2009

Shadowing Chaos Via Optimization

Henrik Haakonsen
Macalester College

Shadowing Chaos Via Optimization

Henrik Bernt Håkonsen

Prof. Stan Wagon, First reader
Prof. Daniel Kaplan, Second reader
Prof. Steve McKelvey, Third reader

April, 2009

Department of Mathematics

■ Abstract

A prominent idea in the theory of chaos is that of shadowing, which says that, in many cases, the numerical results one sees after accuracy is lost are not total nonsense, but are in fact very close to the exact trajectory for an initial value that is near the one used.Using high-precision computation, I have researched the use of optimization as a way of finding exact shadows for several chaotic systems, such as the quadratic map $r\,x(1-x)$ and a billiard problem from the SIAM 100-Digit Challenge.

■ Contents

Abstract	2
Acknowledgements	4
1 Introduction	5
2 Quadratic Map	6
3 Chow and Palmer Applied	10
4 Billiard Trajectory	21
5 Billiard Trajectory Revisited	31
6 Conclusion	33
Bibliography	34

■ Acknowledgements

My advisor Professor Stan Wagon has been an invaluable source of both knowledge and support throughout this project, and I owe him a great thanks. It has been a pleasure to work and contemplate ideas with him, and his enthusiasm for the project has been a great inspiration for me.

Also, I wish to acknowledge Rob Knapp who told us several things about the FindMinimum function in *Mathematica* that helped us understand how to use it.

■ Introduction

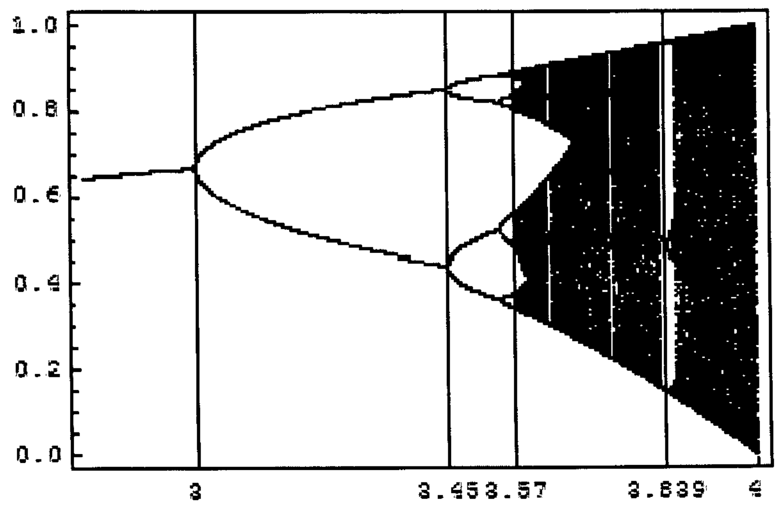
A prominent idea in the theory of chaos is that of shadowing, which says that, in many cases, the numerical results one sees after accuracy is lost are not total nonsense, but are in fact very close to the exact trajectory for an initial value that is near the one used. Previous literature in the field has focused on proving the existence of such a shadow, and on estimating how far the numerically computed trajectory is from an exact trajectory; see Chow and Palmer(1989)[1].

The research in this paper is focused on computing exact shadows, utilizing the high precision computation capacity that is now available. We have successfully computed shadows for the quadratic map, $f(x) = r\,x(1-x)$, for $r = 3.8$ for which there is strong evidence that the map exhibits sensitive dependence on initial conditions, which is to say that small variations in the initial condition may cause large variations in the long-term. Most of our experimental results come from using 0.1 as the starting value.

The idea is that one can use high precision to compute the accurate trajectory, and then find a shadow seed by minimizing the sum of the squares of the residual vector, where the residual is the difference between an exact trajectory and the noisy trajectory at any given iterate. The exact trajectory will shadow the noisy numerically computed trajectory, differing by at most some value. Chow and Palmer(1989) analyzed the case of $r = 3.8$, using a starting value of 0.6, and concluded that the noisy trajectory, calculated using single precision, differs from some shadow trajectory by at most one in the fifth digit. They did not however actually find any shadow seed, and by doing so we aim in this study to find a true story, not just getting estimates. Experiments have shown that even a simple Newton method is adequate to minimize the sum of the squares of the residuals, although using the Levenberg-Marquardt method has proven to be more efficient.

This study has also proved applicable in the context of billiard trajectories, where we have successfully shadowed a trajectory starting at {0.5, 0.1} out to more than 75 iterations.

The image[2] below illustrates the bifurcation, i.e. period doubling, behavior of the quadratic map as r varies. The vertical axis is long-term possible values of x. Most values of r beyond 3.57 exhibit chaotic behavior.



Quadratic Map

```
fQuad[r_][x_] := r x (1 - x);
f = fQuad[38 / 10];
```

Above we have defined the function for the general quadratic map, with r as an parameter, and we have defined a function f with r set to 3.8 since that is what we will use in our experiments.

```
Precision /@ {f[0.1], f[1 / 10], f[SetPrecision[1 / 10, 16]]}

{MachinePrecision, ∞, 15.9542}
```

Note that applying the function to a MachinePrecision number yields another MachinePrecision number, whereas applying the function to a rational number will keep infinite precision. Note however that there is long-term loss of precision when applying the function to a real number, and that it is the roundoff error caused by this that provides the basis for the shadowing problem. Computations using MachinePrecision are done with roughly 16 digits of precision, but it does in no way keep track of lost precision.

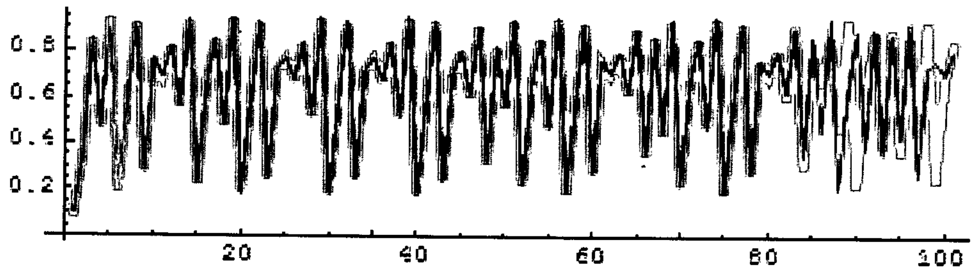
```
Traj[f_][x_Real, n_Integer] := NestList[f, x, n];
fTraj = Traj[fQuad[38 / 10]];
```

Above we have defined the functions for the trajectory of the quadratic map, which is simply formed by iterating the function of the quadratic map.

```
Precision /@ fTraj[SetPrecision[1 / 10, 16], 10]

{16., 15.9542, 15.7725, 14.9334, 14.6571,
 13.3835, 13.2911, 12.9054, 11.8074, 11.6653, 11.0374}
```

Note the gradual loss of precision. Below is a plot of the noisy trajectory (computed using MachinePrecision) and a HighPrecision trajectory out to 100 iterations. Note how they diverge after 80 or so iterations.

```
MPTraj = Traj[fQuad[38 / 10]][0.1, 1000];
HPTraj = Traj[fQuad[38 / 10]][SetPrecision[0.1, 1000], 1000];
ListLinePlot[{Take[MPTraj, 101], Take[HPTraj, 101]},
 PlotStyle → { {Thickness[0.012], GrayLevel[0.5]}, {Thickness[0.001], Black}},
 AspectRatio → 0.25]
```



We know that the derivative of f with respect to x is $38/10 - 76x/10$, and by the Chain Rule it follows that the sensitivity of any given iterate x_n with respect to the starting value x_0 will be the product of all derivatives of the trajectory up to that given iterate. That is to say that $\frac{dx_n}{dx_0} = \frac{dx_n}{dx_{n-1}} \frac{dx_{n-1}}{dx_{n-2}} \ldots \frac{dx_2}{dx_1} \frac{dx_1}{dx_0}$.

Below we define functions that combine the derivatives at each iteration of the quadratic map to form a sensitivity vector which we will use to guide our search for the shadow seed. The sensitivity vector will thus be a vector of $\frac{dx_{i+1}}{dx_i}$ (i = 0,1,...,n-1). Having this makes it possible for the FindMinimum method to utilize the quadratic model approach. The concept of the quadratic model approach, assuming a roughly parabolic shape, is that by knowing (a, f(a), f ' (a)) and (b, f(b)), where a and b are two points, we can uniquely determine a parabola $Ax^2 + Bx + C$and thus also a minimum c = -B/(2A). That will be our first approximation of the minimum, and we proceed to repeat the process with (c, f(c), f ' (c)) and whichever of a and b that was closest to c.

```
fQuadDer[r_][x_] := r (1 - 2 x);
DerivativeVector[r_][x_, n_] := fQuadDer[r] /@ Traj[fQuad[r]][x, n];
JacobianVector[r_?NumericQ][x_?NumericQ, n_?NumericQ] :=
  Most[FoldList[Times, {1}, DerivativeVector[r][x, n]]];
fJacobianVector = JacobianVector[38 / 10];
```

The functions above simply define the Jacobian of x_n with respect to x_0, which is an application of the Chain Rule on the derivative vector to give us a vector of sensitivities to changes in the starting value for each iterate, i.e. a vector of $\frac{dx_i}{dx_0}$ (i = 0,1,...,n).The functions below are what we will use to compute the shadow seed, i.e. the HighPrecision starting value that shadows the noisy trajectory. We find the shadow seed by using the LevenbergMarquardt method together with the Jacobian to find the minimum of the sum of squares residual. The Jacobian essentially gives the information about how the vector of residuals will change for miniscule changes in the starting value, and it is this that allows for effective shadowing.

When attempting to shadow too many steps at a time however, the method above fails. A way to solve this is by shadowing in steps, that is find a shadowseed that shadows out to a certain number of steps, and then use this result as the start of another shadow search out to a higher number of steps. Doing this ensures a better form for the FindMinimum search to focus around.

```
ShadowOneStep[r_][noisy_List, n_Integer, opts___] :=
  (start = StartValue /. {opts} /. Options[ShadowOneStep];
   wp = Max[35, n];
   start = start /. UseNoisyEstimate → SetPrecision[noisy[[1]], wp];
   residual[x0_?NumericQ] := Block[{$MinPrecision = wp, $MaxPrecision = wp},
     Traj[fQuad[r]][x0, n] - Take[SetPrecision[noisy, wp], n + 1]];
   ans = FindMinimum[0, {x0, SetPrecision[start, wp]},
     Evaluate[Sequence @@ FilterRules[{opts}, Options[FindMinimum]]],
     Method → {"LevenbergMarquardt", "Residual" → residual[x0],
       "Jacobian" → JacobianVector[r][N[x0, wp], n]}, WorkingPrecision → wp,
     PrecisionGoal → 20, AccuracyGoal → 20]; x0 /. ans[[2]]);
```

```
ShadowInSteps[r_][noisy_List, n_Integer, {startstep_, stepsize_}, opts___] := (
  Do[sol = ShadowOneStep[r][SetPrecision[noisy, n], nn, opts,
     StartValue → If[nn == startstep, UseNoisyEstimate, SetPrecision[sol, nn]]],
      {nn, Append[Most[Range[startstep, n, stepsize]], n]}];
  sol);
```

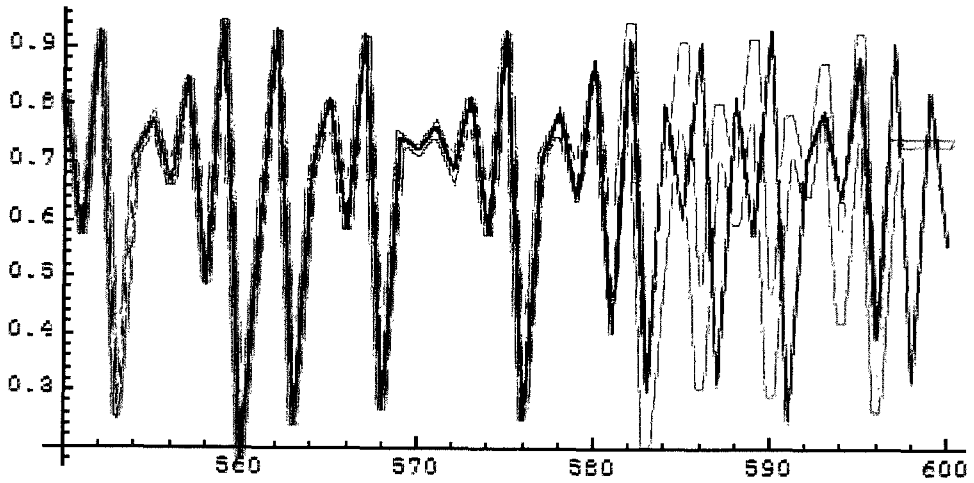
Experiments have shown that shadowing in steps of 50 is effective, and by doing so one can find a shadow for a very large number of iterates. Here is a shadow seed found by searching in steps of 50 out to 500 iterates.

```
ShadowSeed = ShadowInSteps[38 / 10][MPTraj, 500, {50, 50}]
```

```
0.1000000000000000210432393045703215538761144492231330812621760710108989796553⋱
 5985378463483558600223120544712877947582904317713652237709309740884340876670 4⋱
 8051294451620510486271377317851792889680298874903897826246894809836753439312 4⋱
 7085788306450373077969935130909131601583588677119288623539565476113387702157 1⋱
 2687259790599894732011975895470404672319261145756192037523256162890988084801 2⋱
 9479626583050567505795189677299478427950183257377159977860834849102795477288 5⋱
 79791416932172798348740074143963267183 55
```

As one can see below, the ShadowSeed found shadows the noisy trajectory on a visual scale to about 580 iterates, a very satisfying result.

```
HPTraj = Traj[fQuad[38 / 10]][ShadowSeed, 600];
ListLinePlot[{Take[MPTraj, {550, 600}], Take[HPTraj, {550, 600}]},
 DataRange → {550, 600},
 PlotStyle → { {Thickness[0.012], GrayLevel[0.5]}, {Thickness[0.001], Black}},
 AspectRatio → 0.5]
```



To find out how closely the high-precision trajectory shadows the noisy trajectory we define a function of the shadowing error which gives us the maximum difference between the two trajectories out to 500 iterates.

```
ShadowError[r_][{xNoisy_, shadSeed_}, NN_] :=
  Max[Abs[Traj[fQuad[r]][N[shadSeed, NN], NN] -
     SetPrecision[Traj[fQuad[r]][ xNoisy, NN], NN]]];
```

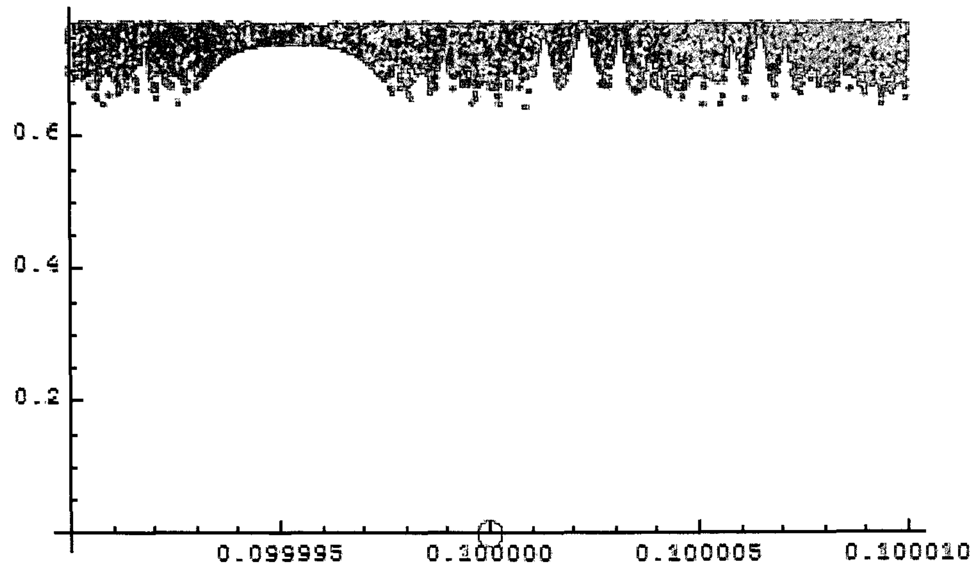
```
ShadowError[38 / 10][{0.1, ShadowSeed}, 500]
```

```
2.18225845356314605656302965625142833458042449855512245703506822944815496294341
  89300196382597641514231277958919034244055372156299717986041618558920185122281
  33120957566888332552486017928556191049875304347346714449690559393032730181290
  0472034310979334542384238427889349944246396 × 10^-14
```

So, the high-precision trajectory shadows the noisy trajectory out to 500 iterations within $2.1823 * 10^{-13}$. It is interesting to note that for a high-precision trajectory to be within some small number of a noisy trajectory, the starting value must also naturally be within that number of the noisy starting value. This limits the search for a shadow seed to the immediate vicinity of the noisy starting value.

Now, one might wonder how "special" such a shadow seed is. Below is a graph illustrating the shadowing error for the ShadowSeed (Red) and 10,000 random points within 10^{-5} of the ShadowSeed. As one can see, not a single one of these starting values except the ShadowSeed is within 0.6 of the noisy trajectory, implying that the ShadowSeed can be considered quite unique.

```
data =
  Append[Table[{0.1 + (RA = Random[Real, {-10^-5, 10^-5}]),
     ShadowError[38 / 10][{0.1, 0.1 + RA}, 100]}, {i, 1, 10000}],
   {ShadowSeed, ShadowError[38 / 10][{0.1, ShadowSeed}, 100]}];
Show[ListPlot[data, {AxesOrigin → {0.1 - 10^-5, 0}}],
  Graphics[{Red, PointSize[0.03], Point[data[[-1]]]}]]
```



■ Chow and Palmer Applied

Chow and Palmer(1989)[1] introduces the concepts of σ and τ in their estimation of a lower- and upper-bound of the sup shadowing-error, i.e. the maximum of the absolute value of the difference between the noisy trajectory and an exact shadow trajectory at any iterate. So, it suffices to say that these are used by Chow and Palmer to estimate how far the noisy trajectory is from an exact, i.e. high - precision trajectory.

They consider the case of one-dimensional maps, determine for any number of iterates if their theorem applies, and if it does we can proceed by calculating the shadowing error.

Following the notation set forth by Chow and Palmer we consider the case of the quadratic map.

A statement of shadowing:
Let $\{y_n\}_{n=0}^{N+1}$ be a noisy trajectory. At each iteration of the noisy trajectory, the roundoff error $| y_{n+1} - f(y_n) |$ is small. We want to find a high-precision trajectory $\{x_n\}_{n=0}^{N}$ such that the difference between the high-precision trajectory and the noisy trajectory $| x_n - y_n |$ is small for each iterate.

Chow and Palmer provides the following definitions of σ and τ:

$$\sigma = \underset{n=0}{\overset{N}{\mathbf{Max}}} \sum_{m=n}^{N} \left| \mathbf{Df}(y_n)^{-1}\ \mathbf{Df}(y_{n+1})^{-1} \ldots \mathbf{Df}(y_m)^{-1} \right|$$

A rough but yet good estimate put in simple terms is that we can say that σ is simply the maximum of the reciprocal of the derivative.

$$\tau = \underset{n=0}{\overset{N}{\mathbf{Max}}} \left| \sum_{m=n}^{N} \mathbf{Df}(y_n)^{-1}\ \mathbf{Df}(y_{n+1})^{-1} \ldots \mathbf{Df}(y_m)^{-1} \left[y_{m+1} - f(y_m)\right] \right|$$

τ is pretty much the same thing, except that each reciprocal of the derivative term is multiplied by $| y_{i+1} - f(y_i) |$, where i is the current term.

Their theorem is then :

Defining M to be the maximum of the reciprocal derivative.

$M = Max\left\{ \left| D^2 f(x) \right| : 0 \le x \le 1 \right\}$

Let $\{y_n\}_{n=0}^{N+1}$ *be a noisy trajectory of f such that*

$2 M \sigma \tau \le 1$

Now there exists an exact high – precision trajectory $\{x_n\}_{n=0}^{N}$ *with*

$$\left(1 + 1/2\left(1 + \sqrt{1 - 2M\sigma\tau}\right)\right)^{-1} \tau \le \underset{n=0}{\overset{N}{Max}} \left| x_n - y_n \right| \le 2\left(1 + \sqrt{1 - 2M\sigma\tau}\right)^{-1} \tau$$

The τ and σ functions below were programmed in Mathematica by Professor Stan ***Wagon***[2] following the definitions set forth by Chow and Palmer.

```
Clear[σ, τ];
σ[r_][x_, NN_] := σ[r][x, NN] = (derlist = DerivativeVector[r][x, NN];
```
$$\text{Max}\left[\text{Table}\left[\sum_{m=n}^{NN} \text{Abs}\left[\frac{1}{\prod_{i=n}^{m} \text{derlist}[[i+1]]}\right], \{n, 0, NN\}\right]\right]);$$
```
τ[r_][x_, NN_] := τ[r][x, NN] = (ypts = SetPrecision[Traj[fQuad[r], x, NN + 1], NN];
  derlist = SetPrecision[DerivativeVector[r][x, NN], NN];
```
$$\text{Max}\left[\text{Table}\left[\text{Abs}\left[\sum_{m=n}^{NN} \frac{\text{ypts}[[m+2]] - \text{fQuad}[r][\text{ypts}[[m+1]]]}{\prod_{i=n}^{m} \text{derlist}[[i+1]]}\right], \{n, 0, NN\}\right]\right]);$$

In fact, τ yields such decent results even when only estimating from 1 iterate that it alone can help improve the FindMinimum method used in the previous section by providing a good place to start the search, i.e. avoiding some local minimums. This is especially helpful when trying to shadow larger intervals, as the structure of the error curve become more complicated. Below is a program which allows for a fast estimation of τ, we call this estimate τ-first. Also, below are the Shadowing functions used in the previous section, but now set to use this estimated τ-value as the default place to start the shadow search. What τ-first does is to provide an estimate of the sup error, meaning maximum residual, where residual is the absolute difference between the high-precision trajectory and the noisy trajectory at any iterate, of the first point in the trajectory between the shadow trajectory and the noisy trajectory. Adding the estimate of the sup error of the first point to the noisy starting value gives a good place to start the FindMinimum search.

```
τFirstFast[r_][x_, NN_] :=
  τFirstFast[r][x, NN] = (ypts = SetPrecision[Traj[fQuad[r]][x, NN + 1], NN + 100];
   derlistProds =
    Rest[FoldList[Times, 1, SetPrecision[DerivativeVector[r][x, NN],
       NN + 100]]];
   Total[(Rest[ypts] - (fQuad[r] @ Most[ypts])) / derlistProds]);

Options[ShadowOneStep] = {StartValue → UseTauEstimate};

ShadowOneStep[r_][noisy_List, n_Integer, opts___] :=
  (start = StartValue /. {opts} /. Options[ShadowOneStep];
   wp = Max[35, n];
   start =
    start /. UseTauEstimate → SetPrecision[noisy[[1]], wp] +
       τFirstFast[r][noisy[[1]], n];
   residual[x0_?NumericQ] := Block[{$MinPrecision = wp, $MaxPrecision = wp},
     Traj[fQuad[r]][x0, n] - Take[SetPrecision[noisy, wp], n + 1]];
   ans = FindMinimum[0, {x0, SetPrecision[start, wp]},
     Evaluate[Sequence @@ FilterRules[{opts}, Options[FindMinimum]]],
     Method → {"LevenbergMarquardt", "Residual" → residual[x0],
       "Jacobian" → JacobianVector[r][N[x0, wp], n]}, WorkingPrecision → wp,
     PrecisionGoal → 20, AccuracyGoal → 20]; x0 /. ans⟦2⟧);

ShadowInSteps[r_][noisy_List, n_Integer, {startstep_, stepsize_}, opts___] := (
   Do[sol = ShadowOneStep[r][SetPrecision[noisy, n], nn, opts,
      StartValue → If[nn == startstep, UseTauEstimate, SetPrecision[sol, nn]]],
     {nn, Append[Most[Range[startstep, n, stepsize]], n]}];
   sol);
```

Experiments have shown that trying to shadow too many iterations in one attempt at local-minimum-finding (e.g., using Mathematica's FindMinimum command) using the noisy starting value is difficult because there are many local-minimum points, but iterations of 50 or less work in most cases. This makes somewhat intuitive sense since a MachinePrecision number is composed of roughly 50-bits. Note however that even this will not allow shadowing many iterations in one step alone, so a second function ShadowIn-Steps is required that shadows the trajectory in steps. By using the result of shadowing part of the trajectory to shadow more of the trajectory, we ensure that we are in the roughly the right place because later iterates are more sensitive to changes in the staring value, and thus requires a search in a smaller range.

The paper by Chow and Palmer (1989)[1] focuses on estimating boundaries for the sup error of a shadow, i.e. the maximum error (absolute value of residual), where by residual we mean difference between the high-precision trajectory and the noisy trajectory at a given iterate, whereas above I have been shadowing trying to minimize the sum of squares of the residual vector. One can however easily use the result above as the starting location for a new FindMinimum search to minimize the sup error.

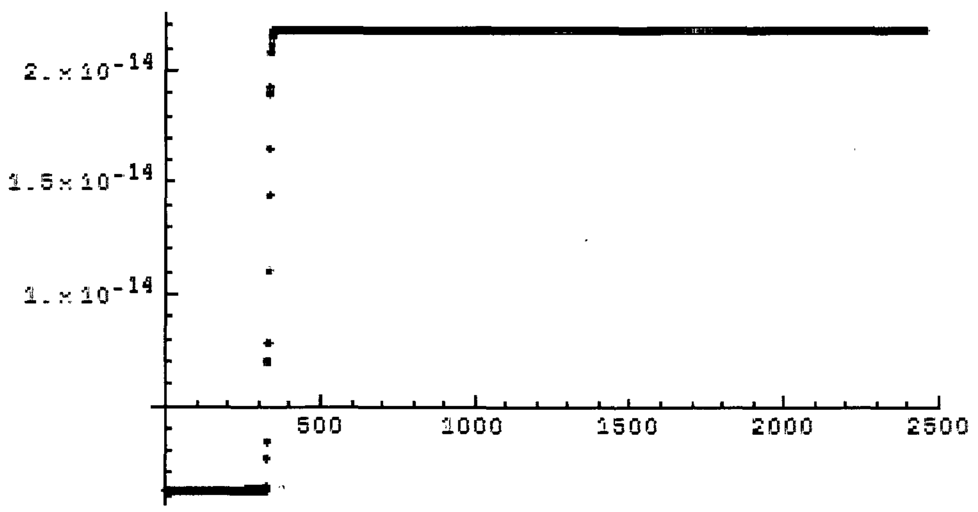
```
ShadowSupNorm[r_][x0_, n_, s_] :=
 (obj[x_?NumericQ] := ShadowError[r][{x0, x}, n]^2;
  dataTemp = Table[{p, Log[10, obj[s + 10^-p]]}, {p, 10, n}];
  mm = Min[Last /@ dataTemp]; pp = Select[dataTemp, Abs[#1[[2]] - mm] < 1/10^6 &, 1];
  pp = pp[[1, 1]];
  xnew /.
   Last[ans = FindMinimum[obj[xnew], {xnew, s - 10^-pp, s + 10^-pp},
      WorkingPrecision → n, PrecisionGoal → 35, AccuracyGoal → 36]]);

ShadowError[r_][{xNoisy_, shadSeed_}, NN_] :=
 Max[Abs[Traj[fQuad[r]][N[shadSeed, NN], NN] -
    SetPrecision[Traj[fQuad[r]][ xNoisy, NN], NN]]];
```

As one can see below, minimizing the sup norm error gives a small but negligible improvement of 8.2362×10^{-58}.

```
SupShadowSeed = ShadowSupNorm[38 / 10][0.1, 500, ShadowSeed];

ShadowError[38 / 10][{0.1, ShadowSeed}, 500]
ShadowError[38 / 10][{0.1, SupShadowSeed}, 500]
N[ShadowError[38 / 10][{0.1, ShadowSeed}, 500] -
  ShadowError[38 / 10][{0.1, SupShadowSeed}, 500]]

1.1981943817430067268373860403641333856823281481727319894249070713721564060211\
 8181930597453820194336983911176848697659406871879784416945862097228339192966l\
 1484477502077743147168821956205510003125372708626616006072871435432016196088б\
 8707838905547374604070245330783839541562759521356838981484596625461297552898\
 64627539927690 × 10^-15

1.1981943817430067268373860403641333856823273245531884675199730773974140927018\
 1736477876896853138842568273773477217002706297862148322166116030912038643622\
 4016832543969255194269695713393495156013309109638323579270529713214133781978\
 9263836743226893937676722674267244800106768643856597070768957401742292042063\
 98311068796200 × 10^-15

8.2362 × 10^-58
```

Now, something of interest is how the sup error increases as the number of iterates increases. Below I have graphed the relationship as the number of iterates goes from 50 to 2500.

```
ListPlot[supErrorList, PlotStyle → {Thickness[0.012], GrayLevel[0]}]
```



As one can see, there are a sequence of fairly consecutive iterates where the sup shadow error jumps. Note that the sup shadowing error curve is strictly increasing since it is the maximum of the shadowing error. Also note that this curve will eventually reach its maximum value since the noisy trajectory must be periodic due to the finite number of MachinePrecision reals. From this we can deduce that shadowing is possible forever.

▪ Code for SupErrorGraph.

```
fShadowInSteps = ShadowInSteps[38 / 10];
fShadowSupNorm = ShadowSupNorm[38 / 10];
fShadowError  =  ShadowError[38 / 10];

Monitor[HP2500NormSolSeed = fShadowInSteps[MPSol, 2500, {50, 50}], nn];

Monitor[
  supErrorList2 =
   Table[N[fShadowError[{0.1, fShadowSupNorm[0.1, nn, N[HP2500NormSolSeed, nn]]},
      nn]], {nn, 300, 450, 1}], nn];
```

▪ Analysis of the Error

First note how τ-first does a great job shadowing out to 100 steps, with a shadow error within 3.02734^{-12}. Also note how it could do better, as is done by ShadowSeed (this is from optimizing out to 500, but it shadows 100 terms within 1000th of what τ does).

```
ShadowError[38 / 10][
 {0.1, τFirstFast[38 / 10][0.1,  100] + SetPrecision[MPTraj[[1]], wp]},
 100]

3.0273383743856052952927515164439×10^-12

ShadowError[38 / 10][{0.1, ShadowSeed}, 100]
```

```
1.2441142664934016544735037677797673131542029678746653217787215819417123011024×
 10^-15
```

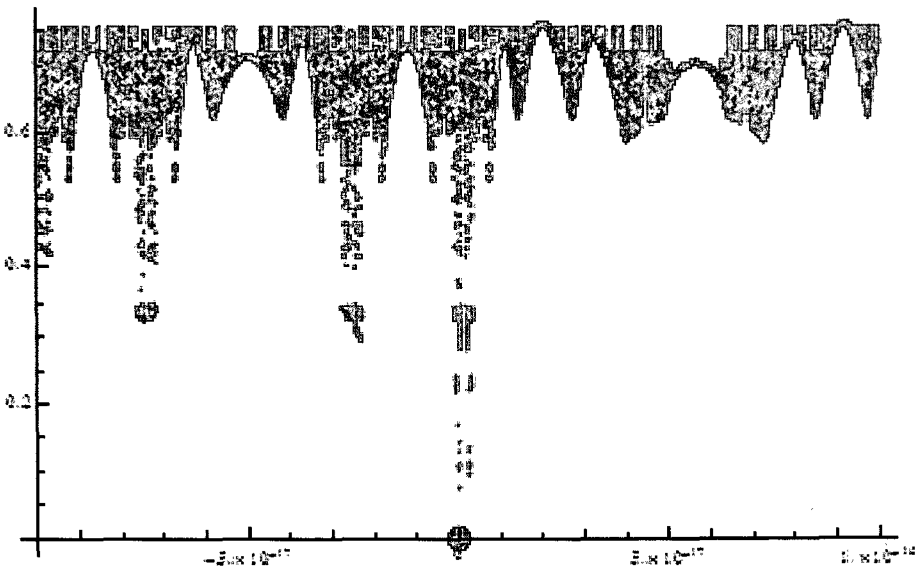
Now note how ShadowSeed is within 10^{-25} of τ-first.

```
ShadowSeed - (τFirstFast[38 / 10][0.1, 50] + SetPrecision[MPTraj[[1]], wp])
```

```
-4.378844953136377922064278155298169485623384421554811330983189133645528108825\
   332258221100514510912387813775170428076757485412 × 10^-26
```

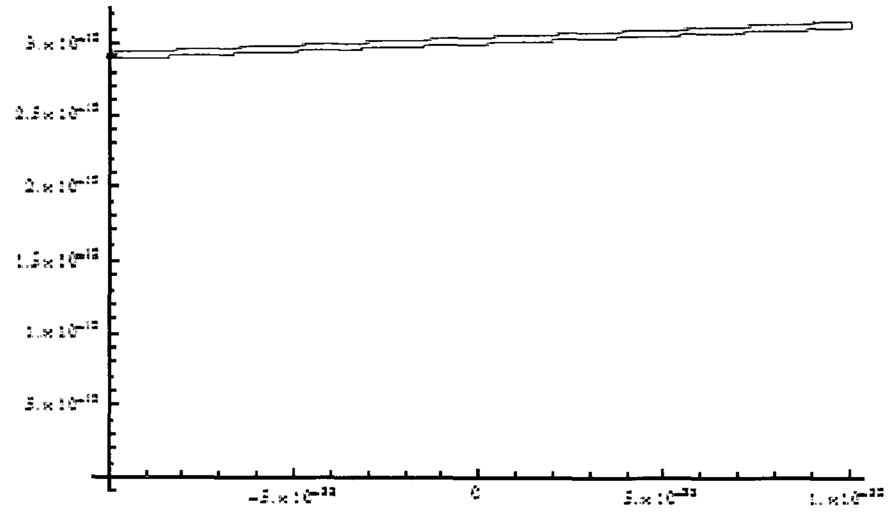
A plot of maximum absolute errors (out to 100 steps), using seeds in within 10^{-16} of the τ - first seed. This shows us that even relatively close to the ShadowSeed the error structure is far from being a nice parabolic shape, and it is this that causes problems when trying to shadow many iterations from a less than optimal starting place for the FindMinimum search.

```
ListPlot[
 Prepend[
  Table[ShadowError[38 / 10][
    {0.1, τFirstFast[38 / 10][0.1, 100] + SetPrecision[MPTraj[[1]], wp] + i}, 100],
   {i, -10^-16, 10^-16, 10^-20}], ShadowError[38 / 10][{0.1, ShadowSeed}, 100]],
 AxesOrigin → {0, 0}]
```

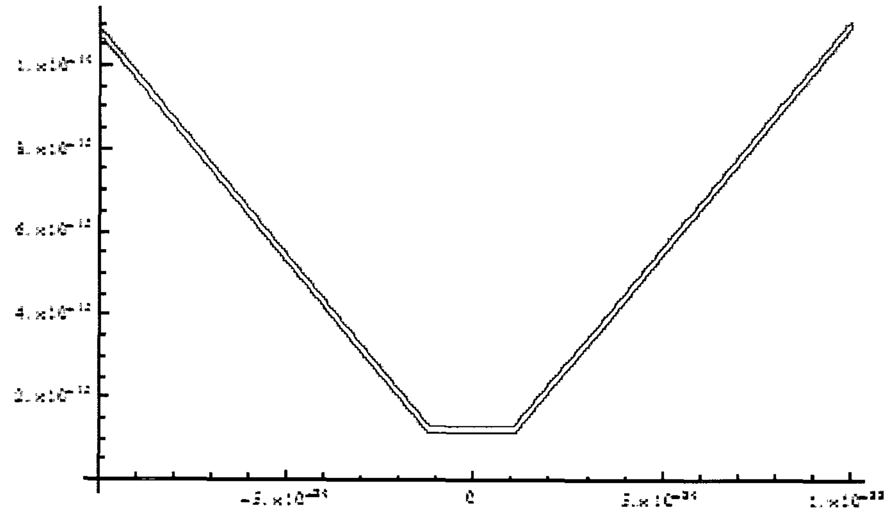


Now looking really closely (within 10^{-32} and 10^{-33}) at the two seeds found by τ-first and FindMinimum using τ-first as a start respectively. It is obvious that the shadowSeed is at a minimum, whereas the τ estimate is not.

```
ListPlot[
 Table[ShadowError[38 / 10][
   {0.1, τFirstFast[38 / 10][0.1, 100] + SetPrecision[MPTraj[[1]], wp] + i}, 100],
  {i, -10^-32, 10^-32, 10^-34}], AxesOrigin → {0, 0}]
```



```
ListPlot[Table[ShadowError[38 / 10][{0.1, ShadowSeed + i}, 100],
  {i, -10^-33, 10^-33, 10^-35}], AxesOrigin → {0, 0}]
```

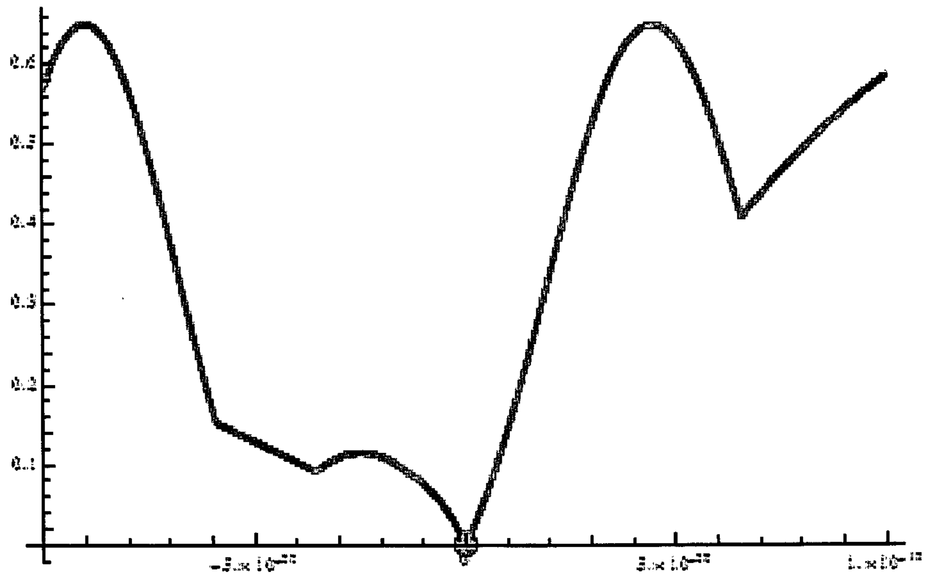


```
ShadowError[38 / 10][{0.1, ShadowSeed - 10^-35}, 100]
ShadowError[38 / 10][{0.1, ShadowSeed}, 100]
ShadowError[38 / 10][{0.1, ShadowSeed + 10^-35}, 100]
```

```
1.244114266493401610906813383623916508058660410719279105336628729987171430884 6 ×
 10^-15

1.2441142664934016544735037677797673131542029678746653217787215819417123011024 ×
 10^-15

1.2441142664934016980401941519356181182497455250265889734285281094489165643629 ×
 10^-15
```

The graph below shows the area within 10^{-19} of the ShadowSeed. We would ideally start our FindMinimum search near the V-shaped region of the graph.

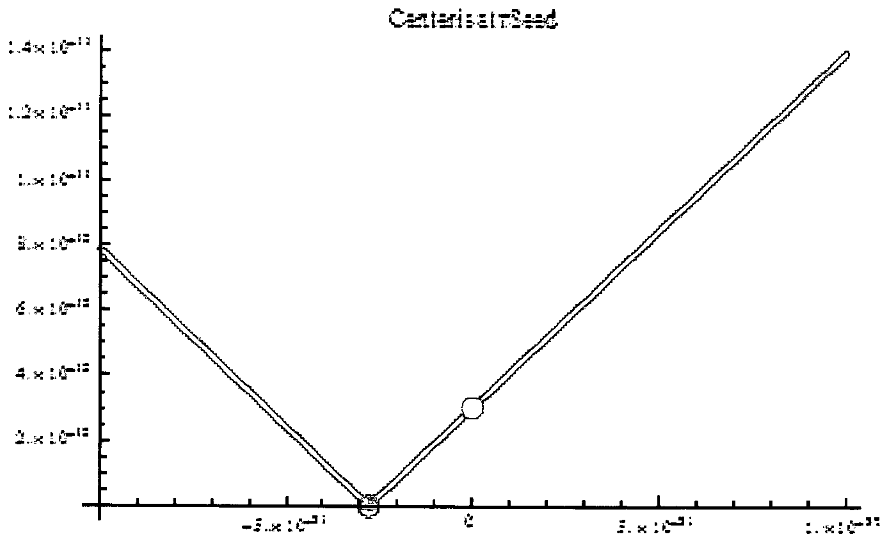
```
ListPlot[Table[ShadowError[38 / 10][{0.1, ShadowSeed + i}, 100],
  {i, -10^-19, 10^-19, 10^-21}], AxesOrigin → {0, 0}]
```



```
tauSeed = τFirstFast[38 / 10][0.1, 100] + SetPrecision[MPTraj[[1]], wp];

ShadowSeed - tauSeed

 -2.778616162040463242052424931189134503431803159211981992227298411969984279778\
    508503001559467680572087646052435172308997397682786318076895471791973559320 2\
    60749823177492950 × 10^-31
```

The red point is the τSeed, whereas the purple point is the ShadowSeed. This graph shows us how close the estimate of τ puts us to the ShadowSeed.

```
tempi = 30;
data = Append[Table[{i, ShadowError[38 / 10][{0.1, tauSeed + i}, 100]},
    {i, -10^-tempi, 10^-tempi, 10^-(tempi + 2)}],
   {ShadowSeed - tauSeed, ShadowError[38 / 10][{0.1, ShadowSeed}, 100]}];
Show[
 ListPlot[data, {AxesOrigin → {data[[1, 1]], 0},
    PlotLabel → "Center is at τSeed"}],
 Graphics[{Red, PointSize[0.03], Point[data[[101]]], Purple, Point[data[[-1]]]}]]
```

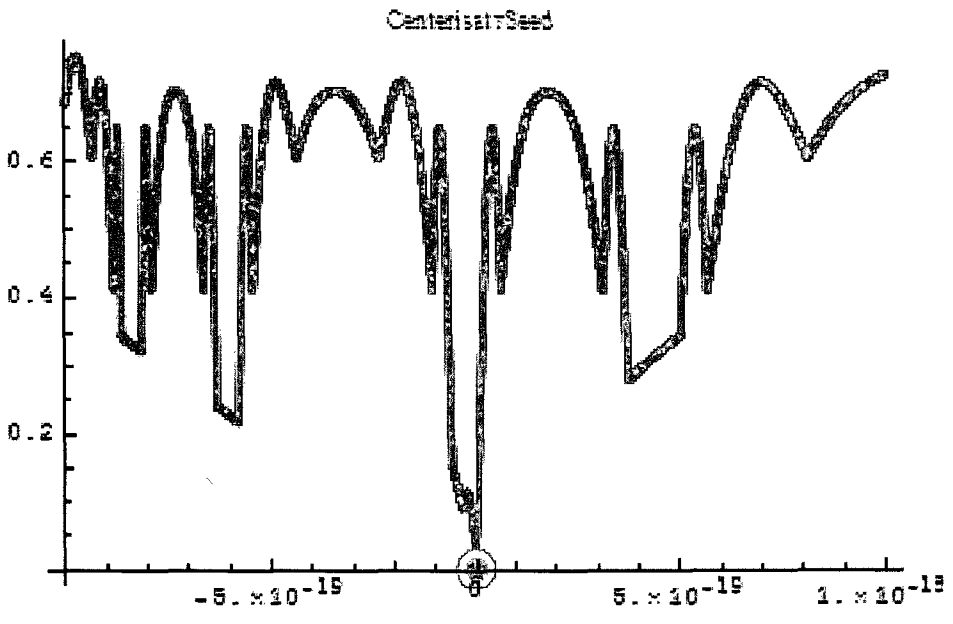


```
data[[-1]]
```

```
{-2.77861616204046324205242493118913450343180315921198199222729841196998427977\
    85085030015594676805720876460524351723089973976827863180768954717919735593 2\
    0260749823177492950 × 10^-31,
  1.24411426649340165447350376777976731315420296787466532177872158194171230 11024
   × 10^-15}
```

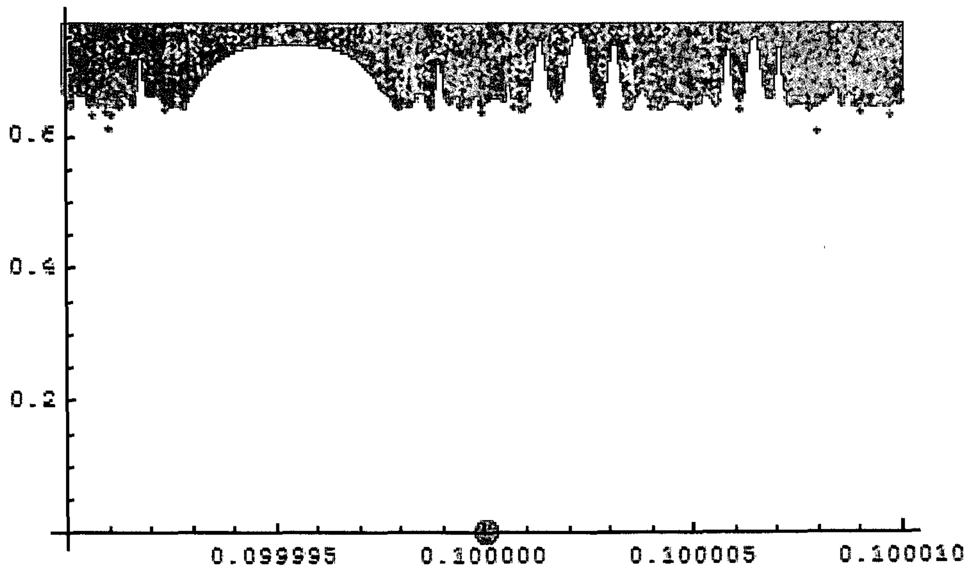
Looking at this from more of a macro-scale really puts it in perspective. The red point is the τ estimate and the purple point is the ShadowSeed.

```
tempi = 18;
data = Append[Table[{i, ShadowError[38 / 10][{0.1, tauSeed + i}, 100]},
    {i, -10^-tempi, 10^-tempi, 10^-(tempi + 4)}],
   {ShadowSeed - tauSeed, ShadowError[38 / 10][{0.1, ShadowSeed}, 100]}];
Show[
 ListPlot[data, {AxesOrigin → {data[[1, 1]], 0},
   PlotLabel → "Center is at τSeed"}],
 Graphics[{Red, PointSize[0.05], Point[data[[10001]]], Purple,
   PointSize[0.03], Point[data[[-1]]]}]]
```



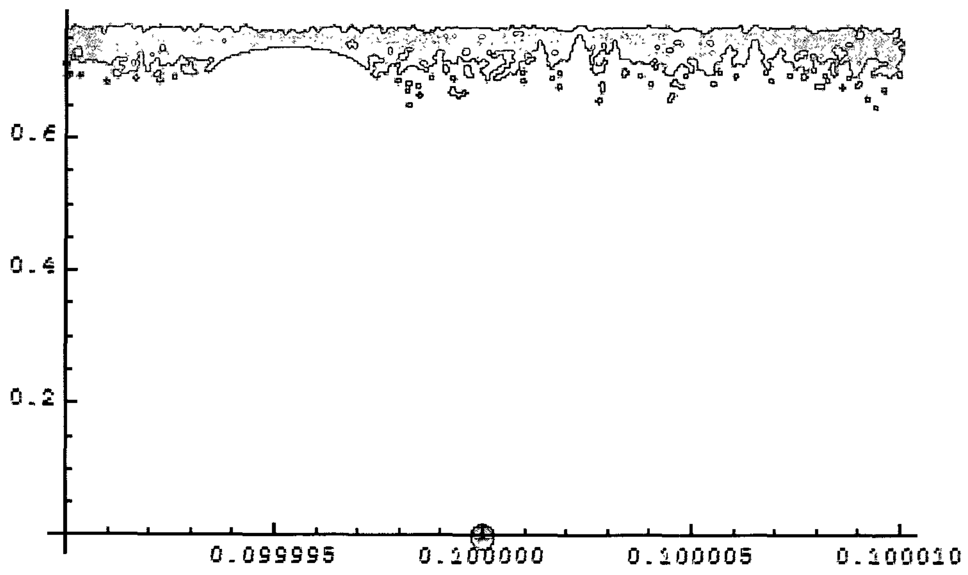
Recalling how "unique" the ShadowSeed found is, it is quite amazing how good of an estimate τ is.

```
data = Append[Table[{0.1 + i, ShadowError[38 / 10][{0.1, 0.1 + i}, 100]},
   {i, -10^-5, 10^-5, 10^-10}],
  {ShadowSeed, ShadowError[38 / 10][{0.1, ShadowSeed}, 100]}];
Show[ListPlot[data, {AxesOrigin → {data[[1, 1]], 0}}],
 Graphics[{Purple, PointSize[0.03], Point[data[[-1]]]}]]
```



Note in the graph above that there are no ShadowErrors larger than 0.7695. This is because the maximum of f(x) is (38/10)*(1/2)*(1/2) = 0.95, implying that the minimum of f(f(x)) is (38/10)*(0.95)*(0.05) = 0.1805, and thus the possible ShadowError, for a starting value in this interval, to be 0.95-0.1805 = 0.7695.

```
data = Append[Table[{0.1 + i, ShadowError[38 / 10][{0.1, 0.1 + i}, 100]},
   {i, -10^-5, 10^-5, 10^-8}],
  {ShadowSeed, ShadowError[38 / 10][{0.1, ShadowSeed}, 100]}];
Show[ListPlot[data, {AxesOrigin → {data[[1, 1]], 0}}],
 Graphics[{Purple, PointSize[0.03], Point[data[[-1]]]}]]
```



■ Billiard Trajectory

The billiard Trajectory problem as stated in the SIAM 100-Digit Challenge[4]:

A photon moving at speed 1 in the x-y plane starts at $t = 0$ at $(x, y) = (1/2, 1/10)$ heading due east. Around every integer lattice point (i, j) in the plane, a circular mirror of radius $1/3$ has been erected. How far from $(0, 0)$ is the photon at $t = 10$?

The initial method here was created by F. H. Simons[3].

Rather than being concerned with where the photon will be at a certain point in time, we can consider this as a discrete problem by simply being concerned with the list of coordinates on the {x, y}-plane of where the photon hits the mirrors. We will refer to a list of such points as a billiard trajectory. We use the same starting value as was stated in the original problem for most of our experiments.

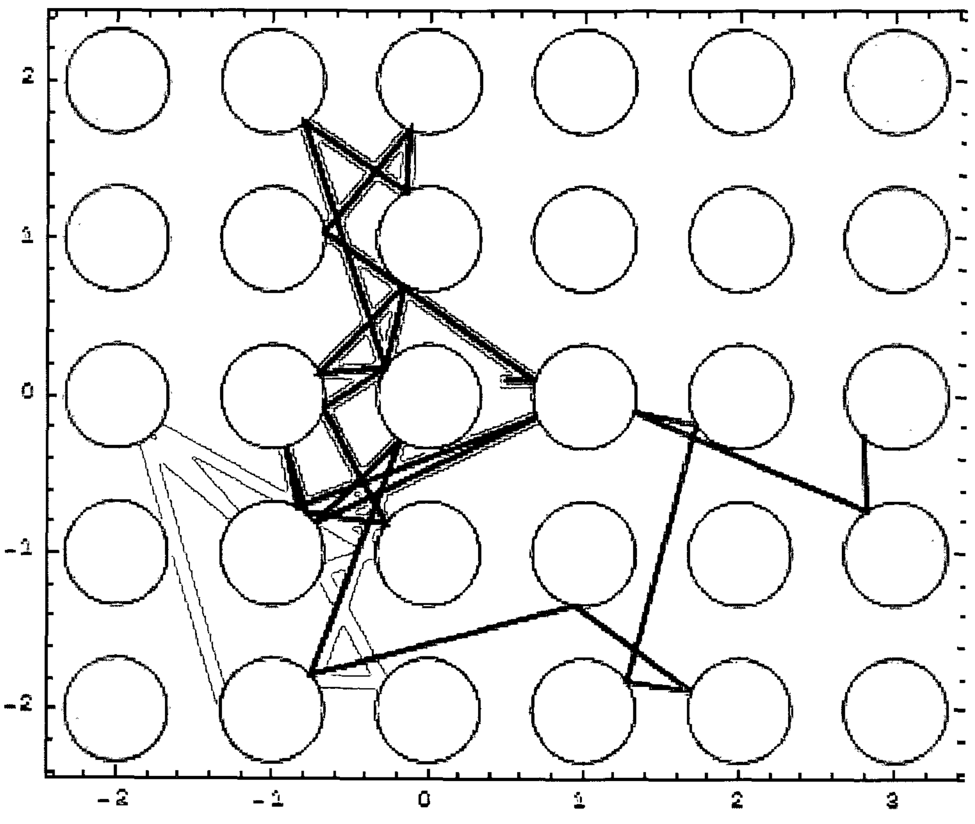
■ Initialization code

The code below defines a slightly modified version of the function presented by F. H. Simons that simulates iterations of the billiard trajectory. Each point consists of the {x,y}-coordinate at which the photon hits a mirror and the direction {u,v} it has after hitting the mirror.

```
fBill[{{xpos_, ypos_}, {xdir_, ydir_}}, wp_: 200] :=
  Module[{v = {xdir, ydir} / Norm[{xdir, ydir}], p1, p2, c1, c2, v1, v2},
   h[{a_, b_}] := 9 * {{b^2 - a^2, -2 * a * b}, {-2 * a * b, a^2 - b^2}};
  norm[a_] := Sqrt[a.a]; p = N[{xpos, ypos}, wp]; v = {xdir, ydir};
  rem = 100; While[rem > 0, c = Round[p + 2 * (v / 3)]; {p1, p2} = p;
  {c1, c2} = c; {v1, v2} = v;
time =
     Min[
      Cases[zzz = (-(1 / (3 * (v1^2 + v2^2)))) *
          (-3 * c1 * v1 + 3 * p1 * v1 - 3 * c2 * v2 + 3 * p2 * v2 +
            {-1, 1} * Sqrt[(1 - 9 * (c2 - p2)^2) * v1^2 +
               18 * (c1 - p1) * (c2 - p2) * v1 * v2 + (1 - 9 * (c1 - p1)^2) * v2^2]),
        _?Positive]]; If[time < rem, p += time * v; v = h[p - c].v;
   Return[{p, v / norm[{xdir, ydir}]}]; rem = 0, time = Min[rem, 2 / 3];
     p += time * v]; rem -= time]];
fBillTraj[{{xpos_, ypos_}, {xdir_, ydir_}}, n_, wp_] :=
  NestList[fBill[#, wp] &, {{xpos, ypos}, {xdir, ydir}}, n];
```

The graph below illustrates the divergence of the high-precision trajectory (black) and the noisy trajectory (red) after just 25 iterations.

```
trajMP = fBillTraj[{{.5, .1}, {1, 0}}, 25, MachinePrecision];
trajHP = fBillTraj[{{1/2, 1/10}, {1, 0}}, 25, 200];
background = {};
Do[background = {background, Disk[{i, j}, 1/3]}, {i, -2, 3}, {j, -2, 2}];
Graphics[
 Flatten[{Thickness[0.015], Red, Line[First /@ N[trajMP]], EdgeForm[Thin],
   RGBColor[0.8, 0.8, 0.5], background, Thick, Black, Line[First /@ N[trajHP]]}],
 Frame → True, Background → White]
```



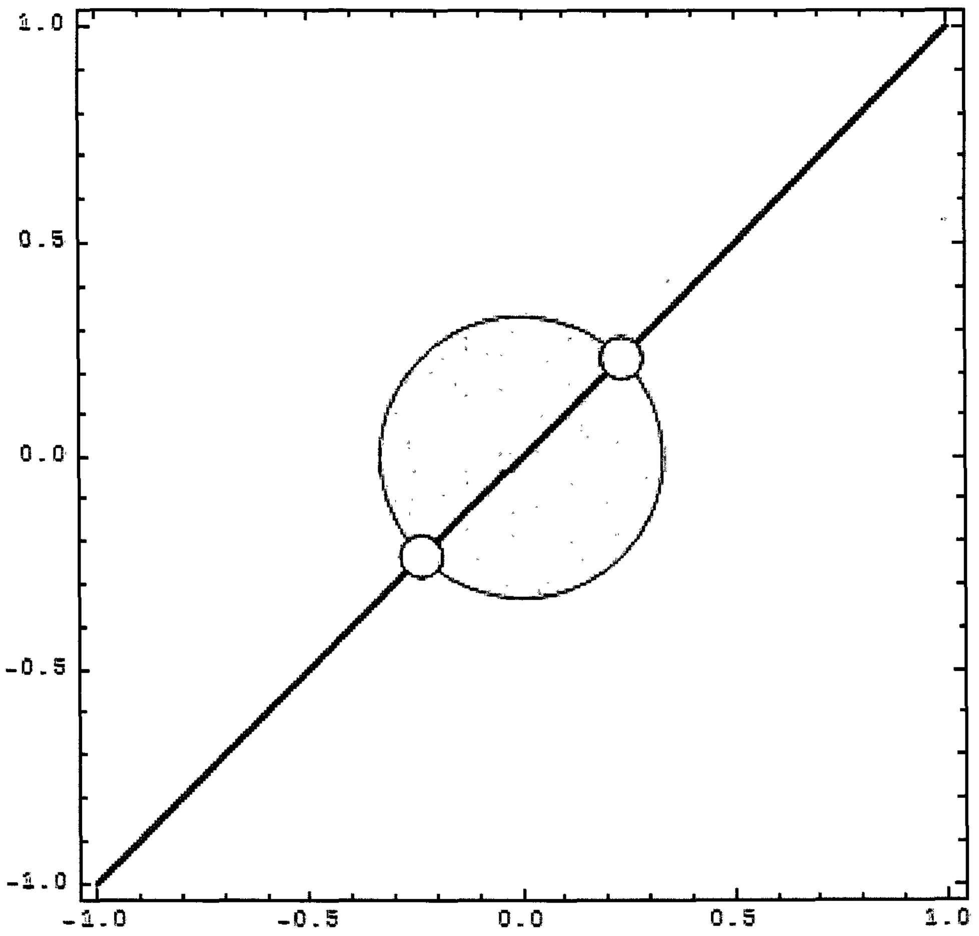
The function above correctly iterates billiard trajectories, but this can also be considered purely as an algebraic problem of lines and circles, where the line represents the path of the photon and the circle represents the mirror. It is useful to note that any shadow must clearly hit the same circles as the noisy trajectory, and that an algebraic method thus will have to include what mirrors are hit. All mirrors are circles with radius of 1/3 centered at integer coordinates, so it is sufficient to create a list of the {c,d}-coordinates of the center of each mirror in the order they are hit.

```
CenterList[billTraj_] := Round[First /@ N[billTraj]]
```

Intersection of line and circle

Knowing what mirrors are hit is however not sufficient to determine the coordinates where the photon hits the mirror. Note how there are two solutions because the line will intersect the circle both at the entrance point and at the exit point.

```
Graphics[{RGBColor[0.8, 0.8, 0.5], EdgeForm[Thin], Disk[{0, 0}, 1/3],
  Thick, Black, Line[{{-1, -1}, {1, 1}}], Red, Disk[{-Sqrt[1/18], -Sqrt[1/18]}, 1/20],
  Disk[{Sqrt[1/18], Sqrt[1/18]}, 1/20]}, Frame → True, Background → White]
```

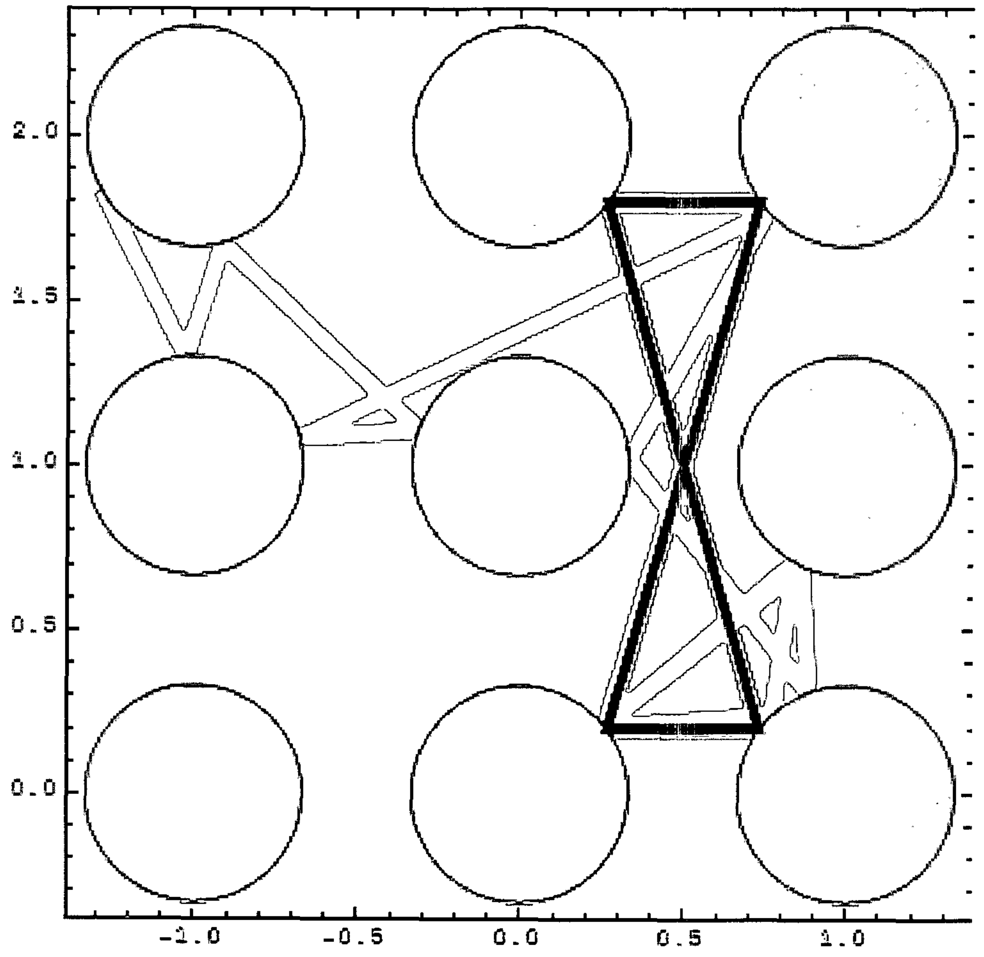


So we need a way to determine which intersection is the correct one. If the photon is coming from the left, i.e. u is positive, then the solution with the lower x-coordinate will be correct, whereas if the photon is coming from the right, i.e. u is negative, the solution with the greater x-value will be correct. Every case where u is 0 can be equivalently represented with a specific case where u = 1, so nothing is lost from excluding it here. We can collect a list similar to as above with the u-coordinate direction after each iteration, 1 if u is positive and -1 if u is negative.

```
SignList[billTraj_] := (signList = {}; i = Length[billTraj];
  Do[signList = {signList, Sign[billTraj[[x, 2, 1]]]}, {x, 1, i}];
  Return[Flatten[signList]])
```

Note how the high-precision trajectory can become periodic, such as the example below where a 4-cycle is present with a starting value of y = 2/10. Also note that the noisy-trajectory does not have this cycle.

```
trajMP = fBillTraj[{{.5, .2}, {1, 0}}, 28, MachinePrecision];
trajHP = fBillTraj[{{1/2, 2/10}, {1, 0}}, 28, 200];
background = {};
Do[background = {background, Disk[{i, j}, 1/3]}, {i, -1, 1}, {j, -0, 2}];
Graphics[
 Flatten[{Thickness[0.02], Red, Line[First /@ N[trajMP]], EdgeForm[Thin],
   RGBColor[0.8, 0.8, 0.5], background, Thickness[0.01], Black,
   Line[First /@ N[trajHP]]}], Frame → True, Background → White]
```



```
trajHP = fBillTraj[{{1/2, 1/10}, {1, 0}}, 20, 200];
trajMP = fBillTraj[{{.5, .1}, {1, 0}}, 200, MachinePrecision];

centerList = CenterList[trajMP];
signList = SignList[trajMP];
```

The code below is a hard-coding of the solution given when solving for the {x,y}-coordinates of the intersection between a line and a circle. The two solutions are consistent with the idea above, where the direction of the photon will determine which of the solutions give the coordinates where the photon hits a mirror. {c1, d1} is here the coordinate of the mirror that is hit.

```
LCIntersectSol =
  {{y → 1/(3 u0 (u0^2 + v0^2))
      (3 d1 u0 v0^2 + 3 u0^2 v0 (c1 - x0) -
        v0
         √(u0^2 (v0^2 (1 - 9 (c1 - x0)^2) + u0^2 (1 - 9 (d1 - y0)^2) +
              18 u0 v0 (c1 - x0) (d1 - y0))) + 3 u0^3 y0),
    x → 1/(3 (u0^2 + v0^2))
      (3 c1 u0^2 -
        √(u0^2 (v0^2 (1 - 9 (c1 - x0)^2) + u0^2 (1 - 9 (d1 - y0)^2) +
             18 u0 v0 (c1 - x0) (d1 - y0))) + 3 v0 (d1 u0 + v0 x0 - u0 y0))},
   {y → 1/(3 u0 (u0^2 + v0^2))
      (3 d1 u0 v0^2 + 3 u0^2 v0 (c1 - x0) +
        v0
         √(u0^2 (v0^2 (1 - 9 (c1 - x0)^2) + u0^2 (1 - 9 (d1 - y0)^2) +
              18 u0 v0 (c1 - x0) (d1 - y0))) + 3 u0^3 y0),
    x → 1/(3 (u0^2 + v0^2))
      (3 c1 u0^2 +
        √(u0^2 (v0^2 (1 - 9 (c1 - x0)^2) + u0^2 (1 - 9 (d1 - y0)^2) +
             18 u0 v0 (c1 - x0) (d1 - y0))) + 3 v0 (d1 u0 + v0 x0 - u0 y0))}};
LCIntersectSolLeft = {x, y} /. LCIntersectSol[[1]];
LCIntersectSolRight = {x, y} /. LCIntersectSol[[2]];
LCIntersectLeft[{x0_, y0_}, {u0_, v0_}, {c1_, d1_}] = LCIntersectSolLeft;
LCIntersectRight[{x0_, y0_}, {u0_, v0_}, {c1_, d1_}] = LCIntersectSolRight;
```

Direction after hitting a circle :

The next step is to find a algebraic solution for the direction after an intersection with a circle. There will be two solutions for the direction of a photon after hitting the mirror, one for each of the two solutions for the intersection of a line and a circle.

The code below is similarly a hard coding of the solution given when solving for the direction {u,v} of the line after hitting a circle.

```
LCDirectionSol =
  {-9 (c1^2 u0 - d1^2 u0 + 2 c1 d1 v0 - 2 c1 u0 x - 2 d1 v0 x + u0 x^2 + 2 (d1 u0 + v0 (-c1 + x)) y -
      u0 y^2),
   9 (c1^2 v0 - d1^2 v0 - 2 u0 x y - 2 c1 (d1 u0 + v0 x - u0 y) + 2 d1 (u0 x + v0 y) + v0 (x^2 - y^2))};
```

This can be further simplified to:

```
LCDirectionSol = {9 (u0 (-(c1 - x)^2 + (d1 - y)^2) - 2 v0 (c1 - x) (d1 - y)) / Sqrt[u0^2 + v0^2],
   9 (v0 ((c1 - x)^2 - (d1 - y)^2) - 2 u0 (c1 - x) (d1 - y)) / Sqrt[u0^2 + v0^2]};
```

$$\text{LCDirectionSol} = \left\{\frac{9\left(u0\left(-(c1-x)^2+(d1-y)^2\right)-2\,v0\,(c1-x)\,(d1-y)\right)}{\sqrt{u0^2+v0^2}},\ \frac{9\left(v0\left((c1-x)^2-(d1-y)^2\right)-2\,u0\,(c1-x)\,(d1-y)\right)}{\sqrt{u0^2+v0^2}}\right\};$$

```
xLeft = LCIntersectSolLeft[[1]];
yLeft = LCIntersectSolLeft[[2]];
xRight = LCIntersectSolRight[[1]];
yRight = LCIntersectSolRight[[2]];

LCDirectionSolLeft = LCDirectionSol /. {x → xLeft, y → yLeft};
LCDirectionSolRight = LCDirectionSol /. {x → xRight, y → yRight};

LCDirectionLeft[{x0_, y0_}, {u0_, v0_}, {c1_, d1_}] = LCDirectionSolLeft;
LCDirectionRight[{x0_, y0_}, {u0_, v0_}, {c1_, d1_}] = LCDirectionSolRight;
```

Combining the two above :

Combining the algebraic solutions for the intersection with a circle and direction after hitting a circle gives us the two possible algebraic solutions needed to iterate the billiard problem.

```
BillVectorLeft[{x0_, y0_, u0_, v0_}, {c1_, d1_}] :=
  Flatten[{LCIntersectLeft[{x0, y0}, {u0, v0}, {c1, d1}],
    LCDirectionLeft[{x0, y0}, {u0, v0}, {c1, d1}]}];
BillVectorRight[{x0_, y0_, u0_, v0_}, {c1_, d1_}] :=
  Flatten[{LCIntersectRight[{x0, y0}, {u0, v0}, {c1, d1}],
    LCDirectionRight[{x0, y0}, {u0, v0}, {c1, d1}]}];
```

Constructing the matrices :

This can be used to create partial-derivative matrices. The two variables, {x,y} in the algebraic solution for the intersection of a line and a circle, will form a 2-by-4 matrix consisting of the partial-derivative of x and y with respect to each of the four variables in question, i.e. {x,y,u,v}. Similarly the partial derivatives of the direction after hitting a circle will also form a 2-by-4 matrix. There are two cases of each of these 2-by-4 matrices because of the two possible intersections with the circle.

```
MatrixLeft =
  Identity[D[LCIntersectLeft[{x0, y0}, {u0, v0}, {c1, d1}], {{x0, y0, u0, v0}}]];
MatrixRight =
  Identity[D[LCIntersectRight[{x0, y0}, {u0, v0}, {c1, d1}], {{x0, y0, u0, v0}}]];
MatrixDirectionLeft =
  Identity[D[LCDirectionLeft[{x0, y0}, {u0, v0}, {c1, d1}], {{x0, y0, u0, v0}}]];
MatrixDirectionRight =
  Identity[D[LCDirectionRight[{x0, y0}, {u0, v0}, {c1, d1}], {{x0, y0, u0, v0}}]];

TotalDerivativeLeft = Join[MatrixLeft, MatrixDirectionLeft];
TotalDerivativeRight = Join[MatrixRight, MatrixDirectionRight];
```

Combining the partial derivatives of the {x,y} coordinates and the {u,v} direction provides us with a 4-by-4 matrix that represents the sensitivity of all the variables at a certain iteration. Naturally we also here have the two cases, with the possibilities being u being negative or positive. So knowing the coordinates of the center of the mirror at the nth iterate and the sign of u gives us the sensitivity at that point to changes in the variables at the previous point.

```
sensMat[n_][state_] :=
  If[signList[[n]] == 1, TotalDerivativeLeft, TotalDerivativeRight] /.
    Thread[{x0, y0, u0, v0} → state] /. Thread[{c1, d1} → centerList[[n + 1]]]
```

The Jacobian is thus the list of sensitivity matrices to changes in the starting y-value. Note that the starting y-value, not being constrained to be on a circle is the only point from which this can be treated as a one dimensional problem. Since at any point on a circle a small change in y will also mean a change in x, u, and v, and the y-value of the next iterate is dependent on all of {x,y,u,v}, it follows from the Chain rule, that the sensitivity of y_n to changes in y_{n-1} will be $\frac{\partial y_n}{\partial y_{n-1}} + \frac{\partial y_n}{\partial x_{n-1}} \frac{\partial x_{n-1}}{\partial y_{n-1}} + \frac{\partial y_n}{\partial u_{n-1}} \frac{\partial u_{n-1}}{\partial y_{n-1}} + \frac{\partial y_n}{\partial v_{n-1}} \frac{\partial v_{n-1}}{\partial y_{n-1}}$. But since y_n depends on $\{x_{n-1}, y_{n-1}, u_{n-1}, v_{n-1}\}$, and each of $\{x_{n-1}, y_{n-1}, u_{n-1}, v_{n-1}\}$ depends on $\{x_{n-2}, y_{n-2}, u_{n-2}, v_{n-2}\}$, this implies that the dot product of all sensitivity matrices to the nth iterate will give us the sensitivity of all 4 variables at the nth point to changes in the starting y-value. When treating it as a 1-dimensional problem, we are only really interested in the sensitivity of the y-value at the nth iterate to changes in the starting y-value, but because this is inherently a 4-dimensional problem the 4-by-4 matrices are necessary to compute the correct sensitivity. So, $\frac{\partial y_n}{\partial y_0}$ will be the value of the second entry on the second row of the Jacobian, where the Jacobian is the list of the 4-by-4 matrices that are the dot product of the first i^{th} sensitivity matrices.

```
I4 = N[{{1, 0, 0, 0}, {0, 1, 0, 0}, {0, 0, 1, 0}, {0, 0, 0, 1}}, 200];
J[y0_?NumericQ, n_] := (
   hpsol = fBillTraj[{{1 / 2, y0}, {1, 0}}, n, Max[n, 17] + 200];
   matlist = Table[sensMat[i][Flatten[hpsol[[i]]]], {i, 1, n}];
   #[[{2}]] & /@ Map[ #[[2]] &, FoldList[#2.#1 &, I4, matlist], {2}])
```

■ Shadowing code

The residual is simply the absolute difference between the high-precision trajectory and the noisy trajectory.

```
residualList[y0_, n_] :=
 Flatten[(First /@ fBillTraj[{{1 / 2, y0}, {1, 0}}, n, 200]) -
   First /@ Take[trajMP, n + 1]]
```

Now, we can shadow the Billiard trajectory using the same procedure as we did for the quadratic map. That is to say, we can attempt to minimize the sum of square residuals using the LevenbergMarquardt method, and feed it the Jacobian that was worked out by solving the billiard trajectory algebraically.

```
BillShadowOneStep[noisy_List, n_Integer, start_] := (
  Clear[residual];
  residual[Y0_?NumericQ] :=
   (Flatten[
     (#[[2]] & /@ (First /@ fBillTraj[{{1 / 2, Y0}, {1, 0}}, n, Max[n, 200]])) -
      (#[[2]] & /@ First /@ SetPrecision[Take[noisy, n + 1], 200])]);
  FindMinimum[0, {Y0, SetPrecision[start, 200]},
   WorkingPrecision → 200,
   Method → {"LevenbergMarquardt", "Residual" -> (residual[Y0]),
     "Jacobian" → J[Y0, n]}])
```

Similarly for shadowing many iterations, we need to shadow in steps to get us in a place where the search is productive.

```
BillShadowInSteps[noisy_List, n_Integer, {startstep_, stepsize_}, opts___] := (

  Do[sol = Y0 /. BillShadowOneStep[SetPrecision[noisy, 200], nn,
       If[nn == startstep, N[1/10, 200], SetPrecision[sol, 200]]][[2]],
     {nn, Append[Most[Range[startstep, n, stepsize]], n]}];
  sol);
```

Looking at the maximum absolute error shows us that the shadows are good.

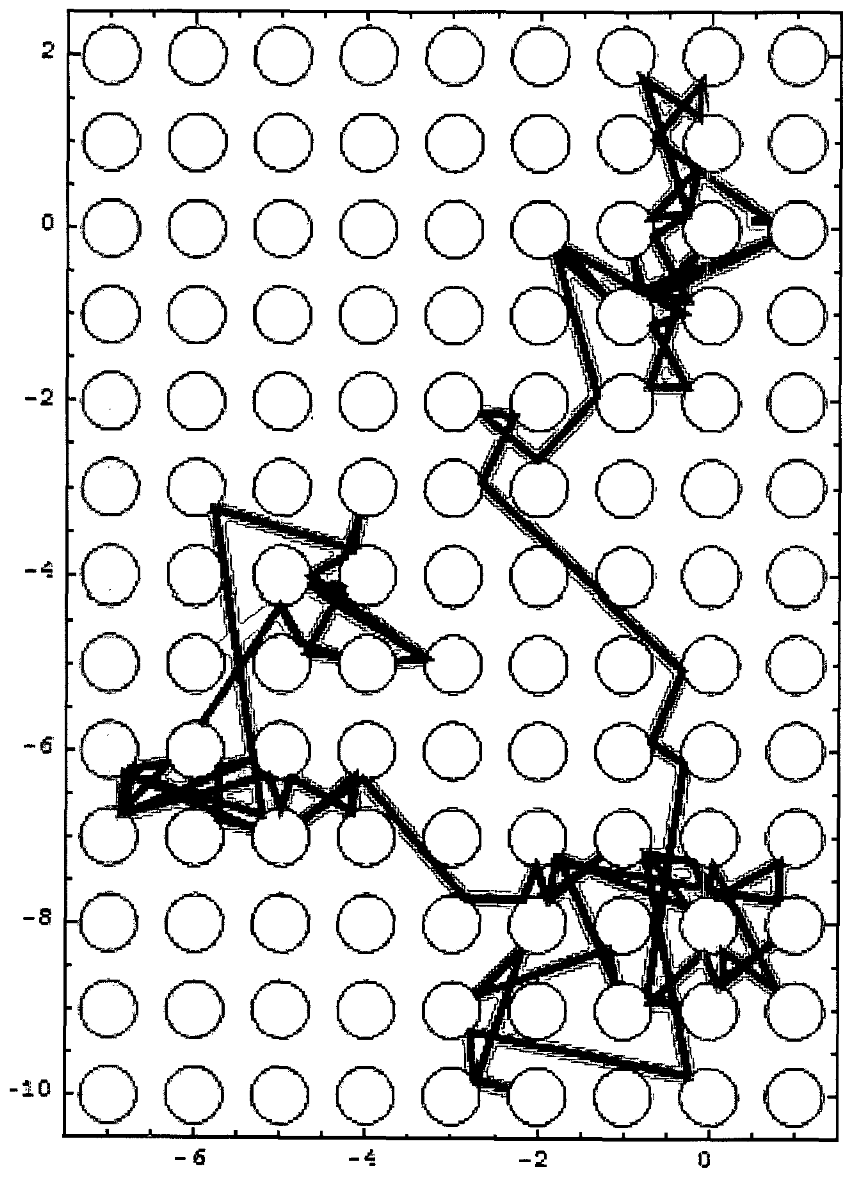
```
trajMP = fBillTraj[{{.5, .1}, {1, 0}}, 200, MachinePrecision];

Sol1 = Y0 /. BillShadowOneStep[trajMP, 10, N[1/10, 200]][[2]]

0.09999999999999999682622077721912127869914629267551266116170954007483909421859`
 470367604530787412970578436426792201909977331271078841492315845812422915079866`
 9101630344871378065259864954463016058437888020185

Sol2 = Y0 /. BillShadowInSteps[trajMP, 50, {10, 10}][[2]]

0.09999999999999999682622079468781564846294583355960160199350140876868118804898`
  20902417901572847989280682201971034936347938633247324752124279417366716174379`
  5933642053886417235641951715954603891468962869240;

Max[Abs[N[residualList[Sol1, 10]]]]

2.22045 × 10^-16

Max[Abs[N[residualList[Sol2, 50]]]]

8.88178 × 10^-16
```

As evident by a residual error of at most 10^-15, we can easily shadow a Billiard trajectory out to many iterations. The following shows that a shadowing is possible out to 94 iterations within 10^-10!

```
Sol3 = BillShadowInSteps[trajMP, 94, {10, 10}]

 Y0 /. {Y0,
   0.09999999999999999682622079468781564846294583355960160199350187826366445653\
    703392838483613471043586801408491119929176272800595061140403709651362115984\
    82060465283259999597557230593294895086895316826100166}

Max[
 Abs[
  N[
   residualList[
    0.09999999999999999682622079468781564846294583355960160199350187826366445653\
     37033928384836134710435868014084911199291762728005950611404037096513621159\
     84820604652832599995975572305932948950868953168261001658637572802903961030\
     5417035286`200., 94]]]]

 3.16529×10^-11
```

Below is a graph illustrating the successfully showing out to 94 iteration. Note how the noisy trajectory (Red) and high-precision trajectory (Black) diverge (on a visual scale) after roughly 100 iterations (shown out to 102).

```
trajMP = fBillTraj[{{.5, .1}, {1, 0}}, 95, MachinePrecision];
trajHP = fBillTraj[{{1/2, Sol3}, {1, 0}}, 95, 200];
background = {};
Do[background = {background, Disk[{i, j}, 1/3]}, {i, -7, 1}, {j, -10, 2}];
Graphics[
 Flatten[{Thickness[0.02], Red, Line[First /@ N[trajMP]], EdgeForm[Thin],
   RGBColor[0.8, 0.8, 0.5], background, Thickness[0.01], Black,
   Line[First /@ N[trajHP]]}], Frame → True, Background → White]
```


2
0
-2
-4
-6
-8
-10
-6
-4
-2
0

■ Billiard Trajectory Revisited

Now, it is interesting that one can actually use the same method, but without feeding it the Jacobian and still get a shadowseed. This is possible because the LevenbergMarquardt method in Mathematica can approximate the derivatives and work from that alone. This is especially noteworthy because computing the Jacobian is the most time-consuming part of finding a shadowseed because it requires running many matrix operations. Below is the shadowing without using the Jacobian, with the associated apparent time improvements (roughly 10x) and showing that they find the same shadow-seed.

```
BillShadowOneStep2[noisy_List, n_Integer, start_] := (
  Clear[residual];
  residual[Y0_?NumericQ] :=
   (Flatten[
     (#[[2]] & /@ (First /@ fBillTraj[{{1 / 2, Y0}, {1, 0}}, n, Max[n, 200]])) -
      (#[[2]] & /@ First /@ SetPrecision[Take[noisy, n + 1], 200])]);
  FindMinimum[0, {Y0, SetPrecision[start, 200]},
   WorkingPrecision → 200,
   Method → {"LevenbergMarquardt", "Residual" -> (residual[Y0])}])
```

```
BillShadowOneStep[trajMP, 10, 1 / 10] // Timing
```

```
{2.121,
 {2.42072065873460348343951944012250884679558188277421812437283307018140358169⋱
   58751011907588256328921765803077191377721180175742235198128646945211751604O⋱
   27388158959374521001815266086951883630269355513746 × 10^-32, {Y0 →
   0.0999999999999999968262207772191212786991462926755126611617095400748390942⋱
    1859470367604530787412970578436426792201909977331271078841492315845812422⋱
    915079869101630344871378065259864954463016058437888020185}}}
```

```
BillShadowOneStep2[trajMP, 10, 1 / 10] // Timing
```

```
{0.156,
 {2.42072065873460348343951944012250884679558188277421812437283307018140358169⋱
   587510119075882563289217658030771913777211801757422351981286469452117516040⋱
   27388158959374521001815266086951883630769293293496 × 10^-32, {Y0 →
   0.0999999999999999968262207772191212786991462926755126611617095400748390942⋱
    1859470367604530787412970578436426792201909973688615757439640733784625809⋱
    728763755537638672718050476434075695674583185578678889411}}}
```

```
BillShadowInSteps2[noisy_List, n_Integer, {startstep_, stepsize_}, opts___] := (

  Do[sol = Y0 /. BillShadowOneStep2[SetPrecision[noisy, 200], nn,

       If[nn == startstep, N[1/10, 200], SetPrecision[sol, 200]]][[2]],

     {nn, Append[Most[Range[startstep, n, stepsize]], n]}];

  sol);
```

```
BillShadowInSteps[trajMP, 50, {10, 10}] // Timing

{45.256,
 0.0999999999999999682622079468781564846294583355960160199350140876868118804 89\
  820902417901572847989280682201971034936347938633247324752124279417366716 1743\
  79593364205388641723564195171595460389146896286 9240}

Max[
 Abs[
  N[
   residualList[
    0.0999999999999999682622079468781564846294583355960160199350140876868118804\
     898209024179015728479892806822019710349363479386332473247521242794173667 16\
     174379593364205388641723564195171595460389146896286923983772849138103490 51\
     7615618032`200., 20]]]]

2.22045×10^-16

BillShadowInSteps2[trajMP, 50, {10, 10}] // Timing

{5.569,
 0.0999999999999999682622079468781564846294583355960160199350140876868118804 89\
  820902417901572847989280682201971034936346915024443198288691641587773552 4721\
  74346980657860312173776445630030209599274435 9559113}

Max[
 Abs[
  N[
   residualList[
    0.0999999999999999682622079468781564846294583355960160199350140876868118804\
     898209024179015728479892806822019710349363469150244431982886916415877735 52\
     472174346980657860312173776445630030209599274435955911338363049168352523 36\
     8294510778`200., 20]]]]

2.22045×10^-16

BillShadowInSteps2[trajMP, 60, {10, 10}] // Timing

{11.528,
 0.0999999999999999682622079468781564846294583355960160199350187826366430069 62\
  836845389177251563493018746892281311404816603486012571033774672509744237 0916\
  98756106477849531228362675844174079713324169 9979997}
```

From this we can see the consistent time improvement one gets by not manually computing the Jacobian, but rather relying on the approximations. Note however that although it may be faster to simply approximate the sensitivity, using the Jacobian allows for shadowing more iterates as the approximations eventually lead to a failure to find a local minimum.

■ Conclusion

Using the computational power now available, we can successfully use optimization to compute high-precision shadowseeds that shadow noisy trajectories to within a small value. It is interesting to note the relative uniqueness of the shadowseeds as evident by the shadowing error graphs, and although we have not proven that the shadowseeds found are the optimal shadowseeds, they certainly indicate that the theories set forth by Chow and Palmer[1] may be better than what was previously though, and indicates that τ-first gives us a remarkably good estimate of a shadowseed. Shadowing was successfully accomplished for both the quadratic map and for billiard trajectories out to many iterations, providing very satisfying results. Machine power is however still a limitation to this methodology when attempting to shadow very far, because the high-precision trajectory becomes cumbersome to deal with. The methodology presented in this paper seems like it should be applicable in shadowing any system that exhibits sensitive dependence to the initial condition.

■ Bibliography

[1]Shui-Nee Chow and Kenneth J. Palmer, "On the numerical computation of orbits of dynamic systems: The one-dimensional case", Journal of Dynamic and Differential Equations, 3, 1991, 361 - 380.

[2]Stan Wagon, "Mathematica in Action, 2nd Edition", 1999, 153 - 166.

[3]Folkmar Bornemann, Dirk Laurie, Stan Wagon and Jörg Waldvogel, "The SIAM 100-Digit Challenge: A Study in High-Accuracy Numerical Computing", 2005, 35 - 43.

[4]Nick Trefethen, "SIAM 100-Digit Challenge", Oxford University, 2007.

[5]T.Sauer, "Computer arithmetic and sensitivity of natural measure", Journal of Difference Equations and Applications, 11, 2005, 669 - 676.