Euler: A System for Numerical Optimization of Programs

Swarat Chaudhuri[1] and Armando Solar-Lezama[2]

[1] Rice University
[2] MIT

Abstract. We give a tutorial introduction to Euler, a system for solving difficult optimization problems involving programs.

1 Introduction

Euler is a system for solving unconstrained optimization problems of the following form:

> Let P be a function written in a standard C-like language. Identify an input $\mathbf{x}$ to P such that the output $P(\mathbf{x})$ of P is *minimized* with respect to an appropriate distance measure on the space of outputs of P.

Many problems in software engineering are naturally framed as optimization questions like the above. While it may appear at first glance that a standard optimization package could solve such problems, this is often not so. For one, "white-box" optimization approaches like linear programming are ruled out here because the objective functions that they permit are too restricted. As for "black-box" numerical techniques like gradient descent or simplex search, they are applicable in principle, but often not in practice. The reason is that these methods work well only in relatively smooth search spaces; in contrast, branches and loops can cause even simple programs to have highly irregular, ill-conditioned behavior [1] (see Sec. 4 for an example). These challenges are arguably why numerical optimization has found so few uses in the world of program engineering.

The central thesis of the line of work leading to Euler is that *program approximation* techniques from program analysis can work together with blackbox optimization toolkits, and make it possible to solve problems where programs are the targets of optimization. Thus, programs do not have to be black boxes, but neither do they have to fall into the constricted space of what is normally taken to be white-box optimization. Specifically, the algorithmic core of Euler is *smooth interpretation* [1, 2], a scheme for approximating a program by a series of smooth mathematical functions. Rather than the original program, it is these smooth approximations that are used as the targets of numerical optimization. As we show in Sec. 4, the result is often vastly improved quality of results.

In this paper, we present a tutorial introduction to Euler. In Sec. 2, we describe, using a running example, how to use the system. In Sec. 3, we sketch the internals of the tool. Finally, Sec. 4 analyzes the results of the tool on the running example.

```
double parallel () {
 Error = 0.0;
 for(t = 0; t < T; t += dT) {
      if(stage==STRAIGHT) {                 // < −− Drive in reverse
         if(t > ??) stage= INTURN; }        // Parameter t1
      if(stage==INTURN) {                   // <−− Turn the wheels towards the curb
         car.ang = car.ang − ??;            // Parameter A1
         if(t > ??) stage= OUTTURN; }       // Parameter t2
      if(stage==OUTTURN) {                  // <−− Turn the wheels away from the curb
        car.ang = car.ang + ??;        }    // Parameter A2
      simulate_car(car); }
 Error = check_destination(car);            // <−− Compute the error as the difference
                                            //      between the desired and actual
 return Error;                              //      positions of the car
}
```

Fig. 1. Sketch of a parallel parking controller

2 Using Euler

2.1 Programming Euler: Parallel parking

Euler can be downloaded from `http://www.cs.rice.edu/~swarat/Euler`. Now we show to use the system to solve a program synthesis problem that reduces to numerical optimization.

The goal here is to design a controller for parallel-parking a car. The programmer knows what such a controller should do at a high level: it should start by driving the car in reverse, then at time t_1, it should start turning the wheels towards the curb (let us say at an angular velocity A_1) and keep moving in reverse. At a subsequent time t_2, it should start turning the wheels away from the curb (at velocity A_2) until the car reaches the intended position. However, the programmer does not know the optimal values of t_1, t_2, A_1, and A_2, and the system must synthesize these values. More precisely, let us define an objective function that determines the quality of a parallel parking attempt in terms of the error between the final position of the car and its intended position. The system's goal is to minimize this function.

To solve this problem using Euler, we write a parameterized program—a *sketch*—that reflects the programmer's partial knowledge of parallel parking.[3] This core of this program is the function `parallel` shown in Fig. 1. For space reasons, we omit the rest of the program—however, the complete sketch is part of the Euler distribution.

It is easy to see that `parallel` encodes our partial knowledge of parallel parking. The terms `??` in the code are "holes" that correspond, in order, to the parameters t_1, A_1, t_2, and A_2, and Euler will find appropriate values for them.

[3] The input language of Euler is essentially the same as in the Sketch programming synthesis system [4].

Note that we can view `parallel` as a function `parallel`(t_1, A_1, t_2, A_2). This is the function that EULER is to minimize.

Occasionally, a programmer may have some insights about the range of optimal values for a program parameter. These may be communicated using holes of the form `??(p,q)`, where `(p,q)` is a real interval. If such an annotation is offered, EULER will begin its search for the parameter from the region `(p, q)`. Note, however, that EULER performs unconstrained optimization, and it is not guaranteed that the value finally found for the parameter will lie in this region.

2.2 Running EULER

Let us we have built EULER and set up the appropriate library paths (specifically, EULER requires GSL—the GNU Scientific Library), and that our sketch of the parallel parking controller has been saved in a file `parallelPark.sk`. To perform our optimization task, we compile the file using EULER by issuing the command "`$ euler parallelPark.sk`". This produces an executable `parallelPark.out`. Upon running this executable ("`$ ./parallelPark.out`"), we obtain an output of the following form:

```
Parameter #1: -37.0916;          Parameter #2: 19.4048;
Parameter #3: -41.1728;          Parameter #4: 1.11344;
Optimal function value: 10.6003
```

That is, the optimal value for t_1 is -37.0916, that for A_1 is 19.4048, and so on.

As mentioned earlier, EULER uses a combination of smooth interpretation and a blackbox optimization method. In the present version of EULER, the latter is fixed to be the Nelder-Mead simplex method [3], a derivative-free nonlinear optimization technique. However, we also allow the programmer to run, without support from smoothing, every optimization method implemented by GSL. For example, to run the Nelder-Mead method without smoothing, the user issues the command "`$ ./parallelPark.out -method neldermead`". For other command-line flags supported by the tool, run "`$ ./parallelPark.out -help`".

3 System internals

Now we briefly examine the internals of EULER. First, we recall the core ideas of smooth interpretation [1, 2].

The central idea of smooth interpretation is to transform a program via *Gaussian smoothing*, a signal processing technique for attenuating noise and discontinuities in real-world signals. A blackbox optimization method is now applied to this "smoothed" program. Consider a program whose denotational semantics is a function $P : \mathbb{R}^k \to \mathbb{R}$: on input $\mathbf{x} \in \mathbb{R}^k$, the program terminates and produces the output $P(\mathbf{x})$. Also, let us consider Gaussian functions $\mathcal{N}_{\mathbf{x},\beta}$ with mean $\mathbf{x} \in \mathbb{R}^k$ and a fixed standard deviation $\beta > 0$. Smooth interpretation aims to compute a function $\overline{P}$ equaling the *convolution* of P_β and $\mathcal{N}_{\mathbf{x},\beta}$: $\overline{P}_\beta(\mathbf{x}) = \int_{\mathbf{y}\in\mathbb{R}^k} P(\mathbf{y})\ \mathcal{N}_{\mathbf{x},\beta}(\mathbf{y})\ d\mathbf{y}$.

For example, consider the program "`z` := 0; `if` ($\mathtt{x}_1 > 0 \wedge \mathtt{x}_2 > 0$) `then` `z` := `z` − 2" where `z` is the output and $\mathtt{x}_1$ and $\mathtt{x}_2$ are inputs. The (discontinuous) semantic function of the program is graphed in Fig. 2-(a). Applying Gaussian convolution to this function gives us a smooth function as in Fig. 2-(b).

As computing the exact convolution of an arbitrary program is undecidable, any algorithm for program smoothing must introduce additional approximations. In prior work, we gave such an algorithm [1]—this is what Euler implements.

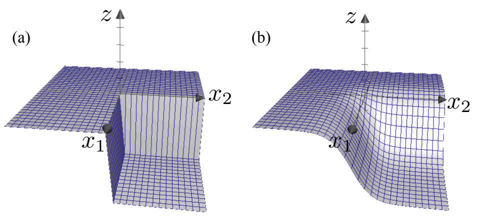

Fig. 2. (a) A discontinuous program (b) Gaussian smoothing

We skip the details of this algorithm here. However, it is worth noting that function $\overline{P}_\beta$ is parameterized by the standard deviation β of $\mathcal{N}_{\mathbf{x},\beta}$. Intuitively, β controls the extent of smoothing: higher values of β lead to greater smoothing and easier numerical search, and lower values imply closer correspondence between P and $\overline{P}_\beta$, and therefore, greater accuracy of results. Finding a "good" value of β thus involves a tradeoff. Euler negotiates this tradeoff by starting with a moderately high value of β, optimizing the resultant smooth function, then iteratively reducing β and refining the search results.

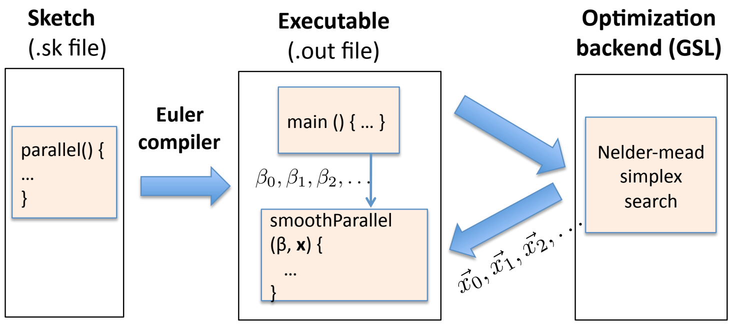
Sketch
(.sk file)
parallel() {
...
}
Euler compiler
Executable
(.out file)
main () { ... }
$\beta_0, \beta_1, \beta_2, \ldots$
smoothParallel
(β, x) {
...
}
Optimization backend (GSL)
Nelder-mead simplex search
$\vec{x}_0, \vec{x}_1, \vec{x}_2, \ldots$

Fig. 3. Control flow in Euler

The high-level control flow in the tool is as in Fig. 3. Given an input file (say `parallelPark.sk`), the Euler compiler produces an executable called `parallelPark.out`. In the latter, the function `parallel`($\mathbf{x}$) has been replaced by a smooth function `smoothParallel`($\beta, \mathbf{x}$). When we execute `parallelPark.out`, β is set to an initial value β_0, and the optimization backend (Gsl) is invoked on the function `smoothParallel`($\beta = \beta_0, \mathbf{x}$). The optimization method repeatedly queries `smoothParallel`, starting with a random initial input $\mathbf{x_0}$ and perturbing it iteratively in subsequent queries, and finally returns a minimum to the top-level loop. At this point, β is set to a new value and the same process is repeated. We continue the outer loop until it converges or there is a timeout.

4 Results

Now we present the results obtained by running Euler on our parallel parking example. These results are compared with the results of running the Nelder-Mead method (Nm), without smoothing, on the problem.

As any local optimization algorithm starts its search from a random point $\mathbf{x_0}$, the minima $\mathbf{x}_{\min}$ computed by Euler and Nm are random as well. However, the difference between the two approaches becomes apparent when we consider the *distribution* of $\texttt{parallel}(\mathbf{x}_{\min})$ in the two methods. In Fig. 4, we show the results of Euler and Nm on 100 runs from initial points generated by a uniform random sampling of the region $-10 < t_1, A_1, t_2, A_2 < 10$. The values of $\texttt{parallel}(\mathbf{x}_{\min})$ on all these runs have been clustered into several intervals, and the number of runs leading to outputs in each interval plotted as a histogram.

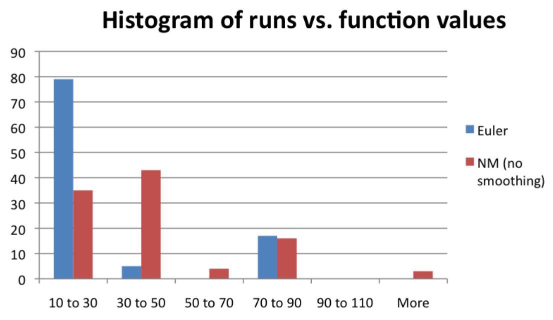

Fig. 4. Percentages of runs that lead to specific ranges of $\texttt{parallel}(\mathbf{x}_{\min})$ (lower ranges are better).

As we see, a much larger percentage of runs in Euler lead to lower (i.e., better) values of $\texttt{parallel}(\mathbf{x}_{\min})$. The difference appears starker when we consider the number of runs that led to the best-case behavior for the two methods. The best output value computed by both methods was 10.6003; however, 68 of the 100 runs of Euler resulted in this value, whereas Nm had 26 runs within the range 10.0-15.0.

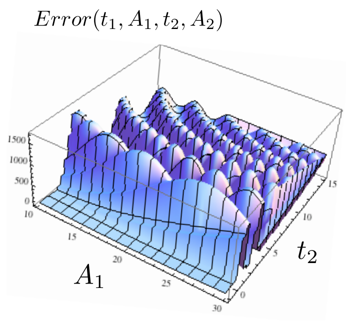

Fig. 5. Landscape of numerical search for parallel parking algorithm

Let us now see what these different values of $\texttt{parallel}(\mathbf{x}_{\min})$ actually *mean*. We show in Fig. 6-(a) the trajectory of a car on an input $\mathbf{x}$ for which the value $\texttt{parallel}(\mathbf{x})$ equals 40.0 (the most frequent output value, rounded to the first decimal, identified by Nm). The initial and final positions of the car are marked; the two black rectangles represent other cars; the arrows indicate the directions that the car faces at different points in its trajectory. Clearly, this parking job would earn any driving student a failing grade.

On the other hand, Fig. 6-(b) shows the trajectory of a car parked using on an input for which the output is 10.6, the most frequent output value found by Euler. Clearly, this is an excellent parking job.

The reason why NM fails becomes apparent when we examine the search space that it must navigate here. In Fig. 5, we plot the function `parallel`($\mathbf{x}$) for different values of t_2 and A_1 (t_1 and A_2 are fixed at optimal values). Note that the search space is rife with numerous discontinuities, plateaus, and local minima. In such extremely irregular search spaces, numerical methods are known not to work—smoothing works by making the space more regular. Unsurprisingly, similar phenomena were observed when we compared EULER with other optimization techniques implemented in GSL.

These observations are not specific to parallel parking—similar effects are seen on other parameter synthesis benchmarks [1] that involve controllers with discontinuous switching (several of these benchmarks are part of the EULER distribution). More generally, the reason why `parallel` is so ill-behaved is fundamental: even simple programs may contain discontinuous if-then-else statements, which can be piled on top of each other through composition and loops, causing exponentially many discontinuous regions. Smooth interpretation is only the first attempt from the software community to overcome these challenges; more approaches to the problem will surely emerge. Meanwhile, readers are welcome to try out EULER and advise us on how to improve it.

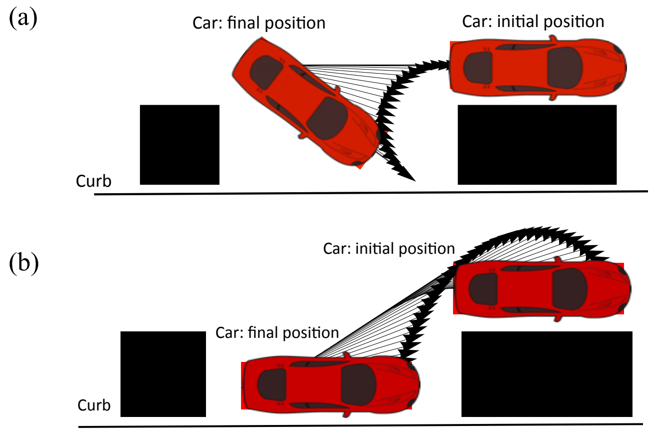

Fig. 6. (a) Parallel parking as done by NM; (b) Parallel parking as done by EULER

References

1. S. Chaudhuri and A. Solar-Lezama. Smooth interpretation. In *PLDI*, pages 279–291, 2010.
2. S. Chaudhuri and A. Solar-Lezama. Smoothing a program soundly and robustly. In *CAV*, pages 277–292, 2011.
3. J.A. Nelder and R. Mead. A simplex method for function minimization. *The computer journal*, 7(4):308, 1965.
4. A. Solar-Lezama, R. Rabbah, R. Bodík, and K. Ebcioğlu. Programming by sketching for bit-streaming programs. In *PLDI*, pages 281–294. ACM, 2005.