MC-Simulation for pathes in Heston's stochastic volatility model

With Version 10 Maple uses a much faster way to generate random variates through the Mersenne Twister. Together with hardware floatings for computations this gives better speed (note that using 'Compile' this can even improved more, not done here).

```
> restart;
> with(plots):
> with(stats):
  RandomTools[MersenneTwister](SetState()):
> HMC:=proc(
    mu,     # drift = rates
    sigma,  # vol of vol
    kappa,  # mean reversion
    theta,  # long run variance
    rho,    # correlation
    S0,     # start values
    V0,     # instant variance
    nSteps::posint)
  local _mu, _sigma, _kappa, _theta, _rho, _S0, _V0, _nSteps,
    st, B,W, result, _B, _W;

  # generate 2 random vectors (and check time used)
  st:=time():
  #RandomTools[MersenneTwister](SetState());
  B:= Statistics:-Sample(
         Statistics:-RandomVariable(Normal(0,1)),nSteps+1);
  W:= Statistics:-Sample(
         Statistics:-RandomVariable(Normal(0,1)),nSteps+1);
  print(`time used for random generation [seconds]`=time()-st);

  # build path (and check time used)
  st:=time():

  _mu:=mu; _sigma:=sigma; _kappa:=kappa; _theta:=theta; _rho:=rho;
  _S0:=S0; _V0:=V0; _nSteps:=nSteps; _B:=B; _W:=W;

  result:=evalhf(
    proc(mu, sigma, kappa, theta, rho, S0, V0, nSteps, B,W)
    local dt, v_t, S, Z, k;

    dt:=1/252.0:
    S:=hfarray(1..nSteps):
    Z:=hfarray(1..nSteps):
    S[1]:=S0:
    v_t:=V0:

    Z:=rho*B+sqrt(1-rho^2)*W;
    for k from 1 to nSteps-1 do
      #Z[k]:= rho*B[k]+sqrt(1-rho^2)*W[k];
      S[k+1]:= (S[k] + mu*S[k]*dt + sigma*sqrt(v_t*dt) *S[k]*B[k]):
      v_t:=abs( (v_t+kappa*(theta-v_t)*dt+sigma*sqrt(v_t*dt)*Z[k]));
    end do:
    return S;
    end proc(_mu, _sigma, _kappa, _theta, _rho, _S0, _V0, _nSteps, _B, _W)
    );

  print(`time used for building path [seconds]`=time()-st);

  return result;
  end proc: #maplemint(%);
```

```
> mu:=   0.025; # drift = rates
  sigma:=0.40;  # vol of vol
  kappa:=2.00;  # mean reversion
  theta:=0.04;  # long run variance
  rho:= -0.50;  # correlation
  #mu:=0.025; sigma:=0.6; kappa:=0.24; theta:=0.02;rho:= -0.64;
  ``;
  S0:=100.0;    # start values
  V0:= 0.04;    # instant variance
```

$$\mu := 0.025$$

$$\sigma := 0.40$$

$$\kappa := 2.00$$

$$\theta := 0.04$$

$$\rho := -0.50$$

$$S0 := 100.0$$

$$V0 := 0.04$$

```
> nTst:=5000;
  st:=time():
  listMC:=HMC(mu, sigma, kappa, theta, rho, S0, V0,nTst):
  `total time used [seconds]`=time()-st;
  listplot(listMC);
```

$$nTst := 5000$$

time used for random generation [seconds] = 0.010

time used for building path [seconds] = 0.054

total time used [seconds] = 0.064

As already said the time for building the path (which here is done using internal hardware floating by 'evalhf') can be heavily reduced even more by automatical compiling the code.

Moreover it is no problem to use an external runtime library which would provide normal variates directly (having encoded Marsaglia's Ziggurat for example).

For this one would separate the above in generating the variates and building the path.

Provide several pathes in one picture:

```
> nTst:=1000;
  pathesWanted:=8; P:='P':
  st:=time():
  for i from 1 to pathesWanted do
    listMC:=HMC(mu, sigma, kappa, theta, rho, S0, V0,1000):
    P[i]:=listplot(listMC, color=COLOR(HUE, i/pathesWanted)):
    #color=COLOR(RGB,rand()/10^12,rand()/10^12,rand()/10^12)):
```

```
end do:
`total time used [seconds]`=time()-st;
display({seq(P[i],i=1..pathesWanted)});
```

nTst := 1000

pathesWanted := 8

total time used [seconds] = 0.596