Introduction to Optimisation: Project

This repository was created as a final project for the subject ELEN90026 Introduction to Optimisation at The University of Melbourne.

The purpose of this project is to demonstrate two aspects of using the method of dual decomposition for solving an optimisation problem with a separable cost function. The first aspect is the application of parallel computing to simulate decomposing the separable problem into subproblems and solving those subproblems in parallel on separate machines. The second aspect is the affect of a limit on the packet size of messages passed between these parallel processes/machines on convergence of the overall algorithm.

In terms of hardware, the examples and results were generated using a 2.90GHz Intel i7-7500U CPU with 2 cores (4 logical processors).

Background

Separable Cost Function

To illustrate a separable problem, let's begin by considering the following cost function that can be split into two functions. Both functions share a complicating variable x_3.

$$\min_{x_1,x_2,x_3} f_1(x_1, x_3) + f_2(x_2, x_3)$$

or equivalently

$$\begin{aligned} \min_{x_1,\xi_1,x_2,\xi_2} \quad & f_1(x_1, \xi_1) + f_2(x_2, \xi_2) \\ s.t. \quad & \xi_1 = \xi_2, \end{aligned}$$

where $x_1 \in \mathbb{R}^n$, $x_2 \in \mathbb{R}^m$ and $x_3, \xi_1, \xi_2 \in \mathbb{R}$ for some n and m.

Dual Decomposition and the Subgradient Method

The dual function for this problem is then

$$\begin{aligned} q(\lambda) &= \inf_{x_1,\xi_1,x_2,\xi_2} [f_1(x_1,\xi_1) + f_2(x_2,\xi_2) - \lambda^\top(\xi_1 - \xi_2)] \\ &= \inf_{x_1,\xi_1} [f_1(x_1,\xi_1) - \lambda^\top(\xi_1)] + \inf_{x_2,\xi_2} [f_2(x_2,\xi_2) + \lambda^\top(\xi_2)] \\ &= q_1(\lambda) + q_2(\lambda) \end{aligned}$$

We seek a λ that maximises this dual function, which can be found using the subgradient method. Here the negative subgradient of the dual function at a given point λ_k is given as $g_k = \xi_1 - \xi_2$. The subgradient method involves iteratively

- finding $(x_1, \xi_1, x_2, \xi_2) \in \arg\min_{x_1,\xi_1,x_2,\xi_2} [f_1(x_1,\xi_1) + f_2(x_2,\xi_2) - \lambda^\top(\xi_1 - \xi_2)]$, and
- updating $\lambda_{k+1} = \lambda_k - \alpha_k(\xi_1 - \xi_2)$, where α_k is the step size.

The method of dual decomposition is simply making use of the fact that the dual function can be decomposed into $q_1(\lambda)$ and $q_2(\lambda)$ as shown above, and so finding

$$(x_1, \xi_1, x_2, \xi_2) \in \arg\min_{x_1,\xi_1,x_2,\xi_2} [f_1(x_1,\xi_1) + f_2(x_2,\xi_2) - \lambda^\top(\xi_1 - \xi_2)]$$

can instead be done by two individual agents separately finding the minimisers

$$(x_1, \xi_1) \in \arg\min_{x_1,\xi_1} [f_1(x_1,\xi_1) - \lambda^\top(\xi_1)]$$

$$(x_2, \xi_2) \in \arg\min_{x_2,\xi_2} [f_2(x_2,\xi_2) + \lambda^\top(\xi_2)]$$

Parallel Computing

The **multiprocessing** module on Python was used to execute processes in parallel. A key point is that to simulate different agents separately executing tasks, this is better reflected by spawning parallel processes rather than threads.

Outcomes

Some expected outcomes/results from this project which will be demonstrated in the coming problems.

1. As a result of dual decomposition, there are steps in the subgradient method that can be done in parallel, namely finding the minimisers of the decomposed dual function.

The first expectation is that parallelising these processes will not affect the convergence of the subgradient method.

2. There is some computational overhead in spawning processes on Python. If the processes are simple computations, then serial computing may end up being faster than parallel computing due to this overhead.

3. The number of separations in the cost function do not matter, only that individually, each separation of the function is convex.

4. Parallel processing makes use of the multiple cores of a processor. Using a CPU with more cores means more processes can be run in parallel.

5. The floating point precision of messages passed between agents affects the convergence of the subgradient method. We lose convergence guarantees if the messages become too imprecise.

Problem 1

This is a pedagogical toy problem. It is designed to show how Process and Pipe objects from the multiprocessing module on Python can be used in a separable optimisation problem. This problem relates to Outcome 2, but also touches briefly on Outcomes 1 and 3.

Let's use a specific example of a problem with cost function that can be partitioned into two parts. The following is the separable unconstrained optimisation problem to be solved:

$$\min_{x_1,x_2,x_3} \quad x_1^2 + x_2^2 + 2x_3^2,$$

where $x_1, x_2, x_3 \in \mathbb{R}$. Let x_1 and x_2 both be private variables, i.e., they can only be accessed by Agent 1 and Agent 2 respectively. Let x_3 be a shared or public variable that can be accessed by all agents. The minimiser of this problem is obviously $(x_1^*, x_2^*, x_3^*) = (0, 0, 0)$. It is a simple problem, the point is really to give Python's multiprocessing module a test run on an optimisation problem.

By applying a change of variables, the cost function can be separated and the new formulation of the problem will be

$$\begin{aligned} \min_{x_1,\xi_1,x_2,\xi_2} \quad & [x_1^2 + \xi_1^2] + [x_2^2 + \xi_2^2] \\ s.t. \quad & \xi_1 = \xi_2, \end{aligned}$$

where $x_1, \xi_1, x_2, \xi_2 \in \mathbb{R}$. The dual function will then be

$$\begin{aligned} q(\lambda) &= q_1(\lambda) + q_2(\lambda) \\ &= \inf_{x_1,\xi_1} [x_1^2 + \xi_1^2 - \lambda\xi_1] + \inf_{x_2,\xi_2} [x_2^2 + \xi_2^2 + \lambda\xi_2]. \end{aligned}$$

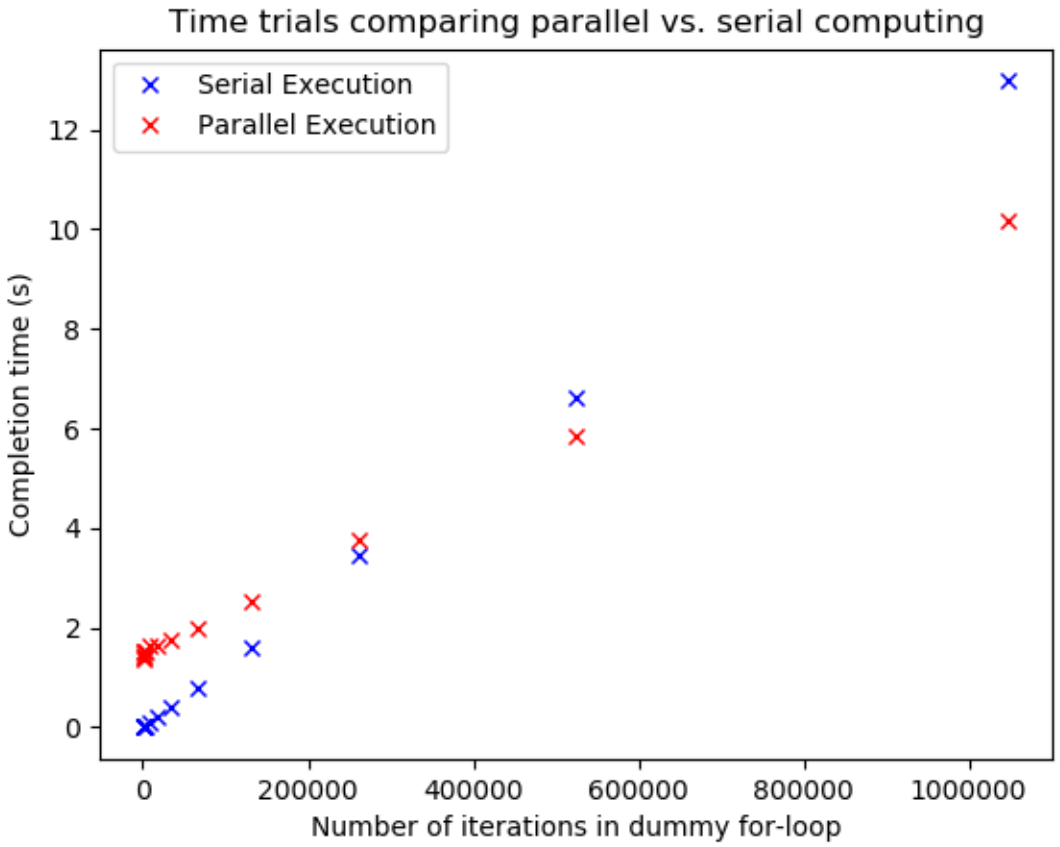
The minimisers of $[x_1^2 + \xi_1^2 - \lambda\xi_1]$ and $[x_2^2 + \xi_2^2 + \lambda\xi_2]$ will be $(x_1^*, \xi_1^*) = (0, \lambda/2)$ and $(x_2^*, \xi_2^*) = (0, -\lambda/2)$ respectively. The algorithm will be as follows:

- At step $k = 0$, some master processor chooses an initial $\lambda(0) = 1.0$ and sends $\lambda(0)$ to Agents 1 and 2. Agent 1 calculates $\xi_1^*(0) = \lambda(0)/2$, while (in parallel) Agent 2 calculates $\xi_2^*(0) = -\lambda(0)/2$. Agents 1 and 2 send $\xi_1^*(0)$ and $\xi_2^*(0)$ back to the master processor.
- At step $k = 1$, the master processor updates $\lambda(1) = \lambda(0) - \alpha(\xi_1^*(0) - \xi_2^*(0))$ and sends $\lambda(1)$ to Agents 1 and 2. Agent 1 calculates $\xi_1^*(1) = \lambda(1)/2$, while (in parallel) Agent 2 calculates $\xi_2^*(1) = -\lambda(1)/2$. Agents 1 and 2 send $\xi_1^*(1)$ and $\xi_2^*(1)$ back to the master processor.
- Keep looping until $\xi_1^*(k) - \xi_2^*(k)$ is small enough, or we reach some chosen maximum number of iterations.

Computational Overhead

There is some computational overhead to spawn a Process on Python. The parallel processes in this problem are computationally cheap compared to this overhead; Agent 1 just needs to compute $\xi_1^*(k) = \lambda(k)/2$ while Agent 2 only needs to compute $\xi_2^*(k) = -\lambda(k)/2$ each iteration. Thus, it is faster to just carry out the processes in series.

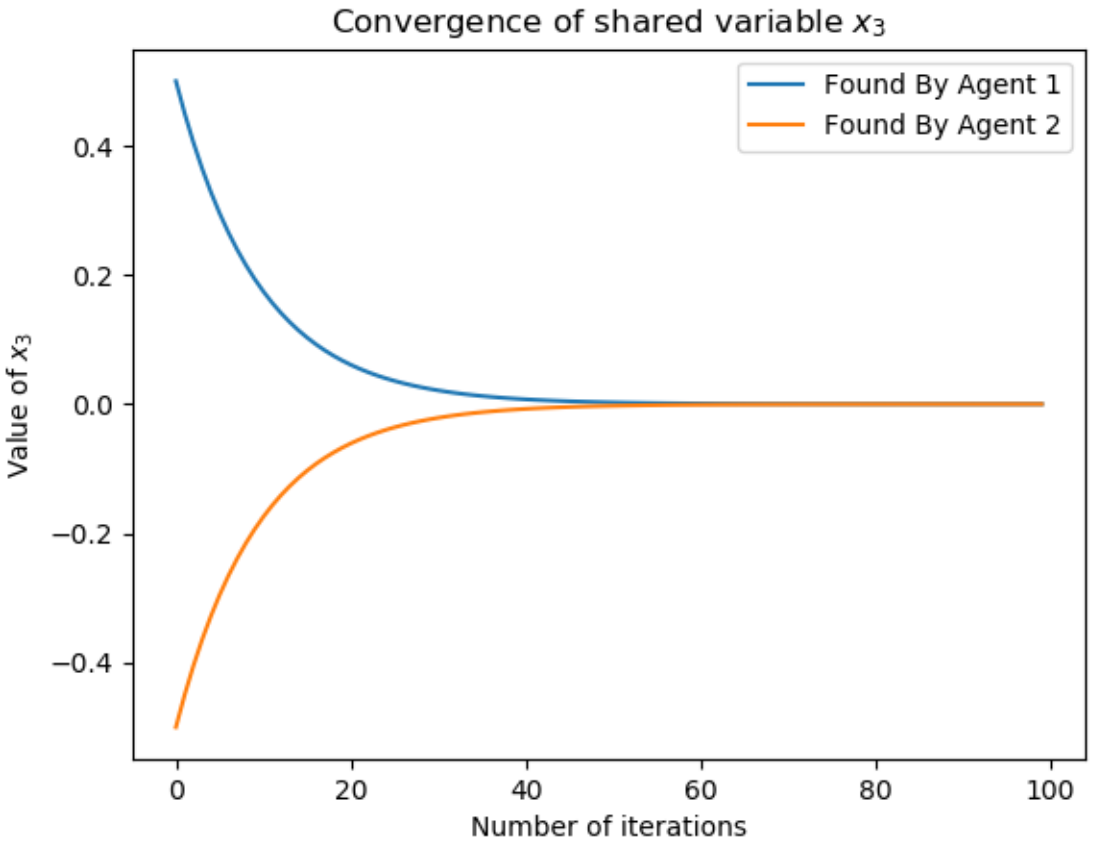
As a sanity check, let's simulate a more "difficult" problem, where the minimisers $\xi_1^*(k)$ and $\xi_2^*(k)$ may be more computationally expensive to compute. This can be done by introducing a for-loop that in the parallel processes so that on top of calculating $\xi_1^*(k)$ and $\xi_2^*(k)$, the agents must also loop through a large "dummy" calculation. When the "dummy" for-loop is made large enough (of the order 1000000 iterations), we start to see the serial method overtake the parallel method in terms of completion time.



For parallel computing to finish faster than serial computing, the cost of computation per process must be larger the computational cost of spawning that process. In terms of our optimisation problem, this means that the decomposed dual function must be decomposed into functions that are computationally complex, in order to observe any fruitful results from using parallel computing.

Convergence of Algorithm

We get convergence to $(x_1^*, x_2^*, x_3^*) = (0, 0, 0)$ as expected. It turns out that convergence occurs regardless of how the variable x_3^2 in the cost function is separated. Given $f = x_1^2 + x_2^2 + 2x_3^2$, convergence occurs for $f = [x_1^2 + 2ax_3^2] + [x_2^2 + 2(1-a)x_3^2]$ for all $a \in (0, 1)$. This will be observed in more detail, in Problem 2.



How to Run

The above examples

Make sure you are in the correct directory. Then to run the test that generated the above plots, execute the **main.py** file, i.e. use the command

>>>**python main.py**

Function Descriptions

The function **parallel.do_parallel**, description.

Syntax: do_general(max_iter,alpha,size_problem=0,verbose=False)

Parameter values:

- max_iter, Required. Number of iterations for the subgradient method.
- alpha, Required. Step size for the subgradient method.
- size_problem, Default 0. Number of iterations for the dummy for-loop.
- verbose, Default False. Print results to screen.

Outputs:

- Output 1. List containing ξ_1^* for all iterations of the subgradient method.
- Output 2. List containing ξ_2^* for all iterations of the subgradient method.
- Output 3. Completion time.

Closing Discussion for Problem 1

This simple problem provides a framework on using the Python **multiprocessing** module, which we can carry forward to the next problem. The expected computational overhead in spawning processes has been demonstrated. As expected, the parallel processing scheme is faster than the serial processing one only for proceses with large enough complexity. And as expected, decomposing the dual function and finding the minimisers as separate processes does not affect convergence to the solution of the original primal problem.

Problem 1.5

Before moving on to Problem 2, it is worth making a quick detour and coding up the subgradient method for a more general separable problem. Consider the following problem, where there is no assumption on the functions other than that they are convex. Recall that with one complicating variable, a separable problem can be formulated as

$$\begin{aligned} \min_{x_1,\xi_1,x_2,\xi_2} \quad & f_1(x_1,\xi_1) + f_2(x_2,\xi_2) \\ s.t. \quad & \xi_1 = \xi_2, \end{aligned}$$

where $x_1 \in \mathbb{R}^n$, $x_2 \in \mathbb{R}^m$ and $x_3, \xi_1, \xi_2 \in \mathbb{R}$ for some n and m.

Then we can decompose the dual function and apply the subgradient method

- At step $k = 0$, some master processor chooses an initial $\lambda(0) = 1.0$ and sends $\lambda(0)$ to Agents 1 and 2. Agent 1 calculates $(x_1(0), \xi_1(0)) \in \arg\min_{x_1,\xi_1}[f_1(x_1,\xi_1) - \lambda(0)(\xi_1)]$, while (in parallel) Agent 2 calculates $(x_2(0), \xi_2(0)) \in \arg\min_{x_2,\xi_2}[f_1(x_2,\xi_2) - \lambda(0)(\xi_2)]$. Agents 1 and 2 send $\xi_1^*(0)$ and $\xi_2^*(0)$ back to the master processor.

- At step $k = 1$, the master processor updates $\lambda(1) = \lambda(0) - \alpha(\xi_1^*(0) - \xi_2^*(0))$ and sends $\lambda(1)$ to Agents 1 and 2. Agent 1 calculates $(x_1(1), \xi_1(1)) \in \arg\min_{x_1,\xi_1}[f_1(x_1, \xi_1) - \lambda(1)(\xi_1)]$, while (in parallel) Agent 2 calculates $(x_2(1), \xi_2(1)) \in \arg\min_{x_2,\xi_2}[f_1(x_2, \xi_2) - \lambda(1)(\xi_2)]$. Agents 1 and 2 send $\xi_1^*(1)$ and $\xi_2^*(1)$ back to the master processor.
- Keep looping until $\xi_1^*(k) - \xi_2^*(k)$ is small enough, or we reach some chosen maximum number of iterations.

This holds for any cost function that can be decomposed into two convex functions, even a problem without a known gradient. Here is an example with $f_1(x_1, x_3) = 10|x_1| + x_3^2 + x_1 + |x_1 x_3| + x_3$ and $f_2(x_2, x_3) = x_3^2 + x_2^2$. The minimiser of the overall problem is $(0, 0, -0.25)$. No equation for their minimisers needs to be provided to the agents; Agent 1 and Agent 2 both use the python function in the **scipy** module **scipy.optimize.minimize** to calculate $(x_1(k), \xi_1(k))$ and $(x_2(k), \xi_2(k))$ respectively in each iteration of the subgradient method k.

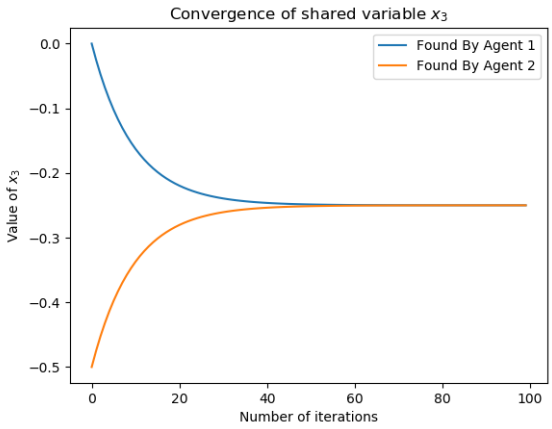


Figure 1: This plot can be generated by the command **python general.py**.

In fact, we can do the same for problems that can be separated into more than two functions (which will be done in Problem 3). But before looking into that, let's first move on to Problem 2, which demonstrates Outcome 2.

How to Run

The function **general.do_general**, description.

Syntax: do_general(max_iter,alpha,n,m,cost1,cost2,verbose=False)

Parameter values:

- max_iter, Required. Number of iterations for the subgradient method.

- alpha, Required. Step size for the subgradient method.
- n, Required. Length of vector x_1.
- m, Required. Length of vector x_2.
- fn1, Required. Cost function object that returns the value of f_1.
- fn2, Required. Cost function object that returns the value of f_2.
- verbose, Default False. Print results to screen.

Outputs:

- Output 1. List containing ξ_1^* for all iterations of the subgradient method.
- Output 2. List containing ξ_2^* for all iterations of the subgradient method.
- Output 3. Completion time.

Problem 2

Picking up where Problem 1 left off, Outcome 2 will now be demonstrated. As found in Problem 1, the complexity of the parallel processes needs to be increased in order to see the benefit of using parallel processing on the subgradient method.

An Example Quadratic Problem

Consider a similar problem to Problem 1, except we now have x_1 and x_2 as vectors. Consider a (convex) quadratic cost function, simply because we can write out the equation for its minimiser explicitly, and do not need to rely on any in-built solvers. In this problem with quadratic costs, the functions are differentiable and their gradients are linear, which makes for an easy demonstration.

The following is the separable unconstrained optimisation problem to be solved:

$$\min_{x,y,z} \qquad f_1(x,z) + f_2(y,z),$$

where $x \in \mathbb{R}^n$, $y \in \mathbb{R}^m$, $z \in \mathbb{R}$ and f_1 and f_2 are quadratic in $n+1$ and $m+1$ variables respectively. In other words, the function f_1 has the form $f_1 = \sum_{i=1}^{n} \sum_{j=i}^{n} \alpha_{ij} x_i x_j + \sum_{i=1}^{n} \beta_i x_i +$

$\gamma + \sum_{i=1}^{n} \delta_i z x_i + \epsilon z + \zeta z^2$, for some coefficients $\alpha_{ij}, \beta_i, \gamma, \delta_i, \epsilon, \zeta$. The function f_2 can be similarly defined, $f_2 = \sum_{i=1}^{m}\sum_{j=i}^{m} \omega_{ij} x_i x_j + \sum_{i=1}^{m} \psi_i x_i + \phi + \sum_{i=1}^{m} \tau_i z x_i + \sigma z + \mu z^2$, for some coefficients $\omega_{ij}, \psi_i, \phi, \tau_i, \sigma, \mu$.

Once again, z is a shared variable that can be accessed by all agents, while x and y are both vectors of private variables that can only be accessed by Agent 1 and Agent 2 respectively.

By applying a change of variables, the cost function can be separated and the new formulation of the problem will be

$$\begin{aligned} \min_{x,\xi_1,y,\xi_2} \quad & f_1(x,\xi_1) + f_2(y,\xi_2) \\ s.t. \quad & \xi_1 = \xi_2. \end{aligned}$$

The dual function will then be

$$\begin{aligned} q(\lambda) &= q_1(\lambda) + q_2(\lambda) \\ &= \inf_{x,\xi_1}[f_1(x,\xi_1) - \lambda\xi_1] + \inf_{y,\xi_2}[f_2(y,\xi_2) + \lambda\xi_2]. \end{aligned}$$

The minimisers can be found by finding the gradients and setting to zero, as follows:

$$\begin{aligned} \nabla[f_1(x,\xi_1) - \lambda\xi_1] &= \begin{bmatrix} 2\alpha_{11}x_1 + \sum_{i\neq 1} \alpha_{1i}x_i + \beta_1 + \delta_1\xi_1 \\ \vdots \\ 2\alpha_{nn}x_n + \sum_{i\neq n} \alpha_{ni}x_n + \beta_n + \delta_n\xi_1 \\ 2\zeta\xi_1 + (\epsilon - \lambda) + \sum_{i=1}^{n} \delta_i x_i \end{bmatrix} \\ &= \underbrace{\begin{bmatrix} 2\alpha_{11} & \dots & \alpha_{1n} & \delta_1 \\ & \vdots & & \\ \alpha_{1n} & \dots & 2\alpha_{nn} & \delta_n \\ \delta_1 & \dots & \delta_n & 2\zeta \end{bmatrix}}_{A_1} \underbrace{\begin{bmatrix} x_1 \\ \vdots \\ x_n \\ \xi_1 \end{bmatrix}}_{v_1} + \underbrace{\begin{bmatrix} \beta_1 \\ \vdots \\ \beta_n \\ (\epsilon - \lambda) \end{bmatrix}}_{B_1} = \mathbf{0} \end{aligned}$$

$$\nabla[f_2(y,\xi_2)+\lambda\xi_2] = \begin{bmatrix} 2\mu\xi_2 + (\sigma+\lambda) + \sum_{i=1}^{m}\tau_i y_i \\ 2\omega_{11}y_1 + \sum_{i\neq 1}\omega_{1i}y_i + \psi_1 + \tau_1\xi_2 \\ \vdots \\ 2\omega_{mm}x_m + \sum_{i\neq m}\psi_{mi}y_m + \omega_m + \tau_m\xi_2 \end{bmatrix}$$

$$= \underbrace{\begin{bmatrix} 2\mu & \tau_1 & \dots & \tau_m \\ \tau_1 & 2\omega_{11} & \dots & \omega_{1m} \\ & & \vdots & \\ \tau_m & \omega_{1m} & \dots & 2\omega_{mm} \end{bmatrix}}_{A_2} \underbrace{\begin{bmatrix} \xi_2 \\ y_1 \\ \vdots \\ y_m \end{bmatrix}}_{v_2} + \underbrace{\begin{bmatrix} (\sigma+\lambda) \\ \psi_1 \\ \vdots \\ \psi_m \end{bmatrix}}_{B_2} = \mathbf{0}$$

The minimiser of $[f_1(x,\xi_1)-\lambda\xi_1]$ is then the solution to the set of $n+1$ linear equations and the minimiser of $[f_2(y,\xi_2)+\lambda\xi_2]$ is the solution to the set of $m+1$ linear equations above. Note, the matrices A_1 and A_2 are symmetric and must be invertible, i.e., their eigenvalues must be strictly positive; the problem is ill-conditioned if the eigenvalues are close to zero.

The same method of using the subgradient and dual decomposition from Problem 1 now applies to this problem.

- At step $k=0$, some master processor chooses an initial $\lambda(0)=1.0$ and sends $\lambda(0)$ to Agents 1 and 2. Agent 1 solves $A_1v_1(0)+B_1(0)=0$ for $v_1(0)$, while (in parallel) Agent 2 solves $A_2v_2(0)+B_2(0)=0$ for $v_2(0)$. Agents 1 and 2 send $\xi_1^*(0)$ and $\xi_2^*(0)$ back to the master processor.
- At step $k=1$, the master processor updates $\lambda(1)=\lambda(0)-step(\xi_1^*(0)-\xi_2^*(0))$ and sends $\lambda(1)$ to Agents 1 and 2. Agent 1 solves $A_1v_1(1)+B_1(1)=0$ for $v_1(1)$, while (in parallel) Agent 2 solves $A_2v_2(1)+B_2(1)=0$ for $v_2(1)$. Agents 1 and 2 send $\xi_1^*(1)$ and $\xi_2^*(1)$ back to the master processor.
- Keep looping until $\xi_1^*(k)-\xi_2^*(k)$ is small enough, or we reach some chosen maximum number of iterations.

Combined Problem

At this stage it is useful to observe the problem without separation of the dual function, call it the combined problem. The problem is quadratic, and the minimiser is found by finding the gradient, and ultimately solving $n+m+1$ linear equations. This means solving the problem $Av=B$, with A and B found by augmenting and overlapping A_1, A_2 and B_1, B_2 respectively. This is visualised below.

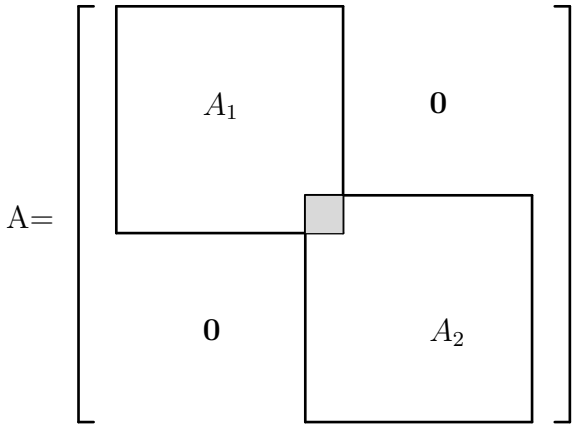


Figure 2: Overlapping squares represent how A_1 and A_2 overlap. There is a single element in the grey overlapping region and it is equal to $2\zeta + 2\mu$, the sum of the elements of A_1 and A_2 that overlap.

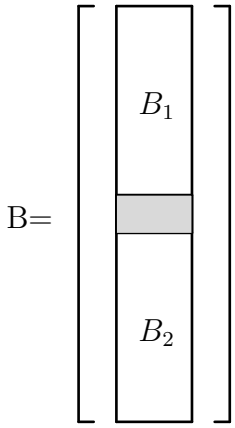


Figure 3: Similarly, B is a vector with B_1 and B_2 overlapping as shown. Only one element is in the grey overlapping region, and that element is equal to $\epsilon + \sigma$, the sum of the elements of B_1 and B_2 that overlap.

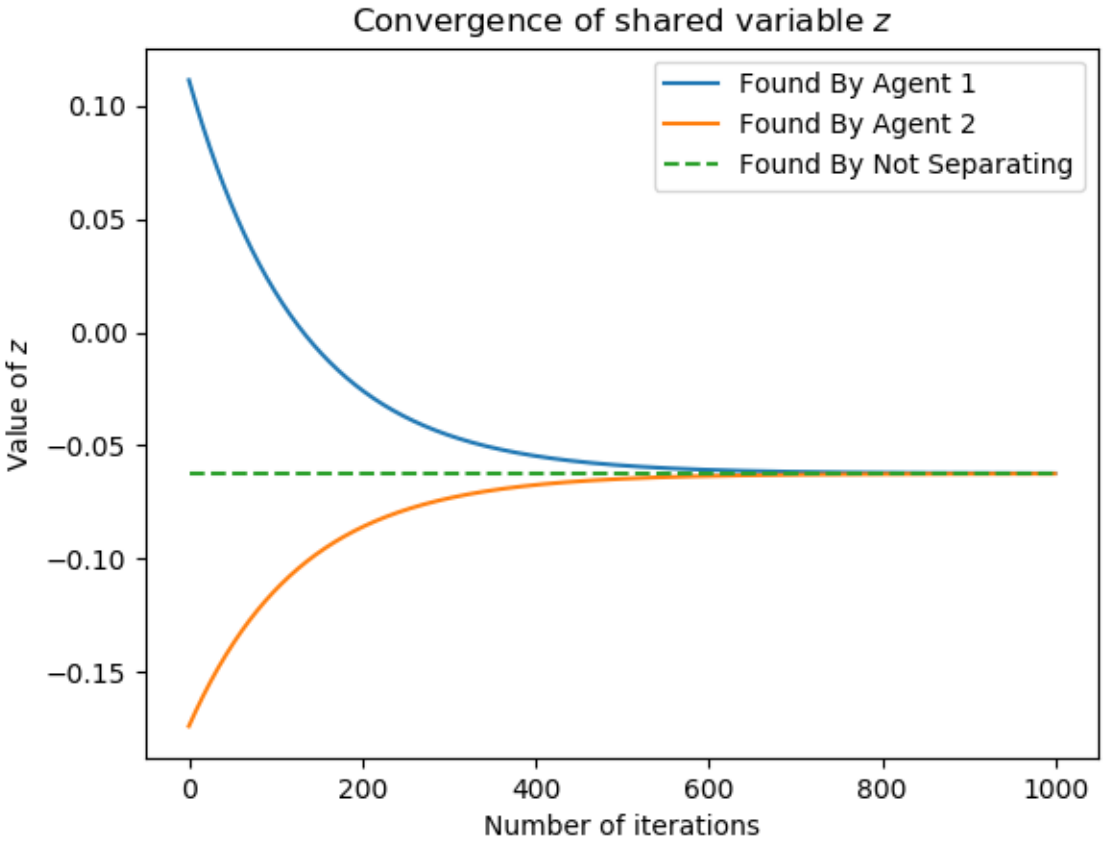
We can solve the primal problem $\min_{x,y,z} f_1(x,z) + f_2(y,z)$ directly by solving $Av + B = 0$. This primal solution can be used to check convergence of solution via dual decomposition.

Convergence

As a reminder, in separating the cost function, it is now a requirement that both A_1 and A_2 are well-conditioned in order to get convergence to the solution found via the combined problem. The combined matrix A may be well conditioned, but depending on how A_1 and A_2 are separated, we may end up with an ill-conditioned matrices.

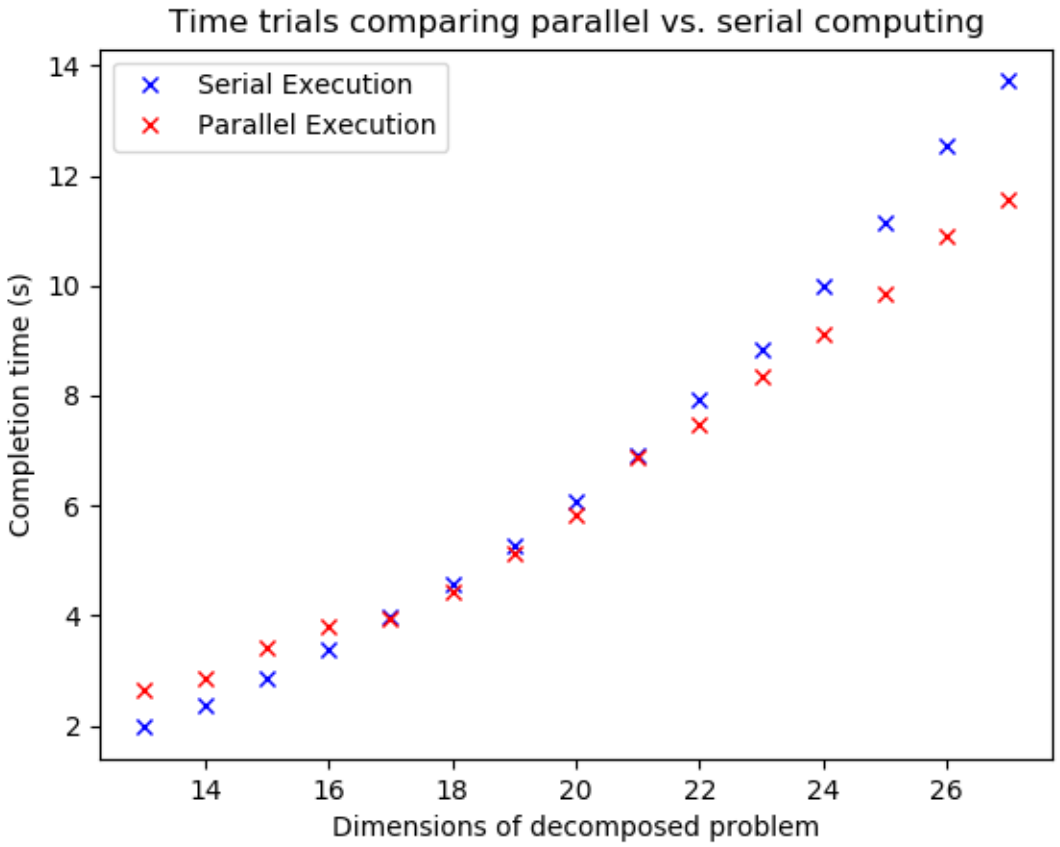
This picks up from where Problem 1 left off. In Problem 1, it was incorrectly concluded that convergence was still guaranteed regardless of how the shared variable x_3^2 was separated. It turns out that this does not hold for coefficients of x_3^2 that are too close to zero. If the coefficient of x_3^2 is too small, convergence no longer occurs. Thus, we see the key requirement that the matrices are well-conditioned appearing in Problem 1 as well.

The following plot shows convergence of the shared variable ξ_1 found by Agent 1 and the variable ξ_2 found by Agent 2 towards the actual value of z found by solving $Av + B = 0$ in the combined problem. Some care is taken to generate well-conditioned A_1 and A_2 matrices.



Computational Overhead

There is some computational overhead to spawn a Process on Python. Previously in Problem 1, a dummy for-loop was added to increase the complexity of each parallel process. For Problem 2, we achieve the same affect by increasing n and m, i.e., increasing the dimensions of A_1 and A_2. The following plot shows the completion times, where the dimensions of A_1 and A_2 have been set to be equal, $n = m$.



In Problem 1, the completion times for both serial and parallel grew linearly with the size of the "dummy" for-loop. In Problem 2, the solution to $A_1x + B_1$ and $A_2x + B_2$ were found by applying Gaussian elemination, which has a complexity of order $\mathcal{O}(n^3)$. The sample plot above may not be enough to conclude cubic growth as the dimensions of the problem grow, but it can be seen that the growth is of order higher than linear growth.

How to Run

The above examples

Make sure you are in the correct directory. Then to run the test that generated the above plots, execute the **main.py** file, i.e. use the command

>>>**python main.py**

Function Descriptions

The function **parallel.do_parallel**, description.

Syntax: do_parallel(max_iter,alpha,A1,A2,b1,b2,verbose=False)

Parameter values:

- max_iter, Required. Number of iterations for the subgradient method.
- alpha, Required. Step size for the subgradient method.
- A1, Required. The matrix of coefficients A_1 as described above.
- A2, Required. The matrix of coefficients A_2 as described above.
- b1, Required. The matrix of coefficients B_1 as described above.
- b2, Required. The matrix of coefficients B_2 as described above.
- verbose, Default False. Print results to screen.

Outputs:

- Output 1. List containing ξ_1^* for all iterations of the subgradient method.
- Output 2. List containing ξ_2^* for all iterations of the subgradient method.
- Output 3. Completion time.

Closing Discussion for Problem 2

In Problem 2, we say for the case of problems whose cost functions can be separated into two functions that we observe as expected Outcome 1 and Outcome 2. As an additional observation, the order of complexity of the oracle that returns the minimisers will determine the order of growth in completion time, and the minimisers must be sufficiently computationally complex in order to see parallel processing finish faster than serial processing.

Problem 3

Given Problem 1 and 2 both dealt with only two agents computing two processes in parallel, this presents a good opportunity to move on to a problem requiring a much larger number of agents, i.e., a demonstration of Outcome 3.

In addition, we can investigate Outcome 4 and compare the performance of hardware. All previous results have been generated on a 2.90GHz Intel i7-7500U CPU with 2 cores (4 logical processors). To demonstrate Outcome 4, a 3.30GHz Intel i5-6600 CPU with 4 cores will be used to compare against the benchmark completion times set by the i7 CPU.

Problem 3 revisits a problem from a previous assignment in this subject. It goes as follows, from the ELEN90026-Introduction to Optimisation: Homework 3 assignment brief verbatim.

Consider a network with n nodes and m directed edges. Let A be the incidence matrix of the network where its ij-th element is

$$A_{ij} = \begin{cases} -1 & \text{Edge } j \text{ enters node } i \\ 1 & \text{Edge } j \text{ exits node } i \\ 0 & \text{otherwise.} \end{cases}$$

Assume that two commodities flow through the network. Let s_i and t_i be the supply/demand of commodities 1 and 2 at node i, respectively. A positive value corresponds to supply and a negative value corresponds to demand. Let vectors $s \in \mathbb{R}^n$ and $t \in \mathbb{R}^n$ be obtained by stacking all s_i and t_i, $i = 1, ..., n$. Let $f_i : \mathbb{R}^2 \to \mathbb{R}$ be a convex function associated with the cost of transporting commodities 1 and 2 along edge i.
The problem formulation:

$$\begin{aligned} \min_{x \in \mathbb{R}, y \in \mathbb{R}} \quad & \sum_{i=1}^{m} f_i(x_i, y_i) \\ s.t. \quad & Ax = s, \quad Ay = t \\ & x \geq 0, \quad y \geq 0 \end{aligned}$$

The cost function used in this problem will be $f_i(x_i + y_i) = (x_i + y_i)^2 + 0.1(x_i^2 + y_i^2)$, for $i = 1, ..., m$. Let's allow the vectors s and t to be dense, they do not have to be sparse.

Now, the point of revisiting this problem is to apply parallel processing and observe an expected speed up in completion time versus serial processing. The other interesting observation we want to make here is to compare the completion times of solving this problem 2 core 2.90GHz Intel i7-7500U CPU versus running it on a 3.30GHz 4 core Intel i5-6600. But before going into that, the following describes how to obtain a solution to the optimisation problem.

In contrast to Problem 2, there are now constraint equations in the formulation, where previously Problem 2 was unconstrained. In addition, there are no shared or complicating variables in the cost function problem, it is completely separable. But the method we are

going to use to solve Problem 3 will be similar to before. The dual function is

$$q(\alpha, \beta) = \inf \left(\sum_{i=1}^{m} f_i(x_i, y_i) - \alpha^\top (Ax - s) - \beta^\top (Ay - t) \right)$$
$$= \alpha^\top s + \beta^\top t + \inf \left(\sum_{i=1}^{m} f_i(x_i, y_i) - [A^\top \alpha]_i x_i - [A^\top \beta]_i y_i \right)$$

Thus, for $i = 1, ..., m$, for a given α and β, we can let m individual agents find the minimisers of $f_i(x_i, y_i) - [A^\top \alpha]_i x_i - [A^\top \beta]_i y_i$ separately, then use the subgradient to update $\alpha = \alpha - step(Ax - s)$ and $\beta = \beta - step(Ay - t)$. Note, because there are also constraints on x, y being non-negative, the minimisers found by each agent must be constrained to the non-negative orthant.

Convergence

For guaranteed convergence, the graph must be connected, i.e., the incidence matrix must have rank $n - 1$. Note, the chosen step size, *step* for updating α and β must decrease as m increases because a larger number of columns of A means more elements are being summed up when doing Ax and Ay.

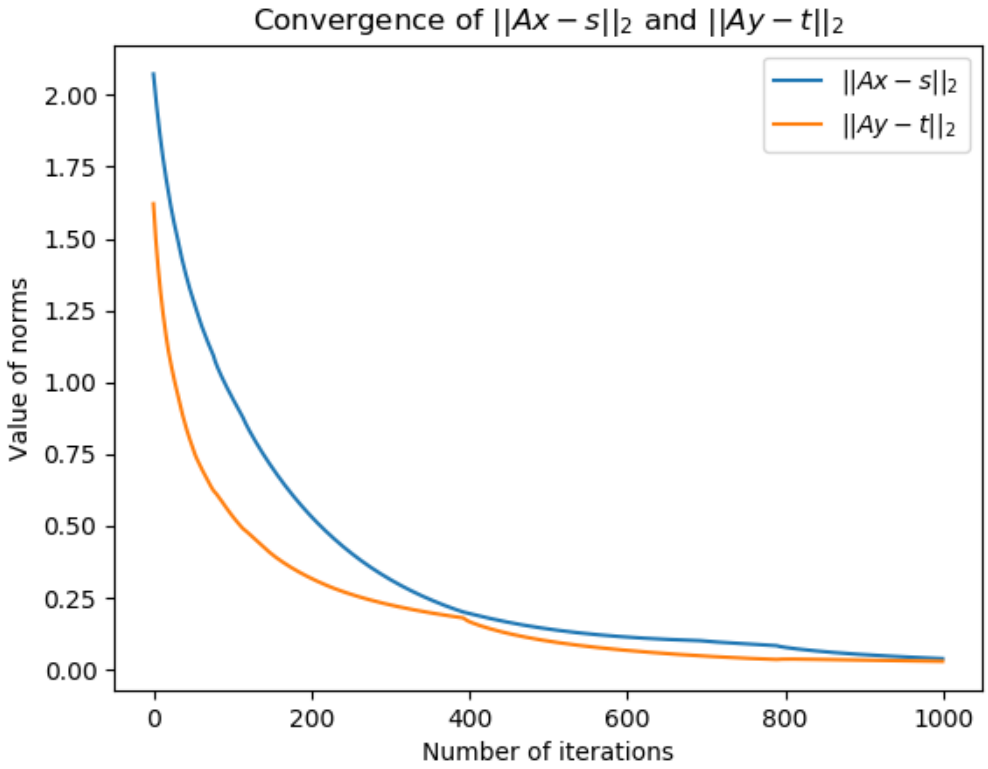


Figure 4: A problem with a scale-free graph with 20 nodes.

This problem has been done in the previous assignment, so we will not dwell on convergence.

Computational Overhead

For this problem, we are trying to simulate m agents each computing 1 process in parallel. This is not possible with the i7 dual-core CPU that can run at most 4 processes in parallel. The closest we can get is to use the Pool object in Python's multiprocessing module and the method Pool.map. The m processes are put into a pool, then offloaded to workers, which will iterate through all the tasks in the pool and compute them in parallel. With 4 logical processors, we have a maximum of 4 workers, so only 4 processes can be carried out in parallel. The following plot compares the completion time of parallelising with the Pool object versus solving the problem in series.

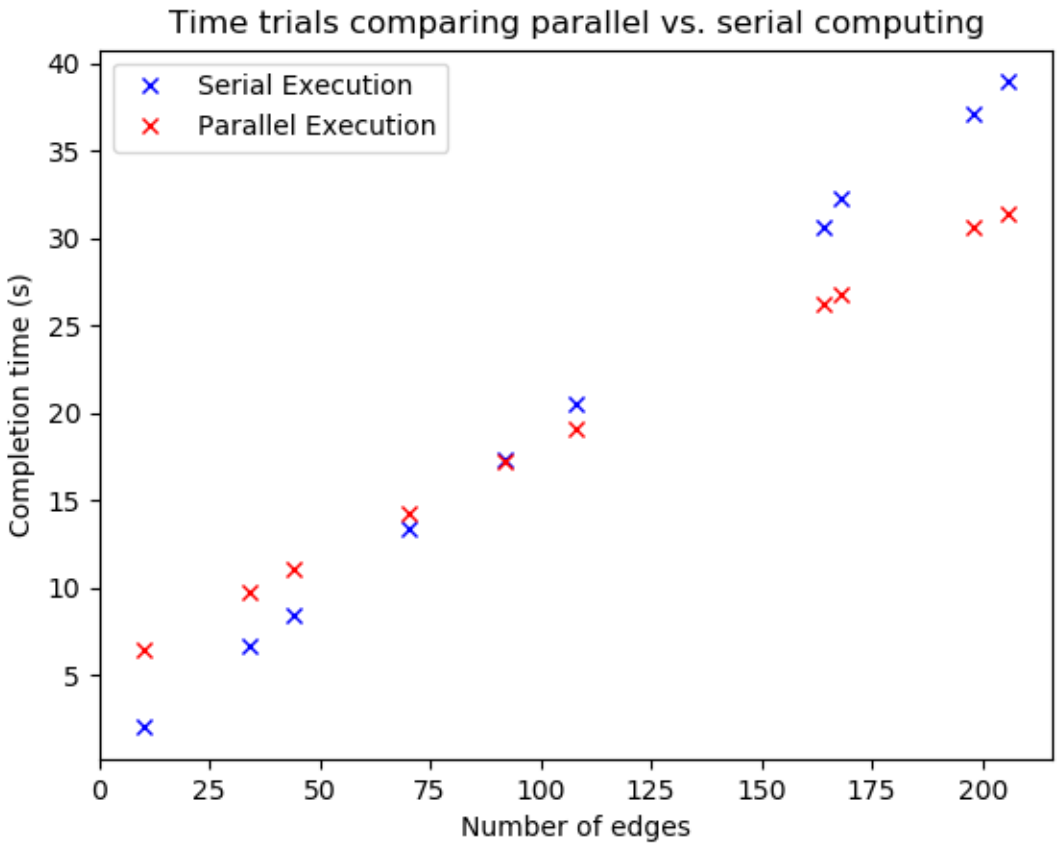


Figure 5: Run on 2.90GHz Intel i7-7500U CPU with 2 cores (4 logical processors).

As expected, for a small problem size, the overhead of spawning processes in parallel means that parallel processing does worse than serial processing in terms of completion

times. However, solving the problem by parallel computing becomes faster than solving via serial computing when the problem size increases. But we've seen this in Problem 1 and 2 already. It is more interesting to observe the same code run on a different machine.

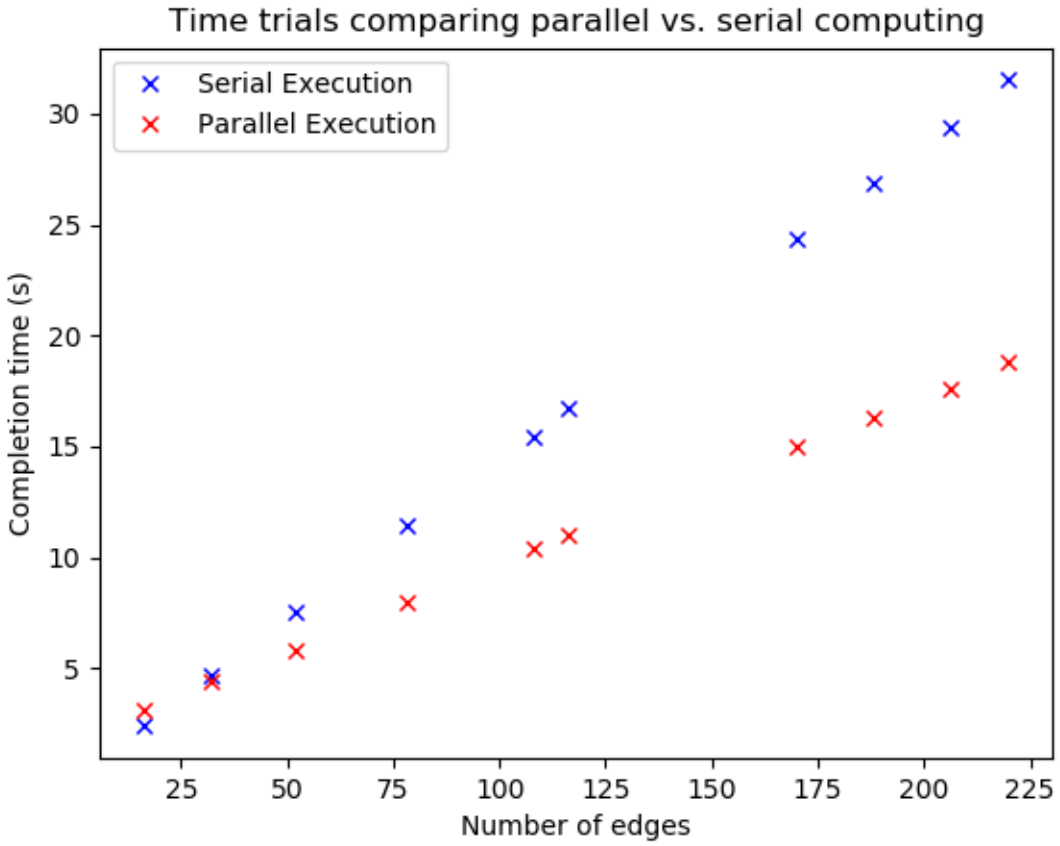


Figure 6: Run on 3.30GHz Intel i5-6600 CPU with 4 cores (4 logical processors).

The i7 is a 7th generation mobile CPU whereas the i5 is a 6th generation desktop CPU, which makes comparing them tricky. The i7 performs slightly better in terms of serial processing, but not by very much. The real benefit of having more cores in terms of parallel processing is demonstrated by the i5 solving the problem in parallel faster than the i7. So although the i7's cores are each more powerful and each core can supposedly run 2 logical processes in parallel, in reality, for the simple processes in this example problem, physically having 4 (weaker) cores solves the problem much faster.

How to Run

The above examples

Make sure you are in the correct directory. Then to run the test that generated the above plots, execute the **main.py** file, i.e. use the command

>>>**python main.py**

Function Descriptions

The function **parallel.do_parallel**, description.

Syntax: do_parallel(max_iter,step_size,incidence_matrix,s,t,verbose=False)

Parameter values:

- max_iter, Required. Number of iterations for the subgradient method.
- alpha, Required. Step size for the subgradient method.
- incidence_matrix, Required. The incidence matrix of a connected graph as described above.
- s, Required. Vector of supply/demand of commodity 1 of every node. Elements of s must sum to 0.
- t, Required. Vector of supply/demand of commodity 2 of every node. Elements of t must sum to 0.
- verbose, Default False. Print results to screen.

Outputs:

- Output 1. List containing $||Ax - s||_2$ for all iterations of the subgradient method.
- Output 2. List containing $||Ay - t||_2$ for all iterations of the subgradient method.
- Output 3. Completion time.

Closing Discussion for Problem 3

Along the lines of the above experiment, it may be of interest to investigate the effect of increasing the complexity of each process on the completions times of the i7 and the i5 CPUs. If the oracles (that return the minimisers) increase in complexity, then more powerful cores may become more beneficial than more numerous cores.

Problem 4

Let's take a look at Outcome 5. We want to investigate the effect of lower precision in messages sent between agents on convergence of the subgradient method. Assume that messages are sent as binary numbers with a limited number of bits. We first consider problems with imprecision in the dual variables, λ.

A Convergence Analysis for Imprecise λ

Consider separable problems that can be separated into p functions, and with q complicating variables.

$$\begin{aligned} \min_{x} \quad & f(x) = \sum_{i=1}^{p} f_i(x_i, \xi_i) \\ s.t. \quad & c_i(x) = \xi_i - \xi_{i+1} = 0, \quad i = 1, ..., q-1, \end{aligned}$$

where x is a vector containing all the primal variables x_i and ξ_i. Then the dual function is

$$q(\lambda) = \inf(f(x) - \lambda^\top c(x))$$

Model the rounding error in λ by having the update step of the subgradient method be

$\lambda_{k+1} = \lambda_k - \delta_k - \alpha g_k,$

where δ_k is the rounding error and g_k is the subgradient at the kth iteration. We will need the following relationship, which is derived from the definition of a subgradient:

$q(\lambda^*) \leq q(\lambda) + g(\lambda)^\top(\lambda - \lambda^*)$

Then, we can use some analysis of the convergence of the stochastic gradient method.

(Source: "Lecture 2: Subgradient Methods", John C. Duchi. Stanford University)

$$\begin{aligned}\frac{1}{2}\|\lambda_{k+1}-\lambda^*\|^2 &= \frac{1}{2}\|\lambda_k-\delta_k-\alpha g_k-\lambda^*\|^2 \\ &= \frac{1}{2}\|\lambda_k-\lambda^*\|^2-\alpha g_k^\top(\lambda_k-\lambda^*)+\frac{1}{2}\|\delta_k+\alpha g_k\|^2-\delta_k^\top(\lambda_k-\lambda^*) \\ &\leq \frac{1}{2}\|\lambda_k-\lambda^*\|^2-\alpha(q(\lambda_k)-q(\lambda^*))+\frac{1}{2}\|\delta_k+\alpha g_k\|^2-\delta_k^\top(\lambda_k-\lambda^*)\end{aligned}$$

Rearranging,

$$\alpha(q(\lambda_k)-q(\lambda^*)) \leq \frac{1}{2}\|\lambda_k-\lambda^*\|^2-\frac{1}{2}\|\lambda_{k+1}-\lambda^*\|^2+\frac{1}{2}\|\delta_k+\alpha g_k\|^2-\delta_k^\top(\lambda_k-\lambda^*)$$

$$\sum_{k=1}^{K}\alpha(q(\lambda_k)-q(\lambda^*)) \leq \frac{1}{2}\|\lambda_1-\lambda^*\|^2+\sum_{k=1}^{K}\frac{1}{2}\|\delta_k+\alpha g_k\|^2-\sum_{k=1}^{K}\delta_k^\top(\lambda_k-\lambda^*)$$

We have to make an assumption at this stage. Assume $E(\delta_k^\top(\lambda_k-\lambda^*))=0$, i.e., the rounding error has mean zero. Then,

$$\mathbf{E}\sum_{k=1}^{K}(q(\lambda_k)-q(\lambda^*)) \leq \tfrac{1}{2\alpha}\|\lambda_1-\lambda^*\|^2+\sum_{k=1}^{K}\tfrac{\alpha}{2}\mathbf{E}\|\tfrac{\delta_k}{\alpha}+g_k\|^2$$

So we get an (expected) convergence of the dual function value when there is imprecision of messages passed between agents.

Word Limit

The following terms will be used to describe a binary number.

$$\underbrace{\pm}_{\text{sign}}\underbrace{10101010101010}_{\text{binary integer}}\underbrace{.}_{\text{binary point}}\underbrace{1010101010101010}_{\text{binary fraction}}$$

Define the word limit as the limit on the number of bits used to represent the binary fraction of a message, not including the sign and integer bits.

Example Problem

Revisit Problem 2, where the optimisation problem was unconstrained and had one shared variable. Recall that the dual function was given by

$$\begin{aligned}q(\lambda) &= q_1(\lambda)+q_2(\lambda) \\ &= \inf_{x,\xi_1}[f_1(x,\xi_1)-\lambda\xi_1]+\inf_{y,\xi_2}[f_2(y,\xi_2)+\lambda\xi_2].\end{aligned}$$

Dual decomposition and the subgradient method were applied to solve this problem, where we iteratively use a given λ to solve two separate sets of linear equations, $A_1 v_1 = B_1$ and $A_2 v_2 = B_2$, and obtain ξ_1 and ξ_2, then use ξ_1 and ξ_2 to update λ.

Consider now three agents Agent 1, Agent 2 and Agent 3. Agent 3 computes the updates $\lambda = \lambda - \alpha(\xi_1 - \xi_2)$, then sends λ to both Agent 1 and 2. Agents 1 and 2 respectively solve $A_1 x_1 = B_1$ and $A_2 x_2 = B_2$, then send ξ_1 and ξ_2 to Agent 3. This is repeated for some specified number of iterations. The figure below shows one iteration of this.

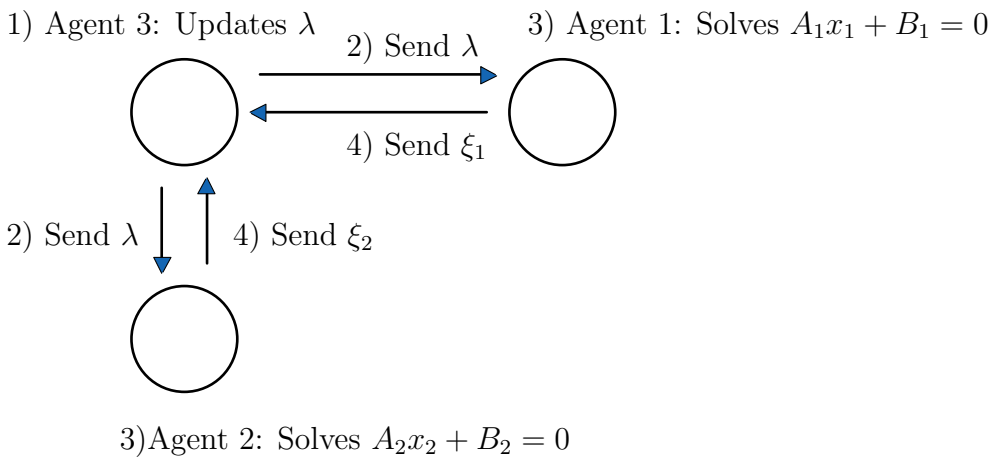


Figure 7: Steps 1, 2, 3, 4 show the procedure in a single iteration of the subgradient method.

Previously, messages were assumed to have unlimited size, i.e., there is no floating point error when λ was sent from Agent 3 to Agent 1 and 2. Let's now consider a limit on message size. Assume messages are sent in binary, then there is a limit on the number of bits that can be sent.

In order to measure convergence, we need a benchmark. Recall from Problem 2 the combined problem, where A_1 and A_2 are overlapped to form A, and B_1 and B_2 overlapped to form B. The combined problem $Av + B = 0$ can be solved, and we can use the vector v as a benchmark to measure convergence. Let v^* be the solution to the problem $Av + B = 0$, let v_1^* be the solution to the problem $A_1 v_1 + B_1 = 0$ and let v_1^* be the solution to the problem $A_2 v_2 + B_2 = 0$. The dimensions do not match, so let's overlap v_1^* and v_2^*

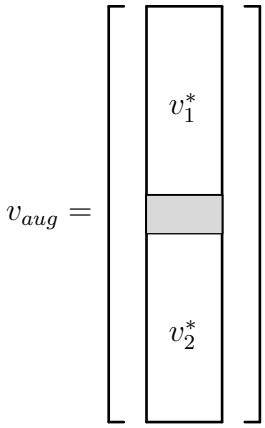


Figure 8: Create v_{aug}, a vector with v_1^* and v_2^* overlapping as shown. Only one element is in the grey overlapping region, and that element is equal to the **mean** of the elements of v_1^* and v_2^* that overlap.

Finally, our measure of convergence will be $||v_{aug} - v^*||_2$.

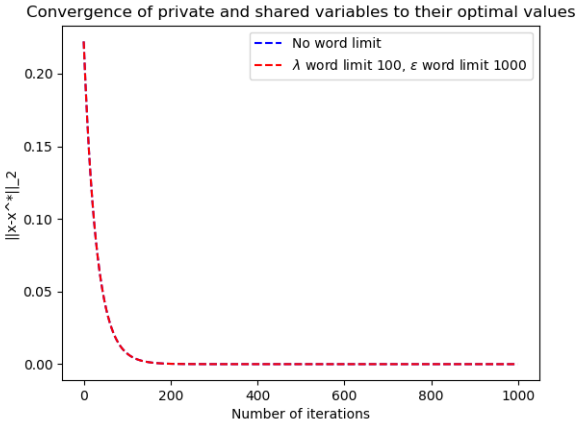


Figure 9: Large word limits.

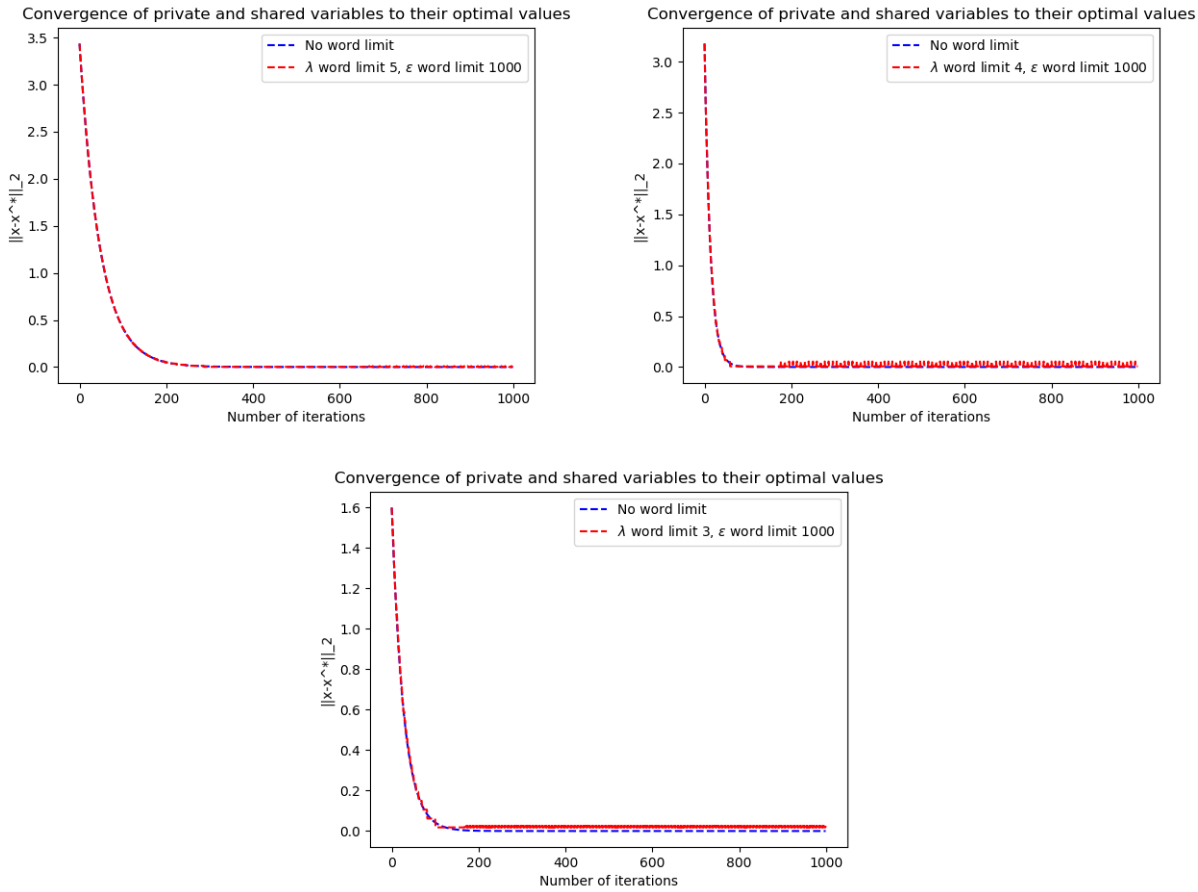


Figure 10: Decreasing the word limit on λ.

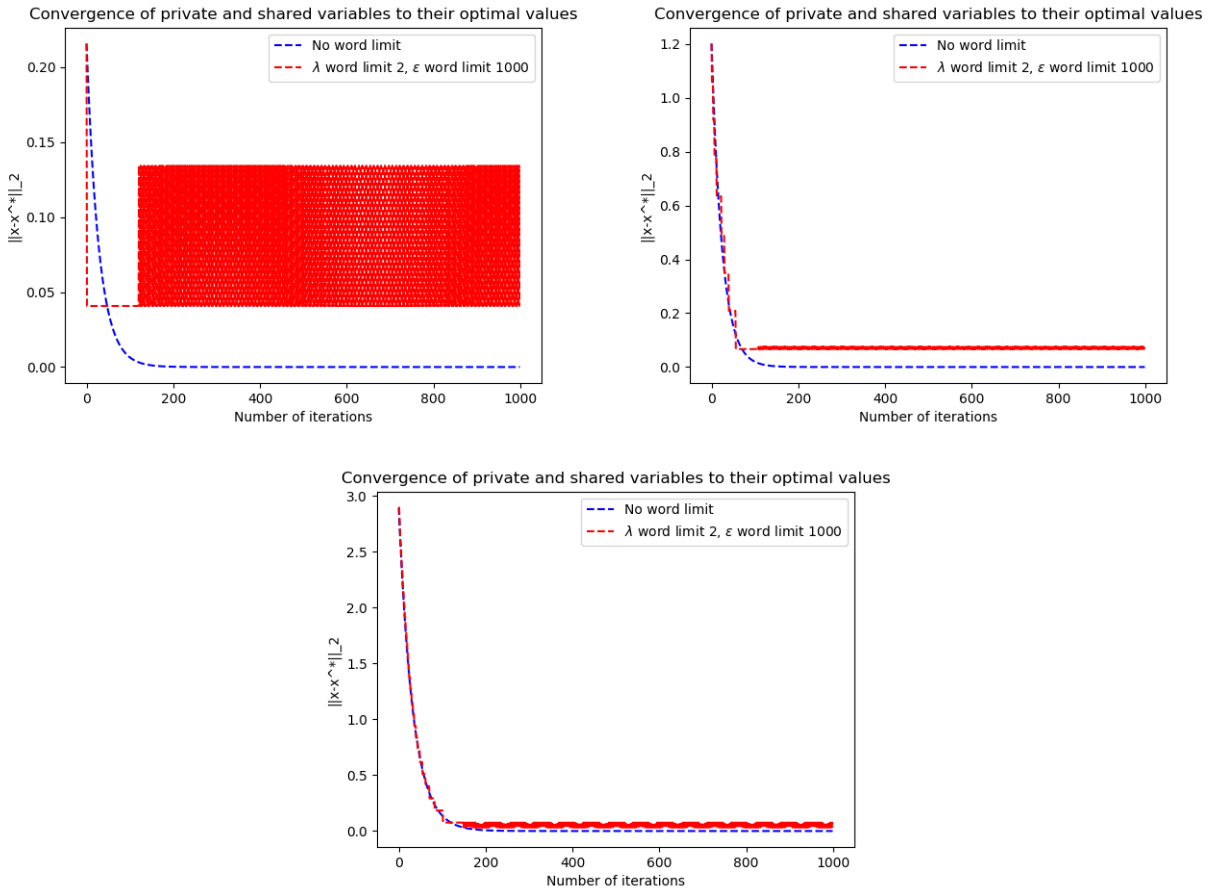


Figure 11: Convergence for a given limit on λ is somewhat dependent on the coefficients of the problem.

For low precision on λ, we enter some limit cycle behaviour when rounding λ to the closest limited-bit binary representation then recalculation ξ, and then reupdating λ, etc. In the plots below, we take a closer look at a specific problem, plotting the steady state behaviour (the last 50 iterations out of 1000) of λ. Three different λ values are shown, λ from the precise problem where no word limit was applied, λ from the imprecise problem with a word limit of 3, λ of the imprecise problem rounded down to the nearest binary number.

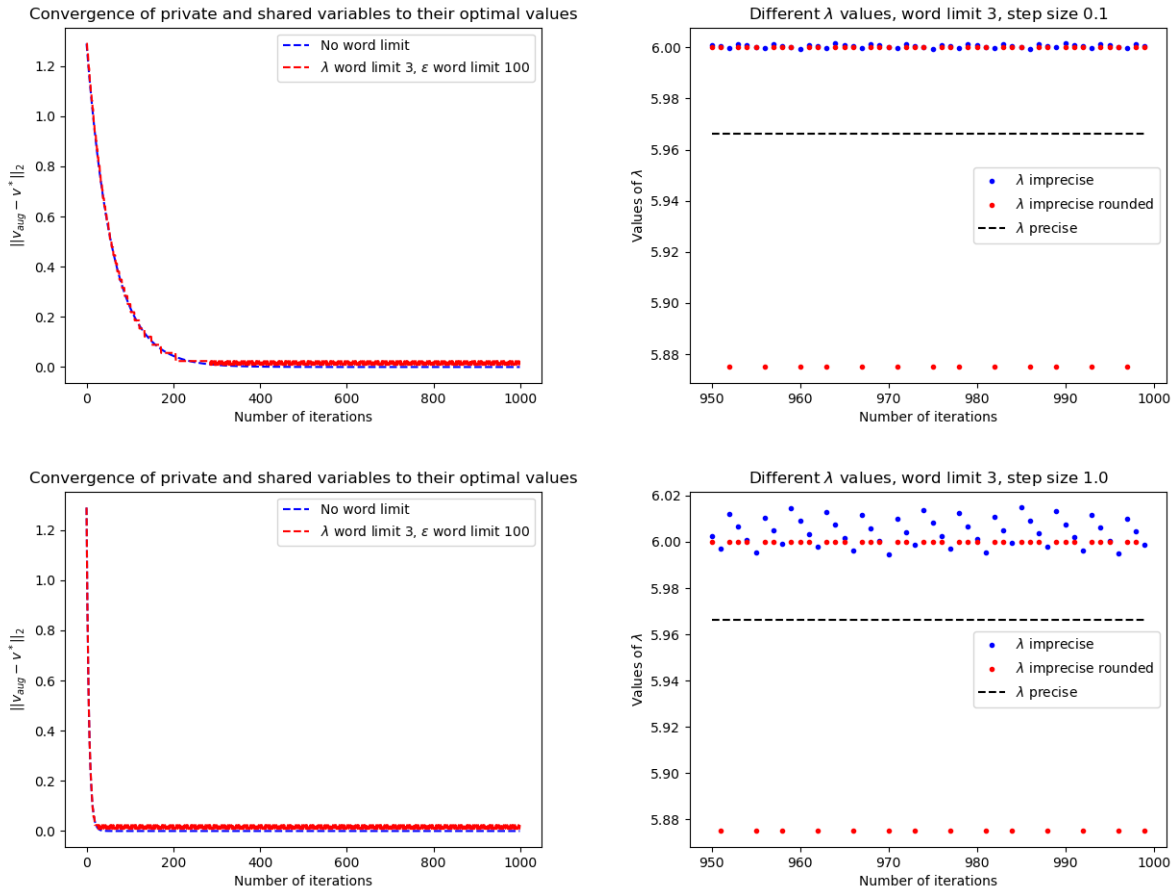


Figure 12: Cyclic behaviour of λ when its true value is 5.966, which lies beetween 6.000 and 5.875.

Imprecise ξ_1 and ξ_2

Let's now consider imprecise messages sent from Agent 1 and Agent 2 to Agent 3.

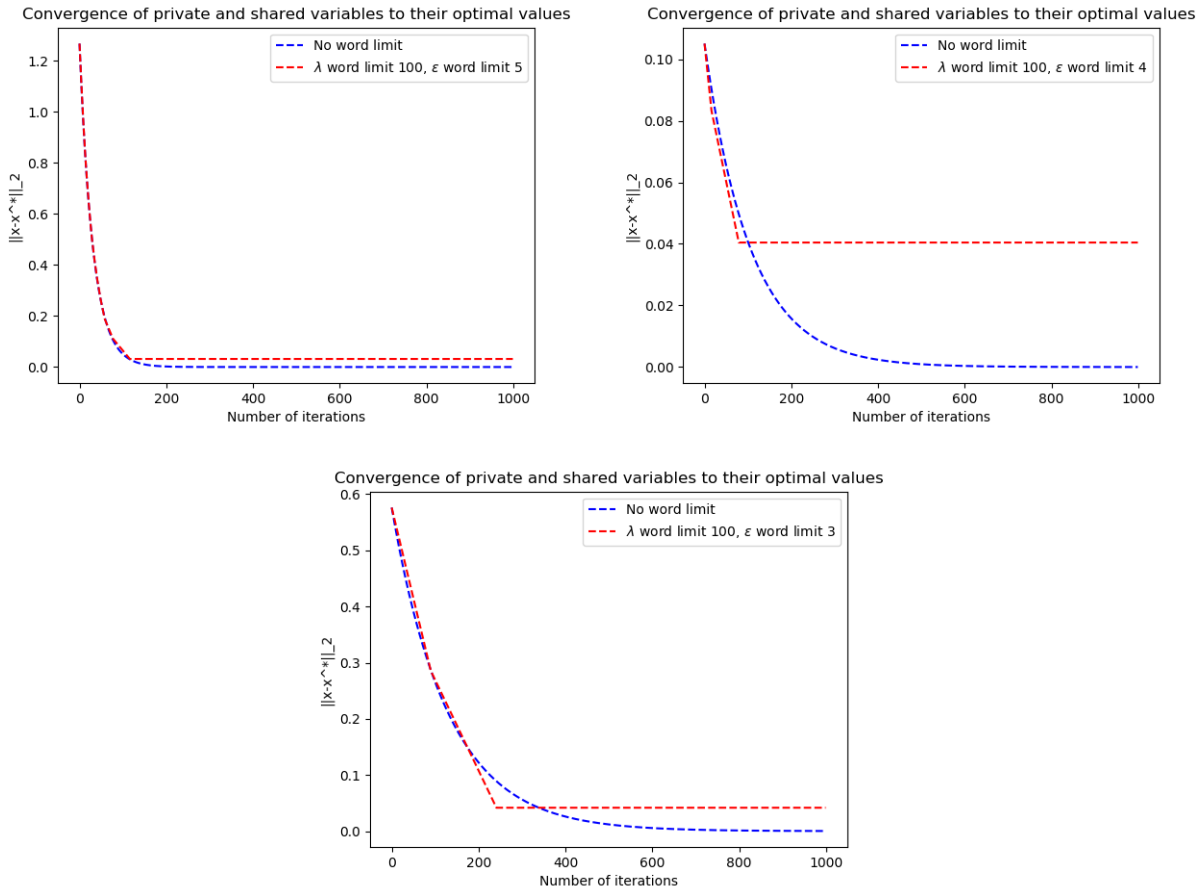


Figure 13: Convergence for a given limit on ξ is also somewhat dependent on the randomly generated the problem.

With low precision on ξ the error appears to be constant in steady state. In the case of low precision on ξ, the constant error is the rounding error caused by rounding ξ to the closest limited-bit binary representation.

Perturbation Analysis

If we formulate the primal problem with low precision on the shared variables as

$$\begin{aligned} \min_{x} \quad & f(x) = \sum_{i=1}^{p} f_i(x_i, \xi_i) \\ s.t. \quad & c_i(x) = \xi_i - \xi_{i+1} = \delta, \quad i = 1, ..., q-1, \end{aligned}$$

then from the perturbation analysis of constrained problems, we know that the dual variables λ can be used to approximate the perturbation in the primal function value, $\frac{\partial p^*}{\partial \delta} = \lambda^*$. We

know that δ will be less than or equal to the smallest binary fraction bit available. Let's consider this problem

$$\begin{aligned} \min_{x} \quad & f(x) = f_1(x_1, \xi_1) + f_2(x_2, \xi_2) \\ s.t. \quad & c_i(x) = \xi_1 - \xi_2 = \delta, \end{aligned}$$

where $f_1(x_1, x_3) = 10|x_1| + x_3^2 + x_1 + |x_1 x_3| + 0.1x_3$ and $f_2(x_2, x_3) = x_2^2 + x_3^3$. The primal value $P^* = -0.00125$, and the minimiser is at $(x_1, x_2, \xi_1, \xi_2) = (0, 0, -0.025, -0.025)$. Running **perturb.py** displays

```
Perturbed primal value = -0.000547
Exact primal value = -0.001249
Lambda star = 0.050025
Perturbation/xi word limit = 6
```

And we can confirm that $p(\delta) - p^* = -0.000547 + 0.00125 = 0.000703 \approx \lambda^* \delta = 0.05(2^{-6}) = 0.00078125$.

How to Run

The above examples

Make sure you are in the correct directory. Then to run the tests that generated the above plots, execute the **main.py** file, i.e. use the command

>>>**python main.py**

Vary lambda_word_limit and eps_word_limit to generate plots with different binary message limits.

Function Descriptions

The function **conversions.float2bin**, description.

Syntax: float2bin(number, places)

Parameter values:

- number, Required. A float variable that is to be converted to binary representation.
- places, Required. Number of bits used to represent the binary fraction of the input number. There is no limit on the number of bits used to represent the binary integer of the input number.

Outputs:

- Output 1. Binary string representation of the input number.

The function **conversions.bin2float**, description.

Syntax: bin2float(bin_str)

Parameter values:

- bin_str, Required. A binary string representation that is to be converted to float.

Outputs:

- Output 1. Float representation of input binary string.

The function **imprecise.do_imprecise**, description.

Syntax: do_imprecise(max_iter,alpha,A1,A2,b1,b2,xstar,

lambda_word_limit,xi_word_limit,verbose=False)

Parameter values:

- max_iter, Required. Number of iterations for the subgradient method.
- alpha, Required. Step size for the subgradient method.
- A1, Required. The matrix of coefficients A_1 as described above.
- A2, Required. The matrix of coefficients A_2 as described above.
- b1, Required. The matrix of coefficients B_1 as described above.
- b2, Required. The matrix of coefficients B_2 as described above.
- xstar, Required. True solution to the problem.
- lambda_word_limit, Required. Word limit of λ.
- xi_word_limit, Required. Word limit of ξ.
- verbose, Default False. Print results to screen.

Outputs:

- Output 1. The measure of convergence described above, i.e., $||x_{aug} - x^*||_2$.