Efficient Earthquake Simulation Of Stiff And High DOF Bridge Expansion Joint Models With Python

Michael Tahedl¹, Fredrik Borchsenius², Andreas Taras³

¹ MAURER Engineering GmbH
Frankfurter Ring 193, 80807 Munich, Germany
m.tahedl@maurer.eu

² Faculty of Mechanical Engineering
OTH Regensburg
Galgenbergstr. 13, 93053 Regensburg,
Germany
fredrik.borchsenius@oth-regensburg.de

³Dept. of Civil, Environmental
and Geomatic Engineering
ETH Zürich
Stefano-Franscini-Platz 5, 8093 Zurich,
Switzerland
taras@ibk.baug.ethz.ch

ABSTRACT

Various types of seismic protection devices has been developed to protect structures like bridges from collapse during an earthquake event, such as hydraulic or metallic hysteresis dampers and spherical pendulum bearings. The expansion joints however, which are already included in most large-span bridges, are not considered as an earthquake protection device regardless of the significant friction forces they produce. These friction forces can be seen as damping forces between the shaking environment and the oscillating bridge. To investigate the effect of those damping forces during different earthquake loads, a multibody dynamics simulation model of the expansion joints will be created. This model should be accurate enough to represent the generation of the damping forces and effects of the geometric setup of the expansion joints. Because large expansion joints for large-span bridges are of special interest, the number of degrees of freedom (DOF) becomes very high. Because this models include stiff bushings, implicit solvers need to be used to gain a stable simulation. Expansion joints are almost unique constructions for every specific bridge, which requires a automated model generation. Because of its excellent modules for numerical mathematics, the scripting language Python is used. To create an efficient simulation model, several optimization techniques such as Just-In-Time (JIT) compilation and parallelization are implemented and tested.

Keywords: Parallelization, Optimization, Python, Earthquake Engineering.

1 INTRODUCTION

Earthquakes are one of the most dangerous hazards to our civilization. Protecting buildings is of special interest in earthquake engineering to save life during an earthquake as well as to maintain critical infrastructure for rescue operations after such an event. To achieve this, so called seismic protection devices are developed. They can be roughly grouped into ground isolation and energy dissipation devices. The former acting as a decoupling between the shaking ground and the construction to reduce the movement which is brought into the structure. They also have the task of recentering the building to its initial position. An example for those devices are sliding pendulum isolation bearings with spherical sliding surfaces. The second category are the energy dissipation devices, which should transform as much movement energy into thermal energy. This is typically done by hydraulic dampers, which are using the friction of a fluid to dissipate energy. Depending on their dimension, hydraulic dampers are capable of large displacements. Steel hysteresis

dampers on the other hand are using the energy dissipation from plastic deformation of steel, which is highly economical and effective, but their movement capacity is very limited compared to hydraulic dampers.

Another device, which is included in most medium to large-span bridges and is initially not considered as an earthquake protection device, are the expansion joints. Their task is to cover gaps between several bridge sections and between the bridge and the respective abutment. These gaps are included to compensate relative movements between those segments due to heat expansion of the bridge. Larger bridges generating larger expansion movements, so those gaps become larger, too. To divide those large gaps into smaller gaps, modular expansion joints come into consideration. They are consisting of steel center beams which are placed perpendicular to the roadway and subdivide the gap into sub-gaps. To distribute the change of the gap over all sub-gaps, a control mechanism must be included. A modern approach for this task, which is suitable for almost all sizes of expansion joints, is the swivel joist control mechanism. The support beams, which are fulfilling the task of the swivel joist, are rotating in consequence of a changing gap. The rubber bearings, which are connected to the support beams and can only rotate around the vertical axis, transmitting the swivel motion to the overlying center beams. The complete control mechanism follows the intercept theorem and guarantees that the gap is distributed equally over all sub-gaps at any time. A schematic representation of this mechanism is displayed in figure 1.

Figure 1: Schematic representation of the kinematic control mechanism of a swivel joist expansion joint

Modular expansion joints are not considered as seismic protection device yet. But they include some interesting properties as they generate high friction forces due to many sliding contacts in large expansion joints. Those friction forces acting in opposite direction of the relative movement and can be therefore seen as damping forces as they are generated by dissipation devices. Those forces are acting at strategically advantageous locations as they are able to reduce the movement of the bridge superstructure. Another outstanding benefit of the modular expansion joint, especially for larger devices, is their enormous movement capacity. Those possibilities are very promising and should be investigated to further enhance the seismic performance of those devices.

Other works like [1] investigated the seismic response of bridges with models which represent the pounding on finger expansion joints. In this work however, a multibody dynamics model of more complex modular expansion joints will be built. Therefore, a simulation model of the dynamic response of modular bridge expansion joints and the connected bridge under earthquake loads is developed. The mechanical control mechanism of modular expansion joints can, depending on the bridge size, consist of several hundred bodies and therefore around thousand degrees of freedom (DOF). This combined with the stiff differential equations arising from stiff force elements requires several optimization steps.

2 MODELING

Large expansion joints are of special interest, because the hazard potential and economical damage on large span bridges with such expansion joints is very high. In this work, the expansion joint

displayed in figure 2b will be modeled and simulated.

Figure 2: Expansion joint, modeled with 307 rigid bodies, 270 bushings, 270 friction force elements and 270 prismatic sliding joints

This expansion joint is of type swivel joist expansion joint with 10 support beams and 27 center beams. The support beam on the left hand side is fixed on both ends and acts as a guide for the center beams. All other support beams are translationally fixed on the bottom side and can therefore only rotate around the vertical axis of the drawing plane. The support beams have angles of 0° , 13° , 17° and 20° relative to the roadway direction and are 7.5 m long. The center beams are 15 m long. Each center beam is connected to the support beams via one rubber bearing, which leads to a total of 270 bearings.

The complete modeling and simulation will be implemented with the scripting language Python and mainly with the numerical module Numpy [2]. Python offers an easy and efficient way to create such simulation models and connect to other development environments with its broad range of modules. Furthermore, Python has great interfaces and frameworks to display, analyze and interactively publish results via notebooks or even as web applications. Numpy comes with many functions for numerical computing and it is mostly written in C for better performance. For optimization purposes, which are described in the next section, the complete calculation parts for the simulation are structured into Python functions. In the end, there will be one function called $\text{qdot}(\tau, q, \dots)$ which computes the time derivative of the global state vector. This function calls the other functions for computing forces, constraint reactions, Lagrange multipliers, solving the equation system and much more.

2.1 Bodies

The presented model consists of 307 rigid bodies. In order to describe them as 3D bodies, Euler angles could be used. This leads to 6 DOF for each body, respectively, and to an overall DOF count of 1842. Because implicit integration schemes must be used as described in section 2.4, the time expensive calculation of a Jacobian matrix is required, which has the shape $[2 \cdot \text{DOF}, 2 \cdot \text{DOF}]$. Because the Jacobian will be computed numerically by finite differences, each extra DOF will increase the effort for each function evaluation as well as the required number of those function evaluations. Because of this, the computation effort for the Jacobian matrix increases exponentially as displayed in figure 3. Using 2D bodies for the simulation, the number of DOF is halved compared to 3D bodies and the computation time for the Jacobian reduces to about one fifth.

The used rigid 2D bodies can translationally move in a plane defined by the x - and y -axis and

Figure 3: Normalized computation time for the calculation of the Jacobian matrix by finite differences depending on the DOF

rotate around the perpendicular z -axis. The state vector of one body is defined as follows

$$q = [x, y, \phi, \dot{x}, \dot{y}, \dot{\phi}]^T \quad (1)$$

2.2 Forces

2.2.1 Bushings

The main parts, of which a swivel joist expansion joint consists, is displayed in figure 4.

Figure 4: Parts of the swivel joist control mechanism. 1: Edge Beam, 2: Center Beam, 3: Support Beam, 4: Bearing

The bearing is connected to the overlying center or edge beam by a bushing, which has translational stiffness but can rotate freely. The translational stiffness is defined to match the corresponding shear stiffness of the rubber inlets of the bearing. The force gets calculated by a Python function with the displacement Δx in respective to the starting displacement u_0 at $t = 0$

$$\Delta x = (r_{01,0} + A_{01} \cdot r_{SP,1} - r_{02,0} - A_{02} \cdot r_{SP,2}) - u_0 \quad (2)$$

where $r_{0n,0}$ is the global position vector, A_{0n} the rotation matrix and $r_{SP,n}$ is the body-fixed vector from the center of mass to the actual connection point, all in respective to the connected bodies $n \in [1, 2]$ and in the global coordinate system 0 or the local, body-fixed coordinate system of the corresponding bodies. The resulting bushing force is then obtained by

$$F_{B,0} = k_B \cdot \Delta x \quad (3)$$

with the stiffness vector k_B including the stiffness in x - and y -direction. If $r_{SP,n} \neq 0$, then the resulting moment

$$M_{Bn,0} = (A_{0n} \cdot r_{SP,n}) \times F_{B,0} \quad (4)$$

is also evaluated.

2.2.2 Friction

The bearings are able to slide along the support beams. To ensure that those components are always in contact, a pre-loading mechanism is included which is not displayed in figure 4. The resulting normal force produces friction forces in the sliding plane. The newest generation of sliding bearings consists of sliding surface made out of ultra high molecular weight polyethylene (UHMWPE) with excellent sliding properties. Although friction coefficients are initially relatively low, it is expected that this coefficient increases over time due to dirt and other particles. With this long term effects in mind, a friction coefficient of 5 % is used for all upcoming simulations. In reality, there is another sliding component at the bottom sliding surface of the support beam included by the aforementioned pre-loading mechanism. Because this component does not affect the control mechanism, it gets ignored for the simulation model and only its friction force, which is similar to the friction force of the bearing, is considered.

The friction forces are calculated by the regularized friction model [3] displayed in figure 5. This model represents the singularity of the Coulomb friction model at $v_t = 0$ with a linear approximation. Friction tests have shown, that the friction force of the UHMWPE sliding bearing is nearly constant at higher velocities, so the assumption of constant friction is acceptable.

Figure 5: Schematic representation of the regularized friction model

The velocity between the two sliding bodies gets determined as

$$\Delta \dot{x} = \dot{r}_{01,0} + \dot{A}_{01} \cdot r_{SP,1} - \dot{r}_{02,0} - \dot{A}_{02} \cdot r_{SP,2} \quad (5)$$

The absolute tangential velocity is then calculated by the Frobenius norm

$$v_t = |\Delta \dot{x}|_F \quad (6)$$

With the friction coefficient, obtained by the tangential velocity, the Friction force can be calculated as follows

$$F_{fric} = \mu F_N \frac{\Delta x}{v_t} \quad (7)$$

2.3 Constraints

The bearings are considered to slide along the corresponding support beam without any significant gaps on the either side. This means, the support beam and bearing can be considered as a sliding

joint where the bearing can only slide along an axis defined by the support beam. All other DOF of the bearing are eliminated by the kinematic constraint. Because this system will be described by a index reduced differential algebraic equation system (DAE), a constraint equation for this kinematic joint is defined. The example below defines that the x -axis of body 1 defines the sliding axis

$$g(y) = \begin{bmatrix} (A_{01} \cdot e_x)^T (A_{02} \cdot e_y) \\ (A_{01} \cdot e_x)^T r_{12,0} \end{bmatrix} = 0 \quad (8)$$

where e_x and e_y are unity vectors in each direction and y is the location vector. This equation system implies in the first equation that the local coordinate systems of each body must be equally oriented and in the second equation that the connecting array must be perpendicular to the sliding direction [4].

2.4 Differential-Algebraic-Equations

As mentioned in the section before, the system is described by DAE of the form

$$\dot{y} = z \quad (9)$$

$$M\dot{z} = F^e + J_g^T \lambda \quad (10)$$

$$0 = g(y) \quad (11)$$

where y is the location vector, z is the velocity vector, M is the mass matrix, F^e are the external forces, J_g is the Jacobian matrix of the constraint equations, λ are the Lagrange multipliers and $g(y)$ are the constraint equations [5]. In order to solve this equation system, the index need to be reduced from 3 to 1, which could be done with to consecutive time derivations of $g(y)$, so that equation 11 becomes

$$0 = J_g \dot{z} + \dot{J}_g z \quad (12)$$

and the DAE can be solved with any ordinary numerical integrator. To reduce the drift in the constraint equation resulting from the index reduction, a stabilization according to Gear, Gupta and Leimkuhler [6] is implemented, which adds another Lagrange multiplier

$$\mu = - (J_g J_g^T)^{-1} J_g z \quad (13)$$

so that equation 9 becomes

$$\dot{y} = z + J_g^T \mu \quad (14)$$

This stabilizes the equation system on the velocity level and reduces the drift in the constraint equations.

3 OPTIMIZATION

When running a simulation with the described Python and Numpy implementation of the model displayed in figure 2b, the overall computation time even for simple and steady movements is extremely high. For example, the so called service case, where the joint opens and closes with a velocity of 0.2 ms^{-1} , takes several days to complete, even on virtual machines with up to 64 processor cores and optimized vector processing capabilities. Numpy is able to us multithreading for several vector workloads, but increasing the core count does not make any noticeable difference and the maximum number of used threads is limited to 32 anyway. This means, the roots of all heavily time consuming calculations has to be identified and specifically treated.

3.1 Just-in-Time Compiling

To identify the bottlenecks inside the simulaiton, a line-by-line profiler for the function $\dot{q} = f(t, q)$ is used. With this techniques, the specific time consumed by a single line in a function can be identified. The main evaluation times for $f(t, q)$ are displayed in table 1.

Task	Time/Hit	% Time
Computaion bushing/friction forces	0.25 s	55.7 %
Computaion constraints	0.2 s	44.3 %

Table 1: Evaluation times of specific parts of the function $f(t, q)$

Almost the complete effort of one function call goes into the determination of bushing, friction and constraint forces. All other tasks, like the solution of the linear equation system from equation 10, are not affecting the computation time by much. The following chart shows the included functions and how often they are called in this setup.

Figure 6: Relationship between different function calls and how often these functions are called

It can be seen, that the function $f(t, q)$ is calling each function about 270 times for this model. The function itself is called 1-5 times, depending on the convergence, by the used BDF integrator described in [7]. The integrator needs the Jacobian matrix

$$J \approx \frac{\partial f(t, q)}{\partial q} \quad (15)$$

to solve the nonlinear equation system, as described in [8]. Although the evaluation of the Jacobian matrix is not done at each time step, overall it introduces a lot of function calls. This shows the importance of efficient function evaluations.

Python is generally a relatively low performing programming language when it comes to computation time because it is interpreted. Even with the optimized Numpy calculations, many loops and other pure Python operations need to be done. To gain more efficiency, a compiled code is preferred. One solution would be to re-write the functions with C/C++, compile them as shared libraries and wrap them via ctypes. A similar performance with less re-programming effort can be reached with compiling the Python functions with the just-in-time compiler Numba [9]. Numba is designed to be used with Numpy arrays. After adding Numba to the functions called by $f(t, q)$, the overall evaluation time of this function gets reduced by 60 %. Because the overall simulation time linearly depends on the function evaluation time, it is reduced by approximately two thirds as well.

3.2 Sparse Jacobian Matrix

When profiling the solver, it can be seen that the evaluation of the Jacobian is by far the most time consuming part. Obtaining the Jacobian with the default dense finite difference approach takes about 90 s. The used integrator `scipy.integrate.BDF` offers the possibility to use a sparse Jacobian. When using this option, the sparsity pattern must be provided in a form, which tells the integrator at which locations the Jacobian is nonzero. For the expansion joint from figure 2b, the sparsity pattern is as displayed in figure 7.

Figure 7: The sparsity pattern of the Jacobian matrix. Blue dots indicate where the matrix entry is not zero.

A count of the nonzero element results in 334747 nonzero element which represent about 10 % of the complete matrix. When providing this sparsity pattern, only those elements are getting calculated when the Jacobian is re-evaluated [10]. This lowers the evaluation time for the Jacobian matrix to about 47 s, which is almost twice as fast as the dense computation. But overall, the whole simulation takes still several days to finish, which is not convenient when aiming for parameter studies.

3.3 Parallel Jacobian Matrix

Another option is parallelization of heavy workloads such as the computation of the Jacobian matrix, which is by far the most demanding part when integrating a time step. Modern computing architectures are offering (virtual) machines with a wide range of hardware capabilities for such tasks. Instead of successively computing parts of the Jacobian matrix, it would be highly beneficial if those tasks could be done simultaneously at different cores of the CPU. The computation of the approximation of the Jacobian matrix, which should be parallelized, can be achieved with finite differences

$$J_i = \frac{f(t, q_{q[i]+\varepsilon}) - f(t, q)}{\varepsilon} \quad (16)$$

where $i \in [1, 1842]$, J_i is the i -th column of J and ε is a small value of typically 1×10^{-8} . This shows, that the columns of J could be calculated independently from each other, which means that multiple columns could be calculated in parallel and getting joined afterwards.

One of the core concept of Python is the global interpreter lock (GIL). This means that every Python code needs the single GIL to be executed and could not be multithreaded, which guarantees a thread-safe memory management. On the other hand, one Python interpreter could only run one thread at a time. This might be beneficial when using multithreading for Input/Output (I/O) bound tasks, but could result in a decreased performance of CPU bound tasks, which the computation of the Jacobian matrix is.

A solution for this problem is multiprocessing. Instead of one Python process using multiple threads, multiple Python processes are spawned where each process is completely isolated from the others. This is well suited for the computation of the Jacobian matrix because the calculation of each column is independent of the other columns. Each of those processes have their own Python interpreter, are therefore not affected by the GIL and can run truly in parallel. The overhead introduced by creating multiple processes and joining the results together is more than compensated by the achieved parallelism at CPU bound tasks.

Multiprocessing in Python could be achieved with the `multiprocessing` module, which is in the standard library of Python. When testing the performance of the multiprocessed computation of the Jacobian matrix, the evaluation time surprisingly increased when using more processes, as displayed in figure 8.

Figure 8: Increasing evaluation time of the Jacobian when using more processes

In consequence of this, the overhead of the multiprocessing steps are investigated. It was suspected, that more processes are leading to more overhead in splitting the workloads and joining the results so that the advantage of parallelism is neglected. But it was observed, that the overhead for each process is very small compared to the evaluation time of the function inside the computation of the corresponding Jacobian matrix column. This indicates that there must be other overhead, which throttles the performance of this processes.

When further investigating the Numpy processes, it was noticed, that Numpy is using true multithreading without a GIL, which is possible because they are written in C. What happens when using more processes, is that each of those processes wants to use multiple threads, which results in additional overhead managing the threads because many workloads demanding a limited amount of CPU threads. Luckily, it is possible to limit the number of threads which Numpy is using for its functions. Adding the thread count as another dimension to the parameter study from image 8, the graph from image 9 is obtained.

This result shows, that the parallelization through multiprocessing becomes more effective when the maximum number of threads used by Numpy is limited to 1. It can be seen also, that more processes result in an exponential decay. To identify the saturation, at which increasing the process count does not have a noticeable effect, the Python simulation is executed with 1-64 processes on a machine with 64 cores. The result is displayed in figure 10.

This indicates that a higher core count of more than 32 cores does not increase the performance of the Jacobian matrix evaluation any further.

Because the parallel computation of the Jacobian matrix is the most efficient with about 3 s, the future computations will use this method. Therefore, a wrapper function is implemented to create an interface which the solver can use for the re-evaluation of this matrix.

Figure 9: Jacobi matrix evaluation time depending on processes and allowed thread usage by Numpy

Figure 10: Jacobi matrix evaluation time depending on processes with clear saturation at about 32 processes

4 ASSESSMENT OF SEISMIC PROTECTION POTENTIAL

4.1 Environment Setup

The initial purpose of this model development was to investigate the seismic protection potential of an expansion joint like displayed in figure 2b. The suggested method to obtain this potential is by creating a response spectrum of a single DOF model which represents the bridge building. A response spectrum displays the maximum acceleration, also called spectral acceleration, in dependence of the period of the single DOF model. These accelerations can be obtained by making several time step simulations with different natural frequencies of the single DOF model and using the maximum acceleration response of those simulations. In order to generate a spectral acceleration, the complete simulation system displayed in figure 11.

The mass gets exposed to an earthquake ground acceleration and oscillates according to its natural frequency. The expansion joint is connected to this mass and experiences a earthquake acceleration from the abutment, too. The parameters from table 2 are used for the simulation.

The ground acceleration from the 1989 Loma Prieta earthquake [11] is used as the earthquake

Figure 11: Earthquake system setup

Mass	m	$1 \times 10^8 \text{ kg}$
Natural frequency	ω_0	6.3 s^{-1}
Friction	μ	0.05

Table 2: Parameters used for the earthquake simulation

acceleration.

4.2 Simulation Results

When considering, that larger bridges containing more than one expansion joint, the used mass for the single DOF model is realistic when computing with one expansion joint only. To see the damping effect, which the expansion joint generates on the oscillating mass, the acceleration of the mass with and without the expansion joint model is displayed in figure 12.

Figure 12: Bridge response acceleration with and without the expansion joint model

This result gives a first impression, on how expansion joints can influence the dynamic response of a bridge caused by an earthquake. The spectral acceleration from this step by step time integration would be 5.6 ms^{-2} without and 4.8 ms^{-2} with the expansion joint. Overall, this is a reduction of over 14 %.

5 CONCLUSIONS

It has been shown, how to achieve an efficient simulation for very large expansion joints. The Python model was structured in a way, that heavy and often called workloads are compiled into efficient functions. The fact, that many stiff force elements are present, an implicit solver was necessary to gain a stable result. This however introduced the need for the calculation of a Jacobian matrix, which is very time consuming when using the standard methods of the given Python solvers. A study on multiprocessing and limiting the number of threads used by the code showed that the most efficient method is using multiprocessing and deactivate multithreading within Numpy. This way, the Jacobi matrix evaluation time could be reduced from 90 s to 3 s, which is a significant speedup. The overall simulation time for such a model is now reduced to few hours instead of several days. The first results with a single DOF model representing a bridge showed very promising damping capabilities of the expansion joint. To investigate this behavior more deeply, several parameter studies will be done in order to identify the dependency of geometry, bridge model, friction parameters, ground motion and much more. The model is now efficient enough to deliver fast and stable results.

ACKNOWLEDGMENTS

I would like to express my very great appreciation the company MAURER SE for the financial support to make this project possible.

REFERENCES

- [1] Quan, G., Kawashima, K.: Effect of finger expansion joints on seismic response of bridge. *Doboku Gakkai Ronbunshuu A* **65** (2009) 243–254
- [2] Harris, C.R., Millman, K.J., van der Walt, S.J., Gommers, R., Virtanen, P., Cournapeau, D., Wieser, E., Taylor, J., Berg, S., Smith, N.J., Kern, R., Picus, M., Hoyer, S., van Kerkwijk, M.H., Brett, M., Haldane, A., del Río, J.F., Wiebe, M., Peterson, P., Gérard-Marchant, P., Sheppard, K., Reddy, T., Weckesser, W., Abbasi, H., Gohlke, C., Oliphant, T.E.: Array programming with NumPy. *Nature* **585**(7825) (September 2020) 357–362
- [3] Oden, J., Martins, J.: Models and computational methods for dynamic friction phenomena. *Computer Methods in Applied Mechanics and Engineering* **52**(1) (1985) 527–634
- [4] Flores, P.: Concepts and Formulations for Spatial Multibody Dynamics. (01 2015)
- [5] Rill, G., Schaeffer, T., Borchsenius, F.: Grundlagen und computergerechte Methodik der Mehrkörpersimulation – Vertieft in Matlab-Beispielen, Übungen und Anwendungen. (02 2020)
- [6] Gear, C., Leimkuhler, B., Gupta, G.: Automatic integration of Euler-Lagrange equations with constraints. *Journal of Computational and Applied Mathematics* **12-13** (1985) 77–90
- [7] Byrne, G.D., Hindmarsh, A.C.: A Polyalgorithm for the Numerical Solution of Ordinary Differential Equations. *ACM Trans. Math. Softw.* **1**(1) (1975) 71–96
- [8] Hairer, E., Wanner, G.: Solving Ordinary Differential Equations II. Volume 14. (2010)
- [9] Lam, S., Pitrou, A., Seibert, S.: Numba: a LLVM-based Python JIT compiler. (2015) 1–6
- [10] Curtis, A.R., Powell, M.J.D., Reid, J.K.: On the Estimation of Sparse Jacobian Matrices. *IMA Journal of Applied Mathematics* **13**(1) (1974) 117–119
- [11] Pollitz, F., Biirgmann, R., Segall, P.: Joint estimation of afterslip rate and postseismic relaxation following the 1989 Loma Prieta earthquake. *Journal of Geophysical Research: Solid Earth* **103992** (11 1998) 975–26