Numerical Design of a Chaotic System—Tilt-A-Whirl

ID:4873

November 16, 2017

Abstract

In this project we design a new Tilt-A-Whirl using numerical optimization techniques. The objective is to maximize the standard derivation of the angular velocity of each car on the platform. The movement for a single car is governed by a second-order ordinary differential equation (ODE). The challenge arises from the fact that when solved numerically, the system exhibits deterministic chaos in the angular velocity of each car. To address this problem, a Gaussian-processing-based surrogate model is used instead of the standard derivation of the angular velocity. To train the surrogate model, we use the Latin hypercube sampling as the method of experiment design. Last, the ODE and optimization problem are solved with Matlab® build-in tools, `ode45` and `fmincon`, respectively.

1 Physics Model

As mentioned, the movement of a single car is determined through a second-order ODE [1]:

$$R = \frac{\mathrm{d}^2\phi}{\mathrm{d}\tau^2} + \frac{\gamma}{Q_0}\frac{\mathrm{d}\phi}{\mathrm{d}\tau} + (\epsilon - \gamma\alpha)\sin\phi + \gamma\beta\cos\phi = 0, \tag{1}$$

where ϕ is the relative position of the car with respect to the beam, and

$$\begin{aligned} \tau &= 3\omega t, \\ \gamma &= \frac{1}{3\omega}\sqrt{\frac{g}{r_2}}, \\ \epsilon &= \frac{r_1}{9r_2}, \\ \alpha &= \alpha_0 - \alpha_1\cos\tau. \end{aligned} \tag{2}$$

Parameters τ, γ and ϵ are introduced to non-dimensionalize equation (16) in [1]. The standard deviation of the angular velocity, $\sigma = \sigma(\frac{\mathrm{d}\phi}{\mathrm{d}t})$, is our objective function. Some variables are shown in Figure 1. For more information, we refer to [1].

As can be see, there are 6 independent parameters in the ODE (1), 3 of which are our design variables while the rest 3 are fixed. Specifically, the design parameters, ω, r_2, and α_1, are constrained by their lower and upper bounds:

$$\begin{aligned} \frac{\pi}{10} \le \omega \le \frac{4\pi}{15} \\ 0.1 \le r_2 \le 1.5 \\ 0 \le \alpha \le 0.3, \end{aligned} \tag{3}$$

while the rest are fixed:

$$\begin{aligned} r_1 &= 4.3 \\ \alpha_0 &= 0.036 \\ Q_0 &= 20 \end{aligned} \tag{4}$$

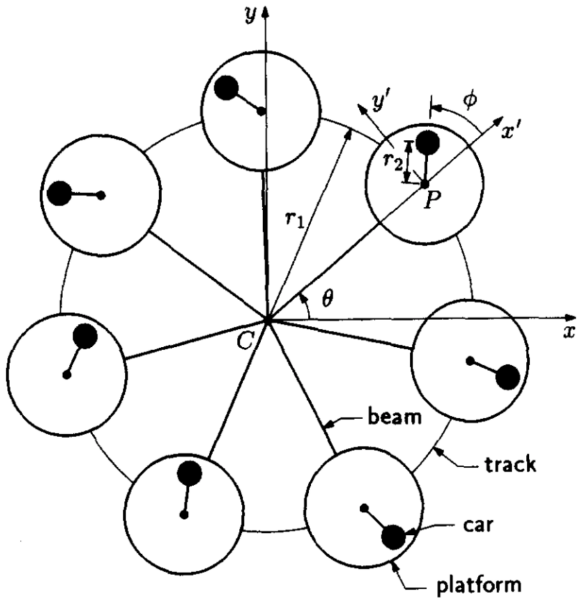

Figure 1: Plan view of the idealized Tilt-A-Whirl [1]

2 Mathematical Model

We start with introducing some notations used in this work. A scalar is denoted as a plain font; bold fonts indicate a variable is a vector, and a subscript number is used to denote the entry of the vector; e.g., $\boldsymbol{p}_i$ is the ith component of vector $\boldsymbol{p}$.

2.1 Optimization problem

To use an numerical optimization technique, the physics model in Section 1 must be rewritten into a mathematical model. The one used in this work is as follows:

$$\begin{aligned} max\ & \sigma(\frac{\mathrm{d}\phi}{\mathrm{d}\tau}) \\ s.t.\ & \boldsymbol{p}_l \leq \boldsymbol{p} \leq \boldsymbol{p}_u \\ & R(\phi, \boldsymbol{p}) = 0 \end{aligned} \tag{5}$$

where σ —the standard deviation of $\frac{\mathrm{d}\phi}{\mathrm{d}\tau}$— is the objective function, $\boldsymbol{p} = \begin{bmatrix}\omega & r_2 & \alpha_1\end{bmatrix}^T$ is the design variable with lower and upper bounds being $\boldsymbol{p}_l$ and $\boldsymbol{p}_u$, respectively, and $R(\phi, \boldsymbol{p})$ is the ODE (1) which serves as a constraint.

3 Analysis model

3.1 Solution to the ODE

To solve (1) numerically, we first rewrite it into a first-order system:

$$\frac{\mathrm{d}\boldsymbol{\psi}}{\mathrm{d}\tau} = \begin{bmatrix} F(\boldsymbol{\psi}) \\ \boldsymbol{\psi}_1 \end{bmatrix} \tag{6}$$

where

$$\begin{aligned} \boldsymbol{\psi} &= \begin{bmatrix} \frac{\mathrm{d}\phi}{\mathrm{d}\tau} & \phi \end{bmatrix}^T \\ F(\boldsymbol{\psi}) &= \frac{\mathrm{d}\boldsymbol{\psi}_1}{\mathrm{d}\tau} + + \frac{\gamma}{Q_0}\boldsymbol{\psi}_1 + (\epsilon - \gamma\alpha)\sin\boldsymbol{\psi}_2 + \gamma\beta\cos\boldsymbol{\psi}_2. \end{aligned} \tag{7}$$

(6) is then solved using `ode45`, giving us both ϕ and $\frac{\mathrm{d}\phi}{\mathrm{d}\tau}$ as functions of τ.

3.2 Surrogate model – Gaussian processing

As mentioned, the dynamic system governed by (1) displays chaotic behavior, which prevents the direct employment of the gradient-based optimization methods because the numerical gradients would see significant oscillations and the function would be nonsmooth. To address this problem, we use the Gaussian processing as a surrogate model to replace the original objective function σ.

Practically, there are at least 3 factors affecting the behavior the of Gaussian processing:

- sampling
- covariance function
- likelihood function

We use the Latin hypercube approach Specifically, the Gaussian processing is trained with a sample generated by Latin hypercube approach. As for covariance function, we choose the Matern Covariance function. Finally, Gaussian likelihood is used as likelihood function. Note that a maximization/minimization problem is solved to determine the hyperparameters in the likelihood funciton.

3.3 Optimization

Since the surrogate function is smooth, we can use a gradient-based method to solve (5). On the other hand, since the gradient information of the surrogate is usually not available, it is calculated inside `fmincon` using finite difference methods. The default algorithm, interior-point, is chosen.

4 Implementation

Attention is paid to the modularity of the code; that is, each components is almost independent of the others, and hence, can be completely replaced by another implementation or definition. This gives us the portability and reusability to some extend. For example, function `sampling` which implements the Latin hypercube approach, can be replaced by Monte-Carlo as long as the API remains unchanged. In this way, the problem is decomposed into the following parts

- `eqn27` defines the ODE
- `objFunc` computes the objective, i.e., the standard deviation, by calling `ode45` which itself takes `eqn27` as an input
- `sampling` generates the data for training `gaussianProcessing`; it takes `objFunc` as input
- `gaussianProcessing` takes the sampling results and outputs the hyperparameters *hyp* which is used to define `surrogate`
- `surrogate` is our surrogate model, i.e., the actual objective function

Once these components are defined, the remaining work is to build the workflow of the optimization process. For example, the optimization tool, `fmincon`, takes a function which itself takes only design variables as parameters. This is different from `gp`. Therefore, we need to define a wrapper functions, `surrogate` in our case, to fit API of `fmincon`.

The workflow used in this work is shown in Algorithm 1, and the implementation in Matlab is attached in Appendix A∼G. It is noted that in operation 10 and 11 we update the sampling by appending a new sampling around the newly obtained optimizer $\boldsymbol{p}^{iter,*}$, rather than just $\boldsymbol{p}^{iter,*}$ itself. This will improve the approximation of the surrogate in a subdomain surrounding $\boldsymbol{p}^{iter,*}$, which is motivated by the assumption that the final optimizer is close to $\boldsymbol{p}^{iter,*}$.

Algorithm 1: Optimization Algorithm with Surrogate

1 $iter \leftarrow 1$;
2 `sampling`;
  Input : *sample_size*
  Output: $\boldsymbol{p}^{iter}$, σ^{iter}
3 **while** $iter \leq max\_iter$ **do**
4 | `gaussianProcessing` ();
  | **Input :** $\boldsymbol{p}^{iter}$, σ^{iter}
  | **Output:** *hyp*
5 | `obj_func` = @($\boldsymbol{p}$) `surrogate`(*hyp*, $\boldsymbol{p}^{iter}$, σ^{iter});
6 | call `fmincon`, get $\boldsymbol{p}^{iter,*}$;
7 | **if** *criteria_satisfied* **then**
8 | | break
9 | **else**
10 | | new sampling ($\boldsymbol{p}^{iter,new}$, $\sigma^{iter,new}$) of size $0.2 * sample\_size$ around $\boldsymbol{p}^{iter,*}$;
11 | | update sampling: $\boldsymbol{p}^{iter+1} = \boldsymbol{p}^{iter} \cup \boldsymbol{p}^{iter,new}$, $\sigma^{iter+1} = \sigma^{iter} \cup \sigma^{iter,new}$
12 | **end**
13 | $iter \leftarrow iter + 1$
14 **end**

5 Results and discussion

The initial sample size is set to be 50; the initial value is set to be $\begin{bmatrix}13/60 * pi & 0.8 & 0.058\end{bmatrix}^T$, which corresponds to a direct objective 1.5274. Since the algorithm we use is adaptive, the sample size will keep increasing by 10 individuals for each iteration.

The convergence history of the objective approximated by the surrogate and the direct objective is plotted in Figure 2. We can see that the surrogate objective flattens after significant oscillations in the first 7 iterations. On the other hand, the direct objective does not see comparable oscillations. After 20 iterations, the direct objective improves by around 75% compared with the initial objective, which proves the justification of the current approach. However, even with an adaptive sampling, the difference between the direct objective and the approximate objective fails to significantly reduce. This may be due to the fact that the direct objective is a nonsmooth and chaotic function, and any smooth surrogate models including the Gaussian processing used in this project is inherently not precise.

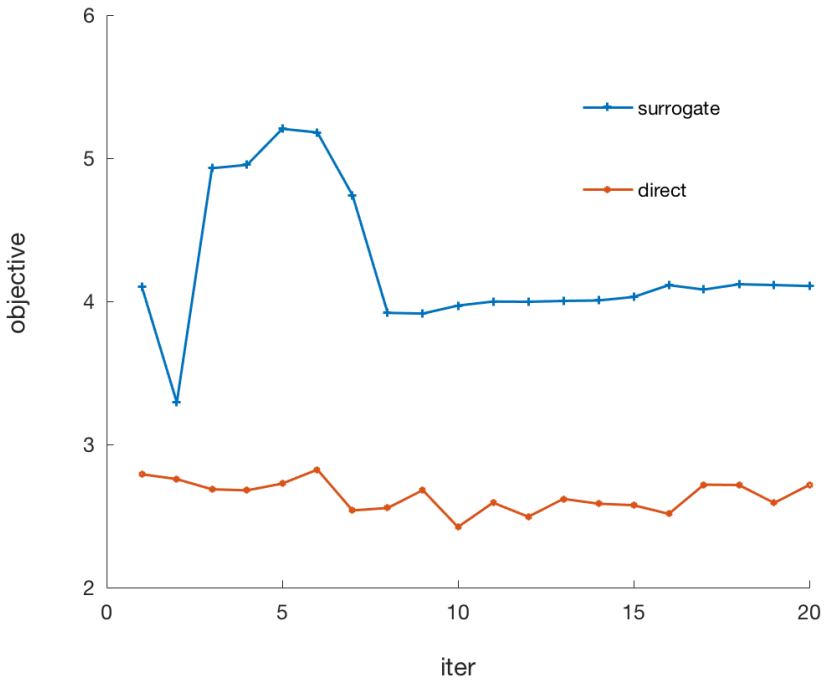

Figure 2: Convergence history of objectives

6 Conclusion

In this work, we use numerical optimization design technique to maximum the Tilt-A-Whirl, a deterministic chaotic system. The Gaussian processing is used as a surrogate model, sampled with the Latin hypercube approach. To improve the performance of the surrogate, we update the sampling in an adaptive way. Although the surrogate is improved little in terms of approaching the direct objective, we do observe an significant improvement in the direct objective value after the optimization.

References

[1] Kautz, R. L. and Huggard, B. M., "Chaos at the amusement park: Dynamics of the TiltAWhirl," *American Journal of Physics*, Vol. 62, No. 1, 1994, pp. 59–66.

Appendices

A eqn27.m

```
function dydt = eqn27( tau, y, omega, r2, alpha1 )
% Reference:
% Chaos at the amusement park: Dynamics of the Tilt?A?Whirl
% by R. L. Kautz.
```

```
%  Given parameters
Q0 = 20;
alpha0 = 0.036;
r1 = 4.3;
g = 9.8;
% derived parameters
% tau = 3 * omega * t;
gamma = 1 / (3*omega) * sqrt(g/r2);
eps = r1 / (9*r2);
alpha = alpha0 - alpha1 * cos(tau);
beta = 3 * alpha1 * sin(tau);
% rhs of ode
c1 = -gamma / Q0;
c2 = -(eps - gamma*gamma * alpha);
c3 = - gamma*gamma * beta;
dydt = [y(2); c1*y(2) + c2*sin(y(1)) + c3*cos(y(1))];
```

B objFunc.m

```
function [ obj ] = objFunc( x )
omega  = x(1);
r2     = x(2);
alpha1 = x(3);
my_ode = @(t, y) eqn27(t, y, omega, r2, alpha1);
T1 = 0.0;
T2 = 1000.0;
% nSteps = 1000;
% t = linspace(T1, T2, nSteps+1)';
y1_0 = 0.0;
y2_0 = 0.0;
[~, y] = ode45(my_ode, [0, 50], [y1_0; y2_0]);
y0 = y(end, :);
[t, y] = ode45(my_ode, [T1, T2], y0);

% the average
dT = T2 - T1;
y2 = y(:, 2);
y2_avg = mean(y2); % trapz(t, y2) / dT;
deviation = trapz(t, (y2 - y2_avg).^2) / dT;
deviation = sqrt(deviation);
obj = -3*omega * deviation;
end
```

C sampling.m

```
function [ var, f ] = sampling( sample_size, lb, ub, func )
if size(lb, 2) ~= size(ub, 2)
  error('size(lb, 1) ~= size(ub, 1)');
end
nvar = size(lb, 2);
if nvar == 0
  error('number of variable is zero!')
end
% sampling using Latin hypercube
x = lhsdesign(sample_size, nvar);
% convert from [0, 1] to [lb, ub]
```

```
var = zeros(sample_size, nvar);
for i = 1 : nvar
  h = ub(i) - lb(i);
  var(:, i) = lb(i) + x(:,i) * h;
end
% compute function value
f = zeros(sample_size, 1);
for i = 1 : sample_size
  f(i) = func(x(i, :));
end
end
```

D gaussianProcessing.m

```
function [hyp] = gaussianProcessing(x, y)
% this is needed if you are not running
% in the Libraries root directory
mydir = '~/Downloads/gpml-matlab-v4.0/';
addpath(mydir(1:end-1))
addpath([mydir,'cov'])
addpath([mydir,'doc'])
addpath([mydir,'inf'])
addpath([mydir,'lik'])
addpath([mydir,'mean'])
addpath([mydir,'prior'])
addpath([mydir,'util'])

% set the squared exponential covariance function
covfunc = {@covMaterniso, 1};
% first component is log(l) and second is log(sigma)
hyp.cov = [log(1/4); log(1.0)];

% set the likelihood function to Gaussian
likfunc = @likGauss;
sn = 0.05; %1e-16; % this is the noise level
hyp.lik = log(sn);

% maximize the likelihood function to find the hyperparameters
hyp = minimize(hyp, @gp, -100, @infExact, [], covfunc, likfunc, x, y);
end
```

E surrogate.m

```
function [ f ] = surrogate(hyp, x, f, z)
covfunc = {@covMaterniso, 1};
likfunc = @likGauss;
m = gp(hyp, @infExact, [], covfunc, likfunc, x, f, z);
f = m(1);
end
```

F run_opt.m

```
function [a, fmin] = run_opt(x, f)
% initial value, lower and upper bounds
x0 = [13/60*pi; 0.8; 0.058];
lb = [pi/10, 0.1, 0];
ub = [4*pi/15, 1.5, 0.3];

% gaussian processing
fprintf('Gaussian processing...\n');
hyp = gaussianProcessing(x, f);
obj_gp = @(z) surrogate(hyp, x, f, z);

% optimization options
fprintf('Optimizing with surrogate function...\n')
options = optimoptions('fmincon', 'Display','iter', 'MaxIter', 150);
problem.options = options;
problem.solver = 'fmincon';
problem.objective = obj_gp;
problem.Aineq = [];
problem.bineq = [];
problem.lb = lb;
problem.ub = ub;
problem.x0 = x0;

% run fmincon
[a, fmin] = fmincon(problem);
end
```

G getGeomConstraints.m

```
% lower and upper bounds
lb = [pi/10, 0.1, 0];
ub = [4*pi/15, 1.5, 0.3];
box = ub - lb;

% initial sampling
fprintf('initial sampling...\n');
sample_size = 50;
[x, f] = sampling(sample_size, lb, ub, @objFunc);

% try multiple runs
n_runs = 20;
f_gp = zeros(n_runs);
f_ex = zeros(n_runs);
a = zeros(3, 1);
tol_1 = 1.e-3;
tol_2 = 1.e-3;
fold = 1.0e10;
for i = 1 : n_runs
  fprintf('opt_run = %i\n', i);
  [a, f_gp(i)] = run_opt(x, f);
  f_ex(i) = objFunc(a);
  fprintf('fe = %d, fgp = %i\n', f_ex(i), f_gp(i));

  if abs(f_ex(i) - fold) < tol_1 && abs(f_ex(i) - f_gp(i)) < tol_2
    break;
  end
  fold = f_gp(i);
  lb_i = a - 0.2*box;
  ub_i = a + 0.2*box;
```

```
  lb_i = min(lb_i, lb);
  ub_i = max(ub_i, ub);
  [x_i, f_i] = sampling(sample_size*0.2, lb_i, ub_i, @objFunc);
  x = [x; x_i];
  f = [f; f_i];
  %     x = [x; a'];
  %     f = [f; fobj];
end

iter = (1 : 1 : n_runs);
plot(iter, -f_gp);
hold on;
plot(iter, -f_ex);
hold off;
```