web: pr-dc.com, email: info@pr-dc.com, github: github.com/PR-DC

JSLAB v1.0.3

DOCUMENTATION

github.com/PR-DC/JSLAB

PRDC d.o.o., Novo naselje bb, 22310 Šimanovci, Republic of Serbia

Contents

1 About JSLAB

Welcome to the **JSLAB** Documentation! This guide provides comprehensive information for users and contributors, detailing the features, usage, and contribution guidelines to ensure a consistent and high-quality codebase.

Figure 1 - JSLAB logo

The **JavaScript Laboratory (JSLAB)** is an open-source environment designed for scientific computing, data visualization, and various other computer operations. Inspired by *GNU Octave* and *Matlab*, **JSLAB** leverages the advantages of JavaScript, including its blazing speed, extensive examples, backing by some of the largest software companies globally, and the vast community of active programmers and software engineers.

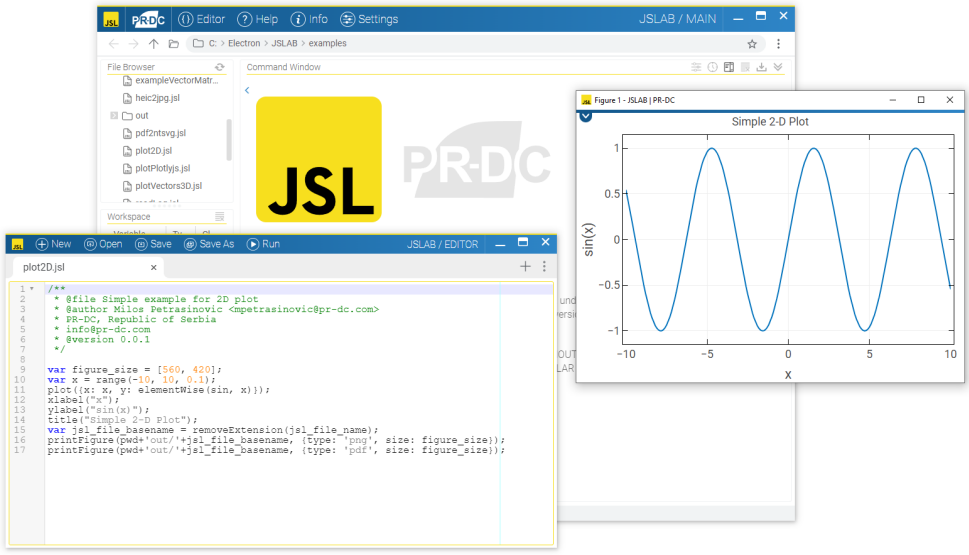

Figure 2 - JavaScript Laboratory (JSLAB) GUI

The program was developed to fulfill the need for performing calculations in a programming language that allows for code reuse in later project stages. JavaScript was chosen for its speed, dynamic nature, interpretability, extensive library support, large

existing codebase, backing by major software companies, and the ability to create both desktop and mobile applications.

JSLAB offers a streamlined, dual-window interface designed to boost productivity and foster innovation. The main window combines a versatile workspace with a sandbox terminal, allowing users to run, test, and iterate on code in real time. The dedicated editor window introduces the **.JSL file format**—a plain text format tailored for **JSLAB** scripts. With advanced linting and intelligent autocompletion, the editor makes it easy to write precise, reusable code with minimal errors.

1.1 Why Choose JSLAB?

- **Backed by Leading Investments**

  JavaScript is supported by major industry investments, ensuring continuous innovation and robust development. Our commitment to excellence makes **JSLAB** a trusted choice for professionals and organizations worldwide.

- **Powered by JavaScript, Trusted by Giants**

  Join the ranks of top companies who leverage JavaScript for their mission-critical applications. With **JSLAB**, you benefit from the same reliable and scalable technology that powers some of the most advanced projects on Earth and beyond.

- **Thriving Community and Massive User Base**

  Become part of a vibrant and growing community of JavaScript developter. Extensive support network and active forums ensure you always have the resources and assistance you need to succeed.

- **Comprehensive Functionality Comparable to Leading Tools**

  JSLAB bridges the gap between JavaScript and specialized scientific tools. Enjoy functionalities equivalent to MATLAB, GNU Octave, Python, R, and Julia, all within a single, unified platform. Perform data analysis, machine learning, numerical computations, and more with ease.

- **Seamless and Native GUI with HTML, CSS, and SVG**

  Design intuitive and visually appealing graphical user interfaces using native HTML, CSS, and SVG. Create interactive dashboards, custom visualizations, and responsive layouts without the need for additional frameworks.

- **Extend with Native Modules via NPM and C++/C**

  Enhance **JSLAB**'s capabilities by integrating native modules from npm, built with C++ and C. Tap into a vast ecosystem of extensions and customize your environment to meet your specific needs, ensuring maximum performance and flexibility.

- **Join the JSLAB Revolution Today!**

  Experience the seamless integration of powerful scientific computing and the flexibility of JavaScript. Whether you're developing complex algorithms, analyzing vast datasets, or creating innovative applications, **JSLAB** empowers you to achieve more.

1.2 License

2 About documentation

This documentation serves as a comprehensive guide for both users and contributors of the **JSLAB** Library. It covers an introduction to the project, detailed features, installation and setup instructions, user interface overview, practical examples, coding standards, build instructions, contribution guidelines, and mechanisms for providing feedback.

The documentation is structured to provide clear and detailed information, ensuring that both new users and seasoned contributors can effectively utilize and contribute to the **JSLAB** project.

3 Coding style

3.1 Documentation and Comments

Clear documentation is essential for maintaining and understanding the codebase. We utilize JSDoc for structured documentation and inline comments to clarify complex logic.

- **JSDoc:** Use JSDoc comments to document files, classes, methods, parameters, and return values.
- **Inline Comments:** Add comments to explain non-trivial code segments.

```
/**
* Constructs the JSLAB library environment.
* @param {Object} config Configuration options.
*/
constructor(config) {
  // Initialize properties
  this.config = config;
  // ... other initializations
}
```

3.2 Naming Conventions

- **Classes and Constants:** Use uppercase letters with underscores (e.g., `PRDC_JSLAB_LIB`).
- **Functions and Methods:** Use camelCase (e.g., `getFullFilePath`).
- **Variables:** Use lowercase letters with underscores (e.g., `new_file_path`).
- **Meaningful Names:** Choose self-explanatory names that convey the purpose clearly.

3.3 Code Structure and Modularization

- **Separation of Concerns:** Organize code into separate modules/files based on functionality.
- **Object-Oriented Design:** Use ES6 classes to encapsulate related properties and methods.

```
const { PRDC_JSLAB_EVAL } = require('./jslab-eval');

class PRDC_JSLAB_LIB {
  constructor(config) {
  this.eval = new PRDC_JSLAB_EVAL(this);
  // ... other initializations
  }
}
```

3.4 Error Handling

- **Try-Catch Blocks:** Use try-catch to handle potential errors gracefully.
- **Custom Errors:** Throw custom error objects for specific error scenarios.

```
try {
  return this.resolve(file_path);
} catch {
  this.jsl.env.error('Path resolution failed.');
}
```

3.5 State Management and Cleanup

- **Resource Tracking:** Maintain arrays and objects to track active asynchronous operations.
- **Cleanup Methods:** Implement methods to clear resources and reset the environment state.

3.6 Configuration Management

- **Centralized Configuration:** Use a dedicated class (`PRDC_APP_CONFIG`) to manage all configuration settings.
- **Conditional Configurations:** Adjust settings based on the runtime environment or command-line arguments.

3.7 Best Practices

- **Consistent Formatting:** Use a consistent code formatter to maintain uniform code style.
- **Meaningful Commit Messages:** Write clear and descriptive commit messages that explain the purpose of the changes.
- **Modular Code:** Write reusable and modular code to enhance maintainability and scalability.
- **Comprehensive Testing:** Implement thorough tests to ensure the reliability of your contributions.

4 Installation

You can install **JSLAB** by either downloading the latest stable release from GitHub or by building it from source. Choose the method that best fits your needs.

4.1 Download the Latest Stable Release

- Visit the JSLAB Releases Page on GitHub Repository:

  `https://github.com/PR-DC/JSLAB/releases`
- Download the appropriate installer and install program.
- Try examples from:

  `https://github.com/PR-DC/JSLAB/tree/master/examples`

4.2 Build from Source

If you prefer to build **JSLAB** from source, follow the detailed build instructions available in this documentation.

5 Build instructions

5.1 Prerequisites

In order to download necessary tools, clone the repository, and install dependencies via npm, you need network access.

- **Node.js:** Ensure that Node.js is installed on your system. You can download it from the official website:

  `https://nodejs.org/`

- **npm:** npm is typically installed alongside Node.js.

- **node-gyp:** node-gyp is installed alongside with application but it requires additional tools and libraries depending on your operating system. Follow the instructions for your specific OS from:

  `https://github.com/nodejs/node-gyp`

- **Git:** Suggested for cloning the repository. Download it from the official website:

  `https://git-scm.com/`

5.2 Installation Steps

1. Clone the JSLAB repository:

   ```
   git clone git clone https://github.com/PR-DC/JSLAB.git
   ```

2. Navigate to the project directory:

   ```
   cd JSLAB
   ```

3. Install the necessary dependencies:

   ```
   npm install
   ```

4. Start the application:

   ```
   npm start
   ```

5. Check examples from:

   `https://github.com/PR-DC/JSLAB/tree/master/examples`

6 Getting Started

6.1 Main Window

The image below shows the main window of the **JSLAB** program.

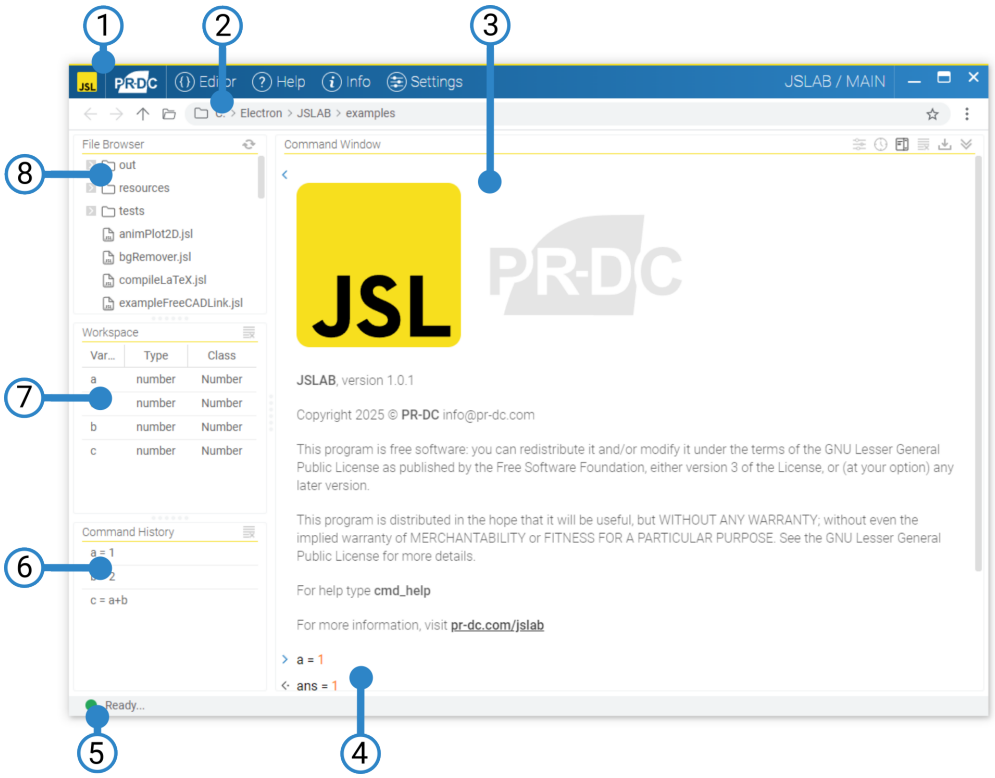

Figure 3 - Main Window of JSLAB

Within the main window, there are various shortcuts defined in the help window—for example, the `[CTRL] + [H]` shortcut opens the help window.

In the image above, the following elements of the main window are numbered:

1. **Main Window Header**
   A menu with options for opening the code editor window, viewing help, accessing program information, and opening the program settings (such as language settings; currently, Serbian in both Latin and Cyrillic scripts and English are available).

2. **Workspace Navigation**
   Icons for navigating through folders and opening a folder selection window. Besides the current workspace folder, you can save additional paths that will be used when running scripts.

3. **Command Window**
   Located in the central right panel, this window displays messages from the workspace after code execution.

4. **Command Input Field**
   Located at the bottom of the central right panel, this field is used to send commands

to the workspace. As you type, suggestions for completing the command automatically appear based on the currently active workspace. You can also navigate through the history of entered commands. After typing a command and pressing `[ENTER]`, the command is executed in the workspace.

5. **Status Bar**
   A bar at the bottom of the window displaying the current state of the workspace. In the bottom left corner, an icon changes color based on that state; clicking the icon displays a tooltip with information about what is currently active in the workspace.

6. **Command History**
   Located in the lower part of the left central panel, this section allows you to track the sequence of executed commands and easily re-execute a command (by double-clicking on it).

7. **Workspace Variables**
   In the central part of the left panel, this area displays the name, type, and class of each active global variable created by the user in the workspace.

8. **File Browser**
   Located in the upper part of the left panel, this area shows folders and files, which can be directly opened within the editor.

Note: The command window is ideal for entering a few commands. However, for more complex tasks, scripts are used.

6.2 Editor Window

For **JSLAB** scripts, the `.jsl` extension is used to distinguish them from the standard `.js` extension for JavaScript. Scripts are executed by providing the script path as the sole argument to the `run` function. For writing or editing code in a script, it is best to use the built-in code editor (which can be opened with the `editor` command or the `edit()` function). The code editor window is shown in the image below.

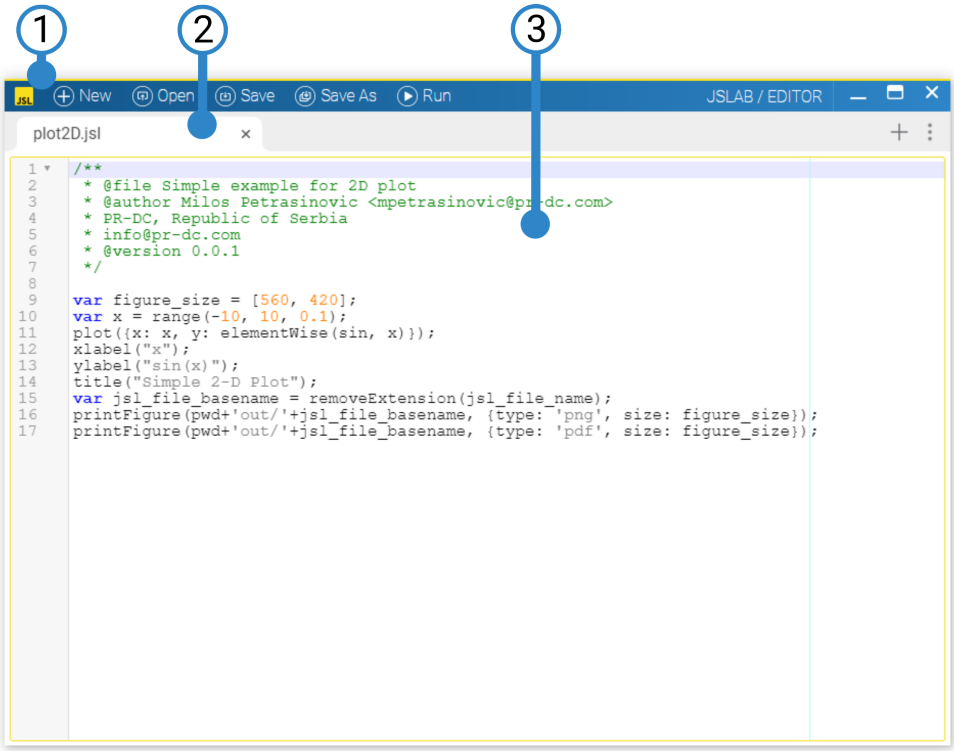

Figure 4 - Editor Window of JSLAB

In the image above, the following elements of the code editor window are numbered:

1. **Editor Window Header**
   At the very top of the window, there is a menu with options for creating new scripts, opening existing ones, saving open scripts, and directly executing scripts in the workspace.

2. **Script Tabs**
   Located just below the header, these tabs allow you to switch between the active scripts being edited and to open or close scripts.

3. **Script Text Editor**
   This is the area where the script code is displayed. As you type, suggestions for completing commands automatically appear based on the currently active workspace—similar to the command window. This text editor provides many advanced features for working with code, including syntax checking, suggestions, the ability to collapse certain code blocks, and highlighting of the current active line.

The editor also includes additional advanced features for code input, such as searching through the code and performing text replacements (the search popup can be opened by pressing `[CTRL] + [F]`).

Most importantly, code can be executed in two ways:

- By entering commands in the command input field of the command window.
- By running a script using the `run()` function or directly from the editor window.

6.3 Programming

Programming in **JSLAB** is based on the full feature set of **JavaScript** and **Node.js**, combined with **HTML** and **CSS** for building graphical user interfaces (GUI). The application also provides a range of **built-in functions** that simplify certain tasks—like plotting with the `plot()` function. All implemented functions and classes are documented for easy reference.

6.3.1 Accessing Documentation

Multiple approaches are available to view documentation directly from **JSLAB**:

- **Built-in help functions:** `help()`, `doc()`, `documentation()`, `helpSearch()`, `docSearch()`, `documentationSearch()`
- **HTML documentation:** Openable via the `openDoc()` or `openDocumentation()` functions
- **Function source code:** See the implementation of a built-in function with `source("functionName")` (e.g., `source("plot")`)

6.3.2 Working with Scripts and Examples

JSLAB scripts typically use a `.jsl` extension to distinguish them from standard `.js` files. You can run scripts from the command window with `run("scriptPath")`, or open them in the built-in editor (using the `editor` command or the `edit("scriptName.jsl")` function).

For bundled example scripts:

- **List examples:** `getExamples()`
- **Open a specific example:** `openExample("exampleName")`
- **Open the examples folder:** `openExamplesFolder()`
- **Navigate to the examples folder:** `goToExamplesFolder()`

6.3.3 Resetting the Environment

- **Reset workspace (sandbox):** `resetSandbox()` clears current workspace variables.
- **Reset the entire application:** `resetApp()` returns the application to its initial state.

6.3.4 Advanced Chromium Tools

Since **JSLAB** is built on **Chromium**, you can leverage its powerful developer tools for debugging and performance analysis:

- `openDevTools()` : Open Developer Tools
- `openWindowDevTools()` : Open Window Developer Tools

The **Performance** tab in the developer tools is particularly useful for optimizing your scripts, helping to identify potential slow-downs and bottlenecks.

7 Examples

7.1 Animated 2D Plot

A 2D plot animation like this is essential for visualizing real-time data changes, enabling dynamic tracking of evolving values and providing immediate insight into trends or fluctuations as they happen.

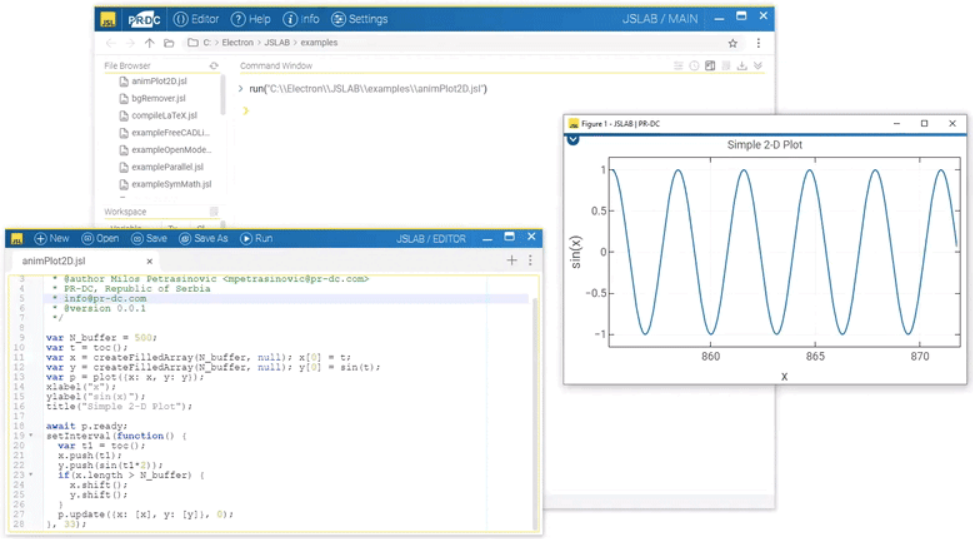

Figure 5 - Animated 2D Plot

```
// JavaScript Example
var N_buffer = 500;
var t = toc();
var x = createFilledArray(N_buffer, null); x[0] = t;
var y = createFilledArray(N_buffer, null); y[0] = sin(t);
var p = plot({x: x, y: y});
xlabel("x");
ylabel("sin(x)");
title("Simple 2-D Plot");

await p.ready;
```

```
setInterval(function() {
  var t1 = toc();
  x.push(t1);
  y.push(sin(t1*2));
  if(x.length > N_buffer) {
    x.shift();
    y.shift();
  }
  p.update({x: [x], y: [y]}, 0);
}, 33);
```

7.2 3D Plot with Vectors

3D plots are essential for illustrating spatial relationships and complex vector interactions, allowing for a deeper understanding of data across three dimensions.

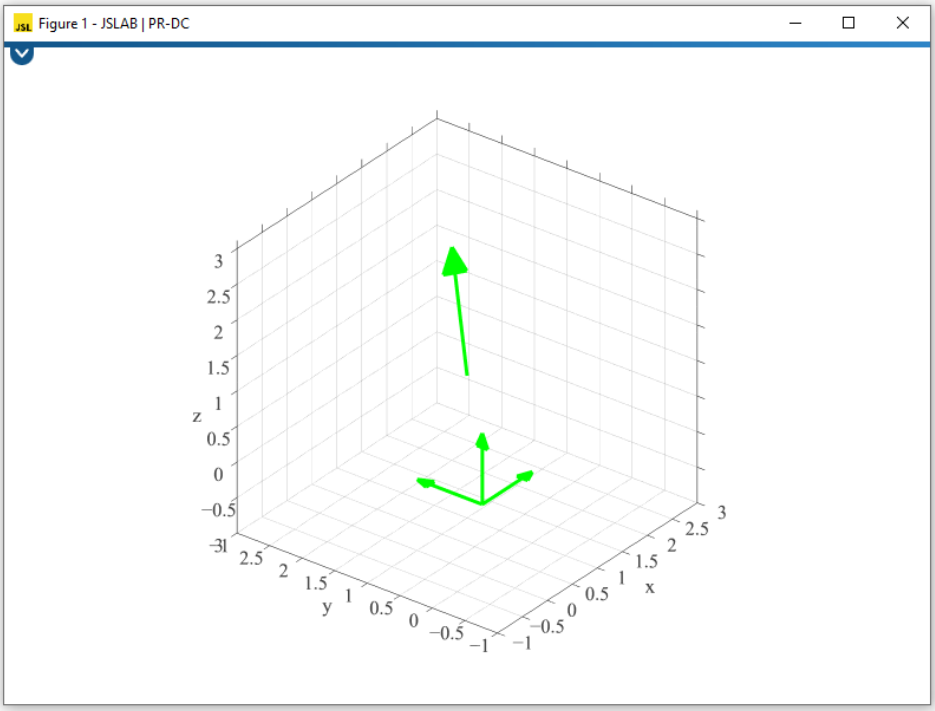

Figure 6 - 3D Plot with Vectors

```
// JavaScript Example
var x = [0, 0, 0, 1, 0];
var y = [0, 0, 0, 1, 0];
var z = [0, 0, 0, 1, 0];

var u = [1, 0, 0, 1, -1];
var v = [0, 1, 0, 1, 0];
var w = [0, 0, 1, 1, 0];

var head_scale = 0.2;
```

```
var head_angleFactor = 0.4;

var vectors = createVectors3D(x, y, z, u, v, w, head_scale, head_angleFactor,
    {color: "#0f0", width: 6});

figure(1);
plot([
  vectors.line, vectors.head
], {'showLegend': false, 'font': {family: 'LatinModern', size: 14}});
xlabel("x");
ylabel("y");
zlabel("z");
xlim([-1, 3]);
ylim([-1, 3]);
zlim([-1, 3]);
```

7.3 3D Graphics

3D graphics are vital for creating immersive visualizations that bring complex structures and spatial relationships to life, enabling a more intuitive understanding and interaction with digital models in fields like simulation, design, and data analysis.

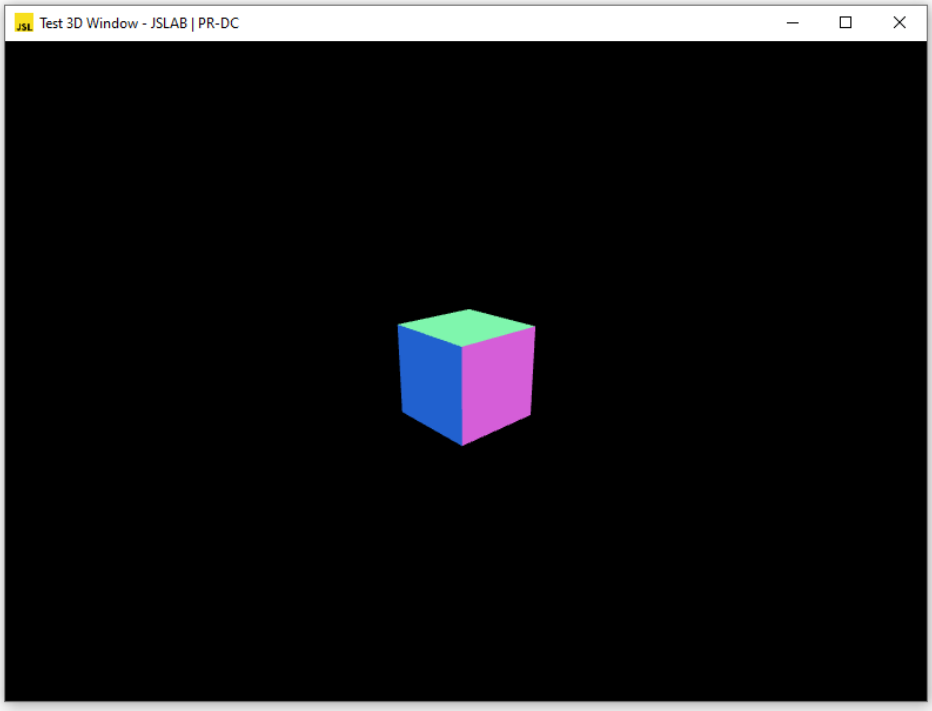

Figure 7 - 3D Graphics Example

```
// JavaScript Example
var win = await openWindow3D();
win.document.title = "Test 3D Window - JSLAB | PR-DC";
```

```
var THREE = win.THREE;

const width = win.innerWidth, height = win.innerHeight;

// init
const camera = new THREE.PerspectiveCamera( 70, width / height, 0.01, 10 );
camera.position.z = 1;

const scene = new THREE.Scene();

const geometry = new THREE.BoxGeometry( 0.2, 0.2, 0.2 );
const material = new THREE.MeshNormalMaterial();

const mesh = new THREE.Mesh( geometry, material );
scene.add( mesh );

const renderer = new THREE.WebGLRenderer( { antialias: true } );
renderer.setSize( width, height );
renderer.setAnimationLoop( animate );
win.document.body.appendChild( renderer.domElement );

// Handle window resizing
window.addEventListener('resize', onWindowResize, false);

function onWindowResize() {
  camera.aspect = win.innerWidth / win.innerHeight;
  camera.updateProjectionMatrix();
  renderer.setSize(win.innerWidth, win.innerHeight);
}

function animate( time ) {
  mesh.rotation.x = time / 2000;
  mesh.rotation.y = time / 1000;

  renderer.render( scene, camera );
}
```

7.4 Parallel Execution

Parallel execution is critical for handling computationally intensive tasks, as it allows multiple operations to run simultaneously, significantly reducing processing time and improving efficiency by utilizing all available CPU cores.

```
// JavaScript Example
var computeSquare = (i) => i * i;

// Run parallel exectuion
var results = await parallel.parfor(0, 20, 1,
  parallel.getProcessorsNum(), {}, undefined, computeSquare);
disp(results);
```

7.5 Vector and Matrix Operations

Vector and matrix operations are fundamental for efficiently performing complex mathematical computations in fields like physics, engineering, and computer graphics, enabling quick transformations, optimizations, and solutions in multidimensional spaces.

```
// JavaScript Example
var v1 = vec.new(1, 2, 3);
var v2 = vec.new([4, 8, 6]);
const v_cross = v1.cross(v2);

var A = mat.new([
    [1, 2],
    [3, 4]
]);
const b = mat.new([
    [5],
    [11]
]);
const x = A.linsolve(b);
disp('Solution to linear system A * x = b:');
disp(x);
```

7.6 Symbolic Math

Symbolic math computations are essential for achieving high precision in mathematical modeling, automating algebraic simplifications, and enabling dynamic formula manipulation, which enhances the accuracy and functionality of tools in scientific, engineering, and educational software.

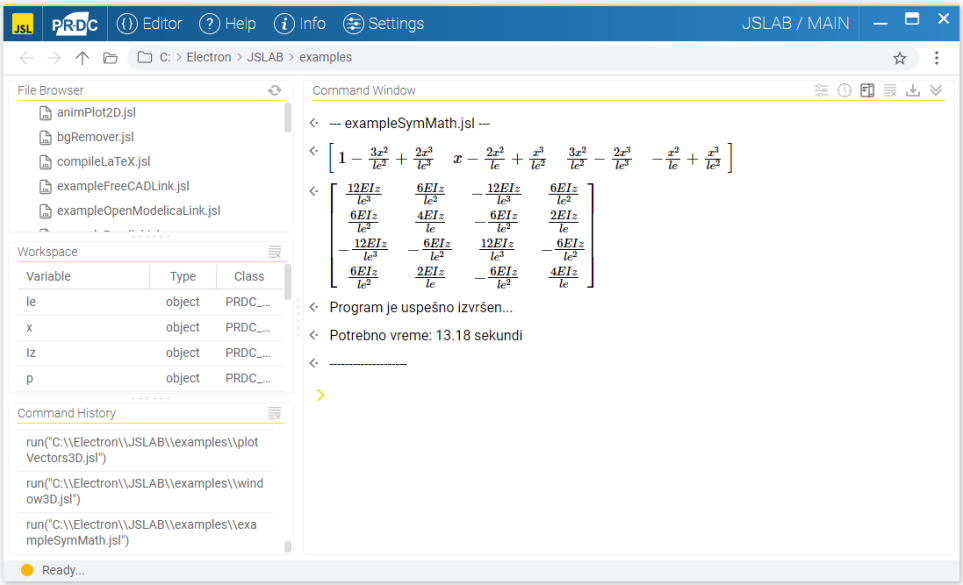

Figure 8 - Symbolic Math Example

```
// JavaScript Example
var le, x, E, Iz;
var p, P, invP, N, d2N;
var k_int, k_e_stretching, k_e_torsion;
var xi = range(0, 1, 0.01);

await sym.load();
[le, x, E, Iz] = sym.syms(['le', 'x', 'E', 'Iz']);

P = sym.mat([
  [1, 0, 0, 0],
  [0, 1, 0, 0],
  [1, le, sym.pow(le, 2), sym.pow(le, 3)],
  [0, 1, sym.mul(2, le), sym.mul(3, sym.pow(le, 2))]
]);
p = sym.mat([[1, x, sym.pow(x, 2), sym.pow(x, 3)]]);

invP = sym.inv(P);
N = sym.mul(p, invP);
d2N = sym.diff(N, 'x', 2);

k_int = sym.mul(E, Iz, sym.intg(sym.mul(sym.transp(d2N), d2N), x, [0, le]));

Ni = sym.subs(sym.subs(N, le, 1), x, xi).toNumeric();
var N_flat = Ni.flat();

sym.showLatex(N);
sym.showLatex(k_int);
```

7.7 FreeCAD Link

Integration with FreeCAD is essential for enabling automated, precise 3D modeling workflows within applications, allowing complex geometries, structures, and engineering designs to be generated, modified, and visualized programmatically, which significantly enhances productivity in design and simulation processes.

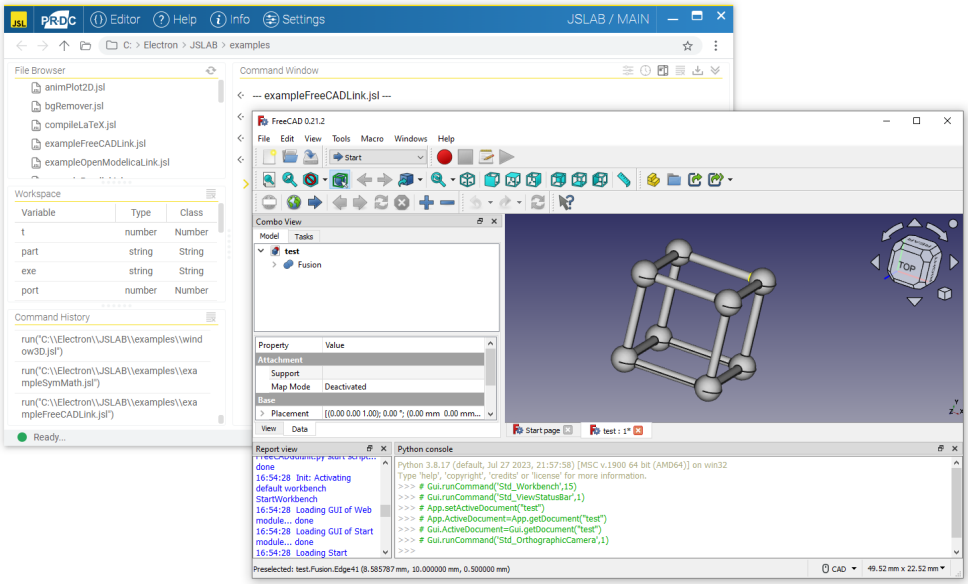

Figure 9 - FreeCAD Link Example

```
// JavaScript Example
var nodes = [
  [0, 0, 0],
  [0, 10, 0],
  [10, 10, 0],
  [10, 0, 0],
  [0, 0, 10],
  [0, 10, 10],
  [10, 10, 10],
  [10, 0, 10]
];
var D = createFilledArray(nodes.length, 3);

var lines = [];
for(var i = 0; i < 4; i++) {
  var j = i+1;
  if(i == 3) {
    j = 0;
  }
  lines.push([...nodes[i], ...nodes[j]]);
  lines.push([...nodes[i+4], ...nodes[j+4]]);
  lines.push([...nodes[i], ...nodes[i+4]]);
}
var d = createFilledArray(lines.length, 1);

// Generate JSON
var nodesFile = pwd + 'out/nodes.json';
var data = {
  'Coordinates': nodes,
  'Diameters': D
};
writeFile(nodesFile, stringify(data));
var data = {
```

```
  'Coordinates': lines,
  'Diameters': d
};
beamsFile = pwd + 'out/beams.json';
writeFile(beamsFile, stringify(data));

// Run FreeCADLink
await freecad_link.start(exe, {
  port: port,
  host: host,
  timeout: timeout,
  startup_timeout: startup_timeout
}); // Start FreeCAD program

await freecad_link.newDocument(part);
await freecad_link.callScript('MakeNodes', nodesFile, timeout);
await freecad_link.callScript('MakeBeams', beamsFile, timeout);
await freecad_link.callScript('MakeFusion', [], timeout);
await freecad_link.saveAs(model, timeout);
//await freecad_link.quit(); // Close program

deleteFile(nodesFile);
deleteFile(beamsFile);
```

7.8 OpenModelica Link

Integration with OpenModelica is crucial for enabling advanced simulation and analysis of complex dynamic systems directly within applications, allowing engineers to model, test, and optimize system behavior seamlessly, which enhances efficiency in design and validation processes.

```
// JavaScript Example
await om_link.start(exe); // Start OpenModelica program
disp(await om_link.sendExpression('getVersion()'));

disp(await om_link.sendExpression("model a end a;"));
disp(await om_link.sendExpression('loadFile("'+model+'")'));
disp(await om_link.sendExpression("getClassNames()"));
disp(await om_link.sendExpression("simulate(BouncingBall)"));
await om_link.close();
```

8 Contributing

8.1 Setting Up the Development Environment

Follow the detailed build instructions available in this documentation.

8.2 Making Changes

Follow the coding style and best practices available in this documentation.

8.3 Submitting Changes

1. Create a new branch for your feature or bugfix:

```
git checkout -b feature/your-feature-name
```

2. Make your changes and commit them with clear messages:

```
git commit -m "Add feature X to improve Y"
```

3. Push your branch to your forked repository:

```
git push origin feature/your-feature-name
```

4. Submit a Pull Request (PR) detailing your changes.

8.4 Testing

Before submitting a PR, ensure that all tests pass and add new tests for any new functionality you introduce.

8.5 Reviewing Process

All PRs are subject to review by the maintainers. Be prepared to make revisions based on feedback to align with project standards.

8.6 Best Practices

- **Consistent Formatting:** Use a consistent code formatter (e.g., Prettier) to maintain uniform code style.
- **Meaningful Commit Messages:** Write clear and descriptive commit messages that explain the purpose of the changes.
- **Modular Code:** Write reusable and modular code to enhance maintainability and scalability.
- **Comprehensive Testing:** Implement thorough tests to ensure the reliability of your contributions.

9 Feedback

Your feedback is invaluable in improving the **JSLAB** application. Whether you encounter bugs, have feature requests, or need assistance, please reach out through the following channels:

- **GitHub Issues:** Report bugs or suggest features by opening an issue in the GitHub repository.

- **Email**: Contact us directly at info@pr-dc.com or main author at mpetrasinovic@pr-dc.com.

We encourage active participation and appreciate all forms of feedback that help us enhance the functionality and usability of **JSLAB**.

10 Code references

10.1 basic

`ans`

Type: `<Unknown>`
Stores the result of the last evaluated expression.

`clear()`

Clears all defined variables in the current context.

`clc()`

Clears the console screen.

`cls()`

Clears the console screen. Alias for `clc`.

`version`

Type: `<Unknown>`
Returns the current version of the JSLAB.

`platform`

Type: `<Unknown>`
Returns the platform on which JSLAB is running.

`jsl_file_name`

Type: `<Unknown>`
Returns the file name of the current JSLAB script.

`info()`

Provides information about the current environment.

`settings()`

Accesses or modifies the user settings for JSLAB.

`cmd_help()`

Provides help information for JSLAB commands.

`editor()`

Accesses the code editor interface within JSLAB.

`pwd`

Type: `<Unknown>`
Returns the current working directory.

`breakpoint()`

Sets a breakpoint in the code for debugging.

`debug_flag`

Type: `<Unknown>`
Returns the current debug flag status.

`debug`

Type: `<Unknown>`
Enables or disables debug mode.

`pause()`

Pauses the execution of the current script.

`stoppoint()`

Sets a stop point in the script execution.

`logpoint()`

Sets a log point to record information during execution.

`updatepoint()`

Updates specific points in the script during execution.

`checkStop()`

Checks if the execution should stop based on conditions.

`endPoint()`

Marks the endpoint of a script or process.

`run(script_path, lines, [silent], [force_run])`

Parameters:

- `script_path <string>`: The path to the script to run.
- `lines <Array.<number>>`: An array of line numbers to run or focus on within the script.
- `silent <boolean>`: Whether to suppress output from the script execution.
- `force_run <Boolean>`: If true, forces the script to run even if stop conditions are met.

Runs a script from a specified path, optionally focusing on specific lines and controlling output visibility.

`helpToJSON([name], [type])`

Parameters:

- `name <string>`: The name of the documentation item.
- `type <string>`: The type of the documentation (e.g., 'category').

Returns: `<string>`: The JSON string of the documentation or undefined if not found. Retrieves documentation in JSON format based on the provided name and type.

`help(name, type)`

Parameters:

- `name <string>`: The name of the documentation item.
- `type <string>`: The type of the documentation.

Returns: `<string>`: The JSON string of the documentation or undefined if not found.
Retrieves documentation based on the provided name and type.

```
doc(name, type)
```

Parameters:

- `name <string>`: The name of the documentation item.
- `type <string>`: The type of the documentation.

Returns: `<string>`: The JSON string of the documentation or undefined if not found.
Retrieves documentation based on the provided name and type.

```
documentation(name, type)
```

Parameters:

- `name <string>`: The name of the documentation item.
- `type <string>`: The type of the documentation.

Returns: `<string>`: The JSON string of the documentation or undefined if not found.
Retrieves documentation based on the provided name and type.

```
helpSearch(query)
```

Parameters:

- `query <string>`: The search query containing keywords to match within the documentation.

Returns: `<Array.<Object>>`: Array of matching documentation entries, each entry containing `type` and `category` properties.
Searches the documentation for methods that match all words in the given query, regardless of order.

```
docSearch(query)
```

Parameters:

- `query <string>`: The search query containing keywords to match within the documentation.

Returns: `<Array.<Object>>`: Array of matching documentation entries, each entry containing `type` and `category` properties.
Searches the documentation for methods that match all words in the given query, regardless of order.

`documentationSearch(query)`

Parameters:

- `query <string>`: The search query containing keywords to match within the documentation.

Returns: `<Array.<Object>>`: Array of matching documentation entries, each entry containing `type` and `category` properties.
Searches the documentation for methods that match all words in the given query, regardless of order.

`source(name)`

Parameters:

- `name <string>`: The name of the source to locate.

Opens the source file and navigates to the specified line based on the provided name.

`docGraph(name)`

Parameters:

- `name <string>`: The name of the function.

Showing graph of function.

`jslFileName()`

Returns: `<string>`: The file name of the JSL script.
Retrieves the file name of the currently active JSL script.

`clearStorage()`

Clears the application's local storage.

`savePath(new_path)`

Parameters:

- `new_path <string>`: The path to save.

Saves a path to the application's list of saved paths.

`removePath(saved_path)`

Parameters:

- `saved_path <string>`: The path to remove.

Removes a previously saved path from the application's list of saved paths.

`cd(new_path)`

Parameters:

- `new_path <string>`: The new path to set as the current working directory.

Changes the current working directory to the specified path.

`workspace()`

Returns: `<Object>`: The current workspace object.
Retrieves the current workspace.

`updateWorkspace()`

Updates the workspace display based on the current state.

`updateFileBrowser()`

Updates the file browser display based on the current state.

`error(msg)`

Parameters:

- `msg <string>`: The error message to display.

Displays an error message.

`disp(msg)`

Parameters:

- `msg <string>`: The message to display.

Displays a general message.

`dispMonospaced(msg)`

Parameters:

- `msg <string>`: The message to display.

Displays a general message with monospaced font.

`warn(msg)`

Parameters:

- `msg <string>`: The warning message to display.

Displays a warning message.

`checkStopLoop()`

Verifies if a loop within the script execution should be terminated, typically used to avoid infinite or lengthy unnecessary execution.

`edit([filepath])`

Parameters:

- `filepath <string>`: Path to the file to be opened in the editor.

Opens a specified file in an editor or opens the editor to a default or previously specified file.

`getExamples()`

Returns: `<Array.<string>>`: An array of paths to the example scripts.
Returns a list of all example scripts available within a predefined directory.

`openExample(filename)`

Parameters:

- `filename <string>`: Name of the example file to open.

Opens a specified example script in the editor window.

`openExamplesFolder()`

Opens examples folder in File Explorer

`goToExamplesFolder()`

Opens examples folder in File Explorer

`showMessageBox(options)`

Parameters:

- `options <Object>`: Configuration options for the message box.

Returns: `<number>`: The index of the button clicked by the user.
Displays a synchronous message box to the user and waits for their response.

`save(file_path, args)`

Parameters:

- `file_path <string>`: Path where the JSON file will be saved.
- `args <string>`: Variables to save. If 'all' is specified, saves all available variables.

Saves specified variables to a JSON file.

`load(args)`

Parameters:

- `args <*>`: A single filename or a scope and filename to specify where to load the variables.

Loads variables from a specified JSON file into the specified scope or the default script context. If an error occurs during file reading or parsing, it logs an error message.

`system(arg)`

Parameters:

- `arg <*>`: The command and its arguments to be executed.

Returns: `<string>`: The output of the executed command.
Executes a system shell command.

`getCompletions(data)`

Parameters:

- `data <Array>`: Data containing the start of the string to complete, context, and keywords.

Returns: `<Array.<string>>`: An array of completion suggestions.
Retrieves completion suggestions based on the current context and input.

`getObjectByProp(obj, prop, val)`

Parameters:

- `obj <Object>`: The object to search through.
- `prop <string>`: The property name to match.
- `val <*>`: The value to match against the property.

Returns: `<Object>`: The found object with key and value, or null if not found.
Retrieves an object by matching a specific property value.

`getObjectsByProp(obj, prop, val)`

Parameters:

- `obj <Object>`: The parent object to search through.
- `prop <string>`: The property name to match.
- `val <*>`: The value to match against the property.

Returns: `<Object>`: An object containing all matched key-value pairs.
Retrieves multiple objects from a parent object by matching a specific property value.

`strcmp(x, y)`

Parameters:

- `x <string>`: The first string.
- `y <string>`: The second string.

Returns: `<number>`: The result of the comparison.
Compares two strings lexicographically.

`compareVersions(a, b)`

Parameters:

- `a <string>`: First version.
- `b <string>`: Second version.

Returns: `<number>`: -1 if a ¡ b, 0 if equal, 1 if a ¿ b.
Compare two version strings (e.g. ”1.4.2”, ”v2.0.0-beta.1”).

`checkForUpdate()`

Returns: `<boolean>`: True if there is available update; otherwise, false.
Checks if there is available update

`unrequire(module)`

Parameters:

- `module <string>`: The module to unrequire.

Unloads a previously required module from the cache.

`resetApp()`

Resets app.

`resetSandbox()`

Resets the sandbox environment to its initial state.

`openDevTools()`

Returns: `<void>`:
Opens the developer tools for the sandbox environment in the current context.

`compileNapi(path, [show_output])`

Parameters:

- `path <string>`: The path to the N-API module.
- `show_output <boolean>`: Whether to show output in the command window.

Returns: `<Array>`: An array containing the result of the compilation and targets.
Compiles a N-API module located at the specified path.

```
installModule(path, [show_output])
```

Parameters:

- `path <string>`: The path to the module.
- `show_output <boolean>`: Whether to show output in the command window.

Installs a module located at the specified path.

```
addForCleanup(obj, fun)
```

Parameters:

- `obj <Object>`: The object to be registered for cleanup.
- `fun <function>`: The function to execute during cleanup.

Registers an object for cleanup with a specified cleanup function.

10.2 math

```
Pi
```

Type: `<Unknown>`
Pi number.

```
d2r
```

Type: `<Unknown>`
Coefficient for converting degrees to radians.

```
r2d
```

Type: `<Unknown>`
Coefficient for converting radians to degrees.

```
eps
```

Type: `<Unknown>`
Floating-point relative accuracy

`EPS`

Type: `<Unknown>`
Floating-point relative accuracy

`seedRandom(args)`

Parameters:

- `args <any>`: Arguments used to seed the random generator.

Returns: `<any>`: The result from the seeded random generator.
Seeds the random number generator with the provided arguments.

`interp(x, y, xq, mode)`

Parameters:

- `x <Array>`: The x-values of the data points.
- `y <Array>`: The y-values of the data points, corresponding to each x-value.
- `xq <Number>`: The x-value(s) for which to interpolate a y-value.
- `mode <String>`: The mode of interpolation. Use 'extrap' for extrapolation.

Returns: `<Number>`: The interpolated y-value(s) at xq.
Performs linear interpolation on a set of data points.

`gridGradient(x, y, z, [N_a])`

Parameters:

- `x <Array>`: X coordinates.
- `y <Array>`: Y coordinates.
- `z <Array.<Array>>`: 2D data array.
- `N_a <number>`: Neighborhood size.

Returns: `<Array.<Array>>`: Gradient components [dz_x, dz_y].
Computes the gradient of a 2D grid.

`gridData(x, y, z, xq, yq, [method], [opts_in])`

Parameters:

- `x <Array>`: X coordinates.
- `y <Array>`: Y coordinates.
- `z <Array>`: Data values.
- `xq <Array>`: Query X coordinates.
- `yq <Array>`: Query Y coordinates.
- `method <string>`: Interpolation method.
- `opts_in <Object>`: Optional settings.

Returns: `<Array.<Array>>`: Interpolated grid [xq, yq, zq].
Interpolates grid data using the specified method.

```
bilinearFunction(x, midPoint, midValue)
```

Parameters:

- `x <number>`: The input value for the function.
- `midPoint <number>`: The midpoint of the function where the slope changes.
- `midValue <number>`: The value of the function at the midpoint.

Returns: `<number>`: The output value of the bilinear function.
Calculates the output of a bilinear function based on input value, midpoint, and mid-value.

```
random([min], [max])
```

Parameters:

- `min <Number>`: The lower bound of the range.
- `max <Number>`: The upper bound of the range.

Returns: `<Number>`: A random number within the specified range.
Generates a random number between a specified range.

```
randInt([min], [max])
```

Parameters:

- `min <Number>`: The lower bound of the range.
- `max <Number>`: The upper bound of the range.

Returns: `<Number>`: A random integer within the specified range.
Generates a random integer within a specified range.

`acosd(x)`

Parameters:

- `x <Number>`: The value to compute the arc cosine for.

Returns: `<Number>`: The arc cosine of x in degrees.
Computes the arc cosine of x, with the result in degrees.

`acotd(x)`

Parameters:

- `x <Number>`: The value to compute the arc cotangent for.

Returns: `<Number>`: The arc cotangent of x in degrees.
Computes the arc cotangent of x, with the result in degrees.

`acscd(x)`

Parameters:

- `x <Number>`: The value to compute the arc cosecant for.

Returns: `<Number>`: The arc cosecant of x in degrees.
Computes the arc cosecant of x, with the result in degrees.

`asecd(x)`

Parameters:

- `x <Number>`: The value to compute the arc secant for.

Returns: `<Number>`: The arc secant of x in degrees.
Computes the arc secant of x, with the result in degrees.

`asind(x)`

Parameters:

- `x <Number>`: The value to compute the arc sine for.

Returns: `<Number>`: The arc sine of x in degrees.
Computes the arc sine of x, with the result in degrees.

```
atand(x)
```

Parameters:

- `x <Number>`: The value to compute the arc tangent for.

Returns: `<Number>`: The arc tangent of x in degrees.
Computes the arc tangent of x, with the result in degrees.

```
atan2d(y, x)
```

Parameters:

- `y <Number>`: The y coordinate.
- `x <Number>`: The x coordinate.

Returns: `<Number>`: The arc tangent of y/x in degrees.
Computes the arc tangent of the quotient of its arguments, with the result in degrees.

```
cosd(x)
```

Parameters:

- `x <Number>`: The angle in degrees.

Returns: `<Number>`: The cosine of x in degrees.
Computes the cosine of x, where x is in degrees.

```
cotd(x)
```

Parameters:

- `x <Number>`: The angle in degrees.

Returns: `<Number>`: The cotangent of x.
Computes the cotangent of x, where x is in degrees.

```
cscd(x)
```

Parameters:

- `x <Number>`: The angle in degrees.

Returns: `<Number>`: The cosecant of x.
Computes the cosecant of x, where x is in degrees.

```
secd(x)
```

Parameters:

- `x <Number>`: The angle in degrees.

Returns: `<Number>`: The secant of x.
Computes the secant of x, where x is in degrees.

```
sind(x)
```

Parameters:

- `x <Number>`: The angle in degrees.

Returns: `<Number>`: The sine of x.
Computes the sine of x, where x is in degrees.

```
tand(x)
```

Parameters:

- `x <Number>`: The angle in degrees.

Returns: `<Number>`: The tangent of x.
Computes the tangent of x, where x is in degrees.

```
poly(A)
```

Parameters:

- `A <Array>`: If `A` is a matrix (2D array), computes its characteristic polynomial. If `A` is an array of roots, computes the polynomial with those roots.

Returns: `<Array>`: - Coefficients of the resulting polynomial.
Computes the characteristic polynomial of a matrix or the polynomial from roots.

```
polyfit(x, y, n)
```

Parameters:

- `x <Array.<number>>`: The array of x-values.
- `y <Array.<number>>`: The array of y-values corresponding to each x-value.

- `n <number>`: The degree of the polynomial to fit.

Returns: `<Array.<number>>`: The coefficients of the fitted polynomial in descending order.
Fits a polynomial of degree n to the given data points and returns the coefficients (highest degree first).

```
polyval(p, x_in)
```

Parameters:

- `p <Array.<number>>`: The coefficients of the polynomial in descending order.
- `x_in <Array.<number>>`: The array of x-values at which to evaluate the polynomial.

Returns: `<Array.<number>>`: The resulting y-values after evaluating the polynomial at x_in.
Evaluates a polynomial with given coefficients at specified x-values.

```
roots(p)
```

Parameters:

- `p <Array.<number>>`: Array of polynomial coefficients, ordered from highest degree to constant term.

Returns: `<Array.<number>>`: Array of roots (real or complex) of the polynomial.
Computes the roots of a polynomial with the given coefficients.

```
polystr(p, [opts], [opts.x_symbol], [opts.y_symbol], [opts.precision],
[opts.lang])
```

Parameters:

- `p <Array.<number>>`: An array of polynomial coefficients, ordered from highest degree to constant term.
- `opts <Object>`: Optional settings for the polynomial string.
- `opts.x_symbol <string>`: The symbol to use for the variable x.
- `opts.y_symbol <string>`: The symbol to use for the variable y.
- `opts.precision <number>`: The number of decimal places for coefficients.
- `opts.lang <string>`: The language format for the output ('tex', 'c', etc.).

Returns: `<string>`: The formatted polynomial string.
Generates a string representation of a polynomial based on the provided coefficients and options.

```
polystrc(p, [opts])
```

Parameters:

- `p <Array.<number>>`: An array of polynomial coefficients, ordered from highest degree to constant term.
- `opts <Object>`: Optional settings for the polynomial string.

Returns: `<string>`: The polynomial string formatted for C language.
Generates a C language formatted string representation of a polynomial.

```
polystrtex(p, [opts])
```

Parameters:

- `p <Array.<number>>`: An array of polynomial coefficients, ordered from highest degree to constant term.
- `opts <Object>`: Optional settings for the polynomial string.

Returns: `<string>`: The polynomial string formatted for LaTeX.
Generates a LaTeX formatted string representation of a polynomial.

```
spikeFilter(x, dx_max, n)
```

Parameters:

- `x <Array.<number>>`: The input sequence of numbers.
- `dx_max <number>`: The maximum allowed difference between consecutive elements before considering it a spike.
- `n <number>`: The maximum number of consecutive spikes to correct.

Returns: `<Array.<number>>`: The filtered sequence with spikes removed.
Filters out spikes in a sequence by replacing sudden large changes with the previous value.

```
magnitude(num)
```

Parameters:

- `num <number>`: A real number or an object with 'real' and 'imag' properties.

Returns: `<number>`: The magnitude of the number.
Calculates the magnitude (absolute value) of a complex number or a real number.

`compareComplex(a, b)`

Parameters:

- `a <number>`: First number to compare.
- `b <number>`: Second number to compare.

Returns: `<number>`: -1 if a ¡ b, 1 if a ¿ b, 0 if equal.
Compares two numbers (real or complex) according to Octave's rules.

`min(arr)`

Parameters:

- `arr <Array.<(number|Object)>>`: An array of numbers or complex number objects.

Returns: `<number>`: The minimum value found in the array.
Finds the minimum value in an array of numbers, which can include both real numbers and complex numbers. Complex numbers are represented as objects with 'real' and 'imag' properties. The comparison follows Octave's rules: 1. Compare magnitudes (absolute values) of the numbers. 2. If magnitudes are equal, compare real parts. 3. If real parts are equal, compare imaginary parts.

`max(arr)`

Parameters:

- `arr <Array.<(number|Object)>>`: An array of numbers or complex number objects.

Returns: `<number>`: The maximum value found in the array.
Finds the maximum value in an array of numbers, which can include both real numbers and complex numbers. Complex numbers are represented as objects with 'real' and 'imag' properties. The comparison follows Octave's rules: 1. Compare magnitudes (absolute values) of the numbers. 2. If magnitudes are equal, compare real parts. 3. If real parts are equal, compare imaginary parts.

`real(input)`

Parameters:

- `input <number>`: A number, complex number object, or array thereof.

Returns: `<number>`: The real part(s) of the input.
Extracts the real part of a number or an array of numbers. Handles mixed inputs containing both real numbers and complex numbers.

```
imag(input)
```

Parameters:

- `input <number>`: A number, complex number object, or array thereof.

Returns: `<number>`: The imaginary part(s) of the input.
Extracts the imaginary part of a number or an array of numbers. Handles mixed inputs containing both real numbers and complex numbers.

```
cumtrapz(args)
```

Parameters:

- `args <any>`: Arguments required for cumulative trapezoidal integration.

Returns: `<any>`: The result of the cumulative trapezoidal integration.
Performs cumulative trapezoidal integration on the provided data.

```
trapz(args)
```

Parameters:

- `args <any>`: Arguments required for trapezoidal integration.

Returns: `<any>`: The result of the trapezoidal integration.
Performs trapezoidal integration on the provided data.

```
mse(A, B)
```

Parameters:

- `A <Array.<number>>`: The first array.
- `B <Array.<number>>`: The second array.

Returns: `<number>`: - The mean squared error between A and B.
Compute the mean squared error (MSE) between two arrays.

```
charpoly(matrix)
```

Parameters:

- `matrix <Array.<Array.<number>>>`: A square matrix (2D array) for which the characteristic polynomial is computed.

Returns: `<Array.<number>>`: - An array of coefficients of the characteristic polynomial.
Calculates the coefficients of the characteristic polynomial of a square matrix.

10.3 non_blocking

`nbwhile(fn)`

Parameters:

- `fn <function>`: A function that returns a boolean value; when false, the loop exits.

Executes a given function in a non-blocking while loop.

`nbrun(fn)`

Parameters:

- `fn <function>`: The function to be executed.

Executes a given function once in a non-blocking manner.

`nbnext(fn)`

Parameters:

- `fn <function>`: The function to execute next.

Schedules the next block of code to be executed in a non-blocking manner.

`waitMSeconds(ms)`

Parameters:

- `ms <Number>`: The number of milliseconds to wait.

Returns: `<Promise.<void>>`: A promise that resolves after the specified time has elapsed.
Waits for a specified number of milliseconds in a non-blocking manner.

`waitSeconds(s)`

Parameters:

- `s <Number>`: The number of seconds to wait.

Returns: `<Promise.<void>>`: A promise that resolves after the specified time has elapsed.
Waits for a specified number of seconds in a non-blocking manner.

`waitMinutes(min)`

Parameters:

- `min <Number>`: The number of minutes to wait.

Returns: `<Promise.<void>>`: A promise that resolves after the specified time has elapsed.
Waits for a specified number of minutes in a non-blocking manner.

`clearIntervalIf(timeout)`

Parameters:

- `timeout <number>`: The interval ID to be cleared.

Returns: `<boolean>`: Always returns false.
Clears the specified interval if it exists.

`clearTimeoutIf(timeout)`

Parameters:

- `timeout <number>`: The timeout ID to be cleared.

Returns: `<boolean>`: Always returns false.
Clears the specified timeout if it exists.

`initWorker(path)`

Parameters:

- `path <string>`: The path to the module to configure the worker.

Returns: `<Worker>`: The initialized Worker instance.
Initializes a new worker with the specified module path.

10.4 path

`getDir(path)`

Parameters:

- `path <String>`: The filesystem path from which to extract the directory.

Returns: `<String>`: The directory from the given path.
Extracts the directory of file.

`getDirName(path)`

Parameters:

- `path <String>`: The filesystem path from which to extract the directory name.

Returns: `<String>`: The name of the directory from the given path.
Extracts the name of the directory from a given filesystem path.

`pathSep()`

Returns: `<String>`: The path separator character used by the system.
Retrieves the platform-specific path separator character.

`isAbsolutePath()`

Returns: `<boolean>`: True if the current path is absolute, false otherwise.
Determines if the current path is absolute.

`pathJoin(paths)`

Parameters:

- `paths <string>`: The path segments to join.

Returns: `<string>`: The combined path.
Joins all given path segments together using the platform-specific separator as a delimiter.

`pathFileName(path)`

Parameters:

- `path <string>`: The complete file path.

Returns: `<string>`: The file name extracted from the path.
Retrieves the file name from the provided file path.

`pathBaseName(path)`

Parameters:

- `path <string>`: The file path to process.

Returns: `<string>`: The last segment of the path.
Returns the last portion of a path, similar to the Unix `basename` command.

`pathFileExt(path)`

Parameters:

- `path <string>`: The complete file path.

Returns: `<string>`: The file extension extracted from the path.
Retrieves the file extension from the provided file path.

`pathExtName(path)`

Parameters:

- `path <string>`: The complete file path.

Returns: `<string>`: The file extension extracted from the path.
Retrieves the file extension from the provided file path.

`pathResolve(path)`

Parameters:

- `path <string>`: The path or sequence of paths to resolve.

Returns: `<string>`: - The resolved absolute path.
Resolves a sequence of path segments into an absolute path using the environment's path resolver.

`pathRelative(from, to)`

Parameters:

- `from <string>`: The starting path.
- `to <string>`: The target path.

Returns: `<string>`: - The relative path from the `from` path to the `to` path.
Computes the relative path from one path to another.

`pathNormalize(path)`

Parameters:

- `path <string>`: The path to normalize.

Returns: `<string>`: - The normalized path.
Normalizes a given path, resolving '..' and '.' segments using the environment's path normalizer.

```
comparePaths(path1, path2)
```

Parameters:

- `path1 <string>`: The first file path to compare.
- `path2 <string>`: The second file path to compare.

Returns: `<boolean>`: Returns true if both paths resolve to the same absolute path, otherwise false.
Compares two file paths after resolving them to their absolute forms to check if they refer to the same location.

```
getUniquePath(path)
```

Parameters:

- `path <String>`: The base path for which a unique version is required.

Returns: `<String>`: A unique filesystem path based on the input path.
Generates a unique filesystem path by appending a number to the input path if the original path exists.

```
getUniqueFilename(path, ext)
```

Parameters:

- `path <string>`: The original file path.
- `ext <string>`: The original file extension.

Returns: `<string>`: A unique folder path.
Generates a unique filename by appending a number to the original path if it already exists.

10.5 windows

```
active_window
```

Type: `<Unknown>`
Current active window ID.

```
open_windows
```

Type: `<Unknown>`
Array of open windows.

`openWindow(file)`

Parameters:

- `file <string>`: The path to the HTML file to open in the new window.

Returns: `<number>`: The identifier (wid) of the newly opened window.
Opens a new window with the specified file.

`openWindowDevTools(wid)`

Parameters:

- `wid <string>`: The window ID.

Returns: `<boolean>`: True if the developer tools were opened; otherwise, false.
Opens the developer tools for a specified window by ID if it exists.

`getWindowMediaSourceId(wid)`

Parameters:

- `wid <string>`: The window ID.

Returns: `<String>`: Media source id if there is window; otherwise, false.
Returns media source id for a specified window in the renderer process.

`startWindowVideoRecording(wid, opts)`

Parameters:

- `wid <string>`: The window ID.
- `opts <Object>`: Optional settings.

Returns: `<Object>`: - Returns recorder object if there is window; otherwise, false.
Starts video recording of the window.

`closeWindows(wid)`

Parameters:

- `wid <number>`: Identifier for the window to close.

Closes the specified window.

`closeWindow(wid)`

Parameters:

- `wid <number>`: Identifier for the window to close.

Closes the specified window.

`getWindow(wid)`

Parameters:

- `wid <number>`: The ID of the window.

Returns: `<Object>`: - The window object if found, otherwise false.
Retrieves the window object with the specified ID.

`getCurrentWindow()`

Returns: `<Object>`: - The window object if found, otherwise false.
Retrieves current active window object.

`gcw()`

Returns: `<Object>`: - The window object if found, otherwise false.
Retrieves current active window object.

`showWindow(wid)`

Parameters:

- `wid <number>`: The ID of the window to show.

Returns: `<boolean>`: - Returns false if the window ID is invalid, otherwise the result of the show() method.
Shows the specified window.

`hideWindow(wid)`

Parameters:

- `wid <number>`: The ID of the window to hide.

Returns: `<boolean>`: - Returns false if the window ID is invalid, otherwise the result of the hide() method.
Hides the specified window.

```
focusWindow(wid)
```

Parameters:

- `wid <number>`: The ID of the window to focus.

Returns: `<boolean>`: - Returns false if the window ID is invalid.
Brings the specified window to the foreground.

```
minimizeWindow(wid)
```

Parameters:

- `wid <number>`: The ID of the window to minimize.

Returns: `<boolean>`: - Returns false if the window ID is invalid, otherwise the result of the minimize() method.
Minimizes the specified window.

```
centerWindow(wid)
```

Parameters:

- `wid <number>`: The ID of the window to center.

Returns: `<boolean>`: - Returns false if the window ID is invalid, otherwise the result of the center() method.
Centers the specified window on the screen.

```
moveTopWindow(wid)
```

Parameters:

- `wid <number>`: The ID of the window to move to the top.

Returns: `<boolean>`: - Returns false if the window ID is invalid, otherwise the result of the moveTop() method.
Moves the specified window to the top of the window stack.

```
setWindowSize(wid, width, height)
```

Parameters:

- `wid <number>`: The ID of the window.
- `width <number>`: The new width of the window.
- `height <number>`: The new height of the window.

Returns: `<boolean>`: - Returns false if the window ID is invalid.
Sets the size of the specified window.

```
setWindowPos(wid, left, top)
```

Parameters:

- `wid <number>`: The ID of the window.
- `left <number>`: The new left position of the window.
- `top <number>`: The new top position of the window.

Returns: `<boolean>`: - Returns false if the window ID is invalid.
Sets the position of the specified window.

```
setWindowResizable(wid, state)
```

Parameters:

- `wid <number>`: The ID of the window.
- `state <boolean>`: Whether the window should be resizable.

Returns: `<boolean>`: - Returns false if the window ID is invalid, otherwise the result of the setResizable() method.
Sets the resizable state of the specified window.

```
setWindowMovable(wid, state)
```

Parameters:

- `wid <number>`: The ID of the window.
- `state <boolean>`: Whether the window should be movable.

Returns: `<boolean>`: - Returns false if the window ID is invalid, otherwise the result of the setMovable() method.
Sets the movable state of the specified window.

```
setWindowAspectRatio(wid, aspect_ratio)
```

Parameters:

- `wid <number>`: The ID of the window.
- `aspect_ratio <number>`: The desired aspect ratio of the window.

Returns: `<boolean>`: - Returns false if the window ID is invalid, otherwise the result of the setAspectRatio() method.
Sets the aspect ratio of the specified window.

```
setWindowOpacity(wid, opacity)
```

Parameters:

- `wid <number>`: The ID of the window.
- `opacity <number>`: The desired opacity level (0 to 1).

Returns: `<boolean>`: - Returns false if the window ID is invalid, otherwise the result of the setOpacity() method.
Sets the opacity of the specified window.

```
setWindowFullscreen(wid, state)
```

Parameters:

- `wid <number>`: The ID of the window.
- `state <number>`: The fullscreen state

Returns: `<boolean>`: - Returns false if the window ID is invalid, otherwise the result of the setFullscreen() method.
Sets the fullscreen state of the specified window.

```
setWindowTitle(wid, left, top)
```

Parameters:

- `wid <number>`: The ID of the window.
- `left <number>`: The new left position of the window.
- `top <number>`: The new top position of the window.

Returns: `<boolean>`: - Returns false if the window ID is invalid.
Sets the position of the specified window.

```
getWindowSize(wid)
```

Parameters:

- `wid <number>`: The ID of the window.

Returns: `<Array>`: - An array [width, height] if the window exists, otherwise false.
Retrieves the size of the specified window.

`getWindowPos(wid)`

Parameters:

- `wid <number>`: The ID of the window.

Returns: `<Array>`: - An array [left, top] if the window exists, otherwise false.
Retrieves the position of the specified window.

`printWindowToPdf(wid, options)`

Parameters:

- `wid <string>`: The window ID.
- `options <Object>`: Options for printing

Returns: `<Buffer>`: Generated PDF data.
Prints window to PDF format.

`openDocumentation()`

Opens window with documentation

`openDoc()`

Opens window with documentation

`openWindow3D([imports])`

Parameters:

- `imports <Array.<Object>>`: An array of import objects specifying modules to import.

Returns: `<Promise.<Object>>`: A promise that resolves to the window object once imports are ready.
Opens a new 3D window and imports specified modules.

`openPlotlyjs()`

Returns: `<Promise.<Window>>`: The window object where Plotly.js is loaded and the plot container is available.
Opens a Plotly.js window and initializes the plot container.

`openCanvas()`

Returns: `<Promise.<Window>>`: The window object where D3 is loaded and the canvas element is available.
Opens a window with canvas and D3 and initializes the canvas element.

`openWindowBlank()`

Returns: `<Promise.<Object>>`: A promise that resolves to the window object once it is ready.
Opens a new blank window.

`showMermaidGraph(graph_definition)`

Parameters:

- `graph_definition <string>`: The Mermaid graph definition.

Returns: `<Promise.<Object>>`: A promise that resolves to the window context once the graph is rendered.
Renders a Mermaid diagram in a new window.

10.6 figures

`open_figures`

Type: `<Unknown>`
Array of open figures.

`active_figure`

Type: `<Unknown>`
Current active figure ID.

`figure(id)`

Parameters:

- `id <Number>`: Identifier for the figure.

Returns: <Number>: The identifier of the opened or updated figure.
Opens or updates a figure with specified options.

`getFigure(fid)`

Parameters:

- `fid` <string>: The identifier of the figure to retrieve.

Returns: <Object>: The figure object if found, otherwise `false`.
Retrieves the figure object associated with the specified figure ID.

`getFigureWindow(fid)`

Parameters:

- `fid` <string>: The identifier of the figure to retrieve.

Returns: <Object>: The figure object if found, otherwise `false`.
Retrieves the window of the figure object associated with the specified figure ID.

`getCurrentFigure()`

Returns: <Object>: The figure object if found, otherwise `false`.
Retrieves current active figure object.

`gcf()`

Returns: <Object>: The figure object if found, otherwise `false`.
Retrieves current active figure object.

`getPlot(fid)`

Parameters:

- `fid` <string>: The identifier of the figure to retrieve the plot for.

Returns: <Object>: The plot object if it exists, otherwise `false`.
Retrieves the plot object for a specified figure ID.

`getAxes(fid)`

Parameters:

- `fid` <string>: The identifier of the figure to retrieve the plot for.

Returns: `<Object>`: The plot object if it exists, otherwise `false`.
Retrieves the plot object for a specified figure ID.

`getCurrentPlot()`

Returns: `<Object>`: The figure object if found, otherwise `false`.
Retrieves plot from current active figure object.

`gcp()`

Returns: `<Object>`: The figure object if found, otherwise `false`.
Retrieves plot from current active figure object.

`getCurrentAxes()`

Returns: `<Object>`: The figure object if found, otherwise `false`.
Retrieves plot from current active figure object.

`gca()`

Returns: `<Object>`: The figure object if found, otherwise `false`.
Retrieves plot from current active figure object.

`focusFigure(fid)`

Parameters:

- `fid <number>`: The ID of the figure to focus.

Returns: `<boolean>`: - Returns false if the figure ID is invalid.
Brings the specified figure to the foreground.

`setFigureSize(fid, width, height)`

Parameters:

- `fid <number>`: The ID of the figure.
- `width <number>`: The new width of the figure.
- `height <number>`: The new height of the figure.

Returns: `<boolean>`: - Returns false if the figure ID is invalid.
Sets the size of a specified figure.

`setFigurePos(fid, left, top)`

Parameters:

- `fid <number>`: The ID of the figure.
- `left <number>`: The new left position of the figure.
- `top <number>`: The new top position of the figure.

Returns: `<boolean>`: - Returns false if the figure ID is invalid.
Sets the position of a specified figure.

`setFigureTitle(fid, title)`

Parameters:

- `fid <string>`: The figure ID.
- `title <string>`: The new title for the figure.

Returns: `<boolean>`: The result of setting the title, or false if the figure does not exist.
Sets the title of the specified figure.

`getFigureSize(fid)`

Parameters:

- `fid <number>`: The ID of the figure.

Returns: `<Array>`: - Returns an array [width, height] or false if the figure ID is invalid.
Retrieves the size of a specified figure.

`getFigurePos(fid)`

Parameters:

- `fid <number>`: The ID of the figure.

Returns: `<Array>`: - Returns an array [left, top] or false if the figure ID is invalid.
Retrieves the position of a specified figure.

`getFigureMediaSourceId(fid)`

Parameters:

- `fid <number>`: The ID of the figure.

Returns: `<String>`: - Returns Media soruce id or false if the figure ID is invalid.
Retrieves the media source id of a specified figure.

```
startFigureVideoRecording(fid, opts)
```

Parameters:

- `fid <number>`: The ID of the figure.
- `opts <Object>`: Optional settings.

Returns: `<Object>`: - Returns recorder object or false if the figure ID is invalid.
Starts video recording of a specified figure.

```
closeFigure(fid)
```

Parameters:

- `fid <number>`: The ID of the figure to close.

Returns: `<boolean>`: - Returns false if the figure ID is invalid.
Closes a specified figure.

```
close(id, [type])
```

Parameters:

- `id <number>`: The identifier of the figure or window to close. Use "all" to close all.
- `type <string>`: The type of object to close ('figure' or 'window').

Closes a figure or window by its identifier.

```
saveFigureDialog(fid)
```

Parameters:

- `fid <String>`: The figure identifier.

Opens a dialog for saving a figure in various formats.

```
saveFigure(fid, figure_path, size)
```

Parameters:

- `fid <String>`: The figure identifier.

- `figure_path <String>`: The path where the figure should be saved.
- `size <Array>`: Optional dimensions [width, height] to use if saving as a PDF.

Saves a figure to a specified path in various formats.

`legend(label)`

Parameters:

- `label <String>`: The label for the x-axis.

Sets the label for the x-axis of the active figure.

`xlabel(label)`

Parameters:

- `label <String>`: The label for the x-axis.

Sets the label for the x-axis of the active figure.

`ylabel(label)`

Parameters:

- `label <String>`: The label for the y-axis.

Sets the label for the y-axis of the active figure.

`zlabel(label)`

Parameters:

- `label <String>`: The label for the z-axis.

Sets the label for the z-axis of the active figure.

`title(label)`

Parameters:

- `label <String>`: The title text.

Sets the title of the active figure.

`xlim(lim)`

Parameters:

- `lim <String>`: x limits.

Sets the xlim of the active figure.

`ylim(lim)`

Parameters:

- `lim <String>`: y limits.

Sets the ylim of the active figure.

`zlim(lim)`

Parameters:

- `lim <String>`: z limits.

Sets the zlim of the active figure.

`view(azimuth, elevation)`

Parameters:

- `azimuth <number>`: The azimuth angle.
- `elevation <number>`: The elevation angle.

Adjusts the view based on azimuth and elevation angles.

`zoom(factor)`

Parameters:

- `factor <number>`: The zoom factor.

Adjusts the zoom based on zoom factor.

`axis(style)`

Parameters:

- `style <Object>`: The style configuration to apply to the axis.

Applies the specified style to the active figure's plot axis.

`printFigure(filename, options)`

Parameters:

- `filename <String>`: The name of the file where the figure should be printed.
- `options <Object>`: Printing options.

Prints the currently active figure to a file.

```
plot(traces, options)
```

Parameters:

- `traces <Array>`: Data traces to plot.
- `options <Object>`: Configuration options for plotting.

Returns: `<Number>`: The plot identifier.
Plots data on the active figure.

```
updatePlot(traces, N)
```

Parameters:

- `traces <Object>`: The trace data to be updated in the plot.
- `N <number>`: The data length or index for updating the plot.

Updates plot data by delegating to the `update` method.

```
updatePlotById(data)
```

Parameters:

- `data <Object>`: Trace update object(s) to apply to the active plot.

Updates plot data by id by delegating to the `updateById` method.

```
hideFigureMenu()
```

Hides figure menu.

```
showFigureMenu()
```

Shows figure menu.

```
loadJsonFigure(file_path, id)
```

Parameters:

- `file_path <String>`: Absolute or relative path to the JSON file of figure.
- `id <Number>`: Identifier for the figure.

Returns: `<Number>`: The identifier of the opened or updated figure.
Loads figure from JSON file

10.7 time

`timezone`

Type: `<Unknown>`
Current timezone string.

`last_tic`

Type: `<Unknown>`
Last tic timestamp.

`tic()`

Starts a timer for measuring elapsed time. To be used with `toc` to measure time intervals.

`toc([tic])`

Parameters:

- `tic <number>`: The start time in milliseconds from which to calculate elapsed time. If omitted, uses the last time recorded by `tic()`.

Returns: `<number>`: The elapsed time in seconds.
Calculates the wall-clock time elapsed since a specified start time or the last call to `tic()`, in seconds.

`tocms([tic])`

Parameters:

- `tic <number>`: The start time in milliseconds from which to calculate elapsed time. If omitted, uses the last time recorded by `tic()`.

Returns: `<number>`: The elapsed time in milliseconds.
Calculates the wall-clock time elapsed since a specified start time or the last call to `tic()`, in seconds.

`getTimestamp()`

Returns: `<number>`: The current Unix timestamp as an integer.
Gets the current Unix timestamp adjusted for a specified timezone.

`getTime()`

Returns: `<string>`: The current time in 'HH:mm:ss' format.
Gets the current time as a string adjusted for a specified timezone.

`getFullTime()`

Returns: `<string>`: The current time in 'HH:mm:ss.SSS' format.
Gets the current time with milliseconds as a string adjusted for a specified timezone.

`getDate()`

Returns: `<string>`: The current date in 'dd.MM.yyyy.' format.
Gets the current date as a string adjusted for a specified timezone.

`getDateTime()`

Returns: `<string>`: The current date and time in 'dd.MM.yyyy. HH:mm:ss' format.
Gets the current date and time as a string adjusted for a specified timezone.

`getDateTimeFull()`

Returns: `<string>`: The current date and time in 'dd.MM.yyyy. HH:mm:ss.SSS' format.
Gets the current date and time with milliseconds as a string adjusted for a specified timezone.

`getDateTimeStr()`

Returns: `<string>`: The current date and time in 'ddMMyyyy_HHmmss' format for use in filenames.
Gets the current date and time as a string suitable for filenames, adjusted for a specified timezone.

`setTimezone(tz)`

Parameters:

- `tz` `<String>`: The timezone identifier (e.g., "America/New_York", "Europe/Paris") to be set for all time-related operations.

Sets the timezone to be used for time calculations and formatting. This method allows the application to adjust displayed times according to a specific timezone.

10.8 array

`array(A)`

Parameters:

- `A <Iterable>`: The iterable or array-like object to convert.

Returns: `<Array>`: A new array containing the elements from the input.
Converts an iterable or array-like object into a standard array.

`end(A)`

Parameters:

- `A <Array>`: The array from which the last element is to be retrieved.

Returns: `<*>`: The last element of the array. If the array is empty, returns undefined.
Retrieves the last element from the provided array.

`endi(A)`

Parameters:

- `A <Array>`: The array to evaluate.

Returns: `<number>`: The index of the last element.
Returns the index of the last element in the array.

`column(A, col)`

Parameters:

- `A <Array>`: The matrix array.
- `col <number>`: The column index to retrieve.

Returns: `<Array>`: The specified column as an array.
Retrieves a specific column from the provided matrix.

`row(A, row)`

Parameters:

- `A <Array>`: The matrix array.
- `row <number>`: The row index to retrieve.

Returns: `<Array>`: The specified row as an array.
Retrieves a specific row from the provided matrix.

```
first(A, [N])
```

Parameters:

- `A <Array>`: The input array.
- `N <number>`: The number of elements to return from the start.

Returns: `<Array>`: - An array containing the first N elements.
Returns the first N elements from an array.

```
last(A, [N])
```

Parameters:

- `A <Array>`: The input array.
- `N <number>`: The number of elements to return from the end.

Returns: `<Array>`: - An array containing the last N elements.
Returns the last N elements from an array.

```
index(rows, cols, rows_max)
```

Parameters:

- `rows <number>`: The row index or array of row indices.
- `cols <number>`: The column index or array of column indices.
- `rows_max <number>`: The maximum number of rows.

Returns: `<Array.<number>>`: An array of calculated indices.
Generates an array of indices based on rows and columns.

```
indexOfAll(A, value)
```

Parameters:

- `A <Array>`: The array to search.
- `value <*>`: The value to find.

Returns: `<Array.<number>>`: An array of indices where the value is found.
Finds all indices of a specified value in the array.

```
indexOf(A, value)
```

Parameters:

- `A <Array>`: Array to search.
- `value <*>`: Value to locate.

Returns: `<number>`: Index of value or -1.
Returns the index of a value in an array.

```
indexOfMulti(A, search_elements, [from_index])
```

Parameters:

- `A <Array>`: The array to search.
- `search_elements <Array>`: The sequence of elements to find.
- `from_index <number>`: The index to start the search from.

Returns: `<number>`: The starting index of the found sequence, or -1 if not found.
Finds the index of a sequence of elements in the array.

```
shuffleIndices(array)
```

Parameters:

- `array <Array>`: Array whose indices are to be shuffled.

Returns: `<Array>`: Shuffled array of indices.
Shuffles indices of an array using the Fisher-Yates algorithm.

```
setSub(A, indices, B)
```

Parameters:

- `A <Array>`: The target array.
- `indices <Array.<number>>`: The indices at which to set values.
- `B <Array>`: The array of values to set.

Sets a subset of array elements based on provided indices.

`setSubB(A, b, B)`

Parameters:

- `A <Array>`: The target array to modify.
- `b <Array.<boolean>>`: A boolean array indicating which elements to set (true = set, false = skip).
- `B <Array>`: The array of values to set at the indices determined by `b`.

Sets a subset of array elements based on provided boolean indices.

`getSub(A, indices)`

Parameters:

- `A <Array>`: The source array.
- `indices <Array.<number>>`: The indices of elements to retrieve.

Returns: `<Array>`: An array containing the retrieved elements.
Retrieves a subset of array elements based on provided indices.

`getSubB(A, b)`

Parameters:

- `A <Array>`: The source array.
- `b <Array.<boolean>>`: A boolean array indicating which elements to retrieve.

Returns: `<Array>`: An array containing the retrieved elements.
Retrieves a subset of array elements based on provided indices.

`moveElement(A, from_index, to_index)`

Parameters:

- `A <Array>`: The array to modify.
- `from_index <number>`: The index of the element to move.
- `to_index <number>`: The target index where the element should be moved.

Returns: `<Array>`: The modified array with the element moved.
Moves an element within the array from one index to another.

`removeElement(A, index)`

Parameters:

- `A <Array>`: The array to modify.
- `index <number>`: The index of the element to remove.

Returns: `<Array>`: The array after the element has been removed.
Removes an element from the array at the specified index.

`removeElementByValue(A, value)`

Parameters:

- `A <Array>`: The array to modify.
- `value <*>`: The value to remove.

Returns: `<Array>`: The array after the value has been removed.
Removes the first occurrence of a specified value from the array.

`removeElementProp(A, B, [prop])`

Parameters:

- `A <Array.<Object>>`: The array to filter.
- `B <Array>`: The array of properties or values to remove.
- `prop <string>`: The property name to check in objects within A.

Returns: `<Array>`: The filtered array.
Removes elements from array A that have properties listed in array B.

`findIndexProp(A, property, value)`

Parameters:

- `A <Array.<Object>>`: The array to search.
- `property <string>`: The property name to compare.
- `value <*>`: The value to match.

Returns: `<number>`: The index of the matching object, or -1 if not found.
Finds the index of the first object in the array where the specified property matches the given value.

`setValueAt(A, indices, value)`

Parameters:

- `A <Array>`: The target array.
- `indices <Array.<number>>`: An array of indices representing the position.
- `value <*>`: The value to set.

Returns: `<*>`: The value that was set.
Sets a value at the specified multi-dimensional indices in the array.

`getValueAt(A, indices)`

Parameters:

- `A <Array>`: The source array.
- `indices <Array.<number>>`: An array of indices representing the position.

Returns: `<*>`: The value at the specified indices, or undefined if out of bounds.
Retrieves a value from the array at the specified multi-dimensional indices.

`setVal(A, indices, value)`

Parameters:

- `A <Array>`: The target array.
- `indices <Array.<number>>`: An array of indices representing the position.
- `value <*>`: The value to set.

Returns: `<*>`: The value that was set.
Alias for setValueAt.

`getVal(A, indices)`

Parameters:

- `A <Array>`: The source array.
- `indices <Array.<number>>`: An array of indices representing the position.

Returns: `<*>`: The value at the specified indices, or undefined if out of bounds.
Alias for getValueAt.

`arrayIntersect(A, B)`

Parameters:

- `A <Array>`: First array.
- `B <Array>`: Second array.

Returns: `<Array>`: Common elements.
Returns the intersection of two arrays.

`cloneArray(A)`

Parameters:

- `A <Array>`: The array to clone.

Returns: `<Array>`: A new array containing all elements from A.
Creates a shallow copy of the provided array.

`createArray(length, [dimensions])`

Parameters:

- `length <number>`: The size of the first dimension.
- `dimensions <number>`: Sizes of subsequent dimensions.

Returns: `<Array>`: The newly created n-dimensional array.
Creates an n-dimensional array.

`createFilledArray(length, val, [dimensions])`

Parameters:

- `length <number>`: The size of the first dimension.
- `val <*>`: The value to fill the array with.
- `dimensions <number>`: Sizes of subsequent dimensions.

Returns: `<Array>`: The filled n-dimensional array.
Creates an n-dimensional array filled with a specific value.

`fill(val, A, length)`

Parameters:

- `val <*>`: The value to fill the array with.
- `A <Array>`: The array to fill.
- `length <number>`: The number of elements to fill.

Fills the array with a specific value.

`NaNs(size)`

Parameters:

- `size <number>`: The size of each dimension.

Returns: `<Array>`: The NaN-filled n-dimensional array.
Creates an n-dimensional array filled with NaN.

`zeros(size)`

Parameters:

- `size <number>`: The size of each dimension.

Returns: `<Array>`: The zero-filled n-dimensional array.
Creates an n-dimensional array filled with zeros.

`ones(size)`

Parameters:

- `size <number>`: The size of each dimension.

Returns: `<Array>`: The one-filled n-dimensional array.
Creates an n-dimensional array filled with ones.

`eye(size)`

Parameters:

- `size <number>`: The size of the identity matrix.

Returns: `<Array>`: The identity matrix as a 2D array.
Creates a 2-dimensional identity matrix.

`scale(A, s)`

Parameters:

- `A <Array.<number>>`: The array to scale.
- `s <number>`: The scalar value.

Returns: `<Array.<number>>`: The scaled array.
Scales an array by a scalar.

```
linspace(x1, x2, [N])
```

Parameters:

- `x1 <number>`: The start value.
- `x2 <number>`: The end value.
- `N <number>`: The number of points.

Returns: `<Array.<number>>`: The linearly spaced vector.
Creates a linearly spaced vector.

```
range(args)
```

Parameters:

- `args <*>`: The start, end, and optional step for the range.

Returns: `<Array.<number>>`: An array containing numbers within the specified range.
Generates an array of numbers within a specified range.

```
colon(x1, x2, dx)
```

Parameters:

- `x1 <number>`: The starting value of the sequence.
- `x2 <number>`: The ending value of the sequence.
- `dx <number>`: The increment between values in the sequence.

Returns: `<Array.<number>>`: An array of numbers from `x1` to `x2` with step size `dx`.
Generates a sequence of numbers from `x1` to `x2` with increments of `dx`.

```
elementWise(func, arrays)
```

Parameters:

- `func <function>`: The function to apply to the elements.

- `arrays <Array>`: One or more arrays to process.

Returns: `<Array>`: A new array with the function applied to each corresponding element.
Applies a function to corresponding elements of one or more arrays.

```
arrayfun(A, dim, fun)
```

Parameters:

- `A <Array>`: The array or matrix to process.
- `dim <number>`: The dimension along which to apply the function.
- `fun <function>`: The function to apply to each element.

Returns: `<Array>`: The result of applying the function to A.
Applies a function to each element of an array or matrix along a specified dimension.

```
divideEl(x, y)
```

Parameters:

- `x <Array>`: The numerator array or matrix.
- `y <Array>`: The denominator array or matrix.

Returns: `<Array>`: The result of the element-wise division.
Performs element-wise division on two arrays or matrices.

```
multiplyEl(x, y)
```

Parameters:

- `x <Array>`: The first array or matrix.
- `y <Array>`: The second array or matrix.

Returns: `<Array>`: The result of the element-wise multiplication.
Performs element-wise multiplication on two arrays or matrices.

```
powEl(x, y)
```

Parameters:

- `x <Array>`: The base array or matrix.
- `y <Array>`: The exponent array, matrix, or scalar.

Returns: `<Array>`: The result of the element-wise exponentiation.
Raises elements of an array or matrix to the power of elements in another array or matrix, element-wise.

`dot(x, y, [cols])`

Parameters:

- `x <Array.<number>>`: The first input vector or flat array.
- `y <Array.<number>>`: The second input vector or flat array.
- `cols <number>`: Optional number of columns to reshape the inputs into matrices.

Returns: `<number>`: The resulting dot product, either as a scalar or a matrix.
Calculates the dot product of two vectors or matrices.

`lZero(e)`

Parameters:

- `e <number>`: The element to check.

Returns: `<boolean>`: True if greater than zero, otherwise false.
Determines if the element is greater than zero.

`b2i(arr)`

Parameters:

- `arr <Array.<boolean>>`: The input array of boolean values.

Returns: `<Array.<number>>`: An array of indices corresponding to `true` values in the input array.
Converts a boolean array into an array of indices where the values are `true`.

`average(arr)`

Parameters:

- `arr <Array.<number>>`: The array to average.

Returns: `<number>`: The average value.
Calculates the average value of an array.

`averageEM(data, alpha)`

Parameters:

- `data <Array.<number>>`: The data array.
- `alpha <number>`: The smoothing factor between 0 and 1.

Returns: `<Array.<number>>`: The Exponential Moving Average of the data.
Calculates the Exponential Moving Average of the data.

```
averageMoving(inputArray, windowSize)
```

Parameters:

- `inputArray <Array.<number>>`: The array of numbers to filter.
- `windowSize <number>`: The size of the moving window (number of elements to average).

Returns: `<Array.<number>>`: The filtered array with the same length as the input array.
Applies a moving average filter to an input array while keeping the output array the same size.

```
movmean(inputArray, windowSize)
```

Parameters:

- `inputArray <Array.<number>>`: The array of numbers to filter.
- `windowSize <number>`: The size of the moving window (number of elements to average).

Returns: `<Array.<number>>`: The filtered array with the same length as the input array.
Applies a moving average filter to an input array while keeping the output array the same size.

```
isequal(A1, A2)
```

Parameters:

- `A1 <Array>`: The first array.
- `A2 <Array>`: The second array.

Returns: `<boolean>`: True if arrays are equal, otherwise false.
Determines if two arrays are equal.

```
neg(A)
```

Parameters:

- `A <Array.<boolean>>`: The array to negate.

Returns: `<Array.<boolean>>`: The negated array.
Negates the boolean values in an array.

`all(A)`

Parameters:

- `A <Array.<boolean>>`: The array to check.

Returns: `<boolean>`: True if all elements are true, otherwise false.
Determines if all elements in an array evaluate to true.

`any(A)`

Parameters:

- `A <Array.<boolean>>`: The array to check.

Returns: `<boolean>`: True if any element is true, otherwise false.
Determines if any element in an array evaluates to true.

`arrayContains(arr, item)`

Parameters:

- `arr <Array>`: The array to search.
- `item <*>`: The item to search for.

Returns: `<boolean>`: True if the item is found, otherwise false.
Checks if the array contains a specified item.

`hasDuplicates(array)`

Parameters:

- `array <Array>`: The array to check for duplicates.

Returns: `<boolean>`: True if duplicates are found, false otherwise.
Checks if an array contains any duplicate elements.

`removeDuplicates(arr)`

Parameters:

- `arr <Array>`: The array from which duplicates are to be removed.

Returns: `<Array>`: An array containing only unique elements from the original array.
Removes duplicate values from an array.

`fliplr(array)`

Parameters:

- `array <Array>`: The matrix array to flip horizontally.

Returns: `<Array>`: The horizontally flipped matrix.
Reverses the order of elements in each row of a matrix.

`movelr(array, n)`

Parameters:

- `array <Array>`: The array to be modified.
- `n <number>`: The number of elements to move from the start to the end.

Returns: `<Array>`: The modified array with the first `n` elements moved to the end.
Moves the first `n` elements of the array to the end.

`plus(args)`

Parameters:

- `args <number>`: The operands, scalar or arrays.

Returns: `<number>`: The result of adding all operands.
Adds multiple operands, which can be either scalars or arrays. If multiple operands are arrays, they must be of the same length.

`add(args)`

Parameters:

- `args <number>`: The operands, scalar or arrays.

Returns: `<number>`: The result of adding all operands.
Adds multiple operands, which can be either scalars or arrays. If multiple operands are arrays, they must be of the same length.

`minus(args)`

Parameters:

- `args <number>`: The operands, scalar or arrays.

Returns: `<number>`: The result of subtracting all subsequent operands from the first one.
Subtracts multiple operands from the first one, which can be either scalars or arrays. If multiple operands are arrays, they must be of the same length.

```
subtract(args)
```

Parameters:

- `args <number>`: The operands, scalar or arrays.

Returns: `<number>`: The result of subtracting all subsequent operands from the first one.
Subtracts multiple operands from the first one, which can be either scalars or arrays. If multiple operands are arrays, they must be of the same length.

```
cross3D(A, B, cols)
```

Parameters:

- `A <Array.<number>>`: The first 3D vector.
- `B <Array.<number>>`: The second 3D vector.
- `cols <number>`: The number of columns (typically 1 for single vectors).

Returns: `<Array.<number>>`: The cross product vector.
Computes the cross product of two 3D vectors.

```
inv(A, size)
```

Parameters:

- `A <Array.<number>>`: The input matrix in a flat array format.
- `size <number>`: The number of rows and columns in the matrix.

Returns: `<Array.<number>>`: The inverted matrix as a flat array, or `false` if the matrix is non-invertible.
Computes the inverse of a matrix represented as a flat array.

```
concatRow(cols_C, args)
```

Parameters:

- `cols_C <number>`: The number of columns in the concatenated matrix.
- `args <Array>`: The matrices to concatenate.

Returns: `<Array>`: The concatenated matrix.
Concatenates multiple matrices row-wise.

`concatCol(rows_C, args)`

Parameters:

- `rows_C <number>`: The number of rows in the concatenated matrix.
- `args <Array>`: The matrices to concatenate.

Returns: `<Array>`: The concatenated matrix.
Concatenates multiple matrices column-wise.

`concat(args)`

Parameters:

- `args <Array>`: The vectors to concatenate.

Returns: `<Array>`: The concatenated vector.
Concatenates multiple vectors.

`repRow(A, rows)`

Parameters:

- `A <Array>`: The row vector to repeat.
- `rows <number>`: The number of times to repeat the row.

Returns: `<Array>`: The repeated row matrix.
Repeats a row vector multiple times.

`repCol(A, cols)`

Parameters:

- `A <Array>`: The column vector to repeat.
- `cols <number>`: The number of times to repeat the column.

Returns: `<Array>`: The repeated column matrix.
Repeats a column vector multiple times.

`sumRow(A, rows, cols)`

Parameters:

- `A <Array.<number>>`: The matrix array.
- `rows <number>`: The number of rows in the matrix.
- `cols <number>`: The number of columns in the matrix.

Returns: `<Array.<number>>`: An array containing the sum of each row.
Sums the elements of each row in a matrix.

```
sumCol(A, rows, cols)
```

Parameters:

- `A <Array.<number>>`: The matrix array.
- `rows <number>`: The number of rows in the matrix.
- `cols <number>`: The number of columns in the matrix.

Returns: `<Array.<number>>`: An array containing the sum of each column.
Sums the elements of each column in a matrix.

```
normRow(A, rows, cols)
```

Parameters:

- `A <Array.<number>>`: The matrix array.
- `rows <number>`: The number of rows in the matrix.
- `cols <number>`: The number of columns in the matrix.

Returns: `<Array.<number>>`: An array containing the norm of each row.
Calculates the Euclidean norm of each row in a matrix.

```
normCol(A, rows, cols)
```

Parameters:

- `A <Array.<number>>`: The matrix array.
- `rows <number>`: The number of rows in the matrix.
- `cols <number>`: The number of columns in the matrix.

Returns: `<Array.<number>>`: An array containing the norm of each column.
Calculates the Euclidean norm of each column in a matrix.

```
transpose(A, rows, cols)
```

Parameters:

- `A <Array.<number>>`: The matrix array to transpose.
- `rows <number>`: The number of rows in the original matrix.
- `cols <number>`: The number of columns in the original matrix.

Returns: `<Array.<number>>`: The transposed matrix array.
Transposes a matrix.

```
multiply(A, B, rows_A, cols_A, cols_B)
```

Parameters:

- `A <Array.<number>>`: The first matrix array.
- `B <Array.<number>>`: The second matrix array.
- `rows_A <number>`: The number of rows in matrix A.
- `cols_A <number>`: The number of columns in matrix A.
- `cols_B <number>`: The number of columns in matrix B.

Returns: `<Array.<number>>`: The resulting matrix array after multiplication.
Multiplies two matrices.

```
dotColumn(A, B, rows, cols)
```

Parameters:

- `A <Array.<number>>`: The first matrix array.
- `B <Array.<number>>`: The second matrix array.
- `rows <number>`: The number of rows in each matrix.
- `cols <number>`: The number of columns in each matrix.

Returns: `<Array.<number>>`: An array containing the dot product of each column.
Performs element-wise dot product on columns of two matrices.

```
diag(A, length)
```

Parameters:

- `A <Array.<number>>`: The array to form the diagonal.
- `length <number>`: The size of the square matrix.

Returns: `<Array.<number>>`: The diagonal matrix as a 1D array.
Creates a diagonal matrix from a given array.

`linsolve(A, B, N)`

Parameters:

- `A <Array.<number>>`: The coefficient matrix.
- `B <Array.<number>>`: The constant terms.
- `N <number>`: The size of the matrix (NxN).

Returns: `<Array.<number>>`: The solution vector.
Solves a linear system of equations using LU decomposition.

`reciprocal(A, length)`

Parameters:

- `A <Array.<number>>`: The array of numbers.
- `length <number>`: The number of elements in the array.

Returns: `<Array.<number>>`: An array containing the reciprocals of the original elements.
Computes the reciprocal of each element in the array.

`reshape(A, rows, cols)`

Parameters:

- `A <Array>`: The array to reshape.
- `rows <number>`: The number of rows in the new shape.
- `cols <number>`: The number of columns in the new shape.

Returns: `<Array>`: The reshaped array.
Reshapes an array into a new dimension.

`maxi(A)`

Parameters:

- `A <Array.<number>>`: The array to search.

Returns: `<Array>`: An array containing the max value and its index.
Finds the maximum element in the array and its index, excluding NaN values.

`mini(A)`

Parameters:

- `A <Array.<number>>`: The array to search.

Returns: `<Array>`: An array containing the min value and its index.
Finds the minimum element in the array and its index, excluding NaN values.

`sorti(A)`

Parameters:

- `A <Array.<number>>`: The array to sort.

Returns: `<Array>`: An array containing the sorted values and their original indices.
Sorts the array in ascending order with indices, excluding NaN values.

`weightedSum(ret, w1, v1, w2, v2)`

Parameters:

- `ret <Array.<number>>`: The array to store the result.
- `w1 <number>`: Weight for the first vector.
- `v1 <Array.<number>>`: The first vector.
- `w2 <number>`: Weight for the second vector.
- `v2 <Array.<number>>`: The second vector.

Computes the weighted sum of two vectors and stores the result in the `ret` array.

`condiff(A)`

Parameters:

- `A <Array.<number>>`: The input array of numbers.

Returns: `<Array.<number>>`: An array containing the differences between consecutive elements of the input array.
Computes the consecutive differences of elements in an array.

`arrayRand(l, u, rows, cols, [randFun])`

Parameters:

- `l <number>`: The lower bound of the range.
- `u <number>`: The upper bound of the range.
- `rows <number>`: The number of rows.
- `cols <number>`: The number of columns.
- `randFun <function>`: The random function to use.

Returns: `<Array.<number>>`: An array filled with random numbers within the specified range.
Generates an array with random floating-point numbers within a specified range.

```
arrayRandi(lu, rows, cols, [randFun])
```

Parameters:

- `lu <Array.<number>>`: An array containing the lower and upper bounds [lower, upper].
- `rows <number>`: The number of rows.
- `cols <number>`: The number of columns.
- `randFun <function>`: The random function to use.

Returns: `<Array.<number>>`: An array filled with random integers within the specified range.
Generates an array with random integer numbers within a specified range.

```
normalizeVector(v)
```

Parameters:

- `v <Array.<number>>`: The vector to normalize.

Returns: `<Array.<number>>`: The normalized vector.
Normalizes a 3D vector.

```
dotVector(a, b)
```

Parameters:

- `a <Array.<number>>`: The first vector.
- `b <Array.<number>>`: The second vector.

Returns: `<number>`: The dot product of the vectors.
Computes the dot product of two 3D vectors.

`angleVectors(a, b)`

Parameters:

- `a <Array.<number>>`: The first vector.
- `b <Array.<number>>`: The second vector.

Returns: `<number>`: The angle in radians between vectors a and b.
Calculates the angle between two vectors.

`skewVector(v)`

Parameters:

- `v <Array.<number>>`: The vector to skew.

Returns: `<Array.<number>>`: The skew-symmetric matrix as a 1D array.
Creates a skew-symmetric matrix from a 3D vector.

`meshgrid(args)`

Parameters:

- `args <Array.<number>>`: The coordinate vectors.

Returns: `<Array>`: The coordinate grids for each dimension.
Generates coordinate matrices from coordinate vectors for N dimensions.

`areEqual(A1, A2)`

Parameters:

- `A1 <Array>`: The first array to compare.
- `A2 <Array>`: The second array to compare.

Returns: `<boolean>`: True if the arrays are equal, otherwise false.
Determines if two arrays are equal by comparing each element.

`dispMatrix(varname, A, rows, cols)`

Parameters:

- `varname <string>`: The name of the variable.
- `A <Array.<number>>`: The matrix array.
- `rows <number>`: The number of rows in the matrix.
- `cols <number>`: The number of columns in the matrix.

Returns: `<string>`: The string representation of the matrix.
Displays a matrix with a variable name.

```
dispRowVector(varname, A, length)
```

Parameters:

- `varname <string>`: The name of the variable.
- `A <Array.<number>>`: The row vector array.
- `length <number>`: The length of the row vector.

Returns: `<string>`: The string representation of the row vector.
Displays a row vector with a variable name.

```
dispColumnVector(varname, A, length)
```

Parameters:

- `varname <string>`: The name of the variable.
- `A <Array.<number>>`: The column vector array.
- `length <number>`: The length of the column vector.

Returns: `<string>`: The string representation of the column vector.
Displays a column vector with a variable name.

10.9 color

```
colororder
```

Type: `<Unknown>`
Color order

```
color(id)
```

Parameters:

- `id <number>`: Identifier for the color. Can be a numeric index, a predefined color name, or an RGB array.

Returns: `<string>`: The hex code of the color.
Returns a color code based on the provided identifier.

```
getColorG2R(value, [k])
```

Parameters:

- `value <number>`: A number between 0 and 1 indicating position on the gradient.
- `k <number>`: A scaling factor to adjust the gradient effect.

Returns: `<string>`: The hsl color string.
Calculates a color on a gradient from green to red based on a value.

```
colourGradientor(p, rgb_beginning, rgb_end)
```

Parameters:

- `p <number>`: The percentage (0 to 1) between the two colors.
- `rgb_beginning <Array.<number>>`: The RGB values of the start color.
- `rgb_end <Array.<number>>`: The RGB values of the end color.

Returns: `<Array.<number>>`: The RGB values of the calculated gradient color.
Calculates the gradient color between two colors based on a percentage.

```
rgb2hex(r, g, b)
```

Parameters:

- `r <number>`: The red color value (0-255).
- `g <number>`: The green color value (0-255).
- `b <number>`: The blue color value (0-255).

Returns: `<Array.<number>>`: The HEX representation of the color.
Converts RGB color values to HEX.

```
rgbToHsl(r, g, b)
```

Parameters:

- `r <number>`: The red color value (0-255).

- `g <number>`: The green color value (0-255).
- `b <number>`: The blue color value (0-255).

Returns: `<Array.<number>>`: The HSL representation of the color.
Converts RGB color values to HSL (Hue, Saturation, Lightness).

```
hslToRgb(h, s, l)
```

Parameters:

- `h <number>`: The hue value (0-360).
- `s <number>`: The saturation value (0-100).
- `l <number>`: The lightness value (0-100).

Returns: `<Array.<number>>`: The RGB representation of the color.
Converts HSL color values to RGB.

```
hueToRgb(v1, v2, vh)
```

Parameters:

- `v1 <number>`: Helper value 1.
- `v2 <number>`: Helper value 2.
- `vh <number>`: The hue value to convert.

Returns: `<number>`: The RGB value for the hue.
Helper function for converting a hue to RGB.

10.10 conversion

```
speak(msg)
```

Parameters:

- `msg <string>`: The text message to be spoken.

Converts text to speech using the Web Speech API.

```
num2str(num, precision)
```

Parameters:

- `num <number>`: The number to convert.
- `precision <number>`: The number of digits after the decimal point.

Returns: `<string>`: The formatted string.
Converts a number to a string with specified precision.

`changeExtension(path, ext_new)`

Parameters:

- `path <string>`: The original file path.
- `ext_new <string>`: The new extension without the dot.

Returns: `<string>`: The file path with the new extension.
Changes the extension of a file path.

`removeExtension(path)`

Parameters:

- `path <string>`: The original file path.

Returns: `<string>`: The file path without extension.
Remove the extension of a file path.

`ned2rpy(roll, pitch, yaw, [A])`

Parameters:

- `roll <number>`: The roll angle in radians.
- `pitch <number>`: The pitch angle in radians.
- `yaw <number>`: The yaw angle in radians.
- `A <Array.<number>>`: Optional matrix to apply transformation to.

Returns: `<Array.<Array.<number>>>`: The transformation matrix.
Transforms coordinates from NED (North, East, Down) frame to RPY (Roll, Pitch, Yaw) frame.

`uint8ToString(data)`

Parameters:

- `data <Uint8Array>`: The array of uint8 numbers.

Returns: `<string>`: The converted string.
Converts an array of uint8 numbers to a string.

`hex2dec(hex)`

Parameters:

- `hex <Array.<string>>`: The array of hexadecimal strings.

Returns: `<Array.<number>>`: The array of decimal numbers.
Converts an array of hexadecimal strings to an array of decimal numbers.

`numToASCII(num)`

Parameters:

- `num <number>`: The number to convert.

Returns: `<string>`: The ASCII character.
Converts a number to its ASCII character equivalent.

`numToHexStr(num, dig, prefix)`

Parameters:

- `num <number>`: The number to convert.
- `dig <number>`: The number of digits in the resulting string.
- `prefix <boolean>`: Whether to add a '0x' prefix.

Returns: `<string>`: The hexadecimal string.
Converts a number to a hexadecimal string with a fixed number of digits.

`int8To2ASCII(num)`

Parameters:

- `num <number>`: The int8 number.

Returns: `<string>`: The two ASCII characters.
Converts an int8 number to two ASCII characters.

`int16To4ASCII(num)`

Parameters:

- `num <number>`: The int16 number to convert.

Returns: `<string>`: A string of four ASCII characters.
Converts an int16 number to four ASCII characters representing its hexadecimal value.

`uint8sToInt16(part1, part2)`

Parameters:

- `part1 <number>`: The first uint8 value.
- `part2 <number>`: The second uint8 value.

Returns: `<number>`: The combined int16 value.
Combines two uint8 values into an int16 value.

`uint8sToInt32(part1, part2, part3, part4)`

Parameters:

- `part1 <number>`: The first uint8 value.
- `part2 <number>`: The second uint8 value.
- `part3 <number>`: The third uint8 value.
- `part4 <number>`: The fourth uint8 value.

Returns: `<number>`: The combined int32 value.
Combines four uint8 values into an int32 value.

`uint8sToFloat(part1, part2, part3, part4)`

Parameters:

- `part1 <number>`: The first uint8 value.
- `part2 <number>`: The second uint8 value.
- `part3 <number>`: The third uint8 value.
- `part4 <number>`: The fourth uint8 value.

Returns: `<number>`: The floating-point number.
Converts four uint8 values into a floating-point number.

`uint16ToInt16(num)`

Parameters:

- `num <number>`: The uint16 value to convert.

Returns: `<number>`: The converted int16 value.
Converts a uint16 value to an int16 value.

`uint8To2ASCII(num)`

Parameters:

- `num <number>`: The uint8 number to convert.

Returns: `<string>`: A string containing two ASCII characters representing the hexadecimal value.
Converts a uint8 number to two ASCII characters.

`ms2time(ms)`

Parameters:

- `ms <number>`: The time in milliseconds.

Returns: `<string>`: The time string.
Converts milliseconds to a time string in mm:ss format.

`dec2bin(x)`

Parameters:

- `x <number>`: The decimal number.

Returns: `<string>`: The binary string.
Converts a decimal number to a binary string.

`dec2hex(x)`

Parameters:

- `x <number>`: The decimal number.

Returns: `<string>`: The hexadecimal string.
Converts a decimal number to a hexadecimal string.

`dec2oct(x)`

Parameters:

- `x <number>`: The decimal number.

Returns: `<string>`: The octal string.
Converts a decimal number to an octal string.

```
round(number, [decimals], [string])
```

Parameters:

- `number <number>`: The number to round.
- `decimals <number>`: The number of decimal places.
- `string <boolean>`: Whether to return the result as a string.

Returns: `<number>`: The rounded number.
Rounds a number to a specified number of decimal places.

```
roundIf(value, p)
```

Parameters:

- `value <number>`: The value to round.
- `p <number>`: The number of digits after the decimal point.

Returns: `<number>`: The rounded number with a fixed number of decimal places, or the original value if it is not a number.
Rounds a number to a fixed number of decimal places if it is a number.

```
roundIfPrec(value, p)
```

Parameters:

- `value <number>`: The value to round.
- `p <number>`: The number of digits after the decimal point.

Returns: `<number>`: The rounded number with a fixed number of decimal places, or the original value if it is not a number.
Rounds a number to a fixed number of decimal places if it is a number.

```
bitString(n)
```

Parameters:

- `n <number>`: The number to convert.

Returns: `<string>`: A bit string representing the number.
Converts a uint8_t number to a bit string.

`getBitFlags(map, name_column, val)`

Parameters:

- `map <Object>`: The mapping of bit positions to flag names.
- `name_column <string>`: The column name in the mapping that contains the flag names.
- `val <number>`: The bitfield value.

Returns: `<Object>`: An object with keys as flag names and values indicating the presence (1) or absence (0) of each flag.
Generates a set of flags from a bitfield based on a mapping.

`getEnumVal(enum_object, prop, val)`

Parameters:

- `enum_object <Object>`: The enumeration object to search.
- `prop <string>`: The property name to match.
- `val <*>`: The property value to match.

Returns: `<number>`: The enumeration key as a number, or the index if not found.
Retrieves the enumeration value based on a property match.

`invertEnum(enum_object, [prop])`

Parameters:

- `enum_object <Object>`: The enumeration object to invert.
- `prop <string>`: An optional property name to use from the enumeration values.

Returns: `<Object>`: The inverted enumeration object.
Inverts an enumeration, swapping keys and values, optionally based on a specific property of the enumeration values.

`arrayToHexStr(A, [prefix])`

Parameters:

- `A <Array.<number>>`: The array to convert.

- `prefix <boolean>`: Whether to add a "0x" prefix to each hex value.

Returns: `<string>`: A string of hexadecimal values.
Converts an array of numbers to a string of hexadecimal values, optionally prefixed with "0x".

`arrayToASCII(array)`

Parameters:

- `array <Array.<number>>`: The array of numbers to convert.

Returns: `<string>`: The ASCII string representation of the array.
Converts an array of numbers to an ASCII string.

`extend(objects)`

Parameters:

- `objects <Object>`: The objects to merge into the target object.

Returns: `<Object>`: The extended object.
Extends an object with properties from additional objects.

`normalizeRC(rc, [deadzone])`

Parameters:

- `rc <number>`: The RC input value.
- `deadzone <number>`: The deadzone value below which the output is set to zero.

Returns: `<number>`: The normalized value.
Normalizes the value of a radio control (RC) input.

`checkValueUpdate(data)`

Parameters:

- `data <Object>`: An object containing the current value and the last value.

Returns: `<boolean>`: True if the value has been updated; false otherwise.
Checks if a value has been updated and updates the last value if it has.

`resetValue(data)`

Parameters:

- `data <Object>`: The object whose value and last value properties will be reset.

Resets the value and last value properties of an object.

```
adcToNewtons(adc_count, load_cell_capacity, [adc_resolution], [adc_gain],
[adc_sensitivity], [adc_bipolar])
```

Parameters:

- `adc_count <number>`: The raw ADC count value.
- `load_cell_capacity <number>`: The capacity of the load cell in Newtons.
- `adc_resolution <number>`: The ADC resolution in bits.
- `adc_gain <number>`: The gain applied to the ADC.
- `adc_sensitivity <number>`: The ADC sensitivity in mV/V.
- `adc_bipolar <boolean>`: Indicates if the ADC is bipolar.

Returns: `<number>`: The calculated force in Newtons.
Converts an ADC count to force in Newtons.

```
data2blobUrl(path)
```

Parameters:

- `path <string>`: Relative path from the application's base path to the file.

Returns: `<string>`: A blob URL representing the file's data.
Converts file data from a given path to a blob URL.

```
trbl2xy(cont_width, cont_height, width, height, top, right, bottom, left)
```

Parameters:

- `cont_width <number>`: Container width.
- `cont_height <number>`: Container height.
- `width <number>`: Element width.
- `height <number>`: Element height.
- `top <number>`: Top margin.
- `right <number>`: Right margin.
- `bottom <number>`: Bottom margin.

- `left <number>`: Left margin.

Returns: `<Array>`: Array containing x and y coordinates.
Converts top, right, bottom, and left margins into x and y coordinates.

```
simpleObj2Csv(data, [delimiter])
```

Parameters:

- `data <Object>`: The object containing arrays of data. Each key will be a column header.
- `delimiter <string>`: The delimiter to use for separating entries in the CSV (defaults to a comma).

Returns: `<string>`: The generated CSV as a string, with each row representing an entry and each column representing data from the corresponding key in the input object.
Generates a CSV string from an simple object containing arrays of values. Each key in the object represents a column in the CSV. This function handles uneven array lengths by filling missing values with an empty string.

```
simpleArray2Csv(data, [delimiter])
```

Parameters:

- `data <Array>`: An array of arrays to be converted into CSV format.
- `delimiter <string>`: The delimiter to separate the values in the CSV.

Returns: `<string>`: The formatted CSV string.
Converts a 2D array into a CSV string format.

10.11 device

```
checkDriver(driver_name)
```

Parameters:

- `driver_name <string>`: Name of the driver to check.

Returns: `<boolean>`: True if the driver is found, false otherwise.
Checks if a specific driver is installed on the system.

```
checkDriverFTDI()
```

Returns: `<boolean>`: True if the drivers are found, false otherwise.

Checks if the drivers for FTDI devices are installed.

`checkDriverCP210x()`

Returns: `<boolean>`: True if the drivers are found, false otherwise.
Checks if the drivers for Silicon Labs CP210x USB to UART bridge are installed.

`checkDriverCH340()`

Returns: `<boolean>`: True if the drivers are found, false otherwise.
Checks if the drivers for CH340 USB to serial converter are installed.

`checkArduino()`

Returns: `<boolean>`: True if available.
Check if Arduino CLI is available.

`compileArduino(dir)`

Parameters:

- `dir <string>`: Project directory.

Returns: `<Object>`: Compilation result or false on error.
Compile Arduino project.

`uploadArduino(dir, [port])`

Parameters:

- `dir <string>`: Project directory.
- `port <string>`: Optional port.

Returns: `<Object>`: Upload result or false on error.
Upload Arduino project.

`parseArduinoOtuput(output)`

Parameters:

- `output <object>`:

Returns: `<object>`:
Parses Arduino output from stdout or stderr.

`getGamepads()`

Returns: `<Array.<Object>>`: An array of connected gamepad objects.
Retrieves the current state of all connected gamepads.

`onGamepadConnected(callback)`

Parameters:

- `callback` `<function>`: The function to execute when a gamepad connects.

Registers a callback function to be called when a gamepad is connected.

`onGamepadDisconnected(callback)`

Parameters:

- `callback` `<function>`: The function to execute when a gamepad disconnects.

Registers a callback function to be called when a gamepad is disconnected.

`getGamepad(id, dt)`

Parameters:

- `id` `<number>`: The index of the gamepad to retrieve.
- `dt` `<number>`: Data reading interval in milliseconds.

Returns: `<PRDC_JSLAB_DEVICE_GAMEPAD>`: The corresponding gamepad object.
Retrieves a specific gamepad by its ID.

`getWebcams()`

Returns: `<Array.<Object>>`: A promise that resolves to an array of video input devices.
Retrieves a list of available webcam (video input) devices.

`getMicrophones()`

Returns: `<Array.<Object>>`: A promise that resolves to an array of audio input devices.
Retrieves a list of available microphone (audio input) devices.

`getAudioOutputs()`

Returns: `<Array.<Object>>`: A promise that resolves to an array of audio output devices.
Retrieves a list of available audio output devices.

```
webcam(device_id)
```

Parameters:

- `device_id <string>`: The unique identifier of the webcam device to use.

Returns: `<Promise.<WebcamResult>>`: An object containing the window instance, video element, and media stream.
Opens a new window to display the webcam feed from the specified device.

```
webcamCapture(opts, frameCallback, [editCallback])
```

Parameters:

- `opts <Object>`: Configuration options for webcam capture.
- `frameCallback <function>`: Callback invoked with each frame's image data buffer.
- `editCallback <function>`: Optional callback to edit each frame before processing.

Initiates webcam video capture.

```
getDesktopSources()
```

Returns: `<Array.<DesktopSource>>`: An array of desktop sources.
Retrieves desktop sources from the current environment.

```
showDesktopSources()
```

Returns: `<void>`:
Displays the available desktop sources by generating and injecting HTML elements for each source.

```
desktopCapture(opts, frameCallback, [editCallback])
```

Parameters:

- `opts <Object>`: Configuration options for desktop capture.
- `frameCallback <function>`: Callback invoked with each frame's image data buffer.
- `editCallback <function>`: Optional callback to edit each frame before processing.

Initiates desktop screen capture.

`capture(opts, opts.id, opts.type, frameCallback, [editCallback])`

Parameters:

- `opts <Object>`: Configuration options for capturing.
- `opts.id <string>`: The ID of the media source.
- `opts.type <string>`: Type of capture ('webcam' or 'desktop').
- `frameCallback <function>`: Callback invoked with each frame's image data buffer.
- `editCallback <function>`: Optional callback to edit each frame before processing.

Returns: `<Object>`: An object containing control functions and resources for the capture session.
Captures media frames based on the provided options.

`getCameraResolutions(device_id)`

Parameters:

- `device_id <string>`: The camera device ID.

Returns: `<Promise.<Array.<Object>>>`: Supported resolutions.
Gets supported camera resolutions for a specific device.

`showAudioWaveform(device_id, [fftSize])`

Parameters:

- `device_id <string>`: The microphone device ID.
- `fftSize <number>`: FFT size for analysis.

Returns: `<Object>`: Controls to stop or reset the waveform.
Displays an audio waveform on the canvas.

`startVideoRecording(source, [opts])`

Parameters:

- `source <HTMLCanvasElement>`: Canvas element, webcam deviceId, or desktop sourceId to capture.
- `opts <Object>`: Optional settings: type ('canvas' — 'webcam' — 'desktop'), fps, mimeType, and videoBitsPerSecond.

Returns: `<MediaRecorder>`: - MediaRecorder that streams the capture and provides a helper to stop and save.
Records video from the specified canvas element, webcam deviceId, or desktop sourceId and returns a MediaRecorder augmented with an async stopRecording() that finalizes and saves the file.

10.12 serial_device

`listSerialPorts()`

Returns: `<Promise.<Array>>`: Resolves with an array of serial port info.
Retrieves all available serial ports.

`checkDeviceUSB(VID, PID)`

Parameters:

- `VID <string>`: Vendor ID of the USB device.
- `PID <string>`: Product ID of the USB device.

Returns: `<boolean>`: True if the device is found, false otherwise.
Checks if there is a USB device connected with the specified Vendor ID and Product ID.

`checkDeviceSTM([PID])`

Parameters:

- `PID <string>`: Product ID of the USB device, default is for Virtual COM Port.

Returns: `<boolean>`: True if the device is found, false otherwise.
Checks for a connected USB device by STM and an optional Product ID.

`checkDeviceCH340()`

Returns: `<boolean>`: True if the device is found, false otherwise.
Checks if there is a USB device connected using a CH340 chip.

`connectSerialPorts(port, [baudrate], [opts])`

Parameters:

- `port <string>`: Port path.
- `baudrate <number>`: Baud rate.

- `opts <object>`: Additional options.

Returns: `<SerialPort>`: The opened SerialPort instance.
Opens a serial port.

```
chooseSerialPort()
```

Returns: `<Promise.<(string|false)>>`:
Opens a window to choose a serial port.

```
chooseSerialOptions()
```

Returns: `<Promise.<(string|false)>>`:
Opens a window to choose serial options.

```
openSerialTerminal(port_path, [baudrate], [opts])
```

Parameters:

- `port_path <string>`: The identifier or path of the serial port to connect to.
- `baudrate <number>`: The communication speed in bits per second.
- `opts <Object>`: An optional configuration object for additional settings.

Returns: `<Promise.<Object>>`: A promise that resolves with the terminal context.
Opens a serial terminal.

```
chooseSerialTerminal([opts])
```

Parameters:

- `opts <Object>`: An optional configuration object for additional settings.

Returns: `<Promise.<(Object|undefined)>>`: A promise that resolves with the terminal context if a serial port is chosen.
Prompts the user to choose serial options and opens a serial terminal if a valid port is selected.

10.13 file_system

```
readFile(file_path)
```

Parameters:

- `file_path <string>`: Path to the file.

Returns: `<Buffer>`: The content of the file or false in case of an error.
Reads the content of a file at the specified path.

`getContentFromCharRange(filepath, range)`

Parameters:

- `filepath <string>`: Path to file
- `range <Array>`: Character range [start, end]

Returns: `<string>`: - Extracted substring
Extract substring from file using range

`writeFile(file_path, data)`

Parameters:

- `file_path <string>`: The path to the file where data will be written.
- `data <Buffer>`: The data to write to the file.

Returns: `<boolean>`: Returns true if the file was written successfully, false if an error occurred.
Writes data to a specified file synchronously. This method should overwrite the file if it already exists.

`deleteFile(file_path)`

Parameters:

- `file_path <string>`: The path to the file that should be deleted.

Returns: `<boolean>`: Returns true if the file was deleted successfully, false if an error occurred.
Deletes a specified file synchronously.

`readDir(folder)`

Parameters:

- `folder <string>`: The path to the directory.

Returns: `<Array.<string>>`: An array of filenames or false in case of an error.
Reads the contents of a directory synchronously.

`deleteDir(file_path)`

Parameters:

- `file_path <string>`: The path to the file that should be deleted.

Returns: `<boolean>`: Returns true if the file was deleted successfully, false if an error occurred.
Deletes a specified file synchronously.

`moveFile(source, destination)`

Parameters:

- `source <string>`: The path to the source file.
- `destination <string>`: The path to the destination file.

Moves a file from source to destination.

`copyFile(source, destination)`

Parameters:

- `source <string>`: The path to the source file.
- `destination <string>`: The path to the destination file.

Copies a file from source to destination.

`filesInFolder(folder, ext)`

Parameters:

- `folder <string>`: Path to the folder.
- `ext <string>`: File extension filter.

Returns: `<Array.<string>>`: Array of file paths matching the extension in the specified folder.
Lists files in a specified folder, optionally filtering by extension.

`allFilesInFolder(folder)`

Parameters:

- `folder <string>`: Path to the folder.

Returns: `<Array.<string>>`: Array of file names.
Lists all files in a specified folder

`chooseFile(options)`

Parameters:

- `options <Object>`: Configuration options for the dialog.

Returns: `<string>`: The selected file path(s) or an empty array if canceled.
Opens a dialog for the user to choose a file, synchronously.

`chooseFolder(options)`

Parameters:

- `options <Object>`: Configuration options for the dialog.

Returns: `<string>`: The selected folder path(s) or an empty array if canceled.
Opens a dialog for the user to choose a folder, synchronously.

`getDefaultPath(type)`

Parameters:

- `type <string>`: Type of the default path (e.g., 'root', 'documents').

Returns: `<string>`: The default path for the specified type.
Retrieves a default path based on a specified type.

`openFolder(filepath)`

Parameters:

- `filepath <string>`: Path to the folder.

Opens the specified folder in the system's file manager.

`makeDirectory(directory)`

Parameters:

- `directory <string>`: The path where the directory will be created.

Returns: `<boolean>`: True if the directory was successfully created or already exists, false if an error occurred.
Creates a directory at the specified path if it does not already exist. This method delegates the directory creation task to the environment's makeDirectory function.

```
mkdir(directory)
```

Parameters:

- `directory <string>`: The path where the directory will be created.

Returns: `<boolean>`: True if the directory was successfully created or already exists, false if an error occurred.
Alias for makeDirectory. Creates a directory at the specified path if it does not already exist. This method delegates the directory creation task to the environment's makeDirectory function.

```
openDir(filepath)
```

Parameters:

- `filepath <string>`: Path to the directory.

Opens the specified directory in the system's file manager. Alias for `openFolder`.

```
showFolder(filepath)
```

Parameters:

- `filepath <string>`: Path to the folder.

Shows the specified folder in the system's file manager. Alias for `openFolder`.

```
showDir(filepath)
```

Parameters:

- `filepath <string>`: Path to the directory.

Shows the specified directory in the system's file manager. Alias for `openDir`.

```
openProgramFolder()
```

Opens the program's root folder in the system's file manager.

```
showFileInFolder(filepath)
```

Parameters:

- `filepath <string>`: Path to the file.

Shows the specified file in its containing folder within the system's file manager.

```
showFileInDir(filepath)
```

Parameters:

- `filepath <string>`: Path to the file.

Shows the specified file in its containing directory within the system's file manager. Alias for `showFileInFolder`.

```
readcsv(filePath, delimiter)
```

Parameters:

- `filePath <string>`: Path to the CSV file.
- `delimiter <string>`: Delimiter used in the CSV file (e.g., ',', ';', '\{}t').

Returns: `<Array.<Object>>`: - Parsed CSV data as an array of objects.
Reads a CSV file and returns a promise that resolves with the parsed data.

```
checkFile(file)
```

Parameters:

- `file <string>`: The path to the file to check.

Checks if the specified file exists.

```
existFile(file)
```

Parameters:

- `file <string>`: The path to the file to check.

Checks if the specified file exists.

```
checkDirectory(directory)
```

Parameters:

- `directory <string>`: The path to the directory to check.

Checks if the specified directory exists.

`existDirectory(directory)`

Parameters:

- `directory <string>`: The path to the directory to check.

Checks if the specified directory exists.

`copyDir(src, dest)`

Parameters:

- `src <string>`: The source directory path.
- `dest <string>`: The destination directory path.

Recursively copies a directory from the source path to the destination path.

`copyFolder(src, dest)`

Parameters:

- `src <string>`: The source folder path.
- `dest <string>`: The destination folder path.

Copies a folder from the source path to the destination path.

`cp(src, dest)`

Parameters:

- `src <string>`: The source directory path.
- `dest <string>`: The destination directory path.

Copies a directory from the source path to the destination path.

`copyDir7z(src, dest)`

Parameters:

- `src <string>`: The source 7z archive path.
- `dest <string>`: The destination directory path.

Copies a 7z archive from the source path to the destination path, extracts it, and removes the archive.

10.14 system

`system(args)`

Parameters:

- `args <*>`: Command arguments.

Returns: `<string>`: The output of the command as a string, or false if an error occurred. Executes a system command and returns the output.

`exec(args)`

Parameters:

- `args <*>`: Command arguments.

Executes a system command.

`spawn(args)`

Parameters:

- `args <*>`: Command arguments.

Executes a system command.

`getTaskList()`

Returns: `<Array>`: An array containing the tasklist output or [false, []] if an error occurred. Retrieves a list of all running tasks on the system.

`isProgramRunning(program_name)`

Parameters:

- `program_name <string>`: The name of the program to check.

Returns: `<Array>`: An array where the first element is a boolean indicating if the program is running, and the second element is an array of process IDs if the program is running. Checks if a specific program is running and retrieves its process IDs.

`killProcess(pid)`

Parameters:

- `pid <number>`: The process ID of the process to kill.

Returns: `<string>`: The output of the kill command as a string, or false if an error occurred.
Attempts to kill a process by its process ID.

10.15 geography

`webmap(args)`

Parameters:

- `args <*>`: Arguments for configuring the web map.

Returns: `<Promise.<PRDC_JSLAB_GEOGRAPHY_MAP>>`: The initialized 2D web map instance.
Initializes and returns a new 2D web map instance.

`geoglobe(args)`

Parameters:

- `args <*>`: Arguments for configuring the 3D geoglobe.

Returns: `<Promise.<PRDC_JSLAB_GEOGRAPHY_MAP_3D>>`: The initialized 3D geoglobe instance.
Initializes and returns a new 3D geoglobe instance.

`calculateTilesBoundingBox(tile_coordinates)`

Parameters:

- `tile_coordinates <Array.<Object>>`: An array of objects with tile coordinates, each having properties `x`, `y`, and `z` for tile X and Y coordinates and zoom level, respectively.

Returns: `<Object>`: An object containing the bounds as an array of `[min_lat, min_lng]` and `[max_lat, max_lng]`, and the center as `[latitude, longitude]`.
Calculates the bounding box and center from an array of tile coordinates.

`offsetLatLon(lat, lon, distance, bearing)`

Parameters:

- `lat <number>`: The latitude of the starting point.
- `lon <number>`: The longitude of the starting point.

- `distance <number>`: The distance from the starting point in meters.
- `bearing <number>`: The bearing in degrees from north.

Returns: `<Array.<number>>`: An array containing the latitude and longitude of the calculated point.
Calculates a new latitude and longitude based on a starting point, distance, and bearing using the Haversine formula.

```
latLonDistance(lat1, lon1, lat2, lon2)
```

Parameters:

- `lat1 <number>`: The latitude of the first point.
- `lon1 <number>`: The longitude of the first point.
- `lat2 <number>`: The latitude of the second point.
- `lon2 <number>`: The longitude of the second point.

Returns: `<number>`: The distance between the two points in meters.
Calculates the distance between two points on Earth using the Haversine formula.

```
latLonAltDistance(lat1, lon1, alt1, lat2, lon2, alt2)
```

Parameters:

- `lat1 <number>`: The latitude of the first point.
- `lon1 <number>`: The longitude of the first point.
- `alt1 <number>`: The altitude of the first point in meters.
- `lat2 <number>`: The latitude of the second point.
- `lon2 <number>`: The longitude of the second point.
- `alt2 <number>`: The altitude of the second point in meters.

Returns: `<number>`: The 3D distance between the two points in meters.
Calculates the distance between two points on Earth including altitude difference using the Haversine formula.

```
checkNewLatLon(latlon)
```

Parameters:

- `latlon <Object>`: An object containing the current and last values of latitude and longitude.

Returns: `<boolean>`: True if the latitude and/or longitude values have been updated; false otherwise.
Checks if the latitude and longitude values have been updated.

`latLonAlt2cartesian(lat, lon, alt)`

Parameters:

- `lat <number>`: Latitude in degrees or array of latitudes.
- `lon <number>`: Longitude in degrees or array of longitudes.
- `alt <number>`: Altitude in meters or array of altitudes.

Returns: `<Cesium.Cartesian3>`: Cartesian coordinate(s).
Converts latitude, longitude, and altitude to Cartesian coordinates.

10.16 networking

`CRC_TABLE_XMODEM`

Type: `<Unknown>`
XMODEM CRC table

`isOnline()`

Returns: `<boolean>`: `true` if online, otherwise `false`.
Checks if the environment is currently online.

`crc16xmodem(byte_array)`

Parameters:

- `byte_array <Uint8Array>`: The input data as a byte array.

Returns: `<number>`: The CRC-16/XMODEM checksum as a numeric value.
Calculates the CRC-16/XMODEM checksum of a byte array.

`getIP()`

Returns: `<string>`: The IP address if found, otherwise an empty string.
Retrieves the primary IPv4 address of the current machine.

`pingAddressTCP(host, port, [timeout])`

Parameters:

- `host <string>`: The IP address or hostname to ping.
- `port <number>`: The port number to use for the connection.
- `timeout <number>`: The timeout in milliseconds before the attempt is considered failed.

Returns: `<Promise.<boolean>>`: A promise that resolves to `true` if the connection is successful, `false` otherwise.
Attempts to establish a TCP connection to the specified host and port to check reachability.

```
pingAddress(host, timeout)
```

Parameters:

- `host <string>`: The IP address or hostname to ping.
- `timeout <number>`: The timeout in milliseconds for the ping command.

Executes a ping command to check if an IP address is reachable.

```
pingAddressSync(host, timeout)
```

Parameters:

- `host <string>`: The IP address or hostname to ping.
- `timeout <number>`: The timeout in milliseconds for the ping command.

Executes a ping command synchronized to check if an IP address is reachable.

```
findFirstUnusedPort(port, min_port, max_port)
```

Parameters:

- `port <number>`: The starting port number to check.
- `min_port <number>`: The minimum port number in the range.
- `max_port <number>`: The maximum port number in the range.

Finds the first unused port within a specified range, checking sequentially from `port` to `max_port`.

```
ip2dec(ip, [subnets])
```

Parameters:

- `ip <string>`: The IPv4 address in dot-decimal notation.

- `subnets <number>`: The number of subnets in the IP address, default is 4.

Returns: `<number>`: The decimal equivalent of the IPv4 address.
Converts an IPv4 address to its decimal equivalent.

```
tcp(host, port, onConnectCallback)
```

Parameters:

- `host <string>`: The hostname or IP address to connect to.
- `port <number>`: The port number on the host to connect to.
- `onConnectCallback <function>`: A callback function that is called when the connection is successfully established.

Returns: `<PRDC_JSLAB_TCP_CLIENT>`: An instance of the TCP client with event handlers set up.
Creates a TCP client for communication with a specified host and port.

```
tcpServer(host, port, onConnectCallback)
```

Parameters:

- `host <string>`: The hostname or IP address.
- `port <number>`: The port number.
- `onConnectCallback <function>`: A callback function that is called when the connection is successfully established.

Returns: `<PRDC_JSLAB_TCP_SERVER>`: An instance of the TCP server.
Creates a TCP server to listen on a specified port.

```
udp(host, port)
```

Parameters:

- `host <string>`: The hostname or IP address to connect to.
- `port <number>`: The port number to connect to.

Returns: `<PRDC_JSLAB_UDP>`: An instance of the UDP client.
Creates a UDP client for sending data to a specified host and port.

```
udpServer(port)
```

Parameters:

- `port <number>`: The port number to listen on.

Returns: `<PRDC_JSLAB_UDP_SERVER>`: An instance of the UDP server.
Creates a UDP server to listen on a specified port.

```
videoCall(type, video_source_type, video_id, mic_id, tcp_host, tcp_port,
opts)
```

Parameters:

- `type <string>`: The call type ('server' or 'client').
- `video_source_type <string>`: The video source type ('webcam' or 'desktop').
- `video_id <string>`: The video device or source ID.
- `mic_id <string>`: The microphone device ID.
- `tcp_host <string>`: The TCP host address.
- `tcp_port <number>`: The TCP port number.
- `opts <object>`: Additional configuration options.

Returns: `<PRDC_JSLAB_VIDEOCALL>`: The created video call object.
Initializes and returns a new video call instance.

10.17 format

```
getContentType(filePath)
```

Parameters:

- `filePath <string>`: The path to the file.

Returns: `<string>`: The corresponding MIME type.
Retrieves the MIME type based on the file extension.

```
formatBytes(bytes, [decimals])
```

Parameters:

- `bytes <Number>`: Number of bytes.
- `decimals <Number>`: Number of decimal places to include in the formatted string.

Returns: `<String>`: Formatted bytes string with appropriate unit.
Formats the given byte count into a readable string.

`formatBPS(bps, [decimals])`

Parameters:

- `bps <Number>`: Number of bits per second.
- `decimals <Number>`: Number of decimal places to include in the formatted string.

Returns: `<String>`: Formatted bits per second string with appropriate unit.
Formats the given bits per second (bps) into a readable string.

`formatPrefix(number, [decimals])`

Parameters:

- `number <Number>`: The number to format.
- `decimals <Number>`: Number of decimal places to include in the formatted string.

Returns: `<String>`: Formatted number string with metric prefix.
Formats a number with metric prefixes (k, M, G, etc.) based on its value.

`formatNum(number, [decimals])`

Parameters:

- `number <Number>`: The number to format.
- `decimals <Number>`: The number of decimal places.

Returns: `<String>`: The formatted number as a string.
Formats a number to a specified number of decimal places.

`formatFloatDigits(number, [digits])`

Parameters:

- `number <number>`: The number to format.
- `digits <number>`: The number of significant digits.

Returns: `<string>`: The formatted number as a string.
Formats a floating-point number to have a fixed number of significant digits.

`clip(number, control_number, clip_value, [direction])`

Parameters:

- `number <number>`: The number to clip.
- `control_number <number>`: The reference number for the clipping condition.
- `clip_value <number>`: The value to clip to.
- `direction <boolean>`: The direction of clipping (true for max, false for min).

Returns: `<number>`: The clipped number.
Clips a number to a specified value based on a condition.

```
clipHeight(value, [min], [max])
```

Parameters:

- `value <number>`: The value to clip.
- `min <number>`: The minimum value of the range.
- `max <number>`: The maximum value of the range.

Returns: `<number>`: The clipped height value.
Clips a height value to a specified range.

```
fieldnames(obj)
```

Parameters:

- `obj <Object>`: The object from which to extract keys.

Returns: `<Array.<string>>`: An array containing the keys of the object.
Retrieves the field names (keys) of the given object.

```
strrep(str, old_string, new_string)
```

Parameters:

- `str <string>`: The original string.
- `old_string <string>`: The substring to be replaced.
- `new_string <string>`: The substring to replace with.

Returns: `<string>`: The resulting string after replacements.
Replaces all occurrences of a specified substring within a string.

```
clipVal(value, [min], [max])
```

Parameters:

- `value <number>`: The value to clip.
- `min <number>`: The minimum value of the range.
- `max <number>`: The maximum value of the range.

Returns: `<number>`: The clipped value.
Clips a value to a specified range.

`toFixedIf(value, p)`

Parameters:

- `value <Number>`: The value to round.
- `p <Number>`: The number of decimal places to round to.

Returns: `<Number>`: The rounded number, or the original number if it wasn't a float.
Rounds a number to a fixed number of decimal places, but only if it's a floating point number.

`getCircularReplacer()`

Returns: `<function>`: A replacer function that can be used with JSON.stringify to handle circular references.
Provides a replacer function for JSON.stringify() to prevent circular references.

`isNaN(value)`

Parameters:

- `value <*>`: The value to check.

Returns: `<boolean>`: True if the value is NaN, false otherwise.
Determines if the provided value is NaN.

`isObject(value, [ignore_array])`

Parameters:

- `value <*>`: The value to check.
- `ignore_array <boolean>`: Whether to consider arrays as not objects.

Returns: `<boolean>`: True if the value is an object, false otherwise.
Determines if the provided value is an object.

`isNumber(value)`

Parameters:

- `value <*>`: The value to check.

Returns: `<boolean>`: True if the value is a number, false otherwise.
Determines if the provided value is a number.

`isString(value)`

Parameters:

- `value <*>`: The value to check.

Returns: `<boolean>`: True if the value is a string, false otherwise.
Determines if the provided value is a string.

`isEmptyString(str)`

Parameters:

- `str <string>`: The string to check.

Returns: `<boolean>`: - True if the string is empty or contains only whitespace, otherwise false.
Checks if a string is empty or contains only whitespace.

`isFunction(value)`

Parameters:

- `value <*>`: The value to check

Returns: `<boolean>`: True if the value is a function, false otherwise.
Determines if the provided value is a function.

`isArray(value)`

Parameters:

- `value <*>`: The value to check.

Returns: `<boolean>`: True if the value is an array, false otherwise.
Determines if the provided value is an array.

`isNull(value)`

Parameters:

- `value <*>`: The value to check.

Returns: `<boolean>`: True if the value is null, false otherwise.
Determines if the provided value is null.

`isEmpty(array)`

Parameters:

- `array <Array>`: The array to check.

Returns: `<boolean>`: True if the array is empty, false otherwise.
Checks if an array is empty.

`isInfinity(value)`

Parameters:

- `value <number>`: The value or array of values to check.

Returns: `<boolean>`: - Returns true if infinite, otherwise false. Returns an array of booleans if input is an array.
Checks if the given value(s) are infinite.

`isNumeric(variable)`

Parameters:

- `variable <*>`: The variable to check.

Returns: `<boolean>`: True if the variable is numeric, false otherwise.
Checks if a variable is numeric.

`hasKey(object, key)`

Parameters:

- `object <Object>`: The object to check for the presence of the key.
- `key <string>`: The key to check for in the object.

Returns: `<boolean>`: - True if the object has the key, false otherwise.
Checks if the specified object has the given key.

`isUndefined(value)`

Parameters:

- `value <*>`: The value to check.

Returns: `<boolean>`: True if the value is undefined, false otherwise.
Determines if the provided value is undefined.

`isUUID(str)`

Parameters:

- `str <string>`: The string to check.

Returns: `<boolean>`: True if the string is a valid UUID, false otherwise.
Checks if a string is a valid UUID.

`normalizeAngle(angle)`

Parameters:

- `angle <number>`: The angle to normalize.

Returns: `<number>`: The normalized angle.
Normalizes an angle to the range of -180 to 180 degrees.

`normalizeAngle360(angle)`

Parameters:

- `angle <number>`: The angle to normalize.

Returns: `<number>`: The normalized angle.
Normalizes an angle to the range of 0 to 360 degrees.

`normalizeLat(latitude, precision)`

Parameters:

- `latitude <number>`: The latitude to normalize.
- `precision <number>`: The precision of the normalization.

Returns: `<number>`: The normalized latitude.
Normalizes latitude to the range of -90 to 90 degrees with specified precision.

`normalizeLon(longitude, precision)`

Parameters:

- `longitude <number>`: The longitude to normalize.

- `precision <number>`: The precision of the normalization.

Returns: `<number>`: The normalized longitude.
Normalizes longitude to the range of -180 to 180 degrees with specified precision.

`checkUndefined(val)`

Parameters:

- `val <*>`: The value to check.

Returns: `<*>`: The original value if not undefined, otherwise an empty string.
Checks a value for undefined and returns an empty string if it is undefined, otherwise returns the value.

`prettyPrint(data)`

Parameters:

- `data <*>`: The data to pretty-print.

Returns: `<Array>`: An array containing the pretty-printed data as a string and a boolean indicating if the data was an object.
Pretty-prints data, converting it into a more readable format for display. Handles strings, objects, and other data types.

`stringify(object)`

Parameters:

- `object <*>`: The object to stringify.

Returns: `<string>`: The JSON string representation of the object or the object itself if not an object.
Converts an object to a JSON string if it is an object, otherwise returns the object as is.

`safeStringify(data, [depth_limit])`

Parameters:

- `data <Object>`: The object to stringify.
- `depth_limit <number>`: The maximum depth to traverse in the object, beyond which the traversal is stopped.

Returns: `<string>`: A JSON string representation of the object, with special handling for deep objects, circular references, and HTML escaping of strings.
Safely serializes an object into a JSON string, handling circular references and deep structures, with depth control. It also escapes strings to prevent HTML injection.

`escapeHTML(string)`

Parameters:

- `string` `<string>`: The string to escape.

Returns: `<string>`: - The escaped string with HTML characters replaced.
Escapes special HTML characters in a string to prevent HTML injection.

`escapeLatex(string)`

Parameters:

- `string` `<string>`: The string to escape.

Returns: `<string>`: - The escaped string with LaTeX characters replaced.
Escapes special LaTeX characters in a string to prevent LaTeX injection.

`getUniqueKey(object, key)`

Parameters:

- `object` `<string>`: Object to add unique key.
- `key` `<string>`: The original object key.

Returns: `<string>`: A unique object key.
Generates a unique object key by appending a number to the original key if it already exists.

`randomString(num)`

Parameters:

- `num` `<number>`: The desired length of the random string.

Returns: `<string>`: A random string.
Generates a random string of the specified length.

`countDecimalPlaces(num)`

Parameters:

- `num <number>`: The number to evaluate.

Returns: `<number>`: The count of decimal places.
Calculates the number of decimal places in a number.

`replaceEditorLinks(text)`

Parameters:

- `text <string>`: The multiline error log text.

Returns: `<string>`: The updated text with file links replaced by HTML spans.
Replaces file links in a text with HTML span elements.

`numberValidator(n)`

Parameters:

- `n <HTMLInputElement>`: Target input element.

Returns: `<boolean>`: True when the value is valid.
Checks that an input contains a finite number

`limitedNumberValidator(n, [min], [max])`

Parameters:

- `n <HTMLInputElement>`: Target input element.
- `min <number>`: Minimum allowed value (optional).
- `max <number>`: Maximum allowed value (optional).

Returns: `<boolean>`: True when the value is valid.
Checks that an input contains a finite number and—if supplied—is within the inclusive range [min, max].

10.18 render

`debounceIn(func, wait)`

Parameters:

- `func <function>`: The function to debounce.
- `wait <number>`: The period to wait before allowing another call, in milliseconds.

Returns: `<function>`: The debounced function.
Debounces a function, ensuring it's only invoked once at the beginning of consecutive calls during the wait period.

```
debounceInOut(func, wait)
```

Parameters:

- `func <function>`: The function to debounce.
- `wait <number>`: The period to wait before allowing another call, in milliseconds.

Returns: `<function>`: The debounced function.
Debounces a function, calling it at the first and last of consecutive calls during the wait period.

```
debounceOut(func, wait)
```

Parameters:

- `func <function>`: The function to debounce.
- `wait <number>`: The period to wait before allowing another call, in milliseconds.

Returns: `<function>`: The debounced function.
Debounces a function, ensuring it's only invoked once at the end of consecutive calls during the wait period.

10.19 geometry

```
spaceSearch()
```

Creates an instance of PRDC_JSLAB_LIB_OPTIM_SPACE_SERACH.

```
findNearestPoints(points1, points2)
```

Parameters:

- `points1 <Array.<Array>>`: Array of points.
- `points2 <Array.<Array>>`: Reference points.

Returns: `<Array.<number>>`: Array of indices for nearest points.
Finds the nearest points in points2 for each point in points1.

```
pointLineDistance(P, A, i)
```

Parameters:

- `P <Array.<number>>`: point [Px, Py, Pz]
- `A <Array.<number>>`: a point on the line
- `i <Array.<number>>`: a unit direction vector of the line

Returns: `<Object>`: d - shortest distance P1 - point on the line with the smallest distance to P
Returns the shortest distance from point P to the line defined by (A, i) and the closest point (P1) on that line.

```
lineCircleIntersection(P, i, O, r)
```

Parameters:

- `P <Array.<number>>`: point on the line
- `i <Array.<number>>`: direction vector of the line (unit)
- `O <Array.<number>>`: center of the circle
- `r <number>`: circle radius

Returns: `<Object>`: flag = 0 -¿ intersection (two points) flag = 1 -¿ tangent (one point) flag = 2 -¿ no intersection
Returns the intersection points of a circle (center O, radius r) in a plane with a line passing through point P with direction i.

```
planesIntersection(P1, n1, P2, n2)
```

Parameters:

- `P1 <Array.<number>>`: a point in plane 1
- `n1 <Array.<number>>`: normal to plane 1
- `P2 <Array.<number>>`: a point in plane 2
- `n2 <Array.<number>>`: normal to plane 2

Returns: `<Object>`: flag = 0 -¿ planes intersect flag = 1 -¿ planes are the same flag = 2 -¿ planes are parallel (no intersection)
Returns the line (point P, direction i) that is the intersection of two planes, or indicates if they are the same or parallel.

```
isPointOnLine(P, A, B)
```

Parameters:

- P <Array.<number>>:
- A <Array.<number>>:
- B <Array.<number>>:

Returns: <number>: 1 (on segment), 0 (not on segment)
Checks if point P lies on the line segment A-B. Returns 1 if on segment, 0 otherwise.

`linesOverlap(P1, P2, P3, P4)`

Parameters:

- P1 <Array.<number>>:
- P2 <Array.<number>>:
- P3 <Array.<number>>:
- P4 <Array.<number>>:

Returns: <Object>:
Returns the overlapping segment (if any) between two segments P1-P2 and P3-P4, or indicates no overlap.

`minPointsDistance3D(P1i, P2i)`

Parameters:

- P1i <Array.<Array.<number>>>: Array of 3D points (e.g. [[x1, y1, z1], [x2, y2, z2], ...]).
- P2i <Array.<Array.<number>>>: Another array of 3D points.

Returns: <Object>: L - The minimal distance found. P1 - The point in P1i corresponding to the minimal distance. P2 - The point in P2i corresponding to the minimal distance.
Finds the minimal 3D distance between all pairs of points in two arrays.

`triangle(p1, p2, p3)`

Parameters:

- p1 <Array>: First vertex.
- p2 <Array>: Second vertex.
- p3 <Array>: Third vertex.

Returns: <PRDC_JSLAB_TRIANGLE>: Triangle instance.
Creates a new triangle instance.

`delaunayTriangulation(points)`

Parameters:

- `points` <Array.<Array>>: Array of points.

Returns: <Array.<PRDC_JSLAB_TRIANGLE>>: Array of triangles.
Performs Delaunay triangulation on a set of points.

`getRotationMatrix(a, b)`

Parameters:

- `a` <Array.<number>>: The initial unit vector.
- `b` <Array.<number>>: The target unit vector.

Returns: <Array.<Array.<number>>>: - The resulting rotation matrix.
Generates a rotation matrix to rotate from vector a to vector b.

`transform(coordinates, scale_factor, rotation_matrix, translation)`

Parameters:

- `coordinates` <Array.<Array.<number>>>: Array of coordinate points.
- `scale_factor` <number>: Factor by which to scale the coordinates.
- `rotation_matrix` <Array.<Array.<number>>>: Matrix used to rotate the coordinates.
- `translation` <Array.<number>>: Vector used to translate the coordinates.

Returns: <Array.<Array.<number>>>: - The transformed coordinates.
Transforms coordinates by scaling, rotating, and translating them.

`createVectors3D(xi, yi, zi, ui, vi, wi, scale, angle_factor, opts)`

Parameters:

- `xi` <Array.<number>>: X coordinates of vector origins.
- `yi` <Array.<number>>: Y coordinates of vector origins.
- `zi` <Array.<number>>: Z coordinates of vector origins.
- `ui` <Array.<number>>: X components of vectors.

- `vi <Array.<number>>`: Y components of vectors.
- `wi <Array.<number>>`: Z components of vectors.
- `scale <number>`: Scale factor for the vectors.
- `angle_factor <number>`: Angle factor for arrowheads.
- `opts <Object>`: Additional plotting options.

Returns: `<Object>`: - An object containing line and head trace data for plotting.
Creates 3D vectors for plotting based on provided parameters.

```
createDisks3D(xi, yi, zi, ri, ui, vi, wi, opts)
```

Parameters:

- `xi <Array.<number>>`: X coordinates of disk centers.
- `yi <Array.<number>>`: Y coordinates of disk centers.
- `zi <Array.<number>>`: Z coordinates of disk centers.
- `ri <Array.<number>>`: Radii of the disks.
- `ui <Array.<number>>`: X components of normal vectors (for disk orientation).
- `vi <Array.<number>>`: Y components of normal vectors (for disk orientation).
- `wi <Array.<number>>`: Z components of normal vectors (for disk orientation).
- `opts <Object>`: Additional plotting options.

Returns: `<Object>`: - An object containing line and area trace data for plotting.
Creates 3D disks for plotting based on provided parameters.

```
createPlanes3D(xi, yi, zi, width_i, height_i, ui, vi, wi, opts)
```

Parameters:

- `xi <Array.<number>>`: X coordinates of planes centers.
- `yi <Array.<number>>`: Y coordinates of planes centers.
- `zi <Array.<number>>`: Z coordinates of planes centers.
- `width_i <Array.<number>>`: Width of the rectangle.
- `height_i <Array.<number>>`: Height of the rectangle.
- `ui <Array.<number>>`: X component of the plane's normal vector.

- `vi <Array.<number>>`: Y component of the plane's normal vector.
- `wi <Array.<number>>`: Z component of the plane's normal vector.
- `opts <Object>`: Additional plotting options (color, opacity, etc.).

Returns: `<Object>`: - An object containing line and area trace data for plotting.
Creates a rectangular planes in 3D space, oriented by a normal vector [u, v, w].

```
createLines3D(x1i, y1i, z1i, x2i, y2i, z2i)
```

Parameters:

- `x1i <Array.<number>>`: X1 coordinates of lines.
- `y1i <Array.<number>>`: Y1 coordinates of lines.
- `z1i <Array.<number>>`: Z1 coordinates of lines.
- `x2i <Array.<number>>`: X2 coordinates of lines.
- `y2i <Array.<number>>`: Y2 coordinates of lines.
- `z2i <Array.<number>>`: Z2 coordinates of lines.

Returns: `<Object>`: - lines object.
Creates a lines in 3D space.

```
createPoints3D(xi, yi, zi)
```

Parameters:

- `xi <Array.<number>>`: X coordinates of points.
- `yi <Array.<number>>`: Y coordinates of points.
- `zi <Array.<number>>`: Z coordinates of points.

Returns: `<Object>`: - points object.
Creates a points in 3D space.

```
createText3D(xi, yi, zi)
```

Parameters:

- `xi <Array.<number>>`: X coordinates of points.
- `yi <Array.<number>>`: Y coordinates of points.
- `zi <Array.<number>>`: Z coordinates of points.

Returns: `<Object>`: - points object.
Creates a points in 3D space.

`symRectangle(W, H, [Z])`

Parameters:

- `W <number>`: Width of the rectangle.
- `H <number>`: Height of the rectangle.
- `Z <number>`: Z-coordinate for the rectangle plane.

Returns: `<Array.<number>>`: - Array of vertex coordinates for the rectangle.
Creates a symmetrical rectangle in 3D space.

`circle(radius, segments)`

Parameters:

- `radius <number>`: Radius of the circle.
- `segments <number>`: Number of segments for the circle.

Returns: `<Array.<Array.<number>>>`: - Array of points forming the circle.
Helper method to generate points for a circle in the XY plane.

`disk(radius, segments_a, segments_r)`

Parameters:

- `radius <number>`: Radius of the disk.
- `segments_a <number>`: Number of angular segments for the disk.
- `segments_r <number>`: Number of radial segments for the disk.

Returns: `<Array.<Array.<number>>>`: - Array of points forming the disk.
Helper method to generate points for a disk in the XY plane.

`boundary3D(points, [shrink])`

Parameters:

- `points <Array.<Array.<number>>>`: Array of points defining the shape.
- `shrink <number>`: Factor by which to shrink the boundary.

Returns: `<Array>`: - An array containing boundary facets and the volume.
Generates the boundary of a 3D shape based on points and a shrink factor.

```
writeOff(filename, vertices, faces)
```

Parameters:

- `filename <string>`: The path to the OFF file.
- `vertices <Array.<Array.<number>>>`: Array of vertex coordinates.
- `faces <Array.<Array.<number>>>`: Array of face indices.

Writes geometry data to an OFF file.

```
readOff(filename)
```

Parameters:

- `filename <string>`: The path to the OFF file.

Returns: `<Object>`: - An object containing vertices and faces arrays.
Reads an OFF file and returns the vertices and faces.

10.20 control

```
tf(num, den, [Ts])
```

Parameters:

- `num <Array.<number>>`: Numerator coefficients of the transfer function.
- `den <Array.<number>>`: Denominator coefficients of the transfer function.
- `Ts <number>`: Sampling time, defaults to 0 for continuous-time systems.

Returns: `<object>`: An object representing the transfer function { num, den, Ts }.
Create a transfer function representation.

```
ss(A, B, C, D, [Ts])
```

Parameters:

- `A <Array.<Array.<number>>>`: System matrix.
- `B <Array.<Array.<number>>>`: Input matrix.
- `C <Array.<Array.<number>>>`: Output matrix.

- `D <Array.<Array.<number>>>`: Feedthrough matrix.
- `Ts <number>`: Sampling time, defaults to 0 for continuous-time systems.

Returns: `<object>`: An object representing the state-space system { A, B, C, D, Ts }.
Create a state-space representation.

```
tf2ss(num, den)
```

Parameters:

- `num <Array.<number>>`: Numerator coefficients of the transfer function.
- `den <Array.<number>>`: Denominator coefficients of the transfer function.

Returns: `<object>`: State-space representation of the system.
Convert a transfer function to a state-space representation.

```
ss2tf(sys)
```

Parameters:

- `sys <object>`: State-space system { A, B, C, D }.

Returns: `<object>`: Transfer function representation { num, den }.
Convert a state-space representation to a transfer function.

```
c2d(numc, denc, Ts)
```

Parameters:

- `numc <Array.<number>>`: Continuous-time numerator coefficients.
- `denc <Array.<number>>`: Continuous-time denominator coefficients.
- `Ts <number>`: Sampling time.

Returns: `<object>`: Discrete-time transfer function representation { num, den }.
Convert a continuous-time transfer function to discrete-time.

```
lsim(sys, u, t)
```

Parameters:

- `sys <Array.<number>>`: transfer function of system.
- `u <Array.<number>>`: Input signal array.
- `t <Array.<number>>`: Time vector array.

Returns: `<object>`: An object containing the response: - y: Output signal array. - t: Time vector array (same as input).
Simulate the time response of a discrete-time linear system.

```
step(sys, Tfinal)
```

Parameters:

- `sys <Object>`: The system to simulate.
- `Tfinal <number>`: The final time for the simulation.

Returns: `<Object>`: The simulation result.
Simulates the system response over a specified time period.

```
tfest(t, u, y, np, nz, [method])
```

Parameters:

- `t <Array.<number>>`: Time vector.
- `u <Array.<number>>`: Input signal vector.
- `y <Array.<number>>`: Output signal vector.
- `np <number>`: Number of poles.
- `nz <number>`: Number of zeros.
- `method <string>`: Optimization method ('NelderMead' or 'Powell').

Returns: `<Object>`: Estimated transfer function and mean squared error.
Estimates a transfer function model using numerical optimization.

10.21 optim

```
optimPowell(fnc, x0, [options])
```

Parameters:

- `fnc <function>`: Function to be minimized. Accepts an array of size N and returns a scalar.
- `x0 <Array>`: Initial guess for the parameters as an array of size N.
- `options <Object>`: Optional parameters: - eps: Convergence threshold (default: 1e-6) - alpha: Initial step size scaling factor (default: 0.001) - stepSize: Finite difference step size for gradient estimation (default: 1e-6) - maxIterations: Maximum number of iterations to prevent infinite loops (default: 1000)

Returns: `<Object>`: An object with two fields: - argument: The parameter array that minimizes the function. - fncvalue: The function value at the minimized parameters.
Minimizes an unconstrained function using a coordinate descent-like Powell algorithm.

```
optimNelderMead(f, x0, [parameters], [parameters.maxIterations],
[parameters.nonZeroDelta], [parameters.zeroDelta],
[parameters.minErrorDelta], [parameters.minTolerance], [parameters.rho],
[parameters.chi], [parameters.psi], [parameters.sigma],
[parameters.history])
```

Parameters:

- `f <function>`: The objective function to minimize. It should accept an array of numbers and return a scalar value.
- `x0 <Array.<number>>`: An initial guess for the parameters as an array of numbers.
- `parameters <Object>`: Optional parameters to control the optimization process.
- `parameters.maxIterations <number>`: Maximum number of iterations to perform.
- `parameters.nonZeroDelta <number>`: Scaling factor for non-zero initial steps in the simplex.
- `parameters.zeroDelta <number>`: Initial step size for parameters that are initially zero.
- `parameters.minErrorDelta <number>`: Minimum change in function value to continue iterations.
- `parameters.minTolerance <number>`: Minimum change in parameters to continue iterations.
- `parameters.rho <number>`: Reflection coefficient.
- `parameters.chi <number>`: Expansion coefficient.
- `parameters.psi <number>`: Contraction coefficient.
- `parameters.sigma <number>`: Reduction coefficient.
- `parameters.history <Array.<Object>>`: Optional array to store the history of simplex states for analysis.

Returns: `<Object>`: An object containing: - `fx`: The minimum function value found. - `x`: The parameters corresponding to the minimum function value.
Performs optimization using the Nelder-Mead algorithm.

```
optimConjugateGradient(f, initial, [params], [params.maxIterations],
[params.history])
```

Parameters:

- `f <function>`: The objective function to minimize. It should accept an array of numbers and return a scalar value and its gradient.
- `initial <Array.<number>>`: An initial guess for the parameters as an array of numbers.
- `params <Object>`: Optional parameters to control the optimization process.
- `params.maxIterations <number>`: Maximum number of iterations to perform.
- `params.history <Array.<Object>>`: Optional array to store the history of optimization steps for analysis.

Returns: `<Object>`: An object containing: - `fx`: The minimum function value found. - `x`: The parameters corresponding to the minimum function value. - `fxprime`: The gradient of the function at the minimum.
Performs optimization using the Conjugate Gradient method.

```
optimGradientDescent(f, initial, [params], [params.maxIterations],
[params.learnRate], [params.history])
```

Parameters:

- `f <function>`: The objective function to minimize. It should accept an array of numbers and return a scalar value and its gradient.
- `initial <Array.<number>>`: An initial guess for the parameters as an array of numbers.
- `params <Object>`: Optional parameters to control the optimization process.
- `params.maxIterations <number>`: Maximum number of iterations to perform.
- `params.learnRate <number>`: Learning rate or step size for each iteration.
- `params.history <Array.<Object>>`: Optional array to store the history of optimization steps for analysis.

Returns: `<Object>`: An object containing: - `fx`: The minimum function value found. - `x`: The parameters corresponding to the minimum function value. - `fxprime`: The gradient of the function at the minimum.
Performs optimization using the Gradient Descent method.

```
optimGradientDescentLineSearch(f, initial, [params],
[params.maxIterations], [params.learnRate], [params.c1], [params.c2],
[params.history])
```

Parameters:

- `f <function>`: The objective function to minimize. It should accept an array of numbers and return a scalar value and its gradient.
- `initial <Array.<number>>`: An initial guess for the parameters as an array of numbers.
- `params <Object>`: Optional parameters to control the optimization process.
- `params.maxIterations <number>`: Maximum number of iterations to perform.
- `params.learnRate <number>`: Initial learning rate or step size for the line search.
- `params.c1 <number>`: Parameter for the Armijo condition in Wolfe Line Search.
- `params.c2 <number>`: Parameter for the curvature condition in Wolfe Line Search.
- `params.history <Array.<Object>>`: Optional array to store the history of optimization steps for analysis, including line search details.

Returns: `<Object>`: An object containing: - `fx`: The minimum function value found. - `x`: The parameters corresponding to the minimum function value. - `fxprime`: The gradient of the function at the minimum.
Performs optimization using the Gradient Descent method with Wolfe Line Search.

```
optimBisect(f, a, b, [parameters], [parameters.maxIterations],
[parameters.tolerance])
```

Parameters:

- `f <function>`: The function for which to find a root. It should accept a number and return a number.
- `a <number>`: The start of the interval. Must satisfy f(a) and f(b) have opposite signs.
- `b <number>`: The end of the interval. Must satisfy f(a) and f(b) have opposite signs.
- `parameters <Object>`: Optional parameters to control the root-finding process.
- `parameters.maxIterations <number>`: Maximum number of iterations to perform.
- `parameters.tolerance <number>`: Tolerance for convergence. The method stops when the interval width is below this value.

Returns: `<number>`: The root found within the interval [a, b].
Performs root finding using the Bisection method.

```
fminsearch(f, x0, [parameters], [parameters.maxIterations],
[parameters.nonZeroDelta], [parameters.zeroDelta],
[parameters.minErrorDelta], [parameters.minTolerance], [parameters.rho],
[parameters.chi], [parameters.psi], [parameters.sigma],
[parameters.history])
```

Parameters:

- `f <function>`: The objective function to minimize. It should accept an array of numbers and return a scalar value.
- `x0 <Array.<number>>`: An initial guess for the parameters as an array of numbers.
- `parameters <Object>`: Optional parameters to control the optimization process.
- `parameters.maxIterations <number>`: Maximum number of iterations to perform.
- `parameters.nonZeroDelta <number>`: Scaling factor for non-zero initial steps in the simplex.
- `parameters.zeroDelta <number>`: Initial step size for parameters that are initially zero.
- `parameters.minErrorDelta <number>`: Minimum change in function value to continue iterations.
- `parameters.minTolerance <number>`: Minimum change in parameters to continue iterations.
- `parameters.rho <number>`: Reflection coefficient.
- `parameters.chi <number>`: Expansion coefficient.
- `parameters.psi <number>`: Contraction coefficient.
- `parameters.sigma <number>`: Reduction coefficient.
- `parameters.history <Array.<Object>>`: Optional array to store the history of simplex states for analysis.

Returns: `<Object>`: An object containing: - `fx`: The minimum function value found. - `x`: The parameters corresponding to the minimum function value.
Performs search using the Nelder-Mead algorithm.

```
fminbnd(func, a, b, [tol])
```

Parameters:

- `func <function>`: The function to minimize. Should accept a single number and return a number.
- `a <number>`: The lower bound of the interval.
- `b <number>`: The upper bound of the interval.
- `tol <number>`: The tolerance for convergence (optional).

Returns: <Object>: An object containing: - `fx`: The minimum function value found. - `x`: The x-value where the function attains its minimum within [a, b].
Finds the minimum of a univariate function within a specified interval using a bracketing method.

```
rcmiga(problem, opts)
```

Parameters:

- `problem` <Object>: The optimization problem definition.
- `opts` <Object>: Configuration options for the algorithm.

Creates an instance of PRDC_JSLAB_LIB_OPTIM_RCMIGA.

10.22 presentation

```
openPresentation(file_path, type)
```

Parameters:

- `file_path` <String>: Absolute or relative path to the presentation directory.
- `type` <String>: Type of presentation.

Returns: <Promise.<Window>>: Resolves to the window context of the opened presentation.
Opens an existing presentation in a new window and returns its context.

```
editPresentation(file_path, type)
```

Parameters:

- `file_path` <String>: Absolute or relative path to the presentation directory.
- `type` <String>: Type of presentation.

Returns: <Promise.<Window>>: Resolves to the window context of the editor window.
Opens the presentation editor for the specified project and returns its context.

```
createPresentation(file_path, [opts_in], [open_editor])
```

Parameters:

- `file_path` <String>: Target directory where the presentation project will be created.
- `opts_in` <Object>: Extra options

- `open_editor <Boolean>`: If true, automatically opens the new project in the editor.

Creates a new presentation project on disk and optionally opens it in the editor.

```
packPresentation(file_path)
```

Parameters:

- `file_path <String>`: Path to the presentation directory to be archived.

Packages an existing presentation directory into a ZIP archive beside it.

```
makeStandalonePresentation(file_path)
```

Parameters:

- `file_path <String>`: Path to the presentation directory to be archived.

Converts an existing presentation to standalone presentation.

```
presentationToPdf(file_path, run_make_standalone)
```

Parameters:

- `file_path <String>`: Path to the presentation directory.
- `run_make_standalone <Boolean>`: Whether to run makeStandalonePresentation method or not.

Converts an presentation to PDF format.

10.23 mechanics

```
plotBeamDiagrams(data, [opts_in])
```

Parameters:

- `data <Array>`: Array of objects with x, y, title, xlabel, ylabel
- `opts_in <Object>`: Extra plotting options

Returns: `<Promise.<{extrems: Array.<String>, context: Object}>>`:
Plots beam diagrams.

10.24 gui

`escapeHTML(string)`

Parameters:

- `string` `<string>`: The raw text whose special characters should be converted to HTML entities.

Returns: `<string>`: The sanitized string safe for insertion into the DOM.
Escapes reserved HTML characters in a string to prevent injection or XSS.

`isVisible(el)`

Parameters:

- `el` `<HTMLElement>`: The element to test for visibility in normal document flow.

Returns: `<boolean>`: `true` if the element has a non-null `offsetParent`; otherwise `false`.
Reports whether an element participates in layout flow (i.e., is not `display:none`).

`getElVal(el_or_sel)`

Parameters:

- `el_or_sel` `<string>`: The element itself or a selector/ID string used to locate it.

Returns: `<string>`: The escaped string value, or an empty string if the element is not found.
Reads an input-like element's value, escapes it for HTML, and returns the result.

`getElValNum(el_or_sel)`

Parameters:

- `el_or_sel` `<string>`: The element itself or a selector/ID string used to locate it.

Returns: `<number>`: The numeric value or `NaN` if conversion fails.
Reads an input-like element's value, converts it to a number, and returns the result.

`onSliderInput(el, fun)`

Parameters:

- `el` `<string>`: The slider element or a selector for delegated listening.

- `fun <function>`: The function to invoke each time the slider emits `input` or `change`.

Attaches `input` and `change` listeners to a slider and forwards events to a callback.

```
onInput(el, fun, [validator])
```

Parameters:

- `el <string>`: Target element or CSS selector for delegation.
- `fun <function>`: Callback executed with `this` bound to the element that triggered the event.
- `validator <function>`: Optional predicate that must return `true` for the callback to fire.

Fires a callback when an input loses focus or the user presses Enter, with optional delegation.

```
onInputChange(el, fun, [validator])
```

Parameters:

- `el <string>`: Target element or selector for delegated listening.
- `fun <function>`: Callback executed when the value truly changes.
- `validator <function>`: Optional predicate to approve or reject the change.

Detects actual value changes in an input and invokes a callback after validation.

```
onActiveInputChange(el, fun, [validator])
```

Parameters:

- `el <string>`: Target element or selector for delegated listening.
- `fun <function>`: Callback executed when the value changes during active editing.
- `validator <function>`: Optional predicate to approve or reject the change.

Like `onInputChange` but triggers only while the element remains focused (active input).

```
setInputValue(el_or_sel, val)
```

Parameters:

- `el_or_sel <string>`: The input element or selector identifying it.

- `val <string>`: The value to assign and remember as the "set" value.

Programmatically sets an input's value and records it as the baseline for change tracking.

```
updateInputValue(el_or_sel, val)
```

Parameters:

- `el_or_sel <string>`: The input element or selector identifying it.
- `val <string>`: The new value to write and store as the baseline.

Updates an input's value only if it is different from the current content, and records it.

```
resetInputValue(el_or_sel)
```

Parameters:

- `el_or_sel <string>`: The input element or selector identifying it.

Restores an input to the value previously stored with `setInputValue`.

```
showInputChanged(el_or_sel)
```

Parameters:

- `el_or_sel <string>`: The input element or selector identifying it.

Adds or removes the `changed` CSS class based on whether the current value differs from the stored baseline.

```
validateInputNumber(el_or_sel)
```

Parameters:

- `el_or_sel <string>`: The input element or selector identifying it.

Returns: `<Array>`: Tuple of the parsed number and a validity flag.
Validates that an input contains a numeric value and briefly highlights errors.

```
setInputWithWarning(el_or_sel, val)
```

Parameters:

- `el_or_sel <string>`: The input element or selector identifying it.
- `val <string>`: The value to assign before flashing the warning.

Sets an input's value and temporarily applies a `warning` class for visual feedback.

`addClassMs(node, cls, ms)`

Parameters:

- `node <HTMLElement>`: The element to which the class will be applied.
- `cls <string>`: The CSS class name to toggle.
- `ms <number>`: The number of milliseconds to keep the class before removal.

Adds a CSS class to an element for a specified duration, then removes it.

`onResize(el_or_sel, cb)`

Parameters:

- `el_or_sel <string>`: The element or selector to observe.
- `cb <function>`: Callback that receives the `ResizeObserverEntry.contentRect` whenever the element resizes.

Returns: `<ResizeObserver>`: The observer instance, or `undefined` if the element is not found.
Observes an element's size changes and reports each new `contentRect` to a callback.

10.25 Matrix

`column(index)`

Parameters:

- `index <number>`: The index of the column to extract.

Returns: `<Array>`: The extracted column as an array.
Extracts a specific column from a matrix.

`row(index)`

Parameters:

- `index <number>`: The index of the row to extract.

Returns: `<Array>`: The extracted row as an array.
Extracts a specific row from a matrix.

`length(dim)`

Parameters:

- `dim <number>`: The dimension (0 for rows, 1 for columns).

Returns: `<number>`: The length along the specified dimension.
Gets the length of the matrix along a specified dimension.

`numel()`

Returns: `<number>`: The number of elements.
Gets the number of elements in the matrix.

`size(dim)`

Parameters:

- `dim <number>`: The dimension (0 for rows, 1 for columns).

Returns: `<number>`: The size along the specified dimension.
Gets the size of the matrix along a specified dimension.

`reshape(rows, cols)`

Parameters:

- `rows <number>`: New number of rows.
- `cols <number>`: New number of columns.

Returns: `<PRDC_JSLAB_MATRIX>`: The reshaped matrix.
Reshapes the matrix to the specified dimensions.

`repmat(rowReps, colReps)`

Parameters:

- `rowReps <number>`: Number of row repetitions.
- `colReps <number>`: Number of column repetitions.

Returns: `<PRDC_JSLAB_MATRIX>`: The replicated matrix.
Replicates the matrix a specified number of times.

`transpose()`

Returns: `<PRDC_JSLAB_MATRIX>`: The transposed matrix.
Transposes the matrix.

`inv()`

Returns: `<PRDC_JSLAB_MATRIX>`: The inverse matrix.
Computes the inverse of the matrix.

`det()`

Returns: `<number>`: The determinant.
Computes the determinant of the matrix.

`trace()`

Returns: `<number>`: The trace of the matrix.
Computes the trace of the matrix (sum of diagonal elements).

`norm()`

Returns: `<number>`: The Frobenius norm.
Computes the Frobenius norm of the matrix.

`powm(p)`

Parameters:

- `p` `<number>`: The exponent.

Returns: `<PRDC_JSLAB_MATRIX>`: The resulting matrix.
Raises the matrix to a power.

`expm()`

Returns: `<PRDC_JSLAB_MATRIX>`: The exponential matrix.
Computes the matrix exponential.

`add(A)`

Parameters:

- `A <PRDC_JSLAB_MATRIX>`: The matrix to add.

Returns: `<PRDC_JSLAB_MATRIX>`: The resulting matrix.
Adds two matrices.

`plus(A)`

Parameters:

- `A <PRDC_JSLAB_MATRIX>`: The matrix to add.

Returns: `<PRDC_JSLAB_MATRIX>`: The resulting matrix.
Adds two matrices (alias for add).

`subtract(A)`

Parameters:

- `A <PRDC_JSLAB_MATRIX>`: The matrix to subtract.

Returns: `<PRDC_JSLAB_MATRIX>`: The resulting matrix.
Subtracts matrix A from the current matrix.

`minus(A)`

Parameters:

- `A <PRDC_JSLAB_MATRIX>`: The matrix to subtract.

Returns: `<PRDC_JSLAB_MATRIX>`: The resulting matrix.
Subtracts matrix A from the current matrix (alias for subtract).

`multiply(A)`

Parameters:

- `A <PRDC_JSLAB_MATRIX>`: The matrix to multiply with.

Returns: `<PRDC_JSLAB_MATRIX>`: The resulting matrix.
Multiplies two matrices.

`linsolve(B)`

Parameters:

- B <PRDC_JSLAB_MATRIX>: The right-hand side matrix.

Returns: <PRDC_JSLAB_MATRIX>: The solution matrix.
Solves a linear system.

`divideEl(A)`

Parameters:

- A <PRDC_JSLAB_MATRIX>: The matrix or scalar to divide by.

Returns: <PRDC_JSLAB_MATRIX>: The resulting matrix.
Divides each element by another matrix or scalar.

`multiplyEl(A)`

Parameters:

- A <PRDC_JSLAB_MATRIX>: The matrix or scalar to multiply by.

Returns: <PRDC_JSLAB_MATRIX>: The resulting matrix.
Multiplies each element by another matrix or scalar.

`powEl(p)`

Parameters:

- p <number>: The exponent.

Returns: <PRDC_JSLAB_MATRIX>: The resulting matrix.
Raises each element to a power.

`elementWise(func)`

Parameters:

- func <function>: The function to apply.

Returns: <PRDC_JSLAB_MATRIX>: The resulting matrix.
Applies a function to each element of the matrix.

`reciprocal()`

Returns: <PRDC_JSLAB_MATRIX>: The matrix with reciprocals.
Computes the reciprocal of each element in the matrix.

`sum()`

Returns: `<number>`: The sum of all elements.
Computes the sum of all elements in the matrix.

`sort([order])`

Parameters:

- `order <string>`: The order of sorting ('asc' or 'desc').

Returns: `<PRDC_JSLAB_MATRIX>`: The sorted matrix.
Sorts the elements of the matrix.

`min()`

Returns: `<number>`: The minimum value.
Finds the minimum element in the matrix.

`max()`

Returns: `<number>`: The maximum value.
Finds the maximum element in the matrix.

`clone()`

Returns: `<PRDC_JSLAB_MATRIX>`: A cloned matrix instance.
Creates a clone of the current matrix.

`index(args)`

Parameters:

- `args <number>`: Row and column indices.

Returns: `<Array>`: The selected elements.
Retrieves elements based on row and column indices.

`setSub(args)`

Parameters:

- `args <*>`: Indices and values to set.

Sets a subset of the matrix elements.

`getSub(args)`

Parameters:

- `args <*>`: Indices to retrieve.

Returns: `<PRDC_JSLAB_MATRIX>`: The subset matrix.
Gets a subset of the matrix elements.

`toArray()`

Returns: `<Array>`: The matrix data as a two-dimensional array.
Converts the matrix to a two-dimensional array.

`toFlatArray()`

Returns: `<Array>`: The matrix data as a one-dimensional array.
Converts the matrix to a one-dimensional array.

`toString()`

Returns: `<string>`: The string representation of the matrix.
Returns a string representation of the matrix.

`toJSON()`

Returns: `<Array>`: The matrix data as a two-dimensional array.
Returns a JSON representation of the matrix.

`toSafeJSON()`

Returns: `<Array>`: The matrix data as a two-dimensional array.
Returns a safe JSON representation of the matrix.

`toPrettyString()`

Returns: `<string>`: The pretty string representation of the matrix.
Returns a pretty string representation of the matrix.

10.26 Vector

`length()`

Returns: `<number>`: The length of the vector.
Calculates the length (magnitude) of the vector.

`norm()`

Returns: `<number>`: The length of the vector.
Calculates the length (magnitude) of the vector.

`add(v)`

Parameters:

- `v <PRDC_JSLAB_VECTOR>`: The second vector.

Returns: `<PRDC_JSLAB_VECTOR>`: The resulting vector after addition.
Adds two vectors and returns the result.

`plus(v)`

Parameters:

- `v <PRDC_JSLAB_VECTOR>`: The second vector.

Returns: `<PRDC_JSLAB_VECTOR>`: The resulting vector after addition.
Adds two vectors and returns the result.

`subtract(v)`

Parameters:

- `v <PRDC_JSLAB_VECTOR>`: The vector to subtract.

Returns: `<PRDC_JSLAB_VECTOR>`: The resulting vector after subtraction.
Subtracts the second vector from the first and returns the result.

`minus(v)`

Parameters:

- `v <PRDC_JSLAB_VECTOR>`: The vector to subtract.

Returns: `<PRDC_JSLAB_VECTOR>`: The resulting vector after subtraction.
Subtracts the second vector from the first and returns the result.

`scale(scale_x, [scale_y], [scale_z])`

Parameters:

- `scale_x <number>`: The scale factor for the x-component or an object with x, y, z properties.
- `scale_y <number>`: The scale factor for the y-component.
- `scale_z <number>`: The scale factor for the z-component.

Returns: `<PRDC_JSLAB_VECTOR>`: The scaled vector.
Scales a vector by the given factors.

`multiply(s)`

Parameters:

- `s <number>`: The scale factor.

Returns: `<PRDC_JSLAB_VECTOR>`: The scaled vector.
Scales a vector by the given factor.

`divide(s)`

Parameters:

- `s <number>`: The scale factor.

Returns: `<PRDC_JSLAB_VECTOR>`: The scaled vector.
Scales a vector by dividing each element by given factor.

`equals(v)`

Parameters:

- `v <PRDC_JSLAB_VECTOR>`: The second vector.

Returns: `<boolean>`: True if vectors are equal, false otherwise.
Checks if two vectors are equal.

`angleTo(v)`

Parameters:

- `v <PRDC_JSLAB_VECTOR>`: The vector.

Returns: `<number>`: The angle in degrees.
Calculates the angle of a vector.

`projectTo(v)`

Parameters:

- `v <PRDC_JSLAB_VECTOR>`: The vector.

Returns: `<PRDC_JSLAB_VECTOR>`: The projected vector.
Projects vector to given vector.

`angles(v)`

Parameters:

- `v <PRDC_JSLAB_VECTOR>`: The vector.

Returns: `<Array>`: The angles azimuth and elevation in degrees.
Calculates the angles of a vector.

`distance(v)`

Parameters:

- `v <PRDC_JSLAB_VECTOR>`: The second vector.

Returns: `<number>`: The distance between the two vectors.
Calculates the distance between two vectors.

`dot(v)`

Parameters:

- `v <PRDC_JSLAB_VECTOR>`: The second vector.

Returns: `<number>`: The dot product.
Calculates the dot product of two vectors.

`cross(v)`

Parameters:

- `v <PRDC_JSLAB_VECTOR>`: The second vector.

Returns: `<PRDC_JSLAB_VECTOR>`: The cross product vector.
Calculates the cross product of two vectors.

`interpolate(v, f)`

Parameters:

- `v <PRDC_JSLAB_VECTOR>`: The ending vector.

- `f <number>`: The interpolation factor.

Returns: `<PRDC_JSLAB_VECTOR>`: The interpolated vector.
Interpolates between two vectors by a factor.

`offset(x, [y], [z])`

Parameters:

- `x <number>`: The amount to offset in the x-direction or an object with x, y, z properties.
- `y <number>`: The amount to offset in the y-direction.
- `z <number>`: The amount to offset in the z-direction.

Returns: `<PRDC_JSLAB_VECTOR>`: The updated vector instance.
Offsets the vector by the given amounts.

`normalize()`

Returns: `<PRDC_JSLAB_VECTOR>`: The normalized vector.
Normalizes the vector to have a length of 1.

`negate()`

Returns: `<PRDC_JSLAB_VECTOR>`: The negated vector.
Negates the vector components.

`clone()`

Returns: `<PRDC_JSLAB_VECTOR>`: A new vector instance with the same components.
Creates a clone of this vector.

`toArray()`

Returns: `<Array.<number>>`: An array containing the x, y, z components of the vector.
Converts the vector to an array.

`toMatrix()`

Returns: `<Object>`: A matrix representation of the vector.
Converts the vector to a matrix.

`toColMatrix()`

Returns: `<Object>`: A column matrix representation of the vector.
Converts the vector to a column matrix.

`toRowMatrix()`

Returns: `<Object>`: A row matrix representation of the vector.
Converts the vector to a row matrix.

`toString()`

Returns: `<string>`: The string representation in the format 'Vector(x:, y:, z:)'.
Returns a string representation of the vector.

`toJSON()`

Returns: `<string>`: The string representation in the format 'Vector(x:, y:, z:)'.
Returns a string representation of the vector.

`toSafeJSON()`

Returns: `<string>`: The string representation in the format 'Vector(x:, y:, z:)'.
Returns a string representation of the vector.

`toPrettyString()`

Returns: `<string>`: The pretty string representation of the object.
Converts the object to a pretty string representation.

10.27 Symbolic

`setValue(value)`

Parameters:

- `value` `<*>`: The new value to assign to the symbolic variable.

Sets the value of the symbolic variable.

`toString()`

Returns: `<string>`: The string representation in the format 'Vector(x:, y:, z:)'.
Returns a string representation of the vector.

`toNumeric()`

Returns: `<*>`: The numeric representation of the symbolic value.
Converts the symbolic value to a numeric value.

`toJSON()`

Returns: `<string>`: The JSON string representation of the symbolic value.
Converts the symbolic value to a JSON string.

`toSafeJSON()`

Returns: `<Object>`: The safe JSON representation of the object.
Converts the object to a safe JSON representation.

`toPrettyString()`

Returns: `<string>`: The pretty string representation of the object.
Converts the object to a pretty string representation.

10.28 Window

`open(file)`

Parameters:

- `file <string>`: The path to the HTML file or url to open in the window.

Returns: `<Promise.<void>>`: A promise that resolves when the window is opened and ready.
Opens the window with the specified file or url.

`show()`

Returns: `<Promise.<(boolean|undefined)>>`: - Resolves to `true` if the window was shown successfully, or `false` if the window ID is invalid.
Shows the window.

`hide()`

Returns: `<Promise.<(boolean|undefined)>>`: - Resolves to `true` if the window was hidden successfully, or `false` if the window ID is invalid.
Hides the window.

`focus()`

Returns: `<Promise>`: - Resolves when the window size is focused.
Brings focus to the window.

`minimize()`

Returns: `<Promise.<(boolean|undefined)>>`: - Resolves to `true` if the window was minimized successfully, or `false` if the window ID is invalid.
Minimizes the window.

`center()`

Returns: `<Promise.<(boolean|undefined)>>`: - Resolves to `true` if the window was centered successfully, or `false` if the window ID is invalid.
Centers the window on the screen.

`moveTop()`

Returns: `<Promise.<(boolean|undefined)>>`: - Resolves to `true` if the window was moved to the top successfully, or `false` if the window ID is invalid.
Moves the window to the top of the window stack.

`setSize(width, height)`

Parameters:

- `width <number>`: The desired width of the window.
- `height <number>`: The desired height of the window.

Returns: `<Promise>`: - Resolves when the window size is set.
Sets the size of the current window.

`setPos(left, top)`

Parameters:

- `left <number>`: The desired left position of the window.
- `top <number>`: The desired top position of the window.

Returns: `<Promise>`: - Resolves when the window position is set.
Sets the position of the current window.

`setResizable(state)`

Parameters:

- `state` `<boolean>`: Whether the window should be resizable (`true`) or not (`false`).

Returns: `<Promise.<(boolean|undefined)>>`: - Resolves to `true` if the resizable state was set successfully, or `false` if the window ID is invalid.
Sets the resizable state of the window.

`setMovable(state)`

Parameters:

- `state` `<boolean>`: Whether the window should be movable (`true`) or not (`false`).

Returns: `<Promise.<(boolean|undefined)>>`: - Resolves to `true` if the movable state was set successfully, or `false` if the window ID is invalid.
Sets the movable state of the window.

`setAspectRatio(aspect_ratio)`

Parameters:

- `aspect_ratio` `<number>`: The desired aspect ratio (width divided by height) for the window.

Returns: `<Promise.<(boolean|undefined)>>`: - Resolves to `true` if the aspect ratio was set successfully, or `false` if the window ID is invalid.
Sets the aspect ratio of the window.

`setOpacity(opacity)`

Parameters:

- `opacity` `<number>`: The desired opacity level of the window (ranging from `0` for fully transparent to `1` for fully opaque).

Returns: `<Promise.<(boolean|undefined)>>`: - Resolves to `true` if the opacity was set successfully, or `false` if the window ID is invalid.
Sets the opacity of the window.

`setFullscreen(state)`

Parameters:

- `state <number>`: State of fullscreen

Returns: `<Promise.<(boolean|undefined)>>`: - Resolves to `true` if the fullscreen was set successfully, or `false` if the window ID is invalid.
Sets the fullscreen state of the window.

`setTitle(title)`

Parameters:

- `title <string>`: The new title for the window.

Returns: `<Promise.<*>>`: A promise that resolves when the title is set.
Sets the title of the current window.

`printToPdf(options)`

Parameters:

- `options <Object>`: Options for printing.

Returns: `<Buffer>`: Generated PDF data.
Prints the current window to PDF.

`getSize()`

Returns: `<Promise.<Array>>`: - Resolves with an array [width, height].
Retrieves the size of the current window.

`getPos()`

Returns: `<Promise.<Array>>`: - Resolves with an array [left, top].
Retrieves the position of the current window.

`close()`

Returns: `<Promise>`: - Resolves when the window is closed.
Closes the current window.

`openDevTools()`

Returns: `<Promise.<boolean>>`: A promise that resolves to true when dev tools are opened.
Opens the developer tools for the current window asynchronously.

`getMediaSourceId()`

Returns: `<String>`: Media source id if there is window; otherwise, false.
Returns media source id for this window.

`startVideoRecording(opts)`

Parameters:

- `opts <Object>`: Optional settings.

Returns: `<Object>`: - Returns recorder object.
Starts video recording of this window.

`addScript(script_path)`

Parameters:

- `script_path <string>`: The script's URL.

Appends a script to the document head.

`addLinkStylesheet(stylesheet_path)`

Parameters:

- `stylesheet_path <string>`: The stylesheet's URL.

Appends a stylesheet link to the document head.

`addUI()`

Loads the UI script.

10.29 Figure

`init()`

Initializes figure.

`focus()`

Brings the figure window to the foreground.

`setSize(width, height)`

Parameters:

- `width <number>`: The desired width of the window.
- `height <number>`: The desired height of the window.

Returns: `<Promise>`: - Resolves when the window size is set.
Sets the size of the window.

`setPos(left, top)`

Parameters:

- `left <number>`: The desired left position of the window.
- `top <number>`: The desired top position of the window.

Returns: `<Promise>`: - Resolves when the window position is set.
Sets the position of the window.

`setTitle(title)`

Parameters:

- `title <string>`: The new title for the window.

Returns: `<Promise.<*>>`: A promise that resolves when the title is set.
Sets the title of the current window.

`getSize()`

Returns: `<Promise.<Array>>`: - Resolves with an array [width, height].
Retrieves the size of the window.

`getPos()`

Returns: `<Promise.<Array>>`: - Resolves with an array [left, top].
Retrieves the position of the window.

`getMediaSourceId()`

Returns: `<String>`: - Media source id.
Retrieves the media source id of the figure.

`startVideoRecording(opts)`

Parameters:

- `opts <Object>`: Optional settings.

Returns: `<Object>`: - Returns recorder object.
Starts video recording of the figure.

`close()`

Returns: `<Promise>`: - Resolves when the window is closed.
Closes the window.

`hideMenu()`

Hides figure menu.

`showMenu()`

Shows figure menu.

10.30 Plot

`legend(label)`

Parameters:

- `label <String>`: Label text for the x-axis.

Sets the label for the x-axis of the plot.

`xlabel(label)`

Parameters:

- `label <String>`: Label text for the x-axis.

Sets the label for the x-axis of the plot.

`ylabel(label)`

Parameters:

- `label <String>`: Label text for the y-axis.

Sets the label for the y-axis of the plot.

```
zlabel(label)
```

Parameters:

- `label <String>`: Label text for the z-axis.

Sets the label for the z-axis of the plot.

```
title(label)
```

Parameters:

- `label <String>`: Title text for the plot.

Sets the title of the plot.

```
xlim(lim)
```

Parameters:

- `lim <String>`: Limits for the x-axis.

Sets the limits for the x-axis of the plot.

```
ylim(lim)
```

Parameters:

- `lim <String>`: Limits for the y-axis.

Sets the limits for the y-axis of the plot.

```
zlim(lim)
```

Parameters:

- `lim <String>`: Limits for the z-axis.

Sets the limits for the z-axis of the plot.

```
view(azimuth, elevation)
```

Parameters:

- `azimuth <number>`: The azimuth angle.
- `elevation <number>`: The elevation angle.

Adjusts the view based on azimuth and elevation angles.

`zoom(factor)`

Parameters:

- `factor <number>`: The zoom factor.

Adjusts the zoom based on factor.

`axis(style)`

Parameters:

- `style <Object>`: The style configuration to set for the axis.

Sets the axis style value and updates the plot layout if the plot is ready.

`fromJSON(data)`

Parameters:

- `data <Array>`: Data for the plot.

Sets the plot data from JSON.

`print(filename, options)`

Parameters:

- `filename <String>`: The filename for the print job.
- `options <Object>`: Options for the print job.

Adds a print job to the queue and prints it if the system is ready.

`update(traces, N)`

Parameters:

- `traces <Object>`: The trace data to be updated in the plot.
- `N <number>`: The data length or index for updating the plot.

Updates plot data by delegating to the `updateData` method.

`updateById(data)`

Parameters:

- `data <Object>`: Trace update object(s) addressed by `id`.

Updates plot data by delegating to the `updateDataById` method.

`remove()`

Removes the plot from the figure, cleaning up any resources associated with it.

10.31 freecad_link

`start(exe, options)`

Parameters:

- `exe <string>`: The executable path of FreeCAD.
- `options <Object>`: Configuration options such as port and host.

Starts the FreeCAD application and establishes a TCP connection for remote procedure calls. Attempts to start FreeCAD if it's not running and connects to its TCP server.

`findServer()`

Attempts to locate the FreeCAD TCP server within the network, respecting the startup timeout. Checks if the TCP server is reachable by sending a 'PING' command.

`send(message, timeout)`

Parameters:

- `message <string>`: The message to send.
- `timeout <number>`: Timeout in milliseconds to wait for a response.

Returns: `<Buffer>`: - The response from the server or false if the request times out.
Sends a message to the FreeCAD TCP server and waits for a response. Manages the TCP communication by ensuring message integrity and handling timeouts.

`inputPraser(message)`

Parameters:

- `message <string>`: The message received from FreeCAD.

Returns: `<Array>`: - An array of parsed message components.
Parses the input from FreeCAD responses to identify errors and data. Splits the message by '—' and checks for error or data messages.

```
showMessage(message)
```

Parameters:

- `message <string>`: The message received from FreeCAD.

Displays a message from FreeCAD in the JSLAB interface. Parses and displays messages specifically tagged as 'MSG' from FreeCAD.

```
quit()
```

Closes the FreeCAD application gracefully. Sends a quit command and handles the termination of the TCP connection.

```
open(filePath, timeout)
```

Parameters:

- `filePath <string>`: The path to the file to be opened.
- `timeout <number>`: Timeout in milliseconds to wait for a response.

Opens a specified file in FreeCAD. Sends a command to open a file and handles responses to confirm file access.

```
importFile(filePath, timeout)
```

Parameters:

- `filePath <string>`: The path of the file to be imported.
- `timeout <number>`: Timeout in milliseconds to wait for a response.

Imports a file into the current FreeCAD document. Sends an import command and handles responses to confirm the import operation.

```
newDocument(filename, timeout)
```

Parameters:

- `filename <string>`: Optional filename for the new document.

- `timeout <number>`: Timeout in milliseconds to wait for a response.

Creates a new document in FreeCAD, optionally specifying a filename. Sends a command to create a new document and handles the document creation response.

```
save(timeout)
```

Parameters:

- `timeout <number>`: Timeout in milliseconds to wait for a response.

Saves the current document in FreeCAD. Sends a save command and handles responses to confirm the save operation.

```
saveAs(filePath, timeout)
```

Parameters:

- `filePath <string>`: The new file path for the document.
- `timeout <number>`: Timeout in milliseconds to wait for a response.

Saves the current document in FreeCAD under a new filename. Sends a save as command and handles responses to confirm the operation.

```
close(timeout)
```

Parameters:

- `timeout <number>`: Timeout in milliseconds to wait for a response.

Closes the current document in FreeCAD. Sends a close command and handles responses to confirm the document closure.

```
evaluate(command, timeout)
```

Parameters:

- `command <string>`: The command to be evaluated in FreeCAD.
- `timeout <number>`: Timeout in milliseconds to wait for a response.

Executes a command in FreeCAD and returns the evaluation result. Sends an evaluate command with the specified command string.

```
callScript(script, param, timeout)
```

Parameters:

- `script <string>`: The script to run.
- `param <string>`: Parameters to pass to the script.
- `timeout <number>`: Timeout in milliseconds to wait for a response.

Runs a script in FreeCAD with optional parameters. Sends a script command along with parameters and handles the script execution response.

`getArea(timeout)`

Parameters:

- `timeout <number>`: Timeout in milliseconds to wait for a response.

Retrieves the area of the selected object in FreeCAD. Sends a measure area command and parses the response to extract the area value.

`getVolume(timeout)`

Parameters:

- `timeout <number>`: Timeout in milliseconds to wait for a response.

Retrieves the volume of the selected object in FreeCAD. Sends a measure volume command and parses the response to extract the volume value.

10.32 om_link

`start(exe)`

Parameters:

- `exe <string>`: Path to the executable to be run, defaults to the OpenModelica compiler if not provided.

Starts the interaction with an external executable by initializing the necessary environment and parameters. Launches the executable with the appropriate command-line arguments for interaction via ZMQ. Waits for a port file to be created to establish the ZMQ communication.

`sendExpression(expr)`

Parameters:

- `expr <string>`: The expression to be evaluated.

Returns: `<Promise.<any>>`: - A promise that resolves with the parsed result of the expression evaluation.
Sends an expression to be evaluated by the external executable through the ZMQ connection and waits for the result. Parses the response using a dedicated expression parser.

```
ModelicaSystem(filename, modelname, [libraries], [command_line_options])
```

Parameters:

- `filename` `<string>`: The path to the Modelica file.
- `modelname` `<string>`: The name of the Modelica model.
- `libraries` `<Array.<string>>`: An array of library paths to load.
- `command_line_options` `<string>`: Additional command-line options for the simulation.

Initializes and configures a Modelica system with the specified parameters and libraries. Loads necessary files and prepares the environment for simulation.

```
BuildModelicaModel()
```

Builds the Modelica model by sending the appropriate build command and parsing the resulting XML file.

```
getWorkDirectory()
```

Returns: `<string>`: - The path to the working directory.
Retrieves the working directory used for temporary files and simulations.

```
xmlparse()
```

Parses the XML file generated by the Modelica compiler to extract simulation parameters and variables.

```
processVariable(scalar)
```

Parameters:

- `scalar` `<Object>`: The scalar variable to process.

Processes a scalar variable from the XML file and categorizes it based on its properties.

```
getQuantities([args])
```

Parameters:

- `args <Array.<string>>`: An array of quantity names to retrieve. If omitted, returns all quantities.

Returns: `<Array.<Object>>`: - An array of quantity objects.
Retrieves a list of quantities based on the provided arguments.

`getLinearQuantities([args])`

Parameters:

- `args <Array.<string>>`: An array of linear quantity names to retrieve. If omitted, returns all linear quantities.

Returns: `<Array.<Object>>`: - An array of linear quantity objects.
Retrieves a list of linearized quantities based on the provided arguments.

`getParameters([args])`

Parameters:

- `args <string>`: A single parameter name or an array of parameter names to retrieve. If omitted, returns all parameters.

Returns: `<Object>`: - An object containing the requested parameters or a single parameter value.
Retrieves simulation parameters based on the provided arguments.

`getInputs([args])`

Parameters:

- `args <string>`: A single input name or an array of input names to retrieve. If omitted, returns all inputs.

Returns: `<Object>`: - An object containing the requested inputs or a single input value.
Retrieves input variables based on the provided arguments.

`getOutputs([args])`

Parameters:

- `args <string>`: A single output name or an array of output names to retrieve. If omitted, returns all outputs.

Returns: `<Object>`: - An object containing the requested outputs or a single output value.
Retrieves output variables based on the provided arguments.

`getContinuous([args])`

Parameters:

- `args <string>`: A single continuous variable name or an array of names to retrieve. If omitted, returns all continuous variables.

Returns: `<Object>`: - An object containing the requested continuous variables or a single value.
Retrieves continuous variables based on the provided arguments.

`getSimulationOptions([args])`

Parameters:

- `args <string>`: A single simulation option name or an array of names to retrieve. If omitted, returns all simulation options.

Returns: `<Object>`: - An object containing the requested simulation options or a single option value.
Retrieves simulation options based on the provided arguments.

`getLinearizationOptions([args])`

Parameters:

- `args <string>`: A single linearization option name or an array of names to retrieve. If omitted, returns all linearization options.

Returns: `<Object>`: - An object containing the requested linearization options or a single option value.
Retrieves linearization options based on the provided arguments.

`setParameters(args)`

Parameters:

- `args <string>`: A single parameter assignment (e.g., "param=5") or an array of assignments.

Sets simulation parameters based on the provided arguments.

`setSimulationOptions(args)`

Parameters:

- `args <string>`: A single simulation option assignment (e.g., "stepSize=0.01") or an array of assignments.

Sets simulation options based on the provided arguments.

`setLinearizationOptions(args)`

Parameters:

- `args <string>`: A single linearization option assignment or an array of assignments.

Sets linearization options based on the provided arguments.

`setInputs(args)`

Parameters:

- `args <string>`: A single input assignment (e.g., "input1=10") or an array of assignments.

Sets input variables based on the provided arguments.

`createcsvData()`

Creates a CSV file containing input data for the simulation.

`simulate([resultfile], [sim_flags])`

Parameters:

- `resultfile <string>`: The name of the result file to generate.
- `sim_flags <string>`: Additional simulation flags.

Runs the simulation with optional result file and simulation flags.

`linearize()`

Returns: `<Array.<Object>>`: - An array containing the A, B, C, and D matrices.
Performs linearization of the model and retrieves the linear matrices.

`getLinearMatrix()`

Returns: `<Array.<Object>>`: - An array containing the A, B, C, and D matrices.
Retrieves the linear A, B, C, and D matrices.

`getLinearMatrixValues(matrix_name)`

Parameters:

- `matrix_name <Object>`: The linear matrix object to convert.

Returns: `<Array.<Array.<number>>>`: - The converted matrix as a 2D array or 0 if empty.
Converts linear matrix data into a two-dimensional array format.

`getlinear_inputs()`

Returns: `<string>`: - The linear input variables or false if the model is not linearized.
Retrieves the linear input variables.

`getlinear_outputs()`

Returns: `<string>`: - The linear output variables or false if the model is not linearized.
Retrieves the linear output variables.

`getlinear_states()`

Returns: `<string>`: - The linear state variables or false if the model is not linearized.
Retrieves the linear state variables.

`getSolutions([args], [resultfile])`

Parameters:

- `args <string>`: A single variable name or an array of names to retrieve solutions for. If omitted, retrieves all variables.
- `resultfile <string>`: The path to the result file.

Returns: `<Promise.<any>>`: - A promise that resolves with the simulation results or an error message.
Retrieves simulation solutions based on the provided arguments and result file.

`createValidNames(name, value, structname)`

Parameters:

- `name <string>`: The original variable name.
- `value <any>`: The value of the variable.
- `structname <string>`: The structure name (e.g., 'continuous', 'parameter').

Creates valid variable names by replacing invalid characters and categorizes them based on the structure name.

`parseExpression(args)`

Parameters:

- `args <string>`: The expression string to parse.

Returns: `<Array>`: - The parsed data which could be an array, an object, or a string. Parses a given expression string into structured data based on predefined formats. Handles various formats including single and nested lists, records, and single elements.

`close()`

Closes the current session safely by cleaning up resources such as temporary files and network connections. Terminates any active processes and removes temporary port files.

10.33 tcp_client

`setOnData(callback)`

Parameters:

- `callback <function>`: The function to be called when data is received.

Sets the callback function to handle incoming data events.

`setOnError(callback)`

Parameters:

- `callback <function>`: The function to be called when an error occurs.

Sets the callback function to handle error events.

`setKeepAlive([enable], [initialDelay])`

Parameters:

- `enable <boolean>`: Whether to enable keep-alive.
- `initialDelay <number>`: Delay in milliseconds before the first keep-alive probe is sent.

Enables or disables keep-alive functionality on the underlying socket.

`setNoDelay([noDelay])`

Parameters:

- `noDelay <boolean>`: Whether to disable the Nagle algorithm.

Disables the Nagle algorithm, allowing data to be sent immediately.

`setTimeout(timeout, [callback])`

Parameters:

- `timeout <number>`: Timeout in milliseconds.
- `callback <function>`: Optional callback triggered on timeout.

Sets the socket timeout for inactivity.

`read([N])`

Parameters:

- `N <number>`: The number of bytes to read. Reads all available bytes if not specified.

Returns: `<Buffer>`: The data read from the buffer.
Reads a specified number of bytes from the buffer.

`write(data)`

Parameters:

- `data <Buffer>`: The data to send over the TCP connection.

Writes data to the TCP connection if the client is active.

`availableBytes()`

Returns: `<number>`: The number of available bytes.
Returns the number of bytes available in the buffer.

`close()`

Closes the TCP connection and cleans up resources.

10.34 tcp_server

`setOnData(callback)`

Parameters:

- `callback <function>`: Function called when data is received.

Sets the callback function to handle incoming data events.

`setOnError(callback)`

Parameters:

- `callback <function>`: Function called when an error occurs.

Sets the callback function to handle error events.

`setOnDisconnect(callback)`

Parameters:

- `callback <function>`: Function called when a socket disconnects.

Sets the callback function to handle disconnection events.

`write(socket, data)`

Parameters:

- `socket <Object>`: The socket to write data to.
- `data <Buffer>`: The data to send over the TCP connection.

Writes data to a specific TCP connection.

`close()`

Closes the TCP server and all active connections.

10.35 udp_client

`write(data)`

Parameters:

- `data <Buffer>`: The data to send over the UDP connection.

Sends data over the UDP connection if the client is active.

`close()`

Closes the UDP connection and cleans up resources.

10.36 udp_server

`setOnData(callback)`

Parameters:

- `callback <function>`: The function to be called when data is received.

Sets the callback function to handle incoming data events.

`read([N])`

Parameters:

- `N <number>`: The maximum number of bytes to read. Reads all available bytes by default.

Returns: `<Array>`: An array containing the first N bytes of buffered data.
Reads a specified number of bytes from the buffer.

`availableBytes()`

Returns: `<number>`: The number of bytes currently stored in the buffer.
Returns the number of bytes available in the buffer.

`close()`

Closes the UDP server and releases any resources.

10.37 video_call

`setOnMessage(callback)`

Parameters:

- `callback <function>`: The function to call when a message is received.

Sets a callback function to handle incoming messages.

`sendMessage(data)`

Parameters:

- `data <any>`: The data to send.

Sends a message to the connected peer.

`toggleAudio(mute)`

Parameters:

- `mute <boolean>`: If true, mutes the audio; otherwise, unmutes.

Toggles the local audio track on or off.

`toggleVideo(disable)`

Parameters:

- `disable <boolean>`: If true, disables the video; otherwise, enables it.

Toggles the local video track on or off.

`endCall()`

Ends the call by closing peer connections and media streams.

10.38 mathjs

`isComplex(x)`

Parameters:

- `x <*>`: The value to test.

Returns: `<boolean>`: Returns true if `x` is a Complex number, false otherwise.
Test whether a value is a Complex number.

`isBigNumber(x)`

Parameters:

- `x <*>`: The value to test.

Returns: `<boolean>`: Returns true if `x` is a BigNumber, false otherwise.
Test whether a value is a BigNumber.

`isFraction(x)`

Parameters:

- `x <*>`: The value to test.

Returns: `<boolean>`: Returns true if `x` is a Fraction, false otherwise.
Test whether a value is a Fraction.

`isUnit(x)`

Parameters:

- `x <*>`: The value to test.

Returns: `<boolean>`: Returns true if `x` is a Unit, false otherwise.
Test whether a value is a Unit.

`isMatrix(x)`

Parameters:

- `x <*>`: The value to test.

Returns: `<boolean>`: Returns true if `x` is a Matrix, false otherwise.
Test whether a value is a Matrix.

`isCollection(x)`

Parameters:

- `x <*>`: The value to test.

Returns: `<boolean>`: Returns true if `x` is an Array or Matrix, false otherwise.
Test whether a value is a collection (Array or Matrix).

`isDenseMatrix(x)`

Parameters:

- `x <*>`: The value to test.

Returns: `<boolean>`: Returns true if `x` is a DenseMatrix, false otherwise.
Test whether a value is a DenseMatrix.

`isSparseMatrix(x)`

Parameters:

- `x <*>`: The value to test.

Returns: `<boolean>`: Returns true if `x` is a SparseMatrix, false otherwise.
Test whether a value is a SparseMatrix.

`isRange(x)`

Parameters:

- `x <*>`: The value to test.

Returns: `<boolean>`: Returns true if `x` is a Range, false otherwise.
Test whether a value is a Range.

`isIndex(x)`

Parameters:

- `x <*>`: The value to test.

Returns: `<boolean>`: Returns true if `x` is an Index, false otherwise.
Test whether a value is an Index.

`isBoolean(x)`

Parameters:

- `x <*>`: The value to test.

Returns: `<boolean>`: Returns true if `x` is a boolean, false otherwise.
Test whether a value is a boolean.

`isResultSet(x)`

Parameters:

- `x <*>`: The value to test.

Returns: `<boolean>`: Returns true if `x` is a ResultSet, false otherwise.
Test whether a value is a ResultSet.

`isHelp(x)`

Parameters:

- `x <*>`: The value to test.

Returns: `<boolean>`: Returns true if `x` is a Help object, false otherwise.
Test whether a value is a Help object.

`isDate(x)`

Parameters:

- `x <*>`: The value to test.

Returns: `<boolean>`: Returns true if `x` is a Date object, false otherwise.
Test whether a value is a Date.

`isRegExp(x)`

Parameters:

- `x <*>`: The value to test.

Returns: `<boolean>`: Returns true if `x` is a RegExp object, false otherwise.
Test whether a value is a RegExp.

`isAccessorNode(x)`

Parameters:

- `x <*>`: The value to test.

Returns: `<boolean>`: Returns true if `x` is an AccessorNode, false otherwise.
Test whether a value is an AccessorNode.

`isArrayNode(x)`

Parameters:

- `x <*>`: The value to test.

Returns: <boolean>: Returns true if `x` is an ArrayNode, false otherwise.
Test whether a value is an ArrayNode.

`isAssignmentNode(x)`

Parameters:

- x <*>: The value to test.

Returns: <boolean>: Returns true if `x` is an AssignmentNode, false otherwise.
Test whether a value is an AssignmentNode.

`isBlockNode(x)`

Parameters:

- x <*>: The value to test.

Returns: <boolean>: Returns true if `x` is a BlockNode, false otherwise.
Test whether a value is a BlockNode.

`isConditionalNode(x)`

Parameters:

- x <*>: The value to test.

Returns: <boolean>: Returns true if `x` is a ConditionalNode, false otherwise.
Test whether a value is a ConditionalNode.

`isConstantNode(x)`

Parameters:

- x <*>: The value to test.

Returns: <boolean>: Returns true if `x` is a ConstantNode, false otherwise.
Test whether a value is a ConstantNode.

`isFunctionAssignmentNode(x)`

Parameters:

- x <*>: The value to test.

Returns: `<boolean>`: Returns true if `x` is a FunctionAssignmentNode, false otherwise.
Test whether a value is a FunctionAssignmentNode.

`isFunctionNode(x)`

Parameters:

- `x <*>`: The value to test.

Returns: `<boolean>`: Returns true if `x` is a FunctionNode, false otherwise.
Test whether a value is a FunctionNode.

`isIndexNode(x)`

Parameters:

- `x <*>`: The value to test.

Returns: `<boolean>`: Returns true if `x` is an IndexNode, false otherwise.
Test whether a value is an IndexNode.

`isNode(x)`

Parameters:

- `x <*>`: The value to test.

Returns: `<boolean>`: Returns true if `x` is a Node, false otherwise.
Test whether a value is a Node.

`isObjectNode(x)`

Parameters:

- `x <*>`: The value to test.

Returns: `<boolean>`: Returns true if `x` is an ObjectNode, false otherwise.
Test whether a value is an ObjectNode.

`isOperatorNode(x)`

Parameters:

- `x <*>`: The value to test.

Returns: `<boolean>`: Returns true if `x` is an OperatorNode, false otherwise.
Test whether a value is an OperatorNode.

`isParenthesisNode(x)`

Parameters:

- `x <*>`: The value to test.

Returns: `<boolean>`: Returns true if `x` is a ParenthesisNode, false otherwise.
Test whether a value is a ParenthesisNode.

`isRangeNode(x)`

Parameters:

- `x <*>`: The value to test.

Returns: `<boolean>`: Returns true if `x` is a RangeNode, false otherwise.
Test whether a value is a RangeNode.

`isRelationalNode(x)`

Parameters:

- `x <*>`: The value to test.

Returns: `<boolean>`: Returns true if `x` is a RelationalNode, false otherwise.
Test whether a value is a RelationalNode.

`isSymbolNode(x)`

Parameters:

- `x <*>`: The value to test.

Returns: `<boolean>`: Returns true if `x` is a SymbolNode, false otherwise.
Test whether a value is a SymbolNode.

`isChain(x)`

Parameters:

- `x <*>`: The value to test.

Returns: `<boolean>`: Returns true if `x` is a Chain, false otherwise.
Test whether a value is a Chain.

`on(event, callback)`

Parameters:

- `event <string>`: The event name to subscribe to.
- `callback <function>`: The callback function to execute when the event occurs.

Returns: `<void>`:
Subscribe to an event.

`off(event, [callback])`

Parameters:

- `event <string>`: The event name to unsubscribe from.
- `callback <function>`: The callback function to remove.

Returns: `<void>`:
Unsubscribe from an event.

`once(event, callback)`

Parameters:

- `event <string>`: The event name to subscribe to.
- `callback <function>`: The callback function to execute when the event occurs.

Returns: `<void>`:
Subscribe to an event once.

`emit(event, [data])`

Parameters:

- `event <string>`: The event name to emit.
- `data <*>`: The data to pass to the event handlers.

Returns: `<void>`:
Emit an event, triggering all bound callbacks.

`expression`

Type: `<Unknown>`
An object containing functions for parsing and evaluating expressions.

`import(object, [options])`

Parameters:

- `object` `<Object>`: An object or array of objects with functions or constants to import.
- `options` `<Object>`: Optional import options.

Returns: `<void>`:
Import functions or constants into math.js.

`create([config], [factories])`

Parameters:

- `config` `<Object>`: Optional configuration options.
- `factories` `<Array.<function()>>`: Optional list of factories to include.

Returns: `<PRDC_JSLAB_MATHJS_DOC>`: Returns a new instance of math.js.
Create a new, isolated math.js instance.

`factory(name, dependencies, create)`

Parameters:

- `name` `<string>`: The name of the function.
- `dependencies` `<Array.<string>>`: Array of dependency names.
- `create` `<function>`: Function to create the new function.

Returns: `<function>`: Returns the created function.
Factory function to create new functions.

`abs(x)`

Parameters:

- `x` `<number>`: A number or array with numbers.

Returns: `<number>`: Returns the absolute value of `x`.
Calculate the absolute value of a number.

```
AccessorNode(object, index)
```

Parameters:

- `object <Node>`: The object being accessed.
- `index <IndexNode>`: The index used to access the object.

Returns: `<AccessorNode>`: Returns a new AccessorNode.
A node representing access to a property or index.

```
acos(x)
```

Parameters:

- `x <number>`: Function input.

Returns: `<number>`: The arc cosine of `x`.
Calculate the inverse cosine of a value.

```
acosh(x)
```

Parameters:

- `x <number>`: Function input.

Returns: `<number>`: The inverse hyperbolic cosine of `x`.
Calculate the inverse hyperbolic cosine of a value.

```
acot(x)
```

Parameters:

- `x <number>`: Function input.

Returns: `<number>`: The inverse cotangent of `x`.
Calculate the inverse cotangent of a value.

```
acoth(x)
```

Parameters:

- `x <number>`: Function input.

Returns: `<number>`: The inverse hyperbolic cotangent of `x`.
Calculate the inverse hyperbolic cotangent of a value.

```
acsc(x)
```

Parameters:

- `x <number>`: Function input.

Returns: `<number>`: The inverse cosecant of `x`.
Calculate the inverse cosecant of a value.

```
acsch(x)
```

Parameters:

- `x <number>`: Function input.

Returns: `<number>`: The inverse hyperbolic cosecant of `x`.
Calculate the inverse hyperbolic cosecant of a value.

```
addScalar(x, y)
```

Parameters:

- `x <number>`: First value to add.
- `y <number>`: Second value to add.

Returns: `<number>`: Sum of `x` and `y`.
Add two scalar values, `x + y`.

```
and(x, y)
```

Parameters:

- `x <boolean>`: First input.
- `y <boolean>`: Second input.

Returns: `<boolean>`: Returns true when both `x` and `y` are true.
Logical AND. Returns true if both inputs are true.

```
apply(x, dim, callback)
```

Parameters:

- `x <Array>`: The input array or matrix.
- `dim <number>`: The dimension along which to apply the function.
- `callback <function>`: The function to apply.

Returns: `<Array>`: The result of applying the function along the specified dimension.
Apply a function to each entry in a matrix or array.

`arg(x)`

Parameters:

- `x <number>`: Function input.

Returns: `<number>`: The argument of `x`.
Calculate the argument of a complex number.

`ArrayNode(items)`

Parameters:

- `items <Array.<Node>>`: An array of nodes.

Returns: `<ArrayNode>`: Returns a new ArrayNode.
An array node representing an array in the expression tree.

`asec(x)`

Parameters:

- `x <number>`: Function input.

Returns: `<number>`: The inverse secant of `x`.
Calculate the inverse secant of a value.

`asech(x)`

Parameters:

- `x <number>`: Function input.

Returns: `<number>`: The inverse hyperbolic secant of `x`.
Calculate the inverse hyperbolic secant of a value.

`asin(x)`

Parameters:

- `x <number>`: Function input.

Returns: `<number>`: The inverse sine of `x`.
Calculate the inverse sine of a value.

`asinh(x)`

Parameters:

- `x <number>`: Function input.

Returns: `<number>`: The inverse hyperbolic sine of `x`.
Calculate the inverse hyperbolic sine of a value.

`AssignmentNode(object, value)`

Parameters:

- `object <Node>`: The symbol or AccessorNode to assign to.
- `value <Node>`: The value to assign.

Returns: `<AssignmentNode>`: Returns a new AssignmentNode.
An assignment node representing variable assignment.

`atan(x)`

Parameters:

- `x <number>`: Function input.

Returns: `<number>`: The inverse tangent of `x`.
Calculate the inverse tangent of a value.

`atan2(y, x)`

Parameters:

- `y <number>`: Dividend.
- `x <number>`: Divisor.

Returns: `<number>`: The inverse tangent of `y/x`.
Calculate the inverse tangent of `y/x`.

`atanh(x)`

Parameters:

- `x <number>`: Function input.

Returns: `<number>`: The inverse hyperbolic tangent of `x`.
Calculate the inverse hyperbolic tangent of a value.

`atomicMass`

Type: `<Unknown>`
Atomic mass constant, expressed in kg.

`avogadro`

Type: `<Unknown>`
Avogadro's number, approximately `6.022e23` mol¡sup¿-1¡/sup¿.

`bellNumbers(n)`

Parameters:

- `n <number>`: The input value.

Returns: `<number>`: Returns the nth Bell number.
Compute the Bell Numbers, `B(n)`.

`BigNumber(value)`

Parameters:

- `value <number>`: The numeric value.

Returns: `<BigNumber>`: Returns a new BigNumber instance.
BigNumber constructor.

`bignumber(value)`

Parameters:

- `value <number>`: The numeric value.

Returns: `<BigNumber>`: Returns a BigNumber instance.
Create a BigNumber with arbitrary precision.

`bin(n)`

Parameters:

- `n <number>`: The number to format.

Returns: `<string>`: The binary representation of `n`.
Format a number as binary.

`bitAnd(x, y)`

Parameters:

- `x <number>`: First value.
- `y <number>`: Second value.

Returns: `<number>`: Result of `x AND y`.
Bitwise AND operation.

`bitNot(x)`

Parameters:

- `x <number>`: Input value.

Returns: `<number>`: Result of `NOT x`.
Bitwise NOT operation.

`bitOr(x, y)`

Parameters:

- `x <number>`: First value.
- `y <number>`: Second value.

Returns: `<number>`: Result of `x OR y`.
Bitwise OR operation.

`bitXor(x, y)`

Parameters:

- `x <number>`: First value.
- `y <number>`: Second value.

Returns: `<number>`: Result of `x XOR y`.
Bitwise XOR operation.

`BlockNode(blocks)`

Parameters:

- `blocks <Array.<Object>>`: An array of statements.

Returns: `<BlockNode>`: Returns a new BlockNode.
A node representing a block of statements.

`bohrMagneton`

Type: `<Unknown>`
The Bohr magneton in units of `J/T`.

`bohrRadius`

Type: `<Unknown>`
The Bohr radius in meters.

`boltzmann`

Type: `<Unknown>`
The Boltzmann constant in `J/K`.

`boolean(x)`

Parameters:

- `x <*>`: The value to parse.

Returns: `<boolean>`: The parsed boolean value.
Parse a value into a boolean.

`catalan`

Type: `<Unknown>`
The Catalan's constant.

`cbrt(x)`

Parameters:

- `x <number>`: Function input.

Returns: `<number>`: The cube root of `x`.
Calculate the cube root of a value.

`ceil(x)`

Parameters:

- `x <number>`: Input value.

Returns: `<number>`: The rounded value.
Round a value towards plus infinity.

`chain(value)`

Parameters:

- `value <*>`: The initial value of the chain.

Returns: `<Chain>`: A chain object.
Create a chained operation, allowing to chain methods.

`Chain(value)`

Parameters:

- `value <*>`: The initial value.

Returns: `<Chain>`: Returns a new Chain instance.
Chain constructor.

`classicalElectronRadius`

Type: `<Unknown>`
Classical electron radius in meters.

`combinations(n, k)`

Parameters:

- `n <number>`: Total number of items.
- `k <number>`: Number of items to choose.

Returns: `<number>`: Number of possible combinations.
Calculate the number of combinations of n items taken k at a time.

`combinationsWithRep(n, k)`

Parameters:

- `n <number>`: Total number of items.
- `k <number>`: Number of items to choose.

Returns: <number>: Number of possible combinations with replacement.
Calculate the number of combinations with replacement of n items taken k at a time.

```
compare(x, y)
```

Parameters:

- x <number>: First value to compare.
- y <number>: Second value to compare.

Returns: <number>: Returns 1 when x ¿ y, -1 when x ¡ y, and 0 when x == y.
Compare two values numerically.

```
compareNatural(x, y)
```

Parameters:

- x <string>: First string to compare.
- y <string>: Second string to compare.

Returns: <number>: Returns 1 when x ¿ y, -1 when x ¡ y, and 0 when x == y.
Compare two strings using natural order.

```
compareText(x, y)
```

Parameters:

- x <string>: First string to compare.
- y <string>: Second string to compare.

Returns: <number>: Returns 1 when x ¿ y, -1 when x ¡ y, and 0 when x == y.
Compare two strings lexicographically.

```
compile(expr)
```

Parameters:

- expr <string>: The expression to compile.

Returns: <Object>: A compiled expression that can be evaluated with `eval`.
Compile an expression into a compiled function for faster evaluation.

```
complex([re], [im])
```

Parameters:

- `re <number>`: Real part or a string representation.
- `im <number>`: Imaginary part.

Returns: `<Complex>`: Returns a Complex number.
Create a complex number.

```
Complex([re], [im])
```

Parameters:

- `re <number>`: Real part or a string representation.
- `im <number>`: Imaginary part.

Returns: `<Complex>`: A new Complex number.
Complex number constructor.

```
composition(n, k)
```

Parameters:

- `n <number>`: Total number of items.
- `k <number>`: Number of parts.

Returns: `<number>`: Number of compositions.
Calculate the composition count of n into k parts.

```
concat(a, b, [dim])
```

Parameters:

- `a <Array>`: First array or matrix.
- `b <Array>`: Other arrays or matrices.
- `dim <number>`: Dimension along which to concatenate.

Returns: `<Array>`: Concatenated array or matrix.
Concatenate matrices or arrays along a specified dimension.

```
ConditionalNode(condition, trueExpr, falseExpr)
```

Parameters:

- `condition <Node>`: The condition expression.

- `trueExpr <Node>`: Expression to evaluate when condition is true.
- `falseExpr <Node>`: Expression to evaluate when condition is false.

Returns: `<ConditionalNode>`: A new ConditionalNode instance.
A node representing a conditional expression.

`conductanceQuantum`

Type: `<Unknown>`
Conductance quantum in Siemens.

`conj(x)`

Parameters:

- `x <number>`: Input value.

Returns: `<number>`: The complex conjugate of x.
Compute the complex conjugate of a complex number.

`ConstantNode(value)`

Parameters:

- `value <number>`: The constant value.

Returns: `<ConstantNode>`: A new ConstantNode instance.
A node representing a constant value.

`cos(x)`

Parameters:

- `x <number>`: Function input.

Returns: `<number>`: The cosine of x.
Calculate the cosine of a value.

`cosh(x)`

Parameters:

- `x <number>`: Function input.

Returns: `<number>`: The hyperbolic cosine of x.
Calculate the hyperbolic cosine of a value.

`cot(x)`

Parameters:

- `x <number>`: Function input.

Returns: `<number>`: The cotangent of x.
Calculate the cotangent of a value.

`coth(x)`

Parameters:

- `x <number>`: Function input.

Returns: `<number>`: The hyperbolic cotangent of x.
Calculate the hyperbolic cotangent of a value.

`coulomb`

Type: `<Unknown>`
Coulomb's constant in Nm^2/C^2.

`count(x)`

Parameters:

- `x <Array>`: The input array or matrix.

Returns: `<number>`: The number of elements.
Count the number of elements in a matrix or array.

`createUnit(name, definition, [options])`

Parameters:

- `name <string>`: The name of the new unit.
- `definition <string>`: Definition of the unit.
- `options <Object>`: Configuration options.

Returns: `<Unit>`: The created unit.
Create a user-defined unit and register it with the Unit system.

`csc(x)`

Parameters:

- `x <number>`: Function input.

Returns: `<number>`: The cosecant of x.
Calculate the cosecant of a value.

`csch(x)`

Parameters:

- `x <number>`: Function input.

Returns: `<number>`: The hyperbolic cosecant of x.
Calculate the hyperbolic cosecant of a value.

`ctranspose(x)`

Parameters:

- `x <Array>`: The matrix to transpose.

Returns: `<Array>`: The conjugate transpose of x.
Compute the conjugate transpose of a matrix.

`cube(x)`

Parameters:

- `x <number>`: Input value.

Returns: `<number>`: The cube of x.
Compute the cube of a value.

`cumsum(x, [dim])`

Parameters:

- `x <Array>`: The input array or matrix.
- `dim <number>`: Dimension along which to calculate.

Returns: `<Array>`: The cumulative sum.
Compute the cumulative sum of a matrix or array.

`deepEqual(x, y)`

Parameters:

- `x <*>`: First value to compare.
- `y <*>`: Second value to compare.

Returns: `<boolean>`: Returns true if x and y are deep equal.
Test element-wise whether two values are equal.

`DenseMatrix(data)`

Parameters:

- `data <Array>`: The data for the matrix.

Returns: `<DenseMatrix>`: A new DenseMatrix instance.
Dense matrix constructor.

`derivative(expr, variable, [options])`

Parameters:

- `expr <string>`: The expression to differentiate.
- `variable <string>`: The variable with respect to which to differentiate.
- `options <Object>`: Optional options object.

Returns: `<Node>`: The derivative of the expression.
Take the derivative of an expression.

`deuteronMass`

Type: `<Unknown>`
Deuteron mass in kilograms.

`diff(x, [dim])`

Parameters:

- `x <Array>`: Input array or matrix.
- `dim <number>`: Dimension along which to calculate the difference.

Returns: `<Array>`: The differences.
Calculate the differences between adjacent values in a matrix or array.

`divideScalar(x, y)`

Parameters:

- `x <number>`: Numerator.
- `y <number>`: Denominator.

Returns: `<number>`: The result of division.
Divide two scalar values, x / y.

`dotDivide(x, y)`

Parameters:

- `x <Array>`: Numerator matrix.
- `y <Array>`: Denominator matrix.

Returns: `<Array>`: The element-wise division.
Divide two matrices element-wise.

`dotMultiply(x, y)`

Parameters:

- `x <Array>`: First matrix.
- `y <Array>`: Second matrix.

Returns: `<Array>`: The element-wise multiplication.
Multiply two matrices element-wise.

`dotPow(x, y)`

Parameters:

- `x <Array>`: Base matrix.
- `y <Array>`: Exponent matrix.

Returns: `<Array>`: The element-wise exponentiation.
Exponentiate two matrices element-wise.

`e`

Type: `<Unknown>`
Euler's number, the base of natural logarithms.

`efimovFactor`

Type: `<Unknown>`
Efimov factor.

`eigs(x)`

Parameters:

- `x` `<Array>`: A square matrix.

Returns: `<Object>`: An object containing eigenvalues and eigenvectors.
Calculate eigenvalues and eigenvectors of a matrix.

`electricConstant`

Type: `<Unknown>`
Electric constant (vacuum permittivity) in F/m.

`electronMass`

Type: `<Unknown>`
Electron mass in kilograms.

`elementaryCharge`

Type: `<Unknown>`
Elementary charge in coulombs.

`equal(x, y)`

Parameters:

- `x` `<*>`: First value to compare.
- `y` `<*>`: Second value to compare.

Returns: `<boolean>`: Returns true if x equals y.
Test whether two values are equal.

`equalScalar(x, y)`

Parameters:

- `x <number>`: First value.
- `y <number>`: Second value.

Returns: `<boolean>`: Returns true if x equals y.
Test whether two scalar values are equal.

```
equalText(x, y)
```

Parameters:

- `x <string>`: First string.
- `y <string>`: Second string.

Returns: `<boolean>`: Returns true if x equals y.
Test whether two strings are equal.

```
erf(x)
```

Parameters:

- `x <number>`: Input value.

Returns: `<number>`: The error function evaluated at x.
Calculate the error function of a value.

```
exp(x)
```

Parameters:

- `x <number>`: Exponent.

Returns: `<number>`: The exponential of x.
Calculate the exponential of a value.

```
expm1(x)
```

Parameters:

- `x <number>`: Input value.

Returns: `<number>`: The result of exp(x) - 1.
Calculate exp(x) - 1.

`factorial(n)`

Parameters:

- `n <number>`: A non-negative integer.

Returns: `<number>`: The factorial of n.
Calculate the factorial of a value.

`false`

Type: `<Unknown>`
Boolean value false.

`faraday`

Type: `<Unknown>`
Faraday constant in C/mol.

`fermiCoupling`

Type: `<Unknown>`
Fermi coupling constant in GeV^-2.

`fft(x)`

Parameters:

- `x <Array>`: Input array or matrix.

Returns: `<Array>`: The FFT of x.
Compute the Fast Fourier Transform of a matrix or array.

`FibonacciHeap([compare])`

Parameters:

- `compare <function>`: Comparison function.

Returns: `<FibonacciHeap>`: A new FibonacciHeap instance.
Fibonacci heap data structure.

`filter(x, test)`

Parameters:

- `x <Array>`: The input array or matrix.
- `test <function>`: The test function.

Returns: `<Array>`: The filtered array or matrix.
Filter the items in an array or matrix.

`fineStructure`

Type: `<Unknown>`
Fine-structure constant.

`firstRadiation`

Type: `<Unknown>`
First radiation constant in Wmˆ2.

`fix(x)`

Parameters:

- `x <number>`: Input value.

Returns: `<number>`: The rounded value.
Round a value towards zero.

`flatten(x)`

Parameters:

- `x <Array>`: The input array or matrix.

Returns: `<Array>`: The flattened array or matrix.
Flatten a multi-dimensional array or matrix into a single dimension.

`floor(x)`

Parameters:

- `x <number>`: Input value.

Returns: `<number>`: The rounded value.
Round a value towards negative infinity.

`forEach(x, callback)`

Parameters:

- `x <Array>`: The input array or matrix.
- `callback <function>`: The function to execute on each element.

Returns: `<void>`:
Iterate over each element of a matrix or array.

```
format(value, [options])
```

Parameters:

- `value <*>`: The value to format.
- `options <Object>`: Formatting options or custom function.

Returns: `<string>`: The formatted value.
Format a value for display.

```
fraction(numerator, [denominator])
```

Parameters:

- `numerator <number>`: Numerator.
- `denominator <number>`: Denominator.

Returns: `<Fraction>`: A new Fraction instance.
Create a fraction.

```
Fraction(numerator, [denominator])
```

Parameters:

- `numerator <number>`: Numerator.
- `denominator <number>`: Denominator.

Returns: `<Fraction>`: A new Fraction instance.
Fraction constructor.

```
FunctionAssignmentNode(name, params, expr)
```

Parameters:

- `name <string>`: The name of the function being assigned.

- `params <Array.<string>>`: An array of parameter names.
- `expr <Node>`: The function expression.

Returns: `<FunctionAssignmentNode>`: A new FunctionAssignmentNode instance.
A node representing a function assignment in the expression tree.

`FunctionNode(name, args)`

Parameters:

- `name <string>`: The name of the function or a SymbolNode.
- `args <Array.<Node>>`: An array of argument nodes.

Returns: `<FunctionNode>`: A new FunctionNode instance.
A node representing a function call in the expression tree.

`gamma(n)`

Parameters:

- `n <number>`: The input value.

Returns: `<number>`: The gamma of n.
Calculate the gamma function of a value.

`gasConstant`

Type: `<Unknown>`
The molar gas constant, in units of J/(molK).

`gcd(args)`

Parameters:

- `args <number>`: Two or more integer numbers.

Returns: `<number>`: The greatest common divisor.
Compute the greatest common divisor of two or more values.

`getMatrixDataType(matrix)`

Parameters:

- `matrix <Matrix>`: The input matrix.

Returns: `<string>`: The data type of the matrix elements.
Get the data type of a matrix.

`gravitationConstant`

Type: `<Unknown>`
Newtonian constant of gravitation, in mˆ3/(kg sˆ2).

`gravity`

Type: `<Unknown>`
Acceleration due to gravity on Earth, in m/sˆ2.

`hartreeEnergy`

Type: `<Unknown>`
Hartree energy, in joules.

`hasNumericValue(x)`

Parameters:

- `x <*>`: The value to test.

Returns: `<boolean>`: Returns true if x is numeric.
Test whether a value is a numeric value.

`Help(value)`

Parameters:

- `value <*>`: The function or object to get help for.

Returns: `<Help>`: A new Help instance.
Help object constructor.

`hex(value)`

Parameters:

- `value <number>`: The value to format.

Returns: `<string>`: The hexadecimal representation.
Format a number as hexadecimal.

`hypot(args)`

Parameters:

- `args <number>`: The input values.

Returns: `<number>`: The hypotenuse.
Calculate the hypotenuse of a list of values.

`i`

Type: `<Unknown>`
The imaginary unit 'i'.

`identity(size, [format])`

Parameters:

- `size <number>`: The size of the matrix.
- `format <string>`: The matrix format.

Returns: `<Matrix>`: An identity matrix.
Create an identity matrix.

`ifft(x)`

Parameters:

- `x <Array>`: Input array or matrix.

Returns: `<Array>`: The inverse FFT of x.
Compute the inverse Fast Fourier Transform.

`im(x)`

Parameters:

- `x <number>`: Input value.

Returns: `<number>`: The imaginary part of x.
Get the imaginary part of a complex number.

`ImmutableDenseMatrix(data)`

Parameters:

- `data <Array>`: The data for the matrix.

Returns: `<ImmutableDenseMatrix>`: A new ImmutableDenseMatrix instance.
Immutable dense matrix constructor.

`Index(ranges)`

Parameters:

- `ranges <Range>`: Ranges or indices.

Returns: `<Index>`: A new Index instance.
Index constructor for matrices.

`IndexNode(dimensions)`

Parameters:

- `dimensions <Array.<Node>>`: The indices.

Returns: `<IndexNode>`: A new IndexNode instance.
A node representing an index operation in the expression tree.

`intersect(a, b)`

Parameters:

- `a <Array>`: First set.
- `b <Array>`: Second set.

Returns: `<Array>`: The intersection of a and b.
Compute the intersection of two sets.

`inverseConductanceQuantum`

Type: `<Unknown>`
Inverse conductance quantum, in ohms.

`invmod(a, m)`

Parameters:

- `a <number>`: The value.
- `m <number>`: The modulus.

Returns: `<number>`: The modular inverse.
Calculate the modular inverse of a value.

`isInteger(x)`

Parameters:

- `x <number>`: The value to test.

Returns: `<boolean>`: Returns true if x is an integer.
Test whether a value is an integer.

`isNegative(x)`

Parameters:

- `x <number>`: The value to test.

Returns: `<boolean>`: Returns true if x is negative.
Test whether a value is negative.

`isPositive(x)`

Parameters:

- `x <number>`: The value to test.

Returns: `<boolean>`: Returns true if x is positive.
Test whether a value is positive.

`isPrime(x)`

Parameters:

- `x <number>`: The value to test.

Returns: `<boolean>`: Returns true if x is prime.
Test whether a number is prime.

`isZero(x)`

Parameters:

- `x <number>`: The value to test.

Returns: `<boolean>`: Returns true if x is zero.
Test whether a value is zero.

`kldivergence(p, q)`

Parameters:

- `p` `<Array>`: First probability distribution.
- `q` `<Array>`: Second probability distribution.

Returns: `<number>`: The KL divergence D¡sub¿KL¡/sub¿(p —— q).
Calculate the Kullback-Leibler divergence between two distributions.

`klitzing`

Type: `<Unknown>`
Von Klitzing constant, in ohms.

`kron(A, B)`

Parameters:

- `A` `<Array>`: First matrix.
- `B` `<Array>`: Second matrix.

Returns: `<Matrix>`: The Kronecker product of A and B.
Compute the Kronecker product of two matrices.

`LN10`

Type: `<Unknown>`
Natural logarithm of 10.

`LN2`

Type: `<Unknown>`
Natural logarithm of 2.

`LOG10E`

Type: `<Unknown>`
Base 10 logarithm of e.

`LOG2E`

Type: `<Unknown>`
Base 2 logarithm of e.

`larger(x, y)`

Parameters:

- `x <number>`: First value.
- `y <number>`: Second value.

Returns: `<boolean>`: Returns true if x ¿ y.
Test whether value x is larger than y.

`largerEq(x, y)`

Parameters:

- `x <number>`: First value.
- `y <number>`: Second value.

Returns: `<boolean>`: Returns true if x ¿= y.
Test whether value x is larger than or equal to y.

`lcm(args)`

Parameters:

- `args <number>`: Two or more integer numbers.

Returns: `<number>`: The least common multiple.
Compute the least common multiple of two or more values.

`leafCount(node)`

Parameters:

- `node <Node>`: The root node of the expression tree.

Returns: `<number>`: The number of leaf nodes.
Count the number of leaf nodes in an expression tree.

`leftShift(x, y)`

Parameters:

- `x <number>`: Value to be shifted.
- `y <number>`: Amount of bits to shift.

Returns: `<number>`: The shifted value.
Bitwise left shift operation.

`lgamma(n)`

Parameters:

- `n <number>`: The input value.

Returns: `<number>`: The natural logarithm of the gamma function at n.
Compute the natural logarithm of the gamma function.

`log(x, [base])`

Parameters:

- `x <number>`: Value for which to calculate the logarithm.
- `base <number>`: Base of the logarithm.

Returns: `<number>`: The logarithm of x.
Calculate the natural logarithm of a value.

`log10(x)`

Parameters:

- `x <number>`: Value for which to calculate the logarithm.

Returns: `<number>`: The base-10 logarithm of x.
Calculate the base-10 logarithm of a value.

`log1p(x)`

Parameters:

- `x <number>`: Input value.

Returns: `<number>`: The result of log(1 + x).
Calculate the natural logarithm of 1 plus a value.

`log2(x)`

Parameters:

- `x <number>`: Value for which to calculate the logarithm.

Returns: `<number>`: The base-2 logarithm of x.
Calculate the base-2 logarithm of a value.

`loschmidt`

Type: `<Unknown>`
Loschmidt constant at 0°C and 1 atm, in mˆ-3.

`lsolve(L, b)`

Parameters:

- `L <Matrix>`: A lower triangular matrix.
- `b <Matrix>`: A column vector.

Returns: `<Matrix>`: The solution vector x.
Solve a linear system A * x = b where A is a lower triangular matrix.

`lsolveAll(L, b)`

Parameters:

- `L <Matrix>`: A lower triangular matrix.
- `b <Matrix>`: A column vector.

Returns: `<Array>`: An array of solutions.
Find all solutions of a linear system A * x = b where A is a lower triangular matrix.

`lup(A, [threshold])`

Parameters:

- `A <Matrix>`: A square matrix.
- `threshold <number>`: Tolerance threshold.

Returns: `<Object>`: An object containing L, U, and P matrices.
Compute the LU decomposition with partial pivoting.

`lusolve(A, b, [order])`

Parameters:

- `A <Matrix>`: Coefficient matrix.

- `b <Matrix>`: Right-hand side vector or matrix.
- `order <string>`: Matrix storage order.

Returns: `<Matrix>`: The solution vector or matrix.
Solve a linear system using LU decomposition.

`lyap(A, Q)`

Parameters:

- `A <Matrix>`: A square matrix.
- `Q <Matrix>`: A square matrix.

Returns: `<Matrix>`: Solution X of the Lyapunov equation.
Solve the Lyapunov equation A*X + X*A' = Q.

`mad(array, [dim])`

Parameters:

- `array <Array>`: Input array.
- `dim <number>`: Dimension along which to compute.

Returns: `<number>`: The median absolute deviation.
Compute the median absolute deviation of a set of values.

`magneticConstant`

Type: `<Unknown>`
Magnetic constant (vacuum permeability), in N/A^2.

`magneticFluxQuantum`

Type: `<Unknown>`
Magnetic flux quantum, in Wb.

`map(x, callback)`

Parameters:

- `x <Array>`: The input array or matrix.
- `callback <function>`: The function to apply.

Returns: `<Array>`: The result after applying the callback.
Map a function over the elements of a matrix or array.

```
matrix([data], [format])
```

Parameters:

- `data <Array>`: The data for the matrix.
- `format <string>`: The matrix format.

Returns: `<Matrix>`: A new Matrix instance.
Create a matrix.

```
Matrix([data], [format])
```

Parameters:

- `data <Array>`: The data for the matrix.
- `format <string>`: The matrix format.

Returns: `<Matrix>`: A new Matrix instance.
Matrix constructor.

```
matrixFromColumns(columns)
```

Parameters:

- `columns <Array>`: Columns to construct the matrix.

Returns: `<Matrix>`: The constructed matrix.
Create a matrix from given columns.

```
matrixFromFunction(size, callback)
```

Parameters:

- `size <Array>`: The size of the matrix.
- `callback <function>`: Function to generate matrix entries.

Returns: `<Matrix>`: The generated matrix.
Create a matrix using a provided function.

```
matrixFromRows(rows)
```

Parameters:

- `rows <Array>`: Rows to construct the matrix.

Returns: `<Matrix>`: The constructed matrix.
Create a matrix from given rows.

`mean(array, [dim])`

Parameters:

- `array <Array>`: Input array.
- `dim <number>`: Dimension along which to compute.

Returns: `<number>`: The mean value.
Compute the arithmetic mean of a set of values.

`median(array, [dim])`

Parameters:

- `array <Array>`: Input array.
- `dim <number>`: Dimension along which to compute.

Returns: `<number>`: The median value.
Compute the median of a set of values.

`mod(x, y)`

Parameters:

- `x <number>`: Dividend.
- `y <number>`: Divisor.

Returns: `<number>`: The remainder after division.
Calculate the modulus of two numbers.

`mode(array)`

Parameters:

- `array <Array>`: Input array.

Returns: `<Array>`: The mode(s) of the array.
Compute the mode of a set of values.

`molarMass`

Type: `<Unknown>`
Molar mass constant, in kg/mol.

`molarMassC12`

Type: `<Unknown>`
Molar mass of carbon-12, in kg/mol.

`molarPlanckConstant`

Type: `<Unknown>`
Molar Planck constant, in J·s/mol.

`molarVolume`

Type: `<Unknown>`
Molar volume of an ideal gas at 1 atm and 0°C, in m^3/mol.

`multinomial(args)`

Parameters:

- `args <number>`: Integer numbers.

Returns: `<number>`: The multinomial coefficient.
Compute the multinomial coefficient of a list of integers.

`multiplyScalar(x, y)`

Parameters:

- `x <number>`: First value.
- `y <number>`: Second value.

Returns: `<number>`: The product of x and y.
Multiply two scalar values, x * y.

`neutronMass`

Type: `<Unknown>`
Neutron mass, in kilograms.

`not(x)`

Parameters:

- `x <boolean>`: Input value.

Returns: `<boolean>`: The logical negation of x.
Logical NOT operation.

`nthRoot(a, [root])`

Parameters:

- `a <number>`: The value.
- `root <number>`: The root.

Returns: `<number>`: The nth root of a.
Calculate the nth root of a value.

`nthRoots(a, n)`

Parameters:

- `a <number>`: The value.
- `n <number>`: The degree of the root.

Returns: `<Array>`: An array of the n roots.
Calculate the nth roots of a complex number.

`nuclearMagneton`

Type: `<Unknown>`
Nuclear magneton, in J/T.

`null`

Type: `<Unknown>`
JavaScript null value.

`number(value)`

Parameters:

- `value <*>`: The value to parse.

Returns: `<number>`: The numeric value.
Parse a value into a number.

```
numeric(value)
```

Parameters:

- `value <*>`: The value to convert.

Returns: `<number>`: The numeric representation.
Convert a math.js data type to a numeric type.

```
ObjectNode(properties)
```

Parameters:

- `properties <Object>`: Object properties as nodes.

Returns: `<ObjectNode>`: A new ObjectNode instance.
A node representing an object in the expression tree.

```
oct(value)
```

Parameters:

- `value <number>`: The value to format.

Returns: `<string>`: The octal representation.
Format a number as octal.

```
OperatorNode(op, fn, args, [implicit])
```

Parameters:

- `op <string>`: The operator symbol.
- `fn <string>`: The function name.
- `args <Array.<Node>>`: An array of argument nodes.
- `implicit <boolean>`: Is the operator implicit?

Returns: `<OperatorNode>`: A new OperatorNode instance.
A node representing an operator in the expression tree.

```
or(x, y)
```

Parameters:

- `x <boolean>`: First value.
- `y <boolean>`: Second value.

Returns: `<boolean>`: The logical OR of x and y.
Logical OR operation.

`ParenthesisNode(content)`

Parameters:

- `content <Node>`: The node encapsulated by the parentheses.

Returns: `<ParenthesisNode>`: A new ParenthesisNode instance.
A node representing parentheses in the expression tree.

`parse(expr)`

Parameters:

- `expr <string>`: The expression to parse.

Returns: `<Node>`: The parsed expression node(s).
Parse and evaluate an expression.

`parser()`

Returns: `<Parser>`: A new Parser instance.
Create a parser with its own scope and functions.

`Parser()`

Returns: `<Parser>`: A new Parser instance.
Parser constructor.

`partitionSelect(x, k, [compare])`

Parameters:

- `x <Array>`: The input array or matrix.
- `k <number>`: The kth smallest value to select.
- `compare <function>`: Optional comparison function.

Returns: `<number>`: The selected element.
Select an element from a matrix or array based on its sorted position.

`permutations(n, [k])`

Parameters:

- `n <number>`: Total number of items.
- `k <number>`: Number of items to choose.

Returns: `<number>`: Number of possible permutations.
Calculate the number of permutations of n items taken k at a time.

`phi`

Type: `<Unknown>`
The golden ratio, approximately 1.618.

`pi`

Type: `<Unknown>`
The mathematical constant pi.

`pickRandom(array, [number])`

Parameters:

- `array <Array>`: Array to pick values from.
- `number <number>`: Number of values to pick.

Returns: `<*>`: Picked value(s).
Randomly pick one or more values from a list.

`pinv(x)`

Parameters:

- `x <Array>`: A matrix.

Returns: `<Matrix>`: The pseudoinverse of x.
Compute the pseudoinverse of a matrix.

`planckCharge`

Type: `<Unknown>`
Planck charge, in coulombs.

`planckConstant`

Type: `<Unknown>`
Planck constant, in joule seconds.

`planckLength`

Type: `<Unknown>`
Planck length, in meters.

`planckMass`

Type: `<Unknown>`
Planck mass, in kilograms.

`planckTemperature`

Type: `<Unknown>`
Planck temperature, in kelvin.

`planckTime`

Type: `<Unknown>`
Planck time, in seconds.

`polynomialRoot(coefficients)`

Parameters:

- `coefficients` `<Array>`: Coefficients of the polynomial.

Returns: `<Array>`: An array of roots.
Find roots of a univariate polynomial.

`pow(x, y)`

Parameters:

- `x` `<number>`: Base.
- `y` `<number>`: Exponent.

Returns: `<number>`: x raised to the power y.
Calculate the power of x to y, x^y.

`prod(array, [dim])`

Parameters:

- `array <Array>`: Input array.
- `dim <number>`: Dimension along which to compute the product.

Returns: `<number>`: The product of all values.
Compute the product of a set of values.

`protonMass`

Type: `<Unknown>`
Proton mass, in kilograms.

`qr(x)`

Parameters:

- `x <Matrix>`: A matrix.

Returns: `<Object>`: An object containing Q and R matrices.
Compute the QR decomposition of a matrix.

`quantileSeq(data, prob, [sorted])`

Parameters:

- `data <Array>`: Input data.
- `prob <number>`: Probability or array of probabilities.
- `sorted <boolean>`: Is data sorted?

Returns: `<number>`: The quantile(s).
Compute the quantile of a sequence.

`quantumOfCirculation`

Type: `<Unknown>`
Quantum of circulation, in m^2/s.

`randomInt([min], max)`

Parameters:

- `min <number>`: Minimum value, inclusive.
- `max <number>`: Maximum value, inclusive.

Returns: `<number>`: A random integer.
Generate a random integer between min and max.

```
Range(start, end, [step])
```

Parameters:

- `start <*>`: Start of the range.
- `end <*>`: End of the range.
- `step <*>`: Step size.

Returns: `<Range>`: A new Range instance.
Create a range.

```
RangeNode(start, end, [step])
```

Parameters:

- `start <Node>`: Start node.
- `end <Node>`: End node.
- `step <Node>`: Step node.

Returns: `<RangeNode>`: A new RangeNode instance.
A node representing a range in the expression tree.

```
rationalize(expr, [scope], [options])
```

Parameters:

- `expr <string>`: The expression to rationalize.
- `scope <Object>`: Scope of variables.
- `options <Object>`: Options object.

Returns: `<Object>`: An object with expression and denominator.
Rationalize an expression.

`re(x)`

Parameters:

- `x <number>`: Input value.

Returns: `<number>`: The real part of x.
Get the real part of a complex number.

`reducedPlanckConstant`

Type: `<Unknown>`
Reduced Planck constant, in joule seconds.

`RelationalNode(condition, params)`

Parameters:

- `condition <string>`: Relational condition.
- `params <Node>`: Parameters.

Returns: `<RelationalNode>`: A new RelationalNode instance.
A node representing a relational operation.

`replacer(key, value)`

Parameters:

- `key <string>`: Property name.
- `value <*>`: Property value.

Returns: `<*>`: The transformed value.
Replacer function for JSON serialization.

`resize(x, size, [defaultValue])`

Parameters:

- `x <Array>`: The input matrix.
- `size <Array>`: The new size.
- `defaultValue <*>`: Default value for new entries.

Returns: `<Array>`: The resized matrix.
Resize a matrix.

`resolve(name)`

Parameters:

- `name <string>`: The name to resolve.

Returns: `<*>`: The resolved value.
Resolve the value of a symbol or function.

`ResultSet(entries)`

Parameters:

- `entries <Array>`: The entries in the result set.

Returns: `<ResultSet>`: A new ResultSet instance.
ResultSet constructor.

`reviver(key, value)`

Parameters:

- `key <string>`: Property name.
- `value <*>`: Property value.

Returns: `<*>`: The transformed value.
Reviver function for JSON deserialization.

`rightArithShift(x, y)`

Parameters:

- `x <number>`: The value to shift.
- `y <number>`: The amount to shift.

Returns: `<number>`: The shifted value.
Bitwise right arithmetic shift operation.

`rightLogShift(x, y)`

Parameters:

- `x <number>`: The value to shift.

- `y <number>`: The amount to shift.

Returns: `<number>`: The shifted value.
Bitwise right logical shift operation.

`rotate(x, [turns])`

Parameters:

- `x <Array>`: The input matrix.
- `turns <number>`: Number of 90-degree rotations.

Returns: `<Array>`: The rotated matrix.
Rotate the elements of a matrix.

`rotationMatrix(theta)`

Parameters:

- `theta <number>`: Rotation angle in radians.

Returns: `<Matrix>`: The rotation matrix.
Create a 2D rotation matrix.

`rydberg`

Type: `<Unknown>`
Rydberg constant, in mˆ-1.

`SQRT1_2`

Type: `<Unknown>`
Square root of 1/2.

`SQRT2`

Type: `<Unknown>`
Square root of 2.

`sackurTetrode`

Type: `<Unknown>`
Sackur-Tetrode constant at 1 atm, in J/K.

`schur(A)`

Parameters:

- `A <Matrix>`: A square matrix.

Returns: `<Object>`: An object containing matrices U and T.
Compute the Schur decomposition of a matrix.

`sec(x)`

Parameters:

- `x <number>`: Function input.

Returns: `<number>`: The secant of x.
Calculate the secant of a value.

`sech(x)`

Parameters:

- `x <number>`: Function input.

Returns: `<number>`: The hyperbolic secant of x.
Calculate the hyperbolic secant of a value.

`secondRadiation`

Type: `<Unknown>`
Second radiation constant, in m·K.

`setCartesian(sets)`

Parameters:

- `sets <Array>`: The sets to compute the product of.

Returns: `<Array>`: The Cartesian product.
Create the Cartesian product of two or more sets.

`setDifference(a, b)`

Parameters:

- `a <Array>`: First set.
- `b <Array>`: Second set.

Returns: `<Array>`: The difference a \{} b.
Compute the difference between two sets.

`setDistinct(a)`

Parameters:

- `a <Array>`: Input set.

Returns: `<Array>`: A set with distinct elements.
Remove duplicate elements from a set.

`setIntersect(a, b, [others])`

Parameters:

- `a <Array>`: First set.
- `b <Array>`: Second set.
- `others <Array>`: Additional sets.

Returns: `<Array>`: The intersection of the sets.
Compute the intersection of two or more sets.

`setIsSubset(a, b)`

Parameters:

- `a <Array>`: Potential subset.
- `b <Array>`: Superset.

Returns: `<boolean>`: Returns true if a is a subset of b.
Test whether a set is a subset of another set.

`setMultiplicity(e, multiset)`

Parameters:

- `e <*>`: Element to count.
- `multiset <Array>`: The multiset.

Returns: `<number>`: The multiplicity of e.
Count the multiplicity of an element in a multiset.

`setPowerset(set)`

Parameters:

- `set <Array>`: The input set.

Returns: `<Array>`: The power set.
Compute the power set of a set.

`setSize(set)`

Parameters:

- `set <Array>`: The input set.

Returns: `<number>`: The number of elements.
Get the size of a set.

`setSymDifference(a, b)`

Parameters:

- `a <Array>`: First set.
- `b <Array>`: Second set.

Returns: `<Array>`: The symmetric difference.
Compute the symmetric difference of two sets.

`setUnion(a, b, [others])`

Parameters:

- `a <Array>`: First set.
- `b <Array>`: Second set.
- `others <Array>`: Additional sets.

Returns: `<Array>`: The union of the sets.
Compute the union of two or more sets.

`sign(x)`

Parameters:

- `x <number>`: Input value.

Returns: `<number>`: The sign of x.
Compute the sign of a number.

`simplify(expr, [rules], [scope])`

Parameters:

- `expr <string>`: The expression to simplify.
- `rules <Object>`: Simplification rules.
- `scope <Object>`: Scope for variables.

Returns: `<Node>`: The simplified expression.
Simplify an expression.

`simplifyConstant(expr, [options])`

Parameters:

- `expr <Node>`: The expression to simplify.
- `options <Object>`: Simplification options.

Returns: `<Node>`: The simplified expression.
Simplify an expression containing constants.

`simplifyCore(expr)`

Parameters:

- `expr <Node>`: The expression to simplify.

Returns: `<Node>`: The simplified expression.
Perform core simplifications on an expression.

`sin(x)`

Parameters:

- `x <number>`: Function input.

Returns: `<number>`: The sine of x.
Calculate the sine of a value.

`sinh(x)`

Parameters:

- `x <number>`: Function input.

Returns: `<number>`: The hyperbolic sine of x.
Calculate the hyperbolic sine of a value.

`slu(A, order, threshold)`

Parameters:

- `A <Matrix>`: A sparse matrix.
- `order <number>`: Ordering and analysis control parameter.
- `threshold <number>`: Partial pivoting threshold.

Returns: `<Object>`: The LU decomposition.
Sparse Linear Solver using LU decomposition.

```
smaller(x, y)
```

Parameters:

- `x <number>`: First value.
- `y <number>`: Second value.

Returns: `<boolean>`: Returns true if x ¡ y.
Test whether value x is smaller than y.

```
smallerEq(x, y)
```

Parameters:

- `x <number>`: First value.
- `y <number>`: Second value.

Returns: `<boolean>`: Returns true if x ¡= y.
Test whether value x is smaller than or equal to y.

```
Spa(A)
```

Parameters:

- `A <Matrix>`: A sparse matrix.

Returns: `<Object>`: The LU decomposition.
Compute the Sparse LU decomposition of a matrix.

```
sparse([data], [datatype])
```

Parameters:

- `data <Array>`: The data for the matrix.
- `datatype <string>`: The data type.

Returns: `<SparseMatrix>`: A new SparseMatrix instance.
Create a sparse matrix.

`SparseMatrix([data])`

Parameters:

- `data <Object>`: The data for the matrix.

Returns: `<SparseMatrix>`: A new SparseMatrix instance.
Sparse matrix constructor.

`speedOfLight`

Type: `<Unknown>`
Speed of light in vacuum, in m/s.

`splitUnit(unit, parts)`

Parameters:

- `unit <Unit>`: The unit to split.
- `parts <Array>`: Array of units to split into.

Returns: `<Array>`: Array of units.
Split a unit into its numeric value and unit string.

`sqrt(x)`

Parameters:

- `x <number>`: Value for which to calculate the square root.

Returns: `<number>`: The square root of x.
Calculate the square root of a value.

`sqrtm(x)`

Parameters:

- `x <Array>`: A square matrix.

Returns: `<Matrix>`: The principal square root of x.
Calculate the principal square root of a matrix.

`square(x)`

Parameters:

- `x <number>`: Input value.

Returns: `<number>`: The square of x.
Compute the square of a value.

`squeeze(x)`

Parameters:

- `x <Array>`: The input array or matrix.

Returns: `<Array>`: The squeezed array or matrix.
Remove singleton dimensions from an array or matrix.

`std(array, [normalization], [dim])`

Parameters:

- `array <Array>`: Input array.
- `normalization <string>`: Normalization mode.
- `dim <number>`: Dimension along which to compute.

Returns: `<number>`: The standard deviation.
Compute the standard deviation of a set of values.

`stefanBoltzmann`

Type: `<Unknown>`
Stefan-Boltzmann constant, in W/(mˆ2Kˆ4).

`stirlingS2(n, k)`

Parameters:

- `n <number>`: Total number of objects.
- `k <number>`: Number of non-empty subsets.

Returns: `<number>`: The Stirling number.
Stirling numbers of the second kind.

`string(value)`

Parameters:

- `value <*>`: The value to convert.

Returns: `<string>`: The string representation of value.
Parse a value into a string.

```
subset(x, index, [replacement], [defaultValue])
```

Parameters:

- `x <Array>`: The input matrix.
- `index <Index>`: The index.
- `replacement <*>`: The replacement value.
- `defaultValue <boolean>`: Default value for missing entries.

Returns: `<*>`: The subset or updated matrix.
Get or set a subset of a matrix or array.

```
sylvester(A, B, C)
```

Parameters:

- `A <Matrix>`: A square matrix.
- `B <Matrix>`: A square matrix.
- `C <Matrix>`: A matrix.

Returns: `<Matrix>`: Solution X of the Sylvester equation.
Solve the Sylvester equation A*X + X*B = C.

```
SymbolNode(name)
```

Parameters:

- `name <string>`: The symbol name.

Returns: `<SymbolNode>`: A new SymbolNode instance.
A node representing a symbol in the expression tree.

```
symbolicEqual(expr1, expr2, [options])
```

Parameters:

- `expr1 <Node>`: First expression.

- `expr2 <Node>`: Second expression.
- `options <Object>`: Comparison options.

Returns: `<boolean>`: Returns true if expressions are symbolically equal.
Test whether two expressions are symbolically equal.

`tan(x)`

Parameters:

- `x <number>`: Function input.

Returns: `<number>`: The tangent of x.
Calculate the tangent of a value.

`tanh(x)`

Parameters:

- `x <number>`: Function input.

Returns: `<number>`: The hyperbolic tangent of x.
Calculate the hyperbolic tangent of a value.

`tau`

Type: `<Unknown>`
The constant tau, equal to 2pi.

`thomsonCrossSection`

Type: `<Unknown>`
Thomson cross section, in m^2.

`to(x, unit)`

Parameters:

- `x <Unit>`: The unit to be converted.
- `unit <Unit>`: Target unit.

Returns: `<Unit>`: The converted unit.
Convert a unit to another unit.

`true`

Type: `<Unknown>`
Boolean value true.

`typeOf(x)`

Parameters:

- `x <*>`: The variable.

Returns: `<string>`: The type of x.
Get the type of a variable.

`typed(name, signatures)`

Parameters:

- `name <string>`: Function name.
- `signatures <Object>`: Function signatures.

Returns: `<function>`: The typed function.
Create a typed-function.

`unaryMinus(x)`

Parameters:

- `x <number>`: Input value.

Returns: `<number>`: The negated value.
Invert the sign of a value.

`unaryPlus(x)`

Parameters:

- `x <number>`: Input value.

Returns: `<number>`: The input value.
Unary plus operation.

`unequal(x, y)`

Parameters:

- `x <*>`: First value.

- `y <*>`: Second value.

Returns: `<boolean>`: Returns true if x is not equal to y.
Test whether two values are unequal.

`Unit(value, unit)`

Parameters:

- `value <number>`: The numeric value.
- `unit <string>`: The unit string or Unit object.

Returns: `<Unit>`: A new Unit instance.
Unit constructor.

`unit(value, unit)`

Parameters:

- `value <number>`: The numeric value.
- `unit <string>`: The unit string or Unit object.

Returns: `<Unit>`: A new Unit instance.
Create a unit.

`E`

Type: `<Unknown>`
The base of natural logarithms, e.

`PI`

Type: `<Unknown>`
The mathematical constant pi.

`usolve(U, b)`

Parameters:

- `U <Matrix>`: An upper triangular matrix.
- `b <Matrix>`: A column vector.

Returns: `<Matrix>`: The solution vector x.
Solve a linear system A * x = b where A is an upper triangular matrix.

`usolveAll(U, b)`

Parameters:

- `U <Matrix>`: An upper triangular matrix.
- `b <Matrix>`: A column vector.

Returns: `<Array>`: An array of solutions.
Find all solutions of a linear system A * x = b where A is an upper triangular matrix.

`vacuumImpedance`

Type: `<Unknown>`
Vacuum impedance, in ohms.

`variance(array, [normalization], [dim])`

Parameters:

- `array <Array>`: Input array.
- `normalization <string>`: Normalization mode.
- `dim <number>`: Dimension along which to compute.

Returns: `<number>`: The variance.
Compute the variance of a set of values.

`weakMixingAngle`

Type: `<Unknown>`
Weak mixing angle.

`wienDisplacement`

Type: `<Unknown>`
Wien displacement constant, in m·K.

`xgcd(a, b)`

Parameters:

- `a <number>`: An integer.
- `b <number>`: An integer.

Returns: `<Array>`: An array [gcd, x, y] satisfying gcd = a*x + b*y.
Extended greatest common divisor for integers.

```
xor(x, y)
```

Parameters:

- `x <boolean>`: First value.
- `y <boolean>`: Second value.

Returns: `<boolean>`: The logical XOR of x and y.
Logical XOR operation.

```
ArgumentsError(fn, count, min, [max])
```

Parameters:

- `fn <string>`: Function name.
- `count <number>`: Actual argument count.
- `min <number>`: Minimum required arguments.
- `max <number>`: Maximum allowed arguments.

Returns: `<ArgumentsError>`: A new ArgumentsError instance.
Error thrown when an incorrect number of arguments is passed.

```
DimensionError(actual, expected, [relation])
```

Parameters:

- `actual <number>`: Actual dimension.
- `expected <number>`: Expected dimension.
- `relation <string>`: Relation between dimensions.

Returns: `<DimensionError>`: A new DimensionError instance.
Error thrown when matrix dimensions mismatch.

```
IndexError(index, min, max)
```

Parameters:

- `index <number>`: The invalid index.
- `min <number>`: Minimum allowed index.
- `max <number>`: Maximum allowed index.

Returns: `<IndexError>`: A new IndexError instance.
Error thrown when an index is out of range.

10.39 rcmiga

`run()`

Executes the optimization process.

`checkInputs()`

Validates the input parameters and options.

`creationFcn()`

Creates the initial population.

`selectionFcn()`

Selects individuals from the population.

`crossoverFcn()`

Performs crossover between selected parents.

`mutationFcn()`

Mutates individuals in the population.

`createPlotFcn()`

Initializes the plotting function.

`plotFcn()`

Updates the graphical plot with current optimization status.

`displayPlot()`

Renders the optimization plot.

`buttonCallback()`

Handles button events to stop the optimization process.

`outputFcn()`

Manages the output display based on configuration.

`displayOutput()`

Displays the current state of optimization.

`creationMixedUniform()`

Creates the initial population with mixed uniform distribution.

`binaryTournamentSelection()`

Performs binary tournament selection of parents.

`laplaceMixedCrossover()`

Executes Laplace mixed crossover to generate offspring.

`powerMixedMutation()`

Applies power mixed mutation to selected individuals.

`gaussianMutation()`

Applies Gaussian mutation to selected individuals.

`integerRestriction()`

Ensures integer constraints are met for specific variables.

`checkBounds()`

Checks and enforces variable bounds within the population.

`stoppingCriteria()`

Returns: `<number>`: Code indicating the reason to stop or continue.
Evaluates whether stopping criteria have been met.

`initPopulation()`

Initializes the population for the optimization process.

`initState()`

Initializes the internal state of the algorithm.

`updateState()`

Updates the current state of the algorithm based on evaluations.

`updatePenalty()`

Updates penalty values based on constraint violations.

`evalFitnessFcn()`

Evaluates the fitness function for the current population.

`evalConstraintsFcn()`

Evaluates constraint functions for the current population.

`normConstraintsFcn()`

Normalizes constraint values to maintain consistent scaling.

10.40 space_search

`dispElementSize(bounds, subdivisionPerDepth)`

Parameters:

- `bounds <Array>`: Array of [min, max] for each dimension.
- `subdivisionPerDepth <Array>`: Subdivision factors per depth and dimension.

Displays the size of elements based on bounds and subdivisions.

```
splitSearchSpace(x_lim, k, N_proc)
```

Parameters:

- `x_lim <Array.<Array.<number>>>`: The limits of the search space, where each sub-array represents [start, end] for a dimension.
- `k <Array.<Array.<number>>>`: The parameters to optimize.
- `N_proc <number>`: The number of processors to divide the work among.

Returns: `<Array.<Array.<number>>>`: An array containing the split search spaces and the adjusted parameters.
Splits the search space into smaller intervals for parallel processing.

```
runParallel(x_lim, k, context, setup, fun)
```

Parameters:

- `x_lim <Array.<Array.<number>>>`: The limits of the search space, where each sub-array represents [start, end] for a dimension.
- `k <Array.<Array.<number>>>`: The parameters to optimize.
- `context <Object>`: The execution context containing necessary configurations and states.
- `setup <function>`: The setup function to initialize the parallel environment.
- `fun <function>`: The function to execute in parallel on each split of the search space.

Returns: `<Promise.<Array.<Array.<number>>>>`: A promise that resolves to arrays of input and output results from the parallel execution.
Executes a provided function in parallel across the split search spaces.

```
run(bounds, subdivisionPerDepth, conditionFunction)
```

Parameters:

- `bounds <Array.<Array.<number>>>`: Array of [min, max] for each dimension.

- `subdivisionPerDepth <Array.<Array.<number>>>`: Subdivision factors for each depth and dimension.
- `conditionFunction <function>`: Function that determines if a point satisfies the condition.

Returns: `<Array.<Array.<number>>>`: - Arrays of points inside and outside the condition.
Executes the space search algorithm.

```
makeNodesND(startCoordinates, boundaryValues, currentDepth,
stepSizesPerDepth, subdivisionPerDepth, conditionFunction)
```

Parameters:

- `startCoordinates <Array.<number>>`: Starting coordinates for the current grid.
- `boundaryValues <Array.<number>>`: Values at the boundary points of the current hypercube.
- `currentDepth <number>`: Current depth of the recursion.
- `stepSizesPerDepth <Array.<Array.<number>>>`: Step sizes for each depth.
- `subdivisionPerDepth <Array.<Array.<number>>>`: Subdivision factors for each depth.
- `conditionFunction <function>`: User-defined condition function.

Returns: `<Array.<Array.<number>>>`: - Arrays of points inside and outside the condition.
Recursively creates nodes in N dimensions based on subdivisions.

```
generateCornerShifts(numDimensions)
```

Parameters:

- `numDimensions <number>`: Number of dimensions.

Returns: `<Array.<Array.<number>>>`: - Array of shifts for each corner.
Generates all corner shifts for a hypercube in N dimensions.

```
getCornerIndices(gridShape)
```

Parameters:

- `gridShape <Array.<number>>`: Shape of the grid.

Returns: `<Array.<Array.<number>>>`: - List of corner indices.
Retrieves the indices of corner points in the grid.

`generateIndicesList(gridShape, nf)`

Parameters:

- `gridShape <Array.<number>>`: Shape of the grid.
- `nf <Array>`: N-dimensional array indicating nodes to compute.

Returns: `<Array.<Array.<number>>>`: - List of node indices where nf is 1.
Generates a list of indices for nodes that need to be calculated.

`generateInnerCubeIndicesList(gridShape)`

Parameters:

- `gridShape <Array.<number>>`: Shape of the grid.

Returns: `<Array.<Array.<number>>>`: - List of inner cube starting indices.
Generates a list of starting indices for inner hypercubes.

`getCubeCorners(N, indices)`

Parameters:

- `N <Array>`: N-dimensional grid.
- `indices <Array.<number>>`: Starting indices of the hypercube.

Returns: `<Array>`: - Array of corner coordinates and values.
Retrieves the corner coordinates and values of a hypercube.

10.41 map

`createWindow()`

Returns: `<Promise.<void>>`:
Opens a window with Leaflet and initializes the map.

`setCenter(lat, lon)`

Parameters:

- `lat <number>`: Latitude.

- `lon <number>`: Longitude.

Sets the center of the map to the specified latitude and longitude. This will update the internal state and move the Leaflet map if it is ready.

`setZoom(zoom)`

Parameters:

- `zoom <number>`: The zoom level.

Sets the zoom level of the map. This will update the internal state and adjust the Leaflet map if it is ready.

`addMarker(lat, lon)`

Parameters:

- `lat <number>`: Latitude.
- `lon <number>`: Longitude.

Returns: `<PRDC_JSLAB_GEOGRAPHY_MAP_MARKER>`: - The marker instance or null if map is not ready.
Adds a marker to the map at the specified latitude and longitude.

10.42 map_3d

`createWindow()`

Returns: `<Promise.<void>>`:
Opens a window with Cesium and initializes 3D map.

`setView(lat, lon, height, [heading], [pitch])`

Parameters:

- `lat <number>`: Latitude.
- `lon <number>`: Longitude.
- `height <number>`: Height in meters.
- `heading <number>`: Heading in degrees.
- `pitch <number>`: Pitch in degrees.

Sets the camera view to the specified latitude, longitude, and height.

`addEntity(data)`

Parameters:

- `data <Object>`: The data representing the entity to add.

Returns: `<PRDC_JSLAB_GEOGRAPHY_MAP_3D_ENTITY>`: The newly created map entity.
Adds a new entity to the 3D map.

`removeAllEntities()`

Removes all entities from the 3D map viewer.

`flyTo(entity)`

Parameters:

- `entity <Object>`: The entity to fly to.

Animates the camera to fly to the specified entity.

10.43 Gamepad

`setOnData(callback)`

Parameters:

- `callback <function>`: Function to execute when data is received.

Sets the callback function to handle incoming gamepad data.

`setOnConnect(callback)`

Parameters:

- `callback <function>`: Function to execute on connection.

Sets the callback function for gamepad connection events.

`setOnDisconnect(callback)`

Parameters:

- `callback <function>`: Function to execute on disconnection.

Sets the callback function for gamepad disconnection events.

`close()`

Cleans up the gamepad instance and stops data reading.

10.44 parallel

`getProcessorsNum()`

Returns: `<number>`: Number of processors.
Retrieves the number of logical processors available.

`workerFunction([context], work_function_str, [setup_function_str])`

Parameters:

- `context <Object>`: Optional context to pass to the work_function.
- `work_function_str <function>`: The work function to execute.
- `setup_function_str <function>`: Optional setup function to execute on init.

Generates the worker's internal script.

`init(num_workers, [context], [setup_function_str])`

Parameters:

- `num_workers <number>`: Number of workers to initialize.
- `context <Object>`: Optional context to pass to the work_function.
- `setup_function_str <function>`: Optional setup function to execute on init.

Initializes the worker pool with the specified number of workers.

`assignTasksToWorkers()`

Assigns tasks from the queue to available workers.

`run(context, [setup_function], args, work_function, reset_workers)`

Parameters:

- `context <Object>`: Context variables to assign in the worker.
- `setup_function <function>`: Optional setup function to execute on init.
- `args <Array>`: Arguments to pass to the work_function.
- `work_function <function>`: The work function to execute.
- `reset_workers <boolean>`: Wether to reset all workers or not.

Returns: `<Promise>`: - Resolves with the result of the work_function.
Enqueues a task to be executed by the worker pool.

```
parfor(start, end, [step], [num_workers], [context], [setup_function],
work_function, reset_workers)
```

Parameters:

- `start <number>`: The initial value of the loop counter.
- `end <number>`: The terminating value of the loop counter.
- `step <number>`: The amount by which to increment the loop counter each iteration.
- `num_workers <number>`: The number of workers to use.
- `context <Object>`: Optional context to pass to the work_function.
- `setup_function <function>`: Optional setup function to execute before work_function.
- `work_function <function>`: The function to execute on each iteration.
- `reset_workers <boolean>`: Wether to reset all workers or not.

Returns: `<Promise.<Array>>`: - A promise that resolves to an array of results.
Executes a parallel for loop by dividing the iteration range among workers.

```
terminate()
```

Terminates all workers and resets the worker pool.

10.45 mat

```
new(A, rows, cols)
```

Parameters:

- `A <Array>`: The matrix data.

- `rows <number>`: Number of rows.
- `cols <number>`: Number of columns.

Returns: `<PRDC_JSLAB_MATRIX>`: A new matrix instance.
Creates a new matrix.

```
fill(v, rows, [cols])
```

Parameters:

- `v <number>`: The value to fill the matrix with.
- `rows <number>`: Number of rows.
- `cols <number>`: Number of columns.

Returns: `<PRDC_JSLAB_MATRIX>`: The filled matrix.
Creates a matrix filled with a specific value.

```
NaNs(rows, [cols])
```

Parameters:

- `rows <number>`: Number of rows.
- `cols <number>`: Number of columns.

Returns: `<PRDC_JSLAB_MATRIX>`: The NaN-filled matrix.
Creates a matrix filled with NaN values.

```
ones(rows, [cols])
```

Parameters:

- `rows <number>`: Number of rows.
- `cols <number>`: Number of columns.

Returns: `<PRDC_JSLAB_MATRIX>`: The ones-filled matrix.
Creates a matrix filled with ones.

```
zeros(rows, [cols])
```

Parameters:

- `rows <number>`: Number of rows.
- `cols <number>`: Number of columns.

Returns: `<PRDC_JSLAB_MATRIX>`: The zeros-filled matrix.
Creates a matrix filled with zeros.

`diag(A)`

Parameters:

- `A <Array>`: The array to create the diagonal matrix from.

Returns: `<PRDC_JSLAB_MATRIX>`: The diagonal matrix.
Creates a diagonal matrix from an array.

`eye(size)`

Parameters:

- `size <number>`: The size of the identity matrix.

Returns: `<PRDC_JSLAB_MATRIX>`: The identity matrix.
Creates an identity matrix of a given size.

`concatRow(args)`

Parameters:

- `args <PRDC_JSLAB_MATRIX>`: Matrices to concatenate.

Returns: `<PRDC_JSLAB_MATRIX>`: The concatenated matrix.
Concatenates multiple matrices vertically (row-wise).

`concatCol(args)`

Parameters:

- `args <PRDC_JSLAB_MATRIX>`: Matrices to concatenate.

Returns: `<PRDC_JSLAB_MATRIX>`: The concatenated matrix.
Concatenates multiple matrices horizontally (column-wise).

`isMatrix(A)`

Parameters:

- `A <Object>`: The object to check.

Returns: `<boolean>`: True if A is a matrix, else false.
Checks if the provided object is a matrix.

10.46 vec

`new(x, y, z)`

Parameters:

- `x <number>`: The x-component of the vector.
- `y <number>`: The y-component of the vector.
- `z <number>`: The z-component of the vector.

Returns: `<PRDC_JSLAB_VECTOR>`: A new vector instance.
Creates a new vector with specified x, y, z components.

`polar(length, radian)`

Parameters:

- `length <number>`: The length of the vector.
- `radian <number>`: The angle in radians.

Returns: `<PRDC_JSLAB_VECTOR>`: The resulting vector.
Creates a vector from polar coordinates.

`spherical(length, azimuth, elevation)`

Parameters:

- `length <number>`: The length (magnitude) of the vector.
- `azimuth <number>`: The azimuth angle in radians (angle from the X-axis in the XY plane).
- `elevation <number>`: The elevation angle in radians (angle from the XY plane towards Z).

Returns: `<PRDC_JSLAB_VECTOR>`: The resulting vector.
Creates a vector from spherical coordinates.

10.47 sym

`load()`

Returns: `<Promise.<void>>`: A promise that resolves when the libraries are loaded.

Loads the symbolic math libraries (SymPy and NumPy) using Pyodide. Initializes the Python environment for symbolic computations.

`getSymbolName(symbol)`

Parameters:

- `symbol <PRDC_JSLAB_SYMBOLIC_MATH_SYMBOL>`: The symbolic variable or its name.

Returns: `<string>`: The name of the symbolic variable.
Retrieves the name of a symbolic variable.

`checkLoaded()`

Checks if the symbolic libraries are loaded. Throws an error if not.

`eval(code)`

Parameters:

- `code <string>`: The Python code to evaluate.

Returns: `<*>`: The result of the evaluated code.
Evaluates a Python code string within the symbolic math environment.

`sym(name)`

Parameters:

- `name <string>`: The name of the symbolic variable.

Returns: `<PRDC_JSLAB_SYMBOLIC_MATH_SYMBOL>`: The created symbolic variable.
Creates a single symbolic variable.

`syms(names)`

Parameters:

- `names <Array.<string>>`: An array of names for the symbolic variables.

Returns: `<Array.<PRDC_JSLAB_SYMBOLIC_MATH_SYMBOL>>`: An array of created symbolic variables.
Creates multiple symbolic variables.

`mat(expr)`

Parameters:

- `expr` `<Array.<Array.<(PRDC_JSLAB_SYMBOLIC_MATH_SYMBOL|string|number)>>>`: The nested array representing the matrix.

Returns: `<PRDC_JSLAB_SYMBOLIC_MATH_SYMBOL>`: The symbolic matrix.
Creates a symbolic matrix from a nested array expression.

`mul(args)`

Parameters:

- `args` `<PRDC_JSLAB_SYMBOLIC_MATH_SYMBOL>`: The symbolic expressions to multiply.

Returns: `<PRDC_JSLAB_SYMBOLIC_MATH_SYMBOL>`: The resulting symbolic expression after multiplication.
Multiplies multiple symbolic expressions.

`div(args)`

Parameters:

- `args` `<PRDC_JSLAB_SYMBOLIC_MATH_SYMBOL>`: The symbolic expressions to divide.

Returns: `<PRDC_JSLAB_SYMBOLIC_MATH_SYMBOL>`: The resulting symbolic expression after division.
Divides multiple symbolic expressions.

`plus(args)`

Parameters:

- `args` `<PRDC_JSLAB_SYMBOLIC_MATH_SYMBOL>`: The symbolic expressions to add.

Returns: `<PRDC_JSLAB_SYMBOLIC_MATH_SYMBOL>`: The resulting symbolic expression after addition.
Adds multiple symbolic expressions.

`minus(args)`

Parameters:

- `args` `<PRDC_JSLAB_SYMBOLIC_MATH_SYMBOL>`: The symbolic expressions to subtract.

Returns: `<PRDC_JSLAB_SYMBOLIC_MATH_SYMBOL>`: The resulting symbolic expression after subtraction.
Subtracts multiple symbolic expressions.

`pow(expr, n)`

Parameters:

- `expr <PRDC_JSLAB_SYMBOLIC_MATH_SYMBOL>`: The base expression.
- `n <PRDC_JSLAB_SYMBOLIC_MATH_SYMBOL>`: The exponent.

Returns: `<PRDC_JSLAB_SYMBOLIC_MATH_SYMBOL>`: The resulting symbolic expression after exponentiation.
Raises a symbolic expression to a power.

```
transp(expr)
```

Parameters:

- `expr <PRDC_JSLAB_SYMBOLIC_MATH_SYMBOL>`: The matrix expression to transpose.

Returns: `<PRDC_JSLAB_SYMBOLIC_MATH_SYMBOL>`: The transposed matrix expression.
Transposes a symbolic matrix expression.

```
inv(expr)
```

Parameters:

- `expr <PRDC_JSLAB_SYMBOLIC_MATH_SYMBOL>`: The matrix expression to invert.

Returns: `<PRDC_JSLAB_SYMBOLIC_MATH_SYMBOL>`: The inverse of the matrix expression.
Computes the inverse of a symbolic matrix expression.

```
det(expr)
```

Parameters:

- `expr <PRDC_JSLAB_SYMBOLIC_MATH_SYMBOL>`: The matrix expression whose determinant is to be computed.

Returns: `<PRDC_JSLAB_SYMBOLIC_MATH_SYMBOL>`: The determinant of the matrix expression.
Computes the determinant of a symbolic matrix expression.

```
diff(expr, x, [n])
```

Parameters:

- `expr <PRDC_JSLAB_SYMBOLIC_MATH_SYMBOL>`: The expression to differentiate.
- `x <PRDC_JSLAB_SYMBOLIC_MATH_SYMBOL>`: The variable with respect to which differentiation is performed.
- `n <number>`: The order of differentiation.

Returns: <PRDC_JSLAB_SYMBOLIC_MATH_SYMBOL>: The differentiated expression.
Differentiates a symbolic expression with respect to a variable.

```
intg(expr, x, lims)
```

Parameters:

- expr <PRDC_JSLAB_SYMBOLIC_MATH_SYMBOL>: The expression to integrate.
- x <PRDC_JSLAB_SYMBOLIC_MATH_SYMBOL>: The variable with respect to which integration is performed.
- lims <Array.<(PRDC_JSLAB_SYMBOLIC_MATH_SYMBOL|string)>>: The limits of integration as [lower, upper]. If undefined, indefinite integration is performed.

Returns: <PRDC_JSLAB_SYMBOLIC_MATH_SYMBOL>: The integrated expression.
Integrates a symbolic expression with respect to a variable.

```
subs(expr, x, val)
```

Parameters:

- expr <PRDC_JSLAB_SYMBOLIC_MATH_SYMBOL>: The expression in which to substitute.
- x <PRDC_JSLAB_SYMBOLIC_MATH_SYMBOL>: The variable to substitute.
- val <PRDC_JSLAB_SYMBOLIC_MATH_SYMBOL>: The value or array of values to substitute.

Returns: <PRDC_JSLAB_SYMBOLIC_MATH_SYMBOL>: The expression after substitution.
Substitutes a value into a symbolic expression.

```
simplify(expr)
```

Parameters:

- expr <PRDC_JSLAB_SYMBOLIC_MATH_SYMBOL>: The expression to simplify.

Returns: <PRDC_JSLAB_SYMBOLIC_MATH_SYMBOL>: The simplified expression.
Simplifies a symbolic expression.

```
showLatex(expr)
```

Parameters:

- expr <PRDC_JSLAB_SYMBOLIC_MATH_SYMBOL>: The expression to display in LaTeX format.

Displays the LaTeX representation of a symbolic expression.

```
dispLatex(expr)
```

Parameters:

- `expr <PRDC_JSLAB_SYMBOLIC_MATH_SYMBOL>`: The expression to convert to a LaTeX string.

Displays the LaTeX string of a symbolic expression.

```
clear()
```

Clears all symbolic variables and resets the symbolic math environment.