Description of the RustedSciThe library (v. 0.2.12), the software free to download and use under the MIT license.

Section IV.

(work on the manual is ongoing, there may be errors and omissions!)

Boundary value problems.

Scientific background of BVP solver, developer's and user's manuals.

1. Scientific background.

1.1 General remarks.

The solution of problems with boundary conditions for the purposes of modelling combustion, chemical reactors and related problems is implemented in a number of packages, both commercial (CHEMKIN™) and open source (CANTERA). Such problems are characterised by the rigidity of differential equations, their large number (since the number of possible reactants and products can be very large), which led to the need to develop quite specific algorithms for their solution, which will be discussed below.

1.2 Modified damped Newton method (damped Newton).

1.3 Algorithm for keeping Newtonian step within physically reasonable bounds.

Both CHEMKIN™ and CANTERA implement an algorithm for keeping the Newtonian pitch within physically reasonable bounds. Let us explain this idea. [5, page 261] We know, for example, that temperature and mass flow rate must be positive, and that mass fractions of substances must be between zero and one. The concentrations of many substances, such as fuels away from the flame, are close to zero and often and frequently threaten to push the solution out of bounds. The damping parameter (dumping factor) is initially chosen to be as large as possible, but so as not to violate the various constraints imposed on the solution variables.

In the newton solver [3] of the CANTERA (C++) package, this algorithm is implemented as the bound\_step function, see solver source code line 38. In the freely distributed twopnt (fortran) solver [2], based on the classical fortran solver, this algorithm is implemented as a newton\_damping subroutine, see the source code of the solver, line 3207.

As can be seen, the implementation of this algorithm requires the specification of bounds on physical quantities. The package also RST implements an algorithm for keeping Newtonian step within physically reasonable bounds.

1.4 Adaptive meshes.

The authors [5, 8] found that starting the computation on a coarse grid has several important advantages. One is that Newtonian iterations are more likely to converge on a coarse grid than on a fine grid. Also, on a coarse grid, the number of variables is small and hence the cost per iteration is relatively small. Since iteration starts with a user-specified approximation, ‘guessing’ the solution will probably require many iterations. Ultimately, of course, for the solution to be accurate, it must be obtained on a fine grid. However, as the solution is computed on each successive finer grid, the initial estimates become better because they are obtained from a converging solution on the previous coarse grid. In general, the solution on one grid lies in the convergence region of Newton's method on the next finer grid. Thus, although the cost per iteration increases, the number of iterations required decreases. Adaptive placement of grid points to form finer meshes is done in such a way that the total number of grid points required to accurately represent the solution is minimised.

In [9, p. 138], a rather simple grid recomputation technique is proposed. New grid points are now inserted between any pair of neighbouring grid points - say xi and xi+1 for which | yi - yi+1 | exceeds a predetermined limit δ the number of such grid points inserted (uniformly) between xi and xi+1 is approximately equal to | yi - yi+1 | / δ. The equations are then solved again, new grid points are inserted, and so on; the process continues iteratively until | yi - yi+1 |< δ everywhere. The value of δ is adjusted during the computation process so that it always has a fixed ratio (typically 10-3) to the computed spread value (maxi(yi) - mini(yi)). Since the insertion of new grid points may result in a locally abrupt change in the size of the grid interval with some consequent loss of the accuracy with which the grid equations approximate the differential equations, a smoothing process is performed before each new Gaussian run. This smoothing process consists of replacing each grid point xi with a new grid point x; via xi = 0.5(xi+1 + xi-1).

One further clarification needs to be made. It often happens that if the set of grid points is not dense in the right places, the solution of the grid equations is not a good approximation to the solution of the differential equations, then the procedure of grid point insertion will insert new grid points in inappropriate areas. To avoid this difficulty, the problem is solved stepwise with different values of precision e. First, the problem is solved (using the iterative process described above) for a moderate value of e (e.g., e = 10-1 or 10-2) and then, in turn, for values of e smaller than the previous value of e by a factor of about 3. The set of grid points used at the end of the previous step forms the initial set for the new step. To detect irregularities in the computation, the values of maxi(yi) and mini(yi) are recorded for each value of e encountered in the process.

Thus, the idea of [9] is that each recalculation of the grid densifies the points: they are especially dense in the regions where the difference condition is not fulfilled.

The author [7, p. 85], solving the problem about laminar flames, where the vector of unknowns represents N concentrations of substances and one temperature, improve working techniques [9] suggest introducing a new grid, ensuring that the conditions are met

(4.1)

Difference condition

(4.2)

Condition on the derivative.

where δ and γ are small numbers smaller than unity, and the values of maxima |S\_i | and |(dS\_i)/dx| are obtained from the converged numerical solution on the previous grid.

A potential disadvantage of such a procedure is the generation of a grid that may not be smoothly varying. For example, the ratio of successive grid intervals may differ by several orders of magnitude. This can affect both the convergence and accuracy of the method. As a result, we impose the additional constraint that the grid obtained by using the constraints in Equations (4.1) and (4.2) must be locally bounded, i.e., the ratio of neighbouring grid intervals must be bounded at the top and bottom by constants.

We require that

l/C =< hj/hj-1 =< C, j = 2, 3 ,.... , M, (4.3)

where C is a constant >=l. This grid buffering smooths out rapid changes in the size of grid intervals. In the adaptive grid algorithm of [7], the flame equations are first solved on a coarse grid (4-5 subintervals). Then the maximum values of |S\_i | and |(dS\_i)/dx| are obtained

Next, the inequalities in equations (4.1) and (4.2) were checked for each of the N + 1 components of S at the coarse grid nodes. If none of the inequalities are fulfilled, the grid point is inserted in the middle of the interval under consideration.

After obtaining the new mesh, we check whether it is locally bounded. If not, the grid point is inserted in the middle of the intervals in which the inequality from equation (4.3) is not fulfilled. The previously converged numerical solution is interpolated onto this new grid, and the result obtained serves as an estimate of the initial solution for Newton's method on this linear grid.

The work [6, p. 1777] continues and develops the ideas of [9]. New grid points are incorporated into finer grids in such a way as to minimise the total number of grid points required to accurately represent the solution. In particular, to resolve gradients, we restrict the variation of the solution between grid points

(4.4)

and to eliminate curvature, we restrict the variation in the derivatives of the solution between grid points to points.

A new grid point is placed at the centre in each interval where these inequalities are not satisfied. The parameters δ and γ are on the order of one tenth, and we often use δ = 0.2 and γ = 0.5. The algorithm excludes from consideration very small variables and those whose total variation is very small.

The authors conclude that the method does not seem to be sensitive to the interpolation scheme used. When the meshes are very coarse, the solutions are not smooth and may bear little resemblance both to each other and to the physical solution. Thus, the use of high-order interpolation schemes does not appear to be justified. When the meshes become fine enough and the computed solutions begin to resemble physical flames, the solutions change from one mesh to the next mainly in the variation in the location and thickness of the flame front. Thus, complimentary interpolation simply places a highly accurate copy of the incorrect solution profile on a finer grid. Once the solution is settled, Newton's method converges so quickly on finer grids that we do not need more accurate initial estimates.

The developers of CHEMKIN [5, p. 264] followed mainly the work of [6], their programme uses relations 4.4, 4.5 and excludes variables that are below a certain minimum value during the rearrangement. This avoids unnecessary recalculation for variables that are essentially zero but may show locally high derivatives due to rounding errors. The initial approximation of the vector of unknowns is determined by linear interpolation of the solution on the coarse grid to a new finer grid. Once a convergent solution on the new finer grid is determined, the realignment procedure is performed again.

Several grid realignment algorithms are implemented in the RST package.

1.5 Solving a system of linear equations.

The block-diagonal system obtained by linearising a system of difference equations is usually [5 page 269, 6] solved using algorithms developed for the LINPACK library, see description [10], code [11]. The TWOPNT code [2] contains an implementation of these algorithms in modern fortran. The C++ code is part of the CANTERA package (<https://github.com/Cantera/cantera/blob/main/src/numerics/BandMatrix.cpp>).

It should be noted that the time complexity of general methods for solving systems of linear equations is O(N3), and the time complexity of solving systems with a banded matrix is O(N\*l2), where l is the ‘thickness’ of the diagonal. Thus, the latter give a significant gain in performance when N>>l, because l is a value of the order of the number of unknowns, and N = number of unknowns \* number of grid steps, this condition is fulfilled. Thus, the transition to methods specialised for block-diagonal matrices will give a performance gain of the order of N3/N\*l2 = number of grid steps2 times, i.e. from the order of hundreds to the order of ten thousand times.

The applied method of solution is the Gaussian method with partial pivoting for block-diagonal matrices (Gaussian eliminations with partial pivoting for banded matrix).

The most efficient and stable is the following algorithmic implementation.

We will iterate over column i from 0 to n:

- In column i we find the largest modulo element below row i, i.e. below the main diagonal, in other words, the largest subdiagonal element, however, our matrix is block diagonal and the number of non-zero elements below the main diagonal does not exceed kl, hence it is sufficient to search for the largest element in rows i to i+kl+1. Since the number i+kl+1 obviously cannot exceed the dimension of the matrix N, the expression for the lower bound of the search should be written as

low = min(N, i+kl+1).

he largest element found is called the main or pivot element of *A[piv, i]*, and the line with the number piv containing the main element is called the main line;

- if the leading element is not an element on the main diagonal, in other words if i!=piv and *A[piv, i]! = A[i, i]* swap lines i and *piv*, if i=piv, then A[piv, i] is already on the main diagonal and there is no need to rearrange anything;

- remember the permutation applied in the previous point in the vector of permutations, it will be further used to apply the same permutations to the vector of right parts b;

The algorithm of the Gauss method follows.

- In the variant method for complete matrices, we would select a submatrix As = A[i...i. ] starting with a diagonal element, but in the variant for diagonal matrices, we can select such a submatrix that there is no zero column or row: As = A[i..low i. ], where right = min(N, i + ku + kl + 1). Indeed, kl+ku+1 is the width of the diagonal strip;

- cut out from the submatrix As the first column C (recall that its upper element will lie on the main diagonal), which corresponds to column i of the original matrix As, the remaining submatrix will be called As1.

- the first column C except for the first element is divided by the value of the first element;

- we do the following with the remaining matrix As1: cut out the first row R (pivot row) from it, the remaining matrix will be called As2

- iterate over the columns of As2 for each column k we perform:

At the end of the described procedure we obtain LU matrix, i.e. such a matrix which is the sum of L and U matrices minus the units in the main diagonal of the matrix U, in other words, LU = L + U - E, where E is a unit matrix. This matrix is then used to solve the linear system

literature

1. The Twopnt Program for Boundary Value Problems Version 3.10 of March 1992. Sandia National Laboratories Report SAND91-8230, Livermore, California, April 1992. Reprinted February 1996.

2. <https://github.com/perazz/twopnt?tab=readme-ov-file>

3. MultiNewton.cpp (https://github.com/Cantera/cantera/blob/main/src/oneD/MultiNewton.cpp)

4.  Ascher, U. M., Mattheij, R. M., & Russell, R. D. 1988. Numerical solution of boundary value problems for ordinary differential equations. Society for Industrial and Applied Mathematics.

5. Chemkin Theory Manual 17.0. Chemkin® Software. January 20166. Joseph F. Grcar, Robert R.  Kee, Mitchell D. Smooke and James A. Miller. A hybrid newton/time-integration procedure for the solution of steady, laminar, one-dimensional, premixed flames. // Twenty-first Symposium (International) on Combustion/The Combustion Institute, 1986, pp. 1773-1782

7. Mitchell D. Smooke. Solution of Burner-Stabilized Premixed Laminar Flames by Boundary Value Methods// Journal of Computational Physics Volume 48, Issue 1, October 1982, pp. 72-105

8. M. D. Smooke and R. M. M. Mattheij, “On the solution of nonlinear two point boundary value problems on successively reﬁned grids,” Applied Numerical Mathematics, volume 1,

1985, pages 463–487.

9. Carl E. Pearson. On a differential equation of boundary layer type //Journal of Mathematics and Physics, volume 47, issue 1-4, pp. 134-154

10. J. J. Dongarra, C. B. Moler, J. R. Bunch, and G. W. Stewart, “LINPACK Users' Guide”, Society of Industrial and Applied Mathematics, 1979.

11. <https://www.netlib.org/linpack/>

2. Developer's Guide.

The Boundary Condition Problem Solver implements the following features:

- Exact analytical Jacobian

- Adaptive meshes (several grid recalculation algorithms)

- Damped Newton

- Autoselect between the standard Gaussian method for solving SLAEs and a variant of the method for block-diagonal (banded) Jacobian.

- Choice between dense and sparse matrices;

- Logging of the entire calculating process

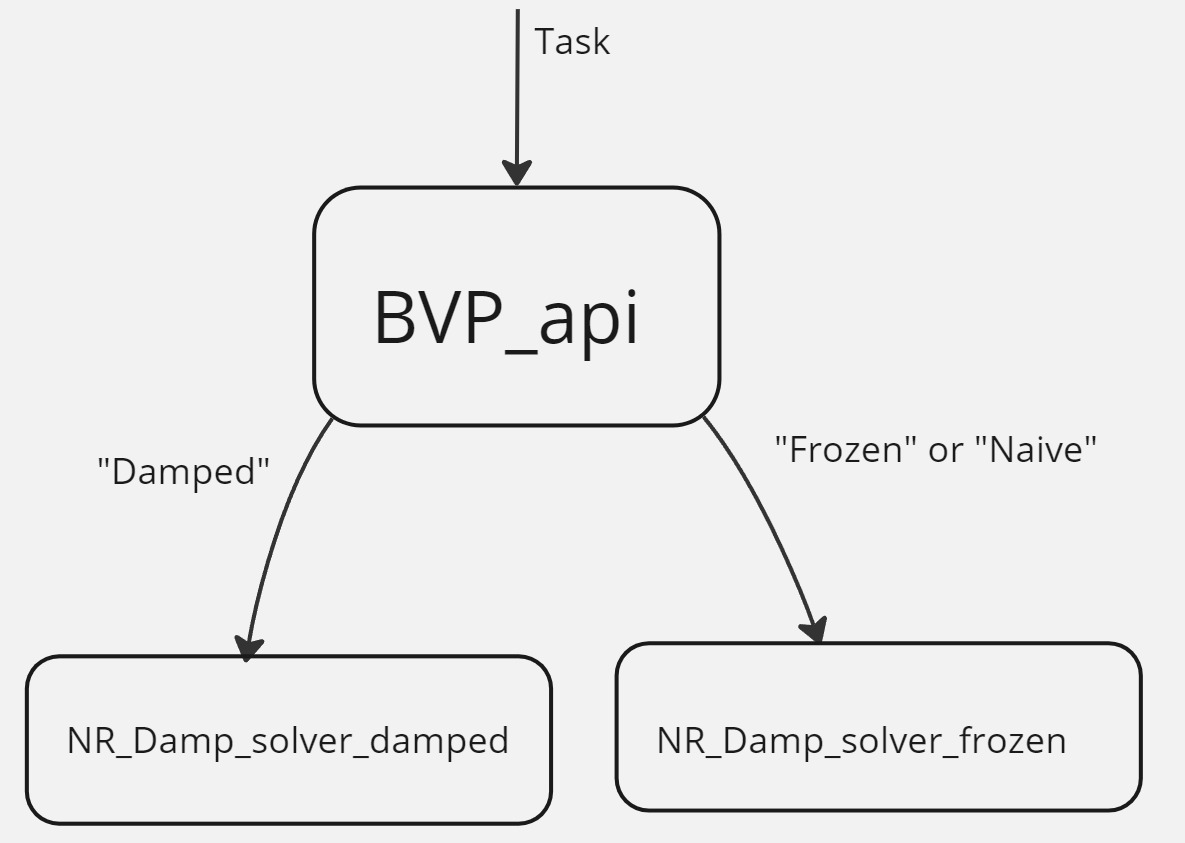
Let us briefly describe the logic of the main algorithms, the structure of the modules, and the main data structures.

The structure containing the calculation task is called BVP and is located in the *numerical::BVP\_api* module. This structure is an intermediary between the user and different variants of Newton's method and it has no functionality except for passing the problem task to modules implementing specific strategies of Newton's method. To initialize the solution, it is necessary to call an instance of this structure and use the new function to pass the problem conditions described in Table 1 to the structure.

Table 1.

|  |  |
| --- | --- |
| Public field (to be filled in when completing the task): field data type | Description. |
| eq\_system: Vec<Expr> | Vector of RHS of the system of equations. Defined in symbolic (analytical) form (NB!). |
| initial\_guess: DMatrix<f64> | Initial approximation to the solution (matrix of dimension equal to the number of unknown variables\*number of grid steps) |
| values: Vec<String> | Vector of unknowns |
| arg: String | Name of argument (x, t, etc.) |
| BorderConditions: HashMap<String, (usize, f64) | Hash table (dictionary) of initial conditions, where the key is the name of the unknown for which the condition is entered, the value is a pair of numbers: the first number 0 or 1 is the initial or boundary condition, respectively, the second number is the value itself. |
| t0: f64 | Initial value of the argument |
| t\_end: f64 | Final argument value |
| n\_steps: usize | Number of discretisation steps (must be consistent with the dimensionality of the initial approximation matrix, see above). If an adaptive grid algorithm is chosen, the specified number of discretization steps will be changed during the solution. |
| scheme: String | Parameter for specifying the numerical scheme:  ‘forward’ - simple forward difference,  ‘trapezoid’ - see e.g. https://en.wikipedia.org/wiki/Trapezoidal\_rule\_(differential\_equations) |
| strategy: String | Newton's method strategy:  ‘Naive’ The Jacobian is recalculated every iteration, there is no dumping, there is no holding of the Newton step within specified bounds, there are no adaptive meshes - the simplest variant of the method;  ‘Frozen’ - Jacobian is recalculated not every iteration, but according to some condition set by parameters in strategy\_params hash table, however, there is no dumping, no Newtonian step retention within given bounds, no adaptive meshes – more advanced variant of the method;  ‘Damped’ - the most advanced variant of the method: dumping, holding Newtonian step in the given bounds, adaptive meshes, Jacobian recalculation by condition. The required control parameters should be specified in the *strategy\_params* hash table |
| strategy\_params: Option<HashMap<String, Option<Vec<f64>>>>, | Solving strategy parameters. Specific for each strategy and will be described separately. |
| linear\_sys\_method: Option<String> | Parameter for specifying the solver for the system of linear equations. |
| method: String | Parameter for specifying the type of specific software implementation of basic linear algebra structures (actually linear algebra library (crate), actually defines the way of matrix data storage).  ‘Dense’ - dense matrices are used;  ‘Sparse’ - sparse (suitable for large and very large problems); |
| max\_iterations: usize | The maximum allowed number of Newton iterations, if the solution is not reached within this number, the program is exited. |
| tolerance: f64 | Absolute accuracy |
| rel\_tolerance: Option<HashMap<String, f64>>, | Relative accuracy. A hash table where the key is the name of a variable and the number is the precision value for that variable. |
| Bounds: Option<HashMap<String, (f64, f64)>> | Permissible physical boundaries of the physical quantity (e.g., obviously the mole fraction is between 0 and 1, and the absolute temperature in combustion physics between the value ‘in the room’ and (conventionally) 5000 K). A hash table, where the key is the name of the unknown variable and the value is a pair of numbers (upper and lower bounds). This information is used in the algorithm for keeping the Newtonian step within the given bounds. |
|  |  |

There are two “branches” in the program with a partially disjoint set of modules, one for the “Damped” method option and one for the other two: “Frozen” and “Naive”. Of course, the existence of two variations of the code contradicts the well-known programming principle of DRY (don’t repeat yourself), however, this organization of the code was chosen deliberately differently when trying to combine all three options, so the complex algorithm would become even more complicated.



The “Damped” branch will be discussed below as the most advanced.

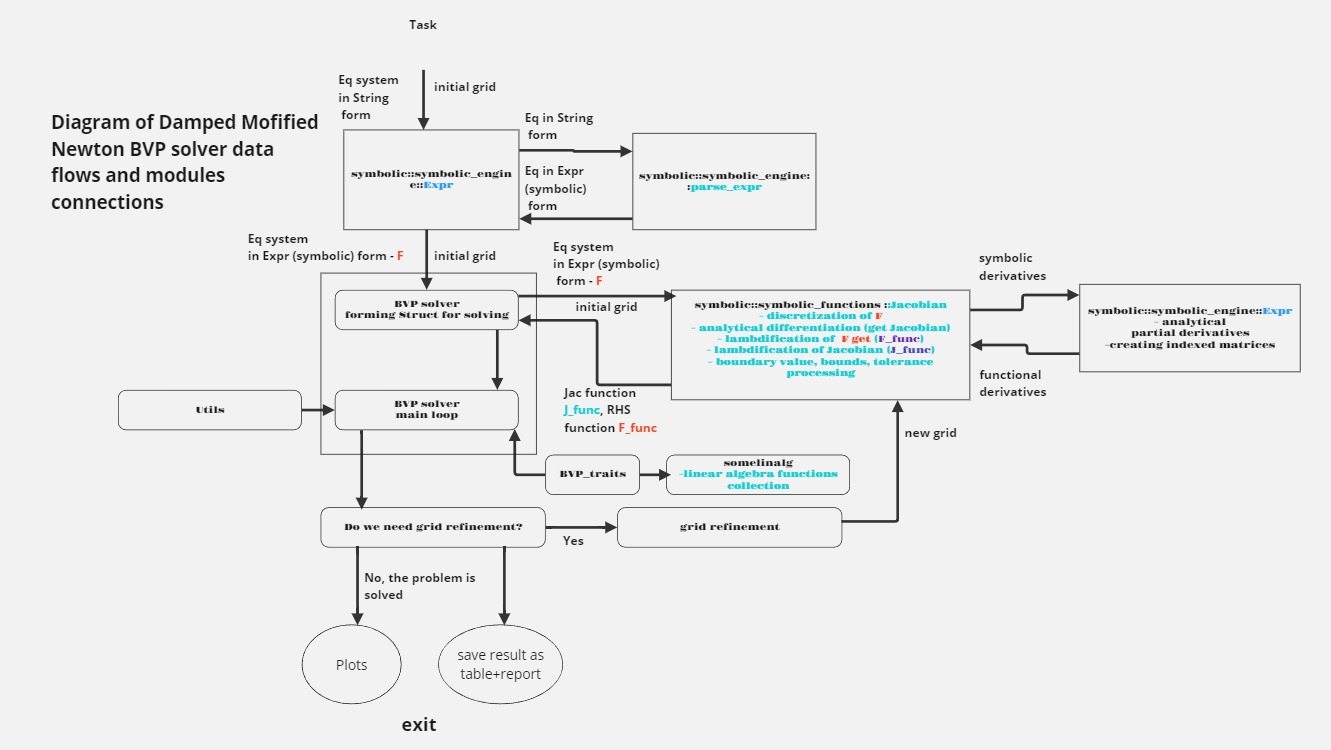
If we have chosen the ‘Damped’ strategy, all these conditions of the problem will be immediately written to the *NRBVP* structure (via its own method *new*), located in the *numerical::BVP\_Damp::NR\_Damp\_solver\_damped* module, which implements the most advanced version of Newton's method available in the RST package. This structure contains in addition to the assignment data of iterations, Jacobian functions and system of equations (residual), etc. data for calculation.

Let's characterize the key-value pairs of the strategy\_params hash table, which is required to be set by the user among other mandatory field

|  |  |  |
| --- | --- | --- |
| Key name | Default value (enabled if user specifies None instead) | Mathematical meaning |
| max\_jac | 3 | Number of iterations without Jacobian recalculation, i.e. with old Jacobian. |
| maxDampIter | 5 | Limit number of iterations of dump factor search |
| DampFacor | 0.5 | The factor to reduce the estimated dump factor if the previous search iteration was not successful |
| adaptive | If the user specified None there will be no recalculation of meshes | Whether or not recalculation of meshes is required. |
| The following pairs characterise the grid recalculation algorithm and are meaningful if the value for adaptive is not None. | | |
| grcar\_smooke | No default values | Algorithm of authors GrCar and Smooke |
| pearson | No default values | Pearson algorithm |

Fig. 1 draws the general scheme of data flow when solving Newton's method using the above strategy.

Fig. 1



Let's describe the data flow, i.e. the transformation of information from the calculation task to the output of results.

Before running the solution, the *NRBVP* structure function, namely *task\_check*, will check if all fields are filled in correctly. If something is not specified or specified incorrectly, the program will generate a diagnostic message and exit.

Next, the *eq\_generate* function will create an instance of the Jacobian structure by accessing the *symbolic::symbolic\_functions* module using the structure function called *new*. The name of the Jacobian structure should not be misleading - its methods are responsible for discretization of the system and for the formation of vector-functions and matrix-functions. Further development of events will take place inside the engine of symbolic calculations, implemented in the modules *symbolic::symbolic\_functions* and *symbolic::symbolic\_engine* (we will return back to the solver *NR\_Damp\_solver\_damped* when we generate all the necessary data structures), so until we return to the solver, naming a function we will mean that it is a method of the Jacobian structure and is located in the module *symbolic::symbolic\_functions*.

Further development of events depends on the value of the method parameter:

- if the value ‘Sparse’ is selected - it means that the *faer* library and its characteristic data structures for matrices, vectors, etc. will be used (we will call them ‘linear algebra primitives’) The function that is called is named *generate\_BVP\_SparseColMat*.

- If the value ‘Dense’ is selected, it means that the *nalgebra* library and its characteristic linear algebra primitives will be used. The function that is called in this case is called

*generate\_BVP*

Both functions essentially do the same thing (but with different, user-selectable linear algebra primitives). Within the functions, a series of the following generations are run.

1) A discretization of the system of equations is performed with the implementation of boundary conditions. In other words, we have a vector of RHS of differential equations, defined as a variable *eq\_system: Vec<Expr>* vector of symbolic expressions, then knowing the grid of argument values, taking into account the discretization scheme (parameter scheme), as well as boundary conditions (*BorderConditions: HashMap<String, (usize, f64)>*) in a double cycle (by grid steps and by equations) the grid (discrete) equations are generated. The described algorithm takes place in the *discretisation\_system\_BVP* function. This function thus returns a discretized system of (still symbolic!) equations. In addition, this function returns a vector of discrete (grid) variables, the number of which is of course determined by the number of unknowns and grid steps. In addition, it generates:

- Vector of limits of acceptable values for each of the grid variables (based on the Bounds assignment parameter);

- A vector of relative accuracies for each of the discrete variables (based on the *rel\_tolerance* assignment parameter).

2) Now that we have a vector of grid equations and grid variables defined we can calculate the symbolic Jacobian, the calculation of the symbolic (analytic) Jacobian is implemented in the function *calc\_jacobian*, and the corresponding low-level procedures (algorithmic implementation of differentiation rules, partial derivatives, simplifications of symbolic expressions, etc.) are implemented in the module of the analytic computing engine *symbolic::symbolic\_engine*.

So now we have the vector of mesh equations and the Jacobi matrix given in symbolic form. That's great!

3) Let's turn the vector of grid equations given in symbolic form into a vector function for further calculations. This is done in the functions *vector\_funvector\_IVP\_Col* and *vector\_funvector\_IVP\_DVector* for the ‘Sparse’ and ‘Dense’ options, respectively

4) Let's convert the Jacobian matrix given in symbolic form into a matrix -function for further calculations. This is done in the functions *jacobian\_generate\_IVP\_SparseColMat* and *jacobian\_generate\_IVP\_DMatrix* for the ‘Sparse’ and ‘Dense’ options respectively.

5) Let's calculate the block diagonal width (bandwidth) of the symbolic matrix and write the widths of non-zero diagonals below the main diagonal (subdiagonal width) and above the main diagonal (superdiagonal) into a tuple (kl, ku)

Exit the *generate\_BVP* function.

So now we have everything we need:

- The vector-function of the mesh equations;

- The matrix-function of the Jacobian;

- Vector of grid variables;

- A vector of admissible values for each of the grid variables;

- Vector of relative accuracies for each of the discrete variables;

- Bandwidth of non-zero elements

Then we can return to the main module of Newton's method *numerical::BVP\_Damp::NR\_Damp\_solver\_damped* and write the generated functions and vectors into the main NRBVP structure.

These will be located in the appropriate fields. See Table 2

Table 2. Fields to be filled in on return from the symbolic::symbolic\_functions module

|  |  |
| --- | --- |
| fun: Box<dyn Fun>, | Vector-function of grid equations; |
| jac: Option<Box<dyn Jac>> | Jacobian matrix-function; |
| bounds\_vec: Vec<(f64, f64)>, | Vector of bounds of admissible values for each of the grid variables; |
| rel\_tolerance\_vec: Vec<f64>, | Vector of relative accuracies for each of the discrete variables; |
| variable\_string: Vec<String> | Vector of grid variables; |
| bandwidth:(usize, isize) | Bandwidth of non-zero elements |

Now everything is ready to start the main iteration loop, which is implemented as the *main\_loop\_damped* function. Let's discuss the main loop:

0. Write the initial approximation to the solution *initial\_guess* in a field of the NRBVP structure called y, which is a vector that contains the current iteration of Newton's method for grid variables. We set the ‘Jacobian age’ counter (the number of iterations without recalculating the Jacobian) to zero m=0. The total iteration counter *i* is set to zero. This is the zero step of the algorithm. Everything is ready for computation.

1. We compute the Boolean function *jac\_recalc*, whose return value (obviously Boolean) is written to the field of the same name in the *NRBVP* structure. When jac\_recalc==true, it is time to recalculate the Jacobian, otherwise we can continue to calculate with the old Jacobian. The behavior of this function is specified by the value corresponding to the *max\_jac key* from the *strategy\_params* hash table (one of the user parameters). This value contains the ‘limit age of the Jacobian’ (the number of iterations without recalculating the Jacobian). When it is reached, the *jac\_recalc* function outputs true. Also, the Jacobian recalculation flag will be true when the value of the Jacobian matrix has not yet been calculated - at the 0th iteration

2. Go to the function *recalc\_jacobian*, if *jac\_recalc==false* immediately exit, continuing the calculations with the old Jacobian, in the opposite case, call the matrix-function of the Jacobian from the field of the structure *jac* and substitute in it *y* vector, which contains the current iteration of the values of the grid variables, calculate the Jacobian, write the matrix of numbers in the field *old\_jac*.

3- Increment the iteration counter i and the iteration counter without recalculating the Jacobian m.

4. Call the function *damped\_step*.

We do an unmodified (i.e., not multiplied by the dumping factor) Newtonian step by calling the step function. We compute the step dyund[k-1] knowing the vector of y[k-1] values.

4.1 The step function computes a simple (unmodified) Newtonian step. We call the vector function *fun*, and substitute in it a y vector that contains the current iteration of the values of the grid variables, compute it, and get the vector of incoherences. As we remember, the *old\_jac* field already contains the computed Jacobian matrix. Now we call the *solve\_sys* function to solve the linear system,

which takes a Jacobian, a vector of right-hand sides, and a user-defined *linear\_sys\_method* parameter, and returns a Newton step vector.

We compute the next iteration y[k] = y[k-1]- dyund [k-1].

We compute the unmodified step dyund[k].

4.2 Compute *fbound* the multiplier by which the Newton step from the previous paragraph is multiplied so that the result of the computation of the grid variables stays within the bounds defined for each of the grid variables, such pairs (left and right bounds) are contained in the *bounds\_vec* field, and the calculation of *fbound* is performed by the *bound\_step* function.

4.3 We enter a loop to find the modified Newton step (damped step) as well as the lambda dumping coefficient for it, a loop counter k operates inside the loop:

- The initial approximation to the final dumping coefficient will be *fbound*:

*lambda = fbound*.

- damped step dyd[k] =lambda\*dyund[k].

- quadratic norm of the damped step Sk =L2(dyd[k])

- iteration with modified step y[k+1] = y[k]- dyd [k]

- the next undamped step dyund[k+1].

- The quadratic norm of the undamped step Sk+1 =L2(dyund[k+1])

- We accept the dumping factor if Sk+1< Sk, and also if Sk+1< convergence condition. In this case we interrupt the loop of finding the dumping coefficient, otherwise we continue.

- reduce the current value of lambda by 2(k+DampFactor) times.

- increment the counter of modified steps k. If this counter reaches the value corresponding to the *maxDampIter* key from the *strategy\_params* hash table (one of the user parameters) during the cycles, the loop will be exited with the status -2.

- go to the next cycle by k.

Exit from the *damped\_step* function and return to the *main\_loop\_damped* function the result of the calculation (None or vector, solving status). The statuses can be as follows:

0 - convergence has not yet been achieved, however, the calculation of the damped Newton step has been successful, return a new approximation to the solution and the solution status;

1 - convergence has been achieved, calculation of the damped Newton step has been successful, return the solution and the solution status;

-2 - dumping factor not found, return None and solution status.

5. Depending on the status of the returned *damped\_step* in main\_loop\_damped, the following happens:

Status 1:

- If no adaptive meshes are assumed, i.e., the value corresponding to the adaptive key from the *strategy\_params* hash table is None, this means that a solution has been found. The final vector of grid variable values will be written to the result field. Congratulations!

- If adaptive meshes are assumed and the flag *new\_grid\_enabled==false*, it means that the additional conditions of convergence on the grid are fulfilled and a complete solution to the problem has been reached. The last vector of grid variable values will be written to the result field. Congratulations!

- If adaptive grids are supposed to be used and the flag *new\_grid\_enabled==true* means that although Newton's method has converged, the additional conditions of convergence on the grid are not fulfilled. The algorithm for recalculating the grid and initial approximation is started - the function *solve\_with\_new\_grid*, which will be discussed separately later.

Status 0:

The vector of grid variable values returned by *damped\_step* will be written to the y field for use in the next iteration.

Status <0:

Require Jacobian recalculation: *jac\_recalc==true*;

End of iteration, if not previously exited at step 5, we repeat the algorithm from para. 1. We continue iterating either until the solution or until *max\_iter* - the limit number of iterations - is exhausted.

Let's summarise the results. So, during the execution of the loop main\_loop\_damped

|  |  |
| --- | --- |
| y: Box<dyn VectorType>, | vector of grid variable values |
| m: usize, | Iteration counter without Jacobian recalculation |
| old\_jac: Option<Box<dyn MatrixType>>, | Numerical value of Jacobian |
| Jac\_recalc:bool | Jacobian Recalculation Flag |

Let us discuss the construction of a new mesh.

Although each algorithm has its own peculiarities, common features should be noted:

- the new mesh is produced from the old mesh by adding points in some parts of the mesh;

- the new mesh naturally requires the generation of a new system with a large number of equations;

- the first approximation for the new system (with a denser grid) has to be prepared from the solution of the old system of equations, but since the new system will have more sampling points than the old one, it requires interpolation just in the cells where the new points have been added.

Let us now see how the above is implemented in the *solve\_with\_new\_grid* function, which is run in step 5.

Its launching entails the following sequence of execution:

A) we run the function *create\_new\_grid*, which checks whether the hash table *strategy\_params* contains a key corresponding to one of the algorithms for grid recalculation, the vector of values at this key contains the parameters of the algorithm for recalculation. There are two possible keys: ‘*pearson*’ and ‘*grcar\_smooke*’, their corresponding algorithms will be discussed below. Next, the *grid\_api* interface intermediate between the algorithms and this function is used and calculations are performed.

The *create\_new\_grid* function returns the following values:

- *new\_mesh* (new grid),

- *initial\_guess* (the initial approximation prepared from the solution of the old system for the new system) - written in the *initial\_guess* field,

- *number\_of\_nonzero\_keys* (number of sites where additional convergence conditions on the grid were not fulfilled) is written in the field of the main structure *number\_of\_refined\_intervals*.

B) The equation generation function *eq\_generate* (see above) is called with the new grid *new\_mesh* as an argument.

C) The function *we\_need\_refinement* is called, which checks that if *number\_of\_refined\_intervals* == 0, it means that the additional convergence conditions on the grid are fulfilled for all intervals, and the grid recalculation flag new\_grid\_enabled is set to false, so that the solution of the newly generated system will be the last step of the solution. There will be no restart of the problem with the new grid.

D) The main loop (see above) *main\_loop\_damped* is started again.

End of *solve\_with\_new\_grid*.

Let's summaries the results. So, we use the following fields of the main structure during the execution of the new grid calculation:

|  |  |
| --- | --- |
| *new\_grid\_enabled: bool* | Grid recalculation flag |
| *grid\_refinemens: usize* | Grid recalculation counter |
| *number\_of\_refined\_intervals: usize,* | number of sites where additional convergence conditions were not fulfilled on the grid |

Let us now discuss the modules designed to solve the linear system.

The *linear\_sys\_sovers\_depot* module works like this:

If the job specifies *method= ‘Dense’* - this means that we want to use the dense matrix apparatus from the *nalgebra* library (crate), so from the *linear\_sys\_sovers\_depo*t module the function *nalgebra\_sovers\_depot* is selected, which works according to the following logic:

- the parameter ‘full’ corresponds to using the solver for ordinary matrices;

- the ‘band’ parameter corresponds to using the solver for band matrices;

- if the *linear\_sys\_method* parameter is None, the code compares the dimension of the Jacobian matrix with the diagonal width (band), if this ratio exceeds 10, the ‘band’ parameter is selected, otherwise ‘full’;

- if the *linear\_sys\_method* assignment parameter is not None, it is either ‘full’ or ‘band’. The solver is selected accordingly to the job parameter.

After successful completion of the solution, the following options are available to the user:

- Printing the calculation result as a table.

- Drawing of graphs with the calculation results for each of the unknowns.

A detailed log containing the course of calculations is kept during the whole time of the program operation. It will also be printed.

This is a brief description of the BVP solution modules for the developer.

3. User's Manual.

The main structure containing the calculation task is called BVP and is located in the numerical::BVP\_api module. To initialise the solution, you need to call an instance of this structure and use the new function to pass the task conditions described in Table 1 to the structure.

Table 1.

Table 1.

|  |  |
| --- | --- |
| Public field (to be filled in when completing the task): field data type | Description. |
| eq\_system: Vec<Expr> | Vector of RHS of the system of equations. Defined in symbolic (analytical) form (NB!). |
| initial\_guess: DMatrix<f64> | Initial approximation to the solution (matrix of dimension equal to the number of unknown variables\*number of grid steps) |
| values: Vec<String> | Vector of unknowns |
| arg: String | Name of argument (x, t, etc.) |
| BorderConditions: HashMap<String, (usize, f64) | Hash table (dictionary) of initial conditions, where the key is the name of the unknown for which the condition is entered, the value is a pair of numbers: the first number 0 or 1 is the initial or boundary condition, respectively, the second number is the value itself. |
| t0: f64 | Initial value of the argument |
| t\_end: f64 | Final argument value |
| n\_steps: usize | Number of discretisation steps (must be consistent with the dimensionality of the initial approximation matrix, see above). If an adaptive grid algorithm is chosen, the specified number of discretization steps will be changed during the solution. |
| scheme: String | Parameter for specifying the numerical scheme:  ‘forward’ - simple forward difference,  ‘trapezoid’ - see e.g. https://en.wikipedia.org/wiki/Trapezoidal\_rule\_(differential\_equations) |
| strategy: String | Newton's method strategy:  ‘Naive’ The Jacobian is recalculated every iteration, there is no dumping, there is no holding of the Newton step within specified bounds, there are no adaptive meshes - the simplest variant of the method;  ‘Frozen’ - Jacobian is recalculated not every iteration, but according to some condition set by parameters in strategy\_params hash table, however, there is no dumping, no Newtonian step retention within given bounds, no adaptive meshes – more advanced variant of the method;  ‘Damped’ - the most advanced variant of the method: dumping, holding Newtonian step in the given bounds, adaptive meshes, Jacobian recalculation by condition. The required control parameters should be specified in the *strategy\_params* hash table |
| strategy\_params: Option<HashMap<String, Option<Vec<f64>>>>, | Solving strategy parameters. Specific for each strategy and will be described separately. |
| linear\_sys\_method: Option<String> | Parameter for specifying the solver for the system of linear equations. |
| method: String | Parameter for specifying the type of specific software implementation of basic linear algebra structures (actually linear algebra library (crate), actually defines the way of matrix data storage).  ‘Dense’ - dense matrices are used;  ‘Sparse’ - sparse (suitable for large and very large problems); |
| max\_iterations: usize | The maximum allowed number of Newton iterations, if the solution is not reached within this number, the program is exited. |
| tolerance: f64 | Absolute accuracy |
| rel\_tolerance: Option<HashMap<String, f64>>, | Relative accuracy. A hash table where the key is the name of a variable and the number is the precision value for that variable. |
| Bounds: Option<HashMap<String, (f64, f64)>> | Permissible physical boundaries of the physical quantity (e.g., obviously the mole fraction is between 0 and 1, and the absolute temperature in combustion physics between the value ‘in the room’ and (conventionally) 5000 K). A hash table, where the key is the name of the unknown variable and the value is a pair of numbers (upper and lower bounds). This information is used in the algorithm for keeping the Newtonian step within the given bounds. |
|  |  |

Let's characterise the key-value pairs of the strategy\_params hash table, which is required to be set by the user among other mandatory fields.

|  |  |  |
| --- | --- | --- |
| Key name | Default value (enabled if user specifies None instead) | Mathematical meaning |
| max\_jac | 3 | Number of iterations without Jacobian recalculation, i.e. with old Jacobian. |
| maxDampIter | 5 | Limit number of iterations of dump factor search |
| DampFacor | 0.5 | The factor to reduce the estimated dump factor if the previous search iteration was not successful |
| adaptive | If the user specified None there will be no recalculation of meshes | Whether or not recalculation of meshes is required. |
| The following pairs characterise the grid recalculation algorithm and are meaningful if the value for adaptive is not None. | | |
| grcar\_smooke | No default values | Algorithm of authors GrCar and Smooke |
| pearson | No default values | Pearson algorithm |