Life-Cycle Problems: Arbitrary Numbers of Periods

In[332]:=
```
x = 0; Remove["Global`*"]; DateList[Date[]] // Most
```

Out[332]=
```
{2020, 3, 9, 21, 27}
```

Life-Cycle Consumption with longer life

Setup

Let Cons[t] be consumption in period t, where "life" begins at t=1 and continues to t=T. We assume that the utility at time t is u[Cons[t]] and that utility is discounted at β. Therefore, the life-cycle objective is

In[121]:=
```
obj := Sum[β^(t-1) u[Cons_t], {t, 1, T}]
```

R will be the discount factor, which is also one plus the interest rate; hence, a dollar saved at time t will be worth R=1+r at time t+1. Let wage[t] be the labor income in period t.
The present values of consumption expenditures and wage income are

In[122]:=
```
PVc := Sum[Cons_t R^(-t+1), {t, 1, T}]
PVw := Sum[wage_t R^(-t+1), {t, 1, T}]
```

The budget constraint implies that present value of consumption equals (or, is less than) the present value of wage income.

In[124]:=
```
Budget := {PVc ≤ PVw}
```

List the variables

In[125]:=
```
vars := Table[Cons_t, {t, 1, T}]
```

Add lower bound constraints on consumption.

In[126]:=
```
LoBnds := Table[Cons_t ≥ 0.0001, {t, 1, T}]
```

Now we form a new constraint set combining the budget constraint with the positivity constraints.

In[127]:=
```
constraints := Union[Budget, LoBnds]
```

First examples

Exponential utility

Choose the utility function and discount factor.

In[128]:=
```
u[c_] = -Exp[-c]; β = 0.96;
```

Choose the interest rate.

In[129]:=
```
r = 0.10; R = 1 + r;
```

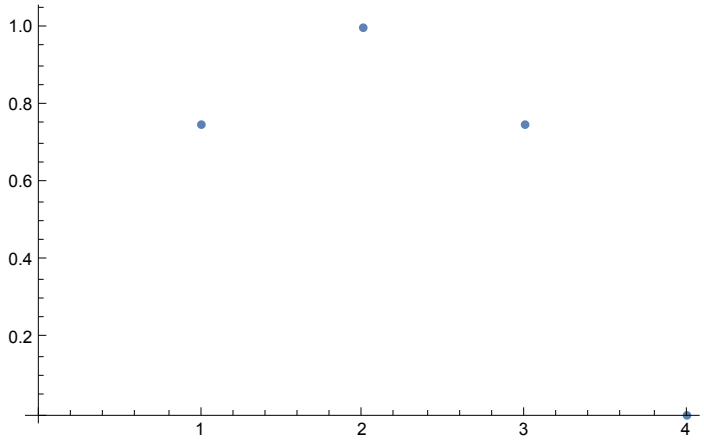
Choose the planning horizon.

In[143]:=
```
T = 4;
```

Specify the income path. We choose a pattern that corresponds to a common (parabolic) pattern of life-cycle earnings,

In[144]:=
```
Do[wage_i = (T - i) i / T, {i, 1, T}]
Table[wage_i, {i, 1, T}];
ListPlot[%]
```

Out[146]=



Display the variables, objective, and constraints:

In[147]:=
```
vars
```

Out[147]=
$$\{\text{Cons}_1, \text{Cons}_2, \text{Cons}_3, \text{Cons}_4\}$$

In[148]:=
```
obj
```

Out[148]=
$$-1.\, e^{-\text{Cons}_1} - 0.96\, e^{-\text{Cons}_2} - 0.9216\, e^{-\text{Cons}_3} - 0.884736\, e^{-\text{Cons}_4}$$

In[149]:=
```
constraints // TableForm
```

Out[149]//TableForm=

$$\text{Cons}_1 \geq 0.0001$$
$$\text{Cons}_2 \geq 0.0001$$
$$\text{Cons}_3 \geq 0.0001$$
$$\text{Cons}_4 \geq 0.0001$$
$$1.\, \text{Cons}_1 + 0.909091\, \text{Cons}_2 + 0.826446\, \text{Cons}_3 + 0.751315\, \text{Cons}_4 \leq 2.27893$$

Call the optimizer

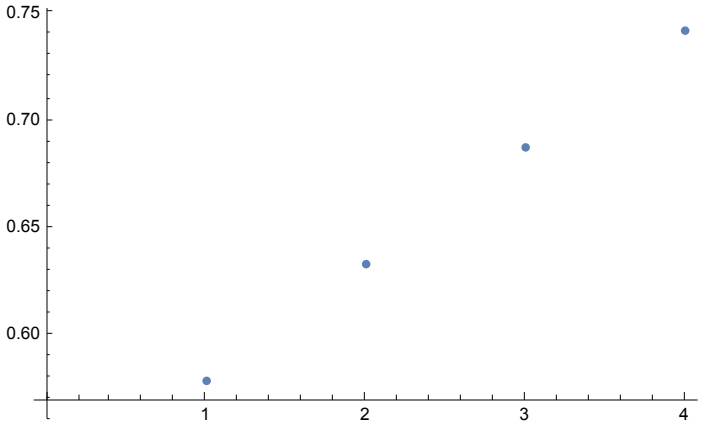
In[150]:=
```
sol = FindMaximum[{obj, constraints}, vars]
```

Out[150]=
$$\{-1.95557, \{\text{Cons}_1 \to 0.578319, \text{Cons}_2 \to 0.632808, \text{Cons}_3 \to 0.687296, \text{Cons}_4 \to 0.741784\}\}$$

Extract the consumption path solution and plot it:

```
Table[Cons_i, {i, 1, T}] /. sol[[2]]
ListPlot[%]
```

Out[152]=



Square root utility

Change the utility function.

In[153]:=
```
u[c_] = Sqrt[c];
```

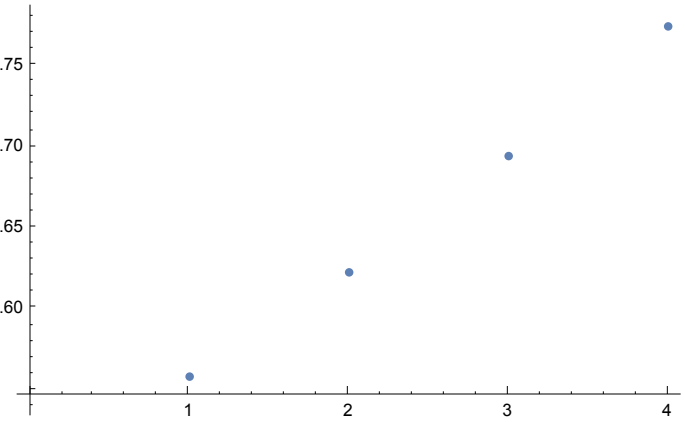
Call the optimizer

In[154]:=
```
sol = FindMaximum[{obj, constraints}, vars]
```

Out[154]= $\{3.05051, \{Cons_1 \to 0.558106, Cons_2 \to 0.622364, Cons_3 \to 0.694021, Cons_4 \to 0.773927\}\}$

In[155]:=
```
Table[Cons_i, {i, 1, T}] /. sol[[2]];
ListPlot[%]
```

Out[156]=

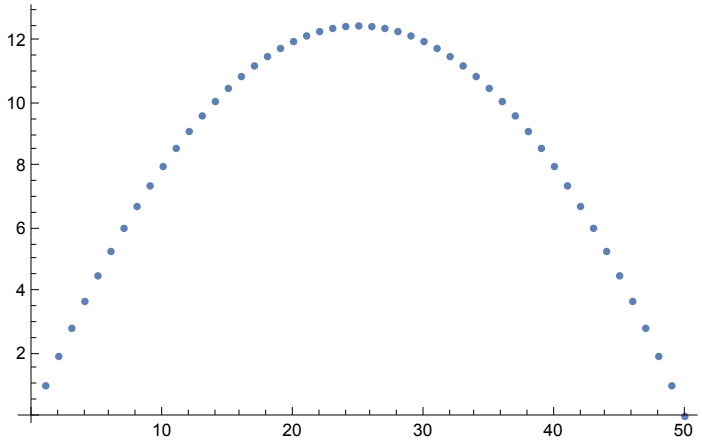


$-c^{-3}$ utility, longer horizon

Change the utility function and horizon, adjusting wage rates accordingly.

```
T = 50;
Do[wage_i = (T - i) i / T, {i, 1, T}]
Table[wage_i, {i, 1, T}];
ListPlot[%]
```

Out[160]=



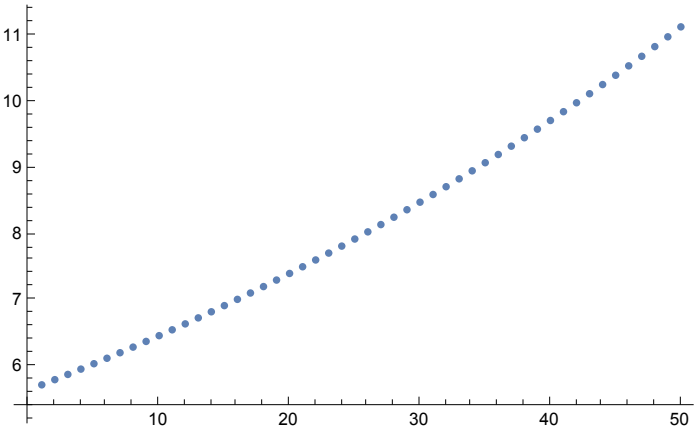
In[161]:=
```
u[c_] = -c^-3;
```

Call the optimizer

In[162]:=
```
sol = FindMaximum[{obj, constraints}, vars];
```

In[163]:=
```
Table[Cons_i, {i, 1, T}] /. sol[[2]];
ListPlot[%]
```

Out[164]=

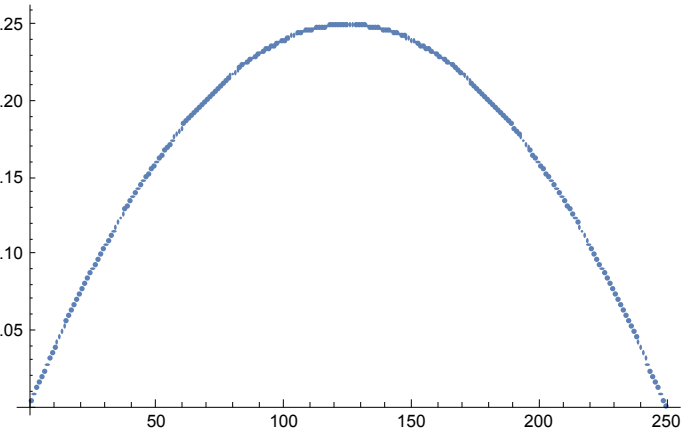


T=250, quarterly calibration

Set T to be 250 and treat time period as a quarter

```
T = 250;
Do[wage_i = (T - i) i / T^2, {i, 1, T}]
Table[wage_i, {i, 1, T}] // ListPlot
```

Out[167]=



Choose an interest rate and a discount factor

In[168]:=
```
r = 0.01; R = 1 + r; β = 0.991;
```

Specify a utility function

In[169]:=
```
u[c_] = -Exp[-c];
```

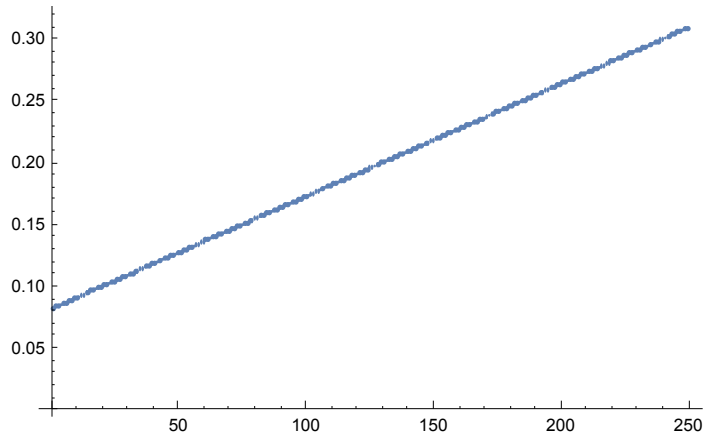
Solve and plot the solution

In[170]:=
```
sol = FindMaximum[{obj, constraints}, vars]; // Timing
```

Out[170]=
```
{0.825668, Null}
```

In[171]:=
```
ListPlot[vars /. sol[[2]]]
```

Out[171]=



Life-Cycle Consumption with long life - examples of algorithm failure

In[335]:=
```
x = 0; Remove["Global`*"]
```

Setup

In[336]:=
```
obj := Sum[β^(t-1) u[Cons[t]], {t, 1, T}]
PVc := Sum[Cons[t] R^(-t+1), {t, 1, T}]
PVw := Sum[wage[t] R^(-t+1), {t, 1, T}]
Budget := {PVc ≤ PVw}
LoBnds := Table[Cons[t] ≥ 0.0001, {t, 1, T}]
constraints := Union[Budget, LoBnds]
vars := Table[Cons[t], {t, 1, T}]
```

Examples of failure

Exponential utility

Choose the utility function and discount factor.

In[343]:=
```
u[c_] = -Exp[-c]; β = 0.9;
```

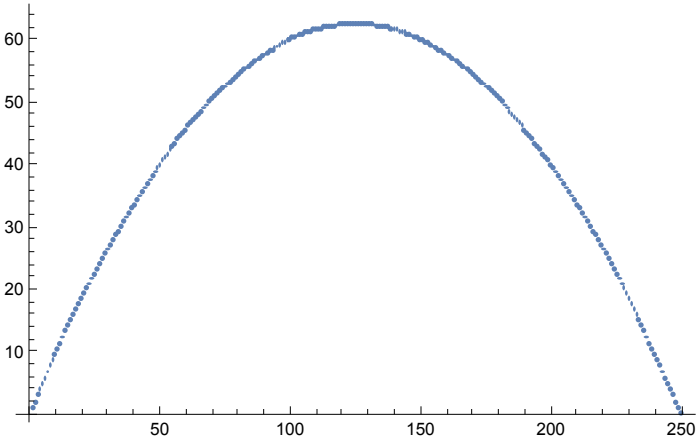
Choose the interest rate and planning horizon.

In[344]:=
```
r = 0.15; R = 1 + r; T = 250;
```

Specify the income path. We choose a pattern that corresponds to a common (parabolic) pattern of life-cycle earnings,

In[345]:=
```
Do[wage[i] = (T - i) i / T, {i, 1, T}]
ListPlot[Table[wage[i], {i, 1, T}]]
```

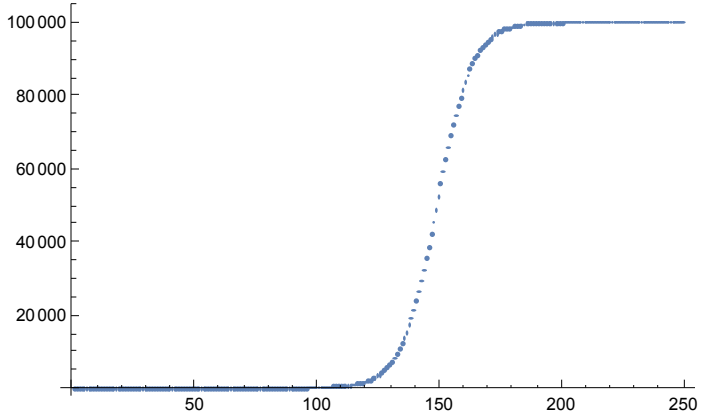
Out[346]=



Call the optimizer, extract the consumption path solution, and plot it

```
sol = FindMaximum[{obj, constraints}, vars];
csol = Table[Cons[i], {i, 1, T}] /. sol[[2]];
ListPlot[csol]
```

Out[•]=



In[•]:= β^T

Out[•]= 3.63603×10^{-12}

Square root utility

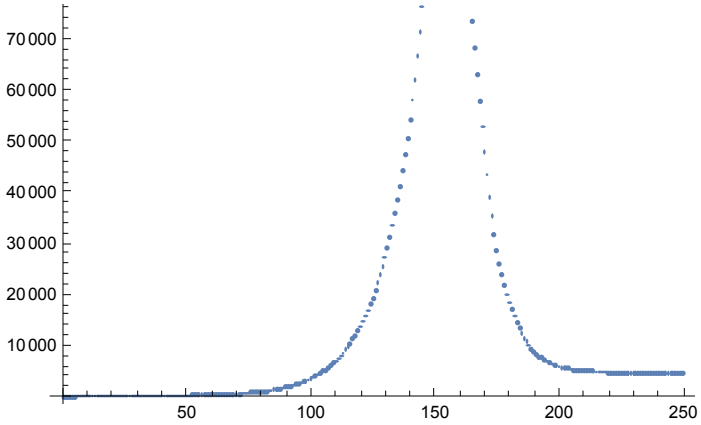
Change the utility function.

In[•]:=
```
u[c_] = Sqrt[c];
```

Call the optimizer, and plot the solution

```
sol = FindMaximum[{obj, constraints}, vars];
csol = Table[Cons[i], {i, 1, T}] /. sol[[2]]; ListPlot[csol]
```

Out[•]=



Life-Cycle Consumption with long life - how to fix the failures

In[347]:=
```
x = 0; Remove["Global`*"]
```

Setup

In[348]:=
```
obj := Sum[β^t u[Cons[t]], {t, 1, T}]
vars := Table[Cons[t], {t, 1, T}]
PVc := Sum[Cons[t] R^(-t+1), {t, 1, T}]
PVw := Sum[wage[t] R^(-t+1), {t, 1, T}]
Budget := {PVc ≤ PVw}
LoBnds := Table[Cons[t] ≥ 0.0001, {t, 1, T}]
```

Here is the new step: express the Euler equations, one for each time t:

In[354]:=
```
euler := Table[u'[Cons[t]] == β R u'[Cons[t + 1]], {t, 1, T - 1}]
```

We add the list euler to the constraints:

In[355]:=
```
constraints := Union[Budget, LoBnds, euler]
```

This problem has T unknown consumptions and T equality constraints: the budget constraint and T-1 Euler equations. So, the number of shadow prices equals for the equality constraints the number of unknowns. As long as there are no active lower bound constraints (which there should not be), there is a good chance that LICQ will hold.

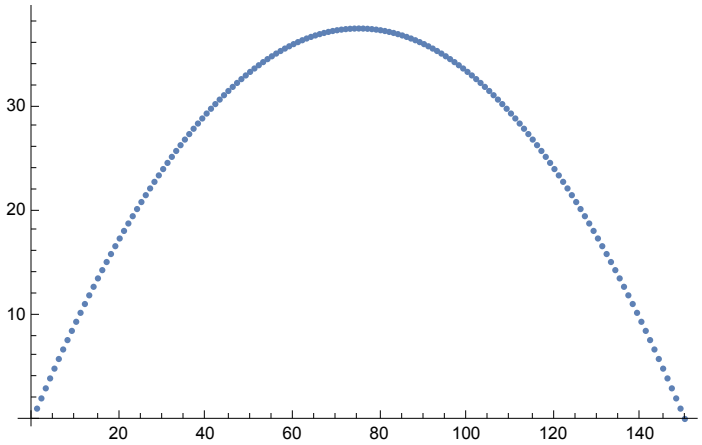
Exponential utility

In[356]:=
```
u[c_] = -Exp[-c]; β = 0.9;
r = 0.15; R = 1 + r; T = 150;
β^T
```

Out[358]= 1.36891×10^{-7}

In[359]:=
```
Do[wage[i] = (T - i) i / T, {i, 1, T}]
tab = Table[wage[i], {i, 1, T}]; ListPlot[tab]
```

Out[360]=

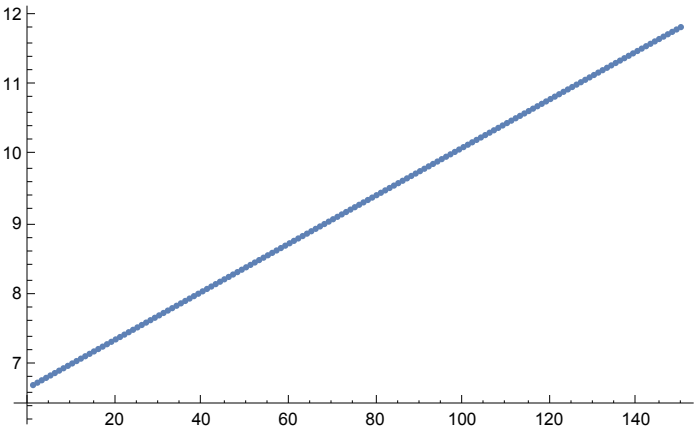


Call the optimizer

In[361]:=
```
sol = FindMaximum[{obj, constraints}, vars]; // Timing
csol = Table[Cons[i], {i, 1, T}] /. sol[[2]]; ListPlot[csol]
```

Out[361]= {0.84729, Null}

Out[362]=



This now works. The Euler equation constraints have pushed the solver to find the true solution.

Make objective degenerate

The approach of adding first-order conditions to the problem is a bit strange. It is better to keep the Euler equations and get rid of the objective function.

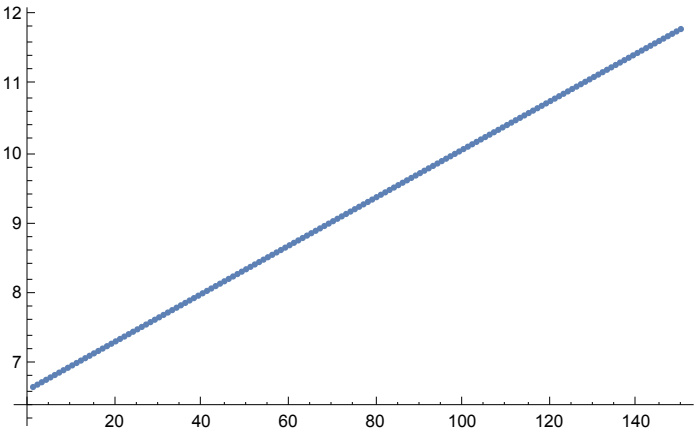
In[363]:=
```
obj = 1;
sol = FindMaximum[{obj, constraints}, vars]; // Timing
```

Out[364]=
```
{12.9876, Null}
```

Extract and plot the consumption path solution

In[365]:=
```
csol = Table[Cons[i], {i, 1, T}] /. sol[[2]]; ListPlot[csol]
```

Out[365]=



Square root utility

Even when we add the Euler equations, other things may go wrong.

In[366]:=
```
T = 150;
```

In[367]:=
```
u[c_] = Sqrt[c];
```

In[368]:=
```
euler := Table[u'[Cons[t]] == β R u'[Cons[t + 1]], {t, 1, T - 1}]
```

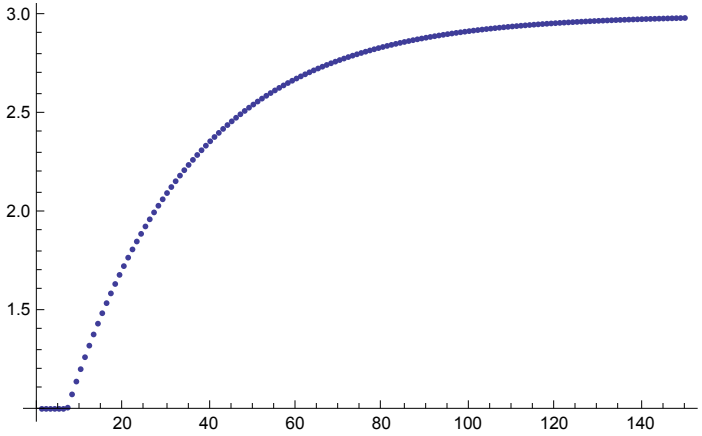
```
sol = FindMaximum[{obj, constraints}, vars]; // Timing
```

FindMaximum::eit : The algorithm does not converge to the tolerance of 4.806217383937354`*^-6 in 500 iterations. The best estimated solution, with feasibility residual, KKT residual, or complementary residual of {49.2337, 1.38494, 2.92027}, is returned. >>

```
{111.61, Null}
```

Convergence was slow. The iteration limit was 500, and the result was not good.

```
csol = Table[Cons[i], {i, 1, T}] /. sol[[2]]; ListPlot[csol]
```



NOTE: The sol=... and Table... commands have been disabled. The most recent version of Mathematica successfully solves this problem.

A fix for the square root utility failure

Change the Euler equation constraint by nonlinear transformation.

If u[c] is the square root function, then the inverse of the u'[c] function is $1/(4\,u'[c]^2)$

```
In[•]:= u[c_] = Sqrt[c];
        upinv[up_] = 1 / (4 up^2);
```

Apply $(u')^{-1}$ to both sides of the Euler equation.

```
In[•]:= euler := Table[Cons[t] == upinv[β R u'[Cons[t + 1]]], {t, 1, T - 1}]
```

List the first three transformed Euler equations

```
In[•]:= euler[[{1, 2, 3}]]
Out[•]= {Cons[1] == 0.933511 Cons[2], Cons[2] == 0.933511 Cons[3], Cons[3] == 0.933511 Cons[4]}
```

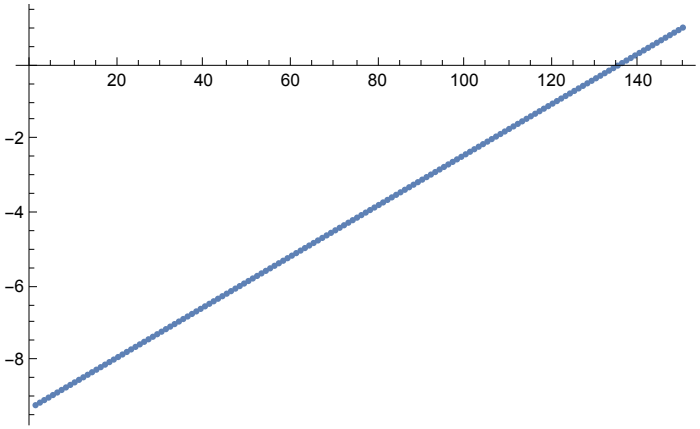
These are now linear constraints. We won't always be able to transform nonlinear constraints to linear constraints but the general strategy is to make each constraint less nonlinear w.r.t. the variables.

```
In[•]:= sol = FindMaximum[{obj, constraints}, vars]; // Timing
Out[•]= {0.016058, Null}
```

The next plot shows that we have the true solution.

```
csol = Table[Log[Cons[i]], {i, 1, T}] /. sol[[2]]; ListPlot[csol]
```

Out[ ]=



Lessons

In optimization problems, make sure that the additive terms are of comparable magnitude.
When solving constrained optimization problems, make the constraints as linear as possible. Make the equations linear via transformations, or put nonlinearity in objective function