Anderson Acceleration for Fixed-Point Iterations

Martino Ischia

Supervisor: Prof. Formaggia

January 2021

Abstract

The goal of this project is twofold.
Firstly, develop a C++ interface for accelerating a converging sequence of vectors. It should be at the same time open for extensions, in case the user wants to implement its own algorithm, and efficient in the solution of large systems.
Secondly, making use of said interface to implement Anderson acceleration strategy and apply it on linear and nonlinear problems arising from the discretization of differential equations.
The reader not interested in the C++ implementation can safely skip section 2, even though the classes described there are sometimes mentioned in section 3.

Contents

1 Introduction

A fixed-point problem consists in finding the point (a vector in $\mathbb{R}^n$) that satisfies the following:

$$x = g(x) \tag{1}$$

where g is a function from $\mathbb{R}^n$ to $\mathbb{R}^n$.
It is clearly equivalent to the problem of finding the roots of a generic function, that is often tackled by employing Newton-Raphson iterative algorithm. In many concrete applications, though, the cost of computing the Jacobian of a function is not practical, not to mention that there might be issues in starting the iterations from a proper guess.
Considering the fixed-point form and solving through fixed-point iterations has also some limitations, the main one being a slow convergence: in fact most of the times the convergence is linear.
Several strategies have been described in the literature to improve the speed of convergence of a vector sequence: we could refer to them as acceleration methods. This project focuses on one of them, proposed by Anderson [2] in 1965.

1.1 Anderson Algorithm

There are many equivalent ways to formulate Anderson acceleration method. I report the one in Walker and Ni [7].
Several adjustment can be made to address specific problems, but in this project only this simple form is considered. The interested reader can refer to [3] [7].

Algorithm 1 Anderson algorithm

Given x_0 and $m \geq 1$
Set $x_1 = g(x_0)$
for $k = 1, 2, ...$ until convergence **do**
Set $m_k = min\{m, k\}$
Set $F_k = (f_{k-m_k}, ..., f_k)$, where $f_i = g(x_i) - x_i$
Determine $\alpha^{(k)} = (\alpha_0^{(k)}, ..., \alpha_{m_k}^{(k)})^T$, subject to $\sum_{i=0}^{m_k} \alpha_i = 1$, that solves

$$\min_{\alpha=(\alpha_0,...,\alpha_{m_k})^T} \|F_k\alpha\|_2 \tag{2}$$

Set $x_{k+1} = (1-\beta)\sum_{i=0}^{m_k} \alpha_i^{(k)} x_{k-m_k+i} + \beta \sum_{i=0}^{m_k} \alpha_i^{(k)} g(x_{k-m_k+i})$
end for

The new value is obtained through a linear combination of the previous iterates and their evaluations. The coefficients are found through a minimization problem, in this case written as a constrained problem, but an unconstrained form is usually chosen for the implementation [3][7].

A compact way to write the update formula, albeit not fully rigorous, is as follows:

$$x_{k+1} = x_k + \beta f_k - (\mathscr{X}_k + \beta\mathscr{F}_k)(\mathscr{F}_k^T\mathscr{F}_k)^{-1}\mathscr{F}_k^T f_k \tag{3}$$

where

- $\mathscr{X}_k = [\Delta x_{k-m} ... \Delta x_{k-1}]$ with $\Delta x_i = x_{i+1} - x_i$
- $\mathscr{F}_k = [\Delta f_{k-m} ... \Delta f_{k-1}]$ with $\Delta f_i = f_{i+1} - f_i$ and f_i is still $g(x_i) - x_i$

Notice that the algorithm depends on two parameters: β, a relaxation parameter which has also a special interpretation when the algorithm is seen as a multisecant method [3], and m, a memory parameter, which represents the number of previous iterates considered in the calculations.

2 C++ interface for fixed-point problems

2.1 Overview of the code

The code can be found at this link and is subdivided into two directories: `Include` and `src`. The `Include` directory contains (mostly) external utilities that are used.
The `source` directory contains the interface and the implementation of the classes, as well as examples of increasing level of complexity.
The only library needed for compiling the code is the header-only Eigen version 3 (I am using 3.3). The path of the Eigen library has to be specified in `/src/Makefile`.
Special care has been put into documenting the code with the use of the `Doxygen` package (`graphviz` package is also required in order to produce cool looking graphs). The interested reader is suggested to consult the documentation by running `make doc` in the `src` directory. An `html` directory will be generated, containing an `index.html` file. Open it in a web browser for consulting the documentation.

In the next sections I will describe the most interesting features of the code, so again I redirect to the documentation the reader who wants a complete picture of the functionalities provided by the interface.
The code is based on two classes: `Iterator`, a functor that provides the next value of a sequence, and `FixedPointIterator`, which composes the former and provides a `compute` method for producing iterations until some criteria are satisfied.

2.2 Traits

The classes mentioned above, `Iterator` and `FixedPointIterator`, are generic in the sense that they do not define the types they use: they inherit them from an external struct. The code offers two options, `DenseTraits` and `SparseTraits`, but user can define his own.
From eq. 1 it is clear that a fixed-point problem needs a type for vector x and a type for iteration function g. Moreover, since Anderson algorithm requires calculations on matrices, a matrix type is needed. The matrix type is in fact what differentiates `DenseTraits` from `SparseTraits`, with obvious meaning.
For the definition of this types, we rely on the Eigen library, a fast and versatile C++ template library for linear algebra.

This Traits structs are used by other classes by mean of inheritance: the `Traits` macro is defined in `Accelerators.hpp` as, for example, `SparseTraits`, and all classes inherit from `Traits`, obtaining in such a way a definition of `Vector`, `IterationFunction` and `Matrix` types as well as some utilities related to this types.

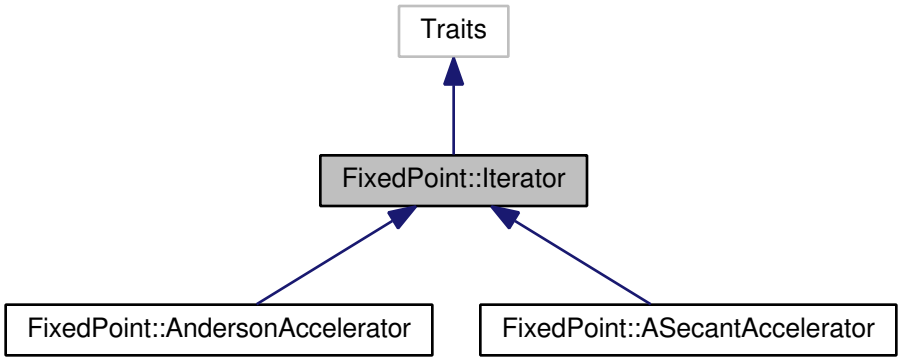

Figure 1: *Inheritance diagram for the* `Iterator` *class*

2.3 The Iterator family

The first ingredient is a class that stores the fixed-point iteration function in eq. 1 and provides the next value of a vector sequence. The `Iterator` class overloads the call operator to provide such new value. In this base class, what the call operator does is rather trivial: it just calls the iteration function applied to the last value stored in the argument.
To be more precise, the operator has the signature

```
Traits::Vector
Iterator::operator()(const std::deque<Vector>& past)
```

It is intuitive that the `past` parameter represents an object storing the past vectors assumed by a certain sequence. The choice for the data structure to be an `std::deque` will be clear when we will describe the `FixedPointIterator` class.
The constructor is

```
template <class IterationFun>
Iterator(IterationFun && IF, std::size_t dim):
phi(std::forward<IterationFun>(IF)) ...
```

The use of the so-called universal reference in combination with `std::forward` allows to move the argument when possible (an `std::function`, which is the type of the iteration function, may potentially have a big size in the form of a function object).

The `Iterator` class is the base class for more advanced classes which try to speed-up the convergence process. The derived classes make use of polymorphism to overload the function call operator.
In fig. 1 is shown the inheritance diagram of the `Iterator` class. In addition to the `AndersonAccelerator` class, another class called `ASecantAccelerator` is provided, wihch implements a simpler version of Anderson algorithm in which the memory parameter has value two.

Let us dive deeper into the implementation of Anderson algorithm.

2.4 Implementation of Anderson algorithm

Equation 3 shows that to compute the next value of the sequence two matrices are needed: $\mathscr{X}_k$ and $\mathscr{F}_k$.
The `AndersonAccelerator` class stores two matrices `X` and `F` which are updated only in the last column at each call of the call operator.
These two matrices have a special type to perform the update operation easily: it is the `RotatingMatrix` template class. The template argument is chosen to provide the most efficient insert strategy, the one in which the new inserted column replaces the oldest column when the maximum size of the matrix has been reached.
Since the call operator only updates the last column, it **can not** be called on a generic vector sequence but only on a sequence that was produced by the class itself. To circumvent this problem, the `SetUp` method computes $\mathscr{X}_k$ and $\mathscr{F}_k$ up to the values involving the last iteration. If an Anderson step wants to be applied to a vector sequence produced in another way, the `SetUp` method has to be called before the call operator.

Concerning the update formula 3, the term $\mathscr{F}_k^T\mathscr{F}_k^{-1}\mathscr{F}_k^T f_k$ represents the solution of a least square problem. It is well known that solving this so-called normal equations is a bad choice from the numerical point of view, due to ill-conditioning of the matrices. For this reason, the least square problem is solved by mean of the QR decomposition of the matrix $\mathscr{F}_k$. In particular the Householder rank-revealing QR decomposition with column-pivoting is used, that is a good compromise between numerical stability and efficiency.
Interestingly enough, the result of the update formula 3 is independent of the ordering of the columns of $\mathscr{X}_k$ and $\mathscr{F}_k$. This is relevant since the proper reordering of the columns of the `RotatingMatrix` class would add another computational cost.
The fact can be seen from the QR decomposition but is more easily seen from the normal equations.
Calling P a generic permutation matrix (which has the property that $P^T = P^{-1}$), swapping column of a matrix A is achieved by performing the matrix multiplication AP. Substituting $\mathscr{X}_k$ and $\mathscr{F}_k$ with their permuted counterpart, eq. 3 becomes

$$x_{k+1} = x_k + \beta f_k - (\mathscr{X}_k + \beta\mathscr{F}_k)P(P^T\mathscr{F}_k^T\mathscr{F}_k P)^{-1}P^T\mathscr{F}_k^T f_k \tag{4}$$

Thus, for properties of the inverse of a matrix

$$x_{k+1} = x_k + \beta f_k - (\mathscr{X}_k + \beta\mathscr{F}_k)PP^T(\mathscr{F}_k^T\mathscr{F}_k)^{-1}PP^T\mathscr{F}_k^T f_k \tag{5}$$

By simplifying PP^T we obtain that the solution is invariant with respect to permutation of columns of $\mathscr{X}_k$ and $\mathscr{F}_k$.

One last note: Walker and Ni [7] make use of factor-updating QR decomposition, which is a more efficient algorithm for computing the QR factors of $\mathscr{F}_k$ from the ones of $\mathscr{F}_{k-1}$. We did not consider it in this project because such algorithm was not available in the Eigen library. Moreover the rank-1 updated QR decomposition would not work when the oldest column of the matrix gets replaced.

2.5 The `FixedPointIterator` class

The `FixedPointIterator` class governs the solving of the fixed-point problem.
It has the following private attributes:

- `iterator`, an `std::unique_ptr` to an `Iterator` object that produces the iterates. The unique pointer allows to compose an object in the polymorphic `Iterator` family, for example an `AndersonAccelerator` object.
- `options`, an options struct, controlling maximum iterations, tolerances and printing styles.
- `history`, an `std::deque<Vector>` containing the history of the vector sequence. `std::deque` was prefered to `std::vector` because when the number of stored vectors reaches the threshold of `options::memory`, the oldest vector is eliminated from the front and the new vector is added to the back. This operation is much more efficient on `std::deque`.
- `iteration`, an integer representing the number of executed iterations by the `compute` method.

Concerning the public member functions, the class provides:

- a `compute()` method taking no arguments
- a `compute(const Vector&)` method taking an initial vector as a starting guess
- methods to output a result (`printResult` is one possibility).

The `compute` methods make use of the `history` member to compute iterations until convergence. In fact, the initial guess is used only in the case `history` is empty. If `history` is empty and no guess is provided, a default vector of zeros is used.

3 Examples and applications

3.1 A first example

The file `main_simple_problem.cpp` constitutes a first working example of the previously defined classes.
It solves a simple nonlinear problem in 3D, taken from book [5], whose fixed-point form converges to the solution $(0, 1/3, 0)$.
All the parameters of interest are read from an input file, exploiting GetPot, a simple but effective utility. This allows to not recompile the code when we want to test a different parameter.

Figure 2 shows the residual at each iteration when standard fixed-point iterations are used and when Anderson algorithm is used. Parameters of Anderson algorithm are $\beta = 1$ and memory $m = 5$.
Fixed-point iterations converge in 28 iterations whereas Anderson algorithm converges in 17.

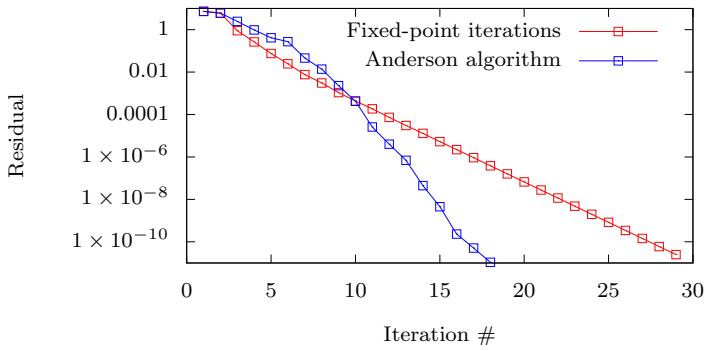

Figure 2: *Residual vs Interation number for the nonlinear 3D problem*

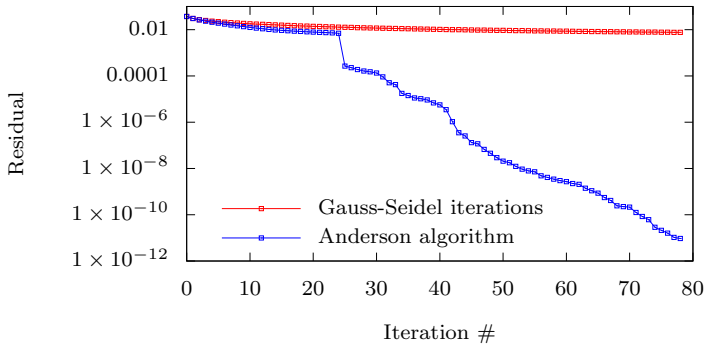

Figure 3: *Residual vs Interation number for the Gauss-Seidel problem*

3.2 A linear FEM example

The file `main_gauss_seidel.cpp` solves the heat equation, in the form of a diffusion reaction problem, on a one-dimensional bar with Dirichlet condition on one side and Neumann condition on the other. For further description, consult [4].
By FEM discretization (or equivalently by finite differences, since we are in 1D), a tridiagonal matrix is obtained.
We decided to implement the simple Gauss-Seidel iterative scheme, notoriously slow. Using as parameters of Anderson algorithm the same as the example before, and as mesh size 25, we can observe in fig.3 that Anderson accelerator does a good job at accelerating this method: the Gauss-Seidel method would need 3339 iterations to converge!

An analysis concerning the parameters m and β was carried out for this example. In fig. 4 the heat map represents on a log-scale the time needed to reach convergence. Where the map is white, it means the iterations did not converge (they most likely diverged). For every parameter set the code was executed 10 times to obtain an average execution time.
You can see that $\beta = 1$ is by far the best choice, while for the memory parameter several choices are good. For high number of the memory parameter, the code converged in less than 30 iterations, so increasing m would not change the result.

Also a simple profiling of the code has been performed, through `gprof`. It showed that

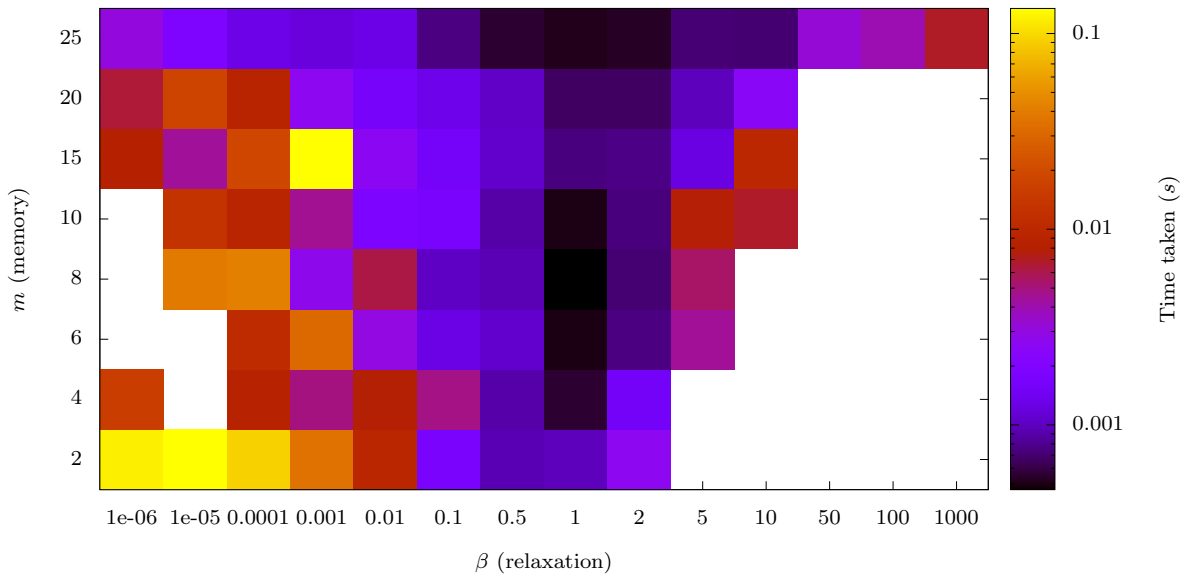

Figure 4: *Heat map of execution time with different values of the parameters m and β: the darker means the code was faster. White squares means convergence was not reached.*

most of the execution time for the Anderson iterations, by far, is spent in the QR decomposition of the $\mathscr{F}_k$ matrix (run `make profiler` for seeing with your eyes in the generated `profiling.txt` file).

3.3 Alternating method

Following the idea in paper [6], an alternating Anderson scheme was implemented.
The idea is to run simple fixed-point interations for a number p of iterations, followed by an Anderson step and keep alternating in such a way. It is particularly good when applied to the Jacobi iterative scheme because it can be simply implemented in a parallel architecture. Since I am using a normal computer, I have used instead the Gauss-Seidel scheme of the previous example, but changing two characters in the code would transform the Gauss-Seidel method into the Jacobi method, without substantially changing the results.

Implementation-wise, I chose to exploit all the functionalities provided by the interface to implement the alternating scheme in `main_gauss_seidel.cpp`, rather than wrapping `AndersonAccelerator` into a suitable `Iterator` class. The `swap` method of `std::unique_ptr` was used for alternating the `AndersonAccelerator` and the `Iterator` classes stored in the `FixedPointIterator` object.

Different values of the number p have been tested for parameters $\beta = 1$ and memory $m = 8$, but very similar behaviour is observed for different values of m. Fig. 5 shows that increasing p to a value slightly greater than m accelerates the process. It is not shown in the figure, but note that the standard Anderson method converges in about $0.0005s$.
With the alternating method we were able to reach a speed-up of almost 10, which is quite impressive.

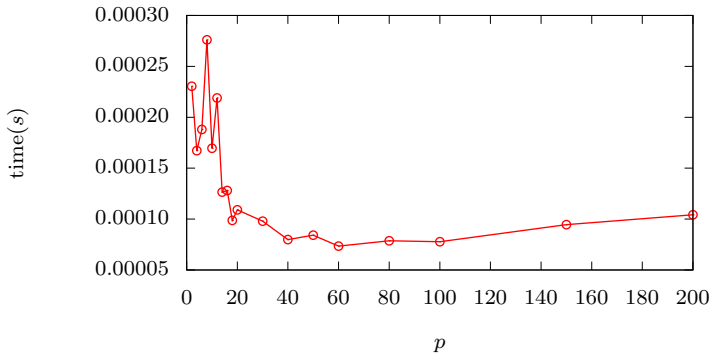

Figure 5: *Testing different values of parameter p with* $\beta = 1$ *and* $m = 8$

3.4 Nonlinear FEM example

To study a nonlinear problem, we decided to add to the heat equation of previous examples a thermal radiation term.
At the discrete level, this means solving the nonlinear system of equations

$$Au + \sigma I u^4 = f \tag{6}$$

where A is the same finite elements matrix corresponding to the previous example, whose exact form is found in [4], I is the identity matrix, σ is the radiation coefficient and f is an appropriate right-hand side (I did not stick to the physics of the problem, because the radiation term is usually negligible, so I assigned $\sigma = 1$ and I did not account for necessary corrections of the right-hand side).

This formulation is easy to derive when we approximate the problem with finite differences, while with the use of finite elements it is obtained through the lumping of the mass matrix, a procedure used to avoid extra-diagonal terms.

We solve system 6 written in a fixed-point form in the following manner:

$$u = (A + \sigma I u^3)^{-1} f \tag{7}$$

resulting in the iterative scheme

$$u_{k+1} = (A + \sigma I u_k^3)^{-1} f \tag{8}$$

This in turns results in solving at each step the linear system

$$(A + \sigma I u_k^3) u_{k+1} = f \tag{9}$$

which is always possible being the corresponding matrix positive definite.
This step was performed in the same manner of previous section, that is through the `FixedPointIterator` class, accelerating the Gauss-Seidel iterative procedure with the alternating Anderson scheme. We have therefore created a fixed-point problem inside another fixed-point problem.
In case the inner iterator does not converge, the code exits throwing a run-time exception

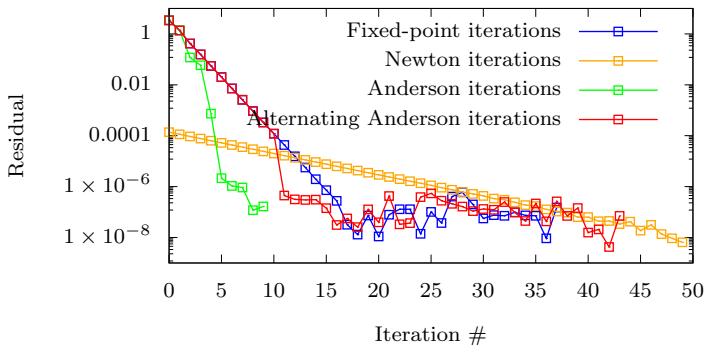

Figure 6: *Residual vs Interation number for the nonlinear problem*

(but given a suitable number of iterations it should always converge thanks to the properties of the Gauss-Seidel scheme).

In fig. 6 you can see the residual history for the iterative scheme 8 for different solvers. The fixed-point solver has linear convergence, as it usually happens. Anderson accelerator manages to speed-up significantly the fixed-point iterations and is able to converge in very few iterations. Alternating Anderson seems not to benefit particularly of the normal fixed-point iterations and much more from the Anderson step, but this can not be said from just this figure.
The Newton method for problem 6 reads

$$x_{k+1} = x_k - (A + 4\sigma I x_k^3)^{-1}(Ax_k + \sigma I x_k^4 - f) \tag{10}$$

In the figure the Newton method has linear convergence and the slope is less steep than the one of fixed-point 8.

3.5 Conclusions

A C++ interface for fixed-point iterations was successfully implemented. It is quite simple to use (just reference to the first example), and well documented. It is customizable and can be used possibly with both sparse and dense matrices.

The Anderson algorithm was implemented in an efficient manner and a sufficiently stable algorithm for the QR decomposition of the matrices involved was used (but this policy can be changed easily by just refering to the Eigen library).
In all the examples made the Anderson algorithm was able to speed-up the computational time. On linear problems was extremely successful the alternating scheme in [6]. On the nonlinear heat equation, Anderson algorithm allowed to reach convergence in few iterations and outperformed severly Newton method.
Further study should be made on more complex problems, increasing the dimensionality of the system, and on different types of nonlinear problems.

A Appendix

A.1 A C++ program for parameters testing

This simple C++ code, which makes use of many Linux shell commands, was used in the course of the project for performing several parameters tests.
I think it is worth reporting it here because it can be generalized (and improved) with little effort and used very easily in similar contexts (that is where we have an input file for getting parameters into the C++ code). Simple for loops inside the C++ code would not provide the same flexibility and elegance.

The following `exec` function (source [1]), based on the shell command `popen`, executes a command in the shell and returns the resulting output in an `std::string`.

```
#include <cstdlib>
#include <cstdio>

std::string exec ( const char* cmd )
{
        std::array <char, 128> buffer;
        std::string result;
        std::unique_ptr <FILE, decltype(&pclose)> pipe
                                (popen(cmd, "r"), pclose);

        if (!pipe) {
                throw std::runtime_error("popen() failed!");
        }
        while ( fgets(buffer.data(), buffer.size(), pipe.get() )
                                                != nullptr)
        {
                result += buffer.data();
        }
        return result;
}
```

It is utilized in the files whose name contain `main_test` in the following manner:

```
        std::vector <double > parameter {2, 4, ...};
        std::ostringstream tmp;
        tmp << "Parameter = " << parameter[i] << std::flush;
        std::system ( ("sed -i '45s/.*/" +
        tmp.str() + "/' data.input").c_str() );

        double value = stod (exec("./main_gauss_seidel |
        grep 'TIME: Alternating Anderson method' |
                        tr -dc '.0-9'"));
```

Explanation of the code:

- `parameter` contains the values of the parameters we want to test.
- `tmp` is used to build the string we want to substitute in the input file to change value of the parameter.
- `std::system` executes a command in the shell.
- `sed` is a shell command for editing files (*stream editor*). Used in that manner, it means replace line 45 of `data.input` with the string in `tmp` (+ is an operator of `std::string` to concatenate them).
- `stod` converts the string returned by `exec` into a double.
- `grep` and `tr` are shell commands that I used to extract the desired characters from the output of the `main` function.

Bibliography

[1] URL: `https://stackoverflow.com/questions/478898/how-do-i-execute-a-command-and-get-the-output-of-the-command-within-c-using-po`. (accessed: 04.02.2021).

[2] Donald Anderson. "Iterative Procedures for Nonlinear Integral Equations". In: *J. ACM* 12 (Oct. 1965), pp. 547–560. DOI: 10.1145/321296.321305.

[3] Haw-ren Fang and Yousef Saad. "Two classes of multisecant methods for nonlinear acceleration". In: *Numerical Linear Algebra with Applications* 16 (Mar. 2009), pp. 197–221. DOI: 10.1002/nla.617.

[4] Luca Formaggia. URL: `https://github.com/pacs-course/pacs-examples/tree/master/Examples/src/HeatExchange`. (accessed: 04.02.2021).

[5] Paola Gervasio et al. *Matematica Numerica*. Feb. 2014. ISBN: 978-88-470-5644-2.

[6] Phanisri Pratapa, Phanish Suryanarayana, and John Pask. "Anderson acceleration of the Jacobi iterative method: An efficient alternative to Krylov methods for large, sparse linear systems". In: *Journal of Computational Physics* 306 (Dec. 2015). DOI: 10.1016/j.jcp.2015.11.018.

[7] Homer Walker and Peng Ni. "Anderson Acceleration for Fixed-Point Iterations". In: *SIAM J. Numerical Analysis* 49 (Aug. 2011), pp. 1715–1735. DOI: 10.2307/23074353.