Pipe Stress Infinity

Release 0.0.1

Denis Gomes

Sep 05, 2021

TABLE OF CONTENTS

CHAPTER

ONE

PREFACE

Pipe Stress Infinity (PSI) is an engineering design and analysis software used to evaluate the structural behavior and stresses of piping systems to a variety of different codes and standards.

PSI has an active developer and user community. If you find a bug or a problem with the documentation, please open an issue with the issue tracker. Anyone is welcome to join our discord server were a lot of the development discussion is going on. It's also a great place to ask for help.

1.1 Contact

PSI is developed by many individual volunteers, and there is no central point of contact. If you have a question about developing with PSI, or you wish to contribute, please join the mailing list or the discord server.

For license questions, please contact Denis Gomes, the primary author.

CHAPTER

TWO

QUICK REFERENCE

Welcome to the PSI Quick Reference Guide!

If you find a bug or a problem with the documentation, please open an issue with the issue tracker. If you have a question about developing with PSI, or you wish to contribute, please join the mailing list or the discord server.

For license questions, please contact Denis Gomes, the primary author.

2.1 Overview

2.1.1 Requirements

PSI can run on any system Python runs because it is built on top of the Python interpreter. Python 3.5 is required at a minimum.

2.1.2 Contributing

For full disclosure, as there is an Enterprise Edition of PSI, contributors are made aware that any contributions made to the Community Edition may be merged with the Enterprise version. All contributors are therefore asked to acknowledge and agree to this by signing a consent form before being allowed to make contributions to the project.

Note: All funding going towards the Enterprise Edition will ultimately go towards improving and maintaining the Community Edition.

Active members and contributors will be invited to become part of the core development team. Off these members, a select few will have the opportunity to work on the Enterprise Edition if they so choose. These members will be funded for their contributions dependent on the overall success of the project.

All prominent contributors are acknowledged in the CREDITS file found in the project root directory.

2.1.3 Support

You can support PSI in a variety of different ways.

- Become a contributer and help maintain the code base, Code PSI.
- Join the *Community* and share your stories with others, Promote PSI.
- You can fund open source development via GitHub sponsers, Fund PSI.

2.1.4 Community

Become an active member of the community.

- Talk to other developers and users at the PSI discord server.
- Join the PSI mailing list.
- Submit bugs and feature requests on the PSI issue tracker.

2.1.5 Change Log

master

This is the first ever release of Pipe Stress Infinity (PSI). It is a proof of concept release and as such consists of limited functionality. The API has yet to be completely flushed out and fully documented.

Features

- Interpreter built on top of Python's
- Terminal progress bar display for analysis run status
- Point object and straight run element modeling
- Imperial and International system of units (SI) support
- Anchor and GlobalY support modeling
- Deadweight, Pressure and Thermal loading
- Loadcase support for different types of loads
- Load Combination using Algebraic combination method
- Linear static analysis for piping models
- Code checking per B31.1 for straight runs
- Movements, reactions, forces and stress results

Fixes

None

Other Changes

None

2.2 Licensing

PSI is offered under an 'open core' license model. As such, there are two different versions available:

1. A Community Edition

   Open source software based on the GPLv3 license, which allows it to be freely used by individuals and companies. This version is hosted on GitHub.

2. An Enterprise Edition

   Commercial version with accesss to all features and the flexibility to modify the source code.

Note: The Enterprise Edition is in active development. To learn more about it's status and pricing setup please email me directly. Customers who are very satisfied with the Community Edition and/or need the added feature set are welcome to inquire.

Check out the *Features* section for a breakdown of what each version has to offer.

2.3 Installation

2.3.1 Install from PyPi

To install PSI, type the following command in the terminal:

```
$ pip install psi --user
```

Note: Python 3.5 or above must already be installed on your system. If you are on windows, pip should be available after the install.

If pip was not installed by default, get and install it using the directions provided here.

2.3.2 Install From Source

If you're reading this from the README, i.e. you have the source distribution, you can install PSI using:

```
$ python setup.py install --user
```

2.3.3 Installers

Windows installers will be made available for the *Enterprise Edition* of PSI at xscode.

2.4 Features

2.4.1 Community Edition

The Community Edition of PSI is feature packed and capable of solving a wide variety of different piping problems. The following is a list of its primary capabilities:

- Modeling of major piping components such as Runs, Bends, Reducers, Valves, and Flanges.
- Ability to specify section and material data properties for different cross-sections.
- Access to a variety of different support types including Anchors, GlobalX, GlobalY, and GlobalZ.
- Non-linear support capability *excluding* gaps and friction.
- Assign loads such as Weight, Pressure, Thermal, Wind and Seismic among others.
- Static linear analysis of loadcases and combinations.
- Stress evaluation based on B31.1 piping code.
- Movements, support reactions and internal force results.
- Open source and free to use.
- And more to come...

2.4.2 Enterprise Edition

All the features available in the Community Edition of PSI are also included in the Enterprise edition by default. In addition, the features listed below are only available with the upgraded version.

- Access to all element types, such as bellows.
- Comprehensive section and material database.
- Valve and flange weight database.
- Additional spring vendors for spring selection.
- Pipe support gaps and friction.
- Support for additional codes and standards.
- Dynamic modal, response spectrum and time history analyses capabilities.
- Trunnion analysis.
- Nozzle evaluation for various equipment.
- Compability with other pipe stress programs such as AutoPIPE and Caesar II.
- And more to come...

In addition, the Enterprise Edition is subjected to rigorous verification and validation, going far above and beyond the Tier 1 textbook examples employed for the Community Edition. A 3-tier approach is used to throughly benchmark the software for a variety of different test cases.

Note: The Enterprise Edition is in active development. To learn more about it's status and pricing setup please email me directly. Users who are very satisfied with the Community Edition and/or need the added feature set are welcome to inquire.

2.5 Codes

2.5.1 United States

Longitudinal Stress Due to Pressure - S_{lp}

$S_{lp} = P\ D_o / 4\ t_n$ [Approximation]

$S_{lp} = P\ D_i^{\ 2} / (D_o^{\ 2} - D_i^{\ 2})$ [Exact Formulation]

Equation	Allowable	Stress Type
B31.1 (1967)		
$S_l = S_{lp} + (S_b^{\ 2} + 4\ S_t^{\ 2})^{1/2}$	$< S_h$	Sustained
$(S_b^{\ 2} + 4\ S_t^{\ 2})^{1/2}$	$< f[1.25\ S_c + 0.25\ S_h + (S_h - S_l)]$	Expansion
$S_{lp} + (S_b^{\ 2} + 4\ S_t^{\ 2})^{1/2}$	$< k\ S_h$	Occasional
Under Construction!		

Where:

- A_m - Cross-sectional metal area in pipe.
- D_o - Outer pipe diamter
- D_i - Inner pipe diamter
- t_n - Pipe nominal thickness
- P - Pressure
- S_{lp} - Longitudinal pressure stress
- S_l - Longitudinal stress (pressure + bending)
- S_b - Bending stress
- S_t - Torsional stress
- S_h - Material hot allowable stress
- S_c - Material cold allowable stress
- E - Material Young's Modulus
- f - Fatigue cycle reduction factor
- k - Cumulative usage factor

2.5.2 International

Under Construction!

2.6 Example

The purpose of this example is to demonstrate how to solve a cantilevered pipe problem using standard engineering strength of materials formulations and using PSI's finite element capabilities.

Problem

A 10 *feet* long, 10" schedule 40 cantilevered pipe is anchored at one end with a 1000 *lbf* force (P) acting on the other. Determine the tip deflection at the end with the force? What is the reaction force (R) at the fixed end?

Methodology

The deflections and reaction forces are derived using strength of materials formulations and calculated using the formulas in *Figure 1*.

Acceptance Criteria

1. ASME B&PV B31.1 Power Piping Code 1967 Edition

Assumptions

1. The pipe is made of Standard Steel with a Young's Modulus of 2.9e7 *psi*.
2. Shear deflection effects are negligible.

Inputs

1. L = 10 ft - Pipe length
2. E = 2.9e7 psi - Young's modulus of pipe material
3. P = 1000 lbf - Force applied at the end

Analysis

The applied end force results in:

1. a downward deflection at the tip and zero deflection at the anchor point
2. an upward reaction force and clockwise moment at the fixed end. See the internal force and moment diagram.

$$\Delta_{max} = Pl/3EI \tag{2.1}$$

$$R = P \tag{2.2}$$

$$M_{max} = Pl \tag{2.3}$$

Where:

$$I = (\pi/64)(d_o^4 - d_i^4) \tag{2.4}$$

Plugging into the formulas from *Figure 1* and solving for the deflection (2.1), shear (2.2), and max moment (2.3) gives:

```
Mmax = 10000 foot * force_pound
```

Source Code

The PSI code listing below is used to solve the cantilevered beam pipe example above.

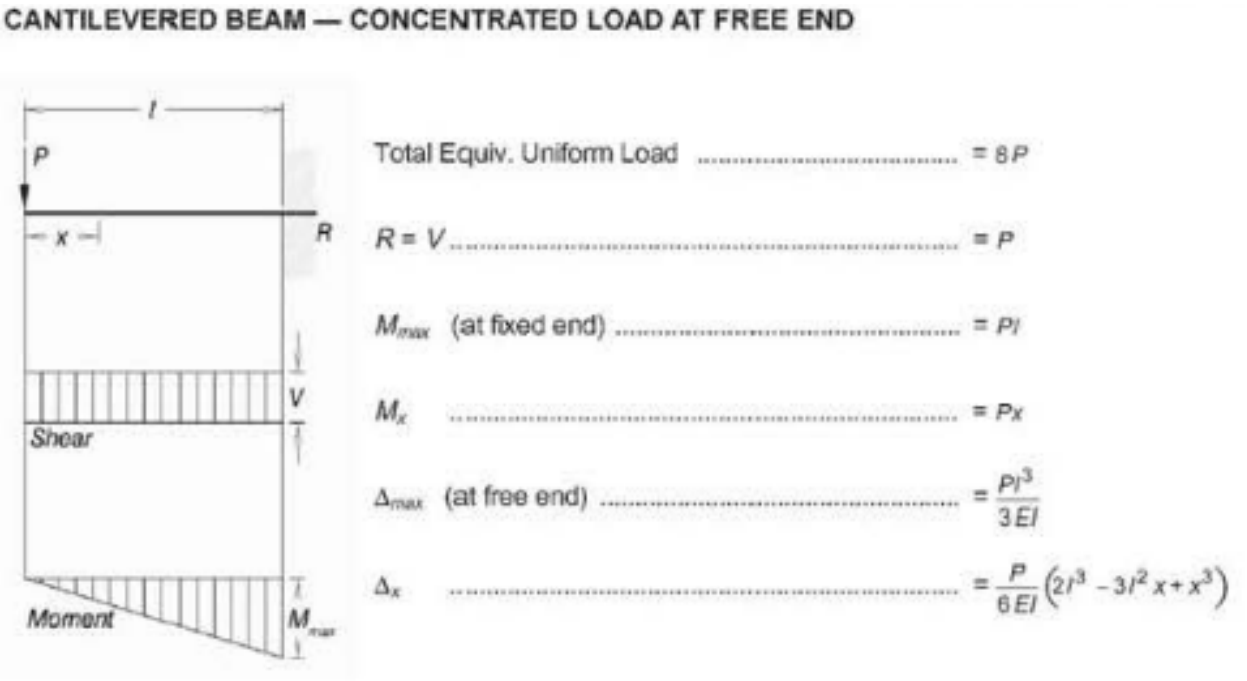

Fig. 1: *Figure 1:* Shear and Bending Moment Diagrams *[1]*

Listing 1: Code Listing

```
#! /usr/bin

# parameters
L = 10 * 12

# create model
mdl = Model('demo')

# define properties
pipe1 = Pipe.from_file('pipe1', '10', '40')
mat1 = Material.from_file('mat1', 'A53A', 'B31.1')

# create geometry
pt10 = Point(10)
run20 = Run(20, L)

# assign supports
anc1 = Anchor('A1', 10)
anc1.apply([run20])

# define loads for operating case 1
w1 = Weight('W1', 1)
w1.apply([run20])

p1 = Pressure('P1', 1, 250)
p1.apply([run20])

# define a loadcase
l1 = LoadCase('l1', 'sus', [Weight, Pressure], [1, 1])
```

(continues on next page)

(continued from previous page)

```

# code
b311 = B311('B31.1')
b311.apply([run20])

# run the analysis
mdl.analyze()

# postprocess
disp = Movements('r1', [l1])
disp.to_screen()

reac = Reactions('r2', [l1])
reac.to_screen()

stress = Stresses('r3', [l1])
stress.to_screen()
```

Results

Under Construction!

References

1. AISC ASD 9th Edition
2. ASME B&PV B31.1 Power Piping Code 1967 Edition

CHAPTER

THREE

USER GUIDE

Welcome to the PSI User Guide!

If you find a bug or a problem with the documentation, please open an issue with the issue tracker. If you have a question about developing with PSI, or you wish to contribute, please join the mailing list or the discord server.

For license questions, please contact Denis Gomes, the primary author.

3.1 Overview

At its core, PSI is a finite element analysis program which allows users to model complex piping systems subjected to a variety of loads and loading conditions.

The general process is as follows:

1. Model the piping system using various piping components such as runs, bends valves, flanges, etc.
2. Apply boundary conditions, such as anchors, linestops and guides.
3. Apply deadweight, thermal and fluid loadings, just to name a few.
4. Solve the computer model for static, modal, harmonic and time history type loadings, and finally,
5. Post-process the results to gain valuable insight and to drive design decisions.

3.2 Settings

3.3 Modeling

Under the hood, PSI uses finite element beam elements to model various piping components typically encountered in piping system.

3.3.1 Components

The geometry of piping components are defined by the coordinates of the 'from' and 'to' points. Each component must also have a section and material assigned to be fully defined. Extra data such as insulation/cladding/refractory, piping code, and sifs/connections may also be applied based on design requirements.

Run

The most generic piping component is a pipe Run; all other elements are derived from a Run or approximated using a Run. A Run element is simply a 3D beam element consisting of two nodes, each with 6 Degrees of Freedom (DOF), resulting in a total of 12 DOFs per element. The user must define the global (x,y,z) coordinate positions of the 'from' and 'to' nodal points for a Run. Doing so effectively set the length (i.e. the extents) of the element.

The stiffness matrix derivation for a beam element is commonly documented in various books and in literature. PSI uses the Timoshenko beam derivation from *"Theory of Matrix Structural Analysis"* by J.S. Przemieniecki, which accounts for shear deflection, an effect more pronounced for thick wall piping. Note that shear effects are turned on my default but may be turned off by the user from model settings.

The element (local) stiffness matrix is a 12x12 matrix:

$$k_{m,n} = \begin{pmatrix} k_{1,1} & k_{1,2} & \cdots & k_{1,n} \\ k_{2,1} & k_{2,2} & \cdots & k_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ k_{m,1} & k_{m,2} & \cdots & k_{m,n} \end{pmatrix} \tag{3.1}$$

with m=n=12 and where,

$$k_{1,1} = \frac{EA}{L}$$
$$k_{1,7} = k_{7,1} = \frac{-EA}{L}$$
$$k_{2,2} = \frac{12EI_z}{L^3(1+\Phi_y)}$$
$$k_{2,6} = k_{6,2} = \frac{6EI_z}{L^2(1+\Phi_y)}$$
$$k_{2,8} = k_{8,2} = \frac{-12EI_z}{L^3(1+\Phi_y)}$$

$$k_{2,12} = k_{12,2} = \frac{6EI_z}{L^2(1+\Phi_y)}$$
$$k_{3,3} = \frac{12EI_y}{L^3(1+\Phi_z)}$$
$$k_{3,5} = k_{5,3} = \frac{-6EI_y}{L^2(1+\Phi_z)}$$
$$k_{3,9} = k_{9,3} = \frac{-12EI_y}{L^3(1+\Phi_z)}$$
$$k_{3,11} = k_{11,3} = \frac{-6EI_y}{L^2(1+\Phi_z)}$$

$$k_{4,4} = \frac{GJ}{L}$$
$$k_{10,4} = k_{4,10} = \frac{-GJ}{L}$$
$$k_{5,5} = \frac{(4+\Phi_z)EI_y}{L(1+\Phi_z)}$$
$$k_{5,9} = k_{9,5} = \frac{6EI_y}{L^2(1+\Phi_z)}$$
$$k_{5,11} = k_{11,5} = \frac{(2-\Phi_z)EI_y}{L(1+\Phi_z)}$$

$$k_{6,6} = \frac{(4+\Phi_y)EI_z}{L(1+\Phi_y)}$$
$$k_{6,8} = k_{8,6} = \frac{-6EI_z}{L^2(1+\Phi_y)}$$
$$k_{6,12} = k_{12,6} = \frac{(2-\Phi_y)EI_z}{L(1+\Phi_y)}$$
$$k_{7,7} = \frac{EA}{L}$$
$$k_{8,8} = \frac{12EI_z}{L^3(1+\Phi_y)}$$

$$k_{8,12} = k_{12,8} = \frac{-6EI_z}{L^2(1+\Phi_y)}$$
$$k_{9,9} = \frac{12EI_y}{L^3(1+\Phi_z)}$$
$$k_{9,11} = k_{11,9} = \frac{6EI_y}{L^2(1+\Phi_z)}$$
$$k_{10,10} = \frac{GJ}{L}$$
$$k_{11,11} = \frac{(4+\Phi_z)EI_y}{L(1+\Phi_z)}$$
$$k_{12,12} = \frac{(4+\Phi_y)EI_z}{L(1+\Phi_y)}$$

and where,

$$\Phi_y = \frac{12EI_z}{GA_yL^2}$$

$$\Phi_z = \frac{}{}$$

The element global stiffness matrix is calculated by pre and post multiplying the element local stiffness matrix by the transpose of the transformation matrix and transformation matrix respectively, as shown below:

$$k_{global} = T^T * k_{local} * T \tag{3.2}$$

The local to global transformation matrix is used to convert quantaties that are defined with respect to element coordinates to global coordinates. It consists of the direction cosines of the element local axes given in matrix format as

shown below where m=n=12:

$$T_{m,n} = \begin{pmatrix} dc_{1,1} & dc_{1,2} & dc_{1,3} & 0 & \cdots & 0 \\ dc_{2,1} & dc_{2,2} & dc_{2,3} & 0 & \cdots & 0 \\ dc_{3,1} & dc_{3,2} & dc_{3,3} & 0 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots & & \\ T_{m,1} & T_{m,2} & \cdots & T_{m,n} & & \end{pmatrix} \tag{3.3}$$

The 3x3 direction cosine matrix (dc) is given below:

$$dc_{3,3} = \begin{pmatrix} localx_1 & localy_1 & localz_1 localx_2 & localy_2 & localz_2 localx_3 & localy_3 & localz_3 \end{pmatrix} \tag{3.4}$$

For a Run element, the local x direction is given by the vector from the 'from' point to the 'to' point. The local y is parallel to the global vertical direction and the local z is the cross product of the local x and y axes.

When the local x is parallel to the vertical direction, the local y is aligned with global x (arbitrarily) and the local z is the cross product of local x with local y.

Bend

Pipe bends are approximated using multiple Run elements strung together based on an underlying rational bezier curve used to locate each point coordinate. Similar to a Run, a Bend has a 'from' and 'to' point. It also has a 'near', 'far' and 'mid' point. The 'near' and 'far' points are the start and end points for the Bend. The 'mid' point of the bend is the physical center of the Bend located on the arc length. Physical quantaties are calculated at these three point locations and directly exposed to the user. The results for all other points are solved but not directly accessible.

Note: The 'to' point is the corner of the bend element and does not fall on the Bend arc length.

Based on the geometry of the Bend and the governing piping code, the Bend flexibility factor is calculated, and the stiffness matrices of all the approximating Run elements are divided by the same factor.

Note: Only the bending stiffnesses are affected by the increase in flexibility and therefore altered.

Reducer

Similar to a pipe Bend, a Reducer is approximated using multiple Run elements with reducing cross-sections.

Rigid

A Rigid element is used to model relatively stiff components in a piping system such as equipment for example to implicitly determine the nozzle movements and nozzle loads at the interface points. These elements can be weightless or have a mass assigned to them. They can also have a temperature assigned to them or have the fluid contents turned off.

Note: The relative stiffness is achieved by multiplying the thickness of the piping section by a factor of 10 times.

Valve

A Valve element is basically a Rigid element with a mass assigned to it. The mass can be user defined or pulled for a data file. From a stress standpoint a Valve locally stiffens a piping system. It may also have Flanges defined that effectively increases the overall weight of the Valve.

Flange

Similar to a Valve, a Flange is also a Rigid element. The mass of the Flange can be user defined or pulled for a data file. Leak testing of Flanges is possible dependent on the code.

Bellow

Under Construction!

3.3.2 Component Data

Sections

Under Construction!

Materials

Under Construction!

Insulation / Refractory / Cladding

Under Construction!

Codes

Under Construction!

SIFs/Connections

Under Construction!

3.4 BCs

3.4.1 Anchor

Under Construction!

3.4.2 X

Under Construction!

3.4.3 Y

Under Construction!

3.4.4 Z

Under Construction!

3.4.5 Guide

Under Construction!

3.4.6 LineStop

Under Construction!

3.4.7 Spring

Under Construction!

3.4.8 Snubber

Under Construction!

3.4.9 Incline

Under Construction!

3.5 Loadings

3.5.1 Weight

Under Construction!

3.5.2 Pressure

Under Construction!

3.5.3 Fluid

Under Construction!

3.5.4 Thermal

Under Construction!

3.5.5 Hydro

Under Construction!

3.5.6 Wind

Under Construction!

3.5.7 Seismic

Under Construction!

3.5.8 Displacement

Under Construction!

3.6 Loadcases

Under Construction!

3.7 Solvers

3.7.1 Static

Go through all loadcases, solve each one and then combine them to generate the final results.

For each element calculate the element stiffness matrix and assemble the system stiffness matrix using the nodal degree of freedom. The DOFs of a node is based on the position/index of the point in the points list

The loads defined for each element primary/primitive loadcase is combined into a single load vector. The load vector for each element loadcase is then assembled into a global load matrix. Multiple global load vectors are solved for at once using gaussian elimination.

Note that to simplify the FEA solution, all elements are simple beams, including bends and reducers which are approximated by multiple beams chained together.

General Procedure

1. Define NDOF indices for element nodes.
2. **For each element**
    a. Construct the local stiffness matrix.
    b. Construct local force vector, one for each primitive loadcase.
    c. Transform local stiffness and force matrix.
    d. Add global element stiffness matrix and the global element force vectors to the global system stiffness and forces matrix, respectively.
3. Apply the boundary conditions using the penalty method.
4. Solve the global system using guassian elimination techniques.

    AX=B

    Where A is the global system matrix.

    B is the matrix of force vectors, one for each primitive loadcase, and X is the matrix of solutions vectors for each primitive loadcase.
5. Use the calculated displacements for each primitive loadcase to calculate the element force and moments.
6. Use the results from step 5 to calculate nodal reactions and element forces. Finally use the element forces to calculate code stresses.

Reducers and Bends

Both reducers and bends are topologically curves and therefore approximations. The midpoint of a bend is created as a side effect of defining the bend when it is created. All the other vertices are inaccessible from a nodal standpoint. In other words, the user does not have control of the other underlying vertices because only the vertex that corresponds to the midnode is referenced by the midpoint.

The static solver preprocesses all reducers and bends such that a point is made for each underlying vertex at runtime. Once the solution is generated all the temporary points are deleted along with the nodal data corresponding to each point. For the case of a bend, the solution for the midpoint and end points are kept. For the reducer, the results for the end points are kept only similar to all the other element.

Tee Flexibility

Under Construction!

Skewed Supports

Implemented using local stiffness transformation.

Spring Algorithm

The program tries to place a rigid support, variable or constant spring where a spring support is specified by the user. The algorithm must satisfy all the operating cases such that it does not bottom or top out. For each operating case a linear deadweight analysis (W+P+D+F) is performed and a +Y support is placed at each supports. Remove the force from the weight case at each spring support location with the equivalent upward force and run the operating case with the temperature. Determine the upward movement of the pipe at the spring support. Use the applied force (i.e. the hot load) and the movement to pick a hanger from the manufacturer catalog.

Master Slave (CNode)

Constraint equations.

Non-Linear Supports

For each loadcase, for each nonlinear support, per each solve:

Initially a non-linear (+Y) support is converted to a linear full Y support. If the solution shows a (-Y) load on the support, the support is modeled correctly and "active" for the loadcase. If not, the program marks it as "inactive" and gets rid of the support altogether and tracks the displacement at that point. If the point shows a (+Y) deflection, this support is modeled correclty for this loadcase. Else, the stiffness matrix is reset to a full Y. If any of the nonlinear assumption proves incorrect reanalyze with the updated stiffness and force vector (if required). Continue the process checking all the "active" supports for loads and all the "inactive" supports for displacement. When all "active" supports have a (-Y) and the "inactive" suports show a positive up displacement. A status change flag variable is used to indicate a support that that initially shows a (-Y) load but then starts to show an uplift.

Note: A *static* nonlinear analysis is performed by iterating until the solution converges.

Friction

Under Construction!

Note: A *static* nonlinear analysis is performed by iterating until the solution converges.

Loading Sequence and Non-Linear Supports

Non-linear supports use an iterative approach to determine the final support loads and pipe configuration. The sequence in which loads are applied matters as superposition is not valid for non-linear analysis. Each load is applied and the displacements extracted. These movements are then used for the next step, ie. the model mesh is modified to incorporate the new point locations. As a result the system stiffness matrix is updated each iteration.

3.7.2 Modal

Under Construction!

3.7.3 Harmonic

Under Construction!

3.7.4 Spectra

Under Construction!

3.7.5 Time History

Under Construction!

3.8 Post-Process

CHAPTER

FOUR

APPLICATION GUIDE

Welcome to the PSI Application Guide!

If you find a bug or a problem with the documentation, please open an issue with the issue tracker. If you have a question about developing with PSI, or you wish to contribute, please join the mailing list or the discord server.

For license questions, please contact Denis Gomes, the primary author.

4.1 Overview

Under Construction!

4.2 What's New

Coming soon!

4.3 Programming

Under Construction!

CHAPTER

FIVE

API REFERENCE

Welcome to the PSI API Reference!

If you find a bug or a problem with the documentation, please open an issue with the issue tracker. If you have a question about developing with PSI, or you wish to contribute, please join the mailing list or the discord server.

For license questions, please contact Denis Gomes, the primary author.

5.1 psi

Under Construction!

5.2 psi.settings

Default application settings.

A configuration object is created when a model is first instantiated. The program will use the model settings defined in the configuration instance.

Application and model settings may diverge over time due to different versions. When the software is updated new application settings will be merged with the model settings to ensure backwards compatibility between version. Note that application settings in this context are the settings in this file with respect to the latest version of the software.

5.2.1 Configuration

class psi.settings.**Configuration**(*default_units='english'*)

Default model settings.

Parameters

- **units** (*str*) – Define the model units. 'english' by default.
- **vertical** (*str*) – Define the vertical direction for the model. 'y' by default.
- **stress_case_corroded** (*bool*) – Use reduced thickness to evaluate pipe stresses. Reduced pipe wall is not used for sections property calculations.
- **bourdon_effect** (*bool*) – Include bourdon effects.
- **pressure_thrust** (*bool*) – Include pressure thrust effects.
- **liberal_stress** (*bool*) – Use liberal stress allowable for expansion stresses.
- **weak_springs** (*bool*) – Include weak springs for numerical stability.

- **nonlinear_iterations** (*int*) – Number of iterations for non-linear supports per loadcase.
- **translation_stiffness** (*float*) – Support stiffness used in the translation directions.
- **rotation_stiffness** (*float*) – Support stiffness used in the rotation directions.
- **axial_force** (*bool*) – Include axial force due to structural loading for code stress calculations.
- **tref** (*float*) – Reference temperture used for thermal expansion calculations.
- **timoshenko** (*bool*) – Use Timoshenko beam formulation accounting for shear deformation.
- **version** (*str*) – Latest software version.

Methods

export (*fname*)
Export the current model settings

import_ (*fname*)
Import settings from an outside source

Properties

property units

5.3 psi.entity

Base entity and entity container for all psi objects.

Entity manages objects created by an user defined unique name.

After an object is created all functions work with objects only.

A specific object can be called (i.e. __call__) by its user defined name.

5.3.1 Entity

class psi.entity.**Entity**(*name*, *register=True*)
Base class for psi objects

5.3.2 EntityContainer

class psi.entity.**EntityContainer**
Base container object for an entity

5.4 psi.model

5.4.1 Model

class psi.model.**Model**(*name*)
The model object contains all internal objects.

Methods

__init__(*name*)
Create a model instance.

Example

```
>>> mdl = Model("mdl1")
```

analyze(*mode='static'*)
Run the analysis

5.4.2 ModelContainer

class psi.model.**ModelContainer**

Methods

5.5 psi.elements

Piping finite elements.

To create a model, the user must first define a series of runs that make the overall centerline of the model. After this process, additional components can be added by giving a point and a direction in which to place the new item or by "converting" a run to a different type of element.

Example

First define the model centerline using Point and Run.

```
>>> Model("test")
>>> Point(10, 0, 0, 0)
>>> Bend(20, -2, radius=...)     # can be created by a point
>>> Bend(30, 0, 2, radius=...)
>>> Run(40, 0, 0, 2)
```

(continues on next page)

(continued from previous page)

```
>>> Bend(50, 0, 0, 2, radius=...)
>>> Bend(60, 0, -2, radius=...)
>>> Run(70, 2)
>>> Point(80, -10)
>>> Run(90, 0, 2)
>>> Run(100, 0, 0.5)
>>> Point(90)
>>> Run(40)                        # defined twice and loop-closed (1) (2)
```

Now define the other components by point.

```
>>> elements(20, 30).split(25)       # split a bend - create run and bend
>>> Valve(25, length, mass, "mid")      # two elements, 3 nodes made
>>> Tee(40, "welding")                  # sif.Tee by point only
>>> Flange(10, length, mass, "down")    # flanged nozzle
>>> Flange(70, length, mass, "up")
```

There are three possible ways in which a point defined for a component can be interpreted.

1. If the point is defined as a midpoint, two elements are made, one on each adjacent run.
2. If the component is defined to be upstream, it will be placed in the oposite direction to the flow, with respect to the element nodal direction.
3. If the component is defined to be downstream, it will be placed in the same direction as the flow. In both the latter two cases, only one element is created.

When a component is defined at a boundary node, it is possible to create another boundary node by choosing the direction that 'extends' the model.

Components can also be created by picking a run element and 'converting' it to a different type of element such as a valve or a flange.

```
>>> elements(50, 60).split(54, 0.5, ref=50)      # used node 50 as reference
>>> elements(54, 60).split(55, 0.5, ref=54)
>>> elements(54, 55).convert(Valve, mass=...)
```

Note: Only runs can be split and merged. To split a valve for example, it must first be changed to a run and then split.

The following items require extra attention:

- Dealing with element weight vs mass when it comes to units.
- For Rigid elements the weight must be specified not the mass.
- Bend and reducer having multiple run approximation.
- How to easily create bends when defining geometry and how to update the geometry when point coordinates or bend radius is updated.
- How to covert different types of elements to runs and vice versa.
- Unit conversion should be disabled before the analysis is performed.

5.5.1 Run

class psi.elements.**Run**(*point*, *dx*, *dy=0*, *dz=0*, *from_point=None*, *section=None*, *material=None*, *insulation=None*, *code=None*)

Define a pipe run by incremental offsets in the global x, y and z directions.

Methods

__init__(*point*, *dx*, *dy=0*, *dz=0*, *from_point=None*, *section=None*, *material=None*, *insulation=None*, *code=None*)

Initialize self. See help(type(self)) for accurate signature.

Properties

property dx

Get and set the vertex x-coordinate of the 'to' point.

property dy

Get and set the vertex y-coordinate of the 'to' point.

property dz

Get and set the vertex z-coordinate of the 'to' point.

property length

Get and set the length of the element.

5.5.2 ElementContainer

class psi.elements.**ElementContainer**

Methods

5.6 psi.sections

5.6.1 Pipe

class psi.sections.**Pipe**(*name*, *od*, *thk*, *corra=None*, *milltol=None*)

Methods

__init__(*name*, *od*, *thk*, *corra=None*, *milltol=None*)

Create a pipe object.

Parameters

- **name** (*str*) – Unique name for pipe instance.
- **od** (*float*) – Actual outer diameter of pipe, not to be confused with the nomimal od. Note that for 14 inch pipe and above, the actual and nominal diameters are the same.
- **thk** (*float*) – Pipe wall thickness, ie. the nominal thickness.

- **corro** (*float*) – Corrosion allowance (CA).

  For systems that transport corrosive substances, the pipe wall may errode over the course of time. Typically, this type of corrosion is added on when the minimum design thickness is determined for pressure retention. Systems that experience significant corrosion are far more prone to fatigue related failures. A typical value for CA is 0.062 inches for piping.

- **milltol** (*float*) – Mill tolerance.

  When a mandrel is pushed through a hot billet to create the "hole" in a seamless pipe, a variation in thickness may occur if the mandrel is off-course. This tolerance, which is (+/-) 12.5% is typically accounted for in the min wall calculation per the applicable piping code so the user need not put in a value unless otherwise required. It does not apply to welded pipes which need to consider the weld joint efficiency factor since steel plates are easier to manufacture with high accuracy.

  The specified mill tolerance is typically used to calculate the hoop stress of the pipe depending on the code and has not bearing on the stiffeness calculation of the element.

  Similarly to the corrosion allowable, the milltol is accounted for when the minimum pipe wall calculations are performed. Therefore, by default both the CA and milltol values are set to None.

Example

Create a 10" schedule 40 pipe and activate the section.

```
>>> p1 = Pipe("p1", 10.75, 0.365)
```

classmethod from_file(*name*, *nps*, *sch*, *corra=None*, *milltol=None*, *fname=None*, *default_units='english'*)

Create a pipe object from a csv data file.

Parameters

- **name** (*str*) – Unique name for pipe object.
- **nps** (*str*) – Nominal pipe size.
- **sch** (*str*) – Pipe schedule designation.
- **corro** (*float*) – Corrosion allowance.
- **milltol** (*float*) – Mill tolerance. Enter the percent value such as 12.5.
- **fname** (*str*) – Full path to the csv data file used to do the lookup.
- **default_units** (*str*) – The units used for the data. Must be one of the units defined in the psi.UNITS_DIRECTORY path.

Example

Create a 10” schedule 40 pipe and activate the section.

```
>>> p1 = Pipe.from_file("p1", "10", "40")
```

Attributes

thke
The effective wall thickness used for stress calcultions when the corrosion allowance and mill tolerance are taken into consideration. Both the pressure and bending stresses are affected, if the code calls for it. For example B31.1 does not require for the reduced effective thickness be used for stress calculation.

Only used for code stress calculations and as a result will generate higher stresses. The reduced wall is not used for pipe weight or stiffness calculations, since a lighter and more flexible wall will result in lower loads and ultimately give less conservative stress values. In other words, the actual thickness is used for the moment of inertia and area of pipe formulations.

Depending on the code, the mill tolerance may be included for hoop stress calculation only to ensure the proper min wall was specified by the designer.

nps
The nominal pipe diameter is used to determine the bend radius for a pipe bend.

Note that the actual and nominal diameter are the same number for 14 inch pipe and large.

id_
The pipe inner diameter, not to be confused with the object id.

area
Cross sectional area of the pipe.

izz
Area moment of inertia about the local z-z axis. The horizontal bending axis.

iyy
Area moment of inertia about the local y-y axis. The vertical bending axis.

ixx
Polar moment of inertia about the longitudinal local x-axis

is_thin_wall
Check to see if the pipe is thin wall.

Per the applicable code, the pipe pressure will influence the stiffness of a thin wall pipe. This is usually taken into account by modifying the flexibility factor of the element matrix by a pressure factor.

If the od to thk ratio of a pipe is greater than 100 then the code computed sifs and flexibility factors no longer apply. This is usually true of large diameter duct piping, for which local effects have a significant influence on the overall behavior of the pipe and results.

is_heavy_wall
Check to determine if the pipe is heavy wall.

Pipes with a diameter to thickness ratio less than 10 are considered having heavy walls.

is_large_bore
Pipe sizes larger than 2” are considered large bore.

d2t
The D/t ratio of the section.

If a pipe has a D/t ratio of larger than 100 it behaves more like a shell than a beam.

Code based SIF and flexibility factors do not apply for D/t ratios above 100 due to limitations in the testing performed by Markl and company.

5.6.2 SectionContainer

class psi.sections.**SectionContainer**

Methods

apply(*inst*, *elements=None*)
Apply a section to elements.

Parameters

- **inst** (*Section*) – An instance of a section.
- **elements** (*List of Elements*) – A list of elements.

Note: If elements is None, the active set of elements is used.

Example

Create a 10” schedule 40 pipe and assign it to all active elements.

```
>>> p1 = Pipe.from_file("p1", "10", "40")
>>> sections.apply(p1)
```

5.7 psi.material

5.7.1 Material

class psi.material.**Material**(*name*)

Methods

__init__(*name*)
Create a material object.

Parameters **name** (*str*) – Unique name for material instance.

Example

Create a material and define its density.

```
>>> mat1 = Material("A53A")
>>> mat1.rho = 0.365
```

Warning: The user is responsible for entering all required material data such as sh, alp, rho, nu and ymod.

Note: A quicker and safer way to define a material is to use the method Material.from_file and to change the properties as needed.

Example

To input the material hot allowable enter a list of tuples consisting of the temperature and corresponding value:

```
>>> mat1.sh.table = [(t1, v1), (t2, v2), ..., (tn, vn)]
```

Note: Temperature related data need not be input in ascending order (i.e. t1 < t2 < tn), however it is good practice to do so for clarity.

classmethod from_file(*name*, *material*, *code*, *fname=None*, *default_units='english'*)

Create a material from a csv data file.

Parameters

- **name** (*str*) – Unique name for pipe object.
- **material** (*str*) – Name of material in database.
- **code** (*str*) – The piping code the material data comes from, 'B31.1' for example.
- **fname** (*str*) – Pull path to the csv data file used to do the lookup.

  Note: The default path is set to the variable psi.MATERIAL_DATA_FILE.

- **default_units** (*str*) – The units used for the data. Must be one of the units defined in the psi.UNITS_DIRECTORY path.

  Note: The values are converted to base units upon loading. Conversion to and from base to user units occurs on the fly.

Example

Create a A53A material instance and activate it.

```
>>> mat1 = Material.from_file("A53A", "A53A", "B31.1")
```

apply(*elements=None*)
Assign a material instance to a piping element.

activate()
Activate the material instance.

Note: All elements created after this method call will automatically be assigned the material activated.

Attributes

parent
Returns the MaterialContainer instance.

sh
Return the material hot allowable.

Note: This can mean Sh or Sm, etc, depending on the code used. It is up to the user to interpret the results accordingly.

Example

To input the hot allowable for a material:

```
>>> mat1.sh.table = [(t1, sh1), (t2, sh2), ..., (tn, shn)]
```

ymod
Return the material young's modulus.

Example

To input the thermal expansion coefficient:

```
>>> mat1.ymod.table = [(t1, ymod1), (t2, ymod2), ..., (tn, ymodn)]
```

alp
Return the material thermal expansion coefficient.

Example

To input the thermal expansion coefficient:

```
>>> mat1.alp.table = [(t1, alp1), (t2, alp2), ..., (tn, alpn)]
```

rho
Returns the material density.

Example

To input the material density.

```
>>> mat1.rho.value = 0.365
```

Note: The density of a material is not temperature dependant at the moment, therefore the value at room temperature should be used. In general the density of steels do not change drastically due to thermal effects.

nu
Returs the material poisson's ratio.

Example

To input the material poisson's ratio.

```
>>> mat1.nu.value = 0.3
```

Note: Similar to the density the poisson's ratio for steels is not strongly dependent on the temperature. Therefore a default value of 0.3 should be used.

5.7.2 MaterialContainer

class psi.material.**MaterialContainer**

Methods

apply(*inst*, *elements=None*)
Apply a material to elements.

Parameters

- **inst** (Material) – An instance of a material.
- **elements** (*List of Elements*) – A list of elements.

Note: If elements is None, the active set of elements is used.

Example

Create a material and assign it to all active elements.

```
>>> mat1 = Material.from_file("A53A", "A53A", "B31.1")
>>> mat1.apply(p1)
```

5.8 psi.supports

Implementation of different types of theoritical pipe supports.

Supports are implemented using the penalty approach where the global system stiffness and force matrices are modified by the support stiffness value and displacement respectively.

Axis aligned supports directly scale the diagonal elements of the global stiffness matrix. Inclined or skewed supports are implemented using constraint equations and modify more than the diagonal terms as the displacements are coupled via the direction cosines.

For support displacements, the support displacement vector is multiplied by the support stiffness and added to the corresponding force vector. Refer to "Introduction to Finite Elements in Engineering" by Chandrupatla and Belegundu for additional details.

A stiffness value of 1000*K, where K is the largest stiffness in the global stiffness matrix has shown to produce good results based on textbook examples. Reasonable default values for translation and rotation stiffness are specified for each support. The user can change the default values via model settings.

X, Y and Z supports are inherited from Inclined supports and can take several different forms. They can be snubbers (only active in occasional load cases), single or bi-directional, translational or rotational and/or define friction and gaps.

5.8.1 Anchor

class psi.supports.**Anchor**(*name*, *point*)

Support with all 6 degrees of freedom at a node fixed.

Methods

__init__(*name*, *point*)

Create an anchor support instance at a node point.

Parameters

- **name** (*str*) – Unique name for pipe object.
- **point** (*Point*) – Point instance where support is located.
- **translation_stiffnesss** (*float*) – Stiffness in the translational direction.
- **rotation_stiffness** (*float*) – Stiffness in the rotational direction.

apply(*element*)

Apply the support to the element.

Parameters

- **element** (*Element*) – An element object.

- **are applied to an element at a node on the element.** (*Supports*) –

Example

Create a support at node 20 of run element 20.

```
>>> run20 = elements(10, 20)
>>> anc = Anchor("anc20", 20)
>>> anc.apply([run20])
```

5.8.2 X

class psi.supports.**X**(*name*, *point*, *direction=None*, *gap=0.0*, *mu=0.0*, *is_rotational=False*, *is_snubber=False*)

Support aligned with the global x direction.

Methods

__init__(*name*, *point*, *direction=None*, *gap=0.0*, *mu=0.0*, *is_rotational=False*, *is_snubber=False*)

Create a support instance at a node point.

Parameters

- **name** (*str*) – Unique name for pipe object.
- **point** (*Point*) – Point instance where support is located.
- **direction** (*str*) – Support direction. Default is None. “+” and “-” is used to specify a directional support (non-linear).
- **mu** (*float*) – Support friction (non-linear).
- **is_rotational** (*bool*) – True if the support is a rotational restraint.
- **is_snubber** (*bool*) – True if the support is snubber.
- **translation_stiffnesss** (*float*) – Stiffness in the translational direction.
- **rotation_stiffness** (*float*) – Stiffness in the rotational direction.
- **gap** (*float*) – Support gap (non-linear).

apply(*element*)

Apply the support to the element.

Parameters

- **element** (*Element*) – An element object.
- **are applied to an element at a node on the element.** (*Supports*) –

Example

Create a support at node 20 of run element 20.

```
>>> run20 = elements(10, 20)
>>> anc = Anchor("anc20", 20)
>>> anc.apply([run20])
```

5.8.3 Y

class psi.supports.**Y**(*name*, *point*, *direction=None*, *gap=0.0*, *mu=0.0*, *is_rotational=False*, *is_snubber=False*)

Support aligned with the global y direction.

Methods

__init__(*name*, *point*, *direction=None*, *gap=0.0*, *mu=0.0*, *is_rotational=False*, *is_snubber=False*)

Create a support instance at a node point.

Parameters

- **name** (*str*) – Unique name for pipe object.
- **point** (*Point*) – Point instance where support is located.
- **direction** (*str*) – Support direction. Default is None. "+" and "-" is used to specify a directional support (non-linear).
- **mu** (*float*) – Support friction (non-linear).
- **is_rotational** (*bool*) – True if the support is a rotational restraint.
- **is_snubber** (*bool*) – True if the support is snubber.
- **translation_stiffnesss** (*float*) – Stiffness in the translational direction.
- **rotation_stiffness** (*float*) – Stiffness in the rotational direction.
- **gap** (*float*) – Support gap (non-linear).

apply(*element*)

Apply the support to the element.

Parameters

- **element** (*Element*) – An element object.
- **are applied to an element at a node on the element.** (*Supports*) –

Example

Create a support at node 20 of run element 20.

```
>>> run20 = elements(10, 20)
>>> anc = Anchor("anc20", 20)
>>> anc.apply([run20])
```

5.8.4 Z

class psi.supports.**Z**(*name*, *point*, *direction=None*, *gap=0.0*, *mu=0.0*, *is_rotational=False*, *is_snubber=False*)

Support aligned with the global z direction.

Methods

__init__(*name*, *point*, *direction=None*, *gap=0.0*, *mu=0.0*, *is_rotational=False*, *is_snubber=False*)

Create a support instance at a node point.

Parameters

- **name** (*str*) – Unique name for pipe object.
- **point** (*Point*) – Point instance where support is located.
- **direction** (*str*) – Support direction. Default is None. "+" and "-" is used to specify a directional support (non-linear).
- **mu** (*float*) – Support friction (non-linear).
- **is_rotational** (*bool*) – True if the support is a rotational restraint.
- **is_snubber** (*bool*) – True if the support is snubber.
- **translation_stiffnesss** (*float*) – Stiffness in the translational direction.
- **rotation_stiffness** (*float*) – Stiffness in the rotational direction.
- **gap** (*float*) – Support gap (non-linear).

apply(*element*)

Apply the support to the element.

Parameters

- **element** (*Element*) – An element object.
- **are applied to an element at a node on the element.** (*Supports*) –

Example

Create a support at node 20 of run element 20.

```
>>> run20 = elements(10, 20)
>>> anc = Anchor("anc20", 20)
>>> anc.apply([run20])
```

5.8.5 LineStop

class psi.supports.**LineStop**(*name*, *point*, *direction=None*, *gap=0.0*, *mu=0.0*, *is_rotational=False*, *is_snubber=False*)

Support aligned with the axial direction of the pipe.

LineStop supports are used to redirect thermal movement. They are commonly used for rack piping with expansion loops.

Warning: Implementation incomplete, do not use!

Methods

__init__(*name*, *point*, *direction=None*, *gap=0.0*, *mu=0.0*, *is_rotational=False*, *is_snubber=False*)

Create a support instance at a node point.

Parameters

- **name** (*str*) – Unique name for pipe object.
- **point** (*Point*) – Point instance where support is located.
- **direction** (*str*) – Support direction. Default is None. "+" and "-" is used to specify a directional support (non-linear).
- **mu** (*float*) – Support friction (non-linear).
- **is_rotational** (*bool*) – True if the support is a rotational restraint.
- **is_snubber** (*bool*) – True if the support is snubber.
- **translation_stiffnesss** (*float*) – Stiffness in the translational direction.
- **rotation_stiffness** (*float*) – Stiffness in the rotational direction.
- **gap** (*float*) – Support gap (non-linear).

apply(*element*)

Apply the support to the element.

Parameters

- **element** (*Element*) – An element object.
- **are applied to an element at a node on the element.** (*Supports*) –

Example

Create a support at node 20 of run element 20.

```
>>> run20 = elements(10, 20)
>>> anc = Anchor("anc20", 20)
>>> anc.apply([run20])
```

5.8.6 Guide

class psi.supports.**Guide**(*name*, *point*, *direction=None*, *gap=0.0*, *mu=0.0*, *is_rotational=False*, *is_snubber=False*)

Support perpendicular to the pipe run direction.

An exceptional case is a guided riser support which restricts movement in the horizontal plane.

> **Warning:** Implementation incomplete, do not use!

Methods

__init__(*name*, *point*, *direction=None*, *gap=0.0*, *mu=0.0*, *is_rotational=False*, *is_snubber=False*)

Create a support instance at a node point.

Parameters

- **name** (*str*) – Unique name for pipe object.
- **point** (*Point*) – Point instance where support is located.
- **direction** (*str*) – Support direction. Default is None. "+" and "-" is used to specify a directional support (non-linear).
- **mu** (*float*) – Support friction (non-linear).
- **is_rotational** (*bool*) – True if the support is a rotational restraint.
- **is_snubber** (*bool*) – True if the support is snubber.
- **translation_stiffnesss** (*float*) – Stiffness in the translational direction.
- **rotation_stiffness** (*float*) – Stiffness in the rotational direction.
- **gap** (*float*) – Support gap (non-linear).

apply(*element*)

Apply the support to the element.

Parameters

- **element** (*Element*) – An element object.
- **are applied to an element at a node on the element.** (*Supports*) –

Example

Create a support at node 20 of run element 20.

```
>>> run20 = elements(10, 20)
>>> anc = Anchor("anc20", 20)
>>> anc.apply([run20])
```

5.8.7 Spring

class psi.supports.**Spring**(*name*, *point*, *spring_rate*, *cold_load*, *variability=25*, *is_constant=False*)

> **Warning:** Implementation incomplete, do not use!

Methods

__init__(*name*, *point*, *spring_rate*, *cold_load*, *variability=25*, *is_constant=False*)
Create a support instance at a node point.

Parameters

- **name** (*str*) – Unique name for pipe object.
- **point** (*Point*) – Point instance where support is located.
- **translation_stiffnesss** (*float*) – Stiffness in the translational directions.

  Note: The default value is based on english units.

- **rotation_stiffness** (*float*) – Stiffness in the rotational directions.

  Note: The default value is based on english units.

apply(*element*)
Apply the support to the element.

Parameters

- **element** (*Element*) – An element object.
- **are applied to an element at a node on the element.** (*Supports*) –

Example

Create a support at node 20 of run element 20.

```
>>> run20 = elements(10, 20)
>>> anc = Anchor("anc20", 20)
>>> anc.apply([run20])
```

5.8.8 Displacement

class psi.supports.**Displacement**(*name*, *opercase*, *point*, *dx=None*, *dy=None*, *dz=None*, *rx=None*, *ry=None*, *rz=None*)

A displacement support.

Displacements are applied to a stiffness matrix similar to how supports are. Supports are in essence a special case with 0 movement in the direction of stiffness. Using the penalty approach, the stiffness and force terms in the global system matrix are modified.

Support displacements are associated to an operating case and typically used with a thermal case to model equipment nozzle movements.

Note: If a displacement is not explicitly defined for a particular direction, (i.e. None) the pipe is free to move in that direction.

Methods

__init__(*name*, *opercase*, *point*, *dx=None*, *dy=None*, *dz=None*, *rx=None*, *ry=None*, *rz=None*)
Create a displacement support instance.

apply(*element*)
Apply the support to the element.

Parameters

- **element** (*Element*) – An element object.
- **are applied to an element at a node on the element.** (*Supports*) –

Example

Create a support at node 20 of run element 20.

```
>>> run20 = elements(10, 20)
>>> anc = Anchor("anc20", 20)
>>> anc.apply([run20])
```

5.8.9 SupportContainer

class psi.supports.**SupportContainer**

Methods

apply(*supports=[]*, *elements=[]*)
Apply supports to elements.

A reference of the support is attached to each element, a one to one assignment.

Parameters

- **supports** (*list*) – A list of supports
- **elements** (*list*) – A list of elements. If elements is None, loads are applied to all elements.

5.9 psi.loads

Applying loads to model nodes and elements.

Example

Suppose you have a list of elements 'element_list' that you want to apply a single or multiple loads to.

To create a Thermal load for operating case 1, type:

```
>>> t1 = Thermal('T1', 1, 500)
>>> t1.apply(element_list)
```

Similarly, to add a deadweight load to all active elements corresponding to operating case 1, type:

```
>>> w1 = Weight('W1', 1)
>>> w1.apply()
```

For multiple loads use the LoadContainer apply method:

```
>>> loads.apply([L1, L2, ..., LN], element_list)
```

Warning: An element can have multiple loads of the same type however each load should be of a different operating case. This is not currently enforced by the program so an element can have two Weight loads defined for the same operating case in which case the weights will be added together.

5.9.1 Weight

class psi.loads.**Weight**(*name*, *opercase*, *gfac=1.0*)
The element deadweight load.

Includes pipe, insulation, cladding and refractory weight. The weight load is applied as an uniform load across the length of the entire element. The direction of the load vector is always down with respect to the vertical.

Methods

__init__(*name*, *opercase*, *gfac=1.0*)
The weight load for each element.

Parameters

- **name** (*str*) – Unique name for load.
- **opercase** (*int*) – Operating case the load belongs to.
- **gfac** (*float*) – The gfac is set to 1 by default for earth gravity.

apply(*elements=None*)
Apply the load to the elements.

Parameters **elements** (*list*) – A list of elements. If elements is None, load is applied to the active elements.

Properties

property parent
Returns the LoadContainer instance.

5.9.2 Pressure

class psi.loads.**Pressure**(*name*, *opercase*, *pres=0*)
The element pressure load.

The calculated sustained stress in most piping codes uses the internal pressure to calculate a primary longitudinal stress typically equal to P*D/4*t which is added to the primary bending stress.

The pressure on capped ends due to the system being closed also has the effect of pulling on the pipe axially and imparting a tensile thrust load.

Note: If the pressure stress per applicable piping code is accounted for, the pressure due to capped ends need not be considered as doing so will in effect double the pressure stress.

The pressure can have a stiffening effect on the piping system called the Bourdon effect. For large D/t ratio pipe bends with large system pressures, the effects of ovalization can be made worse due to this effect. When a straight pipe is pressurized it wants to shrink axially due to the radial growth whereas a pipe bend wants to open up.

A pressure dependent hoop stress can also be calculated and typically used for pipe wall sizing based on code requirements. The hoop stress is based on the highest pressure the system is expected to have. The final wall thickness is determined by adding the corrosion allowable, mill tolerance and also accounting for reductions due to threads. The calculated value is then added to the minimum thickess calculation and rounded up.

Methods

__init__(*name*, *opercase*, *pres=0*)
The pressure load for each element.

Parameters

- **name** (*str*) – Unique name for load.
- **opercase** (*int*) – Operating case the load belongs to.
- **pres** (*float*) – The pressure is set to 0 by default.

apply(*elements=None*)
Apply the load to the elements.

Parameters **elements** (*list*) – A list of elements. If elements is None, load is applied to the active elements.

Properties

property parent
Returns the LoadContainer instance.

5.9.3 Thermal

class psi.loads.**Thermal**(*name*, *opercase*, *temp*, *tref*)
Thermal expansion load.

Methods

__init__(*name*, *opercase*, *temp*, *tref*)
The thermal load for each element.

Parameters

- **name** (*str*) – Unique name for load.
- **opercase** (*int*) – Operating case the load belongs to.
- **temp** (*float*) – The temperature of the element(s).
- **tref** (*float*) – The reference temperature used to calculate delta T.

apply(*elements=None*)
Apply the load to the elements.

Parameters **elements** (*list*) – A list of elements. If elements is None, load is applied to the active elements.

Properties

property parent
Returns the LoadContainer instance.

5.9.4 Hydro

class psi.loads.**Hydro**(*name*, *opercase*, *pres=0*)
Hydro test pressure.

Test pressure is typically 1.5 times the design pressure. Hydro testing is typically performed in the cold installed configuration to ensure there are no leaks.

During testing, spring cans are locked using travel stops so that they behave as full Y supports. Spring can bodies are designed to support additional deadweight loads imposed during testing. Extra precaution should be taken to ensure these loads are acceptable for very large piping.

Pneumatic testing can also be used along with RT (x-ray) to avoid overloading a system. The individual spool pieces can also be tested on the factory floor by capping both ends with a blind flange and then pumping it with water.

Warning: The hydro load only accounts for the sustained pressure stress due to test pressure. A fluid weight load should also be added in conjunction to account for the mechanical loading. See code below:

```
>>> hp = Hydro('HP', 1, 200)
>>> fl = Fluid.from_file('F1', 1, "water")
>>> loads.apply([hp, fl])    # active elements
...
>>> lc1 = LoadCase('L1', 'sus', [Hydro, Fluid], [1, 1])
```

Methods

__init__(*name*, *opercase*, *pres=0*)
The pressure load for each element.

Parameters

- **name** (*str*) – Unique name for load.
- **opercase** (*int*) – Operating case the load belongs to.
- **pres** (*float*) – The pressure is set to 0 by default.

apply(*elements=None*)
Apply the load to the elements.

Parameters **elements** (*list*) – A list of elements. If elements is None, load is applied to the active elements.

Properties

property parent
Returns the LoadContainer instance.

5.9.5 Fluid

class psi.loads.**Fluid**(*name*, *opercase*, *rho*, *gfac=1.0*)
Contents load

Methods

__init__(*name*, *opercase*, *rho*, *gfac=1.0*)
Initialize self. See help(type(self)) for accurate signature.

apply(*elements=None*)
Apply the load to the elements.

Parameters **elements** (*list*) – A list of elements. If elements is None, load is applied to the active elements.

Properties

property parent
Returns the LoadContainer instance.

5.9.6 Wind

class psi.loads.**Wind**(*name*, *opercase*, *profile=[]*, *dirvec=(1, 0, 0)*, *shape=0.7*, *gelev=0*, *is_projected=True*)
Wind force applied as uniform loading.

The pressure due to wind is applied as a uniform force. It is a function of the pipe elevation. A pressure profile versus elevation table is used to determine the pressure at node i and j of a piping element. Then the computed average of the two pressures is applied over the entire element projected length as an uniform load.

Note: For more accurate results make sure to create a node anywhere the piping system crosses the ground elevation.

If the pipe elevation at a node is less than the ground elevation the pressure contribution from that node is 0. If both from and to point elevations are less than ground, both pressures are 0 and thus the average pressure is also 0.

Parameters

- **name** (*str*) – Unique name for load.
- **opercase** (*int*) – Operating case the load belongs to.
- **profile** (*list of tuples*) – Wind pressure profile.

  List of elevation versus pressure tuples given in the format [(elev1, pres1), (elev2, pres2), ... (elevn, presn)] with respect to the ground elevation.

Note: The first elevation (elev1) corresponds to the ground elevation and must be set to zero. In other words, the 0 reference of the profile is located at ground elevation. Use the Wind.gelev attribute to set the global vertical position of ground.

- **shape** (*float*) – The element wind shape factor. Set to a typical default value of 0.7.
- **dirvec** (*tuple*) – The wind vector direction given in global coordinates (x, y, z).
- **gelev** (*float*) – The ground elevation with respect to global coordinates. Elevation below which the wind pressure is zero. Default is set to 0.
- **is_projected** (*bool*) – If true, the load is applied over the projected length of the element in the respective global direction. Default is set to True.

Methods

__init__(*name*, *opercase*, *profile=[]*, *dirvec=(1, 0, 0)*, *shape=0.7*, *gelev=0*, *is_projected=True*)
Initialize self. See help(type(self)) for accurate signature.

apply(*elements=None*)
Apply the load to the elements.

Parameters **elements** (*list*) – A list of elements. If elements is None, load is applied to the active elements.

Properties

property parent
Returns the LoadContainer instance.

5.9.7 Seismic

class psi.loads.**Seismic**(*name*, *opercase*, *gx=0.0*, *gy=0.0*, *gz=0.0*, *gfac=1.0*)
Three directional seismic loading applied as a gravity (g) factor in global coordinates.

Parameters

- **name** (*str*) – Unique name for load.
- **opercase** (*int*) – Operating case the load belongs to.
- **gx** (*float*) – Seismic g load factor in the global x direction. Defaults to 0.
- **gy** (*float*) – Seismic g load factor in the global y direction. Defaults to 0.
- **gz** (*float*) – Seismic g load factor in the global z direction. Defaults to 0.
- **gfac** (*float*) – Gravity factor with 1 being earth gravity. Defaults to 1.

Methods

__init__ (*name*, *opercase*, *gx=0.0*, *gy=0.0*, *gz=0.0*, *gfac=1.0*)
The weight load for each element.

Parameters

- **name** (*str*) – Unique name for load.
- **opercase** (*int*) – Operating case the load belongs to.
- **gfac** (*float*) – The gfac is set to 1 by default for earth gravity.

apply (*elements=None*)
Apply the load to the elements.

Parameters **elements** (*list*) – A list of elements. If elements is None, load is applied to the active elements.

Properties

property parent
Returns the LoadContainer instance.

5.9.8 LoadContainer

class psi.loads.**LoadContainer**

Methods

apply (*loads=[]*, *elements=None*)
Apply loads to elements.

A reference for each load is assigned to each element.

Note: One pipe element can be assigned multiple loads of the same type. Each load must be associated with a different operating case.

Parameters

- **loads** (*list*) – A list of loads
- **elements** (*list*) – A list of elements. If elements is None, loads are applied to all active elements.

5.10 psi.loadcase

Each loadcase consists of one or many different types of loads applied to the piping system at once. For instance, the sustained case for a system without springs consists of the weight(W) case along with a pressure case(P1). Both these loads are first applied to the underlying finite element model and the solution is generated for that particular case. This is a primary load case since it is composed of primary loads.

A load combination case consists of one or more primary load cases combined using a specific combination method. The solution vectors from the primary load cases are added, subtracted, etc., in order to produce these kind of secondary load cases.

A loadcase may consist of primary loads such as Weight, Thermal, etc., or it may be a be a combination of two or more secondary (ie. result cases) combined together.

The displacement, support reactions and internal force results for the loadcase are stored internally in the load case object. Note that properties with units for different values are stored separately and combined on the fly.

5.10.1 LoadCase

class psi.loadcase.**LoadCase**(*name*, *stype='sus'*, *loadtypes=[]*, *opercases=[]*)

A set of primary load cases consisting of different types of loads and the operating case the load belongs to.

Example

Create a deadweight loadcase.

```
>>> w1 = Weight('w1', 1)
>>> p1 = Pressure('p1', 1)
...
>>> lc1 = LoadCase('lc1', 'sus', [Weight, Pressure], [1, 1])
```

Note: The weight and pressure load have been defined for operating case 1.

Attention: The loads for a given case must be unique. In other words, the same load cannot be specified twice. An equality check is performed based on the load type, name and operating case it belongs to.

Methods

__init__(*name*, *stype='sus'*, *loadtypes=[]*, *opercases=[]*)

Initialize self. See help(type(self)) for accurate signature.

Properties

property movements
Return the nodal displacement results array.

property reactions
Return the nodal reaction results array.

property forces
Return the force reaction results array.

5.10.2 LoadComb

class psi.loadcase.**LoadComb**(*name*, *stype='ope'*, *method='algebraic'*, *loadcases=[]*, *factors=[]*)
Combine primary loadcases using different combination methods.

Note: Combinations pull stored data from loadcases on the fly and do the necessary combination operations.

Attention: A loadcase and a loadcomb are derived from the same basecase and so they have the same namespace when it comes to name.

Methods

__init__(*name*, *stype='ope'*, *method='algebraic'*, *loadcases=[]*, *factors=[]*)
Create a loadcomb instance.

Parameters

- **name** (*str*) – Unique name for load combination object.
- **stype** (*str*) – Type of code stress. Defaults to sustained stress.
  - HGR - Hanger load case
  - HYD - Hydro load case
  - SUS - Sustained stress case
  - EXP - Thermal expansion stress.case
  - OCC - Occasional stress case
  - OPE - Operating stress case
  - FAT - Fatigue stress case
- **method** (*str*) – Result combination method.
  - Algebriac - Disp/force results added vectorially. Stresses are derived from the vectorially added force results.
  - Scalar - Disp/force results added vectorially similar to the algebraic method. Stresses are added together.
  - SRSS - Square root of the sum squared. Direction independant.
  - Abs - Absolute summation.

  - Signmax - Signed max.
  - Signmin - Signed min.
- **loadcases** (*list of loadcases.*) – List of load cases.
- **factors** (*list of numbers.*) – A list of factors corresponding to each loadcase.

Note: If a factor is not given, a default value of 1 is used. Also, the number of factors must match the number of loadcases.

Properties

property movements
Return the combined nodal displacement array.

property reactions
Return the combined nodal reaction array.

property forces
Return the combined nodal forces array.

5.10.3 LoadCaseContainer

class psi.loadcase.**LoadCaseContainer**

Methods

5.11 psi.reports

Display various result reports generated from the loadcases solved.

5.11.1 Movements

class psi.reports.**Movements**(*name*, *loadcases*)
Nodal displacement results.

Methods

__init__(*name*, *loadcases*)
Create a movement report instance.

Parameters

- **name** (*str*) – Unique name for report object.
- **loadcases** (*list of loadcases*) – Loadcases for which results are displayed.

to_screen()
Print movement report results to screen.

5.11.2 Reactions

class psi.reports.**Reactions**(*name*, *loadcases*)
Support reaction results.

Methods

__init__(*name*, *loadcases*)
Create a reactions report instance.

Parameters

- **name** (*str*) – Unique name for report object.
- **loadcases** (*list of loadcases*) – Loadcases for which results are displayed.

to_screen()
Print reaction report results to screen.

5.11.3 Forces

class psi.reports.**Forces**(*name*, *loadcases*)
Internal forces results.

Methods

__init__(*name*, *loadcases*)
Create a forces report instance.

Parameters

- **name** (*str*) – Unique name for report object.
- **loadcases** (*list of loadcases*) – Loadcases for which results are displayed.

to_screen()
Print forces report results to screen.

CHAPTER

SIX

DEVELOPER GUIDE

Welcome to the PSI Developer Guide!

If you find a bug or a problem with the documentation, please open an issue with the issue tracker. If you have a question about developing with PSI, or you wish to contribute, please join the mailing list or the discord server.

For license questions, please contact Denis Gomes, the primary author.

6.1 Overview

Under Construction!

CHAPTER

SEVEN

V & V

Welcome to the PSI Verification and Validation Manual!

If you find a bug or a problem with the documentation, please open an issue with the issue tracker. If you have a question about developing with PSI, or you wish to contribute, please join the mailing list or the discord server.

For license questions, please contact Denis Gomes, the primary author.

7.1 Problem 1

Under Construction!

CHAPTER

EIGHT

FAQS

1. Is PSI free to use for commercial use?

   Yes, PSI can be used for commercial work. Note however, if the source code is modified, the altered source must be released under the same GPL license. Read the PSI *license*.

CHAPTER

NINE

PREFACE

Pipe Stress Infinity (PSI) is an engineering design and analysis software used to evaluate the structural behavior and stresses of piping systems to a variety of different codes and standards. *Read more...*

CHAPTER

TEN

QUICK REFERENCE

The PSI Quick Reference is a good starting point for new users to get up and going quickly. It provides a basic introduction to the project and a small taste of the program's capabilities.

If this is your first time reading about PSI, we suggest you start at the *Example* section.

CHAPTER

ELEVEN

USER GUIDE

The PSI User Guide contains general roadmaps, much of the theory behind the program and other user related topics.

CHAPTER

TWELVE

APPLICATION GUIDE

The PSI Application Guide consists of tutorials, examples and various modeling and analysis techniques. It also provides in-depth documentation for writing applications using PSI. Many topics described here reference the PSI API reference, which is listed below.

CHAPTER

THIRTEEN

API REFERENCE

The PSI Application Programming Interface (API) provides the reference for the publically available classes, methods and function calls. It is automatically generated from the source code documentation and can be used by developers and users alike to gain a deeper understanding of the algorithms and programming techinques used.

CHAPTER

FOURTEEN

DEVELOPER GUIDE

These documents describe details on how to develop PSI itself further. Read these to get more detailed insights into how PSI is designed, and how to help its future development.

CHAPTER

FIFTEEN

V & V

A list of example problems used for program/machine verification and validation.

CHAPTER

SIXTEEN

FAQS

A list of frequently asked questions raised on the mailing list or the discord server.

PYTHON MODULE INDEX

p

INDEX

Symbols

A

C

D

E

F

G

H

I

L

M

N

P

R

S

T

U

W

X

Y

Z