Bayesian optimization

Notebook: https://github.com/jbmouret/bo_tutorial/blob/main/bo.ipynb
[colab] https://colab.research.google.com/github/jbmouret/bo_tutorial/blob/main/bo.ipynb

 **@jbmouret**

 **@jb_mouret**

jean-baptiste.mouret@inria.fr — https://members.loria.fr/jbmouret

Concept of Bayesian optimization

Objective:

- find the maximum (or minimum) of an unknown (black-box) function F
- in RL F=reward at the end of an episode

  … with as few calls as possible to the function (< 100)

Parameters of the policy

$$J(\boldsymbol{\theta}) = \mathbb{E}\left[\sum_{t=1}^{T} r(\mathbf{x}_t) \middle| \boldsymbol{\theta}\right]$$

Reward

Strategy : surrogate of the objective

Model: $p(J(\boldsymbol{\theta})|D_{1,\dots,n})$

Concept:

1. learn a model of the function using a few random points
2. use the model to select the most interesting point to try next
   - balance exploitation (best point according to the model)
   - … and exploration (points that are far)
3. evaluate the function
4. update the model

Other names:

- Kriging
- Surrogate modeling

Brochu E, Cora VM, De Freitas N. *A tutorial on Bayesian optimization of expensive cost functions, with application to active user modeling and hierarchical reinforcement learning.* arXiv preprint arXiv:1012.2599. **2010**

Shahriari, Bobak, et al. "*Taking the human out of the loop: A review of bayesian optimization*." ***Proceedings of the IEEE*** **104.1 (2016).**

What model for Bayesian optimization?

- **Objective:** given the data, predict the shape of the underlying function

➔ This is a regression problem

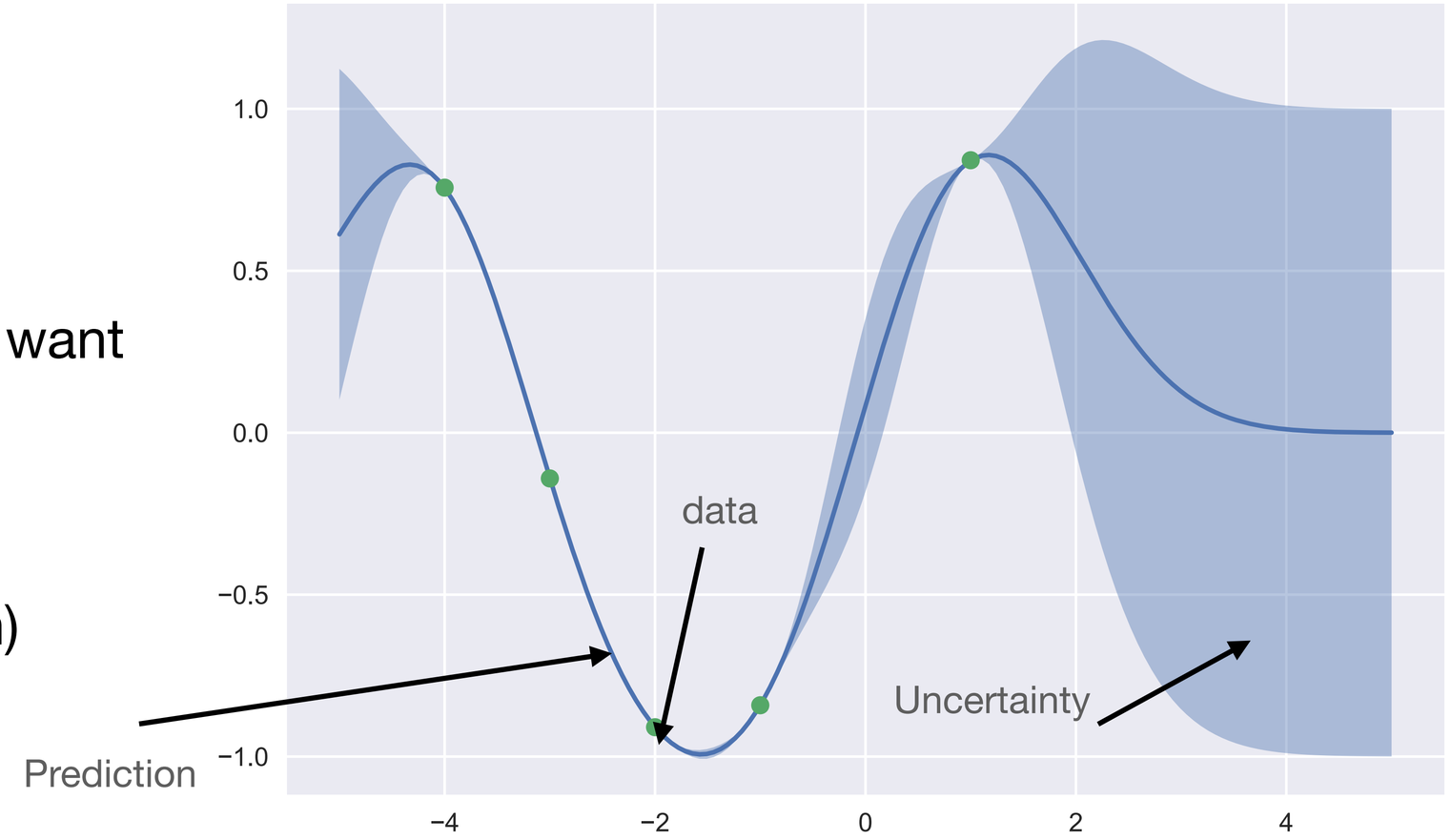
- Can we use neural networks?
  - of course!

  … but they require a lot of data

  … which is the opposite of what we want

- *What do we want?*
  - works well with little data
  - predict uncertainty (for exploration)

➔ **Gaussian processes**



Notebook: https://github.com/jbmouret/bo_tutorial/blob/main/bo.ipynb

[colab] https://colab.research.google.com/github/jbmouret/bo_tutorial/blob/main/bo.ipynb

Basics of Gaussian processes

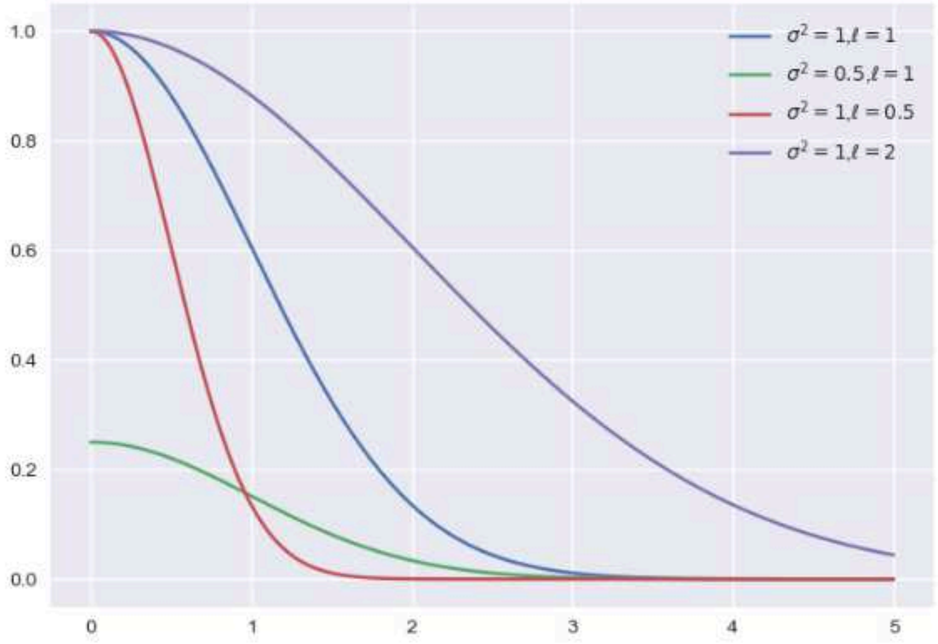
- A Gaussian process is a distribution over functions that associates to every $\mathbf{x}$ a Gaussian distribution with a mean $\mu(x)$ and a variance $\sigma(x)$
- $D_{1:t}^d = \{f_d(\mathbf{x_1}), \ldots, f_d(\mathbf{x}_t)\}$: observations

$$p(\hat{f}(\mathbf{x}) \,|\, D_{1:t}^d, \mathbf{x}) = \mathcal{N}(\mu(\mathbf{x}), \sigma^2(\mathbf{x}))$$

- Each datapoint influence the others according to a *kernel*
- *e.g. :* exponential-squared function:
  - ℓ^2: length scale
  - σ^2: variance when unknown

$$k(x_1, x_2) = \sigma^2 \exp(-\frac{||x_1 - x_2||^2}{2\ell^2})$$

```
def expsq(dx, alpha=(1,1)):
    sigma, ell = alpha
    return sigma**2 * np.exp(-dx**2 / (2*ell**2))
```



Notebook: https://github.com/jbmouret/bo_tutorial/blob/main/bo.ipynb
[colab] https://colab.research.google.com/github/jbmouret/bo_tutorial/blob/main/bo.ipynb

Mean and variance

$$\mathbf{k} = \left[k(\mathbf{x}_1, \mathbf{x}), \cdots, k(\mathbf{x}_t, \mathbf{x})\right]$$

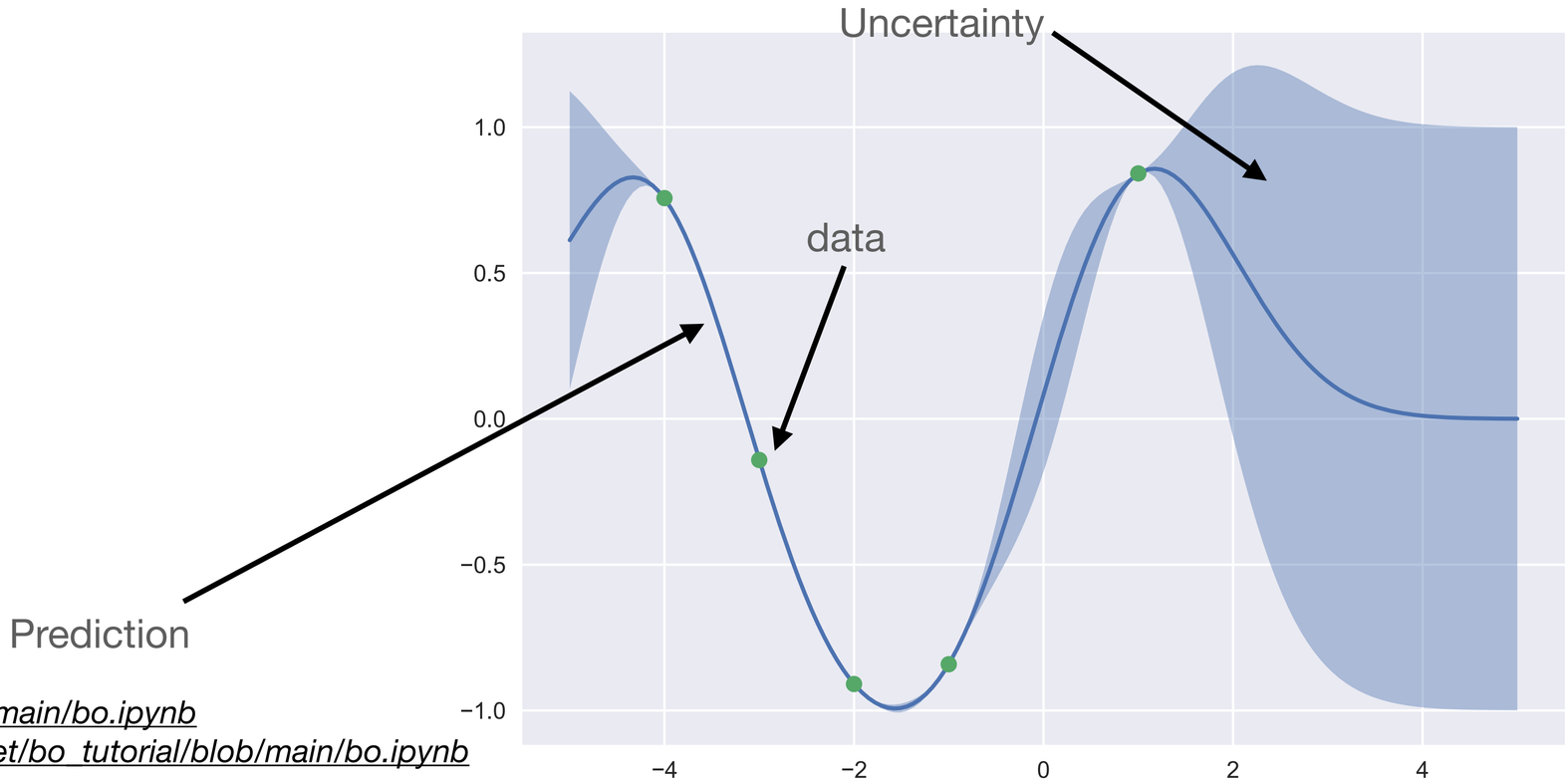
$$K = \begin{bmatrix} k(x1_1, x_1) & \cdots & k(x_1, x_t) \\ \vdots & \ddots & \vdots \\ k(x_t, x_1) & \cdots & k(x_t, y_t) \end{bmatrix} + \sigma^2_{noise} I$$

$$\mu(\mathbf{x}) = \mathbf{k}^\top K^{-1} D^d_{1:t}$$

$$\sigma(\mathbf{x}) = k(\mathbf{x}, \mathbf{x}) - \mathbf{k}^\top K^{-1} \mathbf{k}$$

In real code: do not use the inversion of K
(use the Cholesky decomposition)

```python
def kernel_matrix(x1, x2, p):
    # this is to get a matrix of the distance between x1 and x2
    dx =  np.subtract.outer(x1, x2)
    return expsq(dx,p)

# Don't do this in real code -- use the Cholesky decomposition (faster, more stable
# see the GP book, p19
def gp(x_test, x_train, y_train, p):
    K = kernel_matrix(x_train, x_train, p)  + 1e-3* np.eye(len(x_train)) # add some
    K_inv = np.linalg.inv(K)
    k = kernel_matrix(x_train, x_test,p)
    mu = k.T.dot(K_inv).dot(y_train)
    # this computes a lot of useless terms, but we avoid a loop (for the notebook)
    sigma = (kernel_matrix(x_test, x_test, p) - k.T.dot(K_inv).dot(k)).diagonal()
    return mu, sigma
```



Notebook: https://github.com/jbmouret/bo_tutorial/blob/main/bo.ipynb
[colab] https://colab.research.google.com/github/jbmouret/bo_tutorial/blob/main/bo.ipynb

Gaussian processes: conclusion

- GPs are very good for regression
- … but:
  - query in $O(n^2)$ (n = number of training samples)
  - "training" in $O(n^3)$ (Matrix inversion / Cholesky), more if optimizing the HP
  - they do not scale well! (< 1000 points)

To go further…

- learn the hyper-parameters of the kernel (σ^2, ℓ^2) by maximizing the likelihood
- use more specialized kernels
- sparse GPs (scale better)

GPs for Bayesian optimization

Concept:

1. learn a GP of the function using a few random points
2. use the model to select the most interesting point to try next
3. evaluate the function
4. update the model

Parameters of the policy

$$J(\boldsymbol{\theta}) = \mathbb{E}\left[\sum_{t=1}^{T} r(\mathbf{x}_t) \middle| \boldsymbol{\theta}\right]$$

Reward

Strategy : surrogate of the objective

Model: $p(J(\boldsymbol{\theta})|D_{1,\dots,n})$

What is the most interesting point?

Expected improvement:

$$x_{t+1} = \arg\max_x \mathbb{E}\left(||f(x) - f(x^*)|| \; |D_{1:t}\right)$$

closed form for GPs (a bit too complicated for here)

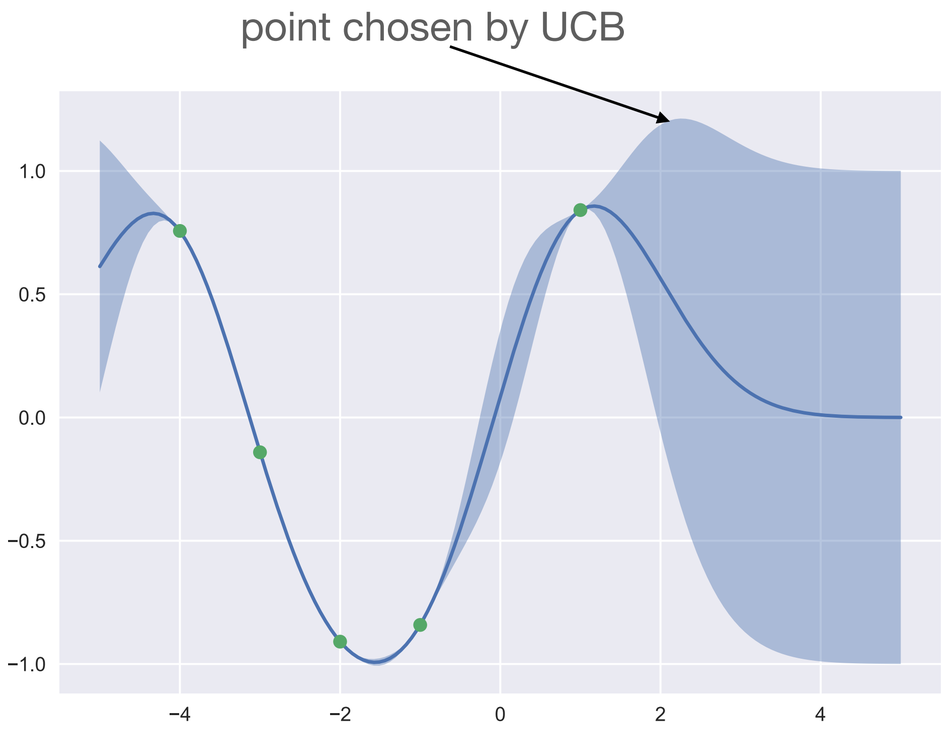
UCB:

$$x_{t+1} = \arg\max_x \mu(x) + \kappa\sigma(x)$$

tune exploration

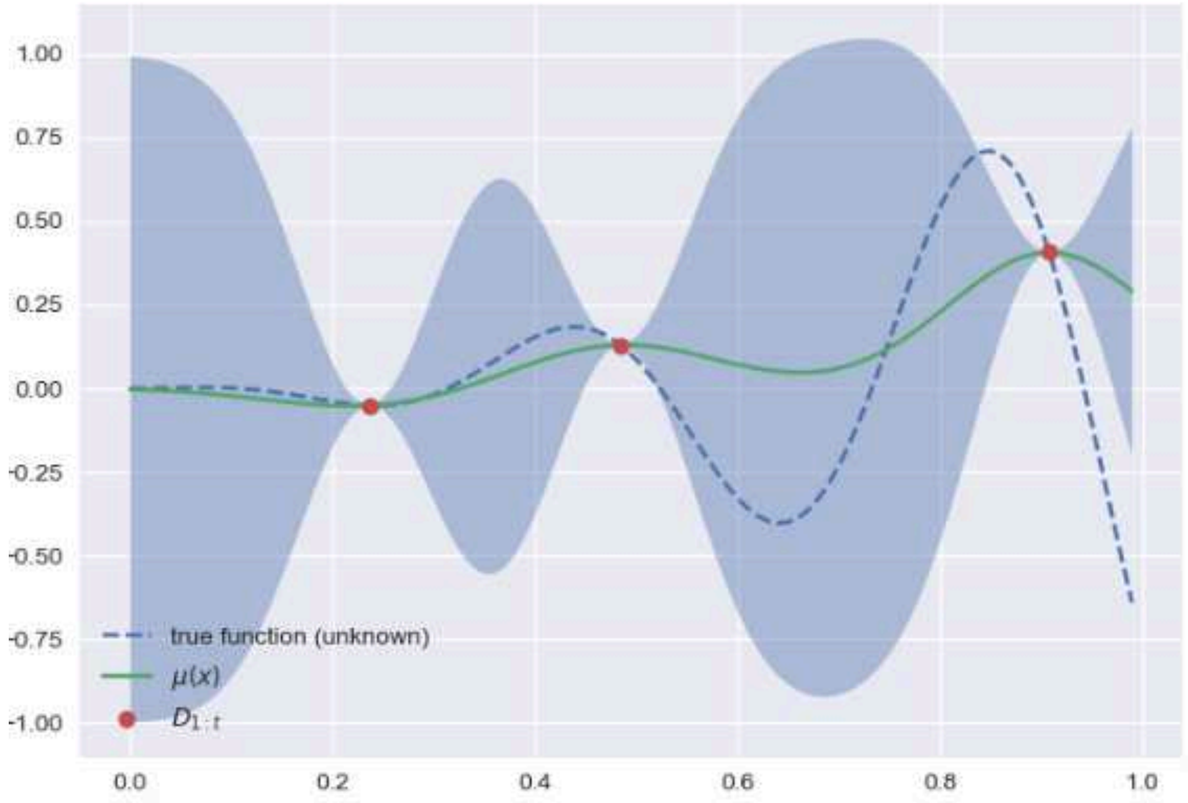
predicted value

high uncertainty
(far from tested points)



GPs for Bayesian optimization

Step 1: random initialization

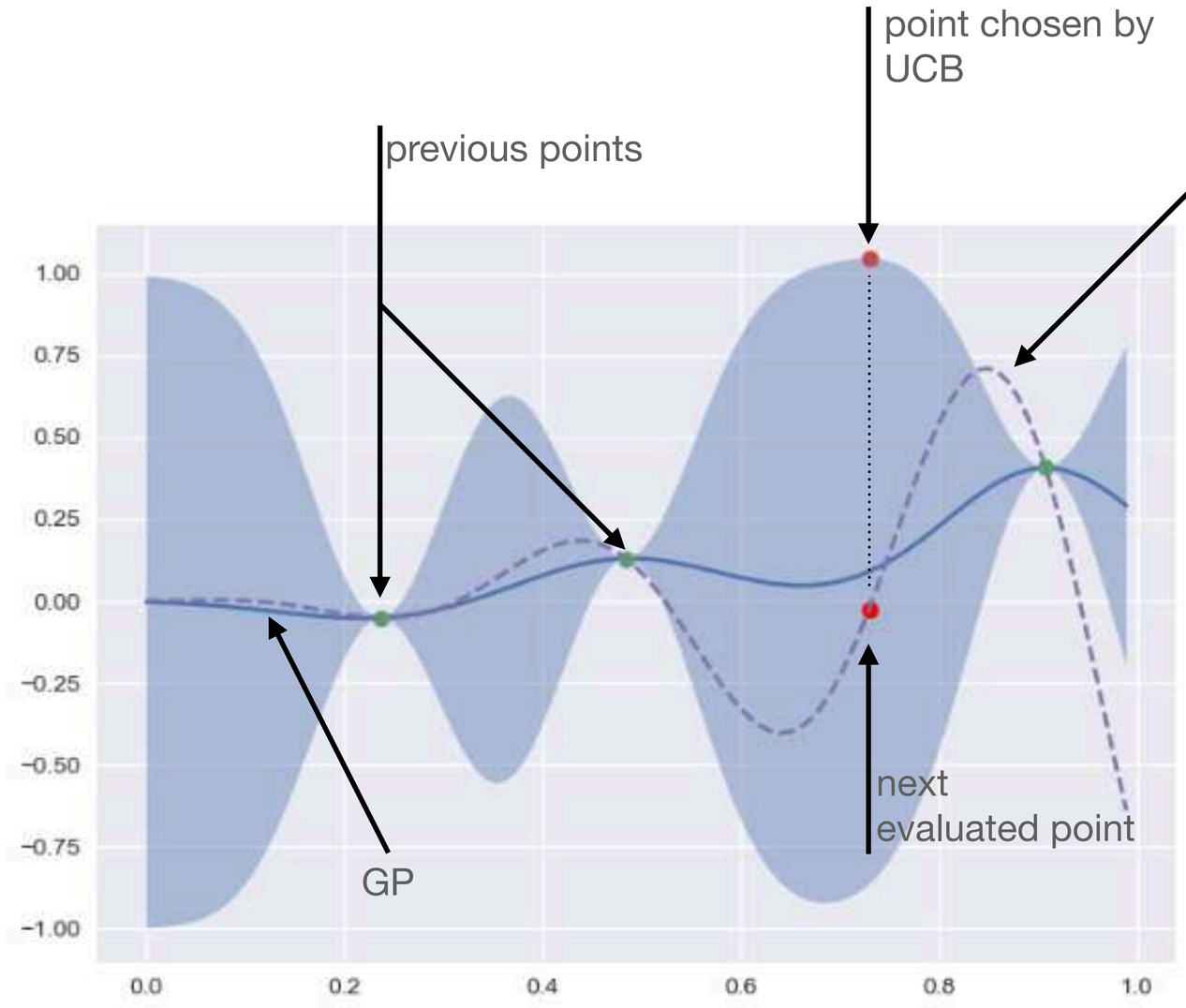



```python
x = np.random.random(3)
y = f(x)
x_test = np.arange(0, 1, 0.01)
sigma_sq = 1
ell = 0.1
# ell and sigma_sq should be tuned to match
# the magnitude of the function to optimize
# (prior knowledge)
params = (sigma_sq, ell)
mu, sigma = gp(x_test, x, y, params)
```

Notebook: https://github.com/jbmouret/bo_tutorial/blob/main/bo.ipynb
[colab] https://colab.research.google.com/github/jbmouret/bo_tutorial/blob/main/bo.ipynb

GPs for Bayesian optimization

Step 2: UCB




```python
# now get the next point
def ucb(m, s):
    return m + s

# we should use an optimizer (e.g., CMA-ES), but we "cheat" here.
# fit the GP and find the max of UCB
mu, sigma = gp(x_test, x, y, params)
i_next = np.argmax(ucb(mu, sigma))
x_next = x_test[i_next]
y_next = f(x_next)
```

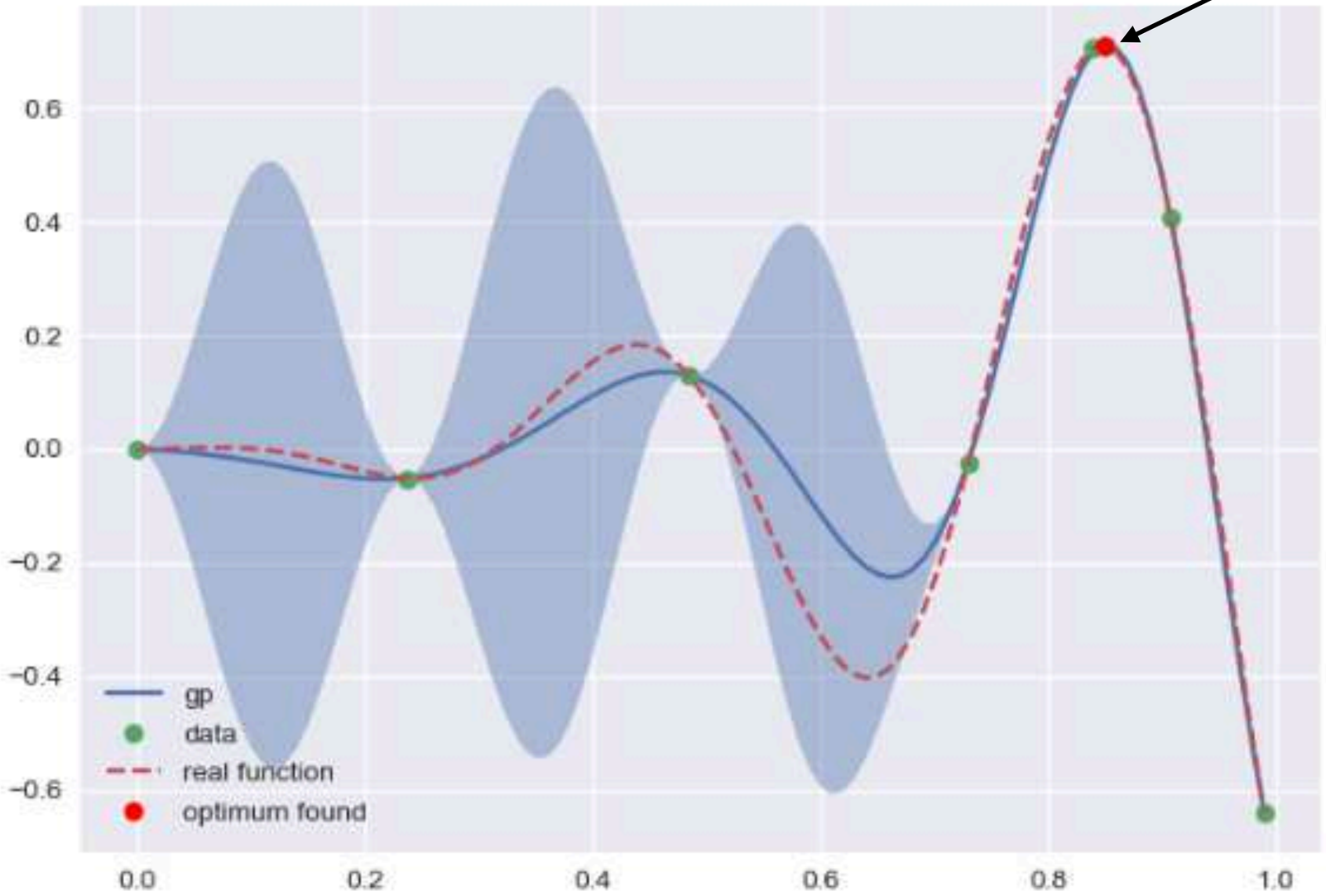
Here we optimize the GP by discretizing (for the example).

We should use a non-linear optimizer (gradient descent, CMA-ES)

Notebook: https://github.com/jbmouret/bo_tutorial/blob/main/bo.ipynb
[colab] https://colab.research.google.com/github/jbmouret/bo_tutorial/blob/main/bo.ipynb

GPs for Bayesian optimization

A few iterations later




```
while x.shape[0] < 10:
    mu, sigma = gp(x_test, x, y, params)
    i_next = np.argmax(ucb(mu, sigma))
    x_next = x_test[i_next]
    print(x_next, i_next)
    y_next = f(x_next)
    x = np.append(x, x_next)
    y = np.append(y, y_next)
```

```
BEST: x= 0.85 y= 0.710352930491885  data= 10
```

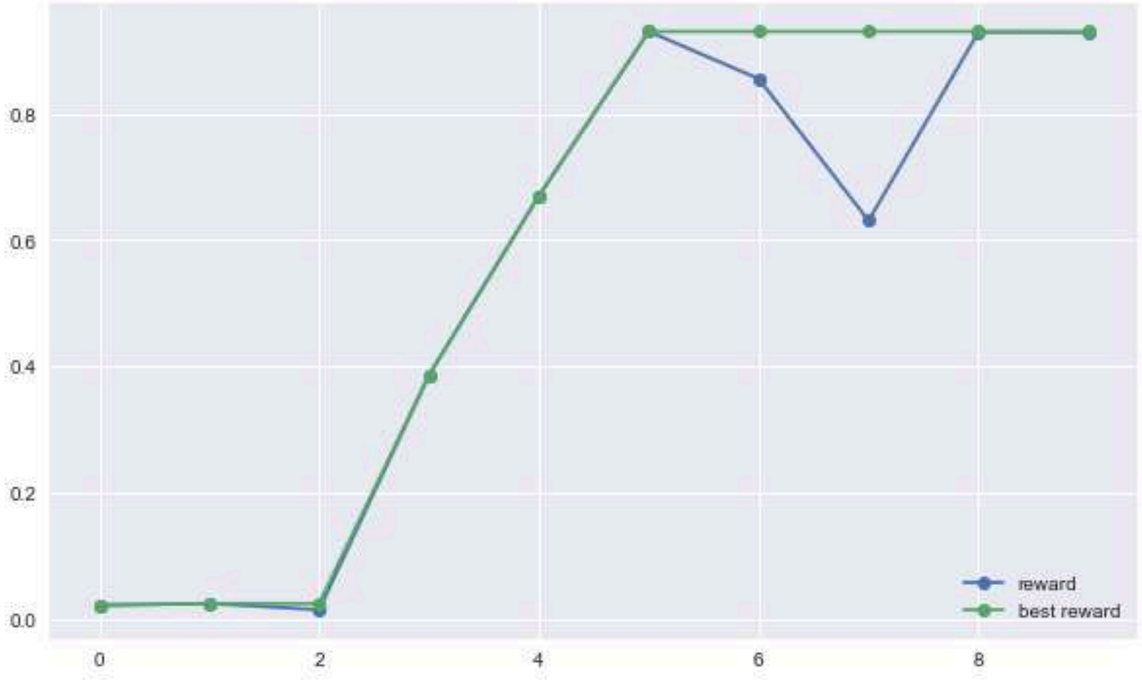
Notebook: https://github.com/jbmouret/bo_tutorial/blob/main/bo.ipynb
[colab] https://colab.research.google.com/github/jbmouret/bo_tutorial/blob/main/bo.ipynb

“Policy search”

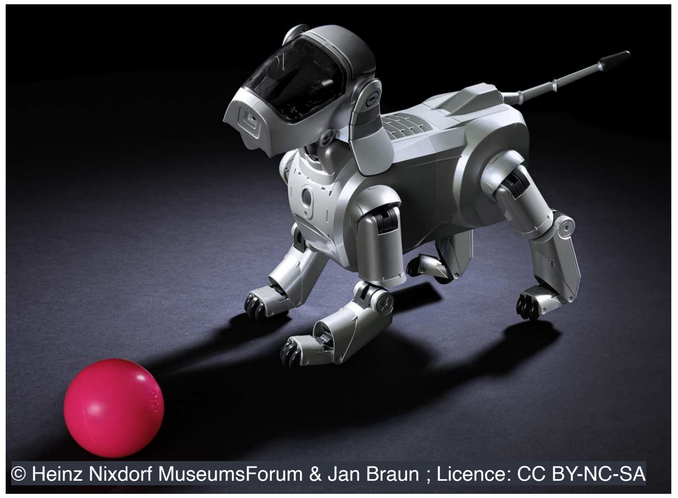
Learning to control a (modified) pendulum

```
class PDController():
    def  __init__(self, p, d):
        self.p = p
        self.d = d
    def action(self, state):
        return np.array([self.p * (math.pi - state[0]) - self.d * state[1]])*10.
```

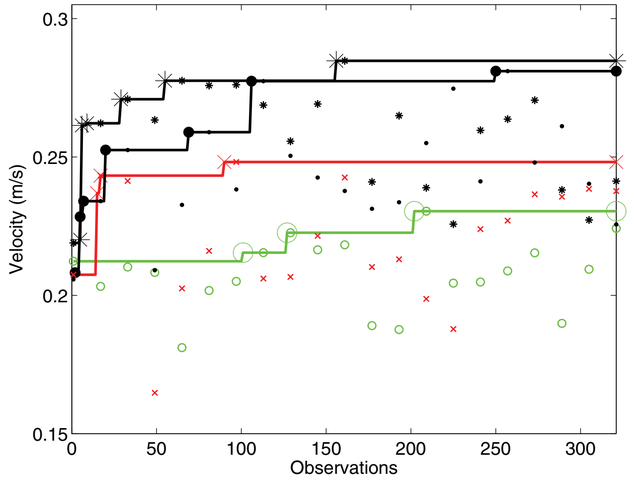
```
# we use GPy (a real GP) because our "teaching implementation" is 1-D only
import GPy
# sample inputs and outputs
n_init = 2
n_iteration = 8
x = np.random.uniform(0.,1.,(n_init,2))
y = np.zeros((x.shape[0], 1))
for i in range(0, x.shape[0]):
    pd = PDController(x[i,0], x[i,1])
    y[i] = sim_pendulum(pd, 100, False)

for i in range(0, n_iteration):
    # define kernel
    ker = GPy.kern.Matern52(2,ARD=True) + GPy.kern.White(2)
    # create GP model
    m = GPy.models.GPRegression(x, y, ker, normalizer=True)
    # optimize and plot
    m.optimize(messages=False,max_f_eval = 1000)
    # search for the UCB max, using basic random search
    q = np.random.uniform(0.,1.,(5000,2))
    mu,sigma = m.predict(q)
    ucb = mu + sigma
    i_next = np.argmax(ucb)
    # evaluate the new point
    x_next = q[i_next,:]
    pd = PDController(x_next[0], x_next[1])
    y_next = sim_pendulum(pd, 100, False)
    x = np.vstack((x, [x_next]))
    y = np.vstack((y, [y_next]))
    print("BEST:", np.max(y), x[np.argmax(y)])
```



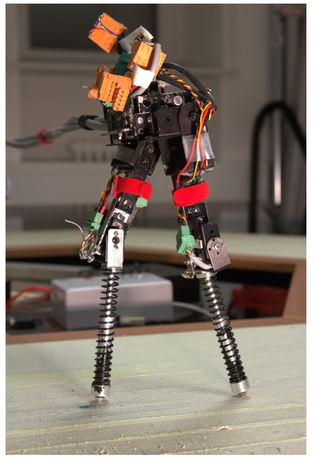
Examples — locomotion



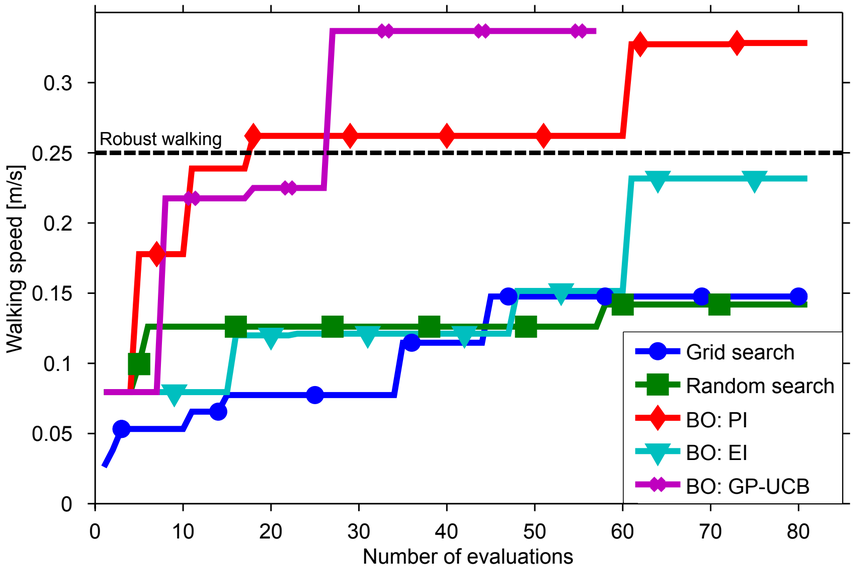
- **Policy:** Finite state machine
- **Reward:** walking speed
- **Parameters: 15**



(•) GP w/MPI	(∗) GP w/MPI	(×) H.Clmb	(○) U.Rand
0.281 m/s	0.285 m/s	0.248 m/s	0.230 m/s
$\sigma_f^2 = 0.06$	$\sigma_f^2 = 0.6$		



- **Policy:** Finite state machine
- **Reward:** walking speed
- **Parameters:** 4



Calandra, R., Seyfarth, A., Peters, J., & Deisenroth, M. P. (2014). An experimental comparison of Bayesian optimization for bipedal locomotion. In *2014 IEEE International Conference on Robotics and Automation (ICRA)* (pp. 1951-1958). IEEE.
Lizotte, D. J., Wang, T., Bowling, M. H., & Schuurmans, D. (2007, January). Automatic Gait Optimization with Gaussian Process Regression. In *IJCAI* (Vol. 7, pp. 944-949).

BO with a custom kernel based on simulation

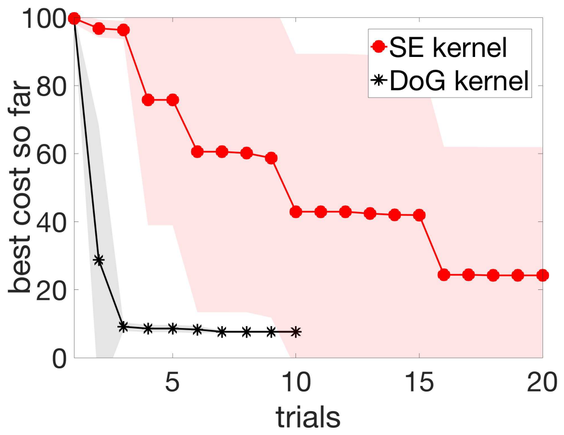
- Learn gait of a biped robot
- Use 4 determinants of gaits (inspired by physiotherapists) to compare points

(instead of the parameters values)

- *swing leg retraction*
- *center of mass height*
- *trunk lean*
- *average walking speed*

➡ ***evaluated in simulation before testing***

➡ ***5 parameters to learn***



Rai, A., Antonova, R., Song, S., Martin, W., Geyer, H., & Atkeson, C. (2018). Bayesian optimization using domain knowledge on the ATRIAS biped. In *2018 IEEE International Conference on Robotics and Automation (ICRA)* (pp. 1771-1778). IEEE.

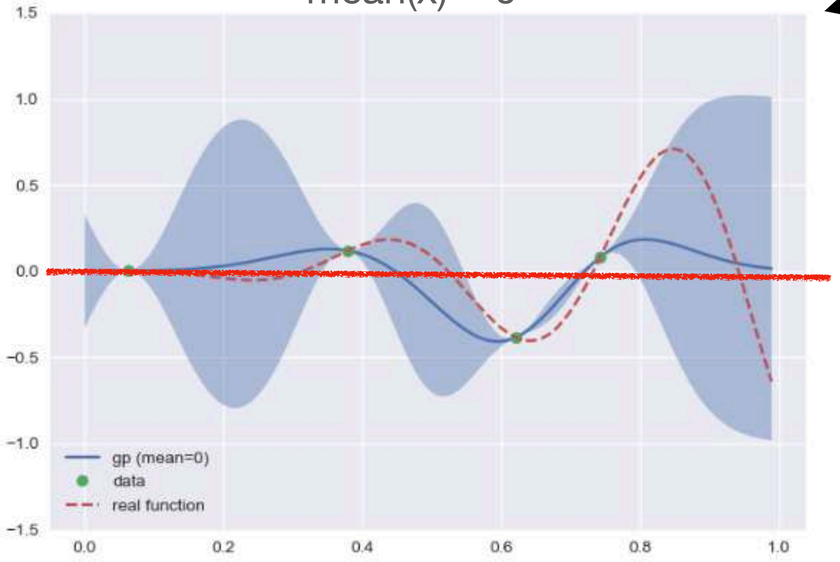
Rai, A., Antonova, R., Meier, F., & Atkeson, C. G. (2019). Using Simulation to Improve Sample-Efficiency of Bayesian Optimization for Bipedal Robots. *J. Mach. Learn. Res.*, *20*(49), 1-24.

Bayesian optimization with a mean function

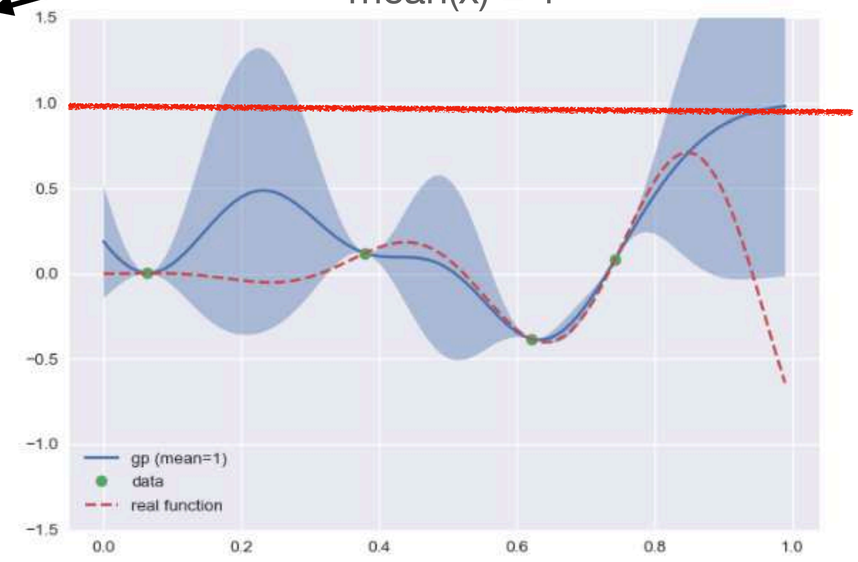
```python
# we add a new parameter: the mean function
def gp(x_test, x_train, y_train, mean_function, p):
    y_train = y_train - mean_function(x_train)# substract the mean before learning the GP
    # same as before
    K = kernel_matrix(x_train, x_train, p)  + 1e-3* np.eye(len(x_train)) # add some noise (stability)
    K_inv = np.linalg.inv(K)
    k = kernel_matrix(x_train, x_test,p)
    # we put the mean back
    mu = k.T.dot(K_inv).dot(y_train) + mean_function(x_test)
    sigma = (kernel_matrix(x_test, x_test, p) - k.T.dot(K_inv).dot(k)).diagonal()
    return mu, sigma
```

Same data!

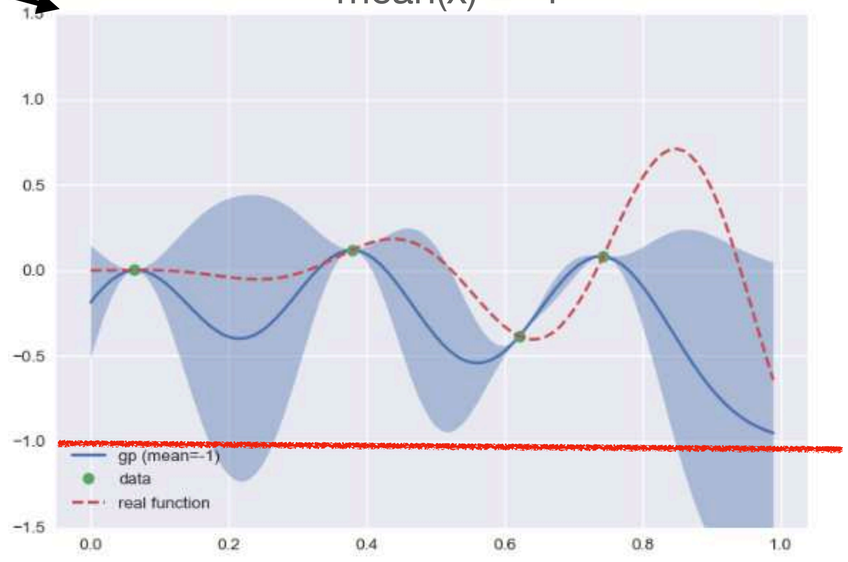
mean(x) = 0



mean(x) = 1



mean(x) = -1



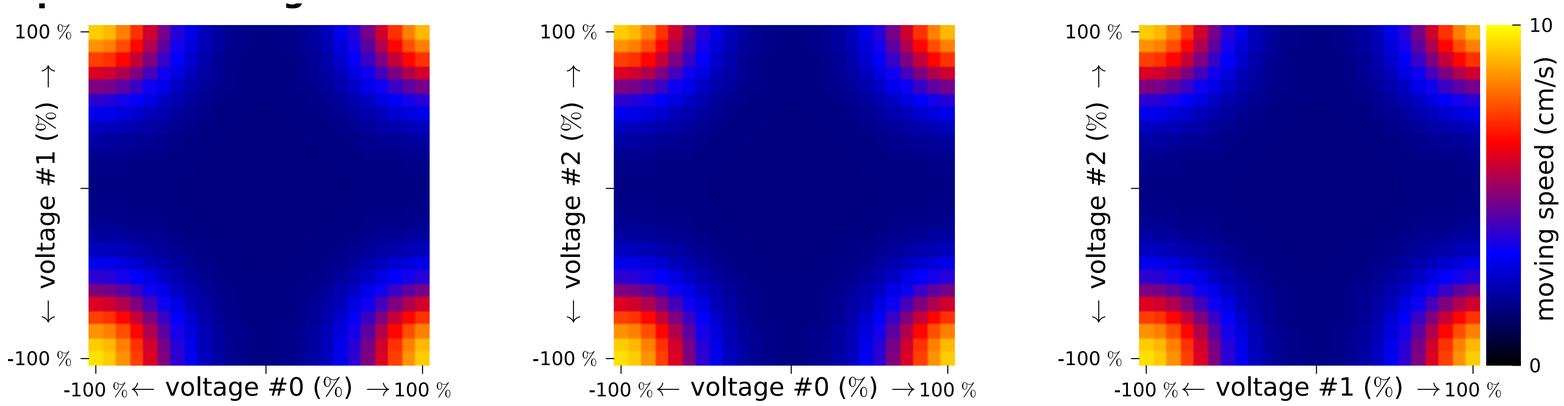
➔ The GP is “pulled" by the mean function

Example: soft tensegrity robot

- **Policy:** 3 voltages (one for each vibrator), for 3 seconds
- **Fitness/objective/reward :** locomotion speed (measured with external motion capture)
- External power (for now)

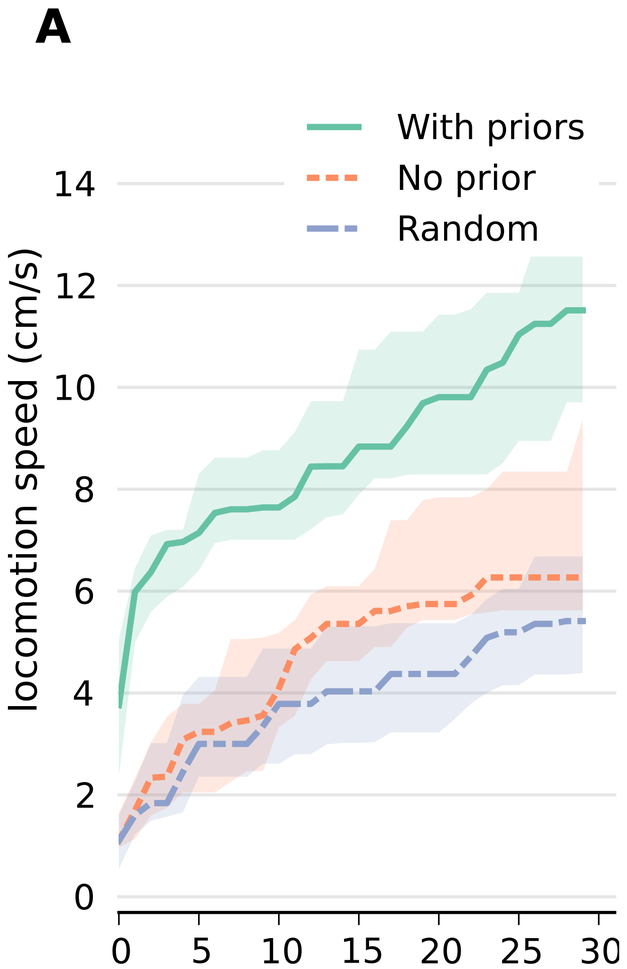
- We expect that the fastest gaits will involve a combination of motor at full speed or near full speed

mean function:



Rieffel, John, and Jean-Baptiste Mouret. Soft tensegrity robots. Soft Robotics. 2018

Results (30 evaluations)



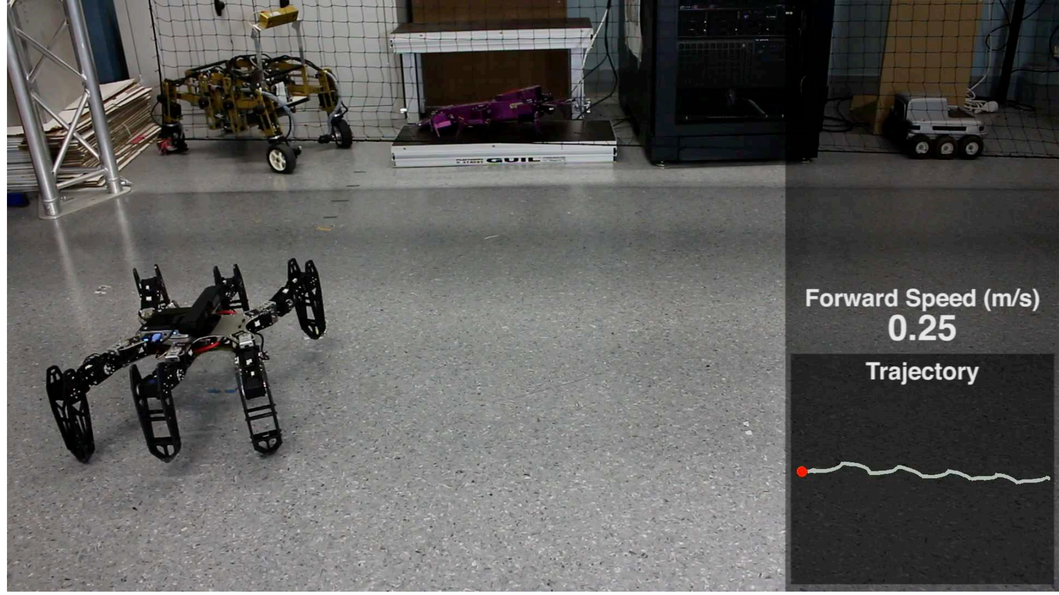
Rieffel, John, and Jean-Baptiste Mouret. Soft tensegrity robots. Soft Robotics. 2018

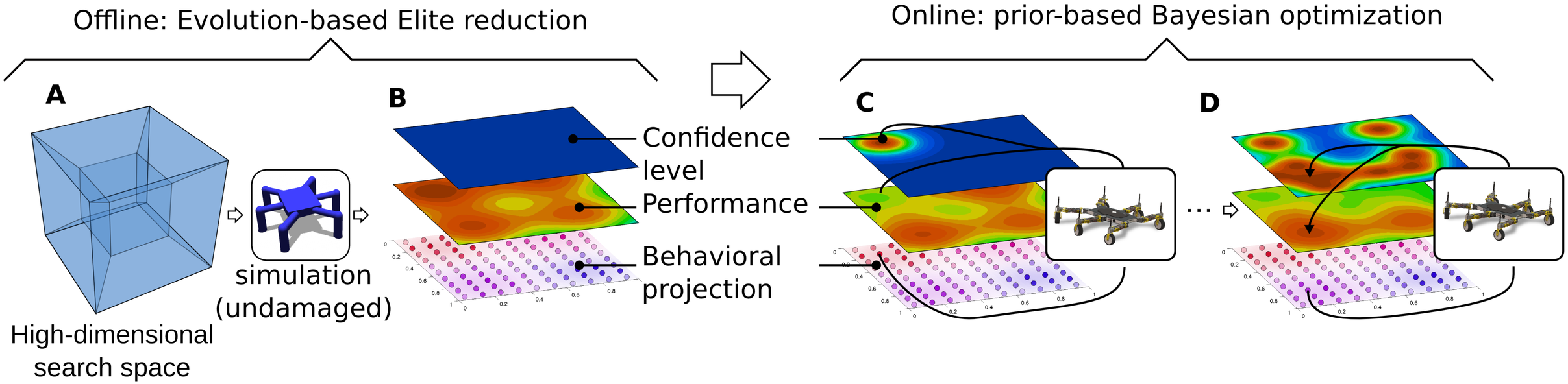
Scaling up with MAP-Elites

- **Concept:** use the simulator as the mean function

… but the problem is too high-dimensional (e.g. d=36)

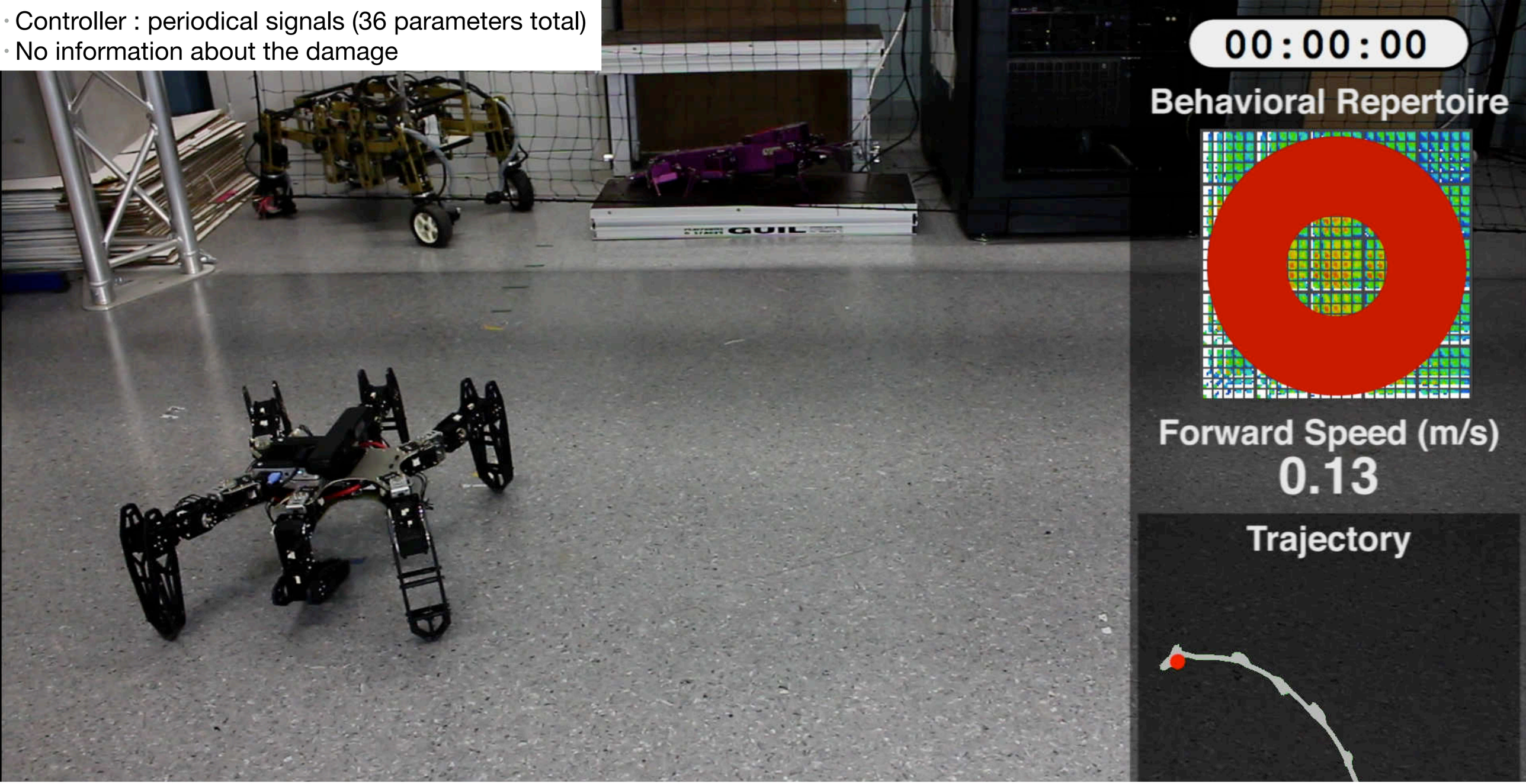
… GP do not work well in more than 4-6 D!



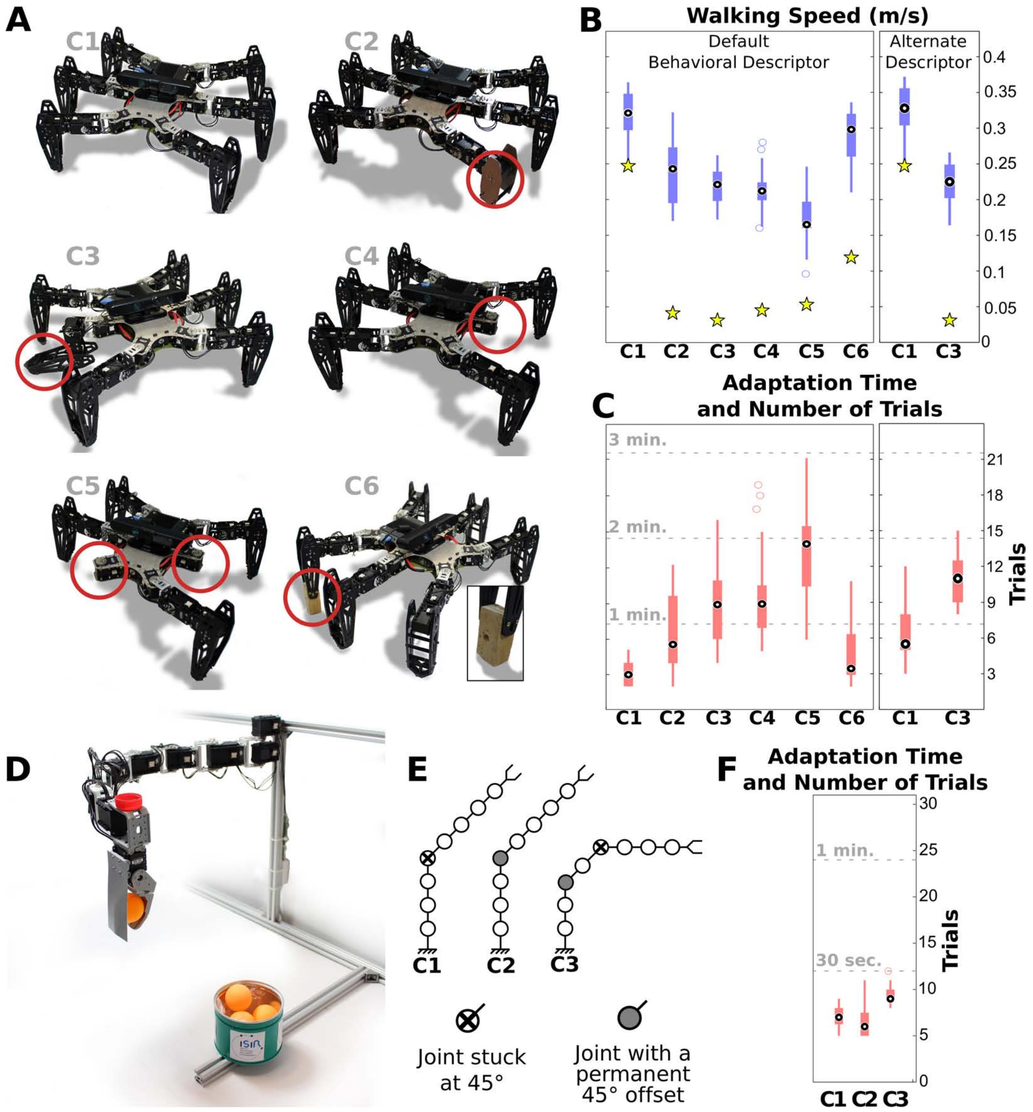


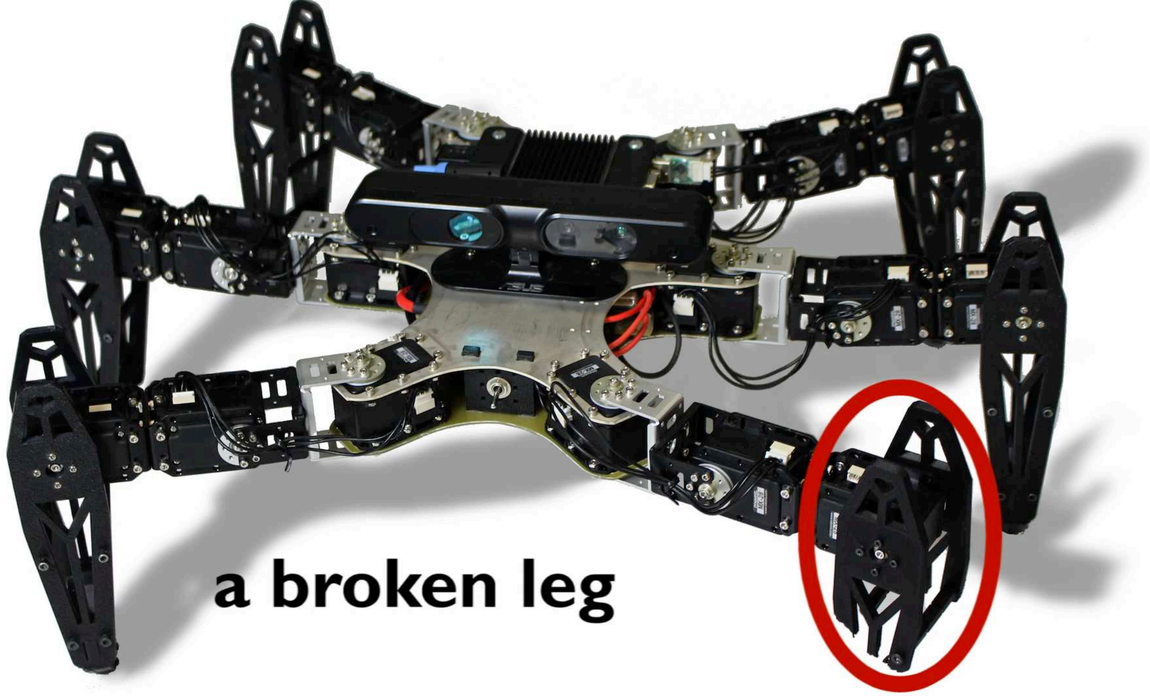
Cully, A. and Clune, J. and Tarapore, D. and Mouret, J.-B. (2015). Robots that can adapt like animals. Nature. Vol 521 Pages 503-507.

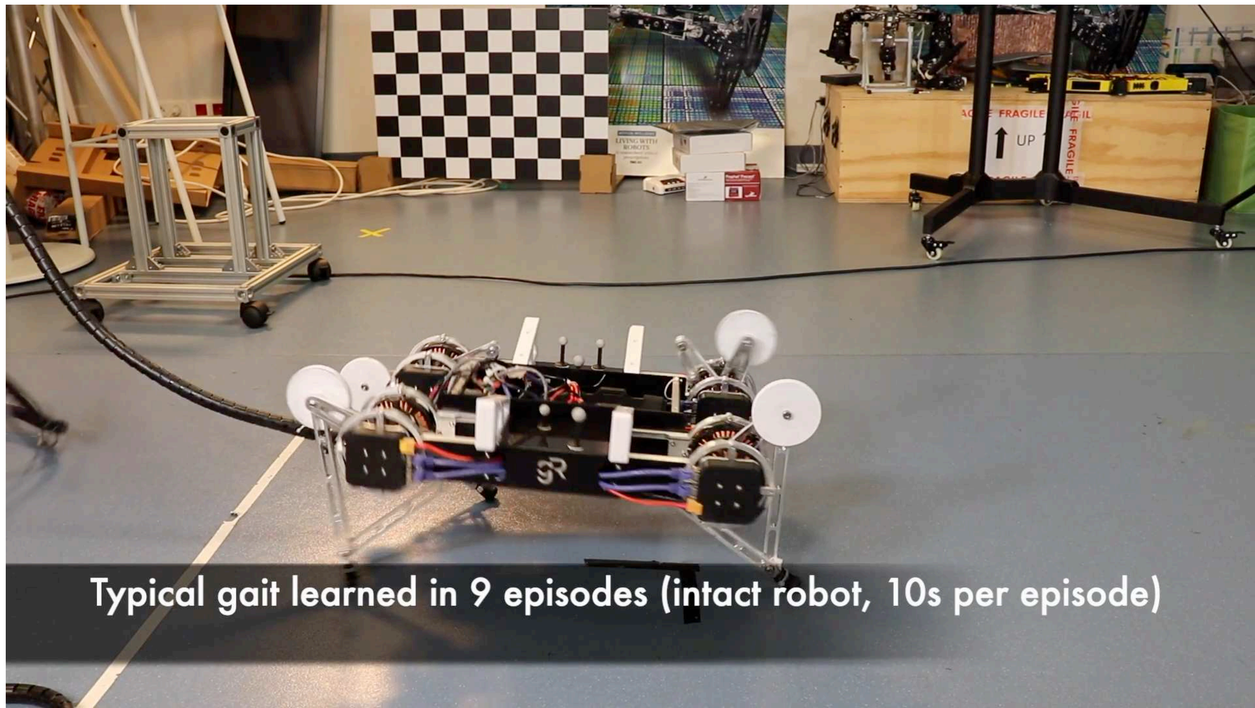
- Controller : periodical signals (36 parameters total)
- No information about the damage



Cully, A. and Clune, J. and Tarapore, D. and Mouret, J.-B. (2015). *Robots that can adapt like animals*. **Nature**. Vol 521 Pages 503-507.







Cully, A. and Clune, J. and Tarapore, D. and Mouret, J.-B. (2015). *Robots that can adapt like animals*. **Nature**. Vol 521 Pages 503-507.
Dalin, Eloïse, Pierre Desreumaux, and Jean-Baptiste Mouret. "Learning and adapting quadruped gaits with the" Intelligent Trial & Error" algorithm." (2019).

Extensions — multiple priors

- Objective: use several priors (e.g., potential damage), that is, several maps
- Change the acquisition function:

Expected improvement

Likelihood of the prior

$$\mathrm{EIP}(\boldsymbol{x}, \mathcal{P}) = \mathrm{EI}(\boldsymbol{x}) \times P(\boldsymbol{f}(\boldsymbol{x}_{1..t}) \mid \boldsymbol{x}_{1..t}, \boldsymbol{\mathcal{P}}(\boldsymbol{x}_{1..t}))$$

$$\mathrm{MLEI}(\boldsymbol{x}, \mathcal{P}_1, \cdots, \mathcal{P}_m) = \max_{p \in \mathcal{P}_1, \cdots, \mathcal{P}_m} \mathrm{EIP}(\boldsymbol{x}, p)$$

best combination

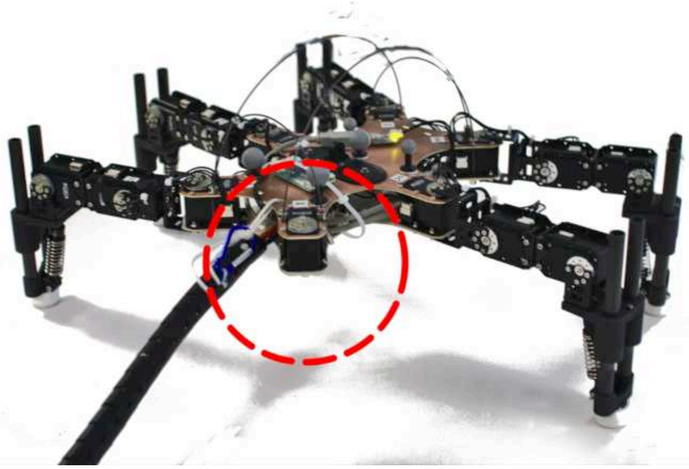
Experiment 1
damage recovery

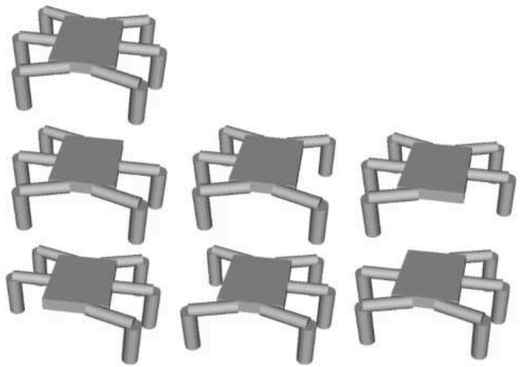
Unknown damage condition

105 priors (15 for each condition)

Reward: walking distance

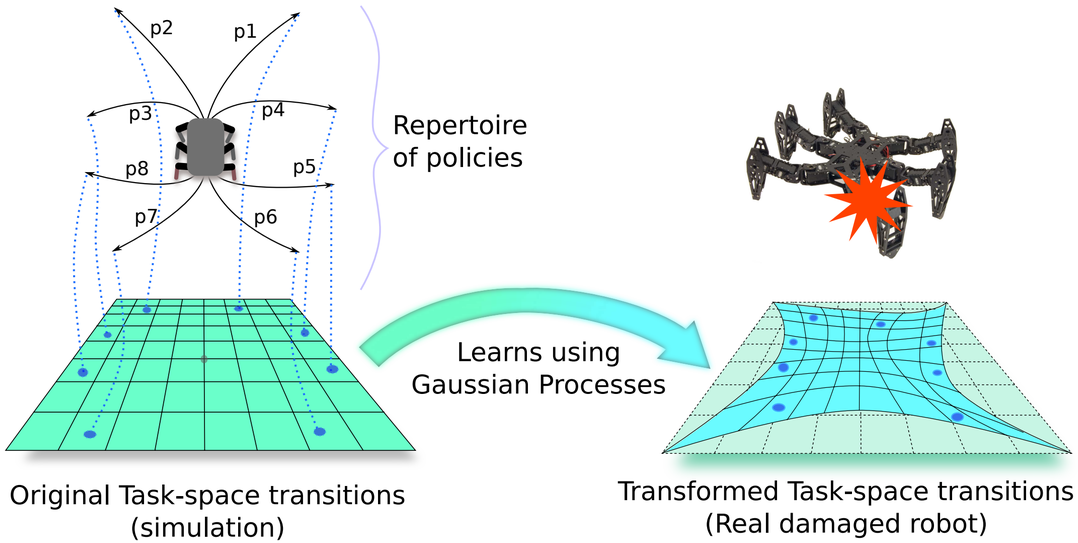
BO with MLEI acquisition function



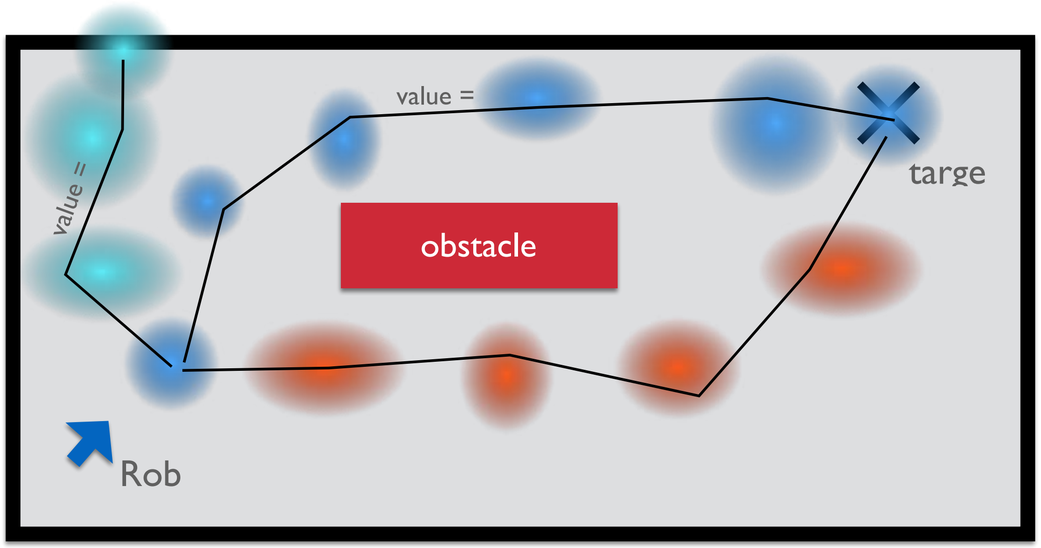


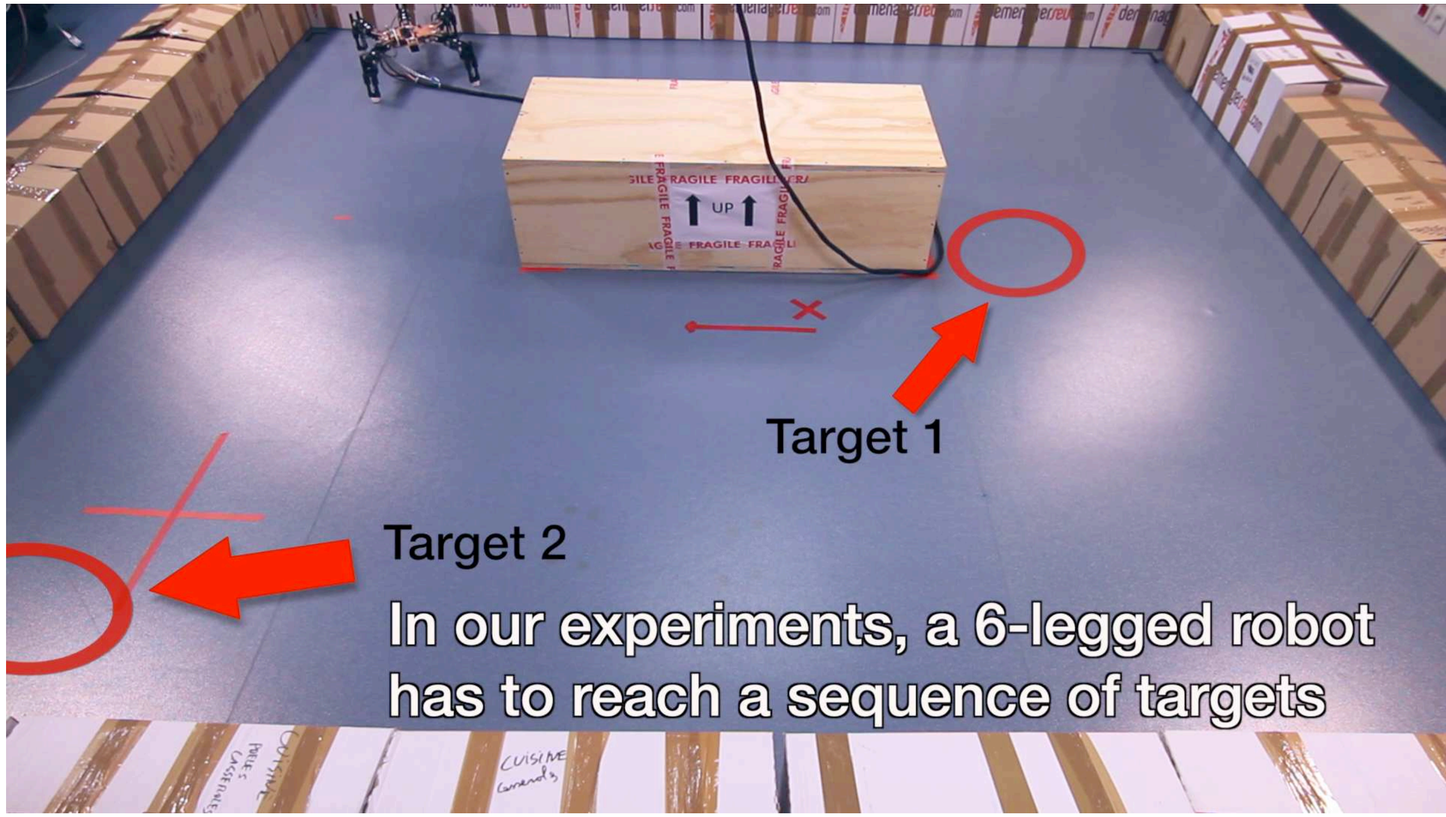
Pautrat, R., Chatzilygeroudis, K., & Mouret, J.-B. (2018). Bayesian Optimization with Automatic Prior Selection for Data-Efficient Direct Policy Search. Proc. of IEEE ICRA.

Extension — MCTS instead of BO



- learn without reset
  … while taking the environment into account (obstacles)
- “learn while doing”: trials useful for the task





Chatzilygeroudis K, Vassiliades V, Mouret JB. Reset-free Trial-and-Error Learning for Robot Damage Recovery. Robotics and Autonomous Systems. **2018** Feb 28;100:236-50.

Conclusion Bayesian optimization

Learn a surrogate of the reward function

Effective when:

- very low budget (< 500 or < 1000, good for 50)
- low dimensionality (< 6, see hand-designed policies)

Priors are key:

- smoothness prior
- priors by using a mean function (e.g. from simulation)
- combination with MAP-Elites (IT&E)

Parameters of the policy

$$J(\boldsymbol{\theta}) = \mathbb{E}\left[\sum_{t=1}^{T} r(\mathbf{x}_t) \middle| \boldsymbol{\theta}\right]$$

Reward

Strategy : surrogate of the objective

Model: $p(J(\boldsymbol{\theta})|D_{1,\cdots,n})$