Cantor Square, speed comparisson

GONZALO GARCÍA ALARCÓN ESTRADA, **********
Dec / 26 / 2017
University of Toronto, Shameless Mathematica

Cantor Square, speed comparisson:

Draw an approximations of the Cantor Square using different algorithms. Compare the speed of the algorithms (including displaying the image).

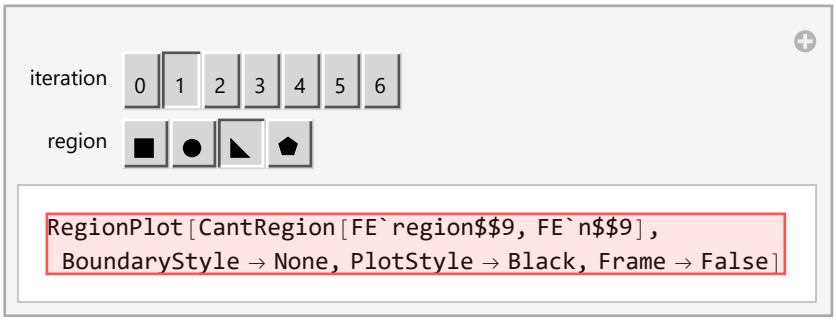
These different methods for plotting apporximations to the Cantor Square are the result on my different approaches to the Cantor Shadows Problem. Out of curiosity, I decided to compare how fast they are.

1. CantRegion[r,n] - Using Iterated Function Systems and Regions:

Core idea: Apply set of transformations to an initial region.
Pros: the Cantor Square is the atractor of the IFS, hence it does not depend on the initial region. Ready to use projections based on *Mathematica* existence operators.

```
t1 = AffineTransform[IdentityMatrix[2] * (1/3)];
t2 = AffineTransform[{IdentityMatrix[2] * (1/3), {0, 2/3}}];
t3 = AffineTransform[{IdentityMatrix[2] * (1/3), {2/3, 2/3}}];
t4 = AffineTransform[{IdentityMatrix[2] * (1/3), {2/3, 0}}];
T[r_] := RegionUnion[TransformedRegion[r, t1],
   TransformedRegion[r, t2], TransformedRegion[r, t3], TransformedRegion[r, t4]];

CantRegion[r_, n_Integer?NonNegative] := Nest[T, r, n];
```

```
Manipulate[RegionPlot[CantRegion[region, n], BoundaryStyle → None,
  PlotStyle → Black, Frame → False], {{n, 1, "iteration"}, Range[7] - 1, Setter},
 {{region, Triangle[]}, {Rectangle[] → Graphics[Rectangle[], ImageSize → 10], Disk[] →
    Graphics[Disk[], ImageSize → 10], Triangle[] → Graphics[Triangle[], ImageSize → 10],
   RegularPolygon[5] → Graphics[RegularPolygon[5], ImageSize → 10]}}]
```

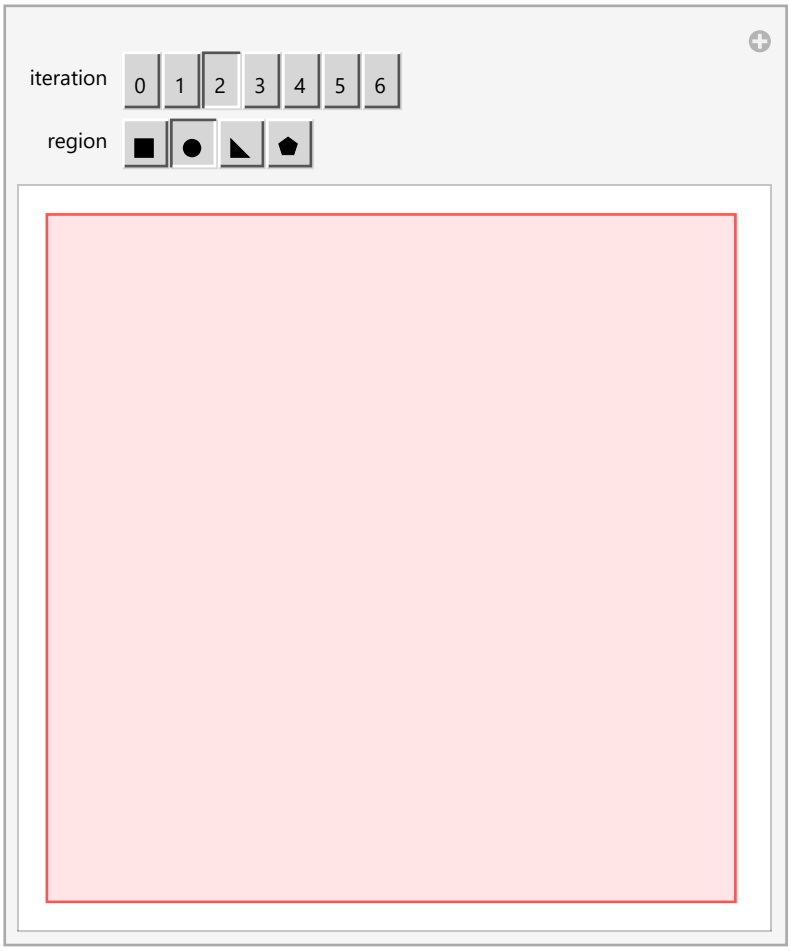


2. CantGraphics[r,n] - Using Iterated Function Systems and Transformed Graphics:

Core idea: Apply set of transformations to an initial graphics.
Pros: the Cantor Square is the atractor of the IFS, hence it does not depend on the initial region.

```
f1[g_] := Scale[g, 1/3, {0, 0}];
f2[g_] := Translate[Scale[g, 1/3, {0, 0}], {0, 2/3}];
f3[g_] := Translate[Scale[g, 1/3, {0, 0}], {2/3, 2/3}];
f4[g_] := Translate[Scale[g, 1/3, {0, 0}], {2/3, 0}];

cstep[g_] := {f1[g], f2[g], f3[g], f4[g]};

CantGraphics[n_Integer, r_] := Nest[cstep, r, n];
```

```
Manipulate[Graphics[CantGraphics[n, region]], {{n, 2, "iteration"}, Range[7] - 1, Setter},
 {{region, Disk[]}, {Rectangle[] → Graphics[Rectangle[], ImageSize → 10], Disk[] →
    Graphics[Disk[], ImageSize → 10], Triangle[] → Graphics[Triangle[], ImageSize → 10],
   RegularPolygon[5] → Graphics[RegularPolygon[5], ImageSize → 10]}}]
```




```
r
```

```
r
```

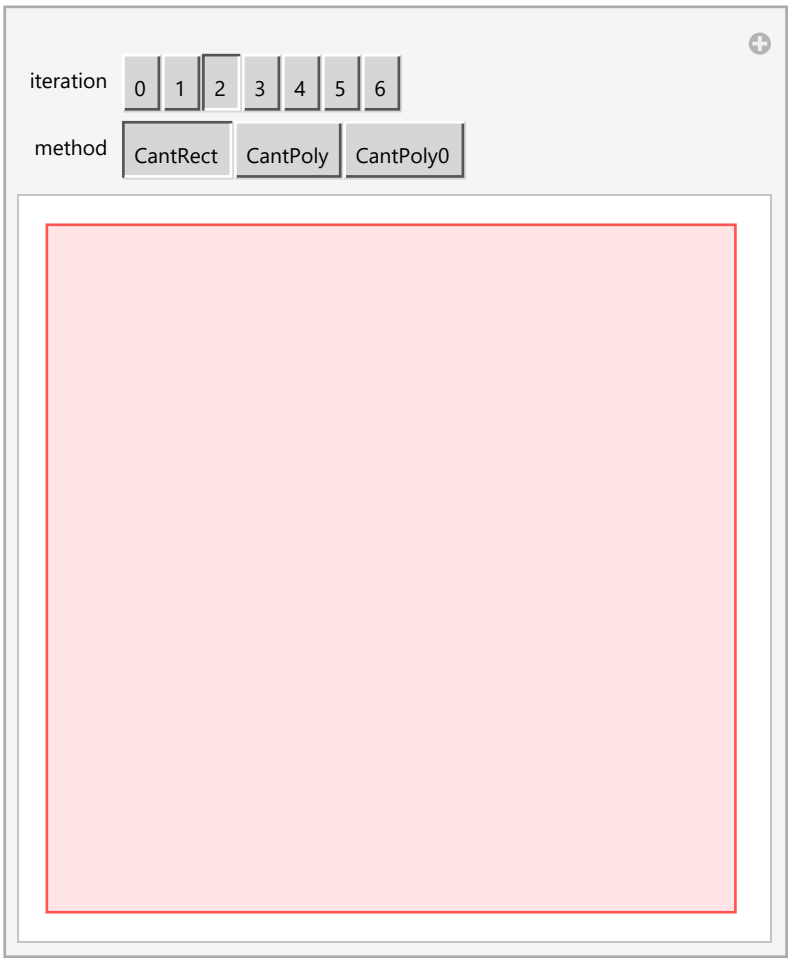
3. Using binary expression of cantor set:

Core idea: Generate the coordinates of the corners of the Cantor Square from the binary expression of the Cantor Set.
Plot with Graphics.
Pros: ready for other computations based on the Cantor Square, such as rotations and projections.

```
left[binary_List] := binary.Table[2/3^k, {k, Length[binary]}];

CantCorners[n_Integer?NonNegative] := Flatten@Table[
    C@{left[binx], left[biny]},
    {binx, Tuples[{0, 1}, n]}, {biny, Tuples[{0, 1}, n]}
   ];

(* three Cantor Square functions... CantRect is NOT suitable for rotations,
CantPoly is *)
CantRect[n_Integer?NonNegative] :=
  CantCorners[n] /. C[p_List] :> Rectangle[p - {0.5, 0.5}, p + {1/3^n, 1/3^n} - {0.5, 0.5}];
```

```
CantPoly[n_Integer?NonNegative] :=
  CantCorners[n] /. C[p_List] :> Polygon[{p - {0.5, 0.5}, p + {1/3^n, 0} - {0.5, 0.5},
      p + {1/3^n, 1/3^n} - {0.5, 0.5}, p + {0, 1/3^n} - {0.5, 0.5}}];
CantPoly0[n_Integer?NonNegative] := CantPoly[n, 0]; (* I need
 CantPoly0 one to set it as a Manipulate option *)

Rot[θ_] := {{Cos[θ], -Sin[θ]}, {Sin[θ], Cos[θ]}};

CantPoly[n_Integer?NonNegative, θ_] :=
  CantCorners[n] /. C[p_List] :> Polygon[{p - {0.5, 0.5}, p + {1/3^n, 0} - {0.5, 0.5},
      p + {1/3^n, 1/3^n} - {0.5, 0.5}, p + {0, 1/3^n} - {0.5, 0.5}}.Rot[θ]];
```

```
Manipulate[Graphics[method[n]], {{n, 2, "iteration"}, Range[7] - 1, Setter},
 {method, {CantRect, CantPoly, CantPoly0}}]
```



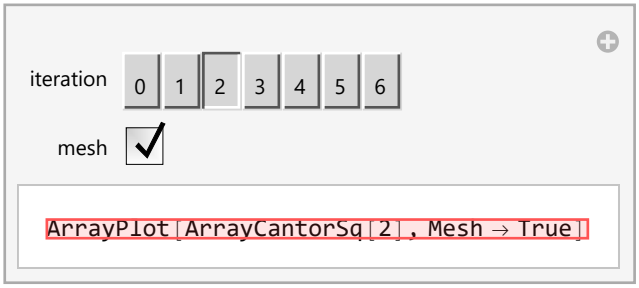
4. Iterative replacing of matrix elements:

Core idea: Matrix, replace 1 by first iteration of Cantor Square, padd with zeroes.
Plot with ArrayPlot.

Cons: not ready for rotation. Restricted to Mesh.

```
ArrayCantorSq[n_] :=
 Module[{a, o = {{1, 0, 1}, {0, 0, 0}, {1, 0, 1}}, z = {{0, 0, 0}, {0, 0, 0}, {0, 0, 0}}},
  a = {{1}};
  For[i = 1, i ≤ n, i++,
   a = ArrayFlatten[a /. {1 → o, 0 → z}]
  ];
  Return[a]
 ]
```

```
Manipulate[ArrayPlot[ArrayCantorSq[n], Mesh → mesh],
 {{n, 2, "iteration"}, Range[7] - 1, Setter}, {{mesh, True}, {True, False}}]
```



5. SPEED COMPARISSON

```
(* number of iterations of the CS to compute and plot *)
M = 6;
```

(-1-)

```
r = Rectangle[];
```

```
TCantRegion = Map[Timing[CantRegion[r, #]][[1]] &, Range[M]]
```

```
{0., 0.03125, 0.125, 0.390625, 1.65625, 7.45313}
```

```
If[(M) > 3, MM = 4, MM = M];
TPlotCantRegion =
 Map[Timing[RegionPlot[CantRegion[r, #], BoundaryStyle → None, PlotStyle → Black,
       Frame → False]][[1]] &, Range[MM]]
```

```
{0.046875, 1.0625, 1.67188, 11.4219}
```

(-2-)

```
TCantGraphics = Map[Timing[CantGraphics[r, #]][[1]] &, Range[M]];
```

```
TPlotCantGraphics = Map[Timing[Graphics[CantGraphics[#, r]]][[1]] &, Range[M]]
```

```
{0., 0., 0., 0., 0., 0.}
```

(-3-)

```
TCantRect = Map[Timing[CantRect[#]][[1]] &, Range[M]]
```

```
{0., 0., 0.015625, 0., 0.03125, 0.109375}
```

```
TPlotCantRect = Map[Timing[Graphics[CantRect[#]]][[1]] &, Range[M]]
```

```
{0., 0., 0., 0., 0.03125, 0.109375}
```

```
TCantPoly = Map[Timing[CantPoly[#]][[1]] &, Range[M]]
```

```
{0.015625, 0., 0., 0., 0.03125, 0.125}
```

```
TPlotCantPoly = Map[Timing[Graphics[CantPoly[#]]][[1]] &, Range[M]]
```

```
{0., 0., 0., 0.015625, 0.03125, 0.140625}
```

```
TCantPoly0 = Map[Timing[CantPoly0[#]][[1]] &, Range[M]]
```

```
{0., 0., 0.015625, 0., 0.046875, 0.15625}
```

```
TPlotCantPoly0 = Map[Timing[Graphics[CantPoly0[#]]][[1]] &, Range[M]]
```

```
{0., 0., 0., 0.015625, 0.046875, 0.140625}
```

(-4-)

```
TArrayCantorSq = Map[Timing[ArrayCantorSq[#]][[1]] &, Range[M]]
```

```
{0., 0., 0., 0., 0.015625, 0.078125}
```

```
TPlotArrayCantorSq = Map[Timing[ArrayPlot[ArrayCantorSq[#]]][[1]] &, Range[M]]
```

```
{0.015625, 0., 0., 0.015625, 0., 0.078125}
```

(-plotting-)

```
x = Range[M];
```

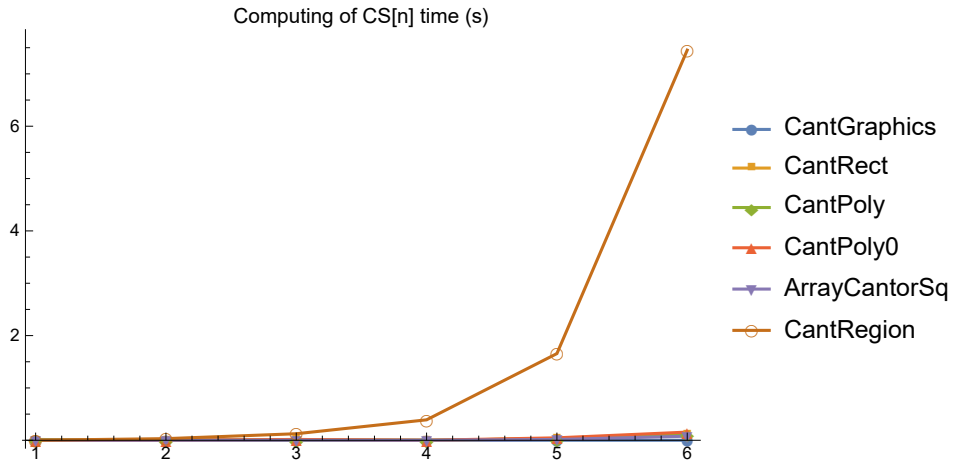
```
data = {TCantGraphics, TCantRect, TCantPoly, TCantPoly0, TArrayCantorSq, TCantRegion};
```

```
dataPlot = {TPlotCantGraphics, TPlotCantRect,
   TPlotCantPoly, TPlotCantPoly0, TPlotArrayCantorSq, TPlotCantRegion};
```

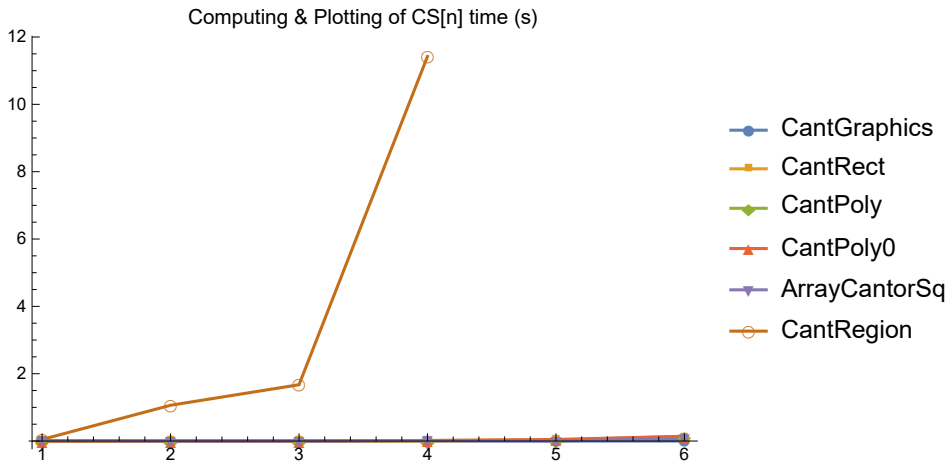
```
TPlotCantRegion
```

```
{0.046875, 1.0625, 1.67188, 11.4219}
```

```
ListLinePlot[data, PlotMarkers → Automatic, PlotLegends →
  {"CantGraphics", "CantRect", "CantPoly", "CantPoly0", "ArrayCantorSq", "CantRegion"},
 PlotRange → All, PlotLabel → "Computing of CS[n] time (s)"]
```

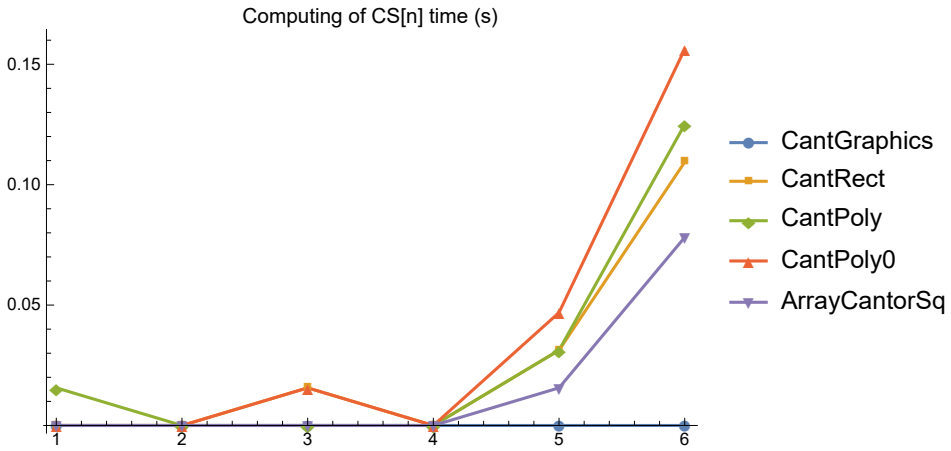


```
ListLinePlot[dataPlot, PlotMarkers → Automatic, PlotLegends →
  {"CantGraphics", "CantRect", "CantPoly", "CantPoly0", "ArrayCantorSq", "CantRegion"},
 PlotRange → All, PlotLabel → "Computing & Plotting of CS[n] time (s)"]
```

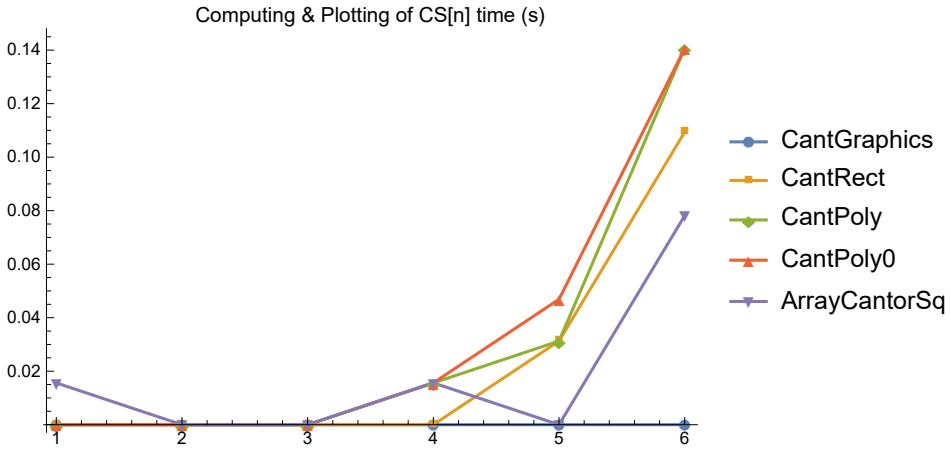


CantRegion is way too slow... I will eliminate it from the plots:

```
ListLinePlot[data // Most, PlotMarkers → Automatic, PlotLegends →
  {"CantGraphics", "CantRect", "CantPoly", "CantPoly0", "ArrayCantorSq", "CantRegion"},
 PlotRange → All, PlotLabel → "Computing of CS[n] time (s)"]
```



```
ListLinePlot[dataPlot // Most, PlotMarkers → Automatic, PlotLegends →
  {"CantGraphics", "CantRect", "CantPoly", "CantPoly0", "ArrayCantorSq", "CantRegion"},
 PlotRange → All, PlotLabel → "Computing & Plotting of CS[n] time (s)"]
```



CONCLUSION:

CantRegion is by far the slowest. Both computing and plotting Regions is slower than the rest.

CantGraphics is the fastest by far, and remains equally fast as n increases. Therefore this is the best option for plotting an approximation to the cantor square with large n, but is not so easy to manipulate to compute other things (such as rotations or projections).

CantRect, CantPoly and CantPoly0 are in escence the same algorithm based on the binary expression of the cantor set. Including rotations (CantPoly0) makes it a bit slower.

In general, other than using RegionPlot, plotting does not take too much extra time.