Square Wave ODE Example 1 (State Space Solver):

This example is meant to show how to use a square wave to drive a mass, spring, damper system. I wanted to use the state space solver for this. However, towards the end, I found a big problem with the state space solver. It seems to output velocity. However, you can't calculate the velocity it comes up with on your own, using the time and displacement output. This makes trying to compute the mass kinetic energy and acceleration very questionable. Once I ran into this problem, I created a similar example using odesolve. This removes the uncertainty that comes with the state space solver. Also, the state space solver uses Force/mass as the amplitude of the forcing function. This causes additional confusion later on.

Second Order ODE that describes the physics:

$$m\cdot\frac{d^2}{d\tau^2}x(\tau) + b\cdot\frac{d}{d\tau}x(\tau) + k\cdot x(\tau) = F_0\cdot\cos(\omega_F\cdot\tau)$$

Note; $A_0 = \frac{F_0}{m}$ $\left(\omega_0\right)^2 = \frac{k}{m}$

$\zeta := .01$ Damping Ratio

Defining Inputs:

$m := 60$ $k := 1\cdot10^1$ $\omega_0 := \sqrt{\frac{k}{m}} = 0.408248$

When the damping is low and the forcing function oscillates at the natural frequency, resonance will occur.

$\omega_f := \omega_0\cdot1.0$ $F_0 := 1$ $init := \begin{pmatrix}0\\0\end{pmatrix}$ $t1 := 0$ $b := \zeta\cdot\left(2\cdot m\cdot\omega_0\right)$ $b = 0.489898$

$n := 10$ Number of Cycles $t2 := n\cdot\left(\frac{2\cdot\pi}{\omega_f}\right)$

$npoints := 100\cdot n = 1\times10^3$ $intvls := npoints - 1 = 999$

$Fs := \frac{\omega_f}{2\cdot\pi} = 0.064975$ $tmax := t2 = 153.90598$

$A_0 := \frac{F_0}{m} = 0.016667$ The state space solution uses this value as the forcing function magnitude

$tstep := \frac{tmax}{intvls} = 0.15406$ $d := 10\%$ Duty Cycle $A := A_0 = 0.016667$

Solving the second order ODE using the Mathcad State Space ODE Solver:

$\frac{d}{d\tau}x(\tau) = A(\tau)\cdot x(\tau) + B(\tau)\cdot u(\tau)$ State space representation of the second order ODE from above.

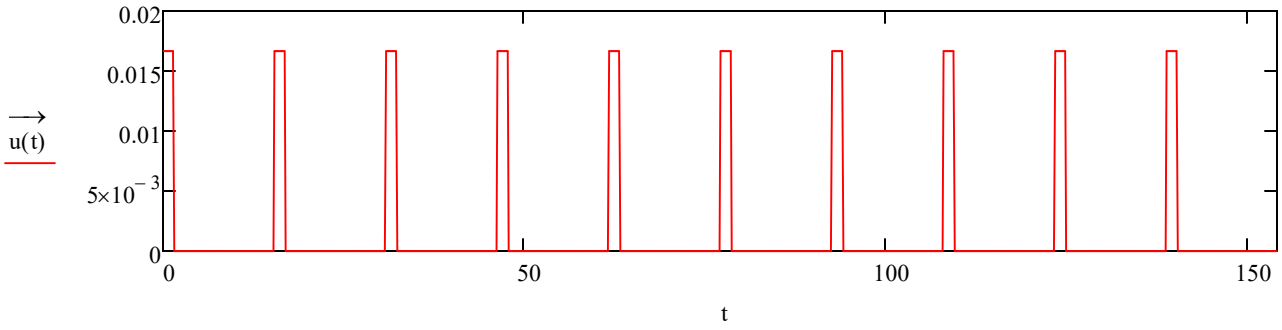
$u1(\tau) := A_0\cdot\cos(\omega_f\,\tau)$ Note; the forcing function amplitude is divided by mass

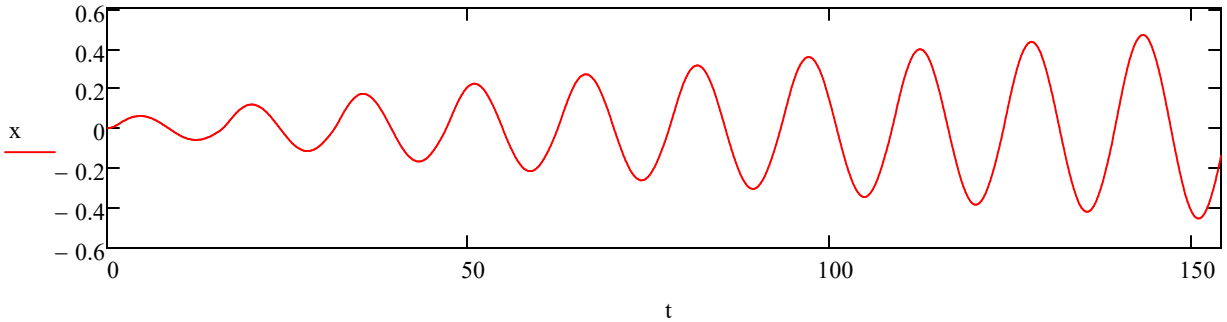
$u2(\tau) := A\cdot\left(\mathrm{mod}\left(\tau,\frac{1}{Fs}\right)\cdot Fs \le d\right)\cdot(\tau < tmax)$ Note; the forcing function amplitude is divided by mass

$$A(\tau) := \begin{bmatrix} 0 & 1 \\ -(\omega_0)^2 & \frac{-b}{m} \end{bmatrix} \qquad B(\tau) := \begin{pmatrix} 0 \\ 1 \end{pmatrix}$$

$u(\tau) := u2(\tau)$ Enter u1 or u2 as the forcing function

$sol := statespace(init, t1, t2, intvls, A, B, u)$ $t := sol^{\langle 0\rangle}$ $x := sol^{\langle 1\rangle}$ $v := sol^{\langle 2\rangle}$





Note; mass is in the equations below, because the forcing function amplitude was divided by mass.

$$\frac{\max(x)}{\left(\frac{F_0}{k}\right)} = 4.660072$$ Amplification Factor

When the damping ratio is .01, classical solutions state that the above should equal 50. Moreover, the dynamic displacement should be 50x the static spring displacement. The above is used to serve as a check. Notice what happens when you switch to a square wave. Also note what happens when you change the duty cycle of the square wave. You will find that classical solutions come with a lot of 'ands, ifs, and buts'. Therefore, it's better to just solve the ODE rather than use classical simplifications. Solving the ODE was very difficult in the old days 'pre-computer'. Therefore, classical solutions gained wide spread use.

Note how the number of cycles affects this as well. You need about 300 cycles to get to the classical solution. This makes viewing the wave forms impossible. So you can switch to something like 5 cycles, to see the waves.

$i := 1 .. intvls - 1$ $intvls = 999$

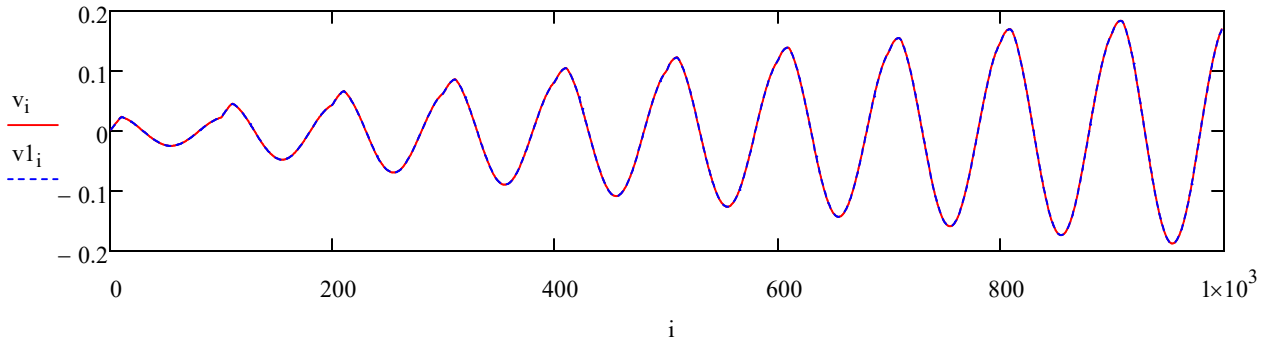
$\Delta t_i := t_{i+1} - t_{i-1}$ $\Delta x_i := x_{i+1} - x_{i-1}$

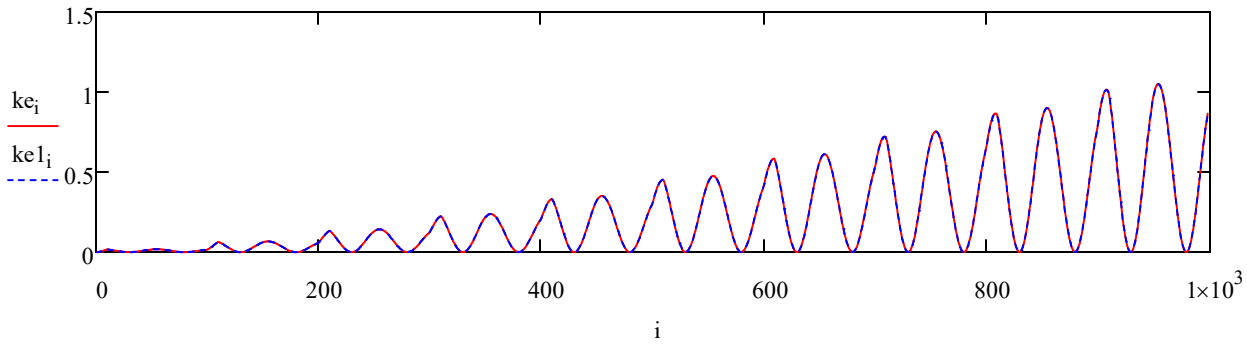
$v1_i := \frac{\Delta x_i}{\Delta t_i}$ Checking the mass velocity

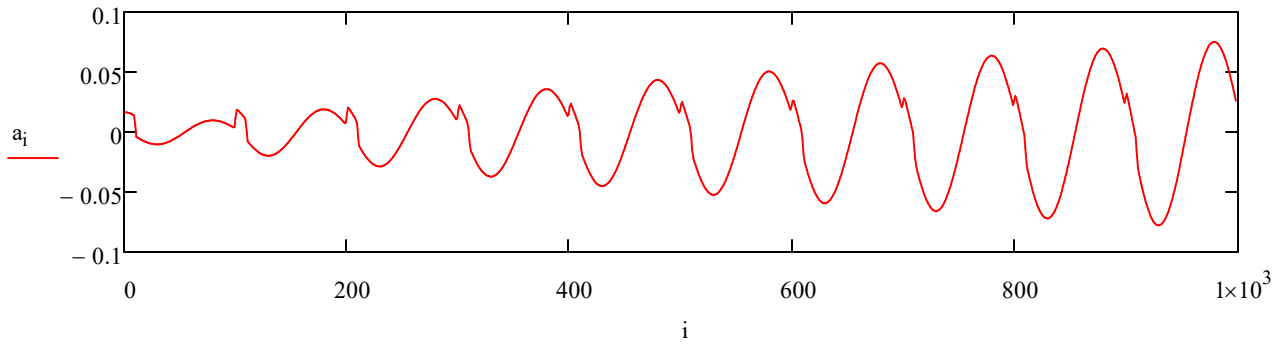
$ke_i := .5 \cdot m \cdot \left(v_i\right)^2$ Calculating the mass kinetic energy

$ke1_i := .5 \cdot m \cdot \left(v1_i\right)^2$ Checking the mass kinetic energy

$a_i := \frac{v_{i+1} - v_{i-1}}{\Delta t_i}$ Calculating the mass acceleration







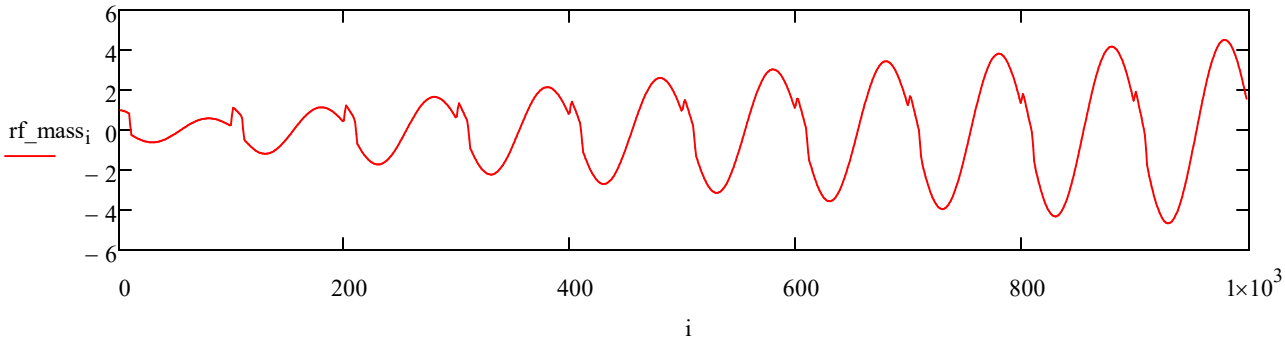
Reaction Forces:

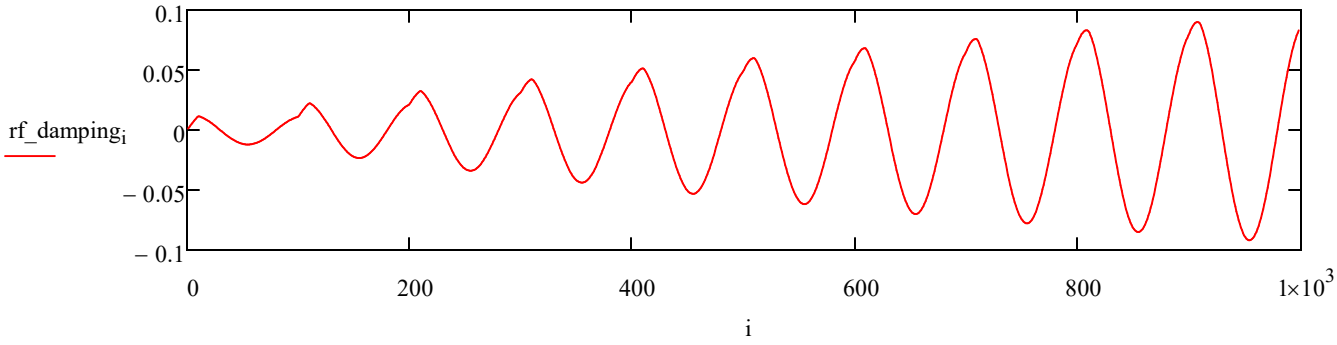
$rf\_mass_i := m \cdot a_i$ $rf\_damping_i := b \cdot v1_i$ $rf\_spring_i := k \cdot x_i$

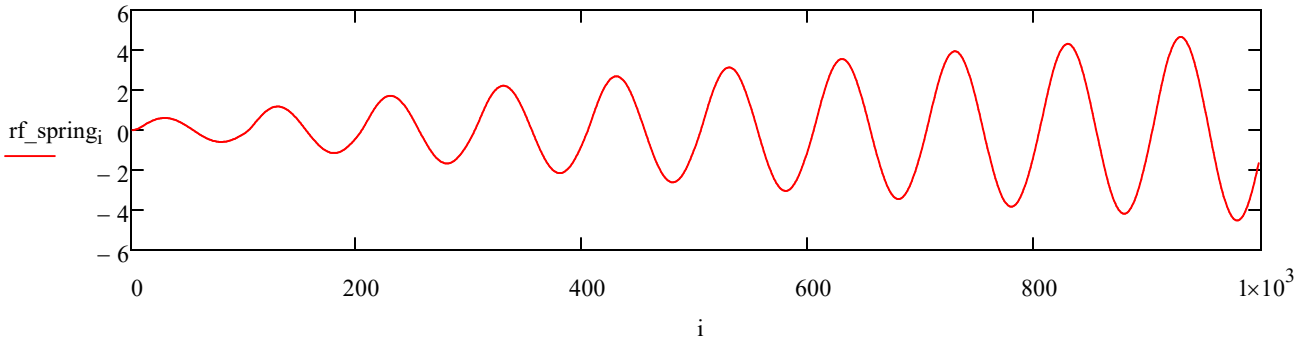
$rf\_total_i := rf\_mass_i + rf\_damping_i + rf\_spring_i$

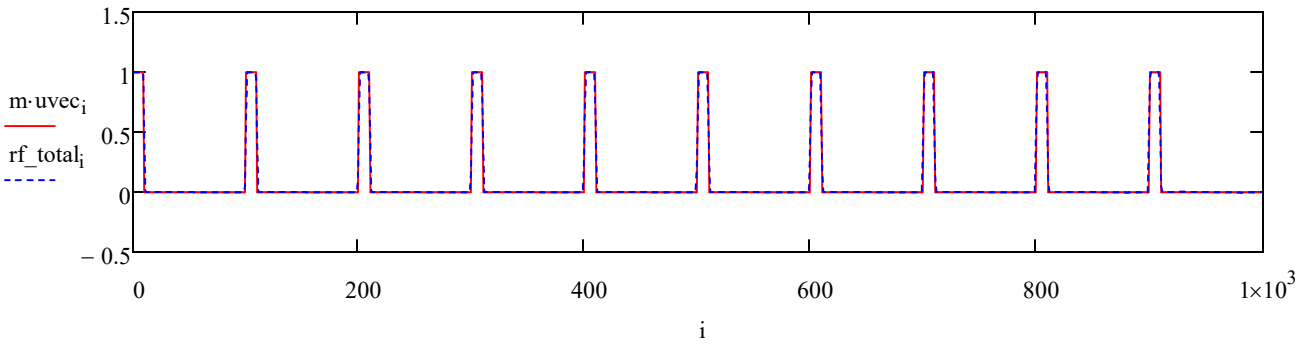
$uvec_i := u(t_i)$

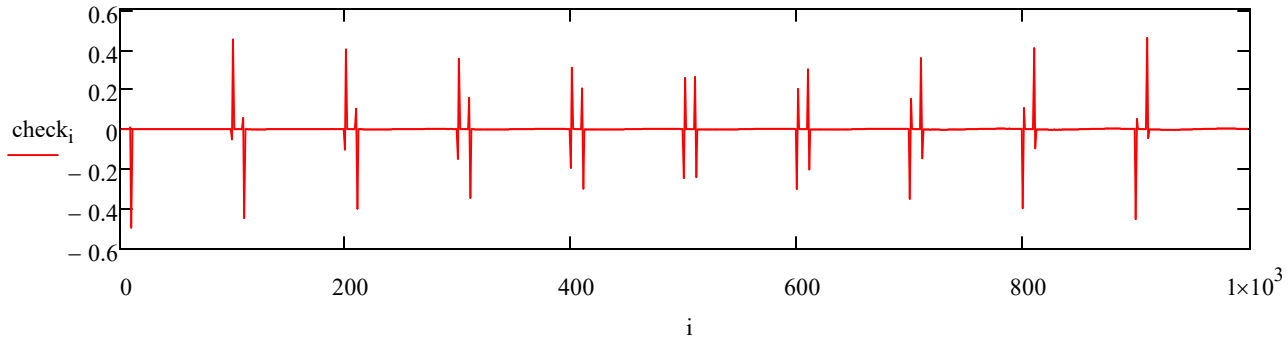
$check_i := m \cdot uvec_i - rf\_total_i$ The check should equal zero, due to Newton's third law


rf_mass_i
6
4
2
0
– 2
– 4
– 6
0
200
400
600
800
1×10^3
i


rf_damping_i
0.1
0.05
0
– 0.05
– 0.1
0
200
400
600
800
1×10^3
i


rf_spring_i
6
4
2
0
– 2
– 4
– 6
0
200
400
600
800
1×10^3
i


m·uvec_i
rf_total_i
1.5
1
0.5
0
– 0.5
0
200
400
600
800
1×10^3
i



$$\frac{\max(\text{check})}{F_0} = 45.621806 \cdot \%$$

$$\frac{\min(\text{check})}{F_0} = -49.297219 \cdot \%$$

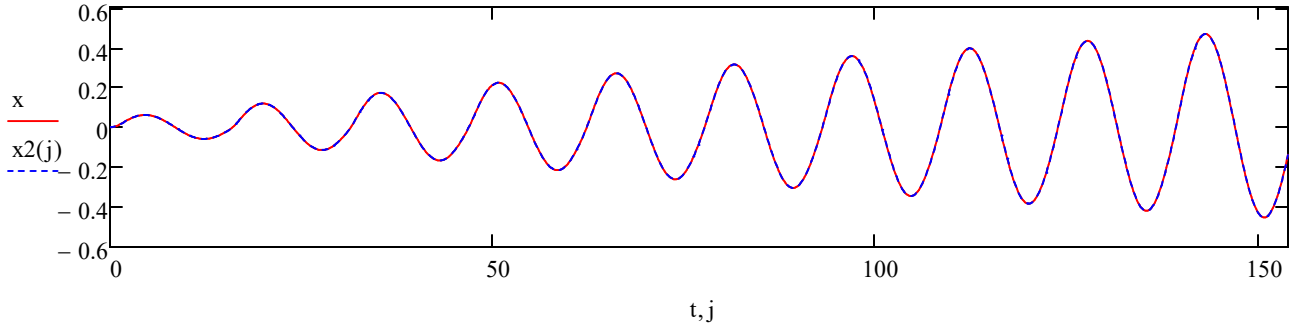
Experimented with converting the matrix to a function. I tried to follow what Mathcad says it does to the odesolve output. I found strange results using the derivative function. It's also a lot slower. So it's best to work with the matrix output and manually calculate velocity, kinetic energy, and acceleration.

$j := 0, tstep .. tmax$

$coef := lspline(t, x)$

$x2(j) := interp(coef, t, x, j)$

$x2(tstep) = 1.976134 \times 10^{-4}$

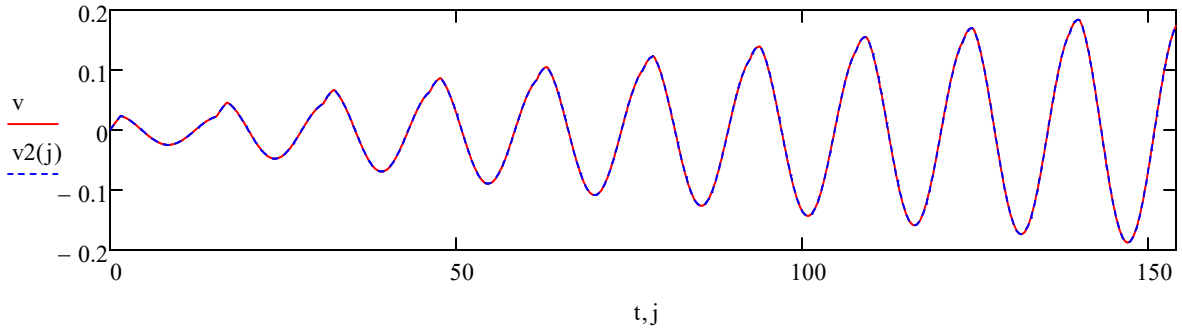


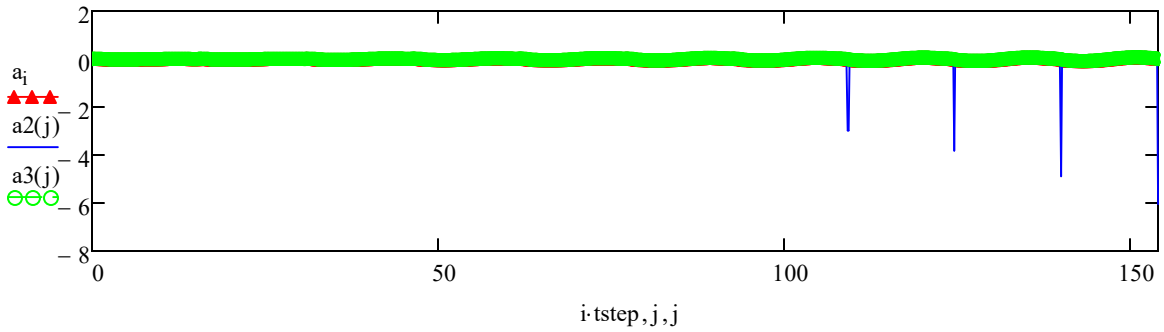
$$v2(j) := \frac{d}{dj} x2(j)$$

$$a2(j) := \frac{d^2}{dj^2} x2(j)$$

Taking the second derivative of displacement produces strange results in this case. In the odesolve example, taking the first derivative of velocity will cause it to hang. You can only take the second derivative of displacement in that example. So the derivative option is slow and problematic.

$$a3(j) := \frac{d}{dj} v2(j)$$





$v_i =$

$2.564363 \cdot 10^{-3}$
$5.115371 \cdot 10^{-3}$
$7.642959 \cdot 10^{-3}$
0.010137
0.012588
0.014987
0.017323
0.019588
0.021772
0.023833
0.023267
0.02261
0.021865
0.021034
0.020122
...

$v1_i =$

$2.561861 \cdot 10^{-3}$
$5.111064 \cdot 10^{-3}$
$7.636968 \cdot 10^{-3}$
0.010129
0.012578
0.014974
0.017309
0.019574
0.021756
0.023188
0.023247
0.022593
0.021858
0.02104
0.020125
...

$ke_i =$

$1.972787 \cdot 10^{-4}$
$7.850105 \cdot 10^{-4}$
$1.752445 \cdot 10^{-3}$
$3.082866 \cdot 10^{-3}$
$4.753894 \cdot 10^{-3}$
$6.737954 \cdot 10^{-3}$
$9.002507 \cdot 10^{-3}$
0.011511
0.014221
0.017041
0.016241
0.015337
0.014342
0.013273
0.012146
...

$ke1_i =$

$1.968939 \cdot 10^{-4}$
$7.836893 \cdot 10^{-4}$
$1.749698 \cdot 10^{-3}$
$3.07812 \cdot 10^{-3}$
$4.746012 \cdot 10^{-3}$
$6.72648 \cdot 10^{-3}$
$8.988409 \cdot 10^{-3}$
0.011494
0.014199
0.01613
0.016213
0.015314
0.014333
0.013281
0.012151
...

x =

	0
0	0
1	$1.976134 \cdot 10^{-4}$
2	$7.893608 \cdot 10^{-4}$
3	$1.772435 \cdot 10^{-3}$
4	$3.142464 \cdot 10^{-3}$
5	$4.893495 \cdot 10^{-3}$
6	$7.017929 \cdot 10^{-3}$
7	$9.507237 \cdot 10^{-3}$
8	0.012351
9	0.015538
10	0.019055
11	0.022683
12	0.026218
13	0.029644
14	0.032953
15	...

x2(j) =

0
$1.976134 \cdot 10^{-4}$
$7.893608 \cdot 10^{-4}$
$1.772435 \cdot 10^{-3}$
$3.142464 \cdot 10^{-3}$
$4.893495 \cdot 10^{-3}$
$7.017929 \cdot 10^{-3}$
$9.507237 \cdot 10^{-3}$
0.012351
0.015538
0.019055
0.022683
0.026218
0.029644
0.032953
...

a =

	0
0	0
1	0.016602
2	0.016483
3	0.016298
4	0.01605
5	0.015739
6	0.015366
7	0.014933
8	0.014441
9	0.013778
10	$4.851097 \cdot 10^{-3}$
11	$-3.96898 \cdot 10^{-3}$
12	$-4.550288 \cdot 10^{-3}$
13	$-5.114843 \cdot 10^{-3}$
14	$-5.658335 \cdot 10^{-3}$
15	...

a2(j) =

0
0.021083
0.015302
0.016635
0.015978
0.015771
0.015335
0.015129
0.013828
0.01624
$4.509643 \cdot 10^{-3}$
$-6.025362 \cdot 10^{-3}$
$-4.046782 \cdot 10^{-3}$
$-5.079439 \cdot 10^{-3}$
$-5.598449 \cdot 10^{-3}$
...

a3(j) =

0
0.021052
0.015318
0.016629
0.01598
0.015763
0.015345
0.015129
0.013973
0.015653
$4.55931 \cdot 10^{-3}$
$-5.550335 \cdot 10^{-3}$
$-4.171575 \cdot 10^{-3}$
$-5.058176 \cdot 10^{-3}$
$-5.605506 \cdot 10^{-3}$
...