Real Time On Board Flight Path Optimization Under Constraints Using Surrogate Flutter Function

Arthur Paul-Dubois-Taine*, Stanford University, Stanford, CA 94305

A real time approach for aircraft flight path planning is considered. The approach uses a dynamic programming optimization to find the best possible trajectory in a discretized state space under flight envelope, fuel and flutter constraints. In order to perform real-time evaluation of the flutter constraint function, a surrogate model of the flutter function is introduced and trained using various machine learning techniques. In particular, both regression and classification approaches are considered. In addition, due to the high cost of generating data, sampling issues will be considered. At first, a simple a priori of the feature space is used to generate the training set. Then a greedy sampling approach using an appropriate criterion along with Kmeans clustering is introduced and used to generate a better training set.

I. Introduction

This project addresses the issue of airplane real-time flight path optimization. Flight path optimization has numerous applications. Airlines dispatchers use simple flight path optimizations algorithms daily to decide how to fly their airplanes and save on fuel. Military airplanes rely on such algorithms to minimize their time to target while minimizing operational risks and stress on the airplane structure. Aircraft designers use it as a design tool to improve performances. However, currently, most flight path optimization algorithms require somewhat involved numerical simulations which must be run from the ground a priori using estimated flight data, leaving the pilot with little ability to re-optimize his flight path in real-time based on actual flight conditions or in case of unplanned circumstances. There is a growing interest in designing so called real-time flight path optimization algorithms that can instantly compute optimal trajectories while satisfying some flight constraint such as remaining in the aircraft flight envelope (i.e. below a certain speed-altitude combination), fuel level constraints (the aircraft must not run out of fuel), or avoiding flutter for example. However, real-time flight path optimization is only possible if all constraint and objective functions can be evaluated cheaply.

In an optimization context, multiple queries must be made to both the objective and constraint functions. However, some of the constraints can only be evaluated using expensive computer simulations on large computer clusters making the algorithm impossible to embed on the airplane computer without modification. In particular, the so-called flutter function is extremely expensive to compute as it requires a full 3D simulation of the flow around the aircraft as well as its structure. In this project, machine learning techniques will be used to construct a cheap proxy of the flutter function in order to dramatically reduce the cost of the optimization routine. The aircraft considered is the Lockheed Martin F-22 Raptor fitted with an external fuel tank.

A. Background Information on Aerodynamic Flutter

Flutter is an *aeroelastic* phenomenon that can occur in airplanes when oscillations in the fluid flow around the aircraft excites its internal structure close to its natural modes of vibrations. In such cases, the amplitude of vibration of the structure can be increased to the point where the wing can rupture. Flutter has been the cause of many catastrophic failures in aircraft development history, and is one of the main concerns of aircraft designers. One of the most famous flutter failures to occur is shown in figure (1). The accident happened on a Northrop F-89 Scorpion aircraft during a flypast at the International Aviation Exposition in 1952. More recently the Boeing F18 aircraft has faced flutter issues, which forces its rear tail to be replaced after a couple hundred hours of flight. In general, flutter is a function of speed (also called Mach Number M_{∞}), altitude h and fuel level f (fuel level influences the vibration modes of the structure).



Figure 1: F-89 Scorpion Crash due to Flutter Failure

Variable Name	Symbol	Unit
Speed (Mach Number)	M_{∞}	None
Altitude	h	[ft]
Fuel Level	f	[%]

Table 1: Features of the Problem

^{*}Graduate Research Assistant, Department of Aerospace Engineering, Farhat Research Group, artpdt@stanford.edu

II. Learning the Flutter Function

A. Features

As explained above, computing the flutter is extremely expensive and cannot be done in real time on an aircraft computer. In this section, the flutter evaluation routine will be replaced by a surrogate constructed via machine learning. In addition, the issue of sampling the training set is investigated. In particular, the idea of a greedy sampling for generating the best possible training set is explored. Mathematically, the problem of finding a proxy for the flutter function F can be expressed as that of finding some hypothesis h_{θ} such that:

$$h_{\theta}(h, M_{\infty}, f) \approx F(h, M_{\infty}, f)$$
 (1)

The features of the problem are described in table 1. In the remainder of this paper, the space of all possible (h, M_{∞}, f) will be referred to as the feature space.

B. Generating Flutter Data

1. Approach

Without going into too much details, determining whether an aircraft will experience flutter requires computing the vibration modes of the structure, solve for the fluid flow parameters around the aircraft, and finally coupling the two results to determine whether the structure is excited in a critical fashion. In this project, this was performed using two state of the art simulation codes (both open source) developed by the Farhat Research Group at Stanford University:

- AeroS for the structure simulation
- · AeroF for the fluid simulation

Both codes feaure large parallelization capabilities which make them computationally very powerful. Simulations are run on a cluster using 33 CPUs, and one run requires about 1 hour. Despite the efficiency of the code, computing flutter solutions for the test set in the discretized state space took weeks and roughly 20,000 CPu hours for the chosen aircraft.

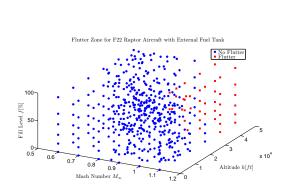
Flutter can be represented either as a continuous or a binary variable:

- Discrete representation: describes whether flutter is occuring $(F^0(h, M_\infty, f))$ or $F^1(h, M_\infty, f)$). This representation is illustrated in Figure (2)-(a) and suggests a classification approach.
- Continuous representation: flutter occurs if the so-called damping ratio is negative. Hence, instead of representing flutter, we can model the continuous damping ratio $(\delta(h, M_{\infty}, f))$ function. Then, flutter occurs if $\delta(h, M_{\infty}, f) < 1$. This representation is illustrated in Figure (2)-(b) and suggests the use of a regression like approach.

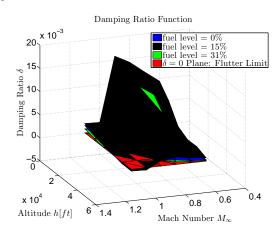
Both approaches will be examined here.

2. Test Set

In order to test the accuracy of the different training procedures, a test set has been generated by running the exact flutter function for 500 different (M_{∞}, h, f) combinations. 216 $(6 \times 6 \times 6)$ of the test data were sampled evenly in the state space, and the remaining 284 were randomly placed using Latin Hypercube Sampling [1].



a) Binary Flutter Representation



b) Continuous Flutter Representation via Damping Ratio

Figure 2: Binary and Continuous Flutter Representation

C. Training Using Direct Sampling Methods

Sampling is an important issue in this problem because the flutter function is expensive to compute, and the size of the training set should therefore be kept to a minimum. In this section, the flutter function will be learned using a training set generated via a a priori sampling, also called direct sampling. Here, the training set is sampled using a mixture of $N_{uniform}$ uniformly placed points in feature space, and N_{LHS} Latin Hypercube Sampling (LHS) points [1]. The goal of this approach is to cover the feature space as evenly as possible while still allowing some random feature space exploration through the LHS. Using this approach, the chances of finding *interesting* training samples are maximized.

1. Classification Approach

Here, flutter is treated as a binary variable $(F^0(h, M_\infty, f))$ or $F^1(h, M_\infty, f)$). A Support Vector Machine (SVM) model is chosen for the classification, and trained on data sets of various sizes. Since flutter can result in catastrophic failures of the aircraft, there is a very low tolerance for misclassifications, and the relaxation constraint of the SVM is therefore set to zero (i.e. $\xi_i = 0$). Each hypothesis is evaluated using the 500 test samples. The percentage and number of misclassified points are reported. Table 2 shows the accuracy for different SVM Kernels for a given sampling strategy. The best Kernel (Polynomial) is retained and used for the remainder of this section. Different sampling strategies are then tested as shown in table 3. Results show that the number of uniformly placed samples has little influence on the testing error. This is because uniform points are typically located near the feature space boundary, which means far from the decision boundary. On the other hand, adding extra LHS points produces more informational data, and has a better chance of helping SVM identify the correct decision boundary. In addition, as expected, results vary as the total number of samples increases. The most accurate SVM hypothesis for 25, 42 and 50 samples have 42, 6 and 2 misclassifications respectively.

SVM Kernel	Error [%] $(N_{misclass.}/500)$
Gaussian	11.8(59/500)
Linear	11.4(57/500)
Quadratic	11.0(55/500)
Polynomial	8.4(42/500)
Multilayer Perceptron	13.6(68/500)

$N_{uniform}$	N_{LHS}	Error [%] $(N_{misclass.}/500)$
8 (2x2x2)	17	8.4(42/500)
27 (3x3x3)	17	7.8(39/500)
8 (2x2x2)	34	1.4(7/500)
27 (3x3x3)	34	1.2(6/500)
8 (2x2x2)	42	.4(2/500)

Table 2: SVM Classification Error for $N_{uniform} = 2 \times 2 \times 2$, **Table 3:** SVM Classification Error for Various Sampling Strate- $N_{LHS} = 17$ and Various Kernels gies, Polynomial Kernel

2. Regression Approach

The flutter function can also be represented as a continuous variable: the damping ratio δ , as shown in Figure (2)-(b). If the damping ratio is negative, flutter occurs, i.e. $F = 1\{\delta(h, M_\infty, f) < 0\}$. The same sampling strategy as previously is applied. The damping ratio hypothesis is trained using a Kriging model (the DACE Matlab Toolbox [3] was used here) which uses a combined regression-correlation approach to represent the data. Kriging models have been shown to be efficient at modelling deterministic computer models [2]. The model is fitted via Maximum Likelihood and also comes with a built-in error estimator. As for SVM, multiple Kriging kernels and regression polynomials are then tested as shown in table 4. An exponential kernel along with a 1^{st} order polynomial is retained for the remainder of this section. Table 5 then shows the effect of sampling on the resulting hypothesis accuracy. The same trends as for the SVM approach are noted, but the overall accuracy of the regression based approach seems to be better with 24, 6 and 1 misclassification for 25, 42 and 50 training samples respectively. Therefore, for the remainder of this paper, the regression approach is retained.

$P^{(*)}$	Kriging Kernel	Error [%] $(N_{misclass.}/500)$
0	Gaussian	6.6(33/500)
1	Gaussian	5.6(28/500)
2	Gaussian	10.2(51/500)
0	Exponential	5.8(29/500)
1	Exponential	4.8(24/500)
2	Exponential	8.0(40/500)
0	Spline	6.6(33/500)
1	Spline	5.4(27/500)
2	Spline	8.8(44/500)

$N_{uniform}$	N_{LHS}	Error [%] $(N_{misclass.}/500)$
8 (2x2x2)	17	4.8(24/500)
27 (3x3x3)	17	3.8(19/500)
8 (2x2x2)	34	1.2(6/500)
27 (3x3x3)	34	.6(3/500)
8 (2x2x2)	42	.2(1/500)

Table 5: Kriging Model Error for Various Sampling Strategies, Exponential Kernel and 1^{st} Order Regression Polynomial

Table 4: Kriging Model Error for $N_{uniform} = 2 \times 2 \times 2$, $N_{LHS} = 17$ and Various Kernels. * *P is the order of the regression polynomial in the Kriging model.*

D. Greedy Sampling Approach

Direct sampling approaches such as those presented in section II B are simple, but have one major shortcoming: they do not include any information about the specificities of the objective function, which makes them somewhat inefficient. For example, in the present case, the only region of the feature space that matters is the decision boundary. Adding training sample points far from the decision boundary has essentially no effect on the accuracy of the hypothesis. On the other hand, sampling points near the decision boundary will improve the hypothesis significantly. However, the is no way to know the position of the decision boundary a priori. In order to progressively locate the decision boundary and refine the training set in its vicinity, a greedy sampling approach is now proposed.

The idea behind greedy sampling is to start with a small set of sample points in feature space and train a rough hypothesis from that using the Kriging model. The hypothesis is then evaluated to locate the decision boundary, and new sample points are queried. After that, the hypothesis is re-trained with the updated training set, and the procedure is repeated until desired accuracy is reached. In practice, this procedure allows to generate more *informative* samples than direct sampling.

1. Training Set Enrichment

At each greedy step, the goal is to refine the model near the decision boundary. In order to efficiently do so, one must:

- 1. Identify regions where the model is inaccurate (i.e. it is useless to add a new sample near an already existing one). This can be achieved by using the Mean Square Error estimate built in the Kriging model which provides an a-posteriori error estimator. Mathematically, this can be expressed at $E_{Kriging}(h, M_{\infty}, f) \approx MSE(h, M_{\infty}, f)$. The MSE is readily available from the DACE toolbox and comes at negligible extra cost for the given problem size.
- 2. Locate the decision boundary. This is achieved by using the damping values predicted by the Kriging model $\delta_{Kriging}(h, M_{\infty}, f)$ and finding their absolute minimum. Recall that flutter occurs when $\delta < 0$, therefore, identifying the decision boundary comes down to finding the roots of $\delta(h, M_{\infty}, f)$, and finding the absolute minimum is one way to do it.

Using the two above ideas, a greedy score criterion is devised. The criterion is designed to find the feature space locations (h^*, M_{∞}^*, f^*) that (i) are located in regions where the model is inaccuracte (i.e. MSE is high) and (ii) are in the vicinity of the decision boundary (δ is minimal). The selected criterion is the following:

$$(h^*, M_{\infty}^*, f^*) = argmax_{(h, M_{\infty}, f)}GS(h, M_{\infty}, f) = argmax_{(h, M_{\infty}, f)} \frac{MSE(h, M_{\infty}, f)}{\epsilon + |\delta_{Kriging}((h, M_{\infty}, f))|}$$
(2)

where ϵ is just a small smoothing term. In addition, the algorithm is able to add multiple samples per greedy iteration by looking at the N largest scores, and clustering the resulting feature space locations via K-means clustering in order to identify multiple regions to refine at once. The optimal number of clusters is automatically identified using an approach developed in [4].

2. Results

Fig. 3 presents the evolution of the hypothesis accuracy as points are greedily added to the training set. Each greedy approach is represented by the number of uniform and LHS points it started with, and the number of samples as the end of the greedy sampling. For example, the blue curve represents a greedy sampling that started with $N_{uniform} = 2x2xs2 = 8$ uniform and $N_{LHS} = 5$ LHS points. Then, $N_{Greedy} = N_{total} - Nuniform - N_{LHS} = 25 - 8 - 5 = 12$ new training samples are added using the greedy approach. Results show that the greedy approach to sampling is much more efficient than the direct one. In fact, as shown in table 6, for the same number of training samples, the best direct sampling methods have at least three times as much misclassifications than the best greedy ones. This shows that greedy sampling significantly improves the quality of the hypothesis whithout increasing the number of samples (and therefore the computational cost). This is again because the greedy sampling has the ability to locate the decision boundary and add extra samples in its vicinity.

	Best Hyp. Error [%] $(N_{misclass.}/500)$		
$N_{samples}$	Direct sampling	Greedy Sampling	
25	4.8(24/500)	1.6(8/500)	
42	3.8(6/500)	.2(1/500)	

	L
Table 6: SVM Classification Error for Various Sampling Strate-	
gies, Exponential Kernel and 1^{st} Order Regression Polynomial	T

	True Function	Proxy
N_{calls} to Function	1072	42
N_{calls} to Proxy	0	1072
Total CPU Time [CPU-s]	1.27×10^{8}	4.99×10^{6}
Speed-up Factor	25	
Speed-up Factor(*)	1×10^{7}	

Table 7: Optimization Time With and Without Flutter Proxy – Time for 1 function call: $1.18 \times 10^6 CPU - s$. Time for 1 proxy call: .01CPU - s. (*) Not including training.

III. Optimization Results

Note: there is no additional machine learning aspect in this section, but it is the motivation for this project, so I decided to include is anyways.

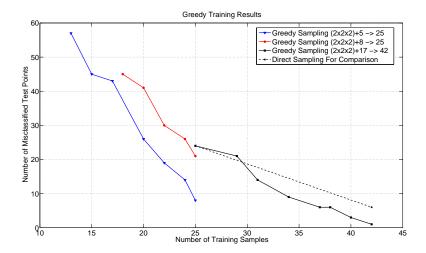


Figure 3: Greedy Training – Kriging Model – Test Set Error vs Number of Samples – Different Greedy Approaches are Represented as $(N_{uniform}) + N_{LHS} \rightarrow N_{total}$.

The best hypothesis is then selected and integrated in the optimization routine as a proxy for the flutter function. The path optimization problem is solved using a Dynamic Programming approach [5]. Fig. 4 presents optimization results and shows speed (M_{∞}) , altitude and fuel level vs. distance traveled for the best flight path. In addition, speed comparisons between optimization using the true flutter model and that using the proxy are shown in table 7. If the proxy was not trained offline previously, the training must be done offline. Despite that, the optimization is still 25 times faster than the version using the true flutter function. If the proxy is trained offline, the full optimization routine runs in a matter of seconds leading to speedups of as much as 10 millions.

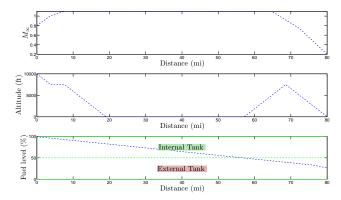


Figure 4: Optimization Results - Speed, Altitude and Fuel Level vs. Distance for Optimal Path

IV. Conclusions and Future Work

In this work, the need for a cheap proxy of the flutter function in oder to perform real-time flight path optimization is explained. To do so, both classification and regression approaches with various models and Kernels are examined and tested. The regression approach is shown to provide better accuracy for the problem of interest and is hence retained. Because data generation comes at a high cost, the size of the training is minimized via appropriate sampling. Direct and greedy sampling are explored, and greedy sampling is shown to provide superior accuracy. Finally, addition of the best proxy to the optimization routine is shown to provide speedup factors of at least 25.

In the future, the accuracy of the hypothesis could be improved further by adding gradient information to the training set. A gradient enhanced Kriging model called CoKriging could then be generated with superior accuracy. In addition, alternate greedy sampling criterions could be explored. One could imagine some kind of Cross Validation based criterion for example.

References

- [1] Iman, R., Latin Hypercube Sampling, Encyclopedia of Quantitative Risk Analysis and Assessment III, 2008.
- [2] Martin, J.D., Simpson T.W. On the Use of Kriging Models to Approximate Deterministic Computer Models, 30th Design Automation Conference Salt Lake City, Utah, USA, September 28October 2, 2004.
- [3] Lophaven et. al. DACE, A Matlab Kriging Toolbox, IMM Department, Technical University of Denmark.
- [4] T Hastie, R Tibshirani, and J H Friedman. The Elements of Statistical Learning. Data Mining, Inference, and Prediction, Second Edition. Springer, August 2009.
- [5] Amsallem et. al. Model Predictive Control Under Coupled Fluid-Structure Constraints Using a Database of Reduced Order Models on a Tablet 21st AIAA Computational Fluid Dynamics Conference, June 24-27, 2013, San Diego CA.