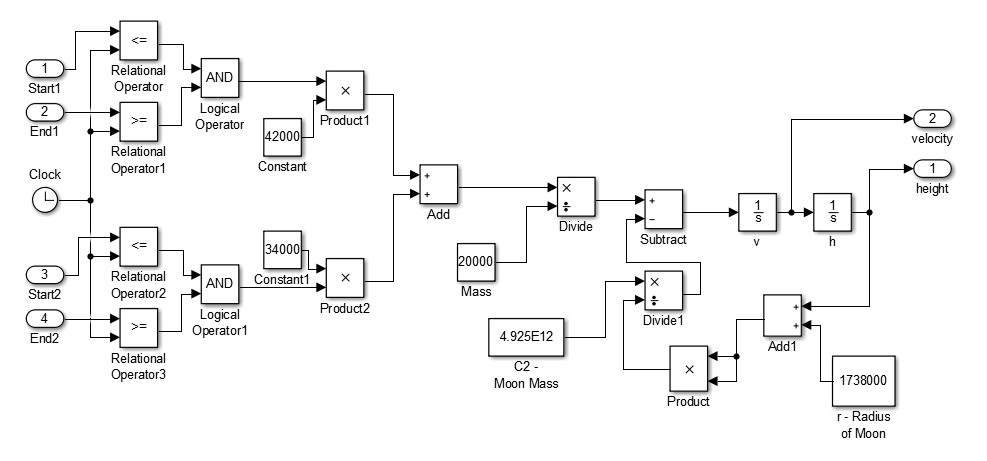
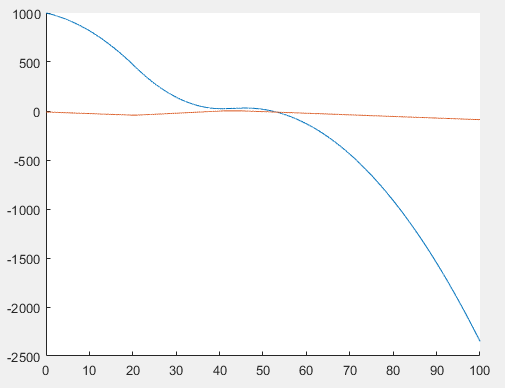
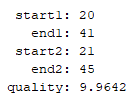
# Aufgabe 1

## a)



Durch Ausprobieren findet sich zum Beispiel folgende Lösung:  
(blau = Höhe, orange = Geschwindigkeit)





Sie schlägt also mit 9.9 m/s auf.

## b)

Einschläge bei unterschiedlichen step sizes und solvern:

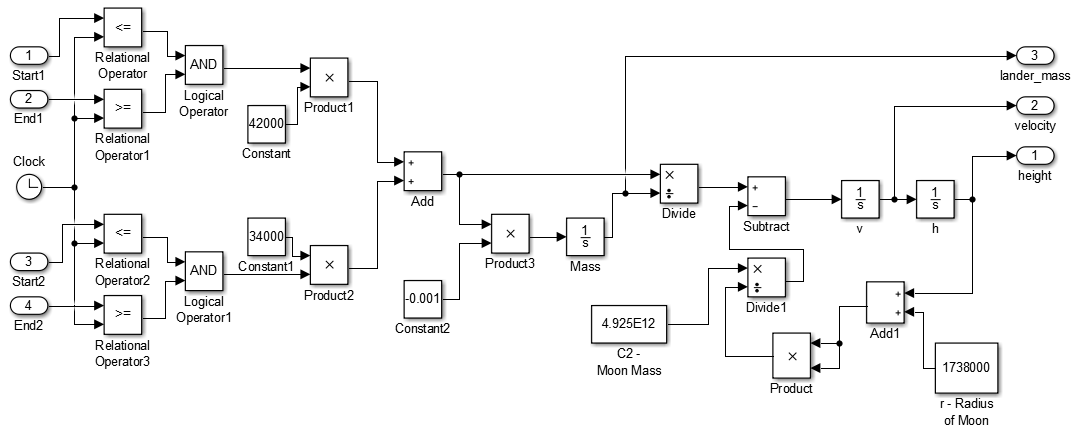
|  |  |  |  |
| --- | --- | --- | --- |
| **Step size** | **Euler (ode1)** | **Ode4-Solver** | **Ode8-Solver** |
| **0.01** | t: 52.10  v: -9.955  h: -0.097 | t: 52.09  v: -9.964  h: -0.091 | t: 52.06  v: -9.926  h: -0.023 |
| **0.05** | t: 52.30  v: -10.064  h: -0.273 | t: 52.25  v: -10.109  h: -0.255 | t: 52.15  v: -9.999  h: -0.405 |
| **0.25** | t: 53.25  v: -10.526  h: -0.559 | t: 53  v: -10.752  h: -0.707 | t: 52.50  v: -10.203  h: -1.335 |

Die Ergebnisse variieren leicht je nachdem, welche Integrationsmethode und step size verwendet wird. Der Einschlag wird bei größeren step sizes erst später registriert, außerdem wird bei größerer step size der Unterschied zwischen den Methoden größer.

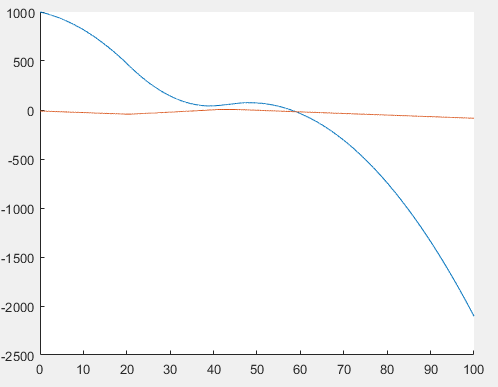
Die Integrationsmethoden sind alle nur Annäherungen. Der Fehler wird bei kleinerer Schrittgröße ebenfalls kleiner.

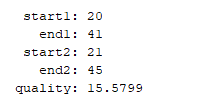
## c)

Modell mit Verringerung der Masse



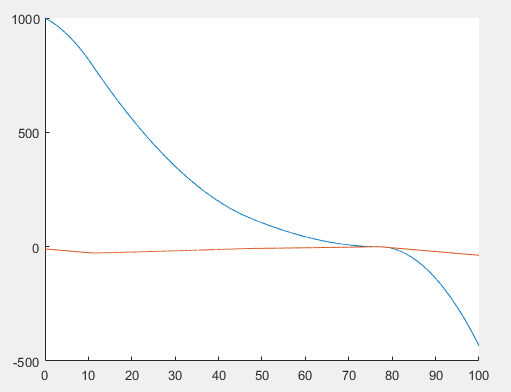
Selber Test wie bei a) läuft nun etwas anders:  
(blau = Höhe, orange = Geschwindigkeit)

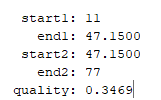




Der Lander schlägt nun mit 15.58 m/s auf.

Mit ein paar weiteren Experimenten wird die Einschlaggeschwindigkeit auf 0.35 m/s reduziert.  
(blau = Höhe, orange = Geschwindigkeit)





## Aufgabe 2

## a)

function [ sol ] = init( )

sol.start1 = rand \* 100;

sol.end1 = rand \* 100;

sol.start2 = rand \* 100;

sol.end2 = rand \* 100;

if sol.start1 > sol.end1

[sol.end1, sol.start1] = deal(sol.start1, sol.end1);

end

if sol.start2 > sol.end2

[sol.end2, sol.start2] = deal(sol.start2, sol.end2);

end

sol.quality = [];

sol.height = [];

sol.t = [];

sol.velocity = [];

sol.landed = [];

end

function mutated = mutate( parent, sigma )

mutated.start1 = parent.start1 + sigma \* randn;

mutated.end1 = parent.end1 + sigma \* randn;

mutated.start2 = parent.start2 + sigma \* randn;

mutated.end2 = parent.end2 + sigma \* randn;

if mutated.start1 > mutated.end1

[mutated.end1, mutated.start1] = deal(mutated.start1, mutated.end1);

end

if mutated.start2 > mutated.end2

[mutated.end2, mutated.start2] = deal(mutated.start2, mutated.end2);

end

mutated.quality = [];

mutated.height = [];

mutated.velocity = [];

end

function [ solution ] = evaluate( solution, stepSize )

model = 'moonlander\_parametrized';

notLandedPenalty = 1;

simParams = simget(model);

simParams = simset(simParams, 'FixedStep', stepSize);

modelParams = [1, solution.start1, solution.end1, ...

solution.start2, solution.end2];

[T, ~, Y] = sim(model, 100, simParams, modelParams);

solution.height = Y(:, 1);

solution.t = T;

solution.velocity = Y(:, 2);

impactIndex = find(solution.height < 0, 1);

if isempty(impactIndex)

solution.landed = false;

[minHeight, minIndex] = min(solution.height);

% subtract velocity at min height to discourage algorithm from

% crashing with high velocity right after the simulation ends

solution.quality = minHeight + notLandedPenalty - ...

solution.velocity(minIndex);

else

solution.landed = true;

solution.quality = -solution.velocity(impactIndex);

end

end

function [ bestSolution, qualityProgress ] = optimize( mu, lambda, ...

sigma, maxGen, useParents, sigmaMult, stepSize )

for i = 1:mu

s = init();

s = evaluate(s, stepSize);

population(i) = s;

end

bestSolution = population(1);

for gen = 1:maxGen

nr\_success = 0;

% create new population

parfor j = 1:lambda

% select random parent

parent = population(randi(length(population)));

offspring = mutate(parent, sigma);

% evaluate

offspring = evaluate(offspring, stepSize);

if offspring.quality < parent.quality

nr\_success = nr\_success + 1;

end

newPopulation(j) = offspring;

end

if useParents

% add population to newPopulation

for i = lambda + 1: lambda + mu

newPopulation(i) = population(i - lambda);

end

end

% generational replacement TODO

% sort new population

[~, idx] = sort([newPopulation.quality]);

newPopulation = newPopulation(idx);

if newPopulation(1).quality < bestSolution.quality

bestSolution = newPopulation(1);

end

% alles speichern!

% Verlauf von quality, sigma, ...

qualityProgress(gen) = newPopulation(1).quality;

bestProgress(gen) = bestSolution.quality;

sigmaProgress(gen) = sigma;

% setup for next generation

for j = 1:mu

population(j) = newPopulation(j);

end

% sigma control (1/5 Erfolgsregel)

if nr\_success > 0.2 \* lambda

sigma = sigma \* sigmaMult;

else

sigma = sigma / sigmaMult;

end

end

figure;

hold on;

gens = 1:gen;

plot(gens, qualityProgress, '+r');

plot(gens, bestProgress, '-');

plot(gens, sigmaProgress, ':');

xlabel('Generation');

legend('Best quality for generation', 'Best quality overall', 'sigma');

hold off;

end

## b)